Rapport de APS

Analyse des Programmes et Sémantique

Author :
Qiwei XIAN

Professeur :
Pascal.manoury

18 mai 2020

Table des matières

1 APS0 **2**

1.1 Etape1. Analyse lexicale . . . 2
1.2 Etape2. Analyse syntaxique . . . 2
1.2.1 Expr . . . 2
1.2.2 Type et Arguments . . . 3
1.2.3 Statement . . . 3
1.2.4 Déclaration . . . 3
1.2.5 Commande . . . 3
1.2.6 Programme . . . 4
1.3 Etape3. Typage . . . 4
1.3.1 Expression . . . 4
1.3.2 Déclaration . . . 4
1.3.3 Statement . . . 4
1.3.4 Commande . . . 4
1.3.5 Program . . . 4
1.4 Etape4. Analyse sémantique . . . 5
1.4.1 Définition de Value . . . 5
1.4.2 Implémante structure de l'environement . . . 5
1.4.3 Expr . . . 5

1 APS0

1.1 Etape1. Analyse lexicale

Pour l'étape d'analyse lexicale, j'ai créé les associations entre les mots clefs qui peuvent être utilisé dans APS0 avec les **tokens** qui sont définits dans **parser.y**.
Après cet étape, l'interprèteur de APS0 reconnaît tous les mots clefs comme les opérations de primitives, les chiffres, le booléan, les identifiants, les parenthèses, etc. Mais il ne connaît pas la phrase, parce que l'on n'a pas encore définit la grammaire.

1.2 Etape2. Analyse syntaxique

1.2.1 Expr

Expr est la base du langage, les autres structures sont composées par le mot clef et expression, donc je commence par réaliser **Expr**.

1. Grammaire. Dans cet étape, on ajoute la grammaire des expressions dans **parser.y**. Une grammaire est une suite des tokens, donc c'est juste de composer les tokens dans cet étape.

```
expr:
NUM                         { $$ = newASTNum($1); }
| bool                      { $$ = newASTBool($1); }
| IDENT                       { $$ = newASTId($1); }
| LPAR oprim exprs RPAR     { $$ = newASTPrim($2,$3); }
| LPAR IF expr expr expr RPAR {$$ = newASTIf($3, $4, $5);}
| LCO args RCO expr {$$ = newASTLambda($2, $4);}
| LPAR exprs RPAR {$$ = newASTBloc($2); }
;
```

(a) grammar of expression

2. AST. C'est une arbre d'analyse syntaxique. Dès qu'on a écrit la grammaire des expressions, l'interprèteur connaît la phrase, mais il ne fait rien. S'il veut la traiter, il a besoin de AST. Donc j'ai définit le noeud de AST **Expr** dans **ast.h** et réaliser les fonctions de constructeurs **newASTNum**, **newASTId**, **newASTBool**, **newASTPrim**, **newASTIf**, **newASTLambda**, **newASTBloc** et **appendExprs** dans **ast.c**. Puis appeler ces constructeurs dans **parser.y** dès qu'on a lu une phrase d'expression.

3. Prolog. Après construit AST, l'interprèteur peut le parcourir. Pour cet étape, on veut juste afficher AST d'expression en sortie standard. Donc il faut juste écrire la fonctions d'affichage pour chaque noeud de AST dans **prologTerm.c**.

1.2.2 Type et Arguments

Type et **Arguments** est aussi bas niveau, donc je les réalise just après **Expr**.

1. Grammair. On ajoute la grammair pour comment exprimer un type dans le langage APS0. Dans notre langage, le type peut exprimé récursivement. Donc j'ai écrit la grammair comme l'image.
2. AST. Ajouter les structures **Type**, **Types**, **Arg** et **Args** dans **ast.h**, ainsi que les constructeur **newASTType** et **newASTArg** ainsi que **appendTypes** et **appendArgs** dans **ast.c**
3. Prolog. Ajoute les fonctions d'affichage **printType**, **printTypes**, **printArg** et **printArgs** dans **prologTerm.c**

1.2.3 Statement

Il n'y a qu'une seule instruction **ECHO Expr** dans APS0.

