



Conception modulaire de système d'exploitation - Outils pour la programmation modulaire

Martine Lucas

► To cite this version:

Martine Lucas. Conception modulaire de système d'exploitation - Outils pour la programmation modulaire. Réseaux et télécommunications [cs.NI]. Université Joseph-Fourier - Grenoble I; Institut National Polytechnique de Grenoble - INPG, 1977. Français. NNT: . tel-00010388

HAL Id: tel-00010388

<https://theses.hal.science/tel-00010388>

Submitted on 4 Oct 2005

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

ex 013793

C-3

THESE

présentée à

**Université Scientifique et Médicale de Grenoble
Institut National Polytechnique de Grenoble**

pour obtenir le grade de

DOCTEUR INGENIEUR

par

Jean MONTUELLE

et

Martine LUCAS

Informations : 01 37 93 79 33
CNRS - IMPO - 38041
LEONATHIQUE

38041 GRENOBLE CEDEX
FRANCE



CONCEPTION MODULAIRE Tél. 76.51.42 26

DES SYSTEMES D'EXPLOITATION.

ANNEXES



Thèse soutenue le 27 juin 1977 devant la Commission d'Examen :

Monsieur G. VEILLON : Président

Monsieur S. KRAKOWIAK : Rapporteur

Messieurs J. BELLINO

C. BETOURNE : Examineurs

J.P. VERJUS

TABLE DES MATIERES

ANNEXE1	: FIGURES DU CHAPITRE 4 DE LA THESE : METHODE ET EXEMPLE D' APPLICATION	3
ANNEXE2	: FIGURES DU CHAPITRE 5 DE : DESCRIPTION DU PROJET SESAME	17
ANNEXE3	: STRUCTURE DE CONTROLE ET D'ADRESSAGE DU T1600	35
ANNEXE4	: CARTES SYNTAXIQUES DU LANGAGE D'ECRITURE DE SESAME	41

Ministère de l'Éducation
CNRS - INPG - ICSM
MÉDIATIQUE
38041 GRENOBLE CEDEX
FRANCE
Tél. 76.31.42 26

ANNEXES 1

FIGURES DU CHAPITRE 4

DE CONCEPTION MODULAIRE DES SYSTEMES D'EXPLOITATION
METHODE ET EXEMPLE D'APPLICATION.


```
pattern sac (&nb_val : integer; &t_val : type);  
    dummy function aléatoire (inf, sup : integer) : integer;  
        % tire un nombre au hasard entre inf et sup (compris) %  
  
    var S : array [1 .. &nb_val] of &t_val;  
        dernier : 0 .. &nb_val init 0; % contient le nombre de valeurs  
                                dans le sac %  
  
ext procédure vider;  
    begin dernier := 0 end;  
  
ext procédure mettre (e : &t_val);  
    begin dernier := dernier+1;  
        if dernier > &nb_val then trap ("plein")  
            else S [dernier] := e;  
  
    end;  
  
ext function retirer : &t_val;  
    var x : integer;  
    begin if dernier = 0 then trap ("vide")  
        else begin x := aléatoire (1, dernier);  
            retirer := S [x];  
            S[x] := S[dernier]; % remplissage du "trou" créé %  
            dernier := dernier-1;  
  
        end;  
  
    end;  
  
return % retour anormal dans le module %  
    from aléatoire code  
        "incorrect" : fail ("appel aléatoire incorrect")  
    end;  
  
    end;  
  
path vider + mettre + retirer end % exclusion mutuelle %  
  
end.
```

Figure A1.0

```
pattern direct1 (&taille : integer; &type_enreg : type);  
    var A : array [1.. &taille] of &type_enreg;  
        % &taille peut être très grand %  
  
ext procédure écrire (n : integer; e : &type_enreg);  
    begin if (1 ≤ n ≤ &taille) then A[n] := e  
        else trap ("dépassement") ;  
    end;  
  
ext function lire (n : integer) : &type_enreg;  
    begin if (1 ≤ n ≤ &taille) then lire := A[n]  
        else trap ("dépassement") ;  
    end;  
  
path écrire+lire end ;  
end.
```

