Interpréteur style Python

Lénaïc Rousseau INM 4

Principale fonctionnalités

- conditions
- affectations de variables
- plusieurs types (int, char, unit (pour les fonctions qui ne retourne rien))
- fonctions (avec récursion, plusieurs arguments, valeur de retour)
- imbrication (if dans des if, mélanger fonction, variables et valeur au sain d'une même eag)
- détection d'erreur

Limites

- La stack pourrait utliser un hash table pour rechercher plus éfficacement les noms de variables.
- Il aurait fallu isoler les proccess d'analyse syntaxique et d'evaluation. Dans ma version, ils partagent la même interface stack.h pour gerer les variables, ce qui pose probleme pour une version intéractive.
- Pour faciliter l'ajout de fonctionnalités, un automate capable de reconnaître automatiquement les mots-clés et la grammaire qui lui sont donnés serait utile.
- La stack prend beaucoup de place inutile (tableau taille fixe à 3 dimensions)

Détection d'erreur

Constante vérification d'erreur de type, pour les returns et affectation par exemple, ce qui rend les erreurs au runtime peu fréquente. Mais aussi, vérification que les variables soit déclarées dans les bons scopes. Exemple:

```
int a <- 1;
if (1) {
    a = 2; // ceci est possible
    int b <- 1;
}
b = 2; // ceci renverra exactement cette erreur: Erreur syntaxique 11:3 :
NAME non autorisé ici. 'b' not exist.
```

Fonctionnement

Au lancement de l'interprète:

- début de l'analyse syntaxique. On découvre chaque nouveau lexème, quitte si erreur lexical. Les fonctions de l'analyse syntaxique respecte plus ou moins la grammaire, avec certaines fonctions simplifiée pour gagner de la place.
- construction de l'ast. Aussi on stocke les noms des variables dans une "stack", pour détecter des variables non déclarées. Les fonctions sont stockées dans un tableau a part. On stocke le pointeur correspondant a l'ast.
- ensuite, quand le fichier à été vérifié entièrement, sans erreur, on clear la stack (pas les fonctions).
- exécution de l'ast, une nouvelle stack est créé.

Fonctionnement de la stack

Voici la structure qui la représente:

```
int stack_count;
int scope_count[1000];
int nb_var[1000][100];

node *stack[1000][100][1000];
```

- A chaque appel de fonction, on incrémente `stack_count`
- A chaque changement de scope (un if par exemple), `scope_count[stack_count]` est incrémenté.

On peut donc stocker 1000 variables par scope, avec 100 niveaux de scope et 1000 de "stack"

Chaque niveau de "stack" est indépendant, cependant, un état de niveau `[1][50]` rend l'accès au variable de `[1][0], [1][1], [1][2], ..., [1][50]` possible.

Syntax

Pour une initialisation: `int a <- 9;`

Pour une réaffectation: `a = 10;`

Pour une fonction:

```
fun a(int b): int {
    return 9;
}
```

Les eag sont aussi supportées, avec quelque ajouts:

- le moins unaire
- les opérations booléenne (&&, ||, !, <, <=, >, >=)

Les imbrications sont aussi possibles:

```
if (ma_function(800 * 6) - 10 && (!8)) {
    if (1) {

    }
    else {

    }
}
// else non obligatoire
```

Note

Une eag sans affectation, sera affiché sur la console automatiquement.

Exemple:

```
char name <- "Hello world";
name;
my_function(); // imaginons type int et return 7
```

Output:

```
"Hello world"
7
```

Exemple d'algo

syracuse

```
fun mod2(int n): int {
    if (n == 0) {
        return 0;
    }

    if (n == 1) {
        return 1;
```

```
    } else {
        return mod2(n-2);
    }
}



fun syra(int n1): int {
    n1;
    if (n1 == 1) {
        "Le total d'appel recursif est:";
        return 0;
    }
    if (mod2(n1) == 0) {
        return syra(n1 / 2) + 1;
    }
    else {
        return syra(3 * n1 + 1) + 1;
    }
}


syra(40);
```

Output:

```
bash-5.2$ ./Boa ../test/valid/syracuse.boa
40
20
10
5
16
8
4
2
1
"Le total d'appel recursif est:"
8
```

Autre algorithmes:

```
bash-5.2$ ./Boa ./../algorithmes/factoriel.boa
120
bash-5.2$ ./Boa ./../algorithmes/fibonacci.boa
6765
4181
2584
```

```
1597
987
610
377
233
144
89
55
34
21
13
8
5
3
2
1
1
bash-5.2$ ./Boa ./../algorithmes/premier.boa
37
31
29
23
19
17
13
11
7
5
3
2
bash-5.2$ ./Boa ./../algorithmes/syracuse.boa
40
20
10
5
16
8
4
2
1
"Le total d'appel recursif est:"
8
```