Traduction et Sémantique
Introduction

Luigi Santocanale
LIF, Université de Provence
Marseille, FRANCE

12 février 2010

Plan

Préambule

Contexte, historique

LSE : un Langage Simple d'Expressions
- Le langage source
- Une machine virutelle
- La traduction
- Des questions sémantiques
- Évaluation
- Typage

Plan

Préambule

Contexte, historique

LSE : un Langage Simple d'Expressions
- Le langage source
- Une machine virutelle
- La traduction
- Des questions sémantiques
- Évaluation
- Typage

Des coordonnées

- Page web :
  `http://www.lif.univ-mrs.fr/~lsantoca/teaching/TS/`
- Mon courriel : `lsantoca@lif.univ-mrs.fr`
- Formule de la note finale :

  ```
  NF=(2 max (E, (2E+P) / 3) + TP) / 3
  ```

  où
  - `E` est la note (rattrapable) de l'examen,
  - `P` est la note (non rattrapable) du partiel,
  - `TP` est la note (non rattrapable) du projet à elaborer en TP. Présence à la soutenace obligatoire.

D'autres infos

- Partiel, autour du 7ème cours.
- Projet : à la 6ème séance de TP.
  (Présence à la soutenance obligatoire.)

Bibliographie :


Roberto M. Amadio.
Introduction à l'analyse syntaxique et à la compilation, 04 2009.


Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman.
Compilers : Princiles, Techniques, and Tools.
Addison-Wesley, 1986.


J. Levine and D. Mason, T. Brown.
lex and yacc.
O'Reilly, 1992.

Plan

Préambule

Contexte, historique

LSE : un Langage Simple d'Expressions
- Le langage source
- Une machine virutelle
- La traduction
- Des questions sémantiques
- Évaluation
- Typage

Compilation

Compiler :

- Littéraire :
  mettre ensemble.
  Fait par l'éditeur de liens.
- En Info : processus de *traduction*
  d'un langage source (de haut niveau)
  vers un langage cible, (de bas niveau, lisible par la machine).
- Diffèrent de l'*interprétation*,
  traduction et exécution à la volée.
  Voir les langages de scripts.
- Critère de correction de la traduction :
  notion de *sémantique*.

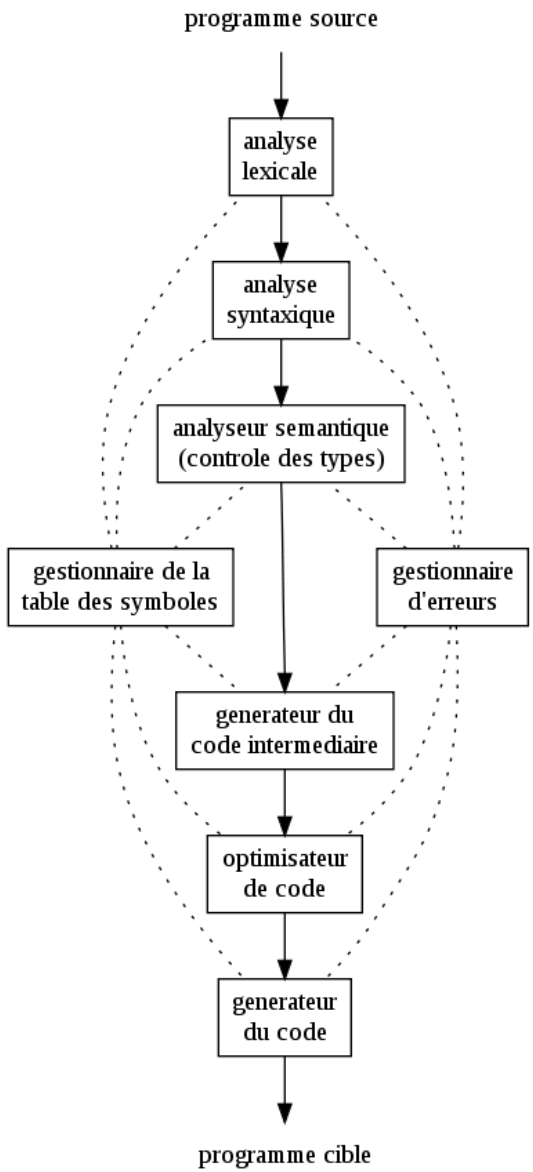
Objectifs

- Comprendre ce qui ce passe dans le processus de compilation.
- Apprendre les techniques et les algorithmes de ce processus.
- Maîtriser les outils (Lex, Yacc) qui nous permettent
  de construire des compilateurs.
- Apprécier ces techniques et ces outils,
  en dehors de leur contexte d'origine,
  pour des nouvelle problématiques.