Figure A1.1

```
pattern séquentiel1 (&taille : integer; &type_enreg : type);  
    var A : array [1.. &taille] of &type_enreg;  
        % &taille peut être très grand %  
    pointeur_écriture, pointeur_lecture : integer init 1;  
  
ext procédure début_écriture;  
    begin pointeur_écriture := 1 end;  
  
ext procédure écrire (e : &type_enreg);  
    begin if (pointeur_écriture ≤ &taille)  
        then begin A [pointeur_écriture] := e;  
            pointeur_écriture := pointeur_écriture+1;  
        end  
        else trap ("dépassement") ;  
    end;  
  
ext procédure début_lecture;  
    begin pointeur_lecture := 1 end;  
  
ext function lire : &type_enreg;  
    begin if (pointeur_lecture < pointeur_écriture)  
        then begin lire := A [pointeur_lecture ];  
            pointeur_lecture := pointeur_lecture+1;  
        end  
        else trap ("fin logique");  
    end;  
  
path début_écriture+écrire+lire end;  
path début_lecture+lire end;  
  
end.
```

Figure A1.2

```

pattern tri (&e : environnement);
    env &e; % les déclarations figurant dans l'environnement &e sont
              insérées à ce niveau %
    dummy module entrée1, entrée2
        (procédure début_lecture;
         function lire : type_enreg);
    dummy module sortie
        (procédure début_écriture;
         procédure écrire (e : type_enreg));
    var indicateur : integer init 1; % multiple de 2 il indique qu'entrée1
                                      a été traité, multiple de 3 qu'entrée2
                                      a été traité %
ext procédure début;
    var t1, t2 : type_enreg;
    begin sortie.début_écriture;
        entrée1.début_lecture;
        entrée2.début_lecture;
        t1 := entrée1.lire;
        t2 := entrée2.lire;
        while indicateur ≠ 6 do
            begin
                while ((ordre(t1, t2)) and (indicateur=1)) or (indicateur =3) do
                    begin sortie.écrire(t1);
                        t1 := entrée1.lire;
                    end;
                while ((ordre(t2, t1)) and (indicateur=1)) or (indicateur=2) do
                    begin sortie.écrire (t2);
                        t2 := entrée2.lire;
                    end;
            end;
        end;
    end;
return from
    entrée1    code "fin logique" : indicateur := indicateur ≠ 2 end;
    entrée2    code "fin logique" : indicateur := indicateur ≠ 3 end;
    end;
end.

```

Figure A1.3

```
pattern public;  
    type type1 = ? % un certain type %  
    ref function f4.lire (n : integer) : type1;  
  
ext procédure début_lecture;  
    begin pointeur := 1 end;  
  
ext function lire : type1;  
    begin lire := f4.lire (pointeur);  
        pointeur := pointeur+1;  
    end;  
  
return from  
    f4 code "dépassement" : trap ("fin logique") end;  
    end;  
  
path début_lecture+lire end;  
end.
```

Figure A1.4

```
pattern public;  
    type type1 = ? % un certain type %  
    ref function un_certain_fichier.lire (n : integer) : type1;  
    var pointeur : integer;  
ext procédure début_lecture;  
    begin pointeur := 1 end;  
ext function lire : type1;  
    begin if n ≤ 500 then  
        begin lire := un_certain_fichier.lire (pointeur);  
            pointeur := pointeur+1;  
        end  
    else trap ("fin logique");  
    end;  
path début_lecture+lire end;  
end.
```

