TP2 - Analyse syntaxique descendante

Raphaël Baron - Paul Chaignon

15 octobre 2013

1 Grammaire

1.1 Nouvelle grammaire

$$
\begin{aligned}
<Fichier> &\rightarrow <Expr>\ EOF \\
<Expr> &\rightarrow <Termb><SuiteExpr> \\
<SuiteExpr> &\rightarrow \text{"}ou\text{"}\ <Expr>\ |\ ? \\
<Termb> &\rightarrow <Facteurb><SuiteTermb> \\
<SuiteTermb> &\rightarrow \text{"}et\text{"}\ <Termb>\ |\ ? \\
<Facteurb> &\rightarrow <Relation>\ |\ \text{"}(\text{"}\ <Expr>\ \text{"})\text{"}\ |\ \text{"}si\text{"}\ <Expr>\ \text{"}alors\text{"}\ <Expr> \\
&\quad \text{"}sinon\text{"}\ <Expr>\ \text{"}fsi\text{"} \\
<Relation> &\rightarrow Ident\ <Op>\ Ident \\
<Op> &\rightarrow \text{" = "}\ |\ \text{" <> "}\ |\ \text{" < "}\ |\ \text{" > "}\ |\ \text{" >= "}\ |\ \text{" <= "}
\end{aligned}
\tag{1}
$$

1.2 Preuve de la propriété LL(1)

Cette grammaire possède trois éléments avec des successeurs non uniques : $<SuiteExpr>$, $<SuiteTermb>$ et $<Facteurb>$.

Pour $<SuiteExpr>$:
$premier(\text{"}ou\text{"}\ <Termb><SuiteExpr>) = \{\text{"}ou\text{"}\}$ et $premier(\epsilon) = \emptyset$,
donc $premier(\text{"}ou\text{"}\ <Termb><SuiteExpr>) \cap premier(\epsilon) = \emptyset$
$null(<SuiteExpr>)$ avec $premier(<SuiteExpr>) = \{\text{"}ou\text{"}\}$ et
$suivant(<SuiteExpr>) = \{<Termb>\}$ toujours différent de {"ou"}
$premier(<SuiteExpr>) \cap suivant(<SuiteExpr>) = \emptyset$

Pour $<SuiteTermb>$:
$premier(\text{"}et\text{"}\ <Facteurb><SuiteTermb>) = \{\text{"}et\text{"}\} etpremier(\epsilon) = \emptyset$,
donc $premier(\text{"}et\text{"}\ <Facteurb><SuiteTermb>) \cap premier(\epsilon) = \emptyset$
$null(<SuiteTermb>)$ avec $premier(<SuiteTermb>) = \{\text{"}et\text{"}\}$ et
$suivant(<SuiteTermb>) = \{<Facteurb>\}$ toujours différent de {"et"}
$premier(<SuiteTermb>) \cap suivant(<SuiteTermb>) = \emptyset$

Pour $<Facteurb>$:
$premier(<Relation>) = Ident$ qui commence toujours par une lettre,

$premier(\text{"("} <Expr> \text{")"}) = \{\text{"("}\}$ et
$premier(\text{"si"} <Expr> \text{"alors"} <Expr> \text{"sinon"} <Expr> \text{"fsi"}) = \{\text{"si"}\}$ donc
$premier(<Relation>) \cap premier(\text{"("} <Expr> \text{")"}) \cap premier(\text{"si"} <Expr> \text{"alors"} <Expr>$
$\text{"sinon"} <Expr> \text{"fsi"}) = \emptyset$
$nonnull(<Facteurb>)$

2 Questions

2.1

Les commentaires ne sont pas nécessaires pour exécuter le code ; c'est pour cela qu'on les ignore dans l'analyse syntaxique.

2.2

En construisant un arbre, on vérifie directement que les types de chaque unité correspondent. Ainsi, le parenthésage est, entre autres, déjà géré, on n'a donc pas besoin d'implémenter une pile.

