/SU/FSI/MASTER/INFO/MU4IN503 (APS)

Analyse des Programmes et Sémantique

Janvier 2021

Pascal Manoury – Romain Demangeon
pascal.manoury@lip6.fr

1 : Principes du cours et Syntaxe

Conception de langages

Reprise et poursuite des idées de MU4IN501 (DLP)
dans un cadre formel

Lietmotiv : *syntaxe* + *typage* + *sémantique*

- **Syntaxe** Définition des chaînes de caractères admises comme *code source* des programmes
- **Typage** Définition du sous-ensemble des codes sources garantissant une cohérence des types
- **Sémantique** Définition du processus d'évaluation des codes sources

Règles

Les langages *APS*

APS0 Noyau fonctionnel (pur)
- Expressions, fonctions, expressions fonctionnelles
- Fonctions récursives, expressions alternative
- Types `int`, `bool`
- Fermetures, fermetures récursives, liaison statique

APS1 Ajout d'un noyau impératif
- Instructions, affectation, alternative, boucle
- Type `void` Procédures
- Mémoire, effet de bord

APS2 Structures de données
- Listes et tableaux

APS3 Fusion fonctionnel/impératif
- Rupture de contrôle, exceptions, continuations

Mise en œuvre des langages APS

Le cours édicte

1. les *régles* syntaxiques
2. les *règles* de typage
3. les *règles* sémantiques d'évaluation

