Introduction

Texte issu du cours de Marie-Laure Potet et Augustin Lux (2006)

Objectifs :

- Manipulation/Définition de langages (de communication)
  - Format de données (XML), langages de programmation, langage de script.
- Comment bien définir un langage (pouvoir l'analyser...)
- Maîtriser finement les langages de programmation.
  - Expressivité
  - Fiabilité
  - Complexité en exécution
- Quel langage est le mieux adapté, pour quelles particularités...

I. Définition des langages

1. Normes sur le langage

La lexicographie qui décrit les mots du langage :

x := 3 ;

repose sur les mots suivants : idf, affectation, cst ...
Elle est définie à partir d'expressions régulières.

Idf = [a..z] (a..z + 0..9)*

La syntaxe décrit comment les mots peuvente s'enchaîner.

Inst -> place := exp;
place -> idf
...

La sémantique statique (ou syntaxe contextuelle) décrit les vérifications à effectuer pour pouvoir interpréter la syntaxe.

- Identification des noms
- Place et exp ont même type ou bien le type de exp est un sous-type de place.

La sémantique à l'exécution décrit l'exécution.

Place := exp

- Evaluation de l'expression en une valeur v.
- Evaluation de la place en une adresse a.
- Rangement en mémoire de v en a.
- Erreur possible : vérification des contraintes de sous-typage.

Sémantique statique / Sémantique dynamique.

Vérification statique :

- Donne des garanties pour toutes exécutions
- Toute valeur exécution est dans le type déclaré

x : integer;

Vérification dynamique : garantit une propriété pour une exécution donnée.

Exemple : Variables non initialisées.

- Java les détecte à la compilation.
- C/C++ : comportement indéfini à l'exécution.

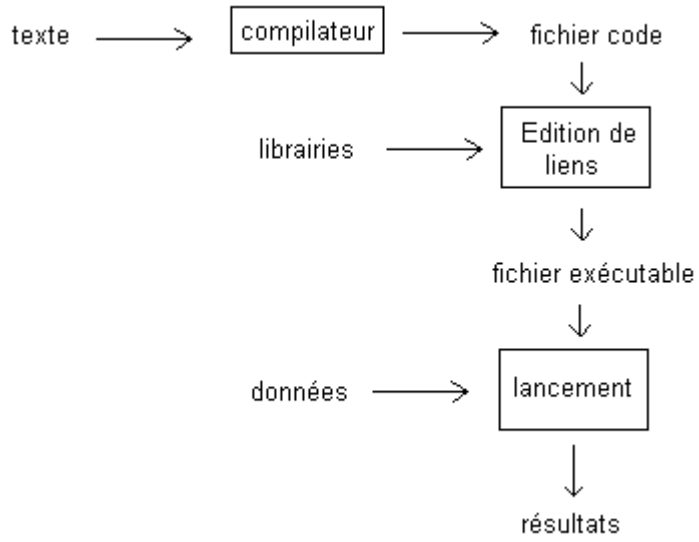
2. Conformité d'un compilateur à une norme.

- Description des limitations du compilateur (imbrication, maximum de blocs, taille des entiers...)
- Détecte toutes les erreurs lexicales, syntaxiques et contextuelles.
- Accepte tous les programmes corrects.
- S'exécute comme décrit par le langage.
- Détecte les erreurs imposées.
- Fait ce qu'il veut sur les exécutions non définies :
  - non initialisation des variables
  - sensibilité à l'ordre d'évaluation.

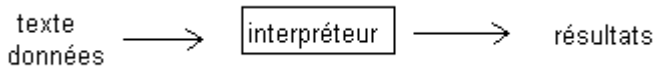
La conformité évaluée par des batteries de tests prédéfinis.
En 2004 :
– C : 3000 tests
– C++ : 4000 tests
– Ada : 4000 tests

II. Compilation et structure d'un compilateur

Compilateur



Interpréteur



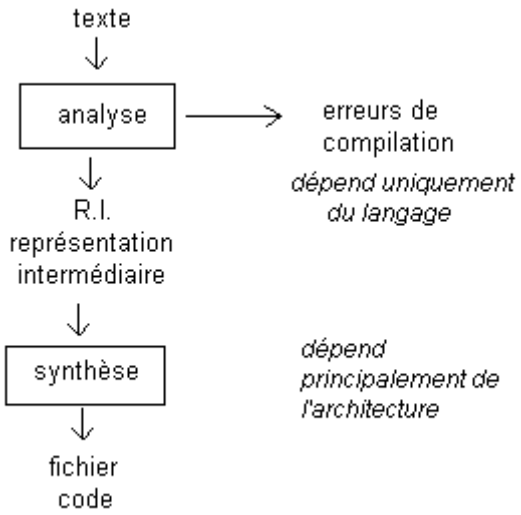
Langages compilés :
- C
- Ada
- C++
- C#

Langages interprétés :
- pearl
- python
- Lisp
- html

Java ? Interprété, compilé ou compilé à la volée.
Permet la mobilité en éliminant le fait de compiler pour chaque architecture.

Structure d'un compilateur

Deux étapes :
– Analyse
– Synthèse



– Pour les développeurs de compilateur. Réutilisation des phases et d'analyse et des phases de synthèse pour une architecture donnée.
  – Exemple : GCC
    GNU Compiler Collection
    gcc pour C
    g++ pour C++
    gnat pour Ada
    fortran, objective C...
  – Objectif : développer « rapidement » un compilateur pour une architecture donnée.

III. Contenu Cours/TD/Projet

Etude de la phase d'analyse découpée en 3 étapes :

– analyse lexicale
– analyse syntaxique
– analyse contextuelle

Etude de la phase de synthèse

– schémas de traduction
– techniques d'optimisation

Détail de la phase d'analyse :

– Lexicographie
  – description : langages réguliers
  – reconnaisseur : automates déterministes
  – programmation : programmes itératifs

Syntaxe :

Description, grammaire hors-contexte.
Reconnaisseur, automates à pile (sous-classes de langage facilement reconnaissables).

Programmation : récursivité

Syntaxe contextuelle :

Description : grammaire attribuée.
Reconnaisseur/Programmation : parcours d'arbre.

Analyse lexicale

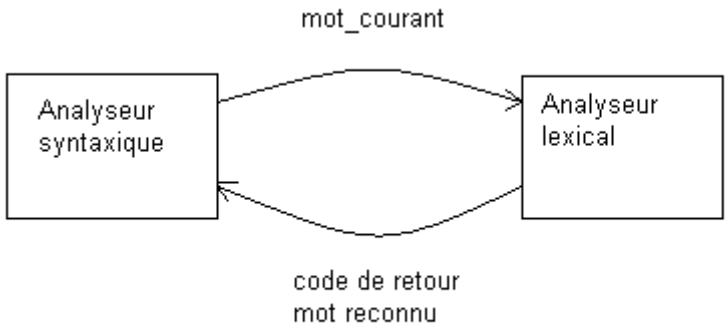
I. Lexicographie

Langages de programmation sont définis par deux niveaux :
- La lexicographie description des mots du langage.
  V_T = { Ensemble des caractères du clavier }
  L : Ensemble des mots admis { idf, cst, if, ... }
  idf : La catégorie des identificateurs
- La syntaxe : description des structures admises dans le langage.
  V_T : Le langage d'écrit par la lexicographie
  L : Les structures admises
  inst -> place := exp
  place -> idf

Exemple : 'x' ':=' 'y'

Lexicographie
idf – signe-aff – idf

Syntaxique
Construction d'affectation



Lexicographie : Langage régulier
Décrite par des listes d'expressions de la forme.
$N_1 = E_1$;
$N_2 = E_2$;
...
$N_k = E_k$;

N_k sont des noms d'expressions régulières.
E_k sont des expressions régulières constituées sur le vocabulaire.

Exemple :
chiffre = '0' + ... + '9'
cst = chiffre $^+$
lettre = 'a' + ... + 'Z'
idf = lettre (lettre + chiffre)*

Rappel : expression régulière sur un vocabulaire.
V s'écrit à l'aide des règles suivantes :
- x Є V alors x est une Expression régulière (ER)
- ε est une ER
- Φ est une ER
- Si E1 et E2 sont des ER alors
  E1* est une ER
  E1 + E2 est une ER
  $E1^+$ est une ER
  E1.E2 est une ER

Contre-exemple :
N1 = a N2 b
N2 = a N1 b + ε
n'est pas une définition régulière car N1 dépend de N2 et réciproquement.

II. Programmation de reconnaisseurs

Automates d'états fini

A = (V, Q, q_0 , F, δ)
q_0 : état initial appartenant à l'ensemble des états Q
F inclus dans Q : état final
δ : relation de transition

δ inclus dans Q x V U { ε } x Q
L(A) = { w Є V* | (q_0 , w) |-- (q, ε) /\ q Є F }

(p , aw) |-- (q , aw) ssi (p,a,q) Є δ
|--* : fermeture réflexive et transitive de |--

Automate déterministes : plus efficace pour la reconnaissance.
δ est une fonction partielle (Q,V) -> Q

Deux types de programmation :
– L'automate est une donnée, un moniteur simule le parcours de l'automate
– L'automate est codé par un flot de contrôle dans le programme (systématique / à la main)

Implémentation 1 :

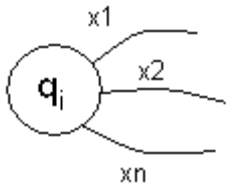
```
Type automate is array ( Q, V ) of Q U {nom_def } ;
        -- non_def : pas de transition (état puit)
        δ : Automate

function lire return V U { $ } ;
-- lit le prochain élément de V
-- retourne $ si tous les caractères ont été lus

procedure reconnaître is
q_cour : Q;
elem : V;
err : exception;
q_cour := q0;
elem := lire;
while elem != '$' loop
        q_cour := δ(q_cour,elem);
        if q_cour = non_def then raise err;
        end if;
        elem := lire;
end loop;
if q_cour not in F then raise err;
end if;
end.
```

Coût : linéaire en fonction de |w|

Cas non déterministe :
– q_cour devient l'ensemble des états atteints
– A chaque pas on calcule les états suivants possible avec ε-fermeture

coût : |w| x |Q|²
|Q|² : complexité du calcul d' ε -fermeture

Implémentation 2 :



q_i devient un point de contrôle dans le programme.
– Les états deviennet des labels dans le code
– Les transitions sont des goto

q_i :
```
elem := lire;
case elem is
        when xi => goto qxi ;
        ...
        when $ => null ;
        when others => raise err;
end case;

procedure reconnaître ;
begin
        goto q0 ;
end;
```

A la main : programmation par des conditions et des boucles (Voir TP)

Application à la lexicographie :

Problème : L'entrée est une suite de caractères à découpes en mots.
'y' '3' 'z' '_' :
– idf de valeur 'y3z'
– idf de valeur 'y' suivi de la constante de valeur 3 suivi de l'idf de valeur z

Solution : principe de la plus longue correspondance : on reconnaît à chaque fois le mot le plus long possible; peut nécessiter l'introduction de caractère séparateur.

Exemple :
'3' '.' '_' : Erreur
'3' '_' '.' '_' : Constante numérique suivi du caractère '.'

Dans les langages de programmation on peut borner le nombre de caractères à regarder en avance pour trouver le bon découpage (1 ou 2 max).

Adaptation de l'algorithme :
– On itère jusqu'à arriver dans l'état q_cour = non_def
– Dès qu'on rencontre un état final, on le mémorise.
– Fin de d'itération : si on n'a pas rencontré d'état final raise erreur sinon OK.

III. Générateur d'analyseurs

Description par Expression Régulière
|
V
Automate associé à l'ER
|
V
Automate déterministe
|
V
Programme reconnaisseur

ANALYSE DESCENDANTE DETERMINISTE METHODE LL(1)

1 Rappels

Les rappels ci-dessous sont succints, pour tout complément se reporter au polycopié *Théorie des Langages*.

1.1 Grammaire hors-contexte

Une grammaire hors-contexte est un quadruplet $G = (V_T, V_N, S, R)$ avec :

- V_T et V_N deux vocabulaires disjoints;
- S un élément distingué de V_N appelé axiome;
- R un ensemble de règles de la forme $A \to \omega$ avec A élément de V_N et ω chaîne sur $V_T \cup V_N$.

Par convention on notera en minuscule les éléments de V_T et en majuscule les éléments de V_N. Lorsque ceci n'est pas précisé le non-terminal apparaissant à gauche de la première règle est l'axiome. Ceci permet de définir une grammaire en ne donnant que ses règles de production.

$\Rightarrow$ dénote la relation de dérivation, $\overset{*}{\Rightarrow}$ dénote la fermeture réflexive et transitive de cette relation.

Remarques :

1. La notation de chaîne vide, ε, n'est pas un élément du vocabulaire terminal.

2. On ne considère dans la suite que des grammaires *réduites*, c'est-à-dire dont tous les symboles sont productifs et accessibles (voir polycopié *Théorie des Langages).*

1.2 Reconnaisseurs pour les langages hors-contexte

Le langage engendré par une grammaire $G = (V_T, V_N, S, R)$, noté $L(G)$, est défini par :

$$L(G) = \{\omega \in V_T^* : S \stackrel{*}{\Rightarrow} \omega\}$$

On montre qu'un mot ω appartient à $L(G)$ en construisant une dérivation de l'axiome vers ω. Une dérivation est souvent représentée sous forme arborescente (arbre de dérivation), cette forme permet de ne pas distinguer des dérivations ne différant que par l'ordre d'applications des règles.

Un reconnaisseur est un automate qui permet, à partir d'une grammaire G et d'une chaine ω de répondre en temps fini à la question "$\omega \in L(G)$?". Répondre *oui* à la question "$\omega \in L(G)$?" revient donc à construire une dérivation $S \stackrel{*}{\Rightarrow} w$ et répondre *non* à cette même question revient à montrer qu'aucune dérivation ne conduit à ω. Les résultats sont les suivants :

- Le problème de la reconnaissance est décidable pour les langages hors-contexte ;
- Les algorithmes généraux de reconnaissance pour les langages hors-contexte sont non-déterministes (ils peuvent nécessiter des retour-arrières sur la chaîne d'entrée) ;
- Il existe des sous-classes de langages hors-contextes pour lesquelles il est possible d'appliquer des méthodes d'analyse déterministe. Par exemple la sous-classe des langages $LL(1)$ étudiée ici.

Ceci diffère des langages réguliers pour lesquels il existe toujours un reconnaisseur déterministe, par exemple sous la forme d'un automate fini déterministe.

2 Grammaires LL(1)

Les grammaires LL(1) permettent d'effectuer une analyse descendante **déterministe** (ou analyse prédictive) en ne considérant que le symbole courant de la phrase à analyser.

2.1 Analyse Descendante

Le principe général de l'analyse descendante est, partant de l'axiome, de réécrire ce dernier en la phrase à analyser. On utilise pour ce faire les règles de la grammaire pour réécrire le **non-terminal le plus à gauche**.

On décrit, avec les deux règles ci-dessous, la manière d'engendrer systématiquement tous les modèles de phrase.

1. L'axiome S est un modèle de phrase.
2. Soit $\omega_1 A \omega_2$ avec $\omega_1 \in V_T^*$, $A \in V_N$ et $\omega_2 \in (V_T \cup V_N)^*$, un modèle de phrase et soit $A \rightarrow \omega$ une règle de la grammaire, alors $\omega_1 \omega \omega_2$ est un modèle de phrase.

On peut imaginer un algorithme non déterministe qui utilise la règle 2 tant que la chaîne à analyser n'a pas été engendrée ou tant qu'on n'a pas construit suffisamment de modèles de phrase pour décider qu'aucun ne conduira à la phrase à analyser. Cette décision dépend de la forme des règles de la grammaire et de la taille de la chaîne à analyser. Il existe toujours une valeur permettant de borner la longueur des dérivations (voir le cours de Théorie des Langages).

Le non déterminisme de l'algorithme, esquissé ci-dessus, résulte uniquement du choix de la règle à utiliser pour réécrire le non terminal le plus à gauche. Les grammaires LL(1) permettent d'effectuer ce choix de façon déterministe en consultant le symbole courant de la phrase à analyser. Dans le sigle LL(1) le premier L (pour Left) signifie qu'on lit la chaîne à analyser de gauche à droite, le second L qu'on construit une dérivation la plus à gauche et le 1 qu'on consulte un seul symbole sur la chaîne à analyser. Cette méthode peut bien sûr s'étendre en autorisant la consultation de k symboles en avance. Avant d'étudier comment déterminer si une grammaire est LL(1) ou non, regardons quelques exemples.

Exemple 1 Soit G_1 la grammaire suivante :

$$\begin{array}{lccl} (1) & S & \rightarrow & a \\ (2) & S & \rightarrow & Ac \\ (3) & A & \rightarrow & bAa \\ (4) & A & \rightarrow & c \end{array}$$

avec $V_T = \{a, b, c\}$.

Dans cet exemple le choix de la règle à appliquer ne pose pas de problème. En effet si on doit réécrire le non-terminal S alors soit le caractère courant est a et on

applique la première règle, soit le caractère courant est b ou bien c et on applique la seconde règle, puisque la réécriture de A produira b ou c (règles 3 et 4). De la même manière, lorsqu'il s'agit de réécrire le non-terminal A, la consultation du caractère courant nous permet de choisir entre les règles (3) et (4). On peut donc construire un tableau qui décrit quelle règle choisir en fonction du non-terminal à réécrire et du caractère courant. On désignera par *erreur* l'impossibilité de choisir une règle. Pour la grammaire G_1 ce tableau se présente sous la forme :

	a	b	c
S	(1)	(2)	(2)
A	*erreur*	(3)	(4)

Pour chaque non-terminal et chaque terminal courant une règle au plus peut être choisie, la grammaire G_1 est donc LL(1).

Exemple 2 Ajoutons à la grammaire G_1 la règle $S \to bS$. La grammaire G_2 obtenue est :

$$\begin{array}{lccl} (1) & S & \to & a \\ (2) & S & \to & Ac \\ (3) & S & \to & bS \\ (4) & A & \to & bAa \\ (5) & A & \to & c \end{array}$$

Cette grammaire n'est pas LL(1) puisque, par exemple lorsqu'un mot débute par b, on peut construire deux dérivations qui produisent toutes deux le terminal b en tête, en utilisant deux façons différentes de réécrire S ($S \Rightarrow bS$ et $S \Rightarrow Aa \Rightarrow bAaa$). Lorsque le non-terminal à réécrire est S et l'élément courant est b, on ne sait donc pas choisir entre les règles 2 et 3.