Figure A1.5

```
pattern table (&taille : integer; &valeur : type);  
    type nom_interne = 1 .. &taille; % &taille peut être très grand %  
    var A : array [1 .. &taille] of &valeur;  
    S : set of nom_interne init [1 .. &taille];  
        % S contient les noms "libres" %  
  
ext function ajouter (v : &valeur) : nom_interne;  
    var n : nom_interne;  
    begin if S = [ ] then trap ("dépassement")  
        else begin n := one of S;  
            S := S-[n];  
            A[n] := v;  
            ajouter := n;  
        end;  
    end;  
  
ext function enlever (n : nom_interne) : &valeur;  
    begin if n in S then trap ("inexistant")  
        else begin enlever := A[n];  
            S := S+[n];  
        end;  
    end;  
  
ext function consulter (n : nom_interne) : &valeur;  
    begin if n in S then trap ("inexistant")  
        else consulter := A[n];  
    end;  
  
ext procedure remplacer (n : nom_interne; v : &valeur);  
    begin if n in S then trap ("inexistant")  
        else A[n] := v;  
    end;  
  
path ajouter end;  
path enlever+consulter+remplacer end;  
end.
```

Figure A1.6

```
pattern association (&taille : integer; &clé, &valeur : type);  
  var A : array [1 .. &taille] of record c : &clé; v : &valeur end;  
  dernier : integer init 0;  
  function index (x : &clé) : integer; % recherche séquentielle %  
    var i : integer;  
    begin i := 1;  
      while ((i <= dernier) and (c of A[i] ≠ x)) do i := i+1;  
      index := i; % i > dernier indique un échec %  
    end;  
  
  ext procédure associer (x : &clé; y : &valeur);  
    begin if (dernier = 100) then trap ("dépassement")  
      else begin  
        if (index (x) <= dernier)  
          then trap ("doublon")  
          else begin dernier := dernier+1;  
            A[dernier] := x;  
          end;  
        end;  
      end;  
  
    end;  
  
  ext function trouver (x : &clé) : &valeur;  
    var i : integer;  
    begin i := index(x);  
      if (i > dernier) then trap ("inconnu")  
      else trouver := A[i];  
    end;  
  
  ext function retirer (x : &clé) : &valeur;  
    var i : integer;  
    begin i := index(x);  
      if (i > dernier) then trap ("inconnu")  
      else begin retirer := A[i];  
        A[i] := A[dernier];  
        dernier := dernier-1;  
      end;  
    end;  
  
  path retirer+trouver+associer end;  
end.
```

Figure A1.7

```
pattern écran (&nom_interne, &nom_externe : type);  
  type descripteur : record... end; organisation : (seq, direct);  
  ref function table_descr.ajouter (d : descripteur) : &nom_interne;  
  ref function table_descr.enlever (n : &nom_interne) : descripteur;  
  ref procédure catalogue.associer (f : &nom_externe; n: &nom_interne);  
  ref function catalogue.retirer (f : &nom_externe) : &nom_interne;  
  % les deux fonctions qui suivent manipulent l'intérieur des  
  % descripteurs. Elles font référence à des procédures externes que  
  % nous n'avons pas déclarées %  
  
  function mise_en_forme (org: organisation; t, l : integer) :  
    descripteur;  
  % cette fonction compose un descripteur en fonction des paramètres  
  % et du résultat de l'allocation de mémoire secondaire %  
  procédure défaire (d : descripteur);  
  % cette procédure relache les ressources figurant dans le  
  % descripteur : désallocation en mémoire secondaire %  
  
ext procédure créer (nom_fichier : &nom_externe; org : organisation;  
  taille, long_enreg : integer);  
  
  var d : descripteur; x : &nom_interne;  
  begin d := mise_en_forme (org.taille, long_enreg);  
    x := table_descr.ajouter (d);  
    catalogue.associer (nom_fichier, x);  
  end;  
  
ext procédure détruire (nom_fichier : &nom_externe);  
  var d : descripteur; x : &nom_interne;  
  begin x := catalogue.retirer (nom_fichier);  
    d := table_descr.enlever (x);  
    défaire (d);  
  end;  
  
return from  
  catalogue code "inconnu" : trap ("inconnu");  
    "doublon" : trap ("doublon");  
  end;  
  
end;  
  
end.
```

