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)

Université de Montréal

Front End

La première partie d'un compilateur implémente les 3 premières règles:

- Analyse lexicale: découper le texte d'entrée en lexèmes
- Analyse syntaxique: transformer ces lexèmes en un arbre de syntaxe
- Sémantique statique: vérifier les types, les règles de portée, ...

Après ça, il ne reste qu'à évaluer ou faire la compilation en-soi

Division dirigée par l'ingénierie plus que la théorie

Règles lexicales

Comment diviser le texte en une séquence de *lexèmes*

$$\mathit{lexer} :: \mathit{List\ Char} \rightarrow \mathit{List\ Token}$$

Inclut souvent 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

Trouver l'erreur?

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

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

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

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

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

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

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

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

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

RE-match with exceptions

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

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

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

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

$\quad \texttt{try}\ \mathit{match}\ \mathit{re}_1\ s\ k\ \texttt{catch}\ \mathit{Fail} \Rightarrow \mathit{match}\ \mathit{re}_2\ s\ k$

$\mathit{match}\ (\mathit{Concat}\ \mathit{re}_1\ \mathit{re}_2)\ s\ k = \mathit{match}\ \mathit{re}_1\ s\ (\lambda s' \rightarrow \mathit{match}\ \mathit{re}_2\ s'\ k)$

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

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

¡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

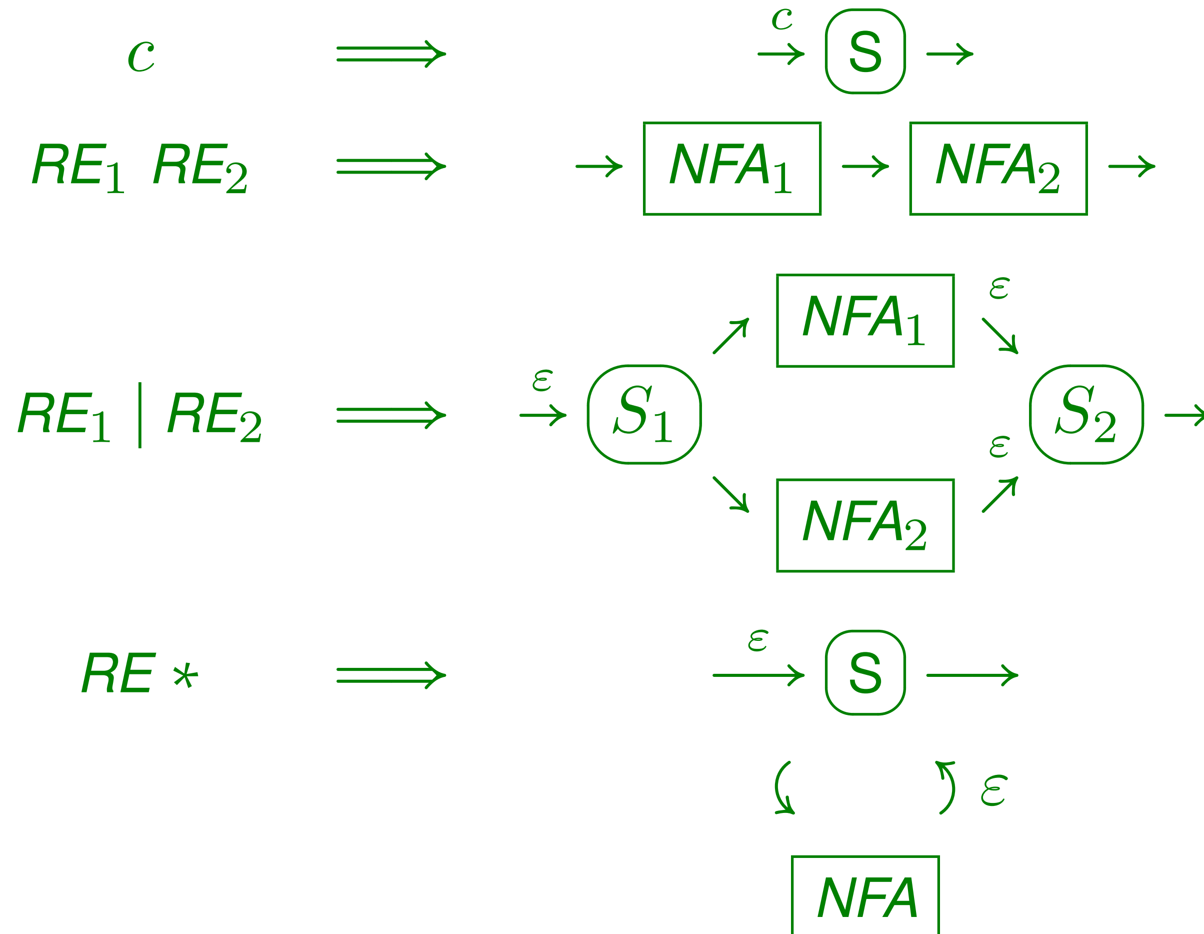
Un des états est déclaré "*état de départ*"