Exemple 3 Sur des grammaires ne contenant pas de ε-règle, il est assez facile d'intuiter si une grammaire est LL(1) ou non. Examinons maintenant des exemples contenant des ε-règles. Remplaçons la dernière règle de la grammaire G_1 par $A \to \varepsilon$. La grammaire G_3 obtenue est :

$$\begin{array}{lccl} (1) & S & \to & a \\ (2) & S & \to & Ac \\ (3) & A & \to & bAa \\ (4) & A & \to & \varepsilon \end{array}$$

Nous devons maintenant pouvoir choisir entre les 2 règles $A \to bAa$ et $A \to \varepsilon$. L'utilisation de la dernière règle a pour effet de transformer un modèle de phrase

de la forme $\omega_1 A \omega_2$ en $\omega_1 \omega_2$ et donc l'élément courant de la chaîne à reconnaître est un élément du vocabulaire terminal qui peut être placé derière le non-terminal A.

Par exemple si on veut reconnaître le mot c on va construire la dérivation $S \Rightarrow Ac \Rightarrow c$, l'élément courant au moment du choix entre les deux règles $A \rightarrow bAa$ et $A \rightarrow \varepsilon$ est alors c. Dans la grammaire G_3 les éléments qui peuvent suivre le non-terminal A dans une dérivation à partir de l'axiome sont a et c, à cause des règles $A \rightarrow bAa$ et $S \rightarrow Ac$. La règle $A \rightarrow bAa$, quant à elle, n'est choisie que lorsque le caractère courant est b, il n'y a donc pas de problème de choix entre les règles 3 et 4.

De la même manière il faut pouvoir choisir entre les deux règles permettant de réécrire le non-terminal S. La règle $S \rightarrow Ac$ peut s'appliquer lorsque le caractère courant est soit b soit c. En effet on peut construire les deux dérivations $S \Rightarrow Ac \Rightarrow bAac$ et $S \Rightarrow Ac \Rightarrow c$. On peut donc choisir les règles à appliquer de la manière suivante :

	a	b	c
S	(1)	(2)	(2)
A	(4)	(3)	(4)

Exemple 4 Si on considère maintenant les règles $S \rightarrow Ac$, $A \rightarrow \varepsilon$ et $A \rightarrow c$ la grammaire obtenue n'est pas LL(1). En effet on peut alors construire les deux dérivations $S \Rightarrow Ac \Rightarrow cc$ et $S \Rightarrow Ac \Rightarrow c$ qui produisent toutes deux une chaîne débutant par c, en utilisant des règles différentes ayant même non-terminal en partie gauche.

2.2 Conditions LL(1)

Nous donnons maintenant une méthode systématique pour décider si une grammaire est LL(1) ou non. Pour cela on calcule pour chaque règle l'ensemble des "symboles courants" qui peuvent être produits par cette règle, dans une dérivation quelconque à partir de l'axiome. Cet ensemble est appelé l'ensemble des symboles **directeurs** associés à une règle. La fonction *Directeur* associe à chaque règle son ensemble de symboles directeurs.

Définition 1

$$Directeur : R \longrightarrow \mathcal{P}(V_T \cup \{\$\})$$

$$avec : Directeur(A \to \omega) =$$

$$\{x \in V_T \cup \{\underline{\$}\} : \exists\ \omega_1,\ \omega_2,\ \omega_3\ S\underline{\$} \stackrel{*}{\Rightarrow} \omega_1 A \omega_2 \Rightarrow \omega_1 \omega \omega_2 \stackrel{*}{\Rightarrow} \omega_1 x \omega_3\}$$

Remarques :

1. Le modèle de phrase initial est $S\underline{\$}$ (où $\$$ est un nouveau symbole n'appartenant pas à V_T), ceci afin que la fonction *Directeur* soit toujours définie. On a alors $Directeur(A \to \omega) \neq \emptyset$, $\forall A \to \omega$.

2. La notation de chaîne vide ε n'est évidemment jamais un symbole directeur de règle (ε n'est pas élément de V_T).

3. Lorsque ω ne dérive pas vers ε (notation $\omega \stackrel{*}{\not\Rightarrow} \varepsilon$) on peut simplifier la définition en $Directeur(A \to \omega) = \{x \in V_T : \exists \omega_1\ \omega \stackrel{*}{\Rightarrow} x\omega_1\}$. En effet, dans ce cas, la réécriture de la partie droite de règle produit toujours un élément de V_T.

On peut alors donner la définition suivante :

Définition 2 *Une grammaire $G = (V_T, V_N, S, R)$ est LL(1) si et seulement si elle vérifie la condition :*

$$\forall\ A \to \omega_1\ \forall\ A \to \omega_2$$

$$\omega_1 \neq \omega_2 \Rightarrow Directeur(A \to \omega_1) \cap Directeur(A \to \omega_2) = \emptyset$$

On énonce ici le fait que, lorsqu'on doit réécrire un non-terminal A et que celui-ci peut être réécrit de différentes manières, on peut toujours choisir la règle à utiliser en fonction de l'élément courant de la phrase à reconnaître. Remarquons que les comparaisons entre ensembles de directeurs ne portent que sur des règles ayant même non-terminal en partie gauche.

Exemple :

Dans l'exemple 2 de la section précédente on peut construire les deux dérivations suivantes :

- $S\underline{\$} \Rightarrow Ac\underline{\$} \Rightarrow bAac\underline{\$}$ (par les règles $S \to Ac$ et $A \to bAa$)
- $S\underline{\$} \Rightarrow bS\underline{\$}$ (par la règle $S \to bS$)

On a donc $b \in Directeur(S \to Ac)$ et $b \in Directeur(S \to bS)$. L'intersection de ces deux ensembles est non vide, la grammaire G_2 n'est donc pas LL(1).

De la même manière dans l'exemple 4 on peut construire les deux dérivations :

- $S\underline{\$} \Rightarrow Ac\underline{\$} \Rightarrow c\underline{\$}$ (par la règle $A \to \varepsilon$)
- $S\underline{\$} \Rightarrow Ac\underline{\$} \Rightarrow cc\underline{\$}$ (par la règle $A \to c$).

On a $c \in Directeur(A \to c) \cap Directeur(A \to \varepsilon)$ et donc la condition LL(1) n'est pas vérifiée.

2.3 Calcul des conditions LL(1)

Nous allons maintenant étudier comment calculer les ensembles de directeurs. La définition donnée dans la section précédente n'est pas constructive dans le sens où elle fait intervenir l'ensemble des dérivations possibles de la forme $S\underline{\$} \stackrel{*}{\Rightarrow} \omega_1 A \omega_2$, cet ensemble est en général infini. Nous allons donner ci-dessous une méthode constructive permettant de ramener le calcul des directeurs à un calcul fini. Pour cela nous définissons ci-dessous deux fonctions.

Soient $V_T^{'} = V_T \cup \{\underline{\$}\}$ et $V = V_N \cup V_T^{'}$.

- *Premier* : $V^* \to \mathcal{P}(V_T^{'})$
  avec : Premier(ω) = $\{\mathbf{x} \in V_T^{'} : \exists\ \omega_1 \in V_T^{'*},\ \omega \stackrel{*}{\Rightarrow} \mathbf{x}\omega_1\}$
- *Suivant* : $V_N \to \mathcal{P}(V_T^{'})$
  avec : Suivant(A) = $\{\mathbf{x} \in V_T^{'} : \exists\ \omega_1 \in V_T^{*}, \exists\ \omega_2 \in V_T^{'*},\ S\underline{\$} \stackrel{*}{\Rightarrow} \omega_1 A \mathbf{x} \omega_2\}$

Remarques :

1. Les définitions de *Premier* et *Suivant* ne sont pas constructives puisqu'elles utilisent aussi la notion de dérivations de longueur quelconque.
2. On peut avoir $Premier(\omega) = \emptyset$ si et seulement si $\mathcal{L}(\omega) = \{\varepsilon\}$.
3. On a toujours $Suivant(A) \neq \emptyset$, puisque $\mathbf{x}\omega_2$ contient au moins $\underline{\$}$.

On peut redéfinir la fonction *Directeur* à l'aide de ces nouvelles fonctions.

Définition 3

$$Directeur(A \to \omega)$$

$$=$$

$$Premier(\omega) \ \cup \ (\texttt{si}\ \omega \stackrel{*}{\Rightarrow} \varepsilon\ \texttt{alors}\ Suivant(A)\ \texttt{sinon}\ \emptyset)$$

Rappelons que $\omega \stackrel{*}{\Rightarrow} \varepsilon$ signifie $\varepsilon \in \mathcal{L}(\omega)$ et non pas $\mathcal{L}(\omega) = \{\varepsilon\}$ et que $\omega \stackrel{*}{\not\Rightarrow} \varepsilon$ signifie que ε n'est pas élément de $\mathcal{L}(\omega)$.

La définition ci-dessus est équivalente à la définition 1 si la grammaire ne contient pas de symboles inaccessibles. En effet si on considère la règle $A \to a$, A étant inaccessible, par la première définition nous aurons $Directeur(A \to a) = \emptyset$ et par la définition ci-dessus $Directeur(A \to a) = \{a\}$. La restriction sur les grammaires provient du fait que nous transformons une définition basée sur les dérivations en une définition basée sur les règles. Pour cela on doit garantir qu'il y a correspondance entre les dérivations et les règles et donc qu'il n'y a pas de "parasites" dans la grammaire.

Nous allons maintenant donner une méthode constructive permettant de calculer les fonctions *Premier* et *Suivant*, à partir des règles de la grammaire. Les mêmes restrictions que précédemment sont faites. La méthode considérée consiste à définir les fonctions *Premier* et *Suivant* comme les plus petites solutions d'équations. Ces fonctions peuvent être définies par cas sur les chaînes et les définitions obtenues peuvent être récursives, en raison de la récursivité qui peut apparaître dans la grammaire.

Définition 4

$$\begin{array}{lcll} Premier(\varepsilon) & = & \emptyset & \\ Premier(x) & = & \{x\} & \text{si } x \in V_T' \\ Premier(A) & = & \bigcup_{A \to \omega_i} Premier(\omega_i) & \text{si } A \in V_N \\ Premier(X.\omega) & = & Premier(X) \cup & \\ & & (\texttt{si}\ X \stackrel{*}{\Rightarrow} \varepsilon\ \texttt{alors}\ Premier(\omega)\ \texttt{sinon}\ \emptyset) & \text{si } X \in V \text{ et } \omega \in V^* \end{array}$$

Définition 5 Rappelons que *Suivant* n'est définie que sur V_N.

Pour tout non-terminal A :

$$Suivant(A) \ = \ \bigcup_{B \to \omega_1 A \omega_2} (Premier(\omega_2) \ \cup \ (\texttt{si}\ \omega_2 \stackrel{*}{\Rightarrow} \varepsilon\ \texttt{alors}\ Suivant(B)\ \texttt{sinon}\ \emptyset))$$

avec $A, B \in V_N$ et $\omega_1, \omega_2 \in V^*$

Pour l'axiome : on ajoute le symbole \$ à l'ensemble des *Suivants* défini par la formule ci-dessus.

Pour calculer *Premier* et *Suivant* on écrit les équations à l'aide des définitions ci-dessus, puis on les transforme par substitution jusqu'à obtenir un système ne contenant en partie droite que des fonctions portant sur des terminaux et des non-terminaux de la grammaire. On peut ici, étant donnée la forme linéaire des équations, résoudre simplement les équations récursives. Une équation de la forme $X = X \cup B$ a pour plus petite solution $X = B$.

Exemple 5 Soit la grammaire dont les règles de production sont $A \to AB$, $A \to \varepsilon$ et $B \to a$. On veut calculer $Premier(A)$. En appliquant la définition on obtient $Premier(A) = Premier(AB) \cup Premier(\varepsilon)$, c'est à dire :

$$\begin{aligned} Premier(A) &= Premier(A) \cup Premier(B) \\ Premier(B) &= Premier(a) \end{aligned}$$

La première équation provient du fait que $A \stackrel{*}{\Rightarrow} \varepsilon$. On obtient alors les équations récursives suivantes :

$$\begin{aligned} Premier(A) &= Premier(A) \cup \{a\} \\ Premier(B) &= \{a\} \end{aligned}$$

On résout la première équation en $Premier(A) = \{a\}$, qui est la plus petite solution de cette équation. Par exemple {a, b} est aussi solution de cette équation, mais ce n'est pas la plus petite. On voit sur cet exemple que retenir la plus petite solution garantit qu'on n'obtient que des éléments atteignables par dérivation.

Appliquons maintenant ces calculs aux exemples 3 et 4 de la section **2.1**.

Exemple 6 Soit la grammaire :

$$\begin{aligned} S &\to a \\ S &\to Ac \\ A &\to bAa \\ A &\to \varepsilon \end{aligned}$$

On a alors :

$$\begin{aligned} Directeur(S \to a) &= Premier(a) = \{a\} \\ Directeur(S \to Ac) &= Premier(A) \cup Premier(c) \\ &= Premier(A) \cup \{c\} \\ Directeur(A \to bAa) &= Premier(b) = \{b\} \\ Directeur(A \to \varepsilon) &= Premier(\varepsilon) \cup Suivant(A) = Suivant(A) \end{aligned}$$

On doit donc calculer $Premier(A)$ et $Suivant(A)$. En appliquant les définitions on obtient $Premier(A) = Premier(bAc) \cup Premier(\varepsilon) = \{b\}$. Pour calculer $Suivant(A)$ on considère les deux règles ayant A en partie droite, c'est à dire $S \rightarrow Ac$ et $A \rightarrow bAa$. On a donc $Suivant(A) = Premier(c) \cup Premier(a) = \{a, c\}$.

$$\begin{array}{lcl} Directeur(S \rightarrow a) & = & \{a\} \\ Directeur(S \rightarrow Ac) & = & \{b, c\} \\ Directeur(A \rightarrow bAa) & = & \{b\} \\ Directeur(A \rightarrow \varepsilon) & = & \{a, c\} \end{array}$$

Les deux premiers ensembles sont disjoints, les deux derniers aussi donc la grammaire est LL(1).

Exemple 7 Rajoutons maintenant la règle $A \rightarrow c$ (exemple 4 précédent). On a alors $Directeur(A \rightarrow \varepsilon) = \{a, c\}$ et $Directeur(A \rightarrow c) = \{c\}$. La grammaire n'est donc pas LL(1).

Exemple 8 Finissons sur un exemple plus conséquent : une grammaire LL(1) des expressions arithmétiques définies sur le vocabulaire $V_T = \{num, add, mult, (,)\}$. Soit la grammaire :

$$\begin{array}{rcl} E & \rightarrow & T\ SE \\ SE & \rightarrow & add\ T\ SE \\ SE & \rightarrow & \varepsilon \\ T & \rightarrow & F\ ST \\ ST & \rightarrow & mult\ F\ ST \\ ST & \rightarrow & \varepsilon \\ F & \rightarrow & (E) \\ F & \rightarrow & num \end{array}$$

On a alors :

$$\begin{array}{lcl} Directeur(E \rightarrow T\ SE) & = & Premier(T) \\ Directeur(SE \rightarrow add\ T\ SE) & = & \{add\} \\ Directeur(SE \rightarrow \varepsilon) & = & Suivant(SE) \\ Directeur(T \rightarrow F\ ST) & = & Premier(F) \\ Directeur(ST \rightarrow mult\ F\ ST) & = & \{mult\} \\ Directeur(ST \rightarrow \varepsilon) & = & Suivant(ST) \\ Directeur(F \rightarrow (E)) & = & \{(\} \\ Directeur(F \rightarrow num) & = & \{num\} \end{array}$$

Nous devons donc calculer $Premier(T)$, $Premier(F)$, $Suivant(SE)$ et $Suivant(ST)$. Commençons par les Suivants :

$$\begin{aligned} Suivant(SE) &= Suivant(SE) \cup Suivant(E) \\ Suivant(E) &= \{), \underline{\$}\} \\ Suivant(ST) &= Suivant(ST) \cup Suivant(T) \\ Suivant(T) &= Premier(SE) \cup Suivant(SE) \cup Suivant(E) \end{aligned}$$

Ce qui donne :

$$\begin{aligned} Suivant(SE) &= \{), \underline{\$}\} \\ Suivant(E) &= \{), \underline{\$}\} \\ Suivant(ST) &= Suivant(T) \\ Suivant(T) &= Premier(SE) \cup \{), \underline{\$}\} \end{aligned}$$

Il reste maintenant à calculer $Premier(T)$, $Premier(SE)$ et $Premier(F)$. On a $Premier(T) = Premier(F)$, $Premier(SE) = \{add\}$ et $Premier(F) = \{(, num\}$. Et donc :

$$\begin{aligned} Directeur(E \rightarrow T\ SE) &= \{(, num\} \\ Directeur(SE \rightarrow add\ T\ SE) &= \{add\} \\ Directeur(SE \rightarrow \varepsilon) &= \{), \underline{\$}\} \\ Directeur(T \rightarrow F\ ST) &= \{(, num\} \\ Directeur(ST \rightarrow mult\ F\ ST) &= \{mult\} \\ Directeur(ST \rightarrow \varepsilon) &= \{add,), \underline{\$}\} \\ Directeur(F \rightarrow (E)) &= \{(\} \\ Directeur(F \rightarrow num) &= \{num\} \end{aligned}$$

Il n'y a pas d'ambiguïté sur le choix des règles pour les non-terminaux SE, ST et F donc la grammaire est LL(1). En effet E et T ne posent pas de problème puisqu'il n'y a qu'une seule règle attachée à chacun de ces non-terminaux.