Figure A1.8

```
pattern direct2 (&nom_externe; &nom_interne : type);  
    type descripteur : record... end;  
    ref function table_descr.consulter (n : &nom_interne) : descripteur;  
    ref procédure table_descr.remplacer (n : &nom_interne; d : descripteur);  
    ref function catalogue.trouver (f : &nom_externe) : &nom_interne;  
    % il faudrait aussi rajouter les références externes faites par  
    % les deux fonctions qui suivent %  
    procédure write (n : integer; tampon : address);  
    procédure read (n : integer; tampon : address);  
    var descr_global : descripteur;  
  
    ext procédure écrire (f : &nom_externe; n : integer; tampon : address);  
        var x : &nom_interne;  
        begin x := catalogue.trouver (f);  
            descr_global := table_descr.consulter (x);  
            write (n, tampon);  
            table_descr.remplacer (x, descr_global);  
            % dernière ligne nécessaire si descr_global a été modifié par write %  
        end;  
  
    ext procédure lire (f : &nom_externe; n : integer; tampon : address);  
        var x : &nom_interne;  
        begin x := catalogue.trouver (f);  
            descr_global := table_descr.consulter (x);  
            read (n, tampon);  
            table_descr.remplacer (x, descr_global);  
            % dernière ligne nécessaire si descr_global a été modifié par read %  
        end;  
    return from  
        catalogue code "inconnu" : trap ("inconnu") end;  
    end;  
  
    path écrire+lire end;  
  
    end.
```

Figure A1.9

```
pattern direct3 (&nom_externe, &nom_interne : type);  
    type descripteur : record... end;  
    ref function table_descr.consulter (n : &nom_interne) : descripteur;  
    ref procédure table_descr.remplacer (n : &nom_interne; d : descripteur);  
    procédure write (n : integer; tampon : address);  
    procédure read (n : integer; tampon : address);  
    var descr_global : descripteur;  
  
    ext procédure écrire (i : &nom_interne; n : integer; tampon : address);  
        begin descr_global := table_descr.consulter (i);  
            write (n, tampon);  
            table_descr.remplacer (x, descr_global);  
        end;  
  
    ext procédure lire (i : &nom-interne; n : integer; tampon : address);  
        begin descr_global := table_descr.consulter (i);  
            read (n, tampon);  
            table_descr.remplacer (x, descr_global);  
        end;  
  
    path lire+écrire end  
end.
```

Figure A1.10

```
pattern voie_directe (&f : ident; &type-enreg : type);  
    ref procédure direct.écrire (d : 1..2000; n : integer; t : address);  
    ref procédure direct.lire (d : 1..2000; n : integer; t : address);  
    ref function catalogue.trouver (f : ident) : 1..2000;  
    var nom_descr : 0..2000 init 0; % contiendra le nom interne %  
  
ext procédure ouvrir;  
    begin nom_descr := catalogue.trouver (&f) end;  
  
ext procédure écrire (n : integer; e : &type-enreg);  
    begin if nom_descr = 0  
        then trap ("non ouvert")  
        else direct.écrire (nom_descr, n, ad (e));  
            % ad est une fonction standard qui fournit l'adresse  
            d'une variable %  
    end;  
  
ext function lire (n : integer) : &type_enreg;  
    begin if nom_descr = 0  
        then trap ("non ouvert")  
        else direct.lire (nom_descr, n, ad (lire));  
    end;  
  
ext procédure fermer;  
    begin nom_descr := 0 end;  
  
retrun from  
    catalogue code "inconnu" : trap ("inconnu") end;  
    end;  
  
path ouvrir+fermer+écrire end;  
path ouvrir+fermer+lire end;  
end.
```

