Gestion des erreurs

Jusqu'à maintenant, le compilateur s'arrête sur la première erreur rencontrée.
Il est bien évidemment intéressant de pouvoir poursuivre la compilation aussi loin que faire se peut.
On peut imaginer que la procédure ERREUR n'arrête pas l'exécution du compilateur, mais se limite à reporter une erreur.
Il est cependant peu vraisemblable que la suite de la compilation puisse se dérouler sans engendrer la détection d'erreurs curieuses et étranges.
La gestion des erreurs doit être plus fine et autoriser la reprise après erreur dans de bonnes conditions.

Gestion des erreurs

Les programmes de compilation peuvent contenir des erreurs à différents niveaux :
-erreurs lexicales, comme l'écriture erronée d'un identificateur, d'un mot clé ou d'un opérateur ;
-erreurs syntaxiques, comme une expression arithmétique mal parenthésée ;
-erreurs sémantiques, comme un opérateur appliqué à un opérande non compatible ;
-erreurs logiques, comme une boucle sans fin.

Souvent, dans un compilateur, la part la plus importante dans la détection et la récupération sur erreur est centrée autour de l'analyse syntaxique.

Gestion des erreurs

On distingue :
- la récupération après erreur
dont le but est, lors de la détection d'une erreur, de positionner le compilateur dans un état lui permettant de continuer sainement ; et

- la correction d'erreur
dont le but est de corriger des erreurs et de continuer la compilation malgré la présence d'erreurs dans le programme.

Gestion des erreurs : Messages d'erreurs

Il semble important que des messages d'erreurs informatifs soient fournis par le compilateur à la rencontre d'une erreur.
Pour ce faire, on définit une liste d'erreurs (exemple) :

1	identificateur attendu	ID_ERR
2	PROGRAM attendu	PROGRAM_ERR
3	) parenthèse fermante attendue	PAR_FER_ERR
4	end attendu	END_ERR
5	; attendu	PT_VIRG_ERR
6	= attendu	EGAL_ERR
7	begin attendu	BEGIN_ERR
8	erreur dans la partie déclaration	ERR_IN_DECL
9	, attendue	VIRG_ERR
10	erreur dans une constante	ERR_IN_CONST
11	:= attendu	AFFEC_ERR
12	then attendu	THEN_ERR
13	do attendu	DO_ERR
14	erreur dans un facteur (expression erronée)	ERR_IN_EXPR
15	identificateur déclaré deux fois	ERR_DBL_ID
16	identificateur non déclaré	ERR_NO_ID
17	nombre attendu	NUM_ERR
18	affectation non permise	ERR_NO_AFFEC
19	constante entière dépassant les limites	ERR_NUM_DEPASS
20	division par zéro	ERR_DIV_ZERO
21	. point attendu	POINT_ERR
22	(parenthèse ouvrante attendue	PAR_OUV_ERR

Gestion des erreurs : Messages d'erreurs

D'autres erreurs peuvent survenir, nous les qualifierons d'erreurs d'administration ; ce sont par exemple l'impossibilité d'ouvrir le fichier contenant le code à compiler, le dépassement de la capacité d'un tableau....

Une autre erreur pouvant être détectée par l'analyseur lexicale est la fin du fichier de programme dans un commentaire (EOF_IN_COMM_ERR).

Gestion des erreurs : Messages d'erreurs

L'émission des messages d'erreurs est assurée par la procédure ERREUR à l'aide du tableau MESSAGES_ERREUR :

```
type
  ERREURT = (ID_ERR, PROGRAM_ERR, PAR_FER_ERR, END_ERR,
    PT_VIRG_ERR, EGAL_ERR, BEGIN_ERR, ERR_IN_DECL, VIRG_ERR
    ERR_IN_CONST, AFFEC_ERR, THEN_ERR, DO_ERR, ERR_IN_EXPR
    ERR_DBL_ID, ERR_NO_ID, NUM_ERR, ERR_NO_AFFEC,
    ERR_NUM_DEPASS, ERR_DIV_ZERO, POINT_ERR) ;
var
  MESSAGES_ERREUR : array [ERREURT] of STRING ;
```

Gestion des erreurs : Messages d'erreurs

La procédure ERREUR accepte maintenant un paramètre :
procedure ERREUR (ERRNUM:ERREURT) ;
La majorité des appels à ERREUR se font par TESTE, que l'on modifie ainsi :

```
procedure TESTE (T:TOKENS ; ERRNUM:ERREURT) ;
begin
   if TOKEN = T
      then NEXT_TOKEN
      else ERREUR (ERRNUM)
end ;
```

Gestion des erreurs : Messages d'erreurs

On modifie de même TESTE_ET_ENTRE et TESTE_ET_CHERCHE. Ces procédures ne produisent directement des erreurs que parce que le prochain token n'est pas un identificateur (pour l'instant), on a donc :

```
procedure TESTE_ET_CHERCHE (T:TOKENS ; PERMIS:CLASSET) ;
begin
  if TOKEN = T then
    begin
      CHERCHERSYM (PLACESYM, PERMIS) ;
      NEXT  TOKEN
    end
  else
    case T of :
      ID_TOKEN : ERREUR (ID_ERR) ;
    end
end ;
```

