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.

Nous allons compléter l'analyseur sémantique construit précedemment pour générer le P-Code correspondant au programme analysé.

Une première chose à faire est de décider de l'allocation des variables. Ensuite, chaque reconnaissance d'une règle de la grammaire déclenche la génération de P-Code à l'aide des procédures GENERER1 (génération d'une instruction P-Code sans opérande) et GENERER2 (génération d'une instruction P-Code à un opérande)

Au niveau du langage à compiler (comme dans les autres langages de haut niveau), on ne se préoccupe pas de la gestion de la mémoire ; on manipule celle-ci par l'intermédiaire de symboles.

C'est le rôle du compilateur d'*allouer* ces symboles en mémoire.
A chaque symbole, le compilateur doit associer un emplacement mémoire dont la taille dépend du type du symbole.

Une manière simple et naturelle de faire est de choisir les adresses au fur et à mesure de l'analyse des déclarations en incrémentant un *offset* qui indique la place occupée par les déclarations précédentes (variable OFFSET).
A la fin des déclarations, il est possible de déterminer l'emplacement mémoire à réserver dans la pile au début de l'exécution du programme (instruction P-Code INT).

Pour chaque symbole, son adresse d'allocation est stockée dans la table des symboles :

```
var
  TABLESYM : array [TABLEINDEX] of record
    NOM : ALFA ;
    CLASSE : CLASSES ;
    ADRESSE : integer
  end ;
  OFFSET : integer ;
```

On modifie la procédure ENTRERSYM pour tenir compte de cette allocation mémoire :

```
procedure ENTRERSYM(C:CLASSES) ;
begin
  if DERNIERSYM = INDEXMAX then ERREUR ;
  DERNIERSYM := DERNIERSYM + 1 ;
  with TABLESYM [DERNIERSYM] do begin
    NOM := SYM ;
    CLASSE := C ;
    if C = VARIABLE then begin
      ADRESSE := OFFSET ;
      OFFSET := OFFSET + 1
    end
  end
end ;
```

Le champ ADRESSE n'est utilisé que pour les variables ;
On peut l'utiliser pour stocker la valeur des constantes ;
On modifiera alors la procédure CONSTS

Génération de début et fin de programme

Une fois l'allocation des données réalisée, il est nécessaire de réserver l'emplacement suffisant dans la pile P-Code. Cette réservation est faite lors de l'analyse d'un BLOCK par la génération d'une instruction P-Code INST :

```
procedure BLOCK ;
begin
   OFFSET := 0 ;
   if TOKEN = CONST_TOKEN then CONSTS ;
   if TOKEN = VAR_TOKEN then VARS ;
   GENERER2 (INT, OFFSET) ;
   INSTS
end ;
```

Génération de début et fin de programme

Lors de la terminaison de l'analyse d'un programme, il est nécessaire de générer une instruction P-Code d'arrêt du programme, HLT :

```
procedure PROGRAM ;
begin
   TESTE (PROGRAM_TOKEN) ;
   TESTE_ET_ENTRE (ID_TOKEN, PROGRAMME) ;
   TEST (PT_VIRG_TOKEN) ;
   BLOCK ;
   GENERER1 (HLT) ;
   if TOKEN <> POINT_TOKEN then ERREUR
end ;
```

Dans un compilateur :

- lorsque l'on termine une procédure d'analyse c'est que l'on a déjà analysé correctement les phrases correspondant aux procédures appelées dans cette procédure.

- Par exemple lors de l'analyse de l'expression a + b, EXPR va appeler TERM deux fois, une fois pour analyser a et une fois pour analyser b ; l'analyse du + se fait au niveau de EXPR. Si les deux appels réussissent (c'est le cas dans notre exemple) et que le + est bien reconnu, EXPR réussit.

Lorsque la génération est dirigée par la syntaxe, on adopte le même raisonnement : lorsqu'une procédure se termine, on considère que la génération des phrases analysées par les procédures appelées est terminée. Il ne reste plus qu'à générer le code pour l'addition, c'est-à-dire simplement l'instruction P-Code ADD.