D'autres applications

- Multitude de formats, transformations entre eux . . .
  . . . voir les exercices.
- Des exemples :
  - langages de composition de textes :
    tex, latex, bibtex, . . .
  - langages graphiques :
    dot, dot2tex, . . .
  - xml, html, & co :
    nsgmls, . . .
- Typage pour la sûreté

Structure d'un compilateur



Analyse lexicale

- Processus qui transforme (reconnaît)

  un flot de caractères vers une suite d'unités logiques (unités lexicales).

- Exemple :

$$\underbrace{\text{je}}_{\text{pronom}}\ \underbrace{\text{mange}}_{\text{verbe}}\ \underbrace{\text{la}}_{\text{article}}\ \underbrace{\text{pomme}}_{\text{nom}}$$

  consiste de 16 caractères et 4 mots.
  On peut associer à chaque mot une catégorie.

- Cadre théorique :
  - expressions et langages régulièr(e)s,
  - automates à états finis.

Analyse lexicale

- Processus qui transforme (reconnaît)

  un flot de caractères vers une suite d'unités logiques (unités lexicales).

- Exemple :

  $$\underbrace{\text{je}}_{\text{pronom}}\ \underbrace{\text{mange}}_{\text{verbe}}\ \underbrace{\text{la}}_{\text{article}}\ \underbrace{\text{pomme}}_{\text{nom}}$$

  consiste de 16 caractères et 4 mots.
  On peut associer à chaque mot une catégorie.

- Cadre théorique :
  - expressions et langages régulièr(e)s,
  - automates à états finis.

Analyse lexicale

- Processus qui transforme (reconnaît)

  un flot de caractères vers une suite d'unités logiques (unités lexicales).

- Exemple :

  $$\underbrace{\text{je}}_{\text{pronom}}\ \underbrace{\text{mange}}_{\text{verbe}}\ \underbrace{\text{la}}_{\text{article}}\ \underbrace{\text{pomme}}_{\text{nom}}$$

  consiste de 16 caractères et 4 mots.
  On peut associer à chaque mot une catégorie.

- Cadre théorique :
  - expressions et langages régulièr(e)s,
  - automates à états finis.

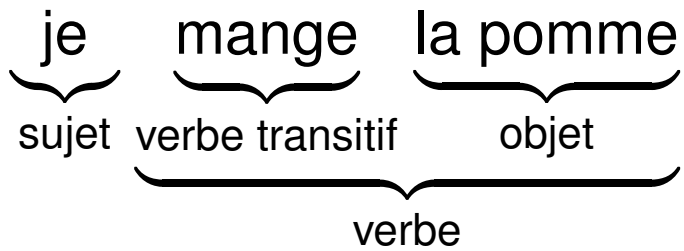
Analyse syntaxique

- Processus qui décerne les flots d'unités lexicales porteur de sens.
- Exemple :

  est un phrase, une phrase est constituée par un sujet et un verbe.
- En compilation :
  processus qui décerne la structure algébrique (articulation logique) d'un programme à partir d'un flot d'unités lexicales.
- Une première signification à un programme.
- Cadre théorique :
  - grammaires non contextuelles, langages algébriques,
  - automates à pile.

Analyse syntaxique

- Processus qui décerne les flots d'unités lexicales porteur de sens.
- Exemple :

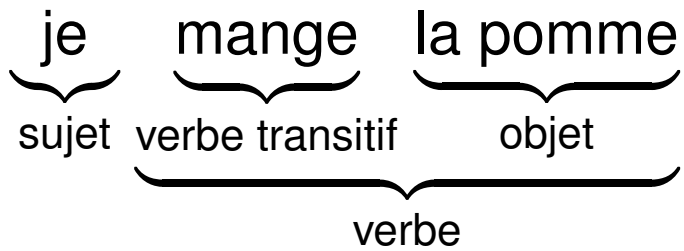


est un phrase, une phrase est constituée par un sujet et un verbe.

- En compilation :
  processus qui décerne la structure algébrique (articulation logique) d'un programme à partir d'un flot d'unités lexicales.
- Une première signification à un programme.
- Cadre théorique :
  - grammaires non contextuelles, langages algébriques,
  - automates à pile.