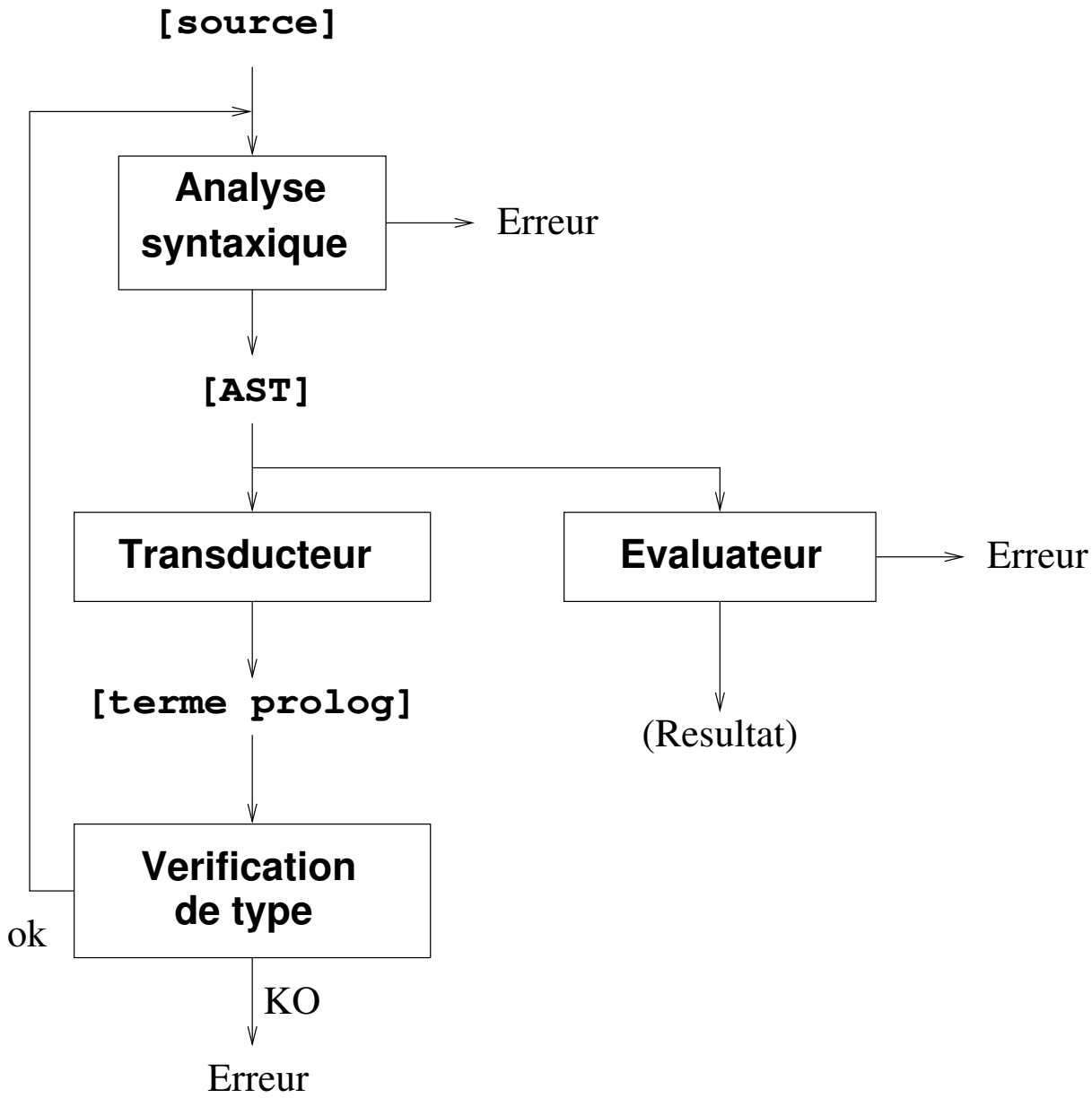
Les TP (TME) réalisent

1. les *analyseurs* lexicaux et syntaxiques
2. le *vérificateur* de types
3. *l'évaluateur* des programmes APS

Pas de «feuilles de TP»
Un «projet»[1] tout au long du semestre

1. Réalisé sur le créneau TME du mercredi matin

Schéma de principe



Principes de réalisation

Syntaxe Utilisation de *générateurs* d'analyseurs lexicaux et syntaxiques
Outils `lex/yacc` (préférentiel)
Langage support libre

Typage Implantation des règles de typage
Relations de typage guidées par la syntaxe
Langage imposé : Prolog

Sémantique Implantation des règles sémantiques d'évaluation
Fonction d'évaluation par récurrence sur l'AST
Langage support libre

APS0 informellement

Programme en *APS0*

- suite de définitions de constantes et de fonctions
- terminée par une instruction d'affichage

```
[
  CONST a int 42;

  FUN f int [x:int] (sub x a);

  FUN REC g bool [x:int]
    (if (lt x 0) false
        (if (eq x 0) true
            (g (f x))));

  ECHO (g 255)
]
```

Outil de formalisation

Théorie des langage

Définitions :

- Soit $\mathcal{A}$ un ensemble fini appelé *alphabet*
- Un *mot* sur $\mathcal{A}$ est une suite finie d'éléments de $\mathcal{A}$
- On note ε le mot vide (suite vide)
- On note $w_1 w_2$ la *concaténation* des mots w_1 et w_2
- On note $\mathcal{A}^*$ l'ensemble des mots sur $\mathcal{A}$
- On appelle *langage* un sous-ensemble de $\mathcal{A}^*$

Deux niveaux de ≪langages≫

- les ≪mots≫ : langage sur les caractères
  définit par des *expressions rationnelles*
- les ≪phrases≫ : langage sur les mots
  définit par des *grammaires*

Lexique (les «mots»)

Informellement

```
[
  # Exemple de programme
  CONST a int 42;

  FUN f int [x:int] (sub x a);

  FUN REC g bool [x:int]
    (if (lt x 0) false
        (if (eq x 0) true
            (g (f x))));

  ECHO (g 255)
]
```

- constantes `true`, 42
- fonctions prédéfinies `sub`, `eq`, `if`
- identificateurs `a`, `x`, `f`
- type `int`, `bool`
- mots réservés `CONST`, `FUN`, `ECHO`
- symboles réservés `;` `:` `(` `]`
- séparateurs *esp. tab. crlf*

Défini par expressions rationnelles

Expressions rationnelles

Soit $\mathcal{A}$, soit $O, 1 \notin \mathcal{A}$
On définit $\mathcal{E}(\mathcal{A})$

- $0, 1 \in \mathcal{E}(\mathcal{A})$
- tout $x \in \mathcal{A}$, $x \in \mathcal{E}(\mathcal{A})$
- si $e_1, e_2 \in \mathcal{A}$ alors $e_1 + e_2 \in \mathcal{A}$ et $e_1 \cdot e_2 \in \mathcal{A}$
- si $e \in \mathcal{A}$ alors $e* \in \mathcal{A}$

