IFT-3065/6232

http://www.iro.umontreal.ca/~monnier/6232

Définition de langages

- Règles lexicales, syntaxiques, de typage et d'évaluation

Techniques d'interprétation

- Analyse syntaxique, interprètes, gestion mémoire

Techniques de compilation

- Gestion de la pile, allocation de registres, génération de code
- Analyses de code, représentations internes, optimisations
- types, objets, fonctions, continuations, interface avec GC

Exemple de compilation -1- Source

```
let fun map f [] = []
      | map f (x::xs) = (f x)::(map f xs)
in map inc
end
```

Exemple de compilation -2- Inférence de type

```
let map : ('a -> 'b) -> 'a list -> 'b list
    = fn f : 'a -> 'b =>
          fn l : 'a list =>
          case l
           of [] => [] : 'b list
            | (x::xs) => (f x)::(map f xs)
in map inc : int list -> int list
```

Exemple de compilation -3- Filtrage

```
let map =
    fn f =>
       fn l =>
          switch l
          of [] => []
           | :: => let x  = hd l
                       xs = tl l
                   in (f x)::(map f xs)
in map inc
```

Exemple de compilation -4- A-normalization

```
map = fn f => fn l =>
              switch l
               of [] => []
                | :: => x  = hd l;
                        xs = tl l;
                        t1 = f x;
                        t2 = map f;
                        t3 = t2 xs;
                        t4 = t1::t3;
                        return t4;

t5 = map inc;
return t5
```

Exemple de compilation -5- FixFix

```
map1 = fn (f, l) =>
       switch l
        of [] => []
         | :: => x  = hd l;
                 xs = tl l;
                 t1 = f x;
                 t2 = map f;
                 t3 = t2 xs;
                 t4 = t1::t3;
                 return t4;
map = fn f => fn l => map1 (f, l);
t5 = map inc;
return t5
```

Exemple de compilation -6- Contract

```
map1 = fn (f, l) =>
       switch l
        of [] => []
         | :: => x  = hd l;
                 xs = tl l;
                 t1 = f x;
                 t3 = map1 (f, xs);
                 t4 = t1::t3;
                 return t4;
t5 = fn l => return map1 (inc, l);
return t5
```

Exemple de compilation -7- CPS

```
map1 = fn (f, l, k1) =>
    switch l
    of [] => k1 []
     | :: => x = hd l;
             xs = tl l;
             k2 = fn t1 =>
                  k3 = fn t3 =>
                       t4 = t1::t3;
                       call k1 (t4);
                  call map1 (f, xs, k3);
             call f (x, k2);
t5 = fn (l, k4) => call map1 (inc, l, k4);
call k (t5)
```

Exemple de compilation -8- Fermetures

```
map1 = fn (f, l, k1) => switch l
   of [] => k1c = k1.0; call k1c ([], k1)
    | :: => x = hd l; xs = tl l;
            k2c = fn (t1, c1) =>
                  f = c1.1; xs = c1.2; k1 = c1.3;
                  k3c = fn (t3, c2) =>
                        t1 = c2.1; k1 = c2.2;
                        t4 = t1::t3;
                        k1c = k1.0;
                        call k1c (t4, k1);
                  k3 = vector (k3c, t1, k1);
                  call map1 (f, xs, k3);
            k2 = vector (k2c, f, xs, k1);
            call f (x, k2);
t5 = fn (l, k4) => call map1 (inc, l, k4);
call k (t5)
```

Exemple de compilation -9- Hoisting (1)

```
map1 (f, l, k1) =
   switch l
    of [] => k1c = k1.0; call k1c ([], k1)
     | :: => x = hd l;
             xs = tl l;
             k2 = vector (k2c, f, xs, k1);
             call f (x, k2);
k2c (t1, c1) =
   f = c1.1;
   xs = c1.2;
   k1 = c1.3;
   k3 = vector (k3c, t1, k1);
   call map1 (f, xs, k3);
```

Exemple de compilation -9- Hoisting (2)

```
k3c (t3, c2) =
    t1 = c2.1;
    k1 = c2.2;
    t4 = t1::t3;
    k1c = k1.0;
    call k1c (t4, k1);

t5 (l, k4) = call map1 (inc, l, k4);

call k (t5)
```

Exemple de compilation -10- CodeGen (1)

```
map1:   mv f, r1          #
        mv l, r2          #
        mv k1, r3         # fn (f, l, k1) =>
        bnz l, case2      # switch l {.. of [] ..}
        ld k1c, k1(0)     # k1c = k1.0
        mv r1, 0          #
        mv r2, k1         #
        jmp k1c           # call k1c ([], k1)
```

Exemple de compilation -10- CodeGen (2)

```
case2:                      # | :: =>
        ld x, l(0)          # x = hd l
        ld xs, l(4)         # xs = tl l
        sub sp, sp, 16      #
        st sp(12), k1       #
        st sp(8), xs        #
        st sp(4), f         #
        st sp(0), k2c       #
        mv k2, sp           # k2 = vector (k2c,f,xs,k1)
        mv r1, x            #
        mv r2, k2           #
        jmp f               # call f (x, k2)
```

Exemple de compilation -10- CodeGen (3)

```
k2c:    mv t1, r1          #
        mv c1, r2          # fn (t1, c1) =>
        ld f, c1(4)        # f = c1.1
        ld xs, c1(8)       # xs = c1.2
        ld k1, c1(12)      # k1 = c1.3
        sub sp, sp, 12     #
        st sp(8), k1       #
        st sp(4), t1       #
        st sp(0), k3c      #
        mv k3, sp          # k3 = vector (k3c, t1, k1)
        mv r1, f           #
        mv r2, xs          #
        mv r3, k3          #
        jmp map1           # call map1 (f, xs, k3)
```

Exemple de compilation -10- CodeGen (4)

```
k3c:    mv t3, r1          #
        mv c2, r2          # fn (t3, c2) =>
        ld t1, c2(4)       # t1 = c2.1
        ld k1, c2(8)       # k1 = c2.2
        sub sp, sp, 8      #
        st sp(4), t3       #
        st sp(0), t1       #
        mv t4, sp          # t4 = t1::t3
        ld k1c, k1(0)      # k1c = k1.0
        mv r1, t4          #
        mv r2, k1          #
        jmp k1c            # call k1c (t4, k1)
```

Exemple de compilation -10- CodeGen (5)

```
t5:     mv l, r1          #
        mv k4, r2         # fn (l, k4) =>
        mv r1, inc        #
        mv r2, l          #
        mv r3, k4         #
        jmp map1          # call map1 (inc, l, k4)
```