Analyse syntaxique

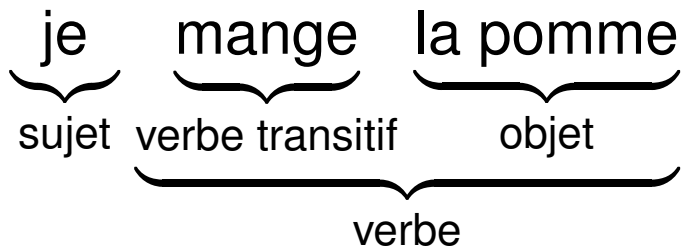
- Processus qui décerne les flots d'unités lexicales porteur de sens.
- Exemple :



  est un phrase, une phrase est constituée par un sujet et un verbe.
- En compilation :
  processus qui décerne la structure algébrique (articulation logique) d'un programme à partir d'un flot d'unités lexicales.
- Une première signification à un programme.
- Cadre théorique :
  - grammaires non contextuelles, langages algébriques,
  - automates à pile.

Analyse syntaxique

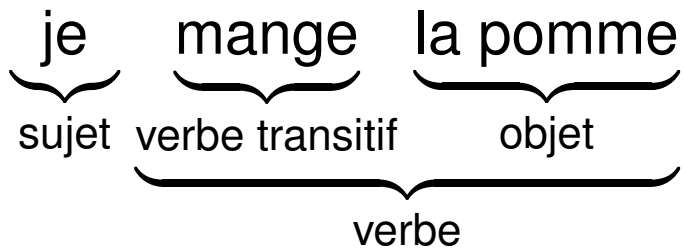
- Processus qui décerne les flots d'unités lexicales porteur de sens.
- Exemple :



  est un phrase, une phrase est constituée par un sujet et un verbe.
- En compilation :
  processus qui décerne la structure algébrique (articulation logique) d'un programme à partir d'un flot d'unités lexicales.
- Une première signification à un programme.
- Cadre théorique :
  - grammaires non contextuelles, langages algébriques,
  - automates à pile.

Analyse syntaxique

- Processus qui décerne les flots d'unités lexicales porteur de sens.
- Exemple :



  est un phrase, une phrase est constituée par un sujet et un verbe.
- En compilation :
  processus qui décerne la structure algébrique (articulation logique) d'un programme à partir d'un flot d'unités lexicales.
- Une première signification à un programme.
- Cadre théorique :
  - grammaires non contextuelles, langages algébriques,
  - automates à pile.

De la linguistique à l'informatique


Noam Chomsky.
Three models for the description of language.
IRE Trans. Inform. Theory, (2 :3) :113–124, 1956.

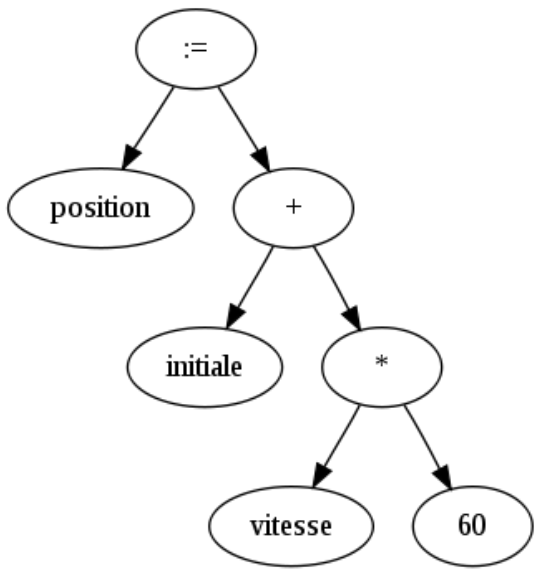

Noam Chomsky.
On certain formal properties of grammars.
Information and Control, 2 :137–167, 1959.


Peter Naur.
ALGOL—the international language for description of logical and numerical processes.
Nordisk Mat. Tidskr., 8 :117–129, 144, 1960.

Analyse sémantique

Typiquement il s'agit de la phase de *contrôle de types*.