Interprétation $\mathcal{E}(\mathcal{A}) \to \mathcal{P}(\mathcal{A}^*)$

- $0 \mapsto \{\}$
- $1 \mapsto \{\varepsilon\}$
- $x \mapsto \{x\}$ pour tout $x \in \mathcal{A}$
- si $e_1 \mapsto X_1$ et $e_2 \mapsto X_2$ alors
  - $e_1 + e_2 \mapsto \{w \mid w \in X_1 \text{ ou } w \in X_2\}$ (union)
  - $e_1 \cdot e_2 \mapsto \{w_1 w_2 \mid w_1 \in X_1 \text{ et } w_2 \in X_2\}$ (produit)
  - $e_1* \mapsto \{w_1 \dots w_n \mid w_i \in X, \text{pour } i \in [1 \dots n]\}$ (étoile)

lex

Notation (ocamllex) pour les expressions rationnelles

L'alphabet est l'ensemble des caractères du clavier [2]

- un caractère (symbole) seul

  `','` `';'` `'('` etc.

- chaînes fixées

  `"true"` `"int"` `"FUN"` etc.

  ou encore `"->"`

- constantes numériques : suites de chiffres (possiblement précédée d'un '-')

  `'-'?['0'-'9']+`

- identificateurs : suites alpha-numériques (hors chaînes fixées) :

  `['a'-'z']['a'-'z''A'-'Z''0'-'9']*`

2. ou presque

Grammaire (les «phrases»)

Informellement

```
[
  # Exemple de programme
  CONST a int 42;

  FUN f int [x:int] (sub x a);

  FUN REC g bool [x:int]
    (if (lt x 0) false
        (if (eq x 0) true
            (g (f x))));

  ECHO (g 255)
]
```

- définition de constantes
  `CONST` nom, type, valeur
- définition de fonction
  `FUN` nom, type de retour, paramètres, corps
- définition récursive de fonction
  `FUN REC` *(idem)*
- instruction d'affichage
  `ECHO` valeur

Défini par une grammaire formelle (algébrique) comme la *concaténation* d'éléments du lexiques

Grammaire

Formellement

Soit un alphabet $\mathcal{A}$. Ses symboles sont appelés *terminaux*

Soit un ensemble $\mathcal{V}$ de symboles (disjoint de $\mathcal{A}$) appelés *non terminaux*

On distingue un élément $S \in \mathcal{V}$ appelé *axiome*

On se donne un ensemble de couples de $\mathcal{V} \times (\mathcal{A} \cup \mathcal{V})^*$ appelés *régles de production*.
On écrit $X \to \alpha$ avec $X \in \mathcal{V}$ et $\alpha \in (\mathcal{A} \cup \mathcal{V})^*$

La grammaire définit un *ensemble de mots* de $\mathcal{A}^*$ plus riche que ce que l'on pouvait faire avec les expressions rationnelles.

Automates à piles *vs* automates finis

Grammaire

Exemple

Les bons parenthèsages

Terminaux : $\mathcal{A} = \{ \ (\ ) \ \}$

Non terminaux : $\mathcal{V} = \{ \ \textsc{Par} \ \}$

L'axiome est $\textsc{Par}$

Les règles sont :

$$\begin{aligned} \textsc{Par} &\rightarrow () \\ \textsc{Par} &\rightarrow (\ \textsc{Par} \ ) \\ \textsc{Par} &\rightarrow \textsc{Par} \ \textsc{Par} \end{aligned}$$

$$\textsc{Par} \subset \{()\}^*$$

Remarque : définition *récursive*

Produire (() ((()) ()))

Suite d'applications de règle en partant de l'axiome

$$
\begin{aligned}
\textsc{Par} &\rightarrow \quad (\ \textsc{Par}\ ) \\
&\rightarrow \quad (\ \textsc{Par}\ \textsc{Par}\ ) \\
&\rightarrow \quad (\ ()\ \textsc{Par}\ ) \\
&\rightarrow \quad (\ ()\ (\ \textsc{Par}\ )\ ) \\
&\rightarrow \quad (\ ()\ (\ \textsc{Par}\ \textsc{Par}\ )\ ) \\
&\rightarrow \quad (\ ()\ (\ (\ \textsc{Par}\ )\ \textsc{Par}\ )\ ) \\
&\rightarrow \quad (\ ()\ (\ (\ ()\ )\ \textsc{Par}\ )\ ) \\
&\rightarrow \quad (\ ()\ (\ (\ ()\ )\ ()\ )\ )
\end{aligned}
$$