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

Université de Montréal

Convertir une RE en une → NFA →



Université de Montréal

NFA sans retour arrière

Le comportement d'un NFA ne dépend pas du passé

"Thompson's NFA"

$$\textit{states} = \texttt{singleton}(\textit{start});$$
$$\texttt{while}(\forall s \in \textit{states}.\ \neg\texttt{final}(s))$$
$$\quad c = \texttt{read_char}();$$
$$\quad \textit{states} = \{s_2 \mid s_1 \in \textit{states} \wedge s_1 \xrightarrow{c} s_2\}$$

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 est 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 où n=nombre d'états du NFA)

Exemple de DFA

Pour les expressions régulières suivantes:

IF	`if`
ENDIF	`endif`
ID	$alpha\ (alpha \mid digit)*$
COMMENT	$/*\ (\text{"not } */\text{"})*\ */$

Construire le NFA puis le DFA

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

$D_c(\mathit{RE})$ = 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 $D_c(\mathit{RE})$.

Propriétés de la DFA par dérivée

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

$\Longrightarrow$ C'est encore des expressions régulières!

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

Besoin d'optimisations pour donner de bons résultats

$\Longrightarrow$ Avec quelques optimisations, donne d'excellents résultats!

Université de Montréal

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: $\langle\text{expression}\rangle$, $\langle\text{entier}\rangle$, $\langle\text{type}\rangle$
- *Production* = règle de la forme

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

où $\langle\text{cat}\rangle$ est une catégorie et x_i est une catégorie ou un symbole

Exemple:

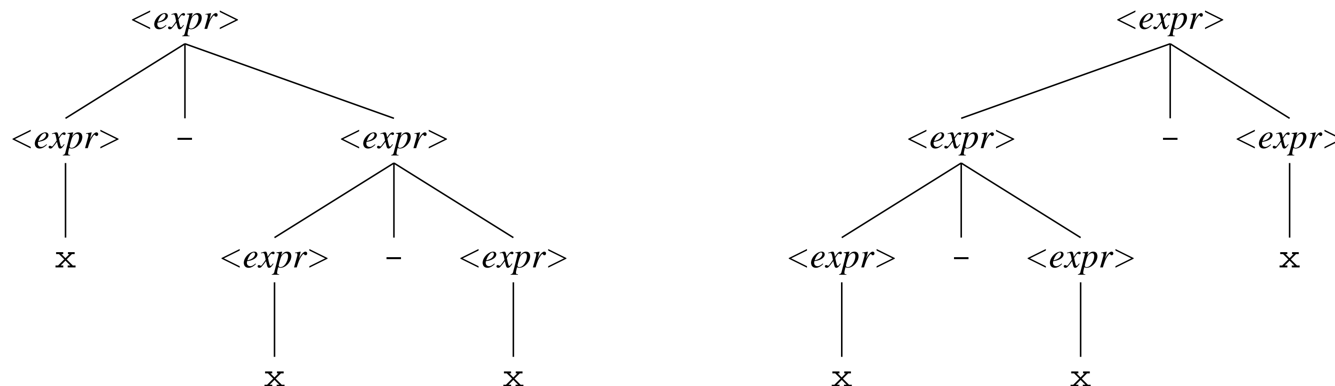
$$\begin{aligned}\langle\text{bin}\rangle &::= 0\\ \langle\text{bin}\rangle &::= 1\\ \langle\text{bin}\rangle &::= \langle\text{bin}\rangle\ \langle\text{bin}\rangle\end{aligned}$$

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:

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

Exemple: $\langle$entier$\rangle$ ::= $\langle$chiffre$\rangle$ $\{$ $\langle$chiffre$\rangle$ $\}$

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:

$$\langle \text{exp} \rangle ::= \langle \text{exp} \rangle \;\mathbf{:=}\; \langle \text{exp} \rangle$$

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

peut se factoriser à gauche:

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

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

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

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 la 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, au coût de $O(n \times m)$ 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

Université de Montréal

Sémantique dénotationnelle

Le programme est *traduit* dans la théorie des ensembles

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

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

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

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

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

Programmes extensionnellement identiques $\Leftrightarrow$ même dénotation

Se prête bien à la programmation déclarative et au raisonnement

Sémantique opérationnelle

Le langage est décrit par un interpréteur sur une machine hypothétique

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

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

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

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

Impossible de comparer différents langages

Pratique pour raisonner sur le langage plus que sur les programmes

Petits pas

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

Divisé en règles de réduction primitives

$$n_1 + n_2 \leadsto n \qquad (\lambda x.e)v \leadsto e[v/x] \qquad \#1(v_1, v_2) \leadsto v_1$$

et règles de congruence

$$\frac{e \leadsto e'}{v\ e \leadsto v\ e'} \quad \frac{e_1 \leadsto e_1'}{e_1\ e_2 \leadsto e_1'\ e_2} \quad \frac{e_1 \leadsto e_1'}{e_1 + e_2 \leadsto e_1' + e_2} \quad \frac{e \leadsto e'}{v + e \leadsto v + e'}$$