3 Résultats et Limitations

On étend les propriétés des grammaires aux langages. On dit donc qu'un langage est LL(1) si et seulement si il existe une grammaire LL(1) qui l'engendre. On sait décider si une grammaire est $LL(1)$ mais on ne sait pas décider si un langage est $LL(1)$. Il n'existe pas d'algorithme prenant en entrée une grammaire hors-contexte et calculant si il existe une grammaire LL(1) équivalente, c'est-à-dire engendrant le même langage. On peut néanmoins donner quelques critères :

1. Une grammaire ambiguë n'est pas $LL(1)$
2. Une grammaire récursive à gauche n'est pas $LL(1)$
3. Un langage régulier est toujours LL(1)

Certains langages sont intrinséquement ambigus, il existe donc des langages qui ne sont pas LL(1) (il y en a d'autres ...). Lorsqu'une grammaire est récursive à gauche il est inutile de se lancer dans des calculs. La méthode d'élimination de la récursivité à gauche dans les grammaires peut dans certains cas rendre la grammaire LL(1), mais pas toujours. En particulier si la grammaire est ambiguë elle le restera. La remarque 3 n'est pas très intéressante puisqu'on dispose déjà d'une méthode d'analyse déterministe pour les langages réguliers.

Analyse Syntaxique

I. Transformation de grammaires

Un langage est LL(1) si et seulement s'il existe une grammaire LL(1) qui décrit ce langage.

- décider si une grammaire est LL(1) : Décidable
- décider si un langage est LL(1) : Indécidable

Transformation de grammaires : heuristiques

Factorisation

$A \Rightarrow w a_1$
$A \Rightarrow w a_2$
peut être transformée en
$A \Rightarrow wB$
$B \Rightarrow a_1$
$B \Rightarrow a_1$
avec un nouveau symbole B

Remplacement

On remplace pour factoriser
$A \Rightarrow w_1 B w_2$
$B \Rightarrow a_1 \mid a_2 \mid ... \mid a_n$

$A \Rightarrow w_1 a_1 w_2$
$A \Rightarrow w_1 a_2 w_2$
...
$A \Rightarrow w_1 a_n w_2$

Exemple 1

(1) S => a {a}
(2) S => Ac {b,c}
(3) A => bAa {b}
(4) A => c {c} conflit LL(1)
(5) A => ε {a,c}

Remplacement de A par sa définition :
(2) devient :
S => cc
S => c
S => bAac

On factorise :
S => cX
X => c {c}
X => ε {$}

Exemple 2

$a^n b^m$
S => aSb {a}
S => X {a,b}
X => aX
X => ε

S => aSb
S => ε
S => aX

On peut factoriser
S => aB
B => Sb
B => X

Langage non LL(k) pour tout k

Élimination de la récursivité à gauche

=> On peut montrer qu'une grammaire récursive à gauche n'est jamais LL(k) pour tout k
Il existe un non-terminal A.
A =>+ Aw
=>+ : dérivation de taille supérieure ou égale à un

Montrons que la grammaire est non LL(k) pour tout k
On suppose qu'on a A =>* u avec u Є V_T*
=> A est productif

cas 1 : w ne fait que dériver vers E pour toute dérivation A =>* v alors v est E
Dans ce cas la grammaire est ambigüe donc non LL(k)
A =>+ Aw =>+ A
A =>+ A

cas 2 : w =>+ v avec v Є V_T* et v ≠ ε
=> |v| supérieur ou égal à 1
S$ =>* w_1 A w_2
S$ =>* w_1 A w^k w_2 k fois la dérivation récursive à gauche

Deux dérivations possibles:
(a) w_1 A w^k w_2 A =>* u
ou
w_1 A w^k w_2 =>* w_1 u w^{k+1} w_2

(b) A =>* Aw A =>* u

Conclusion :
On ne sait pas choisir entre les deux dérivation (a) et (b) par consultation de k caractères en avance

=> élimination de la récursivité à gauche.

Transformation d'une règle récursive à gauche
A => Aw
A => a
avec A => a ~~###~~ Aa'

A => aB
B => wB
B => ε

=> Si récursivité indirecte,
A => Ba B => AB
Alors on remplace et on élimine (voir TL 1A)

On s'intéresse aux expressions postfixées 23 + 5 *
E => num
E => E E oper
oper => + | - | *
=> Récursive à gauche

E => num
E => E E op
a = num
w = E op

E => num SE
SE => E op SE
SE => ε

La grammaire obtenue est LL(1).

E => num SE	{num}
SE => E op SE	{num}
SE => ε	{$,+,-,*}
op => + | - | *	

II. Écriture des analyseurs LL(1)

Soit G une grammaire LL(1). Description d'un méta-algorithme qui produit un analyseur LL(1), à partir d'une grammaire G, LL(1).

Si G est LL(1) et si on veut montrer w Є L(G)
On part de la configuration (S$, w)
On veut atteindre (ε, ε)

2 opérations
- effacement : (xw,xa) -> (w,a)
- expansion : (Aw,xa) ->
  - Soit la règle A => B utilisée si x Є Directeur(A => B)
  - LL(1) garantit au plus 1 choix

Principe de l'algorithme
=> On écrit une procédure par non-terminal
L'appel A, a pour effet de reconnaître un mot de L(A)
Invariant : on est toujours positionné sur le mot suivant (LL(1))
Soit w une chaîne sur V_T U V_N : La procédure produire génère le code reconnaissant (w)

```
produire ( ε ) =^ null;
produire(xw) =^ if mot ≠ x then raise erreur;
                        else
                                mot := lire_mot;
                                produire(w);
                end
```

x Є V_T

procedure (Aw) =^ A; produire(w)
L'appel de la procédure A
Ex : produire ('(' E ')') =

```
begin
        if mot != '(' then raise erreur;
        mot := lire_mot;
        E;
        if mot != ')' then raise erreur;
        mot := lire_mot;
end
```

Ecriture des procédures
- Soit A tel que

A => w_1

...

A => w_n

Les règles de G pour A

```
procedure A is
begin
        case elem is
                when directeur (A =>  w1 ) => produire(w1)
                ...
                when directeur (A =>  wn ) => produire(wn)
                when others => raise erreur;
        end case
end A

Procedure reconnaître is
        mot : Token;
begin
        mot := lire_mot;
        S; -- appel de la procédure associée à l'axiome
        if mot != '$' then raise erreur;
end;
```

Principe de l'algorithme :
On écrit une procédure par non terminal. L'appel A a pour effet de reconnaître un mot de L(A)

Application aux expressions postfixées

```
        E => num SE
        SE => E op SE           {num}
        SE =>  ε                {+,-,*,$}
        op => + | - | *
```

V_T = { + , - , * , num , $ }

```
procedure E is
begin
        if mot != num then raise erreur;
        else
                mot := lire_mot;
                SE;
        end;
end E;

procedure SE is
begin
        case mot is
                when num => E; op; SE;
                when + | - | * | $ => null;
                when others => raise erreur;
        end case;
end SE;

procedure op is
        case mot is
                when + | - | * => mot := lire_mot;
                when others => raise erreur;
        end case;
end op;

Programme principal
begin
        mot := lire_mot;
        E;
        if mot != '$' then raise erreur;
end
```

Argument de terminaison : pas de dérivation récursive à gauche.
On ne peut enchaîner que n appels de la forme A avec n inférieur ou égal $|V_N|$

Conclusion sur l'analyse descendante
- Simple et facile à mettre en oeuvre
- Bien adapté au rattrapage d'erreurs
  - Repositionnement sur les caractères attendus.
- Bien adapté à l'évaluation
- Assez restrictif sur la manière de décrire les langages
  - Ex: les expressions postfixées.
- On aurait préféré utiliser la définition E => E E op

III. Evaluation : Grammaires attribuées

Grammaire : définition inductive d'un ensemble de mots

Grammaire attribuée : ajout de calcul sur la syntaxe

Calculs :
- Evaluation
- Verification de condition

E => E + T
E => E or T

T => T * F
T => T and F

F => num
F => true
F => false
F => idf

On veut filtrer les expressions bien typées :
true + 4 or false
Admis par la syntaxe mais rejeté par une grammaire attribuée

Le but, plutôt que de définir directement la syntaxe, est de garder une grammaire hors-contexte.

Exemple : le langage $a^n b^n c^n$
Langage non hors-contexte

S => X Y
X => a X b
X => ε
Y => c Y
Y => ε

On décrit $a^n b^n c^m$
On veut vérifier n = m
On associe :
- Au terminal X un entier n qui compte le nombre de a reconnus
- Au non terminal Y un entier m qui compte le nombre de c reconnus
- Au non terminal S une condition n = m

(1) S => X Y n1 = m2
(2) X => a X b n0 = n2 +1
(3) X => ε n0 = 0
(4) Y => c Y m0 = m2 + 1
(5) Y => ε n0 = 0

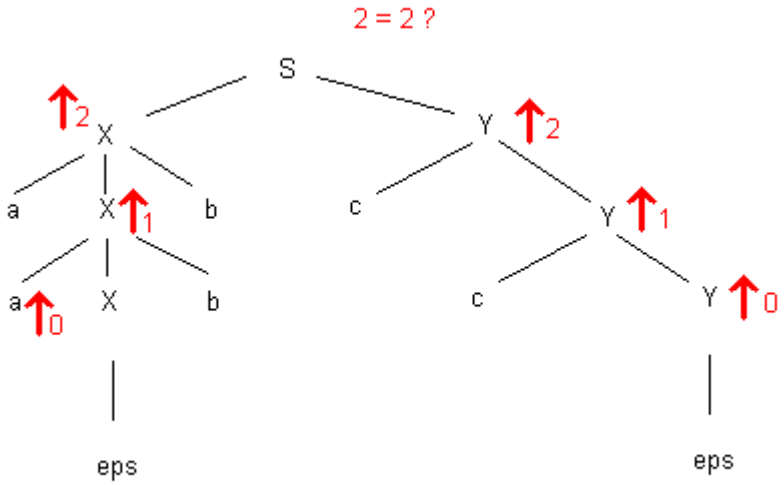
On associe des équations à chaque règle.
On indice les valeurs par leur place dans la règle.

X	=> a1	.....	an
indice 0	indice 1		indice n

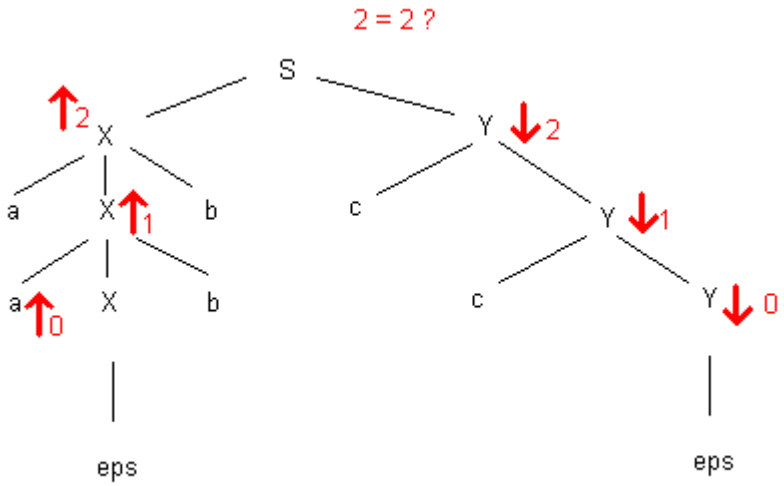
Soit w un mot reconnu par la syntaxe hors contextuelle. On obtient un système d'équation S :
– si S n'a pas de solution, le mot est rejeté
– sinon le mot est accepté

<u>Évaluation des équations</u> : différentes stratégies

aabbcc : Deux solutions



ou



On définit deux types d'attributs :
– Les attributs synthétisés, notés avec une flèche vers le haut, qui décrivent des calculs qui remontent dans des feuilles vers la racine
– Les attributs hérités, notés avec une flèche vers le bas, qui décrivent des calculs des noeuds vers les feuilles

Une grammaire hors-contexte (V_T, V_N , S , R)

Pour chaque élément de V_N , on associe :

- Un ensemble fixé d'attributs
- Pour chaque attribut, on donne
  - Leur domaine de valeur
  - S'ils sont synthétisés ou hérités

Pour chaque règle de R on décrit les calculs d'attributs :

- On peut mettre des conditions
  n = m
- On peut mettre des affectations
  n := 0

Soit la règle suivante : (On notera les flèches haut et bas respectivement ^ et v)

X $^s_0$ vk_0 => Y ^s_i $v\ k_i$

Affectation :

$s_0 := (h_0, s_1, ..., s_n)$

i>=1 $h_i := g(h_0, s_1, h_1, ..., s_n, h_n)$

Exemple 1 : le langage $a^n b^n c^n$ en synthétisé

S => X ^n Y ^m condition n = m
X ^n => a X ^{n_1} b affectation $n := n_1 + 1$
X ^n => ε affectation n := 0
Y ^m => c Y ^{m_1} affectation $m := m_1 + 1$
Y ^m => ε affectation m := 0

Exemple 1 : le langage $a^n b^n c^n$ avec attributs hérités et synthétisé

X : un attribut synthétisé ^n
Y : un attribut hérité vm : nombre de c attendus

S => X ^n Y vm affectation m := m
Les règles sur X ne changent pas
Y vm => c Y vm_1 condition $m \neq 0$ affectation $m_1 := m - 1$
Y vm => ε condition m = 0

Remarques :

Solution 1 : évaluation gauche droite en profondeur d'abord
Solution 2 : même chose avec propagation d'information
Solution 3 : On calcule le nombre de c et on vérifie le nombre de a (parcours droite/gauche). On peut décrire des calculs qui demandent plusieurs parcours de l'arbre

Exemple :

class A { x : B }
class B { y : A }

=> il faut deux passes sur l'arbre

Grammaires attribuée et analyse LL(1) :

LL1 = profondeur d'abord et gauche/droite
Deux classes de grammaires

1) Grammaires S-attribuées :
   => ne contiennent que des attributs synthétisés
2) Grammaires L-attribuées
   X $^s_0$ vk_0 => ... Y ^s_i vk_i ...

Pour ki, l'affectation est de la forme $h_i = g(h_0, h_1, ..., h_{i-1})$
Comment programmer une grammaire LL(1) L-attribuée ?

Attribut synthétisé = paramètre OUT
Attribut hérité = paramètre IN

Programmation de la solution 2 de $a^n b^n c^n$

```
S => X Y
X => a X b      { a }
X => ε          { b, c, $ }
Y => c Y        { c }
Y => ε          { $ }

procedure X (n : OUT integer) ;
procedure Y (n : IN integer) ;

procedure S is
        aux : integer;
begin
        X(aux); -- m := n
        Y(aux);
end S;

procedure X (n : OUT integer) is
begin
        case m in
                where 'a' =>
                        mot := lire_mot;
                        X(n);
                        if mot != b then raise erreur;
                        mot := lire_mot;
                        n := n + 1;
                when 'b' | 'c' | '$' => n := 0;
                when others => raise erreur;
        end case;
end X;

procedure Y (m : IN integer) is
begin
        case mot in
                when 'c' =>
                        mot := lire_mot;
                        if m = 0 then raise erreur;
                        Y (m-1)
                when '$' =>
                        if m != 0 then raise erreur;
                        when others => raise erreur;
        end case;
end Y;
```

GRAMMAIRES ATTRIBUEES : définition et exemples

Les grammaires hors-contexte permettent de définir la syntaxe des mots d'un langage. La syntaxe étant un support pour définir des calculs, les grammaires attribuées permettent, en associant des attributs aux terminaux et non-terminaux d'une grammaire, de décrire de tels calculs. Pour cela il faut préciser :

- les attributs nécessaires ;
- les domaines d'interprétation des attributs ;
- les règles d'évaluation des attributs.

1 Exemple introductif

Considérons un langage très simple de commandes, appelé **Lexpress**, défini syntaxiquement par la grammaire :

$$\mathbf{P} \rightarrow \mathbf{LC} \quad (1)$$

$$\mathbf{LC} \rightarrow \mathbf{C}\ \ \mathbf{LC} \quad (2)$$

$$\mathbf{LC} \rightarrow \mathbf{C} \quad (3)$$

$$\mathbf{C} \rightarrow \underline{\mathbf{id}}\ \underline{:=}\ \mathbf{E}\ \underline{;} \quad (4)$$

$$\mathbf{E} \rightarrow \underline{(}\ \mathbf{E}\ \underline{+}\ \mathbf{E}\ \underline{)} \quad (5)$$

$$\mathbf{E} \rightarrow \underline{\mathbf{id}} \quad (6)$$

$$\mathbf{E} \rightarrow \underline{\mathbf{num}} \quad (7)$$

avec $\mathbf{V_T}$={ $\underline{\mathbf{id}}$, $\underline{:=}$, $\underline{+}$, $\underline{\mathbf{num}}$, $\underline{(}$, $\underline{)}$, $\underline{;}$ } et $\mathbf{V_N}$= { **P**, **LC**, **C**, **E**}. On s'intéresse à la valeur associée aux identificateurs après exécution des commandes. On utilisera la convention suivante : par défaut, la valeur associée à un identificateur est l'entier naturel 0.

1.1 Domaines d'attributs

Soient **A** et **B** des ensembles. On désigne par :

- $\mathbf{A} \rightarrow \mathbf{B}$ une fonction totale de **A** dans **B** ;
- $\mathbf{A} \nrightarrow \mathbf{B}$ une fonction partielle de **A** dans **B** ;

On utilisera les opérations suivantes :

- `dom` le domaine d'une fonction. Si **f** est une fonction totale de **A** dans **B** alors $\mathtt{dom}(\mathbf{f}) = \mathbf{A}$.
- l'application d'une fonction notée (). $\mathbf{f}(\mathbf{n})$ désigne la valeur de la fonction **f** au point **n** si $\mathbf{n} \in \mathtt{dom}(\mathbf{f})$ et est indéfinie sinon.
- $\mathbf{f} \lhd \mathbf{g}$ la réécriture de la fonction f par la fonction g. On a :

$$\mathbf{f} \lhd \mathbf{g} = \{(\mathbf{x}, \mathbf{y}) \mid (\mathbf{x}, \mathbf{y}) \in \mathbf{f} \wedge \neg(\mathbf{x} \in \mathtt{dom}(\mathbf{g}))\} \cup \mathbf{g}$$

On utilisera les domaines d'attributs suivants :

- **N** pour interpréter les valeurs ;
- **CHAINE** pour désigner les noms des identificateurs ;
- **MEM**, le domaine des fonctions de profil $\mathbf{CHAINE} \rightarrow \mathbf{N}$, pour représenter la mémoire (association courante de valeurs aux identificateurs).

