Rapport Compilation - TP 2 : Analyse syntaxique descendante

Nicolas Desfeux
Renaud Philippe

26 novembre 2010

Table des matières

1 Mise en forme de la grammaire

Lors du précédent TP, nous avons utilisé une grammaire qui n'est pas adapté à l'analyse syntaxique.

1.1 Adaptation de la grammaire

Pour pouvoir utiliser la grammaire pour créer l'analyseur syntaxique, nous avons du lui apporter les modifications suivantes :

- Supression des règles qui ne concernent pas l'analyse syntaxique.
- Supression des expressions régulière.

Nous allons donc travailler avec la grammaire suivante :

```
1 <Expr>          --> <Termb> <SuiteExpr>
2 <SuiteExpr>     --> "ou" <Termb> <SuiteExpr> | Epsilon
3
4 <Termb>         --> <Facteurb> <SuiteTermb>
5 <SuiteTermb>    --> "et" <Facteurb> <SuiteTermb> | Epsilon
6
```

```
 7 <Facteurb>      --> <Relation> | "(" <Expr> ")" | "si" <Expr> "
     alors"
 8                        <Expr> "sinon" <Expr> "fsi"
 9 <Relation>      --> <Ident> <Op> <Ident>
10
11 <Op>           --> "=" | "<>" | "<" | ">" | ">=" | "<="
```

1.2 Démonstration du caractère LL(1)

Pour prouver que cette grammaire est LL(1), nous avons utliser le corollaire vu en cours. Il nous faut étudier chaque règles, et pour chacune d'elle vérifier les conditions suivantes :

- Les intersections des ensembles de premiers des parties droites de la règle sont premiers deux à deux.
- Si null(règle), alors l'intersection des ensembles premier et suivant de la partie gauche de la règle doit être vide.
- Si null(règle), alors il existe un unique chemin qui de la partie gauche de la règle mène à epsilon.

On étudie donc chaque règle de notre grammaire :

<Expr> La partie droite de la règle ne contient qu'un seul élément, elle vérifie donc la première condition. On a !null(<Expr>), et donc cette règle vérifie donc les trois conditions.

<SuiteExpr> La partie droite de la règle un élément et epsilon, l'intersection des ensemble de premier sera donc vide, puisque premier(epsilon)={}. Elle vérifie donc la première condition. On a null(<SuiteExpr>). premier(<SuiteExpr>) = {"ou"} et suivant(<SuiteExpr>)=epsilon, la seconde condition est donc respectée. Il n'y a qu'un seul chemin pour accéder à epsilon, la troisième condition est donc respecté. Cette règle vérifie donc les trois conditions.

<Termb> La partie droite de la règle ne contient qu'un seul élément, elle vérifie donc la première condition. On a !null(<Termb>), et donc cette règle vérifie donc les trois conditions.

<SuiteTermb> La partie droite de la règle un élément et epsilon, l'intersection des ensemble de premier sera donc vide, puisque premier(epsilon)={}. Elle vérifie donc la première condition. On a null(<SuiteExpr>). premier(<SuiteTermb>) = {"et"} et suivant(<SuiteTermb>)=epsilon, la seconde condition est donc respectée. Il n'y a qu'un seul chemin pour accéder à epsilon, la troisième condition est donc respecté. Cette règle vérifie donc les trois conditions.

<Facteurb> premier(<Relation>)=<Lettre>
premier("(" <Expr> ")")="("
premier("si" <Expr> "alors" <Expr> "sinon" <Expr> "fsi")="si"
Toutes les intersections des ensembles de premiers sont vides deux à deux. Cette règle vérifie donc la première condition. On a !null(<Facteurb>), et donc cette règle vérifie donc les trois conditions.

<Relation> La partie droite de la règle ne contient qu'un seul élément, elle vérifie donc la première condition. On a !null(<Relation>), et donc cette règle vérifie donc les trois conditions.

<Op> La partie droite de cette règle n'est constitué que de non terminaux, tous différents. L'intersection des ensembles premiers sera donc vide. On a !null(<Op>), et donc cette règle vérifie donc les trois conditions.

Toutes les règles respectent les conditions du corollaire, on peut donc en déduire que

La grammaire est LL(1).

