TP compilateurs 3 : Analyse sémantique

Dans ce TP, nous allons une dernière fois enrichir le langage de notre mini-compilateur puis nous nous attaquerons à l'un des aspects les plus délicats de l'analyse sémantique : la couture de l'arbre syntaxique abstrait.

1 Enrichissement du langage

1.1 But

Pour l'instant, notre langage est purement linéaire et seul le résultat de la dernière expression est affiché. Nous allons le modifier de deux manières :

1. Ajout d'une instruction `print` qui permettra d'afficher une expression à tout moment. Du coup, on renoncera à afficher automatiquement le résultat de la dernière expression du programme.
2. Ajout de la possibilité de faire des boucles avec un `while`. Comme notre langage ne connaît qu'un seul type, numérique, nous partirons de l'idée que le nombre `0` correspond à `faux` et tout autre nombre à `vrai`.

Ainsi nous devrions pouvoir traiter des programmes hautement utiles tels que celui de la figure 1.

Remarque Dans le langage tel que nous l'avons construit, le point-virgule est un *séparateur* d'instruction (et non un *terminateur*). De plus, de ce point de vue, le `while` sera considéré comme une instruction comme les autres. Ceci a deux conséquences (un peu inhabituelles) pour le point-virgule :

1. La dernière instruction d'un programme ou d'un bloc *ne doit pas être suivie* d'un point-virgule ;

```
a=0;
while (a-10) {
    b = 0;
    while (b-2) {
        print a;
        print b;
        b = b+1
    };
    a = a+1
};
print 100
```

FIGURE 1: Exemple de code d'entrée possible pour notre compilateur

2. Le corps d'un while *doit être suivi* d'un point-virgule.

Le code de la figure 1 illustre ces particularités.

1.2 Analyseur lexical

Bien entendu, nous commencerons par compléter notre analyseur lexical. Reprenez votre analyseur lexical, renommez-le en `lex5.py`, et commencez par lui donner la possibilité de reconnaître les accolades `{}`.

Ensuite, nous allons nous occuper des mots réservés. Pour simplifier les choses, nous déciderons que `while` et `print` seront des mots réservés de notre langage.

Pour ce faire, commencez par ajouter une liste des mots réservés dans votre code source :

```
reserved_words = (
    'while',
    'print'
)
```

Ensuite, ajoutez ces mots réservés à la liste des lexèmes. Vous pouvez le faire à la main (mais il faudra alors toujours penser à synchroniser ces deux listes en cas de changement), ou par exemple de la manière suivante [1] :

```
tokens = (
    'NUMBER',
    'ADD_OP',
    'MUL_OP',
    'IDENTIFIER',
) + tuple(map(lambda s:s.upper(),reserved_words))
```

Enfin, il ne reste plus qu'à "intercepter" les lexèmes qui pourraient être des identificateurs mais qui sont au fait des mots réservés en modifiant la définition du lexème `IDENTIFIER` :

```
def t_IDENTIFIER(t):
    r'[A-Za-z_]\w*'
    if t.value in reserved_words:
        t.type = t.value.upper()
    return t
```

Voilà pour l'analyseur lexical ; il ne vous reste plus qu'à le tester sur quelques entrées de votre choix.

1.3 Analyseur syntaxique

Ouvrez une nouvelle version de votre analyseur syntaxique sous le nom `parser5.py` et, avant d'oublier, dites-lui d'utiliser les nouvelles version des modules importés :

```
from lex5 import tokens
import AST
```

Maintenant, modifiez votre grammaire pour la faire correspondre à la description BNF suivante :

```
program:     statement
             | statement ';' program
statement:   assignation
             | structure
             | PRINT expression
structure:   WHILE expression '{' program '}'
expression:  ... (comme avant)
```

Vous rencontrerez deux nouveaux types de noeuds :

1. Nous conservons ici la convention implicite que nous avons adoptée de mettre en majuscule les *types* de lexèmes.

```
a=0;
while (a-10) {
    print a;
    a = a+1
}
```

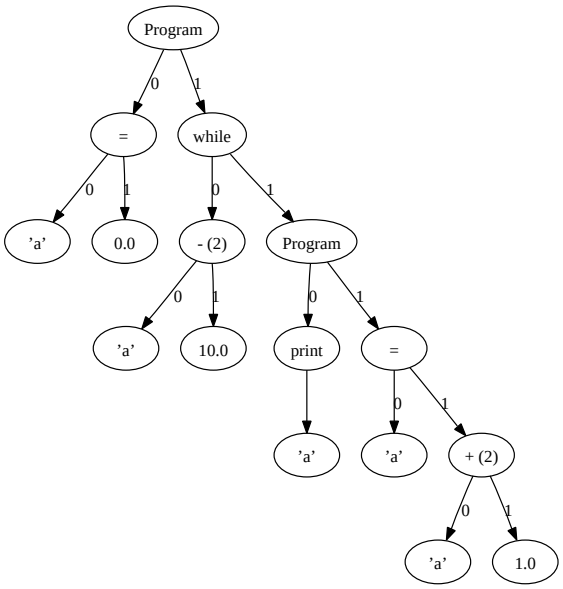

FIGURE 2: Exemple d'analyse de `parser5.py`

PrintNode(expr) correspond à une instruction `print`. Prend en paramètre le sous-arbre de l'expression à afficher.

WhileNode([cond, body]) correspond à un `while`. Le premier élément de la liste est le sous-arbre représentant la condition et le second élément de la liste est le sous-arbre (de racine `ProgramNode`) correspondant au corps de la boucle.