1.2 Association d'attributs aux symboles de la grammaire

Une commande a pour effet de modifier la mémoire. On lui associera deux attributs représentant respectivement la mémoire avant exécution de la commande (**me** pour mémoire en entrée) et la mémoire après exécution de la commande (**ms** pour mémoire en sortie).

- Au symbole terminal **id** est associé un attribut **nom**, qui désigne la chaîne représentant l'identificateur ($\mathbf{nom} \in \mathbf{CHAINE}$).
- Au symbole terminal **num** est associé un attribut **val**, désignant sa valeur ($\mathbf{val} \in \mathbf{N}$).
- Aux symboles non-terminaux **LC** et **C** sont associés deux attributs **me** et **ms**, qui désignent respectivement la mémoire avant et après exécution ($\mathbf{me} \in \mathbf{MEM}$ et $\mathbf{ms} \in \mathbf{MEM}$).
- Au symbole non-terminal **E** est associé un attribut **me**, qui désigne la mémoire courante ($\mathbf{me} \in \mathbf{MEM}$) et un attribut **val**, qui désigne la valeur de l'expression dans la mémoire courante ($\mathbf{val} \in \mathbf{N}$).

1.3 Dépendance fonctionnelle

Il reste maintenant à définir comment les attributs associés aux symboles de la grammaire sont affectés. On distingue deux types d'attributs :

- les attributs *hérités* dont la valeur est définie en fonction du contexte dans lequel une occurrence de symbole est dérivée. Par exemple pour la règle $\mathbf{X_0} \rightarrow \mathbf{X_1} \ldots \mathbf{X_n}$, un attribut hérité du symbole $\mathbf{X_i}$, $1 \leq \mathbf{i} \leq \mathbf{n}$, sera défini en fonction des attributs des symboles $\mathbf{X_0}, \ldots \mathbf{X_n}$.
- les attributs *synthétisés* dont la valeur est définie en fonction des règles appliquées pour dériver un symbole. Par exemple pour la règle $\mathbf{X_0} \rightarrow \mathbf{X_1} \ldots \mathbf{X_n}$, un attribut synthétisé du symbole $\mathbf{X_0}$ sera défini en fonction des attributs des symboles $\mathbf{X_0}, \ldots \mathbf{X_n}$.

Par exemple, l'attribut **me** du non-terminal **E** est un attribut hérité : la valeur de l'expression est définie pour la mémoire courante, qui dépend des affectations précédant l'expression **E**. Le contenu de la mémoire provient du contexte dans lequel l'occurrence du non-terminal **E** est dérivée. Par contre l'attribut **val** est un attribut synthétisé : sa valeur est élaborée à partir des éléments constituant l'expression et de la valeur en entrée de la mémoire.

- L'attribut **nom** du symbole terminal **id** est synthétisé.
- L'attribut **val** du symbole terminal **num** est synthétisé.
- L'attribut **me** des symboles non-terminaux **LC** et **C** est hérité. L'attribut **ms** est synthétisé.
- L'attribut **me** du symbole non-terminal **E** est hérité. Par contre l'attribut **val** est synthétisé.

Pour chaque règle de grammaire, les règles d'affectation définissent la valeur des attributs synthétisés attachés au non-terminal en partie gauche et la valeur des attributs hérités des symboles apparaissant en partie droite. La notation adoptée est la suivante : les noms d'attributs sont indicés par la place du symbole auquel ces attributs se rapportent. La numérotation choisie est la suivante : $\mathbf{X_0} \rightarrow \mathbf{X_1} \ldots \mathbf{X_n}$. Pour faciliter la lecture, et avant d'introduire la notation définitive, nous indiçons aussi les symboles terminaux et non-terminaux par leur place.

- Pour la règle :

$$\mathbf{P}_0 \quad \rightarrow \quad \mathbf{L}\mathbf{C}_1 \tag{1}$$

  on a :

$$\mathbf{me}_1 \quad := \quad \{(\mathbf{n}, 0) \mid \mathbf{n} \in \mathbf{CHAINE}\}$$

  Ceci décrit le fait que, par défaut, la valeur des identificateurs est 0.

- Pour la règle :

$$\mathbf{L}\mathbf{C}_0 \quad \rightarrow \quad \mathbf{C}_1 \ \mathbf{L}\mathbf{C}_2 \tag{2}$$

  on a :

$$\begin{aligned} \mathbf{me}_1 &:= \mathbf{me}_0 \\ \mathbf{me}_2 &:= \mathbf{ms}_1 \\ \mathbf{ms}_0 &:= \mathbf{ms}_2 \end{aligned}$$

  La première commande d'une liste de commandes est exécutée pour la mémoire courante ($\mathbf{me}_1 := \mathbf{me}_0$), puis la suite de commandes est exécutée après modification de cette mémoire par la première commande ($\mathbf{me}_2 := \mathbf{ms}_1$). La mémoire en sortie ($\mathbf{ms}_0$) est celle obtenue après exécution de toute la liste de commandes.

- Pour la règle :

$$\mathbf{LC}_0 \quad \rightarrow \quad \mathbf{C}_1 \tag{3}$$

on a :

$$\begin{array}{lcl} \mathbf{me}_1 & := & \mathbf{me}_0 \\ \mathbf{ms}_0 & := & \mathbf{ms}_1 \end{array}$$

- pour la règle :

$$\mathbf{C}_0 \quad \rightarrow \quad \underline{\mathbf{id}_1} \ \underline{:=_2} \ \mathbf{E}_3 \ \underline{;_4} \tag{4}$$

on a :

$$\begin{array}{lcl} \mathbf{me}_3 & := & \mathbf{me}_0 \\ \mathbf{ms}_0 & := & \mathbf{me}_0 \lhd \{(\mathbf{nom}_1, \mathbf{val}_3)\} \end{array}$$

L'expression est évaluée dans la mémoire courante ($\mathbf{me}_3 := \mathbf{me}_0$). La mémoire en sortie est la mémoire en entrée sauf au point $\mathbf{nom}_1$.

- pour la règle :

$$\mathbf{E}_0 \quad \rightarrow \quad \underline{(_1} \ \mathbf{E}_2 \ \underline{+_3} \ \mathbf{E}_4 \ \underline{)_5} \tag{5}$$

on a :

$$\begin{array}{lcl} \mathbf{me}_2 & := & \mathbf{me}_0 \\ \mathbf{me}_4 & := & \mathbf{me}_0 \\ \mathbf{val}_0 & := & \mathbf{val}_2 + \mathbf{val}_4 \end{array}$$

- pour la règle :

$$\mathbf{E}_0 \quad \rightarrow \quad \underline{\mathbf{id}_1} \tag{6}$$

on a :

$$\mathbf{val}_0 \quad := \quad \mathbf{me}_0(\mathbf{nom}_1)$$

Pour obtenir la valeur d'un identificateur il suffit de consulter sa valeur dans l'association courante.

- pour la règle :

$$\mathbf{E}_0 \quad \rightarrow \quad \underline{\mathbf{num}_1} \tag{7}$$

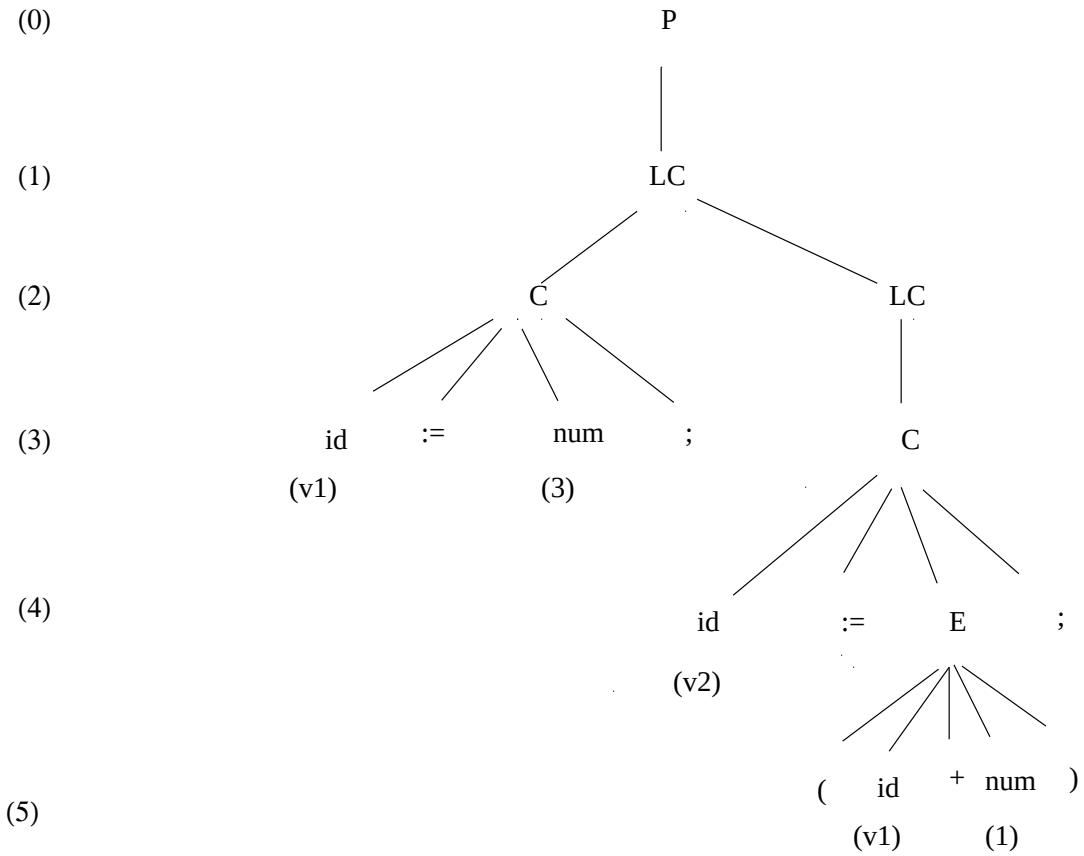
on a :

$$\mathbf{val}_0 \quad := \quad \mathbf{val}_1$$

La valeur d'une constante numérique se calcule à partir de sa représentation.

1.4 Exemple d'évaluation d'attributs

Nous décrivons ici comment les attributs peuvent s'évaluer pour la chaîne d'entrée **v1:= 3; v2 := (v1+1);** . Pour alléger les notations et l'exemple nous considérons l'arbre de dérivation "raccourcï suivant :



Nous décrivons ci-dessous le calcul d'attributs. Les notations utilisées sont les suivantes : une occurrence d'attribut est indicée en premier par le niveau de l'arbre où elle apparaît et en second par sa place, à un niveau donné de l'arbre. Par exemple $\mathbf{me}_{3,5}$ désigne l'attribut associé au cinquième symbole du niveau (3), c'est à dire le non-terminal **C**.

- Au niveau 1 on a :

$$\mathbf{me}_{1,1} \quad := \quad \{(\mathbf{n}, 0) \mid \mathbf{n} \in \mathbf{CHAINE}\}$$

- Au niveau 2 les affectations d'attributs sont :

$$\begin{array}{lcl} \mathbf{me}_{2,1} & := & \mathbf{me}_{1,1} \\ \mathbf{me}_{2,2} & := & \mathbf{ms}_{2,1} \\ \mathbf{ms}_{1,1} & := & \mathbf{ms}_{2,2} \end{array}$$

- Au niveau 3 la dérivation $\mathbf{C} \stackrel{*}{\Rightarrow} \mathbf{v1{:=}\ 3;}$ fournit l'affectation :

$$\mathbf{ms}_{2,1} \quad := \quad \mathbf{me}_{2,1} \lhd \{(\text{"}\mathbf{v}1\text{"}, 3)\}$$

  De la même manière la dérivation $\mathbf{LC} \Rightarrow \mathbf{C}$ du niveau 3 donne :

$$\begin{array}{lcl} \mathbf{me}_{3,5} & := & \mathbf{me}_{2,2} \\ \mathbf{ms}_{2,2} & := & \mathbf{ms}_{3,5} \end{array}$$

- Au niveau 4 la dérivation $\mathbf{C} \stackrel{*}{\Rightarrow} \mathbf{v2 := E;}$ donne :

$$\begin{array}{lcl} \mathbf{me}_{4,3} & := & \mathbf{me}_{3,5} \\ \mathbf{ms}_{3,5} & := & \mathbf{me}_{3,5} \lhd \{(\text{"}\mathbf{v}2\text{"}, \mathbf{val}_{4,3})\} \end{array}$$

- Au niveau 5 la dérivation $\mathbf{E} \stackrel{*}{\Rightarrow} \mathbf{(v1 + 1)}$ fournit l'affectation :

$$\mathbf{val}_{4,3} := \mathbf{me}_{4,3}(\text{"}\mathbf{v}1\text{"}) + 1$$

En résumé on obtient :

$$\begin{array}{lcl} \mathbf{ms}_{2,1} & := & \{(\text{"}\mathbf{v}1\text{"}, 3)\} \cup \{(\mathbf{n}, 0) \mid \mathbf{n} \neq \text{"}\mathbf{v}1\text{"}\} \\ \mathbf{ms}_{1,1} := \mathbf{ms}_{2,2} := \mathbf{ms}_{3,5} & := & \{(\text{"}\mathbf{v}2\text{"}, 4), (\text{"}\mathbf{v}1\text{"}, 3)\} \cup \{(\mathbf{n}, 0) \mid \mathbf{n} \neq \text{"}\mathbf{v}1\text{"} \wedge \mathbf{n} \neq \text{"}\mathbf{v}2\text{"}\} \end{array}$$

2 Grammaires Attribuées

2.1 Définition

Soit **G** une grammaire hors-contexte de la forme $\mathbf{G} = (\mathbf{V_T}, \mathbf{V_N}, \mathbf{S}, \mathbf{R})$. La définition d'une grammaire attribuée à partir de G comprend les éléments suivants :

- L'association à chaque symbole $\mathbf{X}$ de $\mathbf{V_T} \cup \mathbf{V_N}$ de deux ensembles disjoints, éventuellement vides, $\mathbf{HER(X)}$ (les attributs hérités) et $\mathbf{SYN(X)}$ (les attributs synthétisés).
- Pour chaque attribut la donnée d'un domaine de valeurs.
- Pour chaque règle de la forme $\mathbf{X}_0 \rightarrow \mathbf{X}_1 \ldots \mathbf{X_n}$ des règles d'affectation qui permettent :
  - d'affecter les attributs synthétisés de $\mathbf{X}_0$ en fonction des attributs de $\mathbf{X_j}$, $0 \leq \mathbf{j} \leq \mathbf{n}$;
  - d'affecter les attributs hérités des $\mathbf{X_i}$, $\mathbf{i} \neq 0$, en fonction des attributs des $\mathbf{X_j}$, $0 \leq \mathbf{j} \leq \mathbf{n}$.

Remarque : l'axiome n'a pas d'attributs hérités ($\mathbf{HER(S)} = \emptyset$), les attributs synthétisés associés aux symboles terminaux doivent décrire une information qui se calcule à partir de la représentation de ces terminaux.

2.2 Propriétés

Quelle est la sémantique d'une grammaire attribuée ? Pour un arbre de dérivation donné, on obtient un système d'équations dont les inconnues sont des occurrences d'attributs. Ce système s'obtient en appliquant les règles d'affectation aux occurrences d'attributs, ceci pour chaque nœud de l'arbre (voir exemple précédent).

2.2.1 Grammaires attribuées bien formées

Classiquement on s'intéresse à l'existence d'un ordre d'évaluation des attributs, pour tout arbre de dérivation possible. Une grammaire attribuée est dite **bien formée** si et seulement s'il existe un tel ordre. Ceci revient à vérifier que tout système d'équations est sans cycle. Il existe une méthode permettant de déterminer si une grammaire attribuée est bien formée ou non, nous ne la présentons pas ici.

2.2.2 Sélection d'un langage hors-contexte

Les grammaires attribuées peuvent être utilisées pour définir des langages non initialement hors-contexte. Dans ce cas on définit une grammaire attribuée sur un langage incluant celui désiré et on ajoute des conditions sur les attributs. Si ces conditions ne sont pas vérifiées alors le mot est rejeté. Une condition implicite est par exemple l'existence d'une valeur d'attribut, par les règles d'affectation.

Un exemple classique est le langage des mots de la forme $\mathbf{a^n b^n c^n}$ qui n'est pas hors-contexte. On peut définir de manière hors-contexte le langage $\mathbf{a^n b^n c^k}$ et compter le nombre de **a** et **c**. Si ces deux nombres ne sont pas identiques les mots sont rejetés.

Un exemple issu de la compilation est la définition des contraintes de type et de visibilité des identificateurs. Ces contraintes ne peuvent être prises en compte par une grammaire hors-contexte, l'ajout d'attributs permet de les décrire.

2.3 Notations

Les attributs synthétisés sont préfixés par $\uparrow$. Les attributs hérités sont préfixés par $\downarrow$.

2.3.1 Affectation des attributs

On utilisera deux notations pour l'affectation d'attributs :

- **l'affectation explicite** de la forme :

  $$\begin{array}{rcl} \ldots & \rightarrow & \ldots \\ & \textit{affectation} & \mathbf{v := exp} \end{array}$$

  Exemple :

  $$\begin{array}{rcl} \mathbf{E} \downarrow\mathbf{me} \uparrow\mathbf{val} & \rightarrow & \underline{\mathbf{id}} \uparrow\mathbf{nom} \\ & \textit{affectation} & \mathbf{val := me(nom)} \end{array}$$

- **l'affectation implicite** dans laquelle on exprime une valeur d'attribut en donnant une expression fonctionnelle dépendant des autres attributs. Par exemple :

  $$\mathbf{E} \downarrow\mathbf{me} \uparrow\mathbf{me(nom)} \rightarrow \underline{\mathbf{id}} \uparrow\mathbf{nom}$$

  est une autre manière d'exprimer la règle précédente.

2.3.2 Conditions sur les attributs

- **les conditions explicites** de la forme :

  $$\begin{array}{rcl} \ldots & \rightarrow & \ldots \\ & \textit{condition} & \mathbf{P} \end{array}$$

  où **P** est un prédicat sur les attributs de la règle.