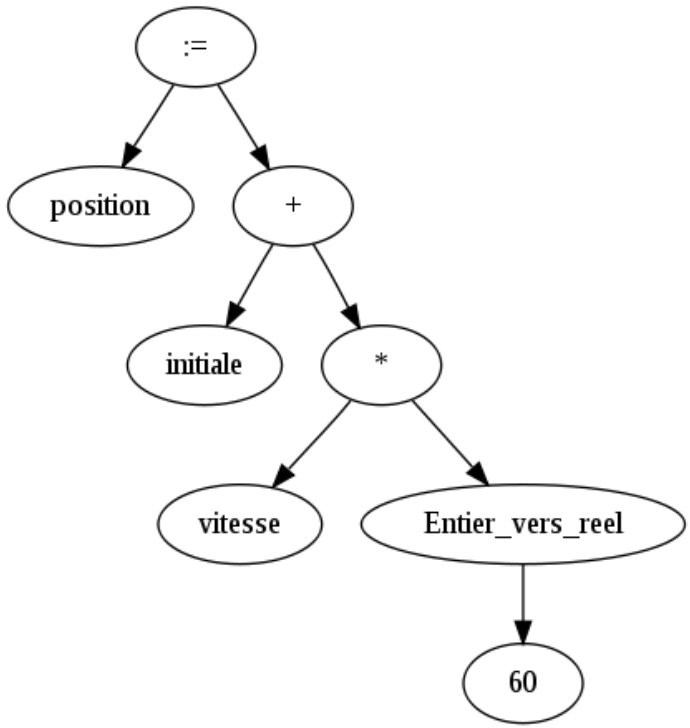
L'expression



peut être jugé incorrecte ou . . .

Transtypage

. . . être tranformée en



Génération du code intermédiaire

- Représentation du programme dans un langage indépendant de l'architecture.
- Il s'agit, en général, d'un langage pour une machine abstraite.
- Code à trois adresses :

```
temp1 := EntiertVersReel(60)
temp2 := id3 * temp1
temp3 := id2 + temp2
id2   := temp3
```

Optimisation

Un exemple :

```
/* temp1 := EntiertVersReel(60) */
temp2 := id3 * 60.0
/* x temp3 := id2 + temp2 */
id2   := id2 + temp2
```

Génération de code

- Traduction dans le langage cible, lisible par une architecture donnée (CISC et x86, RISC et Sun, ...).
- Assignation des variables aux registres.

Ainsi,

```
position := intiale + vitesse * 60
```

devient :

```
11:position.c    ****   position=initiale + vitesse* 60;
34                              .loc 1 11 0
35 001c D945F4                  flds    -12(%ebp)
36 001f D9050000                flds    .LC0
36      0000
37 0025 DEC9                    fmulp   %st, %st(1)
38 0027 D845F8                  fadds   -8(%ebp)
39 002a D95DFC                  fstps   -4(%ebp)
```

Plan

Préambule

Contexte, historique

LSE : un Langage Simple d'Expressions
- Le langage source
- Une machine virutelle
- La traduction
- Des questions sémantiques
- Évaluation
- Typage

Le langage

- constantes numériques :

```
1, 03457
```

- constantes booléennes :

```
true, false
```

- opérateurs :

```
+, *, ...
```

- variables :

```
x, y, z, s46uu,...
```

- déclarations :

```
let ... = ... in ...
```

- contrôle du flot :

```
if ... then ... else ...
```

Analyse lexicale

La chaîne de caractères

```
let x7 = 3 in (x7 + 4)
```

est analisée comme suit :

Chaîne (lexème) :	`let`	`x7`	`=`	`3`	`in`	`(`	`x7`	`+`	`4`	`)`
Unité lexicale :	**let**	**id**	**eq**	**cnst**	**in**	**lpar**	**id**	**plus**	**cnst**	**rpar**
Valeur (attribut) :		*x7*		*3*			*x7*		*4*	

La définition du langage

. . . se fait par une grammaire non contextuelle

$$\mathscr{G} = \langle V, \Sigma, S, P \rangle$$

où

$$V = \{ S, OP, E, B \} \cup \{ plus, prod, \ldots, id, const, lpar, rpar, if, then, else, let, eq, in \}$$

et P est

$$OP \rightarrow plus \mid prod \mid \ldots$$
$$E \rightarrow id \mid const \mid E\ OP\ E \mid lpar\ E\ rpar$$
$$B \rightarrow E \mid if\ E\ then\ B\ else\ B \mid let\ id\ eq\ E\ in\ B \mid lpar\ B\ rpar$$
$$S \rightarrow B$$