Exemple de compilation -11- RegAlloc (1)

```
map1:   mv r4, r1          # fn (f, l, k1) =>
        bnz r2, case2      # switch l {.. of [] ..}
        ld r4, r3(0)       # k1c = k1.0
        mv r1, 0           #
        mv r2, r3          #
        jmp r4             # call k1c ([], k1)
```

Exemple de compilation -11- RegAlloc (2)

```
case2:                      # | :: =>
        ld r1, r2(0)        # x = hd l
        ld r2, r2(4)        # xs = tl l
        sub sp, sp, 16      #
        st sp(12), r3       #
        st sp(8), r2        #
        st sp(4), r4        #
        st sp(0), k2c       #
        mv r2, sp           # k2 = vector (k2c,f,xs,k1)
        jmp r4              # call f (x, k2)
```

Exemple de compilation -11- RegAlloc (3)

```
k2c:                        # fn (t1, c1) =>
        ld r3, r2(4)        # f = c1.1
        ld r4, r2(8)        # xs = c1.2
        ld r2, r2(12)       # k1 = c1.3
        sub sp, sp, 12      #
        st sp(8), r2        #
        st sp(4), r1        #
        st sp(0), k3c       #
        mv r5, sp           # k3 = vector (k3c, t1, k1)
        mv r1, r3           #
        mv r2, r4           #
        mv r3, r5           #
        jmp map1            # call map1 (f, xs, k3)
```

Exemple de compilation -11- RegAlloc (4)

```
k3c:                    # fn (t3, c2) =>
        ld r3, r2(4)    # t1 = c2.1
        ld r2, r2(8)    # k1 = c2.2
        sub sp, sp, 8   #
        st sp(4), r1    #
        st sp(0), r3    #
        mv r1, sp       # t4 = t1::t3
        ld r3, r2(0)    # k1c = k1.0
        jmp r3          # call k1c (t4, k1)
```

Exemple de compilation -11- RegAlloc (5)

```
t5:     mv r4, r1          #
        mv r3, r2          # fn (l, k4) =>
        mv r1, inc         #
        mv r2, r4          #
        jmp map1           # call map1 (inc, l, k4)
```

Définir un langage

Définition de la syntaxe

- Règles lexicales
- Règles syntaxiques

Définition de la sémantique

- Règles de typages (sémantique statique)
- Règles d'évaluation (sémantique dynamique)

Règles lexicales

Comment diviser le texte en une séquence de *lexèmes*

$$lexer :: List\ Char \rightarrow List\ Token$$

Inclut généralement la liste de mots clefs

Et les règles qui définissent les commentaires, chaînes de caractères, identificateurs, nombres, opérateurs, ponctuation, . . .

Expressions régulières

Représentation compacte d'un ensemble (possiblement infini) de chaînes

$$
\begin{aligned}
ER ::= \; & \varepsilon \\
& char \\
& ER\ ER \\
& ER \mid ER \\
& ER* \\
& (\ ER\ )
\end{aligned}
$$

On peut aussi y trouver toutes sortes de sucre syntaxique:

E.g. *string*, $ER+$, $ER?$, etc...

Lexèmes

case	`case`
incr	`++`
openbr	`{`
comment	`/*` *any** `*/`
identifier	*alpha* (*alpha* | *digit* | `_`)*
number	*digit digit**
blank	(SPC | TAB | LF | CR | FF)*

Ambiguïtés: priorité au lexème le plus long, ou à la première règle

Matching RE

Plusieurs manières de vérifier si une chaîne est acceptée par une expression régulière:

RE-match définitionnel, avec retour arrière

NFA: Machine non-déterministe à états finis

DFA: Machine déterministe à états finis

RE-match

$$\mathit{match} :: \mathit{RE} \to \mathit{String} \to (\mathit{String} \to \alpha \to \alpha) \to \alpha \to \alpha$$

$$\mathit{match}\ \mathit{Empty}\ s\ k\ f = k\ s\ f$$

$$\mathit{match}\ (\mathit{Char}\ c)\ s\ k\ f = \text{case}\ s\ \text{of}\ (c' : s') \mid c = c' \Rightarrow k\ s'\ f; \_ \Rightarrow f$$

$$\mathit{match}\ (\mathit{Or}\ \mathit{re}_1\ \mathit{re}_2)\ s\ k\ f = \mathit{match}\ \mathit{re}_1\ s\ k\ (\mathit{match}\ \mathit{re}_2\ s\ k\ f)$$

$$\mathit{match}\ (\mathit{Concat}\ \mathit{re}_1\ \mathit{re}_2)\ s\ k\ f =$$

$$\quad \mathit{match}\ \mathit{re}_1\ s\ (\lambda s'\ f' \to \mathit{match}\ \mathit{re}_2\ s'\ k\ f')\ f$$

$$\mathit{match}\ (\mathit{Star}\ \mathit{re})\ s\ k\ f =$$

$$\quad \mathit{match}\ \mathit{re}\ s\ (\lambda s'\ f' \to \mathit{match}\ (\mathit{Star}\ \mathit{re})\ s'\ k\ f')\ (k\ s\ f)$$

RE-match with exceptions

$$\mathit{match} :: \mathit{RE} \to \mathit{String} \to (\mathit{String} \to \alpha) \to \alpha$$

$$\mathit{match}\ \mathit{Empty}\ s\ k = k\ s$$

$$\mathit{match}\ (\mathit{Char}\ c)\ s\ k = \text{case}\ s\ \text{of}\ c' : s' \mid c = c' \Rightarrow k\ s'; \_ \Rightarrow \text{throw Fail}$$

$$\mathit{match}\ (\mathit{Or}\ re_1\ re_2)\ s\ k\ f = \text{try}\ \mathit{match}\ re_1\ s\ k\ \text{catch}\ \mathit{Fail} \Rightarrow \mathit{match}\ re_2\ s\ k$$

$$\mathit{match}\ (\mathit{Concat}\ re_1\ re_2)\ s\ k = \mathit{match}\ re_1\ s\ (\lambda s' \to \mathit{match}\ re_2\ s'\ k)$$

$$\mathit{match}\ (\mathit{Star}\ re)\ s\ k\ f =$$

$$\quad \text{try}\ \mathit{match}\ re\ s\ (\lambda s' \to \mathit{match}\ (\mathit{Star}\ re)\ s'\ k)\ \text{catch}\ \mathit{Fail} \Rightarrow k\ s$$

¡Haskell n'a pas ces exceptions!