2.3

La propriété LL(1) d'une grammaire garantit que celle-ci ne contient pas d'ambiguïté, c'est à dire que pour une unité lexicale et un token lu, il n'y qu'une seule règle de dérivation possible. La grammaire est déterministe. Ainsi, l'implémentation du compilateur est grandement simplifiée.

2.4

L'intérêt d'un arbre abstrait est surtout la lisibilité. En effet, il est plus plaisant de retrouver l'expression de base en lisant simplement l'arbre de gauche à droite, plutôt qu'en essayant de discerner les racines et noeuds à retenir, comme c'est le cas avec un arbre concret.

3 Code source

Listing 1 – ulex.ml

```
1  (** [token] is the type of the different lexical units. *)
2  type token =
3         |UL_IDENT of string
4         | UL_ET
5         | UL_OU
6         | UL_EGAL
7         | UL_EOF
8         | UL_PAROUV
9         | UL_PARFERM
10        | UL_DIFF
11        | UL_SUP
```

```
12        | UL_INF
13        | UL_SUPEGAL
14        | UL_INFEGAL
15
16 (** [is_eof] : token  -> bool
17     is_eof tk returns true if the lexical unit represents the end_of
          file.
18 *)
19 let is_eof = function
20   | UL_EOF -> true
21   | _      -> false
22
23 (** [print_token] : out_channel -> token -> unit
24     print_token o tk prints on the output channel o the textual
          representation of the token tk *)
25 let print_token o = function
26   | UL_IDENT ident -> Printf.fprintf o "UL_IDENT %s" ident
27   | UL_ET -> Printf.fprintf o "UL_ET"
28   | UL_OU -> Printf.fprintf o "UL_OU"
29   | UL_EGAL -> Printf.fprintf o "UL_EGAL"
30   | UL_PAROUV -> Printf.fprintf o "UL_PAROUV"
31   | UL_PARFERM -> Printf.fprintf o "UL_PARFERM"
32   | UL_DIFF -> Printf.fprintf o "UL_DIFF"
33   | UL_SUP -> Printf.fprintf o "UL_SUP"
34   | UL_INF -> Printf.fprintf o "UL_INF"
35   | UL_SUPEGAL -> Printf.fprintf o "UL_SUPEGAL"
36   | UL_INFEGAL -> Printf.fprintf o "UL_INFEGAL"
37   | UL_EOF  -> Printf.fprintf o "UL_EOF"
```