- **les conditions implicites** liées aux affectations. Il est imposé que tout attribut ait une valeur, ce qui induit des conditions lorsque les attributs sont définis par des expressions partielles. Par exemple la règle :

$\mathbf{E} \downarrow\mathbf{me} \uparrow\mathbf{me}(\mathbf{nom}) \quad \rightarrow \quad \underline{\mathbf{id}} \uparrow\mathbf{nom}$

contient la condition implicite $\mathbf{nom} \in \mathbf{dom}(\mathbf{me})$, si **me** est une fonction partielle. **dom** est un ensemble défini par $\mathbf{dom}(\mathbf{f}) = \{\mathbf{x} \mid \exists\, \mathbf{y}\ (\mathbf{x}, \mathbf{y}) \in \mathbf{f}\}$.

3 Exemples

3.1 Exemple 1

Nous reprenons l'exemple introductif avec les notations précédentes.

$\mathbf{P} \rightarrow \mathbf{LC} \downarrow\{(\mathbf{n}, 0) \mid \mathbf{n} \in \mathbf{CHAINE}\} \uparrow\mathbf{ms}$ (1)

$\mathbf{LC} \downarrow\mathbf{me} \uparrow\mathbf{ms} \rightarrow \mathbf{C} \downarrow\mathbf{me} \uparrow\mathbf{ms}_1\ \mathbf{LC} \downarrow\mathbf{ms}_1 \uparrow\mathbf{ms}$ (2)

$\mathbf{LC} \downarrow\mathbf{me} \uparrow\mathbf{ms} \rightarrow \mathbf{C} \downarrow\mathbf{me} \uparrow\mathbf{ms}$ (3)

$\mathbf{C} \downarrow\mathbf{me} \uparrow\mathbf{ms} \rightarrow \underline{\mathbf{id}} \uparrow\mathbf{nom}\ \underline{:=}\ \mathbf{E} \downarrow\mathbf{me} \uparrow\mathbf{val}\ \underline{;}$ (4)
affectation $\mathbf{ms} := \mathbf{me} \lhd \{(\mathbf{nom}, \mathbf{val})\}$

$\mathbf{E} \downarrow\mathbf{me} \uparrow\mathbf{val}_2 + \mathbf{val}_4 \rightarrow \underline{(}\ \mathbf{E} \downarrow\mathbf{me} \uparrow\mathbf{val}_2\ \underline{+}\ \mathbf{E} \downarrow\mathbf{me} \uparrow\mathbf{val}_4\ \underline{)}$ (5)

$\mathbf{E} \downarrow\mathbf{me} \uparrow\mathbf{me}(\mathbf{nom}) \rightarrow \underline{\mathbf{id}} \uparrow\mathbf{nom}$ (6)

$\mathbf{E} \downarrow\mathbf{me} \uparrow\mathbf{val} \rightarrow \underline{\mathbf{num}} \uparrow\mathbf{val}$ (7)

3.2 Exemple 2

On suppose maintenant que les identificateurs n'ont pas de valeur par défaut et que tout programme qui contient une consultation d'un identificateur non initialisé est rejeté. Il suffit de considérer **MEM** comme un domaine de fonctions partielles et de modifier la règle (1) en :

$\mathbf{P} \rightarrow \mathbf{LC} \downarrow\{\} \uparrow\mathbf{ms}$ (1)

La règle (4) n'est pas modifiée mais la valeur de l'attribut **ms** est une fonction partielle. De la même manière la règle (6) n'est pas modifiée mais contient la condition implicite $\mathbf{nom} \in \mathbf{dom}(\mathbf{me})$, et donc l'utilisation d'un identificateur dans une expression n'est correcte que s'il est déjà apparu en partie gauche d'affectation.

3.3 Exemple 3

Nous donnons ci-dessous un exemple de grammaire attribuée permettant de décrire le langage $\mathbf{a^n b^n c^n}$.

$$\mathbf{S} \rightarrow \mathbf{A}\ \uparrow\mathbf{na}\ \mathbf{C}\ \uparrow\mathbf{nc} \quad (1)$$
$$\textit{condition}\quad \mathbf{na} = \mathbf{nc}$$

$$\mathbf{A}\ \uparrow\mathbf{na}+1 \rightarrow \underline{\mathbf{a}}\ \mathbf{A}\ \uparrow\mathbf{na}\ \underline{\mathbf{b}} \quad (2)$$

$$\mathbf{A}\ \uparrow 0 \rightarrow \varepsilon \quad (3)$$

$$\mathbf{C}\ \uparrow\mathbf{nc}+1 \rightarrow \underline{\mathbf{c}}\ \mathbf{C}\ \uparrow\mathbf{nc} \quad (4)$$

$$\mathbf{C}\ \uparrow 0 \rightarrow \varepsilon \quad (5)$$

3.4 Exemple 4

Voici un second exemple décrivant le même langage, à partir de la même grammaire hors-contexte.

$$\mathbf{S} \rightarrow \mathbf{A}\ \uparrow\mathbf{na}\ \mathbf{C}\ \downarrow\mathbf{na} \quad (1)$$

$$\mathbf{A}\ \uparrow\mathbf{na}+1 \rightarrow \underline{\mathbf{a}}\ \mathbf{A}\ \uparrow\mathbf{na}\ \underline{\mathbf{b}} \quad (2)$$

$$\mathbf{A}\ \uparrow 0 \rightarrow \varepsilon \quad (3)$$

$$\mathbf{C}\ \downarrow\mathbf{nc} \rightarrow \underline{\mathbf{c}}\ \mathbf{C}\ \downarrow\mathbf{nc}-1 \quad (4)$$
$$\textit{condition}\quad \mathbf{nc} \neq 0$$

$$\mathbf{C}\ \downarrow\mathbf{nc} \rightarrow \varepsilon \quad (5)$$
$$\textit{condition}\quad \mathbf{nc} = 0$$

Syntaxe contextuelle

Les programmes syntaxiquement correct (conforme à la grammaire hors-contexte) sont filtrés par la syntaxe contextuelle pour être validés ou non.

I. Identification des noms

Identificateur : symbole apparaissant dans les programmes et désignant des objets (variable, procédure...)
Déclaration : association d'un identificateur à un certain objet

```
x: integer
```

Le symbole 'x' est associé à une variable de type integer
occurrence de définition / occurrence d'utilisation

```
x : integer                    := y;
```

occurrence de définition occurrence d'utilisation

La liaison entre les occurrences de définition et les occurrences de liaison constituent les règles d'identification des noms dans un langage

– règles de portée : définissent les parties d'un programme dans laquelle une déclaration est effective.

Ada : paquetage, procédure/fonction ...
C : fichier, fonction, bloc { }
C++ : classe, namespace, ...

– règles de visibilité

Lien entre les occurrences de définition et d'utilisation.
1) Une occurrence d'utilisation doit être sous la portée de l'occurrence de définition à laquelle elle se rapporte.
2) Règles pour les ambiguïtés
  – Définition la plus imbriquée
  – Référence explicite à une définition

Ada : nom_paquetage.idf
C++ : nom_namespace::idf

Exemple 1 : Ada

```
declare x : integer;        -- debut portée 1ère def de x
begin
        declare x : integer := x ; -- debut portée 2e def de x
        begin
                ... x ...
        end                         -- fin portée 2e def de x
end                         -- fin portée 1ère def de x
```

Ceci est refusé en Ada car au niveau du déclare, on aurait deux 'x' différents.
En effet, la règle de visibilité impose que dans une même région chaque occurrence d'utilisation fait référence à la même occurrence de définition.

Exemple 2 (C) :

```
        int x;
        void main ( ){
                int x = x ;
                ... x ...
        }
```

Accepté en C : les déclarations ne sont effectives qu'après leur apparition.

II. Les environnements

Comment spécifier les règles d'identification des noms dans un langage de programmation.
Un environnement décrit en chaque point des programmes :

– Quels sont les symboles visibles
– Quels sont les objets associés à ces symboles

Modifiés par les déclarations et les fins de région
Consulté par les instructions

Environnement : Nom -|-> Objet
-|-> : fonction partielle
Si langage avec surcharge : Nom -|-> P (objet)
Exemple :

```
package exemple is
        subtype t is integer range 0..5;
        x : t;
        procedure p (x : t) is -- début environnement e1
        begin
                ...
        end p;-- fin environnement e1
        -- environnement e2
end exemple;
```

Environnement e1 :

exemple	->	paquetage
t	->	type
x	->	param_in
p	->	procedure

Environnement e2 :
Environnement e1 en changeant x en

x	->	variable

Construction des environnements :
– Ajouter des déclarations
– Enlever des déclarations en sortie de bloc

On manipule des piles d'environnement qui décrivent la structure de bloc

exemple :

Environnement 1 : x -> param_in
Environnement 2 : exemple -> type x -> variable p -> procedure

Recherche à partir du sommet de la pile : on trouve la définition la plus imbriquée.

Opérations sur les environnement :

(+) : union disjointe
e1 (+) e2 = e1 U e2
si dom(e1) ∩ dom(e2) = Φ
indéfini sinon

dom(e) : domaine de définition de la fonction

Opération utilisée dans une même région pour empêcher les doubles déclarations.
/ : empilement
e1/e2 : e1 empilé sur e2

(e1/e2)(n) = e1(n) si n € dom(e1)
= e2(n) si n € dom(e2)
et n € dom(e1)
= indéfini sinon

Opération utilisée pour les structures de bloc
Exemple : un langage à structure de bloc à la C
{ déclaration, instruction }

```
Programme       ->      bloc ↓Φ
bloc ↓sup       ->      { SD ↑e1
                            ;
                        SI ↓(e1/sup)
                        }

SD ↑e           ->      D ↑e1 e:=e1
SD ↑e           ->      D ↑e1 e:=e1(+)e2
                        ; SD ↑e2

                        : type ↑t e:={nom->t}
D ↑e            ->      idf ↑nom

type ↑t         ->      int t = type_int
                ->      * type ↑t1 t = ref(t1)

SI              ->      I
                        ; SI

SI              ->      place
                        :=
                        exp

I↓e             ->      bloc ↓e
```

III. Vérification du typage

```
SI ↓e           ->      I ↓e
SI ↓e           ->      I ↓e
                        ;
                        SI ↓e

I               ->      place ↓e ↑t1
                              condition t1 = t2
                        := exp ↓e ↑t2

exp ↓e ↑t       ->      place ↓e ↑t1 t:=t1

place ↓e ↑t     ->      idf ↑nom condition nom € dom(e) t:= e(nom)

place ↓e ↑t     ->      place ↓e ↑t1
                        *
```

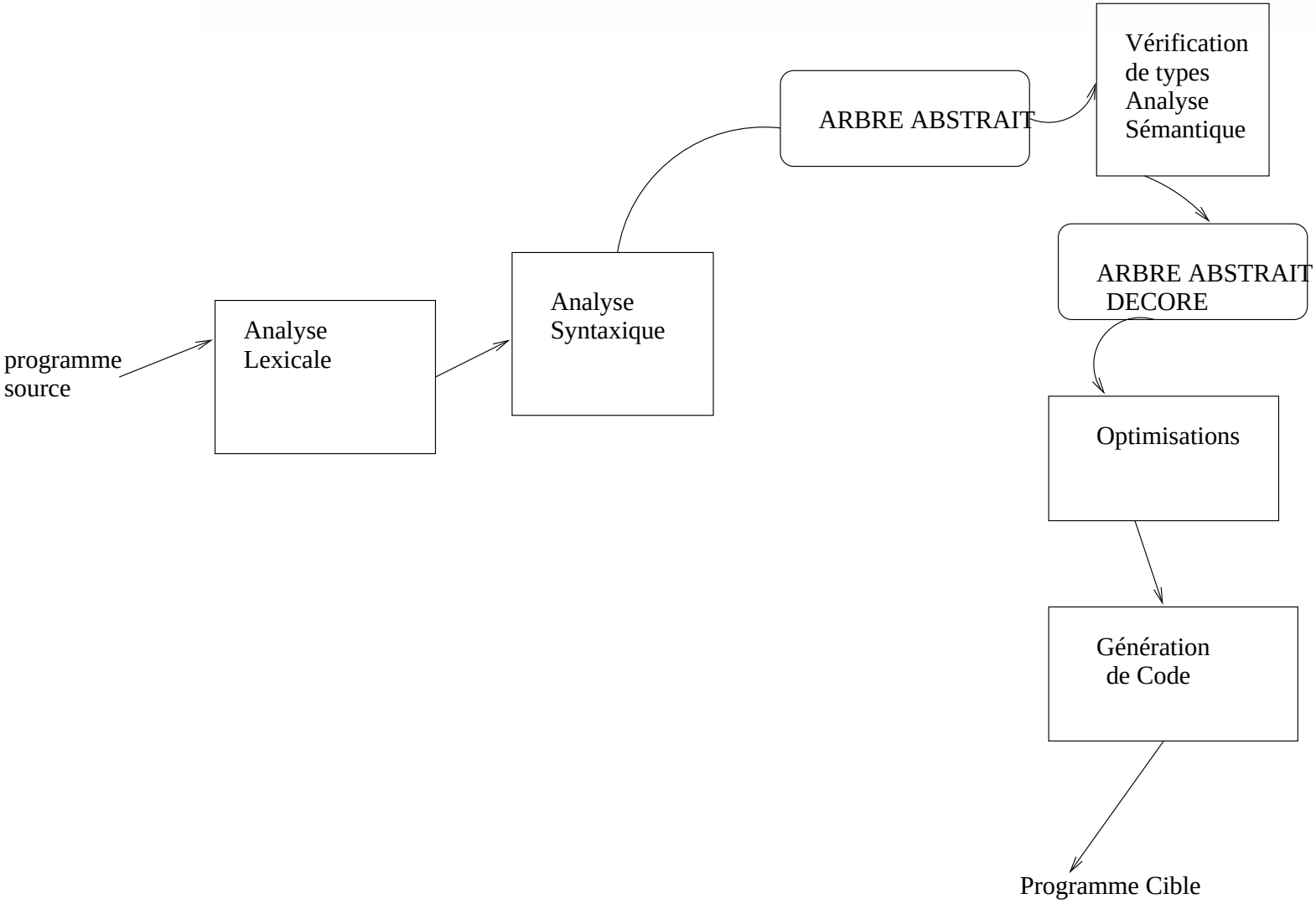
Génération de Code (1)

Structure globale
GC : ArbreAbstrait→list(Inst)

Augustin Lux

Compile Télécom Cours 8 - 14-11-06

Architecture d'un compilateur



Génération de code: départ - arrivée

Données: la représentation interne du programme source

- table des symboles
- arbre abstrait

qui ont été

- vérifiés
- décorés

Notamment: il n'y a pas d'erreurs dans le programme source.

$\Rightarrow$ C'était l'objet des cours précédents.

Génération de code: départ - arrivée

Résultat: programme en “langage machine”

- code binaire en mémoire (jamais)
- fichier format ELF (ou autre)
- en langage d’assemblage spécifique
- en langage d’assemblage générique ou simplifié
- en langage C (oui!)
- . . .

Plan (3 séances)

- L’arbre abstrait
  - grammaires abstraites
  - attributs
  - parcours d’ arbres
- Génération de code en tant que parcours d’arbre
- Schémas de codage
- Représentation des données
- Algorithmes plus sophisiqués (expressions)
- Optimisation

Illustrations avec des demos.

Représentation interne du programme source

Rappel

Le programme source est traduit en 2 structures de donnéees:

- La **table des symboles**, construite à partir des déclarations.
- **L'arbre abstrait** construit à partir des instructions.

Construction de l'arbre abstrait (Rappel)

L'arbre abstrait est construit par l'analyse syntaxique, avec un attribut spécifique, analogue à l'attribut val de l'interpréteur: la valeur est un noeud d'arbre:

$$
\begin{aligned}
E(\uparrow a) &\rightarrow F(\uparrow g)\ E'(\downarrow g \uparrow a) \\
E'(\downarrow g \uparrow a) &\rightarrow oA(\uparrow op) \quad E(\uparrow v1)) \\
&\qquad a = newArbreOP(op, g, v1) \\
E'(\downarrow a \uparrow a) &\rightarrow \epsilon
\end{aligned}
$$

C'est simple - il suffit d'avoir les bons constructeurs.

Arbre abstrait - Grammaire abstraite

Il faudra définir de nombreux types d'arbres, et programmer de nombreux parcours d'arbre.

Cette programmation, en Ada, Java, C . . . introduit des lourdeurs: beaucoup de code à écrire . . . pour peu de choses! *Voir en annexe, et les exemples traités en TD.*

Mais: c'est dependant d'un langage particulier, et plein de détails rebarbatifs. Il nous faut une notation plus abstraite pour les arbres abstraits!

Grammaire Abstraite - Grammaire d'Arbres

Une Grammaire Abstraite est une notation qui spécifie

- un ensemble de structures d'arbres
- un ensemble de procédures de parcours d'arbres

Une Grammaire Abstraite Attribuée est une notation qui spécifie

- un ensemble de structures d'arbres
- un calcul selon un parcours d'arbres.

Intérêt: la spécification est compacte, et indépendante du langage de programmation.

Grammaire Abstraite - exemple 1

Expressions

Grammaire de réécriture:

$$
\begin{aligned}
E &\rightarrow F\ E' \\
E' &\rightarrow oA\ E \mid \epsilon \\
F &\rightarrow T\ F' \\
F' &\rightarrow oM\ F \mid \epsilon \\
T &\rightarrow \text{const} \mid \text{idf} \mid (E) \\
oA &\rightarrow + \mid - \\
oM &\rightarrow * \mid /
\end{aligned}
$$

Grammaire abstraite:

$$E \rightarrow \text{const} \mid \text{idf} \mid op\ E\ E$$

Il reste seulement l'information essentielle.

Grammaires Abstraites

Une grammaire abstraite ou grammaire d'arbres définit un ensemble de structures d'arbres. Elle s'écrit avec des règles, comme une grammaire de réécriture.
Mais le sens des règles est très différent:

- un non-terminal désigne une structure d'arbre
- la partie droite donne une liste de sousarbres et autres champs.

Grammaire d'arbres attribuée

Avec des attributs dans une grammaire d'arbres on définit

les paramètres d'un parcours d'arbre.