NFA

Une NFA se représente par un graphe où chaque nœud est un état, et chaque arc représente une transition, annotée par le caractère qui doit être lu pour permettre la transition

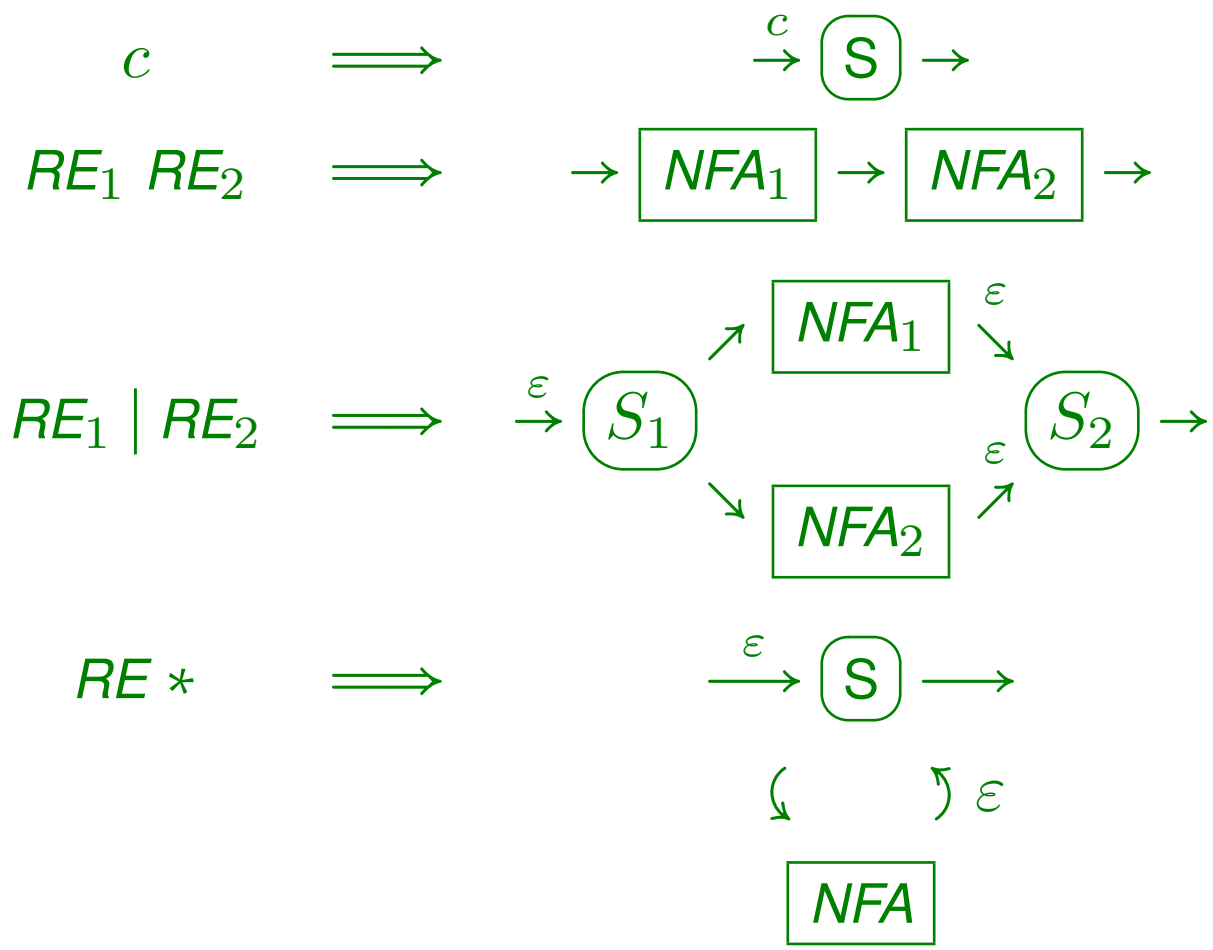
Certains NFA autorisent une annotation ε sur une transition qui indique qu'elle peut se prendre sans consommer de caractère

Un des états est déclaré "*état de départ*"

Certains des états sont notés "*états finaux*"

`lex`: compiler chaque *RE* en un *NFA* puis les combiner;
chaque *RE* a un état final différent, pour indiquer quel *RE* a été utilisé

Convertir une RE en une → NFA →



NFA sans retour arrière

Le comportement d'un NFA ne dépend pas du passé

"Thompson's NFA"

$$
\begin{aligned}
&\mathit{states} = \text{singleton}(\mathit{start});\\
&\text{while}(\forall s \in \mathit{states}.\ \neg\text{final}(s))\\
&\quad c = \text{read\_char}();\\
&\quad \mathit{states} = \{s_2 \mid s_1 \in \mathit{states} \wedge s_1 \xrightarrow{c} s_2\}
\end{aligned}
$$

Les différentes manières d'atteindre un état n'importent pas

Complexité $O(n * m)$ où n = nombre d'états et m = taille du texte

DFA

Une machine déterministe à états finis et comme une NFA sauf qu'une seule transition ne peut être prise pour un état et un caractère donné.

On peut construire un DFA où chaque nœud correspond à un état possible du *Thompson's NFA*:

- Chaque état du DFA correspond à un ensemble d'états du NFA accessibles à un même point d'un texte.
- Il y a un nombre fini d'ensembles d'états NFA possibles (dans le pire des cas 2^n ou n=nombre d'états du NFA)

DFA par dérivée

Derivatives of Regular Expressions, Janusz A. Brzozowski, JACM, 1964

Au lieu de passer par un NFA, on peut passer directement au DFA

$D_c(RE)$ = une *regexp* qui accepte le reste de ce que *RE* peut accepter après le caractère c.

Chaque état du DFA représente la RE qui reste à accepter:

- l'état de départ correspond à la RE initiale
- pour chaque état S_{RE} associé à une *RE* et chaque caractère c possible, la transition de S_{RE} par c mène à l'état associé à $D_c(RE)$.