On commence par le génération des facteurs pour lesquels on laisse sur la pile P-Code une valeur

```
procedure FACT ;
begin
  if TOKEN = ID_TOKEN then begin
    TESTE_ET_CHERCHE (ID_TOKEN, [CONSTANTE, VARIABLE]);
    with TABLESYM [PLACESYM] do
      case CLASSE of
        CONSTANTE : GENERER2 (LDI, ADRESSE) ;
        VARIABLE : begin
          GENERER2 (LDA, ADRESSE) ;
          GENERER1 (LDV)
        end ;
        PROGRAMME : ;
      end
    end
  end ;
  else if TOKEN = NUM_TOKEN then begin
    GENERER2 (LDI, VAL) ;
    NEXT_TOKEN
  end
  else begin
    TESTE (PAR_OUV_TOKEN) ;
    EXPR ;
    TESTE (PAR_FER_TOKEN)
  end
end ;
```

De même, un terme laisse sur la pile P-Code la valeur du terme. Il est nécessaire de mémoriser le token correspondant à l'opération avant l'analyse de l'opérande gauche du terme (variable OP) :

```
procedure TERM ;
var OP : TOKENS ;
begin
  FACT ;
  while TOKEN in [MULT_TOKEN, DIV_TOKEN] do
    begin
      OP := TOKEN : (* memorise l'operation *)
      NEXT_TOKEN ;
      FACT ;
      if OP = MUL_TOKEN
        then GENERER1 (MUL)
        else GENERER1 (DIV)
    end
end ;
```

L'analyse d'une expression laisse aussi une valeur sur la pile P-Code ; le code est similaire à celui de l'analyse des termes :

```
procedure EXPR ;
 var OP : TOKENS ;
 begin
    TERM ;
    while TOKEN in [PLUS_TOKEN, MOINS_TOKEN] do
      begin
        OP := TOKEN ; (* memorise l'operation *)
        NEXT_TOKEN ;
        TERM ;
        if OP = PLUS_TOKEN
          then GENERER1 (ADD)
          else GENERER1 (SUB)
      end
 end ;
```

Génération des conditions

La génération des conditions (procédure COND) est calquée sur celle des expressions.

Génération des instructions simples

Il n'y pas de code à générer lors de l'analyse du bloc d'instructions (INSTS) ou d'une instruction (INST).

On détaillera la génération à réaliser lors de l'analyse d'une instruction d'affectation et des instructions d'entrée/sortie.

Génération d'une affectation

Une affectation A := expression est générée suivant le modèle

LDA <adresse de A>	empile l'adresse de A
<code>	empile la valeur de l'expression
STO	stocke la valeur de l'expression dans A

Génération d'une affectation (suite)

Le P-Code <code> dépose la valeur de expression sur la pile ; il est généré lors de l'analyse de expression par l'appel EXPR. On a donc :

```
procedure AFFEC ;
bcgin
   TESTE_ET_CHERCHE (ID_TOKEN, VARIABLE) ;
   GENERER2 (LDA, TABLESYM [PLACESYM]. ADRESSE) ;
   TESTE (AFFEC_TOKEN) ;
   EXPR ;
   GENERER1 (STO)
end ;
```

Le code à générer pour une instruction d'écriture telle write (e1, e2, e3) est le suivant

<code1>	empile la valeur de l'expression e1
PRN	imprime cette valeur
<code2>	empile la valeur de l'expression e2
PRN	imprime cette valeur
<code3>	empile la valeur de l'expression e3
PRN	imprime cette valeur

Les P-Codes <code1>, <code2> et <code3> sont générés lors de l'analyse des expressions e1, e2 et e3 par des appels à EXPR.
On modifiera la procédure ECRIRE en conséquence.

Le code à générer pour une instruction de lecture telle read (v1, v2, v3) est le suivant :

LDA <adresse de v1>	empile l'adresse de la variable v1
INN	lit un entier, le stocke dans v1
LDA <adresse de v2>	empile l'adresse de la variable v2
INN	lit un entier, le stocke dans v2
LDA <adresse de v3>	empile l'adresse de la variable v3
INN	lit un entier, le stocke dans v3

Modifier la procédure LIRE en conséquence.