Grammaire Abstraite - exemple 2

Le langage des types en LX.
Grammaire de réécriture:

$$\begin{aligned}
\text{Type} &\rightarrow int \\
\text{Type} &\rightarrow bool \\
\text{Type} &\rightarrow access\ \text{Type} \\
\text{Type} &\rightarrow array[num]\ of\ \text{Type}
\end{aligned}$$

Grammaire abstraite:

$$\begin{aligned}
\text{Type} &\rightarrow int \\
\text{Type} &\rightarrow bool \\
\text{Type} &\rightarrow \text{access Type} \\
\text{Type} &\rightarrow array\ num\ \text{Type}
\end{aligned}$$

Grammaire Abstraite Attribuée

Grammaire abstraite avec attributs - calcul de la taille:

$$
\begin{aligned}
\text{Type}(\uparrow 4) &\rightarrow int \\
\text{Type}(\uparrow 1) &\rightarrow bool \\
\text{Type}(\uparrow 4) &\rightarrow access\ \text{Type}(\uparrow s1) \\
\text{Type}(\uparrow n * s1) &\rightarrow array\ num(\uparrow n)\ \text{Type}(\uparrow s1)
\end{aligned}
$$

Comparer avec les codes en Ada (cf. annexe).

Traduction grammaire –> structures d'arbre

Une grammaire abstraite spécifie un ensemble de structures d'arbre, qu'on peut coder des façon systématique de différentes manières, dans différents langages

- A chaque non-terminal N correspond une déclaration (*classe, record, . . .)*
- *L'ensemble des parties droites des règles pour N est codé par des champs (avec discriminant en Ada).*
- *Pour chaque règle, il y a un "constructeur" (une fonction qui construit le nœud d'arbre).*
- *Pour chaque règle, il y a des accesseurs, modifieurs etc. pour les champs.*
- *Il y a une fonction pour tester la nature d'un nœud.*

Traduction grammaire –> parcours d'arbre

Une grammaire abstraite attribuée spécifie un parcours d'arbre. Le passage grammaire $\Rightarrow$ programme de parcours est systématique et peut être fait automatiquement.

- Pour chaque non-terminal N, il y a une procédure $parcours N(Arbre\ a)$.
- Le corps de $parcours N$ est un aiguillage, dans lequel chaque règle pour N contribue un cas.
- L'aiguillage se fait sur Nature(a)
- Chaque symbole en partie droite d'une règle se traduit par un appel de sa procédure $parcours R$

Règle "terminale": pas de champ sousarbre.

Parcours d'arbre

Les détails du codage varient selon le calcul considéré.
Nos exemples de base:

- calcul de l'attribut TAILLE d'un type
- calcul de la décoration EXPTYPE d'un nœud
- la génération de code.

Génération de Code (Principe - 1)

La génération de code est spécifiée par une grammaire d'arbre avec un attribut synthétisée: L : LIST(INSTR).
C'est donc un parcours d'arbre, au cours duquel chaque nœud produit une liste d'instructions à partir des listes fournies par les sous-arbres.
Pour chaque type de nœud, nous allons analyser son schéma de traduction:

- affectations
- expressions
- instruction conditionnelle
- itération WHILE
- itération FOR
- appel de fonction
- ...

Un début d'exemple

Instructions dans un langage comme LX

$$
\begin{array}{rcl}
I(\uparrow l) & \rightarrow & \text{iAff}(\uparrow l) \mid \text{iCond}(\uparrow l) \mid \text{iWhile}(\uparrow l) \mid \text{LI}(\uparrow l) \\
\text{iAff}(\uparrow l) & \rightarrow & \text{aff idf } E(\uparrow le) \\
 & & l := ?? \\
\text{iCond}(\uparrow ?) & \rightarrow & \text{cond } E(\uparrow le)\ I(\uparrow l1)\ I(\uparrow l2) \\
\text{iWhile}(\uparrow ?) & \rightarrow & \text{while } E(\uparrow le)\ LI(\uparrow li) \\
\text{LI}(\uparrow l) & \rightarrow & \text{list } I(\uparrow l1)\ \text{LI}(\uparrow l2) \\
 & & l := concat(l1, l2) \\
\text{LI}(\uparrow ()) & \rightarrow & \text{NULL}
\end{array}
$$

? pose la question: comment composer le résultat *l* ?

Génération de code (Principe - 2)

C’est simple dans le principe, mais d’une grande complexité dans le détail.

- structure du processeur
- longueur du mot machine
- jeu d’instructions
- représentation des données
- liaison avec système d’exploitation
- avec l’éditeur des liens.

Schémas de traduction

Instruction d'affectation (1)

iAff($\uparrow l$) $\rightarrow$ aff idf $E(\uparrow le)$
$l :=??$

Cas simple: `place` est un `idf`.

`idf := expr ;`

code de expr
$\Rightarrow$ résultat dans registre R
`Store R dans adresse de idf`

Instruction d'affectation (2)

iAff($\uparrow l$) $\rightarrow$ aff idf $E(\uparrow le)$

$l :=??$

Cas général: `place` est une expression.

`place := expr ;`

code de expr
$\Rightarrow$ résultat dans registre R
calcul de l'adresse place
$\Rightarrow$ résultat dans registre A
move R,(A)

Pour objet simple, move est 1 instruction. Pour objet plus grand, c'est un appel de memcpy.

Codage des expressions

Le codage des expressions fait appel à des algorithmes sophistiqués - voir plus loin pour les conditions, et le prochain cours.
Attribut supplémentaire: registre cible.

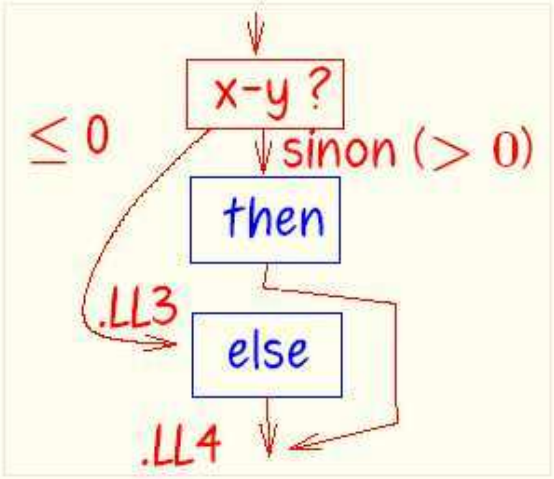
Structure de contrôle IF, WHILE, FOR:

On génère des instructions de branchement conditionnel.

Branchement Conditionnel - Exemple IF

Codage d'une conditionnelle, par gcc pour sparc:

```
int main () {
  int x, y ;
  if (x > y) { x = 42 ; }
  else       { x = 43 ; }
}
```




```
main:
    save    %sp, -120, %sp
    ld  [%fp-20], %o0  ! x
    ld  [%fp-24], %o1  ! y

    cmp %o0, %o1   ! x - y ?
    ble .LL3

    mov 42, %o0
    st  %o0, [%fp-20]
    b   .LL4
.LL3:
    mov 43, %o0
    st  %o0, [%fp-20]
.LL4:
    ret
    restore
```

Codage du while - version 1

Version “normale”: avec un test au début.

```
A; while(x<3) B; C
```

```
    code de A
etiA: cmp Ad(x),3
branchement si not<  à etiB
code de B
branchement toujours à etiA
etiB: code de C
```

Codage du while (2)

Mais:

Pour certains processeurs, le temps d'exécution d'un test diffère selon qu'il réussit *(il y a branchement)* ou non *(on va tout droit)*.

Or, dans une boucle, le test test réussit n fois, et échoue 1 fois.

Codage précédent du WHILE: on va tout droit n fois.

Codage du while - version 2

Version 2: test à la fin - on va tout droit 1 fois.

```
A; while(x<3) B; C
```

code de A
branchement toujours à etiB
etiA: code de B
etiB: cmp Ad(x),3
branchement si< à etiA
code de C

Demo parcours de géneration de code

Une procédure pour chaque “non-terminal” de la grammaire abstraite:

- GenInstr
- GenAffect
- GenIf
- GenWhile
- . . .

partageant toutes le même prototype:

ListInstr GenXX(Arbre i);

Génération de Code (2)

Augustin Lux

Compile Télécom Cours 9 - 21-11-06

Génération de code (Rappel du cours 1)

La génération de code

- est un parcours de l'abre abstrait
- spécifié par grammaire abstraite attribuée
- avec un schéma de traduction pour chaque arbre de base

Exemples d'exécution:
avec
peigneD $(((e/d) * c) - b) + a$
peigneG: $a + (b - (c * (d/e)))$
ex90 $(a - 3 + x) * (x + 10 - (a + b))$
ex91 $(a - b * c) + (d * (e/f) - (g + h * i))$

Schémas de traduction

Les schémas sont assez simples pour les INSTRUCTIONS DE CONTRÔLE:

- instruction composée: BEGIN...END, { . . . }
- conditionnelle: IF
- itération: WHILE, FOR
- cas: SWITCH, CASE

qui ne font que manipuler les listes qui remontent des sousarbres.

Schéma de traduction: Expressions

Avec les expressions, il faut manipuler des valeurs, utiliser des registres, cela introduit des problèmes difficiles (surprise).

Fragment de grammaire utilisé: dans ce cours, nous traitons uniquement les expressions entières composées de

- constante
- variable
- opérateur binaire (arithmétique - booléen)

Indexation, sélection, et appel de fonction seront traités ultérieurement (en TD en en C).

Le langage d'assemblage (1)

Pour notre cours, et pour les exemples, nous introduisons un assembleur "abstrait", simplifié, dont les instructions seront facilement codées pour un assembleur concret.

1. Registres: `R0, R1,...Rk`
2. Valeurs immédiates: `#1 #2` ...
3. On notera `@i` l'adresse de l'identificateur i, dûment identifié.

Nous ajouterons des éléments au langage selon besoin.

Par exemple une pile, des registres flottants ...

Le langage d'assemblage (2)

Instructions d'accès mémoire: `LOAD, STORE`
Modes d'adressage

instruction	effet
`LOAD Rj,Ri`	`Ri:=Rj`
`LOAD #d,Ri`	`Ri:=d`
`LOAD d(Rj),Ri`	`Ri:=Mem[Rj+d]`
`LOAD a,Ri`	`Ri:=Mem[a]`

instruction	effet
`STORE Ri,d(Rj)`	`Mem[Rj+d]:=Ri`
`STORE Ri,d(Rj,Rk)`	`Mem[Rj+Rk+d]:=Ri`
`STORE Ri,a`	`Mem[a]:=Ri`

Le langage d'assemblage (3)

Instructions arithmétiques à 2 adresses

- `ADD ad,Ri` addition `Ri + Mem[ad]`, résultat dans `Ri`
- `SUB ad,Ri`
- `MUL ad,Ri`
- `DIV ad,Ri`
- `...`

Attention à l'ordre des opérandes: op2,op1!

Nous ajouterons des instructions librement, selon nos besoin: comparaisons, branchements ...

Le langage d'assemblage (4)

Modes d'adressage pour instructions arithmétiques et logiques.

Example: SUB

instruction	effet
`SUB Rj,Ri`	`Ri:=Ri-Rj`
`SUB #d,Ri`	`Ri:=Ri-d`
`SUB d(Rj),Ri`	`Ri:=Ri-Mem[Rj+d]`
`SUB a,Ri`	`Ri:=Ri-Mem[a]`

Traduction d'une expression: CoderExp

Problème: comment générer du code efficace pour les expressions?

$$a + 3 - x$$

$$(a + b) * (c - 1)$$

On suppose que l'ordre d'évaluation des opérandes arithmétiques est libre (comme en Ada et en C), et non imposé par le langage (comme en Java).

(A ne pas confondre avec l'ordre de priorité!)

Traduction d'une expression: CoderExp

Problème: comment générer du code efficace pour les expressions?

Efficace signifie: avec un minimum d'instructions.

1. c'est plus compliqué qu'on ne l'imagine!
2. . . . on va couper les cheveux en quatre.

Où est le problème?

Prenons quelques exemples

$a - 3 + x$ dans le registre R0

```
LOAD @a,R0
SUB  #3,R0
ADD  @x,R0
```

$\Rightarrow$ pas de problème.

Où est le problème?

Autre exemple: $(a - 3 + x) * (x + 10 - (a + b))$ dans R0

```
LOAD    @a,R0
ADD     @b,R0     ou mettre le resulta
STORE   R0,@temp  on utilise un tempor
LOAD    @x,R0
ADD     #10,R0
SUB     temp,R0
STORE   R0,temp   on reutilise le temp
LOAD    @a,R0
SUB     #3,R0
ADD     @x,R0
MUL     temp,R0
```

$\Rightarrow$ ca se corse, est-ce que ce code est optimal??

Premier algorithme

Algorithme

1. en une passe
2. un seul registre: R0
3. temporaires gérés explicitment, par exemple avec la pile.

Algorithme codage expr –> R0

Temporaires explicites, résultat dans R0)

```
procedure CoderExp(E) :
  si E est une feuille
    Generer (LOAD, Operande(E), R0)
  sinon si E est de la forme E1 op F
    CoderExpr(E1)   - resultat dans R0
    Generer (OP, Operande(F), R0)
  sinon si E est de la forme E1 op E2
    CoderExp(E2)   - resultat dans R0
    temp := Allouer_Temp()
    Generer(STORE,R0,temp)
    CoderExp(E1)   - en utilisant R0
    Generer (OP, temp, R0)
    liberer_temp()
```

Premier algorithme - variantes

On obtient des variantes de cet algorithme avec différentes façons de gérer les temporaires:

1. Registres: très utile: on peut faire des opérations directement, on évite l'instruction STORE.
2. Solution mixte: k registres, puis sur la pile
3. temporaires ailleurs que sur la pile

Algo1: Peignes droite - gauche

GenExp1 pour

$(((e/d) * c) - b) + a$

```
LOAD    @e      R0
DIV     @d      R0
MUL     @c      R0
SUB     @b      R0
ADD     @a      R0
```

Code optimal

GenExp1 pour

$a + (b - (c * (d/e)))$

```
LOAD    @d      R0
DIV     @e      R0
STORE   R0      1(EBP)
LOAD    @c      R0
MUL     1(EBP)  R0
STORE   R0      1(EBP)
LOAD    @b      R0
SUB     1(EBP)  R0
STORE   R0      1(EBP)
LOAD    @a      R0
ADD     1(EBP)  R0
```

c'est mauvais: 11 instructions.

Peigne gauche: algo-1reg vs algo2

GenExp2 pour
$a + (b - (c * (d/e)))$

```
LOAD    @a      R0
LOAD    @b      R1
LOAD    @c      R2
LOAD    @d      R3
DIV     @e      R3
MUL     R3      R2
SUB     R2      R1
ADD     R1      R0
```

8 instructions, 4 registres.

GenExp3 pour
$a + (b - (c * (d/e)))$

```
LOAD    @c      R0
LOAD    @d      R1
DIV     @e      R1
MUL     R1      R0
LOAD    @b      R1
SUB     R0      R1
LOAD    @a      R0
ADD     R1      R0
```

8 instructions, 2 registres.

Premier algorithme - Conclusion

Algorithme correct, mais peut donner un résultat mauvais.

Peigne droit: cas favorable.
On a un code optimal - un seul registre suffit, il y a 1 instruction par opérateur, plus 1 `LOAD` pour la feuille gauche en bas du peigne.

Peigne gauche: cas défavorable.

Il faut 1 temporaire par opérateur. C'est gênant: si cela dépasse le nombre de registres, il faut des `STORE` coûteux!

Peut-on faire mieux?

Deuxième algorithme

Problème: pendant le calcul du deuxième sous-arbre, le résultat du premier occupe un temporaire (bloque un registre).

Idée: calculer le sous-arbre “le moins gros” d’abord: minimiser le nombre de temporaires.

Algorithme

1. en deux passes
2. k registres: R0 . . .
3. . . . et temporaires

Premier passage

Premier passage: parcours d'arbre pour le calcul

- du nombre minimal de temporaires
- et de $ng > nd$

Deuxième passage

```
InstList pass2(Arbre x,Reg r,RegList regs)
```

On génère les instructions comme le premier algorithme, mais avec le bon ordre de parcours des sous-arbres: le plus petit d'abord.

L'arguments $regs$: liste des registres disponibles - gestion "futée".