Propriétés de la DFA par dérivée

Étendre *RE* avec $\varnothing$, conjonction et négation

Les NFA ne se prêtent pas aux conjonctions et négations

Utilise une fonction auxiliaire $\delta(e)$:
$\delta(e)$ ne matche que la chaîne vide et uniquement si e la matche aussi

Fonctionne de manière similaire à la dérivation en analyse

Moins trivial de montrer que l'automate est fini

A besoin d'optimisations pour donner de bons résultats

DFA par dérivée – $\delta(e)$

$$\delta(\varepsilon) = \varepsilon$$

$$\delta(\varnothing) = \varnothing$$

$$\delta(\mathit{char}) = \varnothing$$

$$\delta(e_1\ e_2) = \delta(e_1)\ \&\ \delta(e_2)$$

$$\delta(e_1 \mid e_2) = \delta(e_1) \mid \delta(e_2)$$

$$\delta(e_1\ \&\ e_2) = \delta(e_1)\ \&\ \delta(e_2)$$

$$\delta(e*) = \varepsilon$$

$$\delta(\neg e) = \begin{cases} \varepsilon & \text{si } \delta(e) = \varnothing \\ \varnothing & \text{si } \delta(e) = \varepsilon \end{cases}$$

DFA par dérivée – $D_c(e)$

$$D_c(\varepsilon) = \varnothing$$

$$D_c(\varnothing) = \varnothing$$

$$D_c(c) = \varepsilon$$

$$D_c(c') = \varnothing \quad \text{si } c \neq c'$$

$$D_c(e_1\ e_2) = D_c(e_1)\ e_2 \mid \delta(e_1)\ D_c(e_2)$$

$$D_c(e_1 \mid e_2) = D_c(e_1) \mid D_c(e_2)$$

$$D_c(e_1\ \&\ e_2) = D_c(e_1)\ \&\ D_c(e_2)$$

$$D_c(e*) = D_c(e)\ e*$$

$$D_c(\neg e) = \neg\ D_c(e)$$

Analyse syntaxique

Langage: ensemble de *phrases*

Phrase: séquence de symboles tiré d'un *alphabet*

Vocabulaire: ensemble de *symboles*

Grammaire: règles qui décrivent un langage

L'analyse syntaxique se base sur la définition syntaxique du langage, qui est constituée d'une grammaire (CFG), typiquement en notation BNF (Backus-Naur-Form)

Backus-Naur Form

Grammaire en format BNF: ensemble de *catégories* et de *productions* avec une catégorie désignée *catégorie de départ*

- *Catégorie* = nom d'un type de fragment de phrase
  Ex: ⟨expression⟩, ⟨entier⟩, ⟨type⟩
- *Production* = règle de la forme

$$\langle \text{cat} \rangle ::= x_1\ x_2\ \ldots\ x_n$$

où ⟨cat⟩ est une catégorie et x_i est une catégorie ou un symbole

Exemple:
$$\langle \text{bin} \rangle ::= 0$$
$$\langle \text{bin} \rangle ::= 1$$
$$\langle \text{bin} \rangle ::= \langle \text{bin} \rangle\ \langle \text{bin} \rangle$$

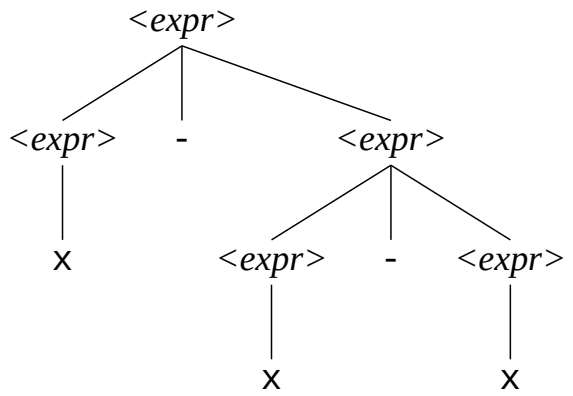
Grammaires ambiguës

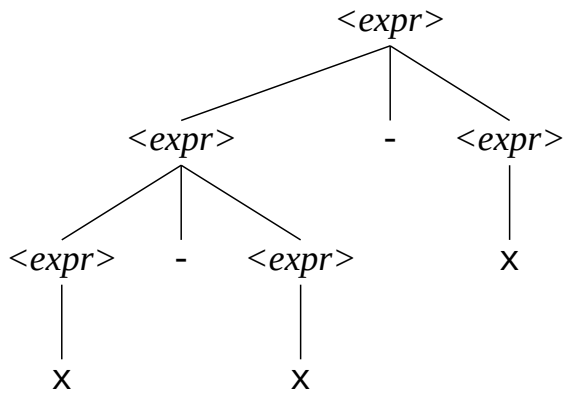
Déf: une grammaire G est *ambiguë* ssi il existe une phrase dans L(G) qui a plusieurs arbres de dérivation (pas juste plusieurs dérivations)

Exemple: $\langle \text{expr} \rangle ::= x$

$\langle \text{expr} \rangle ::= \langle \text{expr} \rangle - \langle \text{expr} \rangle$

$x - x - x$:





Résoudre ou éliminer les ambiguïtés en BNF pour l'utiliser dans l'ASA

Extended BNF

Extension de la syntaxe BNF avec la notation d'expressions régulières:

$x_1 \vert x_2$	peut-être soit x_1 soit x_2
(x)	groupement
$[x]$	parfois noté $x?$ équivalent à $\varepsilon \vert x$
$\{x\}$	parfois noté $x*$

Exemple: ⟨entier⟩ ::= ⟨chiffre⟩ { ⟨chiffre⟩ }

Volontiers utilisé dans la définition des langages, mais rarement accepté par les outils tels que `yacc`

Descente récursive

Chaque catégorie est implantée par une fonction

Incompatible avec les grammaires récursives à gauche

Si la grammaire est dans $LL(1)$, on peut éviter le retour arrière

Pour cela, on définit *First*(C) et *Follow*(C)

First(C) : symboles qui peuvent commencer un élément de C

Follow(C): symboles qui peuvent suivre un élément de C

Élimination de la récursivité à gauche

Une grammaire telle que:

$$\langle \text{exp} \rangle ::= \langle \text{exp} \rangle + \langle \text{exp} \rangle$$

$$\langle \text{exp} \rangle ::= \langle \text{id} \rangle$$

Se transforme en:

$$\langle \text{exp} \rangle ::= \langle \text{id} \rangle \ \langle \text{exp'} \rangle$$

$$\langle \text{exp'} \rangle ::=$$

$$\langle \text{exp'} \rangle ::= + \ \langle \text{id} \rangle \ \langle \text{exp'} \rangle$$

Beware: change l'associativité

Élimination de l'ambiguïté du `else`

Les règles classiques ambiguës:

$$\langle \text{exp} \rangle ::= \texttt{if}\ \langle \text{exp} \rangle\ \texttt{then}\ \langle \text{exp} \rangle$$

$$\langle \text{exp} \rangle ::= \texttt{if}\ \langle \text{exp} \rangle\ \texttt{then}\ \langle \text{exp} \rangle\ \texttt{else}\ \langle \text{exp} \rangle$$

Peuvent se réécrire:

$$\langle \text{exp} \rangle ::= \texttt{if}\ \langle \text{exp} \rangle\ \texttt{then}\ \langle \text{exp} \rangle$$

$$\langle \text{exp} \rangle ::= \langle \text{exp\_closed} \rangle$$

$$\langle \text{exp\_closed} \rangle ::= \texttt{if}\ \langle \text{exp} \rangle\ \texttt{then}\ \langle \text{exp\_closed} \rangle\ \texttt{else}\ \langle \text{exp} \rangle$$