1. Grammair. On ajoute la grammair de **ECHO** dans **paser.y**.
2. AST. Ajouter les structures de **Stat** dans **ast.h**, ainsi que le constructeur dans **ast.c**
3. Prolog. Ajoute la fonction d'affichage **printStat** dans **prologTerm.c**

1.2.4 Déclaration

Il y a 3 types des déclarations dans APS0. On peut déclararer le constant, la fermeture et la fermeture récursive.

1. Grammair. On ajoute la grammair pour comment exprimer une déclaration. Dans notre langage, une déclaration commencée par son type, **CONST**, **FUN** ou **FUNREC**.
2. AST. Ajouter la structure de **Dec** dans **ast.h**, ainsi que son constructeur **newASTDec** dans **ast.c**
3. Prolog. Ajoute la fonction d'affichage **printDec** dans **prologTerm.c**

1.2.5 Commande

La commande est composée par la déclaration ou l'instruction, donc je code **Cmd** après réalise **Dec** et **Stat**.

1. Grammair. C'est plus simple que les grammaires précédentes, comme il n'y a plus de détail à décrire, une commande soit une déclaration soit une instruction.
2. AST. Ajouter la structure de **Cmd** et **Cmds** dans **ast.h**, ainsi que son constructeur **newASTCmd** et **appendCmds** dans **ast.c**
3. Prolog. Ajoute la fonction d'affichage **printCmd** et **printCmds** dans **prologTerm.c**

1.2.6 Programme

Un programme de APS0 est une suite de commande.

1. Grammair. Un programme est une suite de commande dans une pair de crochet.
2. AST. Ajouter la structure de **Proc** dans **ast.h**, ainsi que son constructeur **newASTProg** dans **ast.c**
3. Prolog. Ajoute la fonction d'affichage printProg dans **prologTerm.c**

1.3 Etape3. Typage

1.3.1 Expression

l'expression est la base du langage, donc je définit d'abord de type basique pour l'expression.

1. Le type des booléan, d'entier et des paramères de primitive ainsi que le type de résultat des primitives.
2. Le typage de **ASTIf**, il contient 3 expression : condition, résultat, alternance. On garanti que le type de condition est booléan et les deux autres expression doit avoir le même type.
3. Le typage de l'application de fermeture.
4. Le typage de ABS

1.3.2 Déclaration

1. Déclaration de const est exprimé par const(id, type, expr). On doit garantir que le type de expression est aussi le type de identifiant dans l'environement G.
2. Le typage de function et function récursive.

1.3.3 Statement

le type de l'instruction est void. APS0 peut seulement afficher les entiers, donc on a juste besoin de garantir le type d'expression de **echo** est int.

1.3.4 Commande

le type de la commande est void. la suite de commandes peut commencer par une déclaration ou une instruction.

1.3.5 Program

le type de program est aussi void.

1.4 Etape4. Analyse sémantique

1.4.1 Définition de Value

Value est la structure des données basique de APS0, elle soit un entier **inN**, soit une fermeture **inF**, soit une fermeture récursive **inFr**. Après l'exécution de l'expression, un résultat de **Value** est produit.

1.4.2 Implémante structure de l'environement

Je définit d'abord un struct **Env** dans **eval.h**. Il permet de conserver l'association entre les identifications et les valeur.

1.4.3 Expr

La fonction **evalExpr** est définit dans **eval.c**, cette fonction permet d'interprèter une expression et de rendre un résultat. Dans APS0, il y a 7 genres de l'expression.

1. Entier. C'est le type basique, juste rendre **inN**.
2. Identifiant. Il faut chercher la valeur qui correspond à cet identifiant dans l'environement et rendre cette valeur.
3. Booléan. Rendre inN(0) pour false et inN(1) pour true.
4. Primitives. Calculer le résultat en fonction de l'opération arithmétique.
5. If. Interprèter d'abord l'expression de condition. Si le résultat est positive, on interprète deuxième expression, sinon interprète la troisième.
6. Lambda. Rendre une **inF**.
7. AppFun. C'est la suite des expressions. Comme la règle du typage, on est sûr que la première expression est le nom de fermature, et les restes sont les valeurs des arguments. On doit interprèter les restes d'expressions afin d'obtenir ces valeurs et ajouter à l'environement de la fermeture. A la fin on interprète la corps de fermeture avec son environement.