Nous étudions ici la génération de code pour les instructions de rupture de séquence (if COND then INST et while COND do INST). Cette génération est faite sur les modèles suivants :

Pour l'instruction if

	code généré pour COND
	if not COND then goto LABEL
	code généré pour INST
LABEL	suite...

Pour l'instruction while.

DEBUT	code généré pour COND
	if not COND then goto LABEL
	code généré pour INST
	goto DEBUT
LABEL	suite...

Le problème est que lors de la génération du saut conditionnel à LABEL, on ne connaît pas encore la valeur de ce label puisqu'on ne connaît pas a priori la longueur du code généré pour INST.

On va donc générer des instructions de saut incomplètes (sans numéro d'instruction) et mémoriser les numéros des instructions incomplètes pour pouvoir les compléter lorsque l'information sera disponible

Cette facilité de compléter une instruction préalablement générée est offerte par l'intermédiaire des deux procédures NEXT_INST et REMPLIR_INST :

procedure NEXT_INST (var COMPT:integer) ;
la procédure NEXT_INST retourne dans le compteur indiquant le numéro de la prochaine instruction qui sera générée ;

procedure REMPLIR_INST (NUM_INST, DEP:integer) ;
la procédure REMPLIR_INST complète l'instruction de saut NUM_INST avec le numéro d'instruction DEP.

On modifie donc la procédure SI en conséquence :

```
procedure SI ;
var SAUT, SUITE : integer ;
begin
   TESTE (IF_TOKEN) ;
   COND ;
   TESTE (THEN_TOKEN) ;
   NEXT_INST (SAUT) ;
   GENERER2 (BZE, 0) ; (* 0 car incomplet *)
   INST ;
   NEXT_INST (SUITE) ;
   REMPLIR_INST (SAUT, SUITE)
end ;
```

On fait de même pour la procédure TANTQUE :

```
procedure TANTQUE ;
var DEBUT, SAUT, SUITE ; integer ;
begin
   TESTE (WHILE_TOKEN) ;
   NEXT_INST (DEBUT) ;
   COND ;
   TESTE (DO_TOKEN) ;
   NEXT_INST (SAUT) ;
   GENERER2 (BZE, 0) ; (* 0 car incomplet *)
   INST ;
   GENERER2 (BRN, DEBUT) ;
   NEXT_INST (SUITE) ;
   REMPLIR_INST (SAUT, SUITE)
end ;
```

On reprend les déclarations

```
type  MNEMONIQUES – (ADD,SUB,MUL,DIV,EQL,NEQ,GTR,LSS,GEQ,LEQ,
        PRN,INN,INT,LDI,LDA,LDV,STO,BRN,BZE,HLT) ;
      INSTRUCTION = record
        MNE : MNEMONIQUES ;
        SUITE : integer
      end
var   PCODE : array [0 .. TAILLECODE] of INSTRUCTION ;
      PC : integer ;
```

Les fonctions de génération de code GENERER1 et GENERER2 s'écrivent simplement :

```
procedure GENERER1 (M:MNEMONIQUES) ;
begin
   if PC = TAILLECODE then ERREUR ;
   PC := PC + 1 ;
   with PCODE [PC] do
      MNE := M
end ;
```

Les fonctions de génération de code GENERER2 :

```
procedure GENERER2 (M:MNEMONIQUES ; A:integer) ;
begin
   if PC = TAILLECODE then ERREUR ;
   PC := PC + 1 ;
   with PCODE [PC] do begin
      MNE := M ;
      SUITE := A
   end
end ;
```

Les procédures NEXT_INST et REMPLIR_INST s'écrivent :

```
procedure NEXT_INST (var COMPT:integer ) ;
begin
   COMPT := PC + 1
end ;
procedure REMPLIR_INST (NUM_INST, DEP : integer) ;
begin
   PCODE [NUM_INST]. SUITE := DEP
end ;
```

Références

Aho, A. V. and Ullman, J. D. (1977) The Principles of Compiler Design, Addison Wesley, Reading, Mass.

Bornat, R. (1979), Understanding and Writing Compilers, Macmillan.

Gries, D. (1971), Compiler Construction for Digital Computers, Wiley, N.Y.