Factorisation à gauche

Pour éviter le retour arrière, la grammaire suivante:

$$\langle \text{exp} \rangle ::= \langle \text{exp} \rangle \ \texttt{:=} \ \langle \text{exp} \rangle$$

$$\langle \text{exp} \rangle ::= \langle \text{exp} \rangle \ (\langle \text{exps} \rangle)$$

peut se factoriser à gauche:

$$\langle \text{exp} \rangle ::= \langle \text{exp} \rangle \ \langle \text{exp'} \rangle$$

$$\langle \text{exp'} \rangle ::= \ \texttt{:=} \ \langle \text{exp} \rangle$$

$$\langle \text{exp'} \rangle ::= (\langle \text{exps} \rangle)$$

LALR

Un automate déterministe à pile

shift: lit un token, le pousse sur la pile et saute à un nouvel état

reduce: applique une règle de la grammaire et retourne à l'appelant

goto: lors du retour à l'appelant, transfert à un autre état

Problème LALR

```
%token ID COMMA COLON
%%
def:    param_spec return_spec COMMA ;
param_spec:  type
        |    name_list COLON type ;
return_spec: type
        |    name COLON type ;
type:        ID ;
name:        ID ;
name_list:   name
        |    name COMMA name_list ;
```

GLR

Comme un parseur LALR, mais résout les conflits par le non-déterminisme

Si la grammaire n'est pas ambiguë, c'est comme LALR, $O(n)$

En cas d'ambiguïté, les différentes piles des différents états sont stockés de manière à partager leurs préfixes et suffixes communs

Similaire au "Thompson's NFA"

Complexité dans le pire des cas $O(n^3)$

Parsing Expression Grammar (PEG)

Comme une grammaire non-contextuelle (CFG), mais sans ambiguïtés

La notion d'alternation (notée "$|$") n'est pas symétrique

Se prête aux extensions telles quel le conjonction et la négation

N'aime pas la récursion à gauche (descente récursive)

Packrat: mémoization (programmation dynamique) pour éviter l'explosion combinatoire: $O(n)$ en temps, mais aussi $O(n)$ en mémoire

No free lunch: les ambiguïtés ne peuvent pas être détectées

Interprétation

Définition de la sémantique du langage

Beaucoup d'approches:

- Sémantique axiomatique
- Sémantique dénotationnelle
- Sémantique opérationnelle
- Petits ou grand pas
- Prose

Sémantique axiomatique

Chaque action est décrite par un axiome

$$\{P * x \mapsto \_\}$$

$$x := e$$

$$\{P * x \mapsto e\}$$

Se prête bien à la programmation impérative

Utilisé par la logique de Hoare, pour preuves formelles

Sémantique dénotationnelle

Le programme est *traduit* dans la théorie des ensembles

$$[\![\mathsf{n}]\!] = \lambda s.n$$

$$[\![x]\!] = \lambda s.s\ x$$

$$[\![e_1 e_2]\!] = \lambda s.([\![e_1]\!]\ s)\ ([\![e_2]\!]\ s)$$

$$[\![\lambda x.e]\!] = \lambda s.\lambda v.[\![e]\!]\ \{s, x \mapsto v\}$$

$$[\![\mathsf{fix}\ f]\!] = \mathsf{let}\ F = [\![f]\!]\ \mathsf{in}\ \sqcup_i\ F^i\ \varnothing$$

Programmes extensionnellement identiques $\Leftrightarrow$ même dénotation

Se prête bien à la programmation déclarative

Sémantique opérationnelle

Le langage est décrit par un interpréteur sur une machine hypothétique

$$(M, x \mapsto \_; x := v; e) \Rightarrow (M, x \mapsto v; e)$$

$$(M; \text{let } x = \text{n}_1 + \text{n}_2 \text{ in } e) \Rightarrow (M; \text{let } x = \text{n in } e) \quad \text{where } n = n_1 + n_2$$

$$(M; \text{let } x = v \text{ in } e) \Rightarrow (M; e[v/x])$$

Se prête à tout, à condition de bien choisir sa machine hypothétique

Impossible de comparer différents langages

Petits pas

La règle de base (notée souvent $\rightsquigarrow$ ou $\Rightarrow$) avance d'un seul pas

Divisé en règles de réduction primitives

$$\mathsf{n}_1 + \mathsf{n}_2 \rightsquigarrow \mathsf{n} \qquad (\lambda x.e)v \rightsquigarrow e[v/x] \qquad \#1(v_1, v_2) \rightsquigarrow v_1$$

et règles de congruence

$$\frac{e \rightsquigarrow e'}{v\ e \rightsquigarrow v\ e'} \qquad \frac{e_1 \rightsquigarrow e_1'}{e_1\ e_2 \rightsquigarrow e_1'\ e_2} \qquad \frac{e_1 \rightsquigarrow e_1'}{e_1 + e_2 \rightsquigarrow e_1' + e_2} \qquad \frac{e \rightsquigarrow e'}{v + e \rightsquigarrow v + e'}$$

L'évaluation complète se fait par une séquence de pas: $e \rightsquigarrow^* v$

Chaque état intermédiaire de la machine doit être représentable

Grands pas

La règle de réduction (notée $\downarrow$) renvoie directement le résultat final

$$\lambda x.e \downarrow \lambda x.e$$

$$\frac{e_1 \downarrow \lambda x.e_3 \quad e_2 \downarrow v_2 \quad e_3[v_2/x] \downarrow v}{e_1\ e_2 \downarrow v}$$

$$\frac{e_1 \downarrow \mathsf{n}_1 \quad e_2 \downarrow \mathsf{n}_2 \quad n_1 + n_2 = n}{e_1 + e_2 \downarrow \mathsf{n}}$$

Se prête bien à l'implantation d'un interprète

Les règles ne disent **rien** des programmes qui ne terminent pas

Interprétation

Substitution pas efficace ⇒ environnements (“substitution paresseuse”)

$$(S; \lambda x.e) \downarrow \lambda^S x.e \qquad (S; x) \downarrow S(x)$$

$$\frac{(S; e_1) \downarrow \lambda^{S'} x.e_3 \quad (S; e_2) \downarrow v_2 \quad (S', x \mapsto v_2; e_3) \downarrow v}{(S; e_1\ e_2) \downarrow v}$$

$$\frac{(S; e_1) \downarrow \mathsf{n}_1 \quad (S; e_2) \downarrow \mathsf{n}_2 \quad n_1 + n_2 = n}{(S; e_1 + e_2) \downarrow \mathsf{n}}$$