Figure A1.11

```
pattern voie_modification_séquentielle_sur_direct (&f : ident; &type_enreg :  
                                                    type);  
    ref procédure direct.écrire (d : 1..2000; n : integer; t : address);  
    ref function catalogue.trouver (f : ident) : 1..2000;  
    var nom_descr : 0..2000 init 0;  
        position : integer;  
  
ext procédure ouvrir;  
    begin nom_descr := catalogue.trouver (&f);  
        position := 1;  
    end;  
  
ext procédure avancer;  
    begin if (nom_descr = 0)  
        then trap ("non ouvert")  
        else position := position+1;  
    end;  
  
ext procédure écrire;  
    begin if (nom_descr = 0)  
        then trap ("non ouvert")  
        else direct.écrire (nom_descr, position, ad (e));  
    end;  
  
ext procédure lire;  
    begin if (nom_descr = 0)  
        then trap ("non ouvert")  
        else direct.lire (nom_descr, position, ad (lire));  
    end;  
  
ext procédure fermer;  
    begin nom_descr := 0 end;  
  
return from  
    direct code "dépassement" : trap ("dépassement") end;  
    catalogue code "inconnu" : trap ("inconnu") end;  
    end;  
  
path ouvrir+avancer+écrire+lire+fermer end;  
end.
```

Figure A1.12

```
pattern fichier (&nom : string; &org : organisation; &taille : integer;  
                                     &enreg : type);  
    ref procédure interface.créer (n : ident; 0 : organisation;  
                                     t, l : integer);  
    ref procédure interface.détruire (n : ident);  
ext procédure créer;  
    begin interface.créer (&nom, &org, &taille, &enreg) end;  
ext procédure détruire;  
    begin interface.détruire (&nom) end;  
end.
```

Figure A1.13

ANNEXE 2

FIGURES DU CHAPITRE 5
DE CONCEPTION MODULAIRE DES SYSTEMES D'EXPLOITATION
DESCRIPTION DU PROJET SESAME.

```

pattern scheduler_1 (&maxpr, &ancêtre : integer);
    % &maxpr doit être inférieur à 125 %

    type nom_processus = 1... &maxpr;
    num_tâche = 1... &maxpr+2;
    % les tâches &maxpr+1 et &maxpr+2 correspondent à des tâches
    % nécessaires à la gestion des autres tâches (voir texte) %
    ref procédure allocateur_pile.préparer (n : nom_processus);
    var libre : set of nom_processus; % noms de processus libres %
    situation : array [1 .. &maxpr+2] of descrp; % PST %
    activable : set of num_tâche; % ASTF %
    % contient les tâches correspondant aux processus activables
    % et actif(et la tâche &maxpr+1 en permanence) %
    en_exécution : num_tâche; % mémoire 0 %
    % ces trois dernières variables sont connues du scheduler :
    % elles figurent en mémoire débanalisée %

    ext procédure initialiser;
        begin libre := [1.. &maxpr] - [&ancêtre];
        activable := [&ancêtre, &maxpr+1];
        en_exécution := &ancêtre;
        end;

    ext function créer (initial : descrp) : nom_processus;
        var p : nom_processus;
        begin entrex if libre = [ ] then begin sortix;
            trap ("saturé");
            end
            else begin p := one of libre;
                libre := libre-[p]
                sortix;
                allocateur_pile.préparer [p];
                situation [p] := initial;
                créer := p;
            end;
        end;

    ext procédure débloquer (n : nom_processus; p : integer);
        begin activable := activable U [n] end;

    ext procédure attendre;
        begin activable := activable -[en_exécution] end;

```

Figure A2.1 (début)

```
ext procédure élire (p : integer);  
    begin ACQ end  
    % ACQ réalise en une seule instruction :  
        sauvegarde des registres dans : situation [en_exécution];  
        en_exécution := one of activable;  
        charge les registres avec : situation [en_exécution]; %  
ext function actif : nom_processus;  
    begin actif := en_exécution end;  
ext function priorité : (n : nom_processus) : integer;  
    begin priorité = prio of situation [n] end ;  
ext procédure mourir;  
    begin activable := activable - [en_exécution];  
        libre := libre U [en_exécution];  
        en_exécution := &maxpr+2;  
    end;  
synchro % débloquent, attendre et mourir doivent être non interruptibles %  
end.
```