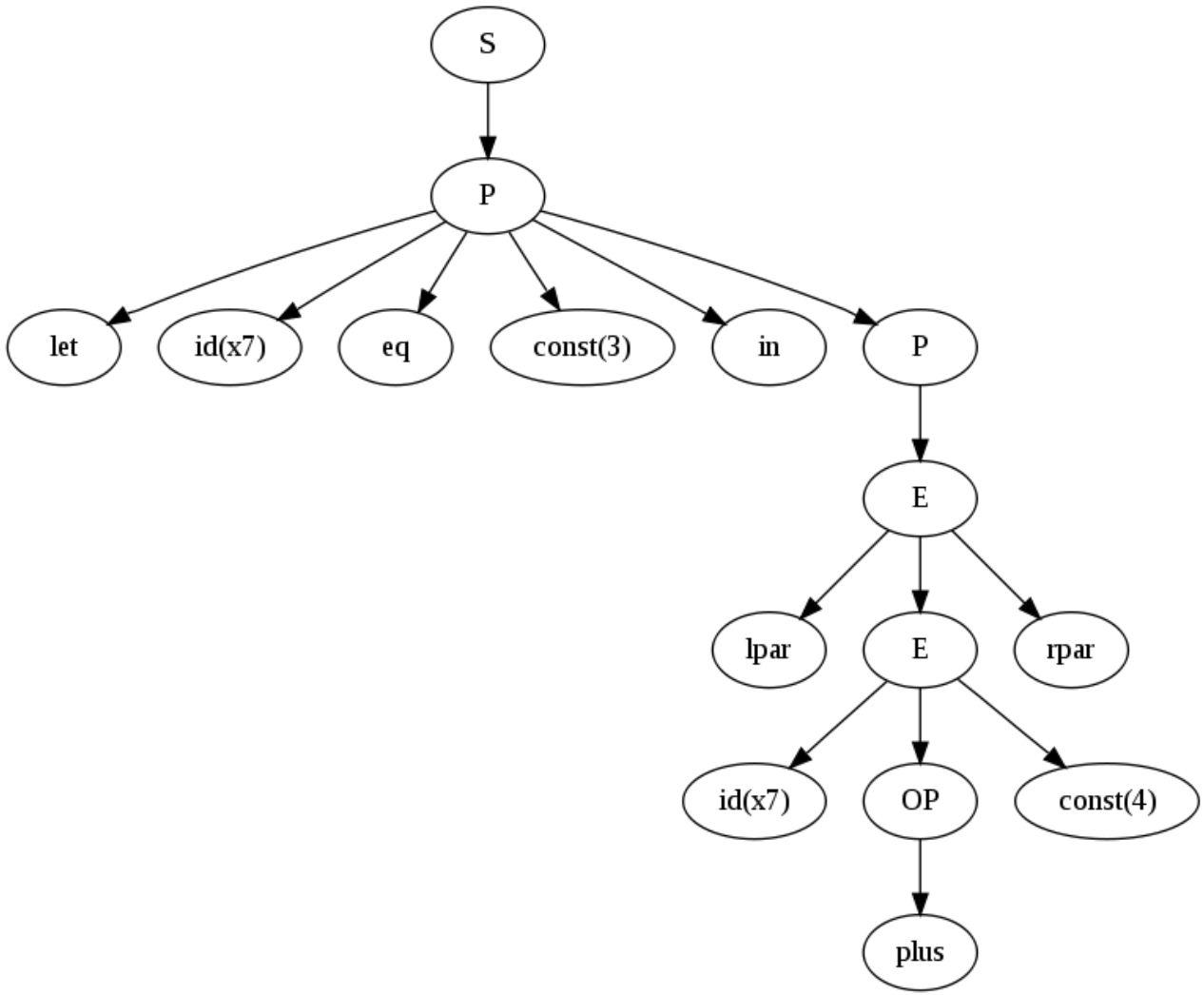
Dérivation (gauche)

La chaîne d'unités lexicales appartient au langage de $\mathscr{G}$,
car il existe une dérivation du symbole intial S vers la chaîne :

$$
\begin{aligned}
S &\Rightarrow B \\
&\Rightarrow \textit{let id eq E in B} \\
&\Rightarrow \textit{let id eq const in B} \\
&\Rightarrow \textit{let id eq const in E} \\
&\Rightarrow \textit{let id eq const in lpar E rpar} \\
&\Rightarrow \textit{let id eq const in lpar E OP E rpar} \\
&\Rightarrow \textit{let id eq const in lpar id OP E rpar} \\
&\Rightarrow \textit{let id eq const in lpar id plus E rpar} \\
&\Rightarrow \textit{let id eq const in lpar id plus cnst rpar}
\end{aligned}
$$