2 Implémentation de l'analyse syntaxique descendante

2.1 Code de notre analyseur

Listing 1 – Analyseur syntaxique descendant déterministe

```
1  open Lex;;
2  open List;;
3  type ul=unite_lexicale;;
4
5  type vn=
6    Expr
7  | SuiteExpr
8  | Termb
9  | SuiteTermb
10 | Facteurb
11 | Relation
12 | Op;;
13
14 type vt=
15   Vt_ou
16 | Vt_et
17 | Vt_parg
18 | Vt_pard
19 | Vt_si
20 | Vt_alors
21 | Vt_sinon
22 | Vt_fsi
23 | Vt_ident of string
24 | Vt_eg
25 | Vt_neg
26 | Vt_inf
27 | Vt_sup
28 | Vt_supeg
29 | Vt_infeg
30 | Vt_eof;;
```

```
31
32 type v = VN of vn | VT of vt ;;
33
34 type arbre_concret = Feuille of vt
35         | Noeud of (vn*arbre_concret list);;
36
37
38
39 let derivation (vn,ul) = match (vn,ul) with
40           (Expr,_) -> [VN(Termb);VN(SuiteExpr)]
41         | (SuiteExpr,U_ou) -> [VT(Vt_ou);VN(Termb);VN(SuiteExpr
              )]
42         | (SuiteExpr,_) -> []
43         | (Termb,_) -> [VN(Facteurb);VN(SuiteTermb)]
44         | (SuiteTermb,U_et) -> [VT(Vt_et);VN(Facteurb);VN(
              SuiteTermb)]
45         | (SuiteTermb,_) -> []
46         | (Facteurb,U_paro) -> [VT(Vt_parg);VN(Expr);VT(Vt_pard
              )]
47         | (Facteurb,U_si) -> [VT(Vt_si);VN(Expr);VT(Vt_alors);
              VN(Expr);VT(Vt_sinon);VN(Expr);VT(Vt_fsi)]
48         | (Facteurb,_) -> [VN(Relation)]
49         | (Relation,_) -> [VT(Vt_ident"");VN(Op);VT(Vt_ident""
              )]
50         | (Op,U_eg) -> [VT(Vt_eg)]
51         | (Op,U_neg) -> [VT(Vt_neg)]
52         | (Op,U_inf) -> [VT(Vt_inf)]
53         | (Op,U_sup) -> [VT(Vt_sup)]
54         | (Op,U_supeg) -> [VT(Vt_supeg)]
55         | (Op,_) -> [VT(Vt_infeg)];; (*Correspond a infeg, mais
               comme ils sont tous passe avant *)
56
57 let ul_vers_vt ul = match ul with
58                     U_ident(_) -> Vt_ident ""
59                   | U_paro -> Vt_parg
60                   | U_parf -> Vt_pard
61                   | U_sup -> Vt_sup
62                   | U_inf -> Vt_inf
63                   | U_eg -> Vt_eg
64                   | U_neg -> Vt_neg
65                   | U_supeg -> Vt_supeg
66                   | U_infeg -> Vt_infeg
67                   | U_et -> Vt_et
```

```
68                  | U_ou -> Vt_ou
69                  | U_si -> Vt_si
70                  | U_alors -> Vt_alors
71                  | U_sinon -> Vt_sinon
72                  | U_fsi -> Vt_fsi
73                  | _ -> Vt_eof;;
74
75
76
77
78  exception Etat_incorrect of string;;
79
80  let rec analyse_caractere (v,ul) = match (v,ul) with
81    | (VT(Vt_et),(U_et)::reste)-> (Feuille Vt_et,reste)
82    | (VT(Vt_et),_)-> raise(Etat_incorrect("Mauvais ul"))
83    | (VT(Vt_ou),(U_ou)::reste)-> (Feuille Vt_ou,reste)
84    | (VT(Vt_ou),_)-> raise(Etat_incorrect("Mauvais ul"))
85    | (VT(Vt_parg),(U_paro)::reste)-> (Feuille Vt_parg,reste)
86    | (VT(Vt_parg),_)-> raise(Etat_incorrect("Mauvais ul"))
87    | (VT(Vt_pard),(U_parf)::reste)-> (Feuille Vt_pard,reste)
88    | (VT(Vt_pard),_)-> raise(Etat_incorrect("Mauvais ul"))
89    | (VT(Vt_si),(U_si)::reste)-> (Feuille Vt_si,reste)
90    | (VT(Vt_si),_)-> raise(Etat_incorrect("Mauvais ul"))
91    | (VT(Vt_sinon),(U_sinon)::reste)-> (Feuille Vt_sinon,reste)
92    | (VT(Vt_sinon),_)-> raise(Etat_incorrect("Mauvais ul"))
93    | (VT(Vt_alors),(U_alors)::reste)-> (Feuille Vt_alors,reste)
94    | (VT(Vt_alors),_)-> raise(Etat_incorrect("Mauvais ul"))
95    | (VT(Vt_fsi),(U_fsi)::reste)-> (Feuille Vt_fsi,reste)
96    | (VT(Vt_fsi),_)-> raise(Etat_incorrect("Mauvais ul"))
97    | (VT(Vt_ident ""),(U_ident(id1))::reste) -> (Feuille (
          Vt_ident id1),reste)
98    | (VT(Vt_ident ""),_)-> raise(Etat_incorrect("Mauvais ul"))
99    | (VT(Vt_eg),(U_eg)::reste)-> (Feuille Vt_eg,reste)
100   | (VT(Vt_eg),_)-> raise(Etat_incorrect("Mauvais ul"))
101   | (VT(Vt_neg),(U_neg)::reste)-> (Feuille Vt_neg,reste)
102   | (VT(Vt_neg),_)-> raise(Etat_incorrect("Mauvais ul"))