Figure A2.1 (fin)

```
pattern scheduler 2 (&maxpr, &ancêtre : integer);
    % &maxpr doit être inférieur à 125 %
    type nom_processus = 1 .. &maxpr;
        num_tâche = 1 .. &maxpr+2;
    ref procédure allocateur_pile.préparer (n : nom_processus);
    ref function file.classer (x,y : num_tâche; p : integer) : num_tâche;
    ref function file.successeur (x : num_tâche) : num_tâche;
    var libre : set of nom_processus; % noms libres de processus %
        premier : num_tâche; % tâche activable ou active la plus prioritaire %
        situation : array [1.. &maxpr+2] of descrip; % PST %
        actif : set of num_tâche; % ASTF %
        % contient un seul numéro : celui de la tâche en exécution ou
        % celle que l'on désire élire %
        en_exécution : num_tâche; % mémoire 0 %
        % ces trois dernières variables sont connues du scheduler %

ext procédure initialiser;
    begin libre := [1.. &maxpr] - [&ancêtre];
        actif := [&ancêtre];
        premier := file.classer (&maxpr+1, &ancêtre, 0);
        en_exécution := ancêtre;
        situation [&maxpr+1] := ?; % voir texte %
    end;

ext function créer (initial : descrip) : nom_processus;
    %comme pour le scheduler_1 %

ext procédure débloquer (n : nom_processus; p : integer);
    begin premier := classer (premier, n, p);
        if premier ≠ en_exécution then actif := [premier];
    end;

ext procédure attendre;
    begin premier := file.successeur (premier);
        actif := [premier];
    end;
```

Figure A2.2 (début)

```
ext procédure élire;  
    begin ACQ end; % voir scheduler_1 %  
  
ext function actif : nom_processus;  
    begin actif := en_exécution end;  
    % est appelé en dehors des périodes de commutation de processus %  
  
ext function priorité (n : nom_processus) : integer ;  
    begin priorité := prio of situation [n] end;  
  
ext procédure mourir;  
    begin premier := file.successeur (premier);  
        actif := [premier];  
        libre := libre U [en_exécution];  
        en_exécution := &maxpr+2;  
    end;  
  
synchro % débloquer, attendre et mourir doivent être non interruptibles %  
end.
```

Figure A2.2 (fin)

```
pattern interruptions(&nbniv, &maxpr : integer);  
    type niveau = 1.. &nbniv; % plus le niveau est élevé plus il est  
        prioritaire  
        cause = ?; % à définir, ici 0 indiquera "information absente"%  
        nom_processus = 1.. &maxpr+1;  
    ref function scheduler.aktif : nom_processus;  
    ref procedure scheduler.attendre;  
    ref procedure scheduler.élire;  
    ref procédure scheduler.débloquer (n:nom_processus; p:integer);  
    ref function file.classer (x,y:nom_processus; p:integer):nom_processus;  
    ref function file.successeur (x:nom_processus):nom_processus;  
    var demandeur : array [1.. &nbniv] of 0.. &maxpr;  
        IT : array [1.. &nbniv] of cause;  
        priorité : array [1.. &nbniv] of integer;  
ext procédure initialiser;  
    begin for i := 1 to &nbniv do  
        begin demandeur [i] := &maxpr+1;  
            IT [i] := 0;  
            priorité [i] := ?; % par exemple fonction croissante de i %  
        end;  
    end;  
ext procédure signaler (n : niveau; c : cause);  
    var d : nom_processus;  
    begin entrex;  
        IT [n] := c;  
        d := demandeur [n];  
        if d ≠ &maxpr+1  
            then begin demandeur := successeur (d);  
                scheduler.débloquer (d, priorité [n]);  
                sortir;  
                Scheduler.élire;  
            end  
        else sortir;  
    end;
```