Arbre de dérivation

L'appartenance au langage de $\mathscr{G}$ est aussi témoigné par un arbre de dérivation :



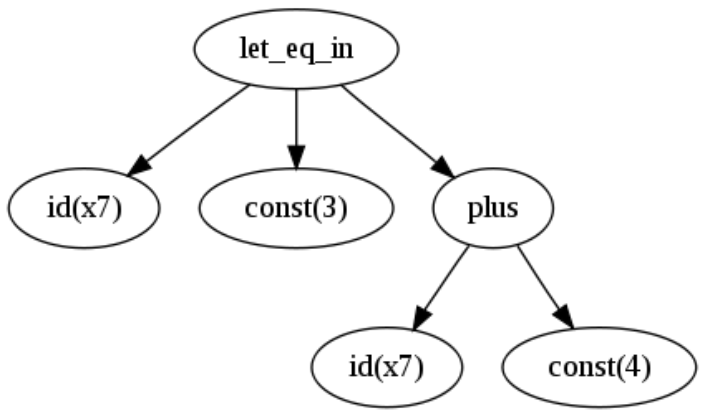
Syntaxe abstraite

A partir de l'arbre de dérivation on construit sa syntaxe abstraite.

Notre exemple donne :

$$\mathit{let\_in\_eq}(\ \langle \mathit{id}, x7\rangle\ ,\ \langle \mathit{const}, 3\rangle\ ,\ \mathit{plus}(\ \langle \mathit{id}, x7\rangle\ ,\ \langle \mathit{const}, 4\rangle\ )\ )$$

ou bien



Considération philosophiques sur la syntaxe abstraite

- Un programme est un terme – comme ceux de la logique.
- On peut représenter un terme comme un arbre étiqueté. Cela permet de se débarrasser de symboles non intéressants (parenthèses, virgules).
- On traite la construction `let...eq...in...` comme une unité, un symbole de fonction de arité 3.

- La structure mathématique des arbres est inductive :
  - on peut définir des fonctions par induction sur la structure,
  - on peut définir des fonctions par induction sur la syntaxe abstraite.

Définition de la syntaxe abstraite

Il s'agit d'arbres étiquetés.

Un *E*-terme :

étiquette	infos	enfants
id	chaine de caractères	0
const	nombre, `true`, `false`	0
plus		2 *E*-termes
prod		

Un *B*-terme : un *E*-terme, ou bien

étiquette	infos	enfants
let_eq_in		2 *E*-termes, 1 *B*-terme
if_then_else_		1 *E*-terme, 2 *B*-termes

Considération moins philosophiques sur la syntaxe abstraite

- Implantation de la syntaxe abstraite :
  - en C, par les pointeurs (arbres = généralisation des listes)
  - en d'autres langages par les types induticfs.
    En OCaml :

```
type valeur = Int of int | True | False
type eterm = Id of string | Const of valeur
           | Plus of eterm * eterm
type bterm = Expr of expr
           | Ifthenelse of eterm * bterm * bterm
           | Leteqin of string * eterm * eterm
```

- Construire explicitement la syntaxte abstraite peut être couteux (en espace).
  Si l'on peut, on laisse cette construction implicite.

Une machine virtuelle

Une structure de donnés : $\langle M, pc, sp \rangle$, où

- M : tableau en mémoire,
- pc : program counter, compteur d'instruction,
- sp : stack pointer, pointeur au sommet de la pile.

Jeu d'instructions

`buildv`	sp++ ; M[sp] := v ; pc++ ;
`branchj`	si M[sp] = `true` alors pc++ sinon pc :=j ; sp–
`loadi`	sp++ ; M[sp] := M[i] ; pc++
`add`	sp++ ; M[sp] := M[sp] + M[sp +1] ; pc++
`return`	pc :=0 ; M[0] := M[sp] ; sp :=0

Fonction de traduction

Fonction de traduction,

définie par induction sur la syntaxe abstraite.