103   | (VT(Vt_sup),(U_sup)::reste)-> (Feuille Vt_sup,reste)
104   | (VT(Vt_sup),_)-> raise(Etat_incorrect("Mauvais ul"))
105   | (VT(Vt_inf),(U_inf)::reste)-> (Feuille Vt_inf,reste)
106   | (VT(Vt_inf),_)-> raise(Etat_incorrect("Mauvais ul"))
107   | (VT(Vt_infeg),(U_infeg)::reste)-> (Feuille Vt_infeg,reste)
108   | (VT(Vt_infeg),_)-> raise(Etat_incorrect("Mauvais ul"))
109   | (VT(Vt_supeg),(U_supeg)::reste)-> (Feuille Vt_supeg,reste)
```

```
110     | (VT(Vt_supeg),_)-> raise(Etat_incorrect("Mauvais␣ul"))
111     | (VN(tn),a::reste)-> let (acl,ullist)= analyse_mot (
          derivation(tn,a),a::reste) in (Noeud(tn,acl),ullist)
112     | (VN(tn),_)-> raise(Etat_incorrect("Mauvais␣ul"))
113     | (_,_)-> raise(Etat_incorrect("Mauvais␣ul"))
114
115 and
116
117 analyse_mot (vl,ull) = match (vl,ull) with
118   |([],ull)-> ([],ull)
119   |([a],ull)-> let (al,l)=analyse_caractere (a,ull) in ([al],l)
120   |(a::reste,ull) ->
121 let (ac,ullist2)=analyse_caractere(a,ull) in
122 let (acl,ullist)=analyse_mot (reste,ullist2) in
123   (ac::acl,ullist);;
124
125 let vlist = [VN Expr];;
126 let ullist= [U_ident("t");U_sup;U_ident("y");U_et;U_paro;
      U_ident("x");U_eg;U_ident("y");U_parf;U_eof];;
127
128 let arbre = analyse_mot(vlist,ullist);;
129
130
131 let vlist2 = [VN Expr];;
132 let ullist2= [U_ident("t");U_ident("y");U_et;U_paro;U_ident("x"
      );U_eg;U_ident("y");U_parf;U_eof];;
133
134 let arbre2 = analyse_mot(vlist2,ullist2);;
135 (* Les tests fonctionnent *)
136
137 (*
138 Seance 3 - Construction de l'arbre syntaxique abstrait
139 *)
140
141 type operator =
142    Op_eg
143   | Op_neg
144   | Op_sup
145   | Op_supeg
146   | Op_inf
147   | Op_infeg;;
148
149 type arbre_abstrait =
```

```
150     Expr_ou of arbre_abstrait * arbre_abstrait
151   | Expr_et of arbre_abstrait * arbre_abstrait
152   | Expr_si of arbre_abstrait * arbre_abstrait* arbre_abstrait
153   | Relations of string * operator * string;;
154
155
156 let rec convert ab = match ab with
157           Noeud(Expr,[a;Noeud(SuiteExpr, [])]) -> convert a
158         | Noeud(Expr,[terme1; Noeud(SuiteExpr, [(Feuille Vt_ou)
              ; terme2; suite])]) -> Expr_ou((convert terme1), (
              convert(Noeud(Expr,[terme2;suite]))))
159         | Noeud(Termb,[fact; Noeud(SuiteTermb,[])]) -> convert
              fact
160         | Noeud(Termb,[fact1; Noeud(SuiteTermb,[(Feuille Vt_et)
              ; fact2; suite])]) -> Expr_et((convert fact1), (
              convert(Noeud(Termb,[fact2;suite]))))
161         | Noeud(Facteurb,[a]) -> convert a
162         | Noeud(Facteurb,[(Feuille Vt_parg); expr;(Feuille
              Vt_pard)]) -> convert expr
163         | Noeud(Facteurb,[(Feuille Vt_si); expr1; (Feuille
              Vt_alors); expr2; (Feuille Vt_sinon); expr3;(Feuille
               Vt_fsi)]) ->
164             Expr_si((convert expr1),(convert expr2),(convert
                  expr3))
165         | Noeud(Relation,(Feuille (Vt_ident id1))::Noeud(
              operator,(Feuille oper)::[])::(Feuille (Vt_ident id2
              ))::[]) -> Relations(id1, (match oper with
166                                       Vt_inf -> Op_inf
167                                     | Vt_infeg -> Op_infeg
168                                     | Vt_sup -> Op_sup
169                                     | Vt_supeg -> Op_supeg
170                                     | Vt_eg -> Op_eg
171                                     | Vt_neg -> Op_neg)
172                                , id2);;
173
174
175 ul_vers_vt (U_ident("test"));;
176 VT(Vt_ident "test");;
177 let ([a],b) = arbre;;
178 convert a;;
179
180 let vlist3 = [VN Expr];;
```

```
181 let ullist3= [U_ident("a");U_infeg;U_ident("b");U_ou;U_paro;
      U_ident("x");U_supeg;U_ident("b");U_ou;U_ident("x");U_inf;
      U_ident("a");U_parf;U_eof];;
182
183 let arbre3 = analyse_mot(vlist3,ullist3);;
184
185 let ([arb],reste) = arbre3;;
186 convert arb;;
```