Très utile aussi pour gérer la récursion

Exemple d'interpréteur

Adaptation de la sémantique à grands pas avec environnement

$$\mathit{eval} :: \mathit{Exp} \to \mathit{Env} \to \mathit{Val}$$

$$\mathit{eval}\ (\mathsf{Lnum}\ n)\ \_ = \mathsf{Vnum}\ n$$

$$\mathit{eval}\ (\mathsf{Lvar}\ x)\ \mathit{env} = \mathsf{lookup}\ x\ \mathit{env}$$

$$\mathit{eval}\ (\mathsf{Lapp}\ e_1\ e_2)\ \mathit{env} =$$

$$\quad \mathsf{case}\ \mathit{eval}\ e_1\ \mathit{env}\ \{\ \mathsf{Vfun}\ f \to f\ (\mathit{eval}\ e_2\ \mathit{env})\ \}$$

$$\mathit{eval}\ (\mathsf{Lfun}\ xe)\ \mathit{env} = \mathsf{Vfun}\ (\lambda\ v \to \mathit{eval}\ e\ (\mathsf{insert}\ \mathit{env}\ x\ v))$$

$$\mathit{eval}\ (\mathsf{Llet}\ x\ e_1\ e_2)\ \mathit{env} = \mathit{eval}\ e_2\ (\mathsf{insert}\ \mathit{env}\ x\ (\mathit{eval}\ e_1\ \mathit{env})$$

La règle de l'addition est placée dans l'environnement initial

Interpréteurs méta-circulaires

Usage de fonctionnalités du méta-langage pour le langage objet

- appel par valeur/nom/besoin
- portée dynamique/statique
- gestion mémoire
- représentation des objets (e.g. tags via datatypes)
- effets de bord
- variables et substitutions

Sources des coûts de l'interprétation

1. Extraire op-code et sauter à la branche correspondante
2. Chercher les opérandes
3. Faire le travail
4. Stocker le résultat
5. Aller chercher l'instruction suivante

Le 3 est "incompressible" (modulo optimisations)

Le reste +/- coûteux selon la technique d'interprétation

Variations des coûts de l'interprétation

Granularité des instructions

- Extraire op-code et sauter à la branche correspondante

  Arbre binaire de décision; saut indirect

  Duplication des sauts pour les rendre plus prévisibles

- Chercher les opérandes et stocker le résultat

  Recherche dans l'environnement; indexage direct

- Aller chercher l'instruction suivante

  Via indirection; ou simple placement consécutif

Normalisation/linéarisation

Augmenter la granularité en ajoutant Llet et Lvar partout

```
data Lexp = Lnum Int Var Lexp
          | Lapp Var Var Var Lexp
          | Lfun Var Lexp Var Lexp
          | Lreturn Var
```

Chaque expression est maintenant une liste d'opérations

Plus de variables et de références à des variables

N'a de sens que si les variables sont gérées efficacement

DeBruijn

Accéder aux variables de manière efficace

```
type Var = Int
data Lexp = Lnum Int Lexp
          | Lapp Var Var Lexp
          | Lfun Lexp Lexp
          | Lreturn Var
```

Chaque variable est distinguée par sa position dans le contexte

Pas de noms ⇒ déclarations implicites

Opérandes implicites

Mettre les variables sur une pile

```
data Lexp = [Lop]
data Lop = Lnum Int
         | Lprim String
         | Lfun Lexp
         | Lapp | Lreturn | Lswap
```

Plus de variables, seuls des noms de primitives ou variables globales

Lnum, Lprim, et Lfun sont généralement remplacés par Lpush Value

Besoin d'ajouter Lswap et autres opérations de manipulation de pile

Bytecode

Variables:

- Bytecode à base de pile (w/ TOS cache)
- Bytecode à base de registres

Interprétation:

- switch + goto
- indirect threading
- direct threading
- superinstructions
- duplicate instructions

Staging

$$\mathit{eval} :: \mathit{Exp} \to [\mathit{Var}] \to [\mathit{Val}] \to \mathit{Val}$$

$$\mathit{eval}\ (\mathsf{Lnum}\ n)\ \_ = \mathsf{let}\ v = \mathsf{Vnum}\ n\ \mathsf{in}\ \lambda\_ \to v$$

$$\mathit{eval}\ (\mathsf{Lvar}\ x)\ xs = \mathsf{let}\ i = \mathsf{lookup}\ x\ xs$$
$$\mathsf{in}\ \lambda vs \to \mathsf{nth}\ i\ vs$$

$$\mathit{eval}\ (\mathsf{Lapp}\ e_1\ e_2)\ xs = \mathsf{let}\ f_1 = \mathit{eval}\ e_1\ xs\ ;\ f_2 = \mathit{eval}\ e_2\ xs$$
$$\mathsf{in}\ \lambda vs \to \mathsf{case}\ f_1\ vs$$
$$\mathsf{Vfun}\ f \to f\,(f_2\ vs)$$

$$\mathit{eval}\ (\mathsf{Lfun}\ x\ e)\ xs = \mathsf{let}\ f_e = \mathit{eval}\ e\ (x : xs)$$
$$\mathsf{in}\ \lambda vs \to \mathsf{Vfun}\ (\lambda v \to f_e(v : vs))$$

Exemple de staging

$$\textit{eval}\ (\text{Lapp}\ (\text{Lapp}\ (\text{Lvar}\ \texttt{"+"})\ (\text{Lnum}\ 1))\ (\text{Lnum}\ 2))\ [\ \texttt{"+"}\ ]$$

$$\Rightarrow$$

$$i = 0$$

$$f_{11} = \lambda vs \rightarrow \text{nth}\ i\ vs$$

$$v_1 = \text{Vnum}\ 1$$

$$f_{12} = \lambda\_ \rightarrow v_1$$

$$f_1 = \lambda vs \rightarrow \text{case}\ f_{11}\ vs\ \{\ \text{Vfun}\ f \rightarrow f(f_{12}\ vs)\ \}$$

$$v_2 = \text{Vnum}\ 2$$

$$f_2 = \lambda\_ \rightarrow v_2$$

$$\lambda vs \rightarrow \text{case}\ f_1\ vs\ \{\ \text{Vfun}\ f \rightarrow f(f_2\ vs)\ \}$$