La figure 2 vous montre le résultat auquel vous devriez arriver.

Pour pouvoir utiliser facilement l'analyseur lexical depuis un autre module, nous rajouterons encore une petite fonction toute simple :

```
def parse(program):
    return yacc.parse(program)
```

Voilà, nous y sommes. Testez votre analyseur autant que vous le souhaitez, produisez de superbes arbres syntaxiques et affichez-les au mur de votre chambre en guise de trophée (cette dernière étape étant bien entendu facultative...).

2 Analyse sémantique

Le langage traité par notre petit compilateur étant très limité, cela limite aussi grandement les possibilités d'analyse sémantique. Comment vérifier les types des affectations lorsque notre langage n'a qu'un seul type ? Comment vérifier les signatures des fonctions alors que notre langage n'a pas de fonctions ? Comment s'occuper de la liaison des variables alors que notre langage n'a pas de notion de portée ?

Nous nous contenterons donc dans ce TP de coudre l'arbre syntaxique. Non seulement cela nous permet d'illustrer un des points délicats de cette étape de la compilation, mais cela nous permettra également, lors du prochain TP, d'implémenter un interpréteur itératif.

Pour implémenter la couture de l'arbre, il faut ajouter des méthodes au différents types de noeuds de `AST.py`. Nous pourrions le faire en modifiant ce dernier fichier, mais pour garder les choses bien séparées (et pour profiter de la souplesse de python !) nous utiliserons un décorateur qui permet d'ajouter une méthode à une classe depuis n'importe où. Ouvrez donc un fichier `threader.py` et commencez par les importations réglementaires :

```
import AST
from AST import addToClass
```

Ensuite, nous pouvons définir la couture standard d'un noeud : pour coudre un noeud, il faut d'abord coudre récursivement ses enfants puis le noeud lui-même. Nous allons donc écrire une méthode qui prend en argument le dernier noeud cousu, coud le sous-arbre considéré et retourne le nouveau "dernier noeud cousu" :

```
@addToClass(AST.Node)
def thread(self, lastNode):
    for c in self.children:
        lastNode = c.thread(lastNode)
    lastNode.addNext(self)
    return self
```

Ajoutons encore une fonction qui servira d'amorce à la couture ; à noter qu'on crée ici un nouveau type de noeud, un `EntryNode`, qui définit le point d'entrée du programme en termes de flux de contrôle. Ce noeud fait partie de l'arbre cousu mais pas de l'arbre syntaxique à proprement parler.

```
def thread(tree):
    entry = AST.EntryNode()
    tree.thread(entry)
    return entry
```

Un p'tit programme principal et on est presque bon :

```
if __name__ == "__main__":
    from parser5 import parse
    import sys, os
    prog = open(sys.argv[1]).read()
    ast = parse(prog)
    entry = thread(ast)

    graph = ast.makegraphicaltree()
    entry.threadTree(graph)

    name = os.path.splitext(sys.argv[1])[0]+'-ast-threaded.pdf'
    graph.write_pdf(name)
    print ("wrote threaded ast to", name)
```

Si vous testez ce programme, vous vous rendrez compte que la couture est presque parfaite, à un détail près : le `WhileNode`. Lorsqu'on coud un noeud de ce type, plusieurs choses doivent se passer :

- La condition doit être cousue *avant* le `WhileNode`, mais le corps du `while` doit venir *après* le `while` lui-même ;
- Le `while` doit avoir *deux* sorties suivant la valeur de sa condition (c'est dailleurs pour cette raison que `Node` implémente une méthode `addNext` et non `setNext`...)

Il ne reste donc plus qu'à surcharger la méthode `thread` dans ce cas :

```
@addToClass(AST.WhileNode)
def thread(self, lastNode):
```

Je vous laisse le soin d'écrire le corps de la fonction de manière à ce que le résultat corresponde à la figure 3.

Testez votre couturier (?) et contemplez avec émotion les beaux (et passablement illisibles) schémas que vous arrivez à générer. Affichez-en un ou deux pour épater les copains, mais n'abusez tout de même pas des imprimantes couleur de l'école !

```
a=0;
while (a-10) {
    print a;
    a = a+1
}
```

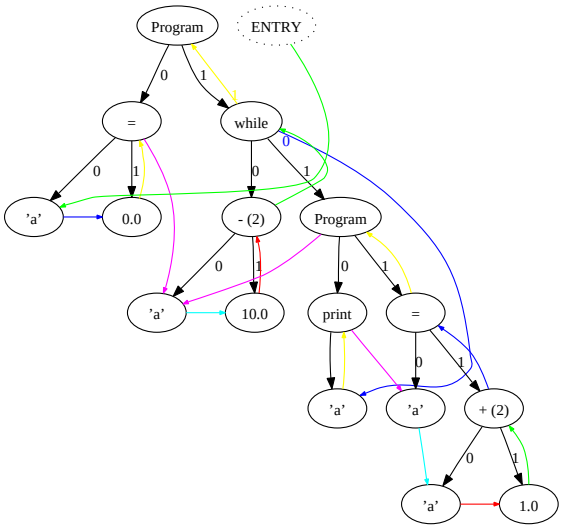

FIGURE 3: Exemple de couture de `threader.py`

3 Conclusion

Avec son `print` et son `while`, notre langage commence maintenant à ressembler à un “vrai” (bien que simpliste) langage de programmation.

De plus, il est maintenant entièrement analysé et même cousu. Dans le prochain TP, nous nous occuperons des différentes façon d’exécuter ce code : interprétations récursive et itérative, et compilation.