1er Examen Réparti
Analyse des Programmes et Sémantique

(APS - MI030 - Master 1)
Université Pierre et Marie Curie Paris 6
Cours de J.Malenfant enseigné par B.Starynkevitch
lundi 25 février 2013

Attention : *Tous les documents sur papier sont autorisés* ; mais *tous les* ***appareils électroniques*** (téléphones, tablettes, liseuses, ordinateurs, baladeurs, écouteurs, etc...) *sont* ***interdits***.

Durée : deux heures ; (sujet de 4 pages sur 2 feuilles recto-verso)

Les questions (§1 au total notées sur 10) sont toutes indépendantes du problème §2 (noté sur 20).

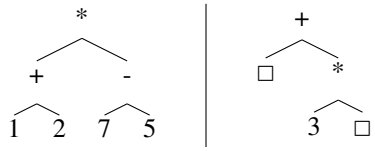
N.B. Les couleurs dans cet énoncé sont là pour vous faciliter la lecture. Ne les utilisez pas sur votre copie. Si vous utilisez deux couleurs distinctes, indiquez vos conventions. N'utilisez pas le rouge.

1 questions

1.1 à propos d'arbres

Concernant les arbres d'expressions arithmétiques complètement parenthésées :

1. à quelle expression correspond à l' arbre de gauche ci-dessous ? En quel nombre s'évalue-t-elle ?
2. donner deux expressions (la première plus simple que la seconde) qui correspondent à l'arbre avec trous □ de droite



1.2 un peu de Prolog

1.2.1 question simple sur l'unification

Dans la table suivante, dire si oui ou non les termes (à gauche Γ, à droite Δ) suivants s'unifient et dans l'affirmative donner les substitutions. Autrement dit que répond l'interprète Prolog (ou ECLiPSe-CLP) à la question (au but) Γ = Δ avec :

	Γ	Δ
a.	`2+3`	`5`
b.	`[X,Y]`	`[2]`
c.	`[X,3,5]`	`[2|Y]`
d.	`[X+2,5,7,9]`	`[1+Y|_]`
e.	`[X,X]`	`[6,Y,a]`

Le code Prolog demandé dans les deux questions ci-dessous est très petit, moins de 5 lignes par question.

1.2.2 extraction des pairs

On suppose qu'on ait deux prédicats `pair/1` (resp. `impair/1`) qui testent si leur argument est un nombre pair (resp. impair), par exemple avec le code :

```
impair(X) :- number(X), 1 is X rem 2.
pair(X) :- number(X), 0 is X rem 2.
```

Remarque : on aurait pu définir `impair` par
`impair(X) :- number(X), Y is X - 1, pair(Y).`

Donner en quelques lignes les clauses Prolog pour le prédicat binaire `liste_impair/2` qui extrait de la liste donnée en premier argument en unifiant le deuxième argument à la liste des nombres impairs extraite de cette première liste. Ainsi le but **`liste_impair([1,2,3,4,16,17],X).`** réussit en donnant `X = [1, 3, 17].`

1.2.3 saute-mouton

On traite des listes composées de noms atomiques ou bien de termes `saute(`n`)` avec n entier naturel, par exemple `[u,saute(2),a,b,c,d,e]`. Ecrire le code Prolog pour le prédicat binaire `mouton/2` qui parcourt cette liste, traite les `saute(`n`)` en sautant les n éléments suivants de la liste, et conserve les autres éléments. Ainsi, **`mouton([u,saute(2),a,b,c,saute(1),d,e],X).`** réussit en donnant `X = [u,c,e].`

2 problème - sémantique opérationnelle structurelle

On considère le mini-langage *Bidulic* de calculette programmable ci-dessous. Les valeurs manipulées sont entières. Toutes les variables sont entières et initialisées à 0. Un programme comprend des variables d'entrée `inputs`, des variables globales `globals`, une séquence de fonctions (qui pourraient être récursives croisées), un corps et calcule un résultat noté `result`. L'exécution d'un programme prend une séquence de nombres en entrée (liés aux `inputs`) et produit - quand elle se termine - un nombre en sortie. Les noms identifiants des entrées *Inputs*, des globaux *Globals* et des fonctions sont tous distincts.