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

Chaque état intermédiaire de la machine doit être représentable

Grands pas

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

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

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

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

Se prête bien à l'implantation d'un interprète

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

Université de Montréal

Interprétation

Substitution pas efficace ⇒ environnements (“substitution paresseuse”)

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

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

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

Très utile aussi pour gérer la récursion

Université de Montréal

Exemple d'interpréteur

Adaptation de la sémantique à grands pas avec environnement

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

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

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

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

$$\quad \mathtt{case}\ \mathit{eval}\ e_1\ \mathit{env}\ \{\ \mathtt{Vfun}\ f \rightarrow f\ (\mathit{eval}\ e_2\ \mathit{env})\ \}$$

$$\mathit{eval}\ (\mathtt{Lfun}\ x\ e)\ \mathit{env} = \mathtt{Vfun}\ (\lambda\ v \rightarrow \mathit{eval}\ e\ (\mathtt{insert}\ \mathit{env}\ x\ v))$$

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

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

$$eval :: Exp \to [Var] \to [Val] \to Val$$

$$eval\,(\texttt{Lnum}\;n)\;_ = \texttt{let}\;v = \texttt{Vnum}\;n\;\texttt{in}\;\lambda_ \to v$$

$$eval\,(\texttt{Lvar}\;x)\;xs = \texttt{let}\;i = \texttt{lookup}\;x\;xs$$
$$\texttt{in}\;\lambda vs \to \texttt{nth}\;i\;vs$$

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

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

Exemple de staging

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

$\Rightarrow$

$$i = 0$$
$$f_{11} = \lambda vs \rightarrow \texttt{nth}\; i\; vs$$
$$v_1 = \texttt{Vnum}\; 1$$
$$f_{12} = \lambda_ \rightarrow v_1$$
$$f_1 = \lambda vs \rightarrow \texttt{case}\; f_{11}\; vs\; \{\, \texttt{Vfun}\; f \rightarrow f(f_{12}\; vs)\, \}$$
$$v_2 = \texttt{Vnum}\; 2$$
$$f_2 = \lambda_ \rightarrow v_2$$
$$\lambda vs \rightarrow \texttt{case}\; f_1\; vs\; \{\, \texttt{Vfun}\; f \rightarrow f(f_2\; vs)\, \}$$

Détail du résultat

$$x_1 = \lambda^{\{i \mapsto 0\}} vs \to \texttt{nth}\ i\ vs$$
$$x_2 = \lambda^{\{v \mapsto \texttt{Vnum}\ 1\}} _ \to v$$
$$x_3 = \lambda^{\{f_1 \mapsto x_1, f_2 \mapsto x_2\}} vs \to \texttt{case}\ f_1\ vs\ \{\ \texttt{Vfun}\ f \to f(f_2\ vs)\ \}$$
$$x_4 = \lambda^{\{v \mapsto \texttt{Vnum}\ 2\}} _ \to v$$
$$x_5 = \lambda^{\{f_1 \mapsto x_3, f_2 \mapsto x_4\}} vs \to \texttt{case}\ f_1\ vs\ \{\ \texttt{Vfun}\ f \to f(f_2\ vs)\ \}$$

Direct threading

Accès à l'environnement sans recherche

Indirection pour accéder à l'instruction suivante

Tags

Typage dynamique ⇒ 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: $\Gamma \vdash e : \tau$

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

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

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

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

Annotation de type nécessaire pour chaque argument

Annotations de types nécessaires pour définitions récursives

Grammaires attribuées

Grammaires annotées avec des règles de calculs d'*attributs*

- attributs *synthétisés*: e.g. AST, variables utilisées
- attributs *hérités*: e.g. contexte, variables disponibles

Le calculs des attributs correspond à une traversée récursive de l'arbre:

- attributs *hérités*: arguments passés aux appels récursifs
- attributs *synthétisés*: valeurs renvoyées par les appels récursifs

`yacc` : seul un attribut, synthétisé

Sous-typage

τ_1 et *sous-type* de τ_2 si $\Gamma \vdash e : \tau_1$ implique $\Gamma \vdash e : \tau_2$

Si *type* $\simeq$ *ensemble*, alors $\tau_1 \subset \tau_2$

Notions liées: classes, objets, héritage, interfaces, prototypes

If-typing: quel est le type de $\texttt{if}\ e\ e_{then}\ e_{else}$?

Casts: quand appliquer la règle de sous-typage?

Covariance/contravariance

Tests d'égalité de types remplacés par des tests de sous-typage

Vérification bidirectionnelle

Maximiser la propagation, minimiser la vérification

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

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

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

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:

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

Inférence de types: Hindley-Milner

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

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

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

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

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