Détail du résultat

$$x_1 = \lambda^{\{i \mapsto 0\}} vs \to \text{nth } i\ vs$$

$$x_2 = \lambda^{\{v \mapsto \text{Vnum } 1\}} \_ \to v$$

$$x_3 = \lambda^{\{f_1 \mapsto x_1, f_2 \mapsto x_2\}} vs \to \text{case } f_1\ vs\ \{\ \text{Vfun } f \to f(f_2\ vs)\ \}$$

$$x_4 = \lambda^{\{v \mapsto \text{Vnum } 2\}} \_ \to v$$

$$x_5 = \lambda^{\{f_1 \mapsto x_3, f_2 \mapsto x_4\}} vs \to \text{case } f_1\ vs\ \{\ \text{Vfun } f \to f(f_2\ vs)\ \}$$

Direct threading

Accès à l'environnement sans recherche

Indirection pour accéder à l'instruction suivante

Tags

Typage dynamique $\Rightarrow$ représentation des types par des *tags*

- Tags dans les objets: typique pour le OO
- Tagbits dans les pointeurs: populaire pour les petits objets
- Tags séparés: permet le partage et l'optimisation
- Bibop: coûteux d'accès, mais très compact
- hybrides: personne n'est parfait

Règles de typage

L'expression e a type τ dans un contexte Γ s'écrit: $\Gamma \vdash e : \tau$

$$\frac{\Gamma(x) = \tau}{\Gamma \vdash x : \tau} \qquad \frac{}{\Gamma \vdash n : \mathsf{Int}} \qquad \frac{\Gamma \vdash e_1 : \tau_1 \to \tau_2 \qquad \Gamma \vdash e_2 : \tau_1}{\Gamma \vdash e_1\ e_2 : \tau_2}$$

$$\frac{\Gamma, x{:}\tau_1 \vdash e : \tau_2}{\Gamma \vdash \lambda x \to e : \tau_1 \to \tau_2}$$

La syntaxe est très flexible

Les règles n'impliquent pas forcément un algorithme

Vérification de types

Le type de chaque variable est fourni

$\Gamma \vdash e : \tau$ est une fonction

Définitions récursives OK

Si les règles de typage sont en prose, il faut interpréter

Grammaires attribuées

attributs synthétisés: e.g. AST

attributs hérités: e.g. contexte

yacc : seul un attribut, synthétisé

Sous-typage

Classes, objets, sous-types, héritage, interfaces, prototypes, méthodes

if-typing

casts

covariance/contravariance

égalité de types

Vérification bidirectionnelle

Maximiser la propagation, minimiser la vérification

$$\frac{\Gamma(x) = \tau}{\Gamma \vdash x \Rightarrow \tau} \qquad \frac{}{\Gamma \vdash n \Rightarrow \mathsf{Int}} \qquad \frac{\Gamma \vdash e \Leftarrow \tau}{\Gamma \vdash (e{:}\tau) \Rightarrow \tau}$$

$$\frac{\Gamma \vdash e_1 \Rightarrow \tau_1 \to \tau_2 \qquad \Gamma \vdash e_2 \Leftarrow \tau_1}{\Gamma \vdash e_1\ e_2 \Rightarrow \tau_2} \qquad \frac{\Gamma \vdash e \Rightarrow \tau' \qquad \tau' \leq \tau}{\Gamma \vdash e \Leftarrow \tau}$$

$$\frac{\Gamma, x{:}\tau_1 \vdash e \Leftarrow \tau_2}{\Gamma \vdash \lambda x \to e \Leftarrow \tau_1 \to \tau_2}$$

Vérification non-directionnelle

Pour les paires, les règles sont habituellement:

$$\frac{\forall i. \quad \Gamma \vdash e_i \Leftarrow \tau_i}{\Gamma \vdash (e_1, e_2) \Leftarrow \tau_1 \times \tau_2} \qquad \frac{\Gamma \vdash e \Rightarrow \tau_1 \times \tau_2}{\Gamma \vdash e.i \Rightarrow \tau_i}$$

Mais d'autres règles sont possibles et utiles:

$$\frac{\forall i. \quad \Gamma \vdash e_i \Rightarrow \tau_i}{\Gamma \vdash (e_1, e_2) \Rightarrow \tau_1 \times \tau_2} \qquad \frac{\Gamma \vdash e_1 \Leftarrow \tau_1 \to \tau_2 \qquad \Gamma \vdash e_2 \Rightarrow \tau_1}{\Gamma \vdash e_1\ e_2 \Leftarrow \tau_2}$$

Inférence de types: Hindley-Milner

$$\frac{\tau \prec \Gamma(X)}{\Gamma \vdash x : \tau} \quad [\text{Var}]_{\text{HM}}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \to \tau_2 \qquad \Gamma \vdash e_2 : \tau_1}{\Gamma \vdash e_1\ e_2 : \tau_2} \quad [\text{App}]_{\text{HM}}$$

$$\frac{\Gamma, x{:}\tau_1 \vdash e : \tau_2}{\Gamma \vdash \lambda x \to e : \tau_1 \to \tau_2} \quad [\text{Abs}]_{\text{HM}}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma, x{:}\forall\vec{\alpha}.\tau_1 \vdash e_2 : \tau_2 \quad \vec{\alpha} = \text{fv}(\tau_1) - \text{fv}(\Gamma)}{\Gamma \vdash \text{let } x = e_1 \text{ in } e_2 : \tau_2} \quad [\text{Let}]_{\text{HM}}$$

Macros

- Macros CPP: $[\mathit{Token}] \rightarrow [\mathit{Token}]$
- Macros Scheme: $\mathit{Sexp} \rightarrow \mathit{Sexp}$

D'autres macros peuvent être $\mathit{AST} \rightarrow \mathit{AST}$ ou encore $[\mathit{Char}] \rightarrow [\mathit{Char}]$

Bien sûr, il faut analyser les définitions "avant" de les utiliser

Gestion mémoire

Comme les boîtes à vitesse:

- Automatique: la désallocation est prise en charge par le langage
- Manuelle: la désallocation est à la charge du programmeur
- Semi-automatique: le langage vérifie les désallocations

Granularité:

- Par objet: chaque objet est alloué/désalloué individuellement
- Par région: chaque objet est alloué dans une région, la désallocation opère sur tous les objets d'une région

Règle générale: facile pour le programmeur $\Rightarrow$ dur pour le compilateur

Gestion mémoire manuelle

En C, le compilateur ne peut rien faire

Sinon, il peut optimiser:

- le code de l'allocation
- le lieu d'allocation (tas, pile, global)

Gestion mémoire automatique

- Comptage de références: à chaque objet est associé un compteur qui indique combien de pointeurs existent. Lorsque le compteur passe à 0, on peut désallouer l'objet.
- GC (Glanage de Cellules ou plutôt Garbage Collection): à partir des variables globales et de la pile, traverser tous les objets atteignables en passant par tous les pointeurs: les objets non-visités peuvent être désalloués.
- Régions: une analyse du code détermine dans quelle région allouer chaque objet, et à quel moment désallouer chaque région.