$$\mathscr{C}_E(x, w) = \texttt{load}\, i(x, w)$$
$$\mathscr{C}_E(v, w) = \texttt{build}\, v$$
$$\mathscr{C}_E(e_1 + e_2, w) = \mathscr{C}_E(e_1, w) \cdot \mathscr{C}_E(e_2, w) \cdot \texttt{add}$$

$$\mathscr{C}_B(x, w) = (\texttt{load}\, i(x, w)) \cdot \texttt{return}$$
$$\mathscr{C}_B(v, w) = (\texttt{build}\, v) \cdot \texttt{return}$$
$$\mathscr{C}_B(e_1 + e_2, w) = \mathscr{C}_E(e_1, w) \cdot \mathscr{C}_E(e_2, w) \cdot \texttt{add} \cdot \texttt{return}$$
$$\mathscr{C}_B(\mathit{let}\, x = e\, \mathit{in}\, p, w) = \mathscr{C}_E(e, w) \cdot \mathscr{C}_B(b, w \cdot x)$$
$$\mathscr{C}_B(\mathit{if}\, e\, \mathit{then}\, b_1\, \mathit{else}\, b_2, w) = \mathscr{C}_E(e, w) \cdot (\texttt{branch}\, \kappa) \cdot \mathscr{C}_B(b_1, w) \cdot \kappa : \mathscr{C}_B(b_2, w)$$

Résultat ;-)

Compiler

$$\begin{array}{l} \textit{let } x = 3 \textit{ in} \\ \quad \textit{let } y = x + x \textit{ in} \\ \quad\quad \textit{let } x = \textit{true in} \\ \quad\quad\quad \textit{if } x \textit{ then } y \textit{ else } x \end{array}$$

donne

```
1  :      build   3
2  :      load    1
3  :      load    1
4  :      add
5  :      build   true
6  :      load    3
7  :      branch  k
8  :      load    2
9  :      return
k=10 :    load    3
11 :      return
```

Des questions sémantiques

- Que veut dire une expression telle que

  ```
  if x then 0 else 1
  ```

  ?
- Est ce que le programme

  ```
  if 33 then 1 else 1
  ```

  est correcte ?
- Est ce que la traduction est correcte ?

Évaluation (sémantique opérationnelle)

Entre les expressions, on dit que

```
true, false, cnst(0), cnst(1), ...
```

sont des valeurs.

On peut définir une relation

$$e \Downarrow v$$

par induction sur la syntaxe abstraite. À lire :

l'expression e s'évalue au valeur v.

Définition de $\Downarrow$

$$\frac{}{v \Downarrow v} \qquad \frac{e_1 \Downarrow n_1 \in \mathbb{N} \quad e_2 \Downarrow n_2 \in \mathbb{N}}{e_1 + e_2 \Downarrow n_1 + n_2}$$

$$\frac{e \Downarrow \texttt{true} \quad p_1 \Downarrow v}{\mathit{if}\ e\ \mathit{then}\ p_1\ \mathit{else}\ p_2 \Downarrow v} \qquad \frac{e \Downarrow \texttt{false} \quad p_2 \Downarrow v}{\mathit{if}\ e\ \mathit{then}\ p_1\ \mathit{else}\ p_2 \Downarrow v}$$

$$\frac{e \Downarrow v' \quad [v'/x]p \Downarrow v}{\mathit{let}\ x = e\ \mathit{in}\ p \Downarrow v}$$

Notre exemple

L'expression $\mathit{let}\ x\ =\ 3\ \mathit{in}\ (x+4)$ s'évalue à 7 :

$$\mathit{let}\ x\ =\ 3\ \mathit{in}\ (x+4) \Downarrow 7$$

car on peut construire un preuve de cette relation :

$$\dfrac{\dfrac{}{3 \Downarrow 3} \qquad \dfrac{\dfrac{}{3 \Downarrow 3} \qquad \dfrac{}{4 \Downarrow 4}}{3+4 \Downarrow 7}}{\mathit{let}\ x\ =\ 3\ \mathit{in}\ (x+4) \Downarrow 7}$$

Considérations

- Prouver que $e \Downarrow v$ « revient à calculer » la valeur de e.
- La relation $\Downarrow$ permet de donner une signification aux expressions,
  indépendamment de ce que fait la machine virtuelle.
- Nous pouvons poser formellement la question de la *correction de la traduction* :

  Si $e \Downarrow v$ *et la machine exécute le code* $\mathscr{C}_B(e, [])$*,*
  alors la machine s'arrête et $M[0]$ *contient* v*.*

  Est ce que cette proposition est vraie ?
- La relation $\Downarrow$ est un (important) outil mathématique.
  Difficilement on peut s'en servir dans le cadre de l'implémentation d'un compilateur.

Typage

On souhaite définir une relation de la forme

$$e : \tau$$

où $\tau \in \{ \mathit{bool}, \mathit{nat} \}$, à lire :

l'expression e est de type τ ,

de façon que :

1. le fait suivant soit vrai :

$$e : \tau \text{ implique } \exists v.\ v : \tau \text{ et } e \Downarrow v ,$$

2. on puisse répondre à la question si $e : \tau$ de façon efficace.