Plusieurs suites possibles pour un même mot

$$
\begin{aligned}
\textsc{Par} &\rightarrow \quad (\ \textsc{Par}\ ) \\
&\rightarrow \quad (\ \textsc{Par}\ \textsc{Par}\ ) \\
&\rightarrow \quad (\ \textsc{Par}\ (\ \textsc{Par}\ )\ ) \\
&\rightarrow \quad \text{etc.}
\end{aligned}
$$

Reconnaître (() ((()) ()))

Suite de *réductions* de règles ≪remontant≫ à l'axiome

$$
\begin{aligned}
\texttt{(() ((()) ()))} &\leftarrow \texttt{(() ((()) Par))} \\
&\leftarrow \texttt{(() ((Par) Par))} \\
&\leftarrow \texttt{(() (Par Par))} \\
&\leftarrow \texttt{(() (Par))} \\
&\leftarrow \texttt{(() Par)} \\
&\leftarrow \texttt{(Par Par)} \\
&\leftarrow \texttt{(Par)} \\
&\leftarrow \texttt{Par}
\end{aligned}
$$

Ici également : plusieurs suites possibles

$$
\begin{aligned}
\texttt{(() ((()) ()))} &\leftarrow \texttt{(Par ((()) ()))} \\
&\leftarrow \text{etc.}
\end{aligned}
$$

Ne pas reconnaître

$$
\begin{array}{lcl}
(\ () \ ) \ ) & \leftarrow & (\ \textsc{Par} \ ) \ ) \\
 & \leftarrow & \textsc{Par} \ ) \\
 & & \text{error}
\end{array}
$$

Aucune règle ne peut produire : $\textsc{Par} \ )$

syntax error

BNF

Syntaxe simplifiée pour la définition de grammaires :

$$
\begin{array}{lll}
\textsc{Par} & ::= & (\;) \\
& | & (\; \textsc{Par} \;) \\
& | & \textsc{Par} \; \textsc{Par}
\end{array}
$$

Des noms pour les *unités lexicales*

`LPAR` pour l'ouvrante (

`RPAR` pour la fermante)

$$
\begin{array}{lll}
\textsc{Par} & ::= & \texttt{LPAR RPAR} \\
& | & \texttt{LPAR} \; \textsc{Par} \; \texttt{RPAR} \\
& | & \textsc{Par} \; \textsc{Par}
\end{array}
$$

Un autre exemple

Des *S-expressions*[3]

On pose `ident` pour un identificateur quelconque
i.e. l'ensemble des $['a'-'z'][('a'-'z''A'-'Z''0'-'9')]*$

Non terminaux : SEXPR et SEXPRS

$$\begin{array}{lcl} \textsc{Sexp} & ::= & \texttt{ident} \\ & | & \texttt{LPAR}\ \textsc{Sexps}\ \texttt{RPAR} \\ \textsc{Sexps} & ::= & \varepsilon \\ & | & \textsc{Sexp}\ \textsc{Sexps} \end{array}$$

$$\textsc{Sexp} \subset (\{\texttt{LPAR}, \texttt{RPAR}\} \cup \texttt{ident})^*$$

`()` `(f x)` `(f x (g y))` `((f x) y)` `(f (g x) (g (y)))` ect.

3. Langage LISP

Les séparateurs

Analyse lexicale

Dissocier les *unités lexicales* dans un flot de caractères

- `(f x)` devient `LPAR` $\texttt{ident}_{(f)}$ $\texttt{ident}_{(x)}$ `RPAR`
- `(fx)` devient `LPAR` $\texttt{ident}_{(fx)}$ `RPAR`

Où commence et où finit une unité lexicale ?

- immédiat pour les caractères seul (comme une paranthèse)
- repérer des caractères *séparateurs* pour les suites de caractères (identificateurs, etc.)

