RICM1 - 2004/05

Langage et Programmation 2

TP2 : Expressions arithmétiques

Il s'agit du premier TP sur plusieurs séances (3 ou sont 4 prévues) **à rendre à la fin**.

L'objectif final est de parvenir à un programme qui permet d'évaluer des expressions. L'écriture d'un tel programme se fait en quatre étapes :

1. La définition du type de la structure de données représentant les expressions
2. Un analyseur lexical
3. Un analyseur syntaxique
4. La fonction *évaluation* proprement dite

On ne traitera que des expressions bien formées.

On procèdera incrémentalement, en plusieurs phases.

- expressions additives (vues en TD)
- expressions arithmétiques
- introduction d'expressions booléennes
- introduction de let-expressions

Dans un premier temps on considèrera donc les expressions additives puis arithmétiques. Si, par exemple, on donne la chaîne de caractères "$4 - 2 * 3$ ", le programme devra répondre 6. On supposera qu'une expression est au moins composée d'un entier.

Phase 1 : expressions additives

À rendre avant lundi 7

Ce ne sera pas noté, mais il s'agit d'éviter d'incrémenter sur une éventuelle mauvaise base.

L'analyseur lexical

Le travail de l'analyseur est de prendre une chaîne de caractère et de la

transformer en liste de lexèmes (tokens) pour qu'elle puisse être traitée par l'analyseur syntaxique. Pour réaliser l'analyseur lexical, vous aurez besoin de :
- Une fonction intermédiaire qui transforme la chaîne de caractères en liste de caractères.
- Le schéma de Horner.
- La définition d'un type *token* qui définit chaque symbole des expressions arithmétiques (TPlus|...).

L'expression "$4-2+3$ " sera alors tranformée en la liste
[$TEnt$ 4 ; $TMoins$; $TEnt$ 2 ; $TPlus$; $TEnt$ 3]

Le type des expressions

Comme on l'a vu en TD, la structure de représentation des expressions arithmétiques la plus appropriée est la structure d'arbre. L'expression "$4-2+3$ " sera alors représentée par l'arbre
Plus(Moins(Int 4, Int 2),Int 3).

$type\ expr\ =$
$|\ Int\ of\ int$
$|\ Plus\ of\ expr\ *\ expr$
$|\ Moins\ of\ expr\ *\ expr$

L'analyseur syntaxique

L'analyseur syntaxique doit transformer la liste de tokens en un arbre de type *expr*. Par exemple, la liste
[$TEnt$ 4 ; $TMoins$; $TEnt$ 2 ; $TPlus$; $TEnt$ 3] doit être transformée en l'arbre Plus(Moins(Int 4, Int 2),Int 3).
Pour pouvoir contruire l'arbre, vous aurez besoin de la grammaire des expressions arithmétiques. On a vu en TD la grammaire pour les expressions composées d'entiers, de $'+'$ et de $'-'$.

Grammaire

E = expression
T = terme
SA = suite d'additions
F = facteur
SM = suite de multiplications

```
E ::= T SA
SA ::= '+' SA | '-' SA | ϵ
T ::= F SM
SM ::= '*' SM | '/' SM | ϵ
F ::= entier | '(' E ')'
```

L'évaluation

La fonction d'évaluation correspond à un parcours de l'arbre :
$let\ rec\ eval\ e\ =\ match\ e\ with$
$|\ Int(n) -> n$
$|\ Plus(e1, e2)\ \rightarrow eval\ e1\ +\ eval\ e2$
$|\ Moins(e1, e2)\ \rightarrow eval\ e1\ -\ eval\ e2$

Phase 2 : expressions arithmétiques

Ne pas se sentir obligé d'attendre la seconde séance pour démarrer cette phase...

L'analyseur lexical

Introduire des lexèmes pour les opérateurs de multiplication, division et pour les parenthèses.

Le type des expressions

L'expression "$(4-2)*3$ " sera alors représentée par l'arbre Mult(Moins(Int 4, Int 2),Int 3).

$type\ expr\ =$
$|\ Int\ of\ int$
$|\ Plus\ of\ expr\ *\ expr$
$|\ Moins\ of\ expr\ *\ expr$
$|\ Mult\ of\ expr\ *\ expr$
$|\ Div\ of\ expr\ *\ expr$

L'analyseur syntaxique

L'analyseur syntaxique doit transformer la liste de tokens en un arbre de type *expr*. Par exemple, la liste
$[TParou\ ;\ TEnt\ 4\ ;\ TMoins\ ;\ TEnt\ 2\ ;\ TParferm\ ;\ TMult\ ;\ TEnt\ 3]$
doit être transformée en l'arbre Mult(Moins(Int 4, Int 2),Int 3).

Il faudra donc améliorer la grammaire précédente (en veillant à ce qu'elle reste récursive à droite) pour qu'elle prenne en compte les parenthèses, la multiplication et la division.

L'évaluation

La fonction d'évaluation correspond à un parcours de l'arbre.

Phase 3 : expressions booléennes

Reproduire la démarche ci-dessus pour des expressions purement booléeenes (sur les deux constantes notées `true` et `false`).

Il faut donc compléter l'analyseur lexical en introduisant un constructeur pour les identificateurs (suites de lettres, éventuellement suites de lettres et de chiffres commençant par une lettre).

Les deux seuls identificateurs réellement utilisés à ce stade sont donc `true` et `false`, mais d'autres apparaîtront par la suite.

Phase 4 : expressions arithmétiques et booléennes

Introduire une expression booléenne pour le test d'égalité entre 2 expressions arithmétiques.

Introduire une construction `si`... `alors`... `sinon` dans les expressions arithmétiques.

Grammaire

E = expression
C = conjonction
SC = suite de conjonctions
SD = suite de disjonctions
L = littéral
EC = expression comparable
CMP = comparaison
T = terme
SA = suite d'additions

F = facteur
SM = suite de multiplications

```
E ::= 'si' E 'alors' E 'sinon' E | C SD
SD ::= '∨' C SD | ϵ
C ::= L SC
SC ::= '∧' L SC | ϵ
L ::= '¬' L | EC CMP
CMP = '=' CMP | ϵ
EC ::= T SA
SA ::= '+' SA | '-' SA | ϵ
T ::= F SM
SM ::= '*' SM | '/' SM | ϵ
F ::= entier | 'vrai'  | 'faux' | '(' E ')'
```

Phase 5

Introduire une construction `let`... `in`...

Il faut prévoir une notion d'identificateur dans les lexèmes et une notion de variable dans les expressions.

Les *modifications* à apporter dans la grammaire sont :

```
E ::=
    'si' E 'alors' E 'sinon' E
  | 'soit' ident '=' E 'dans' E
  | C SD
F ::= entier | 'vrai'  | 'faux' | ident | '(' E ')'
```

Pour l'évaluation, il faut introduire une notion d'environnement.

Bonus

Introduire une construction `fun`... `->`... (au niveau E) ainsi qu'une construction d'application d'une fonction à des arguments (au niveau facteur)

Super bonus

Introduire une construction `let rec`... `in`...