Projet Compilation

Partie I

Travail réalisé par AGRA Maxence, BEKHEDDA Maxence, DÉCAVÉ Gabriel et GIANELLI Thomas

11 janvier 2024

Table des matières

1 Introduction

Le but de cette première partie du projet est la construction d'un arbre abstrait à partir d'un code écrit en `canAda`.

Pour se faire, il nous faut dans un premier temps un analyseur lexical nous permettant de transformer notre code qui est un simple fichier `txt` en une suite de tokens. Ensuite vient l'étape de l'analyse syntaxique. Cela nécessite la construction d'une grammaire pour vérifier que notre code est bien une succession de règles valides. Enfin, une fois que nous avons toutes nos règles, nous pouvons construire l'arbre abstrait de notre code et le visualiser.

Nous avons choisi d'utiliser `python` pour la réalisation de ce projet. D'une part il s'agit du langage que nous maîtrisons le mieux tous les quatre, et d'autre part nous estimons que c'est celui qui offre le plus de liberté grâce à la simplicité des structures et au nombre de bibliothèques disponibles.

2 Analyse Lexicale

Nous avons commencé par coder un parseur en utilisant les regex. Nous avons également créé une liste contenant les mots-clés afin de les différencier aisément des noms de variable, ainsi qu'une liste avec les opérateurs du langage car il n'existe pas de regex qui nous renvoie uniquement ceux-là.

Nous supprimons ensuite les commentaires et les tokens de saut de ligne. Nous les avions gardé jusqu'à présent car il nous permettait de savoir à quelle ligne se trouvait l'erreur si notre parseur détectait une erreur lexicale.

Enfin, nous affectons à chaque variable un entier. Ce code nous permettra de manipuler plus facilement nos variables par la suite.

Nous avons donc ainsi transformé notre fichier `txt` d'entré en une liste de nombres et de tuples. Chacun de ses éléments est un token dont on peut retrouver la valeur en se réferrant au lexique et au code lexical générer à l'execution.

3 Analyse Syntaxique

L'analyse syntaxique commence par la création d'une grammaire efficace. En effet, une grammaire nous est donnée dans le sujet, mais celle-ci est loin d'être LL1. Nous avons quant à nous besoin de savoir quelle règle appliquer dès la lecture du premier token (même si quelques exceptions sont possibles). Nous avons donc ajouté de nombreuses règles afin d'obtenir une grammaire (jointe en annexe) quasi LL1 qui respecte les priorités opératoires.

4 Construction de l'arbre abstrait

5 Conclusion

A La grammaire

```
FICHIER -> with IDENT point text_io pointvirgule use ada point text_io pointvirgule
    procedure IDENT is DECLETOILE begin INSTRPLUS end IDENTINTER pointvirgule eof .

IDENT -> ident .
IDENTPLUS -> IDENT IDENTPLUS2 .
IDENTPLUS2 -> virgule IDENT IDENTPLUS2 | .
IDENTINTER -> IDENT | .

DECL -> type IDENT DECL1
        | IDENTPLUS deuxpoints TYPE ASSIGNINTER pointvirgule
        | procedure IDENT PARAMSINTER is DECLETOILE begin INSTRPLUS end IDENTINTER
    pointvirgule
        | function IDENT PARAMSINTER return TYPE is DECLETOILE begin INSTRPLUS end
    IDENTINTER pointvirgule.
DECL1 -> pointvirgule | is DECL2 .
DECL2 -> access IDENT pointvirgule | record CHAMPSPLUS end record pointvirgule .

DECLETOILE -> DECL DECLETOILE2 | .
DECLETOILE2 -> DECL DECLETOILE2 | .

CHAMPS -> IDENTPLUS deuxpoints TYPE pointvirgule.

CHAMPSPLUS -> CHAMPS CHAMPSPLUS2 .
CHAMPSPLUS2 -> CHAMPS CHAMPSPLUS2 | .

TYPE -> IDENT | access IDENT .

PARAMS -> ( PARAMPLUS ) .

PARAMSINTER -> PARAMS | .

PARAM -> IDENTPLUS deuxpoints MODEINTER TYPE .

PARAMPLUS -> PARAM PARAMPLUS2 .
PARAMPLUS2 -> pointvirgule PARAM PARAMPLUS2 | .

MODE -> in MODE2 .
MODE2 -> out | .

MODEINTER -> MODE | .

EXPR -> OP EXACCES .
EXPR1 -> ENTIER | CARACTER | true | false | null | ( EXPR ) | IDENT EXPR2 | new IDENT |
    character'val ( EXPR ).
EXPR2 -> EXPRPLUS | .
EXACCES -> point IDENT EXACCES | .

EXPRINTER -> EXPR | .

EXPRPLUS ->  ( EXPR EXPRPLUS2 ) .
EXPRPLUS2 -> virgule EXPR EXPRPLUS2 | .

INSTR -> OP point IDENT INSTR2
        | IDENT INSTR2
        | return EXPRINTER pointvirgule
        | begin INSTRPLUS end pointvirgule
        | while EXPR loop INSTRPLUS end loop pointvirgule
        | if EXPR then INSTRPLUS ELSIFETOILE ELSEINTER end if pointvirgule
```

```
        | for IDENT in REVERSEINTER EXPR point point EXPR loop INSTRPLUS end loop
    pointvirgule.

INSTR2 -> deuxpoints eg EXPR pointvirgule
        | pointvirgule
        | EXPRPLUS pointvirgule.

INSTRPLUS -> INSTR INSTRPLUS2.
INSTRPLUS2 -> INSTRPLUS | .

REVERSEINTER -> reverse | .

OP -> OPE1 OPE' .
OPE' -> or ELSE OPE1 OPE' | .
OPE1 -> OPE2 OPE1' .
OPE1' -> and THEN OPE2 OPE1' | .
OPE2 -> OPE3 OPE2' .
OPE2' -> not OPE3 OPE2' | .
OPE3 -> OPE4 OPE3' .
OPE3' -> COMPARATEUR OPE4 OPE3' | .
OPE4 -> OPE5 OPE4'.
OPE4' -> ORDRE OPE5 OPE4' | .
OPE5 -> OPE6 OPE5'.
OPE5' -> ADD OPE6 OPE5' | .
OPE6 -> OPE7 OPE6' .
OPE6' -> MULT OPE7 OPE6' | .
OPE7 -> moins OPE8 | OPE8 .
OPE8 -> EXPR1 .

ELSE -> else | .
THEN -> then | .
COMPARATEUR -> eg | dif .
ORDRE -> inf EGAL | sup EGAL .
EGAL -> eg | .
ADD -> plus | moins .
MULT -> fois | div | rem .

ELSEINTER -> else INSTRPLUS | .
ELSIFETOILE -> elsif EXPR then INSTRPLUS ELSIFETOILE | .
ASSIGNINTER -> deuxpoints eg EXPR | .
ENTIER -> entier .

CARACTER -> caracter .
```

B Exemple

B.1 Le code

```
1  with Ada.Text_IO ; use Ada.Text_IO ;
2
3  procedure MyFile is
4     function MyFunction return Integer is
5     begin
6        return 42;
7     end MyFunction;
8  begin
9     MyFunction;
10 end MyFile;
```

B.2 L'arbre généré