2.2 Test

Nous avons testé plusieurs expressions pour notre analyseur, afin de vérifier son bon fonctionnement, et regarder son comportement en cas d'erreur. Voici les tests que nous avons pratiqué :

```
1 let vlist = [VN Expr];;
2 let ullist= [U_ident;U_sup;U_ident;U_et;U_paro;U_ident;U_eg;
    U_ident;U_parf;U_eof];;
3
4 let arbre = analyse_mot(vlist,ullist);;
5
6 let vlist2 = [VN Expr];;
7 let ullist2= [U_ident;U_ident;U_et;U_paro;U_ident;U_eg;U_ident;
    U_parf];;
8
9 let arbre 2 analyse_mot(vlist2,ullist2);;
```

Normalement, arbre devrait être correct, et nous devrions avoir une erreur avec arbre2. Et c'est effectivement ce que notre analyseur va produire :

```
1 #     val vlist : v list = [VN Expr]
2 # val ullist : Lex.unite_lexicale list =
3   [U_ident; U_sup; U_ident; U_et; U_paro; U_ident; U_eg;
      U_ident; U_parf;
4    U_eof]
5 #   val arbre : arbre_concret list * Lex.unite_lexicale list =
6   ([Noeud
7      (Expr,
8       [Noeud
9          (Termb,
10          [Noeud
11             (Facteurb,
12              [Noeud
13                 (Relation,
14                  [Feuille Vt_ident; Noeud (Op, [Feuille Vt_sup]);
15                   Feuille Vt_ident])])];
```

```
16              Noeud
17               (SuiteTermb,
18                [Feuille Vt_et;
19                 Noeud
20                  (Facteurb,
21                   [Feuille Vt_parg;
22                    Noeud
23                     (Expr,
24                      [Noeud
25                         (Termb,
26                          [Noeud
27                             (Facteurb,
28                              [Noeud
29                                 (Relation,
30                                  [Feuille Vt_ident; Noeud (Op, [
                                        Feuille Vt_eg]);
31                                   Feuille Vt_ident])]);
32                            Noeud (SuiteTermb, [])]);
33                       Noeud (SuiteExpr, [])]);
34                    Feuille Vt_pard]);
35                Noeud (SuiteTermb, [])])];
36          Noeud (SuiteExpr, [])],
37     [U_eof])
38 #      val vlist2 : v list = [VN Expr]
39 # val ullist2 : Lex.unite_lexicale list =
40   [U_ident; U_ident; U_et; U_paro; U_ident; U_eg; U_ident;
       U_parf]
41 #    Exception: Etat_incorrect "Mauvais ul".
```