Séparateurs usuels : espace, tabulation, retour à la ligne

séparateur $\approx$ mot vide

Programme *APS0*

Définition

Un *programme* c'est :

> *une suite définitions terminée par une instruction, le tout entre crochets*

Appelons (non terminaux)

- PROG l'ensemble des programmes
- DEFS l'ensemble des suites de définitions
- STAT l'ensemble des instructions (*statements*)

Appelons (terminaux)

- `LSQBR` le caractère `[`
- `RSQBR` le caractère `]`

On pose

PROG : := `LSQBR` DEFS STAT `RSQBR`

Autre définition

Un *programme*, c'est

une suite de commandes entre crochets

On pose

$$\textsc{Prog} \quad ::= \quad \texttt{LSQBR}\ \textsc{Cmds}\ \texttt{RSQBR}$$

où $\textsc{Cmds}$ dénote une suite de commandes

On pose

$$\begin{aligned} \textsc{Cmds} \quad ::= \quad & \textsc{Stat} \\ | \quad & \textsc{Def}\ \textsc{Cmds} \end{aligned}$$

où $\textsc{Def}$ dénote une *définition*

Grammaires et langage

Les grammaires

$$
\begin{array}{lcl}
\textsc{Prog} & ::= & \texttt{LSQBR}\ \textsc{Defs}\ \textsc{Stat}\ \texttt{RSQBR} \\
\textsc{Defs} & ::= & \textsc{Def} \\
 & | & \textsc{Def}\ \textsc{Defs}
\end{array}
$$

et

$$
\begin{array}{lcl}
\textsc{Prog} & ::= & \texttt{LSQBR}\ \textsc{Cmds}\ \texttt{RSQBR} \\
\textsc{Cmds} & ::= & \textsc{Stat} \\
 & | & \textsc{Def}\ \textsc{Cmds}
\end{array}
$$

définissent les *même langage*
(même ensemble de programmes)

LSQBR DEF ... DEF STAT RSQBR

L'instruction

L'ensemble des instructions est STAT
Il faut définir sa règle de grammaire

En $APS0$, une seule instruction :

affichage de la valeur d'une expression

Lexique
Choisir un *mot réservé* pour désigner l'instruction
On(je) choisit(s) : `ECHO`

Grammaire
Le mot `ECHO` suivi de l'expression

STAT : := `ECHO` EXPR
où EXPR (non terminal) désigne l'ensemble des *expressions*[4]

4. à définir !

Les définitions

Trois sortes de définitions[5] :

- définition de constante
- définition de fonction
- définition de fonction récursives

Lexique

- `CONST` pour les constantes
- `FUN` pour les fonctions
- `REC` si récursive

ici encore, arbitraire du concepteur

5. *Distingo* un peu arbitraire, voir plus tard, sémantique

Les constantes

Définition de

Trois informations nécessaires

- le nom de la constante (un *identificateur*)
- son type
- l'expression qui donne sa valeur

$$\textsc{Def} \quad ::= \quad \texttt{CONST ident } \textsc{Type } \textsc{Expr}$$

où le non terminal $\textsc{Type}$ représente les *expressions de type*

Expressions de type

Exemples

- `(int -> bool)` fonction des entiers dans les booléens
- `(int * bool -> int)` fonction à 2 parmaètres, un entier et un booléen, à valeur dans les entiers
- `((int -> bool) -> int)` fonction à un paramètre qui est une fonction des entiers dans les booléens, et qui est à valeur dans les entiers
- `(int -> (bool -> int))` fonction à un paramètre entier et qui a pour valeur une fonction des booléens dans les entiers
- `((int -> bool) * int -> bool)` fonction à 2 paramètres, une fonction des entiers dans les booléens et un entier, et qui est à valeurs dans les bolléens
- `((int -> bool) -> (int -> bool))` fonction à un paramètre, qui est une fonction des entiers dans les booléens, et qui a pour valeur une fonction des entiers dans les booléens

Expressions de types

Lexique

Deux types de base

1. les booléens : terminal `bool`,
2. les entiers : terminal `int`

Deux *constructeurs* de type

- le produit[6] : terminal `*` nommé `STAR`
- la flèche fonctionnelle : terminal `->` nommé `ARROW`

