Un interpréteur pour le langage WHILE

Arthur Jacquin

Sommaire

Introduction - Objet de la présentation

- Point de départ : langage WHILE tel qu'étudié lundi
- Objectif : concevoir et mettre en oeuvre une implémentation
- Exemples de scripts à la fin

Introduction - Fonctionnement général d'un interpréteur

- Code source
- PARSING $\longrightarrow$ Objects informatiques
- INTERPRÉTATION $\longrightarrow$ Résultats

Introduction - Langage retenu : OCaml

- Typage fort, possibilité de définir de nouveaux types
- Très bon *pattern matching*, même sur les types non standards
- S'interprète ou se compile, au choix

Conception - Définir la nature des objets

- Très important, typage explicite à priviliégier par la suite
- Choix de l'ensemble des identifieurs

```
13 (* Identifiers *)
14 type variable = A | B | C | D | E | F | G
15                | H | I | J | K | L | M | N
16                | O | P | Q | R | S | T | U
17                | V | W | X | Y | Z;;
18
19 (* Arithmetic expressions *)
20 type arithm = N of int
21              | V of variable
22              | Plus of arithm * arithm
23              | Minus of arithm * arithm
24              | Mult of arithm * arithm;;
```

Conception - Définir la nature des objets

```
26 (* Boolean expressions *)
27 type boolean = True
28                | False
29                | Equal of arithm * arithm
30                | Lower of arithm * arithm
31                | Not of boolean
32                | Or of boolean * boolean
33                | And of boolean * boolean;;
34
35 (* Commands *)
36 type command = Skip
37                | Affect of variable * arithm
38                | Concat of command * command
39                | If of boolean * command * command
40                | While of boolean * command
41                | Print of arithm;;
42
43 (* Memory *)
44 type sigma = (variable -> int);;
45
46 (* Errors *)
47 exception ParsingError;;
48 exception InvalidNumberOfArguments;;
```

Conception - Définir la nature des objets

- Fonction principale

```
243 let main : sigma =
244     if (Array.length Sys.argv) != 2 then
245         raise InvalidNumberOfArguments
246     else
247         let initial_memory : sigma = (fun v -> 0)
248         and program : command = parse (Sys.argv.(1))
249         in int_command program initial_memory;;
```

Conception - Implémenter la logique interprétative

```
54 let rec int_arithm (a: arithm) (m: sigma) : int =
55     match a with
56     | N n -> n
57     | V v -> m v
58     | Plus (x, y) -> (int_arithm x m) + (int_arithm y m)
59     | Minus (x, y) -> (int_arithm x m) - (int_arithm y m)
60     | Mult (x, y) -> (int_arithm x m) * (int_arithm y m);;
61
62 let rec int_boolean (b: boolean) (m: sigma) : bool =
63     match b with
64     | True -> true
65     | False -> false
66     | Equal (x, y) -> (int_arithm x m) = (int_arithm y m)
67     | Lower (x, y) -> (int_arithm x m) <= (int_arithm y m)
68     | Not x -> if (int_boolean x m = true) then false else
       true
69     | Or (x, y) -> (int_boolean x m) || (int_boolean y m)
70     | And (x, y) -> (int_boolean x m) && (int_boolean y m);;
```

Conception - Concevoir simultanément syntaxe et parsing

- Simple à utiliser ET simple à analyser (efficacité)
- Prendre en compte les modalités de l'interface avec le fichier

Parsing - RPN pour les expressions arithmétiques

- Les expressions arithmétiques peuvent être complexes
- Élément très courant : ce serait bien de trouver une méthode efficace
- Une solution : la notation polonaise inversée (RPN) lève toute ambiguité
- Permet une lecture linéaire de l'expression, avec accumulation des litéraux (variables, entiers) dans une pile et réduction par les opérateurs (+, -, *)
- Mon implémentation ne fonctionne pas avec les nombres négatifs

Parsing - Fonctions auxiliaires de recherche de motif

- Moteur de recherche d'expression régulière : échoue à extraire les "arguments"

```
199 let rec parse_boolean (s: string) : boolean =
200     if s = "TRUE" then True
201     else if s = "FALSE" then False
202     else let b, x, y = find_w_X_v_Y_u s "(" ") AND (" ")" in
203         if b then And (parse_boolean x, parse_boolean y)
204     else let b, x, y = find_w_X_v_Y_u s "(" ") OR ("  ")" in
205         if b then Or (parse_boolean x, parse_boolean y)
206     else let b, x    = find_w_X_v     s "NOT ("       ")" in
207         if b then Not (parse_boolean x)
208     else let b, x, y = find_X_w_Y     s     " == "        in
209         if b then Equal (parse_arithm x, parse_arithm y)
210     else let b, x, y = find_X_w_Y     s     " <= "        in
211         if b then Lower (parse_arithm x, parse_arithm y)
212     else raise ParsingError;;
```

Parsing - Traitement global

- Accès au code source ? Utilisation d'un `input channel` qui permet de récupérer les lignes les unes après les autres
- Problème d'expressions sur plusieurs lignes : récursivité

Parsing - Traitement global

```
214 let rec parse_program (ic: in_channel) : command =
215     let res : command ref = ref Skip
216     and continue : bool ref = ref true
217     and parse_instruction (s: string) : command =
218         let b, x = find_w_X_v s "while " " do {" in if b then
219             While (parse_boolean x, parse_program ic)
220         else ...
221     in begin
222     while !continue do
223         try let line : string = strip (input_line ic) in
224             if line = "}" || line = "} else {" then continue :=
        false
225             else if line = "" || line.[0] = '#' then ()
226             else res := Concat (!res, parse_instruction line)
227         with error -> match error with
228             | End_of_file -> (close_in ic; continue := false)
229             | ParsingError -> (close_in ic; raise ParsingError)
230             | e -> (close_in_noerr ic; raise e)
231     done;
232     !res;
233     end;;
```

Exemples - factorial.while

```
1  # Factorielle de A
2  # A doit etre positif
3
4  # Initialisation
5  A := 5
6  B := 1
7
8  # Calcul
9  while 2 <= A do {
10     B := B A *
11     A := A 1 -
12 }
13
14 # Affichage des resultats
15 print B
```

Exemples - pgcd.while

```
1  # Plus grand diviseur commun de A et B
2  # A et B doivent etre positifs
3
4  # Initialisation
5  A := 21
6  B := 15
7  C := 0
8
9  # Calcul
10 while NOT (B == 0) do {
11     if NOT (B <= A) then {
12         C := A
13         A := B
14         B := C
15     } else {
16         A := A B -
17     }
18 }
19
20 # Affichage des resultats
21 print A
```

Pour aller plus loin

- `https://github.com/arthur-jacquin/while-lang`
- Utiliser une fonction récursive dans `parse_arithm`
- Corriger une erreur dans le traitement des expressions booléennes