Nous avons ensuite testé la création des arbres syntaxiques abstrait, et cela fonctionne correctement. Voici le résultat que nous obtenons.

```
1  val a : arbre_concret =
2    Noeud
3     (Expr,
4      [Noeud
5         (Termb,
6          [Noeud
7             (Facteurb,
8              [Noeud
9                 (Relation,
10                 [Feuille (Vt_ident "t"); Noeud (Op, [Feuille
                       Vt_sup]);
11                  Feuille (Vt_ident "y")])]);
12           Noeud
```

```
13            (SuiteTermb,
14             [Feuille Vt_et;
15              Noeud
16               (Facteurb,
17                [Feuille Vt_parg;
18                 Noeud
19                  (Expr,
20                   [Noeud
21                     (Termb,
22                      [Noeud
23                        (Facteurb,
24                         [Noeud
25                           (Relation,
26                            [Feuille (Vt_ident "x");
27                             Noeud (Op, [Feuille Vt_eg]);
28                             Feuille (Vt_ident "y")])]);
29                       Noeud (SuiteTermb, [])]);
30                    Noeud (SuiteExpr, [])]);
31                Feuille Vt_pard]);
32             Noeud (SuiteTermb, [])])]);
33       Noeud (SuiteExpr, [])]
34 val b : Lex.unite_lexicale list = [U_eof]
35 # - : arbre_abstrait =
36 Expr_et (Relations ("t", Op_sup, "y"), Relations ("x", Op_eg, "
     y"))
```

Le résultat obtenu correspond donc au résultat attendu.

Autre test :

```
1  val vlist3 : v list = [VN Expr]
2  val ullist3 : Lex.unite_lexicale list =
3    [U_ident "a"; U_infeg; U_ident "b"; U_ou; U_paro; U_ident "x"
        ; U_supeg;
4     U_ident "b"; U_ou; U_ident "x"; U_inf; U_ident "a"; U_parf;
        U_eof]
5  val arb : arbre_concret =
6    Noeud
7     (Expr,
8      [Noeud
9        (Termb,
10        [Noeud
11          (Facteurb,
12           [Noeud
13             (Relation,
```

```
14              [Feuille (Vt_ident "a"); Noeud (Op, [Feuille
                   Vt_infeg]);
15               Feuille (Vt_ident "b")])]);
16          Noeud (SuiteTermb, [])]);
17       Noeud
18        (SuiteExpr,
19         [Feuille Vt_ou;
20          Noeud
21           (Termb,
22            [Noeud
23              (Facteurb,
24               [Feuille Vt_parg;
25                Noeud
26                 (Expr,
27                  [Noeud
28                    (Termb,
29                     [Noeud
30                       (Facteurb,
31                        [Noeud
32                          (Relation,
33                           [Feuille (Vt_ident "x");
34                            Noeud (Op, [Feuille Vt_supeg]);
35                            Feuille (Vt_ident "b")])]);
36                      Noeud (SuiteTermb, [])]);
37                   Noeud
38                    (SuiteExpr,
39                     [Feuille Vt_ou;
40                      Noeud
41                       (Termb,
42                        [Noeud
43                          (Facteurb,
44                           [Noeud
45                             (Relation,
46                              [Feuille (Vt_ident "x");
47                               Noeud (Op, [Feuille Vt_inf]);
48                               Feuille (Vt_ident "a")])]);
49                         Noeud (SuiteTermb, [])]);
50                      Noeud (SuiteExpr, [])])]);
51               Feuille Vt_pard]);
52            Noeud (SuiteTermb, [])]);
53         Noeud (SuiteExpr, [])])]
54 val reste : Lex.unite_lexicale list = [U_eof]
55 # - : arbre_abstrait =
```

```
56 Expr_ou (Relations ("a", Op_infeg, "b"),
57  Expr_ou (Relations ("x", Op_supeg, "b"), Relations ("x",
      Op_inf, "a")))
```

Le résultat correspond tout à fait à ce que l'on attend.

Nous n'avons malheureusement pas eu le temps d'implémenter la gestion des cas d'erreur pour la construction des arbres concret.

2.3 Questions de compréhension

Question 2.1 Si on incluait les commentaires dans la grammaire définissant les constructions du langage, elle ne serait plus déterministe.

Question 2.2 Nous n'avons ici pas besoin de pile dans l'implémentation car Ocaml possède déjà une pile, et c'est de celle là qu'il va se servir pour l'automate à pile.

Question 2.3 Une grammaire LL(1) permet la création d'un analyseur syntaxique descendant car cela nous assure que la grammaire est sans ambiguïté. S'il y avait la moindre ambiguïté, il serait impossible pour notre grammaire de choisir la bonne règle. On a donc l'assurance de choisir la seule règle possible car il n'y a qu'un seul choix à chaque fois.

Question 2.4 L'arbre abstrait, lié à une syntaxe abstraite, est beaucoup plus utile pour l'analyse syntaxique que l'arbre concret. L'arbre concret va donner une représentation externe de la structure de l'arbre, tandis que l'arbre abstrait donne une structure plus profonde de l'arbre.L C'est de cette structure plus profonde dont nous avons besoin pour l'analyse syntaxique descendante de l'arbre.