On utilisera également

- les parenthèses `(` et `)` nommées `LPAR` et `RPAR`

6. Pour les paramètres des fonctions.

Expression de type

Grammaire

Un type (non terminal TYPE) est

1. soit un type de base
2. soit un type de fonction constitué de
   - la suite de type des paranètres (non terminal TYPES)
   - le type de «retour» de la fonction

On pose `tprim` = { `bool`, `int` } (types de base)

On pose

TYPE	: :=	`tprim`
	|	`LPAR` TYPES `ARROW` TYPE `RPAR`
TYPES	: :=	TYPE
	|	TYPE `STAR` TYPES

Définitions mutuellement récursives

Les fonctions

Définition de

Quatre informations nécessaires

- le nom de la fonction (identificateur)
- le type de «retour» de la fonction (expression de type)
- les paramètres de la fonctions (enclos entre crochets)
- le *corps* de la fonction (expression)

On pose

$$\textsc{Def} \quad ::= \quad \ldots$$
$$\quad | \quad \texttt{FUN ident } \textsc{Type } \texttt{LSQBR } \textsc{Args } \texttt{RSQBR } \textsc{Expr}$$

où $\textsc{Args}$: non terminal pour la listes de paramètres

Paramètres

Pour chaque paramètre

- son nom (identificateur)
- son type (expression de type)

Nom et type séparés par deux points
(terminal `:` nommé `DDOTS`)

Arguments séparés par une virgule
(terminal `,` nommé `COMMA`)

On pose

ARGS	: :=	ARG
	|	ARG `COMMA` ARGS
ARG	: :=	`ident` `DDOTS` TYPE

Les fonctions récursives

Comme les fonctions + mention de `REC`

$$\begin{array}{lll} \textsc{Def} & ::= & \ldots \\ & | & \texttt{FUN REC ident } \textsc{Type} \texttt{ LSQBR } \textsc{Args} \texttt{ RSQBR } \textsc{Expr} \end{array}$$

Motivation : *liaison statique* (choix de conception)

Dans `FUN f int [x:int] (g (f x))`
le `f` de `(g (f x))` **n'est pas**
le `f` de la définition (`FUN f ...`)

Dans `FUN REC f int [x:int] (g (f x))`
le `f` de `(g (f x))` **est**
le `f` de la définition (`FUN REC f ...`)

Les expressions

Une *expression* (non terminal EXPR) est soit

- une constante numérique ou booléenne
- un identificateur (paramètre, constante définie, nom de fonction)
- une *application* de fonction ou opérateur à des valeurs données par des *expressions*
  Modèle fonctionnel : la «fonction» appliquée peut-être une *expression*
- une *expression fonctionnelle* appelée *abstraction* composé de
  - une liste de paramètres (*cf* ARGS)
  - un *corps* qui est une *expression*

Expressions atomiques

Non composées, non décomposables

- constantes booléenne, terminal `cbool` $= \{$ `true` `false` $\}$
- constantes numérique, terminal `num` $= '-'?['0'-'9']+$
- identificateurs, terminal `ident`

On pose

$$
\begin{array}{lll}
\textsc{Expr} & ::= & \texttt{cbool} \\
 & | & \texttt{num} \\
 & | & \texttt{ident}
\end{array}
$$

Les applications

Informellement

Choix de conception (simplicité)

Préfixées complètement parenthésées

cf SEXPR

Exemples

- `(and (not x) true)`
- `(eq (add x 5) (mul y x))`
- `(if (lt x 0) (add x b) x)`
- `(f (mul x 3) (g x))`
- `(add x (if (eq (f y) 0) 1 0))`

Mais aussi (modèle fonctionnel)

- `( (f x) y )`
- `( [x:int](add x 3) y)`
- `( [x:int][y:int](add x y) 5 )`

Les applications de symboles atomiques

On veut $(x\ \ e_1\ \ \ldots e_n)$ avec x

- opérateur primitif :
  opérateurs arithmétiques, comparaison, etc.
- opérateur de contrôle :
  alternative (indispensable), etc.
- nom de fonction définie

Choix de conception : *distinguer ou ne pas distinguer*

Conséquence

1. *distinguer* complexifie la définition du lexique et de la grammaire (plus de cas à considérer lors de l'analyse)
2. *ne pas distinguer* simplifie la définition du lexique et de la grammaire mais reporte le problème (distinguer les cas dans la sémantique)