Gestion des erreurs : Messages d'erreurs

On modifie de même TESTE_ET_ENTRE , on a donc :

```
procedure TESTE_ET_ENTRE (T:TOKENS ; C:CLASSES) ;
begin
  if TOKEN = T then
    begin
      ENTRERSYM (C) ;
      NEXT_TOKEN
    end
  else
    case T of :
      ID_TOKEN : ERREUR (ID_ERR) ;
    end
end ;
```

Gestion des erreurs : Messages d'erreurs

La manipulation de la table des symboles peut produire des erreurs : identificateur déjà déclaré (ERR_DBL_ID) pour ENTRERSYM et identificateur non trouvé (non déclaré : ERR_NO_ID) pour CHERCHERSYM. On modifie donc ces deux procédures en conséquence par un appel adéquat à ERREUR.

Gestion des erreurs : Récupération après erreur

La *récupération après erreur* est basée sur le modèle suivant. Quand on découvre une erreur, l'analyseur syntaxique élimine les symboles d'entrée les uns après les autres jusqu'à en rencontrer un qui appartienne à un ensemble de synchronisation.
Usuellement, les tokens de synchronisation sont des délimiteurs tels que le point virgule ou le end dont le rôle dans le texte source est bien défini.

La récupération sur erreur peut être implantée comme suit : à chaque appel de procédure analysant un non-terminal de la grammaire, on passe un ensemble des tokens de synchronisation (paramètre SYNCHRO_TOKENS) ; en cas d'erreur, la procédure est chargée de se synchroniser sur un de ces tokens. De plus, au retour, la procédure appelée doit informer l'appelante de la bonne analyse ou de la synchronisation réalisée (paramètre ETAT).

Selon l'approche de la *correction d'erreur*, quand une erreur est découverte, l'analyseur syntaxique peut effectuer des corrections locales, c'est-à-dire qu'il peut modifier un token afin de permettre la poursuite de l'analyse.

Une correction locale typique consisterait à remplacer une virgule par un point-virgule, à détruire un point-virgule excédentaire, ou à insérer un point-virgule manquant.

Il est préalablement nécessaire de définir une liste des corrections, suppressions et ajouts envisageables.

On peut commencer par la liste suivante :

token recherché	token trouvé	traitement
THEN_TOKEN	DO_TOKEN	substitution
DO_TOKEN	THEN_TOKEN	substitution
VIRG_TOKEN	PT_VIRG_TOKEN	substitution
PT_VIRG_TOKEN	VIRG_TOKEN	substitution
POINT_TOKEN	PT_VIRG_TOKEN	substitution
EGAL_TOKEN	AFFEC_TOKEN	substitution
AFFEC_TOKEN	EGAL_TOKEN	substitution
RELOP_TOKEN[1]	AFFEC_TOKEN	substitution par EGAL_TOKEN
PAR_FER_TOKEN	PT_VIRG_TOKEN	insertion
PAR_FER_TOKEN	ADDOP_TOKEN[2]	insertion
PAR_FER_TOKEN	MULOP_TOKEN[3]	insertion
PAR_OUV_TOKEN	ID_TOKEN	insertion

Il est nécessaire de vérifier que les substitutions ainsi réalisées sont valides dans tous les cas ; on modifie la procédure TESTE en conséquence

```
procedure TESTE (T:TOKENS ; ERRNUM:ERREURT) ;
var CORRECTION : booleen ;
begin
  CORRECTION := false ;
  if TOKEN = T then NEXT_TOKEN
  else begin
    case T of
      THEN_TOKEN :
        if TOKEN = DO_TOKEN
        then begin
          (* substitution *)
          CORRECTION := true ;
          TOKEN = THEN_TOKEN ;
          ERREUR_MESS (THEN_ERR)
        end ;
```

On continue la modification de la procedure TESTE pour tous les tokens

```
DO_TOKEN : (* ... *)
VIRG_TOKEN : (* ... *)
PT_VIRG_TOKEN : (* ... *)
POINT_TOKEN : (* ... *)
AFFEC  TOKEN : (* ... *)
(* RELOP_TOKEN *)
EGAL  TOKEN,
DIFF_TOKEN,
INF_TOKEN,
SUP_TOKEN,
INF_EGAL_TOKEN,
SUP_EGAL_TOKEN : (* ... *)
```

La procédure UNGET_TOKEN met à jour une structure telle que le prochain appel de NEXT_TOKEN retourne le token TOKEN et non un token lu sur le programme traité.

```
    PAR_FER_TOKEN :
      if TOKEN in [PT_VIRG_TOKEN, ADDOP_TOKEN, MULOP_TOKEN]
      then begin
         (* insertion du token recherche *)
         CORRECTION := true ;
         UNGET_TOKEN ;
         TOKEN = PAR_FER_TOKEN ;
         ERREUR_MESS (PAR_FER_ERR)
      end ;
    PAR_OUV_TOKEN : (* ... *)
  end ; (* case *)
  if not CORRECTION
  then ERREUR (ERRNUM)
 end (* else *)
end ;
```

Lors de la mise en place de la correction d'erreurs, le concepteur du compilateur doit assurer de ne pas tomber dans une boucle infinie par des insertions répétées de tokens.