HAUTE ÉCOLE
D'INGÉNIERIE ET DE GESTION
DU CANTON DE VAUD
www.heig-vd.ch

Interpréteur

LABORATOIRE 9-10-11

Forestier Quentin & Herzig Melvyn

PLP – 12.12.2020

Grammaire

Pour définir notre version de SPL nous avons dû définir une grammaire à notre langage. Voici sa structure :

prg → { def } expr

def → **define** funName { var } = expr

expr → **let** var = expr **in** expr
| **if** expr **then** expr **else** expr
| expr (**+**|**-**|*****|**/**|**<**) expr
| **(** (+|-) expr **)**
| # expr
| integer
| varName
| funName **(** {expr} **)**
| **(** expr **)**

varName -> (a-z){A-Za-z0-9}

funName -> (A-Z){A-Za-z0-9}

integer -> (0-9){0-9}

Symbole de départ :
prg

Ensemble des symboles non terminaux :
Prg
Def
Expr
VarName
funName
integer

Ensemble des symboles terminaux :

[A-Z]	let
[a-z]	in
[0-9]	if
+ - * / = < #	then
( )	else
	define

Parser

Conformément à notre grammaire, le parser retournait un Prg formé de 0 ou plusieurs définitions de fonctions et forcément une instruction. Pour faire fonctionner l'interpréteur nous avons modifié un Prg pour être soit une expression soit une définition de fonction.

```
-- Prg : Expr   { Prg [] $1}
-- | FDef Prg  { let Prg defs expr = $2 in Prg ($1:defs) expr}

Prg : Expr   { ExprSimple $1 }
 | FDef { DefSimple $1 }

-- data Prg  = Prg [FDef] Exp deriving Show
data Prg  = ExprSimple Exp | DefSimple FDef deriving Show
```

Fonctionnement

Lancement de l'interpréteur. La fonction launchRepl construit les fonctions par défauts et lance la boucle REPL.

```
Prelude> :l interpreter.hs
*Interpreter> launchRepl
SPL>
```

Test de définition d'une fonction.

Pour une simple fonction à deux paramètres qui les additionnes :

```
SPL>define Add a b = a+b
[TDefine,TFunName "Add",TVarName "a",TVarName "b",TSym '=',TVarName "a",TSym '+',TVarName "b"]
DefSimple (FDef "Add" ["a","b"] (Bin '+' (Var "a") (Var "b")))
Add
```

Les lexèmes obtenus ainsi que l'arbre syntaxique sont corrects.

Tests d'expressions

```
SPL>let a = 8 in Add(#a 5)
[TLet,TVarName "a",TSym '=',TInt 8,TIn,TFunName "Add",TOp,TSym '#',TVarName "a",TInt 5,TCp]
ExprSimple (Let "a" (Cst 8) (FApp "Add" [Una '#' (Var "a"),Cst 5]))
85
```

Cette expression est correctement traduite et interprétée. Nous pouvons y voir notre opérateur personnalisé '#' qui, préfixé à une expression, multiplie par 10. L'appel de fonction est fonctionnel.

```
SPL>Fact (8)
[TFunName "Fact",TOp,TInt 8,TCp]
ExprSimple (FApp "Fact" [Cst 8])
40320
```

L'utilisation de la fonction factorielle est implémentée et fonctionnelle.

Comme fonction personnalisée nous avons défini qu'elle calculait l'air d'un rectangle. Si un des deux paramètres est inférieur à 0 elle retourne -1 sinon l'air issu des deux paramètres.

```
SPL>Surface((-5) 4)
[TFunName "Surface",TOp,TOp,TSym '-',TInt 5,TCp,TInt 4,TCp]
ExprSimple (FApp "Surface" [Una '-' (Cst 5),Cst 4])
-1
SPL>Surface(4 5)
[TFunName "Surface",TOp,TInt 4,TInt 5,TCp]
ExprSimple (FApp "Surface" [Cst 4,Cst 5])
20
```

Conclusion

Au terme de ce laboratoire, notre version de SPL est correctement interprétée. Les opérateurs unaires qui sont également binaires doivent être couplés à des parenthèses pour fonctionner (p.ex : *(-4)*). Les appels de fonction sans paramètres demandent tout de même l'utilisation de parenthèses vides. Si un nombre de paramètres incorrect est fourni, une exception est levée. Notre langage fonctionne uniquement avec des expressions fermées. Si une variable demandée n'existe pas pour l'expression courante, une exception est levée.