Les expressions du langage sont assez usuelles. Toutefois le `result` y designe le résultat de la fonction en cours de definition ou du programme principal et peut apparaître aussi en membre gauche des affectations ; ce résultat est initialisé à 0. Les expressions comporte aussi l'opérateur de choix indeterministe `amb` de McCarthy (inventé en 1961, où `amb` suggère l'ambigüité !) : l'expression e_1 `amb` e_2 est inderministe et peut évaluer e_1 ou évaluer e_2 ; ainsi `1 amb (2+3)` peut s'évaluer en 1 ou bien en 5. Les expressions comportent aussi des conjonctions e_1 `andthen` e_2 et des disjonctions e_1

`orelse` e_2 **paresseuses** [1] qui n'évaluent pas forcément leur sous-expression de droite e_2. Le nombre 0 est compris comme faux, et tout autre nombre non-nul est compris comme vrai. L'application de fonction s'écrit classiquement, par exemple `foo(n*2,3)` ou `bar()` etc. Un identificateur ***id*** suit les conventions lexicales usuelles (formé de lettres, mais doit être distinct des mots-clefs). Les nombres ***num*** sont des entiers en décimal. Attention, les expressions n'ont pas de soustraction binaire, et la division `6/x` peut échouer quand `x` vaut 0 car il n'est pas permis de diviser par 0. L'opérateur `not` est la négation *logique* donc `not 0` $\equiv 1$, `not` $n \equiv 0$ si $n \neq 0$. Le passage d'argument se fait par valeur (comme en Java, C, Ocaml) et les formels sont considérés comme des variables locales (comme en Java ou C) donc peuvent être à gauche d'une affectation.

La grammaire concrète des expressions de notre langage *Bidulic* est donc :

Exp	: :=	*Term* | *Term* **amb** *Exp* | *Term* **orelse** *Exp* | *Term* **+** *Exp*
Term	: :=	*Fact* | *Term* **andthen** *Fact* | *Term* ***** *Fact* | *Term* **/** *Fact*
Fact	: :=	*Primary* | **-** *Primary* | **not** *Primary*
		| *Primary* **=** *Primary* | *Primary* **<** *Primary*
Primary	: :=	***id*** | ***num*** | **result** | ***id*** **(** *Arguments* **)** | **(** *Exp* **)**
Arguments	: :=	ε | *Exp* | *Exp* **,** *Arguments*

Voici la grammaire concrète concernant les programmes, fonctions, et instructions de *Bidulic*, en plus des expressions définies ci-dessus. On peut calculer une expression pour ses effets de bords seuls via `compute`.

Program	: :=	**program** *Inputs Globals Functions Body*
Inputs	: :=	ε | **inputs** *Ids*
Globals	: :=	ε | **globals** *Ids*
Ids	: :=	***id*** | ***id*** **,** *Ids*
Functions	: :=	ε | **function** *FunctionDef Functions*
FunctionDef	: :=	***id*** **(** *Ids* **)** **is** *Vars Body*
Vars	: :=	ε | **vars** *Ids*
Body	: :=	**begin** *InstSeq* **end**
InstSeq	: :=	ε | *Instr* | *Instr* **;** *InstSeq*
Instr	: :=	*Left* **:=** *Exp* | **compute** *Exp*
		| **while** *Exp* **do** *Body* | **if** *Exp* **then** *Body* **else** *Body*
Left	: :=	***id*** | **result**

Voici la **grammaire abstraite** de *Bidulic* :