Comptage de références

Convention d'appel: incrément soit par l'appelant soit par l'appelé

Avant de libérer un objet, il faut décrémenter tous les pointeurs contenus

Synchronisation en cas de concurrence

affectation: incrémenter *avant* de décrémenter

Cher en taille de code et temps d'exécution

Taille des compteurs

Taille d'un compteur: un mot, quelques bits, 1 bit

1 bit suffit pour tous les objets qui n'ont qu'un seul pointeur

En cas de dépassement:

- Compteurs à saturation: lorsqu'on atteint le maximum, on y reste
  Ces objets ne peuvent plus être récupérés, à moins d'un GC
- réserve auxiliaire de grands compteurs

Réduire les mises à jour de compteurs

Ne pas compter les pointeurs de la pile [Deutsch&Bobrow, 1976]:

- Augmente le nombre d'objets où 1bit suffit
- Un compteur à 0 met l'objet sur une liste de candidats
- Scan occasionnel de la pile pour trouver quels candidats libérer

Mise à jour occasionnelle des compteurs [Levanoni&Petrank, 2001]:

- au lieu de $inc(newX); dec(oldX)$, sauve $oldX$ ainsi que $\&X$.
- La liste des $\&X$ peut être compactée
- Plus besoin de synchronisation dans le mutateur
- Ça s'appelle une *barrière en écriture* (*write barrier*)

Collecter les cycles

Utiliser un GC de temps à autres:

- Peut aussi servir à mettre à jour les compteurs saturés
- Ou à compacter le tas pour éviter le tas

Détecter les cycles morts [Martínez&Wachenchauzer&Lins, 1990]:

- Décrément à une valeur autre que 0 $\Rightarrow$ candidat
- Prendre les candidats comme des “racines négatives”;
  Décrémenter transitivement les compteurs des objets pointés;
  Récupérer les objets qui finissent avec un compteur à zéro;
  Ré-incrémenter transitivement les compteurs depuis les survivants

Garbage Collection

Périodiquement (e.g., besoin de mémoire) chercher les déchets:

En partant des *racines*, marquer transitivement les objets accessibles

Puis, récupérer les objets restés inaccessibles

Deux algorithmes de base: mark&sweep et stop&copy

Le *collecteur* et le *mutateur* doivent se synchroniser

Une phase de collection peut coûter très cher

Mark (&Sweep)

```
mark (ptr) {
   if (!ptr->marked) {
      ptr->marked = True;
      for i = 0 to ptr->size
         mark (ptr[i]);
}  }

mark_all () {
   for varptr in roots
      mark (*varptr);
}
```

(Mark&) Sweep

```
sweep_all () {
   ptr = heap_start;
   do {
      if (ptr->marked)
         ptr->marked = False;
      else
         free_object (ptr);
   } while (ptr = next_object (ptr))
}
```

(Stop&)Copy

```
copy (ptr) {
   if (ptr->forward = NULL) {
      for i = 0 to ptr->size
         *(alloc_ptr + i) = ptr[i];
      ptr->forward = alloc_ptr;
      alloc_ptr += ptr->size;
   }
   return ptr->forward;
}
```

Stop&Copy

```
stop&copy () {
   alloc_ptr = scan_ptr = alloc_new_heap ();
   for varptr in roots
      *var = copy (*var);
   while (scan_ptr < alloc_ptr) {
      for i = 0 to scan_ptr->size
         ptr[i] = copy (ptr[i]);
      scan_ptr += scan_ptr->size;
   }
   free_old_heap ();
}
```

Besoins d'un GC

Quand exécuter le GC: GC-check

Racines: liste des variables globales, description de la pile

Trouver la taille des objets, distinguer les pointeurs:

tagbits, descripteurs de type

Idem pour les blocks d'activations de la pile:

safe points, stack maps

Un bit (mark ou forward), voire plus pour concurrence

Barrière en écriture ou en lecture pour algorithmes plus sophistiqués

GC++

GC *incrémental*: la phase de collection est saucissonnée

GC *concurrent*: collecteur et mutateur peuvent rouler simultanément

GC *parallèle*: le collecteur est divisé en sous-processus parallèles

GC *partitionné*: le tas est partitionné de sorte qu'une collection puisse ne récupèrer les déchets que dans une partition à la fois

GC *générationnel*: GC partitionné selon l'âge des objets

GC *distribué*: GC où le tas est distribué sur plusieurs machines

GC *temps-réel*: être mythique aux caractéristiques peu comprises, qui semblent inclure des bornes sur l'usage CPU du GC et sur le temps pendant lequel le mutateur peut être bloqué par le GC

Finalization, pointeurs faibles

Les systèmes à base de GC offrent souvent la possibilité de détecter quand un objet est désalloué:

- *Finaliseur*: fonction à exécuter avant de désallouer un objet
- *Pointeur faible*: pointeur qui n'empêche pas le GC de désallouer l'objet pointé. À chaque usage, il faut vérifier s'il est NULL

D'abord, marquer les objets sans suivre les pointeurs faibles;

ensuite, sauver la liste des objets finalisables non-marqués
et mettre à NULL les pointeurs faibles sur des objets non marqués;

puis, marquer à nouveau, depuis les objets à finaliser;

finalement, appeler chacun des finaliseurs (dans quel ordre?)

FUTUR

Ce qui suit n'existe pas encore!

Structure globale

Phases de compilation et phases d'optimisations

Table des symboles

Implantation fonctionnelle ou impérative de chaque phase

Phases d'analyses

"Canonicalisation" du code

Caractéristiques des phases de compilation

Le langage d'entrée est généralement différent de celui de sortie

Rapproche le programme du langage cible

Élimination de fonctionnalités et concepts particuliers

Rend explicite certaines partie du code

Renvoie du code plus verbeux

Caractéristiques des optimisations

Le langage d'entrée est généralement identique à celui de sortie

Utilise souvent la synergie entre optimisations

Ordre pas forcément évident; application réitérée

Éliminer le coût des abstractions

Optimiser le code qui vient du compilateur lui-même

Laisser le programmeur optimiser son code

Pourtant code idiot n'implique pas programmeur idiot

Continuation Passing Style (CPS)

Basic blocks

Allocation de registres

Caller-save callee-save

Génération de code

Optimisations simples

Élimination de code mort

Constant folding

Constant propagation

Variable propagation

SSA/CPS

Ordonnancement d'instructions