Un premier essai

$$\frac{n \in \mathbb{N}}{n : nat} \qquad \frac{v \in \{ true, false \}}{v : bool}$$

$$\frac{e_1 : nat \quad e_2 : nat}{e_1 + e_2 : nat} \qquad \frac{e : bool \quad b_1 : \tau \quad b_2 : \tau}{if\ e\ then\ b_1\ else\ b_2 : \tau}$$

$$\frac{e : \tau' \qquad b : \tau}{let\ x = e\ in\ b}$$

Nous devons aussi ajouter des contraintes de types sur les variables ! ! !

Un premier essai

$$\frac{n \in \mathbb{N}}{n : nat} \qquad \frac{v \in \{ true, false \}}{v : bool}$$

$$\frac{e_1 : nat \quad e_2 : nat}{e_1 + e_2 : nat} \qquad \frac{e : bool \quad b_1 : \tau \quad b_2 : \tau}{if\ e\ then\ b_1\ else\ b_2 : \tau}$$

$$\frac{e : \tau' \quad x : \tau' \dashv b : \tau}{let\ x = e\ in\ b}$$

Nous devons aussi ajouter des contraintes de types sur les variables ! ! !

Environnement

- Environnement E :
  suite finie de contraintes de type sur les variables :

$$x_1 : \tau_1, \quad x_2 : \tau_2, \quad \ldots, \quad x_n : \tau_n$$

  tel que $x_i \neq x_j$ si $i \neq j$.

- $\emptyset$: environnement vide.

- Environnement $E[\tau/x]$:
  même environnement que E,
  sauf pour x qui est de type τ dans $E[\tau/x]$.

Deuxième essai

$$\frac{n \in \mathbb{N}}{E \vdash n : nat} \qquad \frac{v \in \{ true, false \}}{E \vdash v : bool}$$

$$\frac{x \text{ a type } \tau \text{ dans } E}{E \vdash x : \tau}$$

$$\frac{E \vdash e_1 : nat \quad e_2 : nat}{E \vdash e_1 + e_2 : nat} \qquad \frac{E \vdash e : bool \quad E \vdash b_1 : \tau \quad E \vdash b_2 : \tau}{E \vdash if\ e\ then\ b_1\ else\ b_2 : \tau}$$

$$\frac{E \vdash e : \tau' \quad E[\tau'/x] \vdash b : \tau}{E \vdash let\ x = e\ in\ b}$$

Exemples

L'expression

$$\mathit{if}\ \mathit{true}\ \mathit{then}\ 0\ \mathit{else} 1$$

a type *nat*, car on peut « la typer » par *nat* :

$$\dfrac{\dfrac{\mathit{true} \in \{\mathit{true}, \mathit{false}\}}{\emptyset \vdash \mathit{true} : \mathit{bool}} \quad \dfrac{0 \in \mathbb{N}}{\emptyset \vdash 0 : \mathit{nat}} \quad \dfrac{1 \in \mathbb{N}}{\emptyset \vdash 1 : \mathit{nat}}}{\emptyset \vdash \mathit{if}\ \mathit{true}\ \mathit{then}\ 0\ \mathit{else}\ 1 : \mathit{nat}}$$

On ne peut pas typer l'expression

$$\mathit{if}\ x\ \mathit{then}\ 0\ \mathit{else}\ 1$$

car il n'existe pas $\tau \in \{\mathit{bool}, \mathit{nat}\}$ et un arbre de la forme

$$\dfrac{\vdots}{\emptyset \vdash \mathit{if}\ x\ \mathit{then}\ 0\ \mathit{else}\ 1 : \tau}$$

Le théorème principal

Théorème.
Si on peut typer l'expression e,
alors cette expression s'évalue à un valeur.

Plus en détailles :
Proposition.
Si $\emptyset \vdash e : \tau$, alors il existe v tel que $v : \tau$ et $e \Downarrow v$.

Pour pouvoir montrer cela :
Lemme.
Si $x_1 : \tau_1, \ldots, x_n : \tau_n \vdash e : \tau$ et $\emptyset \vdash v_i : \tau_i$ pour $i = 1, \ldots, n$,
alors il existe v tel que $v : \tau$ et $[v_1/x_1, \ldots, v_n/x_n]e \Downarrow v$.