Listing 2 – anasyn.ml

```
1  open List;;
2
3  type vt = UL_EOF
4      | UL_ERR
5      | UL_IDENT of string
6      | UL_PAROUV
7      | UL_PARFERM
8      | UL_ET
9      | UL_OU
10     | UL_EGAL
11     | UL_DIFF
12     | UL_SUP
13     | UL_INF
14     | UL_SUPEGAL
15     | UL_INFEGAL
16     | UL_SI
17     | UL_ALORS
18     | UL_SINON
19     | UL_FSI;;
20
21 type vn = FILE
22     | EXPR
23     | SUITEEXPR
24     | TERMB
25     | SUITETERMB
26     | FACTEURB
27     | RELATION
28     | OP;;
```

```
29
30 type v = Vt of vt | Vn of vn;;
31
32 type arbre_concret = Leaf of vt | Node of vn * arbre_concret list;;
33
34 type arbre_abstrait =
35   | Cond of arbre_abstrait * arbre_abstrait * arbre_abstrait
36   | Comp of string * vt * string
37   | Ou of arbre_abstrait * arbre_abstrait
38   | Et of arbre_abstrait * arbre_abstrait;;
39
40 exception DerivationException of vn * vt;;
41 exception MatchException of arbre_concret;;
42
43 let derivation = function
44   | (FILE, _) -> [Vn EXPR; Vt UL_EOF]
45   | (EXPR, (UL_PAROUV | UL_SI | UL_IDENT _)) -> [Vn TERMB; Vn SUITEEXPR]
46   | (SUITEEXPR, UL_OU) -> [Vt UL_OU; Vn EXPR]
47   | (SUITEEXPR, (UL_EOF | UL_ALORS | UL_SINON | UL_FSI)) -> []
48   | (TERMB, (UL_PAROUV | UL_SI | UL_IDENT _)) -> [Vn FACTEURB; Vn
       SUITETERMB]
49   | (SUITETERMB, UL_ET) -> [Vt UL_ET; Vn TERMB]
50   | (SUITETERMB, (UL_OU | UL_EOF | UL_ALORS | UL_SINON | UL_FSI)) -> []
51   | (FACTEURB, UL_PAROUV) -> [Vt UL_PAROUV; Vn EXPR; Vt UL_PARFERM]
52   | (FACTEURB, UL_SI) -> [Vt UL_SI; Vn EXPR; Vt UL_ALORS; Vn EXPR; Vt
       UL_SINON; Vn EXPR; Vt UL_FSI]
53   | (FACTEURB, UL_IDENT _) -> [Vn RELATION]
54   | (RELATION, UL_IDENT x) -> [Vt (UL_IDENT x); Vn OP; Vt (UL_IDENT x)]
55   | (OP, ((UL_EGAL | UL_DIFF | UL_SUP | UL_INF | UL_SUPEGAL |
       UL_INFEGAL) as x)) -> [Vt x]
56   | (x, y) -> raise(DerivationException(x, y));;
57
58 let rec analyse_caractere = function
59   | ((Vt _), p::r) -> (Leaf p, r)
60   | (Vn nterm, p::r) -> let listeTerm = derivation (nterm, p) in
61       let (listeV, listeUL) = analyse_mot(listeTerm, p::r) in
62       ((Node (nterm, listeV)), listeUL)
63   | (_, []) -> raise (Failure "analyse_caractere")
64 and analyse_mot = function
65   | ([], liste) -> [], liste
66   | (pTerm::rTerm, listeUL) -> let (ac, newList) =
       analyse_caractere(pTerm, listeUL) in
67                                let (suiteListeV, newList) =
                                     analyse_mot(rTerm, newList) in
68                                (ac::suiteListeV, newList);;
69
70 let rec abstraire = function
71   | Node (FILE, [a; Leaf UL_EOF]) -> abstraire a
72   | Node (EXPR, [a; Node (SUITEEXPR, [Leaf UL_OU; b])]) -> Ou
       (abstraire a, abstraire b)
73   | Node (EXPR, [a; Node (SUITEEXPR, [])]) -> abstraire a
74   | Node (TERMB, [a; Node (SUITETERMB, [Leaf UL_ET; b])]) -> Et
       (abstraire a, abstraire b)
75   | Node (TERMB, [a; Node (SUITETERMB, [])]) -> abstraire a
76   | Node (RELATION, [Leaf (UL_IDENT ident_a); Node(OP, [Leaf op]); Leaf
       (UL_IDENT ident_b)]) -> Comp(ident_a, op, ident_b)
77   | Node (FACTEURB, [Leaf UL_PAROUV; a; Leaf UL_PARFERM]) -> abstraire a
78   | Node (FACTEURB, [a]) -> abstraire a
```

```
79   | Node (FACTEURB, [Leaf UL_SI; a; Leaf UL_ALORS; b; Leaf UL_SINON; c;
       Leaf UL_FSI]) -> Cond(abstraire a, abstraire b, abstraire c)
80   | x -> raise(MatchException x);;
```