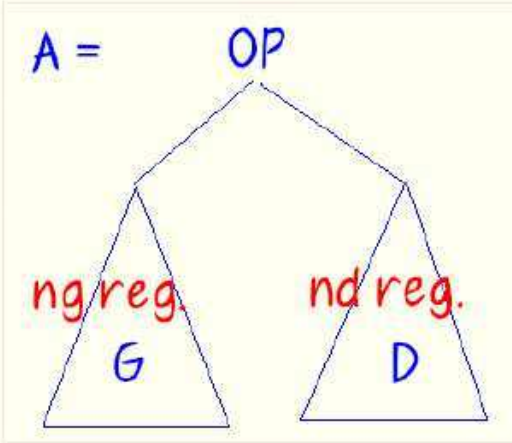
gencode2.pdf

File Edit Document View Window Help

7.1.6.a Nombre de registres nécessaires

En commençant à gauche : max(ng, nd+1)

```
code de G    !  ng regs
...          !
res dans Ri  !  reg ''bloqué''

code de D    !  nd regs
...          !  sans toucher à Ri
res dans Rj  !
op.  Ri,Rj
```



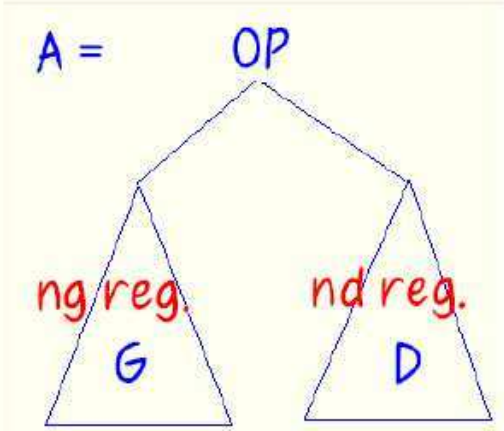
gencode2.pdf

File Edit Document View Window Help

Nombre de registres nécessaires

En commençant à droite : max(nd, ng+1)

```
code de D       !  nd regs
...             !
res dans Rj     !  reg ''bloqué''

code de G       !  ng regs
...             !  sans toucher à Rj
res dans Ri     !

op.  Ri,Rj
```



Calcul nombre de temporaires

NbReg(A) :

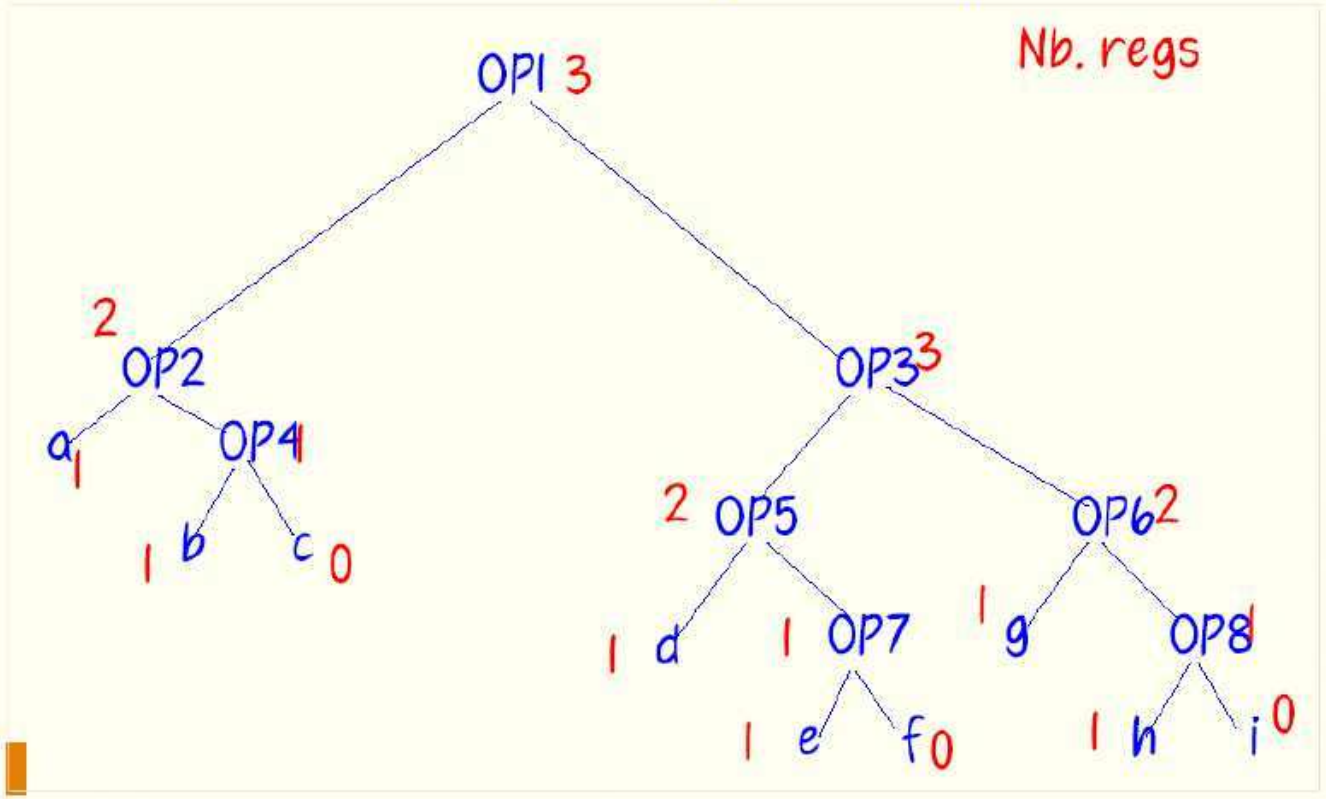
- si A est une feuille droite: – pas besoin de registre 0
- sinon si A est une feuille gauche: il faut 1 registre 1
- sinon A = G OP D;
  ng = NbReg(G); nd = NbReg(D);
  si ng $\neq$ nd alors max(ng,nd)
  sinon ng + 1

NB: il faut un attribut hérité pour savoir si feuille gauche / droite.

gencode2.pdf

File Edit Document View Window Help

7.1.6.b Un exemple complet



Algorithmes détaillés

Vous trouvez les algorithmes complets dans des fichiers annexes (cf. kiosk).

- `GenCode.cpp` squelette général d'un générateur.
- `GenExp.cpp` 3 versions de la génération d'expressions.

Ces fichiers donnent une traduction en "langage algorithmique" du code utilisé pour les demonstrations (écrit en Lisp/Scheme).

Expressions - Conditions

Expressions - Conditions

Problème: comment générer du code efficace pour les conditions complexes?

```
if C1 and (C2 or C3)
    then I1 else I2 end if;
```

Code pour condition complexe

Méthode I: Vision “données”

On calcule explicitement la valeur du booléen de la condition:

```
X := C1 and (C2 or C3);
if X then I1 else I2 end if;
```

- simple - on compile une expression comme ci-dessus.
- ne marche pas pour `and then, or else`, et les conditions en C.

‘L’évaluation “si besoin” des arguments des opérateurs booléens pose un nouveau problème.

Code pour condition complexe

Méthode II: Vision “flot de contrôle

```
if C1 then
  if C2 then
    I1   //C1 and (C2 or C3) est vrai
  else
    if C3 then
      I1   //C1 and (C2 or C3) est vrai
    else
      I2   //C1 and (C2 or C3) est faux
    end if
  end if
else
  I2   //C1 and (C2 or C3) est faux

end if
```

Méthode 2: + et -

Avantages:

- conditions élémentaires
- évaluation seulement si nécessaire

Inconvénients:

- on code plusieurs fois I1, I2 . . .

. . . si on s'y prend mal.

$\Rightarrow$ D'où la procédure suivante.

Algorithme GenCond - spécification

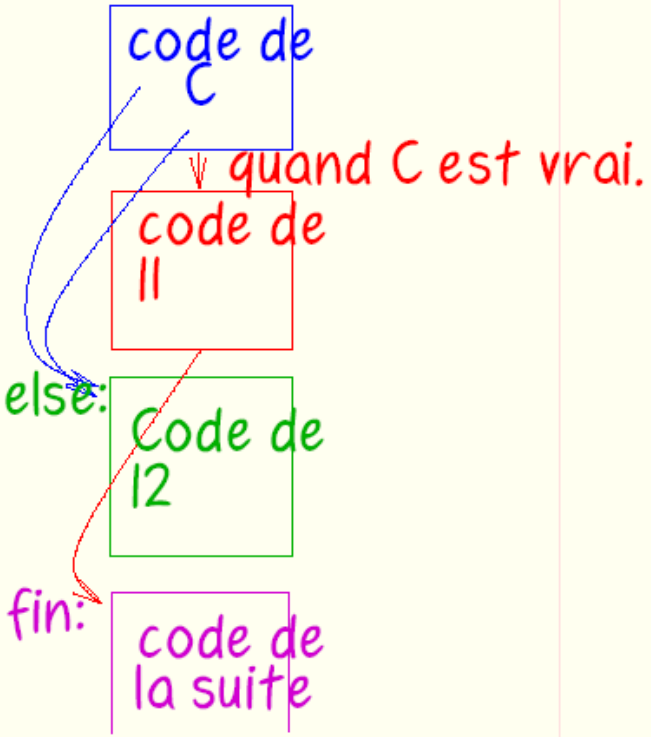
`procedure GenCond(C,saut,cible)`
– `C` est une condition booléenne
– `saut` est un booléen
– `cible` est une étiquette

`GenCond(C,saut,cible)` génére une séquence de code qui vise à déterminer `C`. Lorsqu'on est sûr que `C=saut`, on génére un branchement à `cible` ...sinon l'exécution se poursuit en séquence: c'est le cas `C= not saut`

Algorithme GenCond

Illustration de la situation pour le codage de
`if C then I1 else I2`

```
GenCond (C, faux, ‘‘else’’)
-- on n’arrive ici
que si C est vrai
GenInst (I1)
GenBranch (BRA, ‘‘fin’’)
GenEtiq (‘‘else’’)
GenInst (I2)
GenEtiq (‘‘fin’’)
```



Algorithme GenCond

`procedure GenCond(C,saut,cible)`

si C est une condition simple:

générer un test et un branchement, comme vu plus haut.

Remarque: C peut contenir des calculs arbitrairement com-

plexes - cela renvoie au traitement des expressions.

Algorithme GenCond (suite)

`procedure GenCond(C,saut,cible)`

si `C` est de la forme `not C1`

`GenCond(C1,not saut,cible)`

Algorithme GenCond (suite)

```
procedure GenCond(C,saut,cible)
```

si C est de la forme `C1 and C2`

```
  si saut = faux
    GenCond(C1,faux,cible)
    GenCond(C2,faux,cible)
  si saut = vrai
    GenCond(C1,faux,AUX)
    GenCond(C2,vrai,cible)

    GenEtiquette(AUX)
```

Algorithme GenCond (suite)

```
procedure GenCond(C,saut,cible)
```

si C est de la forme `C1 or C2`

```
  si saut = vrai
    GenCond(C1,vrai,cible)
    GenCond(C2,vrai,cible)
  si saut = faux
    GenCond(C1,vrai,AUX)
    GenCond(C2,faux,cible)

    GenEtiquette(AUX)
```

GenCond - Exemple GCC

```
void main () {
  int a, b, c ;
  int x ;

  a = 1 ; b = 2 ;
  c = 3 ; x = 0 ;

  if ( (a == b) ||
       ( c == b)) {
    x = 123 ;
  }
  else {
    x = 567 ;
  }
}
```

```
main:
    ...
    movl 1,-4(%ebp)
    movl 2,-8(%ebp)
    movl 3,-12(%ebp)
    movl 0,-16(%ebp)

    movl -4(%ebp),%eax   !
    cmpl %eax,-8(%ebp)   ! cmp a, b
    je .L3               ! si egal aller a I

    movl -12(%ebp),%eax  ! cmp c, b
    cmpl %eax,-8(%ebp)
    je .L3               ! si egal aller a I

    jmp .L2              ! aller a ELSE
.L3:                     ! THEN
    movl 123,-16(%ebp)   !
    jmp .L4              ! aller a FIN
.L2:
    movl 567,-16(%ebp)   ! ELSE
.L4:                     ! FIN
    ...
```

Génération de Code (3)
Optimisations - Compléments

Augustin Lux

Compile Ensimag + Télécom Cours 10 28-11-06

GenCod: Rappel des épisodes précédents

Algorithmes de génération de code pour expressions:

- gestion de temporaires
- utilisation de registres
- minimisation du nombre de registres/temporaires
- expressions booléennes avec évaluation apresseus

GenCod: Rappel des épisodes précédents

La génération de code

- est un parcours de l'arbre abstrait
- spécifié par grammaire abstraite attribuée
- avec un schéma de traduction pour chaque type de nœud d'arbre
- une algorithmique riche, notamment pour les expressions.

Exemples d'exécution: avec

ex91 $(a - b * c) + (d * (e/f) - (g + h * i))$

ex92 $(d * (e/f) - (g + h * i)) + (a - b * c)$

Peigne gauche: algo-1reg vs algo2

GenExp2 pour $a + (b - (c * (d/e)))$

```
LOAD    @a      R0
LOAD    @b      R1
LOAD    @c      R2
LOAD    @d      R3
DIV     @e      R3
MUL     R3      R2
SUB     R2      R1
ADD     R1      R0
```

8 instructions, 4 registres.

GenExp3 pour $a + (b - (c * (d/e)))$

```
LOAD    @c      R0
LOAD    @d      R1
DIV     @e      R1
MUL     R1      R0
LOAD    @b      R1
SUB     R0      R1
LOAD    @a      R0
ADD     R1      R0
```

8 instructions, 2 registres.

Bilan

Le résultat est parfois très bon, mais peut souvent être amélioré.
Divers types d'optimisation:

- bon usage des registres: éviter des load /store inuti
- sous-expressions communes: éviter des calculs redondants.

ex92 $(d * (e/f) - (g + h * i)) + (a - b * c)$

GenExp1: 19 instructions, 3 temporaires

```
LOAD    @b      R0
MUL     @c      R0
STORE   R0      1(EBP)
LOAD    @a      R0
SUB     1(EBP)  R0
STORE   R0      1(EBP)
LOAD    @h      R0
MUL     @i      R0
STORE   R0      2(EBP)
LOAD    @g      R0
ADD     2(EBP)  R0
STORE   R0      2(EBP)
LOAD    @e      R0
DIV     @f      R0
STORE   R0      3(EBP)
LOAD    @d      R0
MUL     3(EBP)  R0
SUB     2(EBP)  R0
ADD     1(EBP)  R0
```

GenExp3

```
LOAD    @d      R0
LOAD    @e      R1
DIV     @f      R1
MUL     R1      R0
LOAD    @g      R1
LOAD    @h      R2
MUL     @i      R2
ADD     R2      R1
SUB     R1      R0
LOAD    @a      R1
LOAD    @b      R2
MUL     @c      R2
SUB     R2      R1
ADD     R1      R0
```

– 14 instructions, 2 temporaires

Allocation des registres

Méthode d'optimisation globale, pour tout (ou partie de) programme.

Peut-on garder toutes les variables d'un programme da k registres?

Formalisation du problème: un graphe

- chaque variable du programme $\Rightarrow$ un sommet.
- COLLISION entre variables $\Rightarrow$ un arc.
- La DURÉE DE VIE d'une variable va de la 1ière affectation à la dernière utilisation.
- collision entre variables: leurs durées de vie ont intersection non nulle.

Exemple

```
t1=x;
t2=y;
t3=t1+t2;
t4=t1*t3;
```


Optimisation: élimination de sous-express communes

Coloriage d'un graphe

- k registres suffisent $\Leftrightarrow$ on peut colorier le graphe avec k couleurs.
- Coloriage d'un graphe avec k couleurs: la couleur donné à un sommet est différente de la couleur de chaque voisin.
- Il existe des algorithmes heuristique assez simples et efficaces, mais formellement, c'est un problème dur: il est NP-COMPLET.

Codage à partir d'un DAG

Comme nous l'avons fait en algorithmique pour les calc
redondants, nous représentons une expression par un
Graphe Orienté sans Circuit, appelé DAG (Directed Acy
Graph) Dans lequel chaque expression est représentée

par un seul sommet (y compris les identificateurs).
DAG de calcul.

Example:

```
G = C*(A+B)+(A+B);
C = A+B;
A = (C*D)+(E–F);
```

Algorithme

Après construction du DAG, algorithme en 2 étapes:

- ordonnancement des sommets (tri topologique)
- allocation de registres
- codage

Voir bibliographie, par exemple Fischer&Leblanc: Crafting a Compiler

Optimisations
Jusqu'où peut-on aller?

Exemple de code

Algo avec DAG: 12 instructions, 2 regs.

```
LOAD    @a      R1
ADD     @b      R1
LOAD    @c      R2
MUL     R1      R2
ADD     R1      R2
STORE   @g      R2
LOAD    @e      R2
SUB     @f      R2
STORE   @c      R1
MUL     @d      R1
ADD     @2      R1
STORE   @a      R1
```

GenExp1: 20 instructions, 2 temp (+1 reg)

GenExp3: 17 instructions, 2 registres

```
LOAD    @c      R0
 LOAD    @a      R1
 ADD     @b      R1
 MUL     R1      R0
 LOAD    @a      R1
 ADD     @b      R1
 ADD     R1      R0
 STORE   R0      @g
 LOAD    @a      R0
 ADD     @b      R0
 STORE   R0      @c
 LOAD    @c      R0
 MUL     @d      R0
 LOAD    @e      R1
 SUB     @f      R1
 ADD     R1      R0
 STORE   R0      @a
```


Exemple

```
{w=0;
  while(a>0||a>c)
   while(b<a)
      if(a>c)  if(c==b) w=c;  else w=a-1;
      else
         if(c>b||c*b>=0) w=a-b;
         else w=(a+b*c)*(a+b*c)-(a+b)*b*c;
 res=w*w;
}
```

Code généré en annexe.

Optimisations

Notre générateur élémentaire donne un résultat médiocre, qu'on peut améliorer de plusieurs façons:

- branchements
- gestion des registres
- sous-expressions communes

==> Cela mène loin, et conduit à des algorithmes très sophistiqués

Peephole Optimisations - "à la lucarne"

Nom donné à l'optimisation locale sur la liste d'instructi
en langage machine.

- branchement sur branchement
- branchements maladroits
- STORE R0 @a puis LOAD @a R0
- ...

La liste des possibilités est longue, mais attention: cha

étiquette "coupe" le contexte.

Optimisations

Ensemble de techniques très hétéroclites, dans différentes phases d'un compilateur

- Arbre Abstrait: transformations du code source
- Optimisations locales dans GenCode. Exemple: DAG
- Optimisations globales dans GenCode. Exemple: coloriage.
- Optimisations sur le code objet: "à la lucarne" - peephole optimisations

Optimisations de boucles - déroulement

Déroulement d'une boucle - loop unrolling

Exemple:

Boucle simple

```
for(i=0;i<3;i++)
  A[i]=i;
```

Traduction possible:

```
LEA   @A,R0 -- adresse de A[0]
            -- en LX, pas en C !!
LOAD  #0,R1
STORE R1,(R0)
LOAD  #1,R1
STORE R1,1(R0)
LOAD  #2,R1
STORE R1,2(R0)
```

Dans quelles conditions est-ce intéressant?

Optimisations de boucles (2)

Factorisation de code invariant, indépendent de la variable contrôlée.

Exemple: Algorithme de Warshall *Poly Algorithmique, & 8.3 et 8.5*

Boucle de base

```
for(k=0;k<N;k++)
  for(i=0;i<N;i++)
    for(j=0;j<N;j++)
      w[i][j]=w[i][j]
             |w[i][k]&w[k][j];
```

Avec factorisation

```
for(k=0;k<N;k++)
  for(i=0;i<N;i++)
    {tmp=w[i][k];
     for(j=0;j<N;j++)
       w[i][j]=w[i][j]|tmp&w[k][j];
  }
```

Dans quelles conditions est-ce possible?

Optimisations - Généralités

Cela peut paraitre facile, mais:l'optimisation est un très métier.

Exigeance minimale: une optimisation doit être:

- valide: ne jamais introduire d'erreur.
- efficace: toujours produire un code meilleur.

Ce minimum est très difficile à garantir!