1. La conjonction e_1 `andthen` e_2 de *Bidulic* est semblable au `(and` e_1 e_2`)` d'un Scheme ou d'un Lisp, ou au e_1 `&&` e_2 de Java ou *C* ou *Ocaml*. De même la disjonction e_1 `orelse` e_2 de *Bidulic* est semblable au `(or` e_1 e_2`)` d'un Scheme ou d'un Lisp, ou au e_1 `||` e_2 de Java ou *C* ou *Ocaml*.

$$
\begin{array}{rcl}
\mathit{program} & ::= & \textbf{prog}\ \mathit{id_{inp}}^*\ \mathit{id_{glo}}^*\ \mathit{fundef}^*\ \mathit{inst}^* \\
\mathit{fundef} & ::= & \textbf{fun}\ \mathit{id_{fun}}\ \mathit{id_{for}}^*\ \mathit{id_{var}}^*\ \mathit{inst}^* \\
\mathit{inst} & ::= & \textbf{assign}\ \mathit{left}\ \mathit{exp} \mid \textbf{compute}\ \mathit{exp} \\
 & & \textbf{while}\ \mathit{exp}\ \mathit{inst}^* \mid \textbf{if}\ \mathit{exp_{test}}\ \mathit{inst_{then}}^*\ \mathit{inst_{else}}^* \\
\mathit{left} & ::= & \mathit{id} \mid \textbf{result} \\
\mathit{exp} & ::= & \textbf{num}\ n \mid \mathit{left} \mid \textbf{amb}\ \mathit{exp_1}\ \mathit{exp_2} \mid \textbf{orelse}\ \mathit{exp_1}\ \mathit{exp_2} \mid \textbf{plus}\ \mathit{exp_1}\ \mathit{exp_2} \\
 & & \mid \textbf{andthen}\ \mathit{exp_1}\ \mathit{exp_2} \mid \textbf{mul}\ \mathit{exp_1}\ \mathit{exp_2} \mid \textbf{div}\ \mathit{exp_1}\ \mathit{exp_2} \\
 & & \mid \textbf{equal}\ \mathit{exp_1}\ \mathit{exp_2} \mid \textbf{less}\ \mathit{exp_1}\ \mathit{exp_2} \mid \textbf{not}\ \mathit{exp} \mid \textbf{neg}\ \mathit{exp} \\
 & & \mid \textbf{apply}\ \mathit{id_{fun}}\ \mathit{exp}^*
\end{array}
$$

2.1 un exemple

Donner en quelques lignes le code - en syntaxe concrète - d'un programme qui calcule la factorielle $n!$ de l'entrée `n` en ayant une fonction récursive `fact` dont le formel est `i`.

2.2 état sémantique

Définir les éléments de l'état du programme et la signature des relations $\rightarrow$ qu'il faut pour donner la sémantique operationnelle structurelle du language *Bidulic*. Précisez bien la nature (les types) de chaque élément.

2.3 sémantique operationnelle structurelle

Définir la sémantique operationnelle structurelle de *Bidulic*.

Notez bien que **amb** est non-déterministe, **div** ne peut pas diviser par 0, et **andthen** et **orelse** sont paresseux.

2.4 maxi-bonus : extensions de language

N'abordez cette question que si les questions précédentes du problème ont été traitées.

Que faudrait-il modifier pour introduire une instruction de répétition et une instruction de boucle ascendante, de syntaxe concrète :

$$
\begin{array}{rcl}
\text{Instr} & ::= & \ldots \mid \textbf{repeat}\ \mathit{Body}\ \textbf{until}\ \mathit{Exp} \\
 & & \mid \textbf{for}\ \mathit{Left}\ \textbf{below}\ \mathit{Exp}\ \textbf{do}\ \mathit{Body}
\end{array}
$$

L'instruction **repeat** a le sens usuel ; la boucle **for** évalue une seule fois sa borne *Exp* (en un entier n_{borne}) et itère avec une variable d'induction *Left* locale ou globale (ou le `result`) en l'affectant de 0 jusqu'à $n_{borne} - 1$ pour son corps *Body* ; au sortir de la boucle sa variable d'induction vaut n_{borne}.