Les applications de symboles atomiques

Suite

Autre choix : *utiliser ou non des symboles*
(+ pour l'addition, = pour l'égalité)

Conséquence :

- si symboles alors distinction syntaxique nécessaire des opérateurs primitifs

Décision

- pas de critère de choix clair
- laissée à votre appréciation

MAIS (*oukase*)[7]

pas de symboles

7. Fiacilitation des tests

APS0 : le lexique complet

Symboles réservés

```
[ ] ( ) ; : , *
->
```

Mots réservés

```
CONST FUN REC
```
(définitions)

```
ECHO
```
(instruction)

```
bool int

true false not
eq lt add sub mul div
if and or
```

Identificateurs `['a'-'z']['a'-'z''A'-'Z''0'-'9']*`
(sauf les mots clef) (*ident*)

Constantes numériques `'-'?['0'-'9']+` (*num*)

Séparateurs `[' ' '\t' '\n']`

Les applications

Grammaires

Définition récursive

- si e, e_1, ...,e_n sont des expressions alors $(e\ \ e_1\ \ \ldots e_n)$

Une application est une *expression* suivie d'une suite d'expressions, le tout entre parenthèses

On pose : avec nos choix de simplification

$$
\begin{array}{lll}
\textsc{Expr} & ::= & \ldots \\
 & | & \texttt{LPAR}\ \textsc{Expr}\ \textsc{Exprs}\ \texttt{RPAR} \\
\textsc{Exprs} & ::= & \textsc{Expr} \\
 & | & \textsc{Expr}\ \textsc{Exprs}
\end{array}
$$

Lexique :

`true false not eq lt add sub mul div if and or`

considérés comme *identificateurs*

Les abstractions

Expressions fonctionnelles

Faire de l'expression `(add x 1)` une fonction de `x`

Abstraire `x`, le distinguer comme *paramètre* (typé)

Recycler la notation des paramètres dans les définitions de fonction

On pose

$$\begin{array}{lll} \textsc{Expr} & ::= & \ldots \\ & | & \texttt{LSQBR}\ \textsc{Args}\ \texttt{RSQBR}\ \textsc{Expr} \end{array}$$

Variation sur les expressions fonctionnelles

On distingue :

1. `[x:int, y:int](add x y)`
   fonction à 2 arguments dont le résultat est un entier
2. `[x:int][y:int](add x y)`
   fonction à 1 argument dont le résultat est une fonction à 1 argument dont le résultat est un entier

Application

1. `([x:int, y:int](add x y) 4 2)`
2. `( ([x:int][y:int](add x y) 4) 2 )`

Types

1. `[x:int, y:int](add x y)`
   est de type `(int * int -> int)`
2. `[x:int][y:int](add x y)`
   est de type `(int -> (int -> int))`

Variation sur les définitions

On distingue

- `FUN f1 int [x:int, y:int](add x y)`
- `FUN f2 (int -> int) [x:int] [y:int](add x y)`

Remarque (expressions fonctionnelles)

- `CONST f1 (int * int -> int) [x:int, y:int] (add x y)`
- `CONST f2 (int -> (int -> int)) [x:int][y:int](add x y)`

`f2` est la *curryfication* de `f1`

Application «partielle»

- `(f2 4)` est possible (de type `(int -> int)`)
- `(f1 4)` n'est pas autorisé
  sera refusé au typage

APS0 : une grammaire complète

Prog	: :=	`[` Cmds `]`
Cmds	: :=	Stat
	|	Def `;` Cmds
Stat	: :=	`ECHO` Expr
Def	: :=	`CONST ident` Type Expr
	|	`FUN ident` Type `[` Args `]` Expr
	|	`FUN REC ident` Type `[` Args `]` Expr
Type	: :=	`tprim`
	|	`(` Types `->` Type `)`
Types	: :=	Type
	|	Type `*` Types
Args	: :=	Arg
	|	Arg `,` Args
Arg	: :=	`ident :` Type
Expr	: :=	`cbool`
	|	`num`
	|	`ident`
	|	`(` Expr Exprs `)`
	|	`[` Args `]` Expr
Exprs	: :=	Expr
	|	Expr Exprs