⇒ les optimisations sont (souvent) en option d'un com

teur.

Optimisations de boucles (2)

En C, on peut aller plus loin:

Boucle de base

```
for(k=0;k<N;k++)
  for(i=0;i<N;i++)
    for(j=0;j<N;j++)
      w[i][j]=w[i][j]
             |w[i][k]&w[k][j];
```

Optimisation:

```
for(k=0;k<N;k++)
  for(i=0;i<N;i++)
    {tmp1=w[i][k];
     tmp2=w+i; /* w[i]==*(w+i) */
     for(j=0;j<N;j++)
       tmp2[j]=tmp2[j]|tmp1&w[k][j];
    }
```


Un piège: les alias

alias: expressions différentes qui désignent le même ob

C'est fatal pour le calcul de sous-expressions commune et l'optimisation des registres.

Exemple:

3 affectations:

```
A[i] = 3;

A[j] = A[i]+1;

B = A[i];
```

Traduction:

```
LEA   @A,R0
ADD   @i,R0   -- adresse de A[i]
LOAD  #3,R1
STORE R1,(R0) -- fin 1iere affect
LOAD  (R0),R1 -- valeur A[i]
LOAD  R1,R3   -- on sauve A[i]
ADD   #1,R1   -- A[i]+1
LEA   @A,R0
ADD   @j,R0   -- adresse de A[j]
STORE R1,(R0) -- fin 2ieme affeca
STORE R3, @B  --  ??
```

... si $i = j$??

Elimination de code mort

Que pensez-vous de:

```
if (a*a-b*b!=(b+a)*(a-b))  a=func(a,b);
```

Peut-on éliminer du “code mort” dans cet exemple?

Décidabilité

Le problème de calcul formel:
Une expression est-elle identique à zéro?
est indécidable.

On peut savoir que $(a+b)*(a-b)-(a*a-b*b)=0$ ma
n’y a pas de méthode générale pour ce genre de décisi
Donc: pas de méthode générale pour détecter du code
inutile.

Décidabilité et NP-complétude

L’optimisation de code est un problème mal cerné, et rejoint le problème de programmation en général.

Il y a donc forcement des limites très strictes aux traitements systématiques: en programmation, beaucoup de problèmes (apparemment) simples sont

- indécidables
- ou algorithmiquement complexes

NP-complétude

La classe de Problèmes NP-complets
- étudiée en algorithmique 2 à l’Ensimag -
est une vaste classe de problèmes pour lesquels tous l
algorithmes connus sont de coût exponentiel. La soluti
est donc possible seulement pour des problèmes de pe
taille.
Exemples de problèmes NP-complet

- Ordonnancement
- Sac à dos
- Formule logique est une tautologie
- Coloriage des sommets d’un graphe avec K couleu

Problème NP-complet (1)

Combien faut-il de temporaires au minimum?

Etant donné

- un DAG de calcul
- en entier K

Peut-on compiler le calcul des racines de G en utilisant au plus K registres?

Problème NP-complet (2)

Géneration de code pour une machine avec 1 accumulateur.

On considère une machine à 1 seul registre possédant des instructions `LOAD mem`, `STORE mem`, et des opérations `OP mem`.
Etant donné

- un DAG de calcul
- en entier K

Peut-on compiler le calcul de la racine de G avec au plus K instructions?

Problème NP-complet (3)

Géneration de code avec nombre de registres illimité.

En utilisant seulement des instructions `OP ri,rj` ri et `LOAD ri,rj`, et avec un DAG binaire

peut-on calculer les racines de G en K instructions au p

Conclusions

- Il existe d'innombrables techniques d'optimisation.
- Mais aucun compilateur ne peut garantir l'optimalité code produit, pour tous les programmes.
- On pourra toujours inventer des nouvelles optimisations, notamment pour des cas particuliers

Langages objets :
Principe de définition et de compilation

1) Typage / héritage et redéfinition

Objet :
- ses attributs
- ses méthodes (jeu d'opérations)

La classe est un type.
Héritage : transmission des propriétés d'un objet :
- Les attributs et des extensions
- Les méthodes et des extensions ainsi que des redéfinitions

Si B hérite de A alors B est un sous-type de A.
Principe de substitution associé au sous-typage.
X : B et B sous-type de A alors X : A

Permet de substituer une expression de type A par une expression de type B.

Exemple :
```
class rectangle {
        double largeur, hauteur;
        double perimetre( ) { return 2*(largeur+hauteur) }
}

class carre extends rectangle {
        double perimetre ( ) { return largeur * 4 }
}
```

carré est un sous-type de rectangle.

Principe de substitution :
```
r : rectangle;
c : carré;
r := c; // OK
c := r; // INTERDIT
```

Applicable uniquement en position expression (pas en variable)
- paramètre IN
- résultat de fonction
- partie droite d'affectation

Exemple de hiérarchie java :

```
Object
   |
Rectangle
   |
Carre
   |
{null}
```

Le sous-type réduit à l'élément *null* est sous-type de toute classe.
Le sous-type réduit à l'élément null est sous type de toute classe.
Aspect objet : on distingue le typage statique du typage dynamique.

- Lié aux déclarations
- Déterminer quelles méthodes sont applicables

- Déterminé à l'exécution
- Lié au flot de contrôle
- Choix précis des méthodes à appliquer

Exemple :
r : rectangle;
c : carré;

r := c ; // TypeStatique(r) : rectangle – TypeDynamique(r) : carré

c := r ; // mal-typé : refusé vis à vis du typage statique

c := (carré) r; // bien typé, test à l'exécution TD(r) : si carré OK, sinon exception.

Choix des méthodes : liaison statique (C++)/ liaison dynamique(C++ virtual, Java).

Principe de redéfinition des méthodes par héritage :
Soit une classe A avec une méthode m de profil
m : A x B -> C
A : type de l'objet sur laquelle on applique la méthode.
B : type des arguments

Ex : périmètre : rectangle --> double
Soit A' une sous-classe de A (A' extends A)
Comment est redéfinie ma méthode m ?
m' : A' x B -> C
=> Principe de base : on fait varier le type de l'objet sur lequel s'applique m.
Que se passe-t-il si C=A ? B=A ?

Deux principes :
– Co-variance : On peut remplacer A par A'.
– Invariance : Les types ne changent pas.

Java : Invariant sur les arguments, co-variant sur les résultats de fonction (à partir de java 1.5).
Ada : co-variant sur le résultat des fonctions et les arguments.

Intérêt de la co-variance sur les résultats.
Copy( ) return rectangle;
de profil rectangle -> rectangle

Par redéfinition on obtient,
copy : carré -> carré

Intérêt de la co-variance sur les paramètres :
egal (r2 : rectangle) return bool
de profil rectangle x rectangle -> bool

en Java :
egal : carre x rectangle -> bool
en Ada :
egal : carre x carre -> bool

Problème en Ada : « trou de typage »
On ne peut pas trouver à l'exécution de méthode à appliquer :
egal(c, r) ? avec c : carré et r : rectangle

Java : une propriété forte
Si un programme est bien typé alors on saura toujours choisir la méthode à appliquer à l'exécution.

2) Représentation des objets en mémoire et héritage

Objet :
– Des attributs
– Un ensemble de méthodes applicables sur cet objet

Type = une table des méthodes
C'est à dire quel code est associé à quelle méthode ?

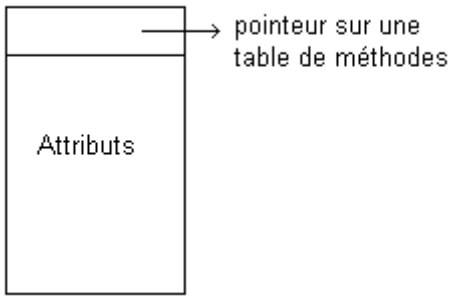
VTM : virtual method table

```
class A { double a,b;
m ( ) return A { code1 }
}
class B extends A { double c;
m ( ) return B { code2 }
h ( ) return int { code3 }
}
```

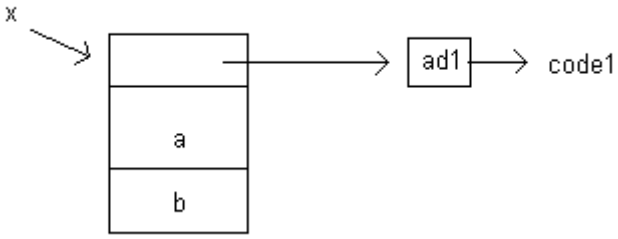
Table de A :
m : ad1 -> code1
m garde le même emplacement dans Table de A et dans Table de B.

Table de B :
h : ad3 -> code3
m : ad2 -> code2
o.m : Table de A si TD(o) = A , Table de B si TD(b) = B

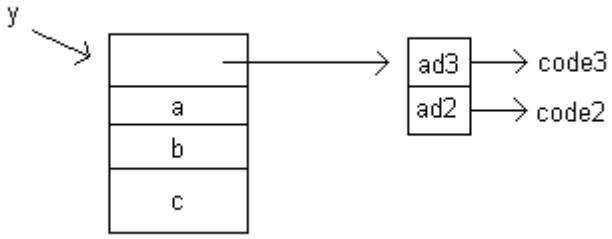
Représentation des objets.
Zone mémoire :



Exemple :
Après la déclaration de x : A on a



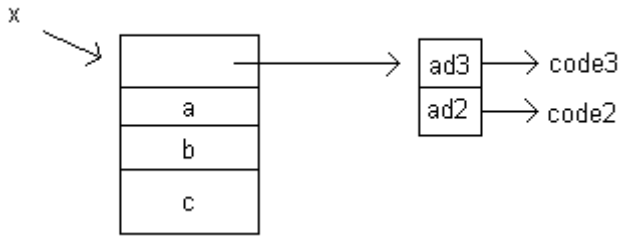
Si on déclare y : B



Codage de l'appel o.m ?
m a toujours le même déplacement dans les tables de chaque sous-classe.
Si l'adresse de l'objet dans R1

LOAD (R1) R1 # l'adresse de la table des méthodes
ADD depl(m) R1 # Case de m dans la table des méthodes
BRA (R1) # Branchement ou code

Codage de l'affectation
x := y ;
Affectation d'adresse, x pointe sur l'objet y



x.m( );

Le code de l'appel exécute un branchement à ad2.

Détermine le type et un objet
C++ : typeid retourne le type dynamique d'un objet
type = adresse de sa table des méthodes

OK pour héritage simple. Beaucoup plus complexe en héritage multiple.

Optimisation, éviter dans la mesure du possible les indirections liées aux appels de méthodes (voir TD).

Mémoire à l'exécution

1) Allocation statique / Allocation dynamique

Modèle à l'exécution mémoire.
Mem : Nom -> Valeur
découpée en deux.

Env : Nom -> Adresse *// fait à la compilation*
Etat : Adresse -> Valeur *// à l'exécution*

Mem = Etat ¤ Env

Allocation statique : durée de vie des objets connue à la compilation.

Allocation dynamique : durée de vie programmée.

Exemple : allocation statique
Objets alloués à l'appel et libéré en fin d'appel.

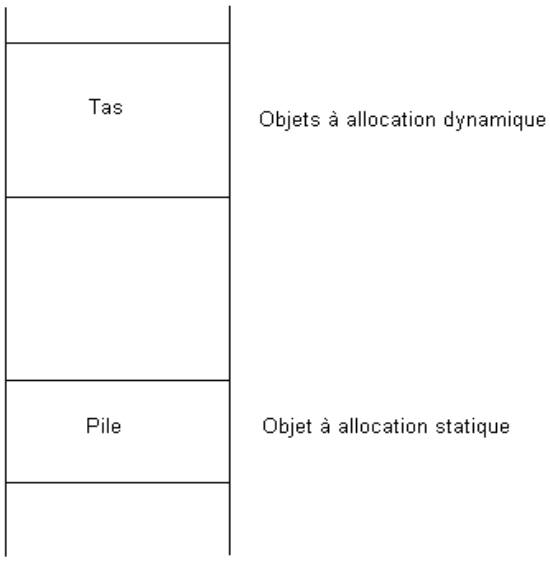
```
Procedure p is
        x : integer;
        a : array(1..3) of integer;
        n : integer;
begin
        ...
end p;
```

Exemple : allocation dynamique

```
procedure q is
        x : access integer;
begin
        x := new (INTEGER);
        x.all = 3;
end
```

L'objet new INTEGER est créé et détruit par instruction.

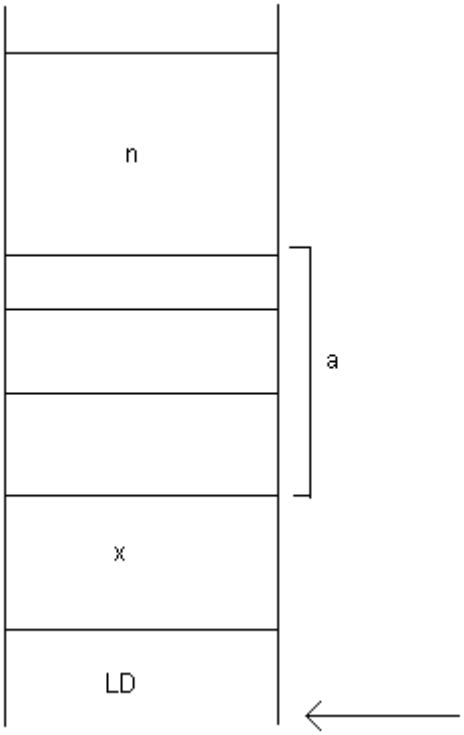
Représentation mémoire :



2) Bloc d'activation

A chaque procédure on associe un bloc d'activation : la forme du bloc mémoire à allouer pour un appel à allouer cette procédure.

Exemple :
On suppose que les entiers sont codés sur un mot.



Les objets sont adressés de manière relative par rapport au début du bloc d'activation
déplacement(x) = 1
déplacement(a) = 2
déplacement(n) = 5

ld : Lien dynamique. Pointeur sur le bloc d'activation précédent dans la pile.

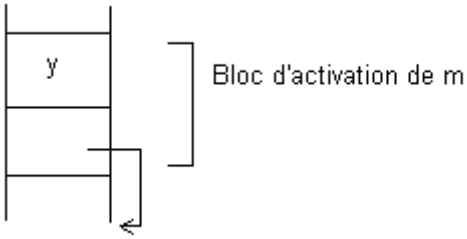
A l'appel de la procédure p :
– On empile une occurrence du bloc d'activation associé à p.
– On fait pointer le lien dynamique sur le bloc précédent.

A la fin de la procédure :
– On dépile le bloc d'activation courant
– On rétablit le lien dynamique

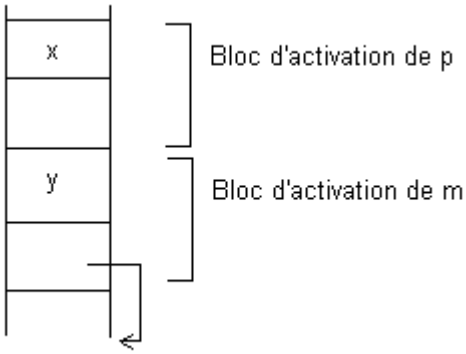
Exemple :

```
procedure p is
        x: integer;
begin
        ...
        p;
        ...
end p;dd
```

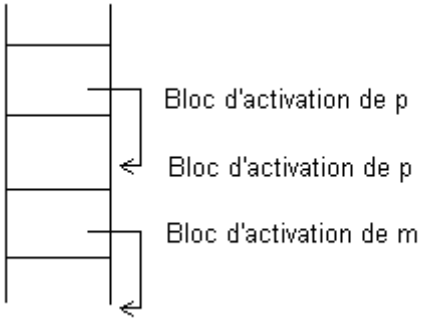
Appel de m :



Appel de p dans m :



Appel de p dans p :



3) Gestion du tas

new type :
ad <- allouer_tas(taille(type))
s'il reste de la mémoire
exception sinon
dispose var :
ad <- adresse(var);
liberer_tas(ad);

Éviter la fragmentation mémoire (des trous de taille non exploitable).
Minimiser la taille mémoire globale.

Algorithme d'allocation :
- first-fit : le 1er emplacement libre
- circular-first-fit : le premier libre à partir d'un pointeur courant
- best-fit : minimise les trous

En compilation :
On connaît statiquement la taille des objets à allouer.

On peut découper le tas en sous-zone :
– Un tas par taille
type t = access t' ;
On peut calculer taille(t') = n

Libération mémoire :
Langage à allocation explicite : (C, C++)
-> instruction dispose
Langage à allocation implicite : (Java)
-> fait par le compilateur
Problème de libération explicite : source d'erreurs importantes
– fuite mémoire : on oublie de libérer
– référence pendante

Exemple de fuite mémoire :

```
procedure p is
        x : access integer;
begin
        x := new (integer);
        x.all = 3;
end p;
```

en fin de p :
x n'existe plus, on a plus de référence sur la case contenant 3.

Exemple de référence pendante :

```
y : access integer;
procedure p is
        x : access integer;
begin
        x := new(integer);
        y.all = 3;
        y := x;
        unchecked_deallocation(x,integer);
end p;
```

y pointe sur une case mémoire qui a été libérée.

– Approche compteur de référence (ex : SMART pointeur de C++)
  – fait à la volée

A chaque objet, on associe :
o.compteur : nombre d'objets accédant à o
o.valeur : valeur de l'objet

On gère les compteurs à l'exécution.
Dès que le compteur passe à 0, on libère la mémoire.
Code pour
x := y;
Algorithme :
  – On décrémente le compteur de l'objet x.all
  – On coupe une référence
  – si 0 libération
  – on fait l'affectation
  – on incrémente le compteur associé à y

Avantages :
– Libération à la volée
– Facile à implémenter

Inconvénient :
– Prend de la place mémoire (un compteur par objet)
– Ne détecte pas tout (pointeur sur lui-même)

- Approche ramasse-miette (garbage collector)
  - appelé de temps en temps

Algorithme :

- On marque chaque case du tas comme libre
- On parcourt tous les objets de la pile et on marque récursivement les objets accessibles à partir de la pile. (marqué = à garder)

On doit pouvoir parcourir les objets et leurs sous-objets.
Problème : fragmentation mémoire
Solution : deux tas. Lorsqu'on récupère la mémoire, on recopie en recompactant la mémoire