Langage Vyking

Grammaire et spécifications du language

Compilateurs

INFO0085

PR. PIERRE GEURTS

CYRIL SOLDANI

11 mars 2013

ROBIN KEUNEN s093137
PIERRE VYNCKE s091918
1ème Master Ingénieur Civil

1 Introduction

Le but de Vyking est d'étudier l'implémentation de langages de programmation fonctionnels. Nous nous inspirerons principalement de Scheme et de Python. Nous avons apprécié la puissance de Scheme et de l'usage des listes. Les listes seront donc une structure de base de notre langage. Nous apprécions aussi la concision et la syntaxe claire de Python.

Le nom du langage vient de Vy (Vyncke) - (Keunen) -ing.

2 Spécifications

Syntaxe

Nous voulons que Vyking soit un langage de haut niveau. le code doit être lisible, concis et élégant. Nous sommes particulièrement sensible au "PEP 20 (The Zen of Python)" :

- Beautiful is better than ugly.
- Explicit is better than implicit.
- Simple is better than complex.
- Complex is better than complicated.
- Readability counts.

Le langage sera donc épuré d'accolades, de points-virgules et autre ponctuation inutile. La tabulation délimitera les blocs. Outre le look épuré du code, ce choix devrait forcer les utilisateurs du langages à se conformer à un seul style d'indentation (contrairement à C et ses styles K&R, Allman, Whitesmiths, ...).

Listes

Nous avons trouvé que l'utilisation des listes dans Scheme était très élégante pour implémenter certain schémas de récursion. La structure de liste sera donc implémentée directement dans le langage. Les listes seront construites à partir de paires pointées.

Vyking fournira plusieurs fonctions de bases pour manipuler les listes : `cons`, `append`, `list`, `map` et `apply`. Nous remplaçons les `car` et `cdr` de Scheme qui ne sont pas très intuitif par `head` et `tail`.

Si nous en avons le temps, nous tenterons d'implémenter des "list comprehension".

Fonctions de première classe

Pour rendre le langage réellement fonctionnel, il semble indispensable de pouvoir manipuler les fonctions comme des objets de première classe. Nous voulons pouvoir passer les fonctions en argument, les retourner à partir d'autres fonctions et permettre les définitions de fonctions imbriquées. Pour pouvoir supporter les fonctions imbriquées, nous avons besoin de lier l'environnement à la définition d'une fonction. Nous tenterons donc d'implémenter des "closures" (fermeture).

L'implémentation des closures semble compliquée. Dans un premier temps, nous nous contenterons de passer les fonctions en argument par des pointeurs de fonctions. Les closures viendront dans un second temps.

3 Choix du langage pour l'implémentation

Le compilateur sera implémenté en Python. La plupart des projets pour les cours sont à rendre en Java ou en C/C++. Scheme (ou Racket) semble adapté pour l'analyse du langage

mais il est peu utilisé dans des projets hors du secteur académique. Nous voulions nous former à un autre langage, plus moderne. Nous avons choisis Python, l'expressivité du langage et la richesse des librairies devrait nous permettre de nous concentrer sur les algorithmes et sur Vyking.

4 Grammaire

```
(* language components *)
file_input = {NEWLINE | statement}, ENDMARKER

keyword = 'return' | 'defun'
        | 'and' | 'or' | 'not' | 'is'
        | 'if' | 'elif' | 'else'
        | 'while' | 'for'
        | 'map' | 'apply' | 'cons' | 'append' | 'list'
        | 'head' | 'tail | 'in';

statement = (funcall
                    | if_stat
                    | while_stat
                    | for_stat
                    | let_stat
                    | return_stat
                    | funcdef_stat
                    | import_stat
                    ), NEWLINE;

funcall = id, '(', [args], ')' | list_fun;
fundef = 'defun', id, parameters, ':', block;
if_stat = 'if', test, ':', block, {'elif' test ':' block}, ['else' ':' blo
while_stat = 'while', test, ':', block;
for_stat = 'for', id, 'in', list, ':', block;
let_stat = id, '=', (object | exp);
return_stat = 'return', (exp | list);
import_stat = 'from', id, 'import', name;

list_fun = cons_fun | append_fun | list_fun | head_fun | tail_fun | map_fu
cons_fun = 'cons', '(', list, ')' | atom, '+', list;
append_fun = 'append', '(', list, ')' | list, '+', list;
list_fun = 'list', '(', list, ')';
head_fun = 'head', '(', list, ')';
tail_fun = 'tail', '(', list, ')';
map_fun = 'map', '(', id, ',', list, ')';

exp = add_exp;
add_exp = mul_exp, {('+' | '-'), mul_exp};
mul_exp = atom {('*' | '/' | '%'), atom}
atom = id | int | float | 'None' | bool | funcall;
```

```
block = statement | NEWLINE INDENT statement {statement} DEDENT;
test = xor_test
xor_test = or_test, {'|', or_test};
or_test = and_test, {'or', and_test};
and_test = not_test {'and', not_test};
not_test = 'not' not_test | identity | atom;
identity = comparison, ['==', atom]
comparison = exp, [comp_op, exp];

parameters = '(', [ids], ')';
ids = id, ',', ids | id;
args = arg, ',', args | arg;
arg = id | funcall | exp;

(* base types *)
int = [(+|-)], digit_0, {digit} | '0';
float = int, '.', [digit, {digit}] [exponent];
exponent = ('e' | 'E'), int;
char = (letter | digit | ' ');
esc_char = '\', letter;
bool = 'True' | 'False';

(* complex types *)
string = '"', {char | esc_char |"'" }, '"' | "'", {char | '"'}, "'";
list = '[', {object, ','}, 'object', ']' | '[', ']' | list_fun;

object = int | float | char | bool | string | list | id;

(* base cases *)
id = (letter | '_'), { letter | digit | '_'} -keyword;
letter = uppercase | lowercase;
uppercase = 'A' | 'B' | 'C' | 'D' | 'E' | 'F' | 'G'
          | 'H' | 'I' | 'J' | 'K' | 'L' | 'M' | 'N'
          | 'O' | 'P' | 'Q' | 'R' | 'S' | 'T' | 'U'
          | 'V' | 'W' | 'X' | 'Y' | 'Z' ;
lowercase = 'a' | 'b' | 'c' | 'd' | 'e' | 'f' | 'g'
          | 'h' | 'i' | 'j' | 'k' | 'l' | 'm' | 'n'
          | 'o' | 'p' | 'q' | 'r' | 's' | 't' | 'u'
          | 'v' | 'w' | 'x' | 'y' | 'z' ;
digit = '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7'
      | '8' | '9';
digit_0 = '1' | '2' | '3' | '4' | '5' | '6' | '7' |
        | '8' | '9';
tab = '    ' | '\t';
comp_op = '==' | '<' | '>' | '<=' | '>=' | '!=' | 'is';
```