4 Tests

Listing 3 – tests.ml

```
1  (* Test 1 : t < y et x = y *)
2  let (arbConc1, listeVide) = analyse_caractere (Vn FILE, [UL_IDENT "t";
       UL_SUP; UL_IDENT "y"; UL_ET; UL_IDENT "x"; UL_EGAL; UL_IDENT "y";
       UL_EOF]);;
3  (*
4  val arbConc1 : arbre_concret =
5    Node (Fichier,
6     [Node (Expr,
7       [Node (Termb,
8         [Node (Facteurb,
9           [Node (Relation,
10            [Leaf (UL_IDENT "t"); Node (Op, [Leaf UL_INF]); Leaf
                 (UL_IDENT "y")])]);
11          Node (SuiteTermb,
12           [Leaf UL_ET;
13            Node (Termb,
14             [Node (Facteurb,
15               [Node (Relation,
16                 [Leaf (UL_IDENT "x"); Node (Op, [Leaf UL_EGAL]);
17                  Leaf (UL_IDENT "y")])]);
18              Node (SuiteTermb, [])])])]);
19       Node (SuiteExpr, [])]);
20      Leaf UL_EOF])
21 val listeVide : vt list = []
22 *)
23
24 (* Test 2 : On essaye davoir larbre abstrait correspondant  lexpression
       prcdente *)
25 let arbAbstr = construit_arbre_abstrait(arbConc1);;
26 (*
27 #     val arbAbstr : arbre_abstrait =
28   Et (Comp ("t", UL_SUP, "t"), Comp ("x", UL_EGAL, "y"))
29 *)
30
31 (* Test 3 : On essaye davoir larbre concret dune expression
       volontairement fausse (manque du Fsi) :  Si a Alors b Sinon c *)
32 let (arbConc2, listeVide) = analyse_caractere (Vn FILE, [UL_SI;
       UL_IDENT "a"; UL_ALORS; UL_IDENT "b"; UL_SINON; UL_IDENT "c";
       UL_EOF]);;
33 (*
34 Exception : Failure
35 *)
36
37 (* Test 4 : (a <= b) ou a = c *)
38 let (arbConc3, listeVide) =  analyse_caractere(Vn FICHIER,[UL_PAROUV;
       UL_IDENT "a"; UL_INFEGAL; UL_IDENT "b"; UL_PARFERM; UL_OU; UL_IDENT
       "a"; UL_EGAL; UL_IDENT "c"; UL_EOF]);;
39 (*
```

```
#           val arbConc : arbre_concret =
  Node (FILE,
   [Node (Expr,
     [Node (Termb,
       [Node (Facteurb,
         [Leaf UL_PAROUV;
          Node (Expr,
           [Node (Termb,
             [Node (Facteurb,
               [Node (Relation,
                 [Leaf (UL_IDENT "a"); Node (Op, [Leaf UL_INFEGAL]);
                  Leaf (UL_IDENT "b")])]);
              Node (SuiteTermb, [])]);
            Node (SuiteExpr, [])]);
          Leaf UL_PARFERM]);
        Node (SuiteTermb, [])]);
      Node (SuiteExpr,
       [Leaf UL_OU;
        Node (Expr,
         [Node (Termb,
           [Node (Facteurb,
             [Node (Relation,
               [Leaf (UL_IDENT "a"); Node (Op, [Leaf UL_EGAL]);
                Leaf (UL_IDENT "c")])]);
            Node (SuiteTermb, [])]);
          Node (SuiteExpr, [])])])]);
    Leaf UL_EOF])
val listeVide : unite_lexicale list = []
*)

(* Test 5 : On essaye davoir larbre abstrait correspondant  lexpression
   prcdente *)
let arbAbstr = construit_arbre_abstrait(arbConc3);;
(*
#     val arbAbstr : arbre_abstrait =
     Ou (Comp ("a", UL_INFEGAL, "b"), Comp ("a", UL_EGAL, "c"))
*)

(* Test 6 : On essaye davoir larbre concret dune expression
   volontairement fausse (deux OU  suivre) :  a ou ou b et c *)
let (arbConc2, listeVide) = analyse_caractere (Vn FILE, [UL_IDENT "a";
   UL_OU; UL_OU; UL_IDENT "b"; UL_ET; UL_IDENT "c"; UL_EOF]);;
(*
Exception : Failure
*)
```