Figure A2.3 (début)

```
ext function recevoir (n : niveau; p : integer) : cause;  
  begin entrex;  
    if IT [n] = 0  
      then begin % pas d'interruption arrivée %  
        demandeur [n] := classer (demandeur [n] , scheduler.actif, p);  
        scheduler.attendre;  
        sortix;  
        scheduler.élire;  
      end  
    else begin recevoir := IT [n];  
      IT [n] := 0;  
      sortix;  
    end;  
  end;  
end.
```

Figure A2.3 (fin)

```
pattern file_1 (&maxelem: integer);  
    type élément : 1 ..&maxelem+1;  
    var suivant : array [élément] of élément;  
  
ext procédure initialiser;  
    begin end;  
  
ext function classer (x, y : élément; p : integer) : élément;  
    var z : élément;  
    begin if y < x then begin classer := y;  
        suivant [y] := x;  
        end  
    else begin classer := x;  
        z := x; % il faut intercaler y %  
        while y > suivant [z]  
            do z := suivant [z];  
            suivant [y] := suivant [z];  
            suivant [z] := y;  
        end;  
    end;  
  
ext function successeur (x : élément) : élément;  
    begin successeur := suivant [x] end;  
  
ext procédure retirer (x,y : element);  
    var z : élément;  
    begin z := x;  
        while suivant [z] ≠ y  
            do z := suivant [z];  
            suivant [z] := suivant [y];  
        end;  
  
synchro % classer, successeur et retirer sont appelés en section critique %  
end
```

Figure A2.4

```
pattern file_2 (&maxelem : integer);  
    type élément : 1.. &maxelem+1;  
    var suivant : array [élément] of élément;  
  
ext procédure initialiser;  
    begin end;  
  
ext function classer (x, y : élément; p : integer) : élément;  
    var z : élément;  
    begin if x = &maxelem+1 then begin classer := y;  
        suivant [y] := &maxelem+1;  
        end  
    else begin classer := x;  
        z := x; % il faut mettre y à la fin %  
        while suivant [z] ≠ &maxelem + 1  
            do z := suivant [z];  
            suivant [y] := &maxelem+1;  
            suivant [z] := y;  
        end;  
    end;  
  
ext function successeur (x : élément) : élément;  
    begin successeur := suivant [x] end  
  
ext procédure retirer (x : élément);  
    % comme dans file_1 %  
  
synchro % classer, successeur et retirer sont appelés en section critique %  
end
```

Figure A2.5

```

pattern file_3 (&maxelem : integer);
    type élément : 1 .. &maxelem+1;
    dummy function priorité (n : élément) : integer;
    var suivant : array [élément] of élément;
        prior : array [élément] of integer;

ext procédure initialiser;
    begin prior [maxelem+1] := -1 end; % file vide %

ext function classer (x, y : élément; p : integer) : élément;
    var z : élément;
    begin if p = -1 then prior [y] := scheduler.priorité (y);
        else prior [y] := p;
        if prior [y] > prior [x] then begin classer := y;
            suivant [y] := x;
        end
    else begin % il faut intercaler y %
        classer := x;
        z := x;
        while prior [y] ≤ prior [suivant [z]]
            do z := suivant [z];
            suivant [y] := suivant [z];
            suivant [z] := y;
        end;
    end;

ext function successeur (x : élément) : élément;
    begin successeur := suivant [x] end;

ext procédure retirer (x : élément);
    % comme dans file_1 %

synchro % classer, successeur et retirer sont appelés dans une section
    critique %

end.

```

Figure A2.6

```
pattern allocateur_pile_1 (&maxpr : integer; &bas, &haut : address);  
    % &bas et &haut délimitent l'espace réservé aux piles %  
  
    type nom_processus = 1.. &maxpr;  
        nom_bloc = address;  
  
    ref function scheduler.aktif : nom_processus;  
    const taille : integer = (&haut-&bas)/&maxpr; % taille d'une pile %  
    var base_initiale : array [1.. &maxpr+1] of nom_bloc;  
        % contient la valeur de la base du premier bloc de chacune des  
        piles %  
    base_future : array [1.. &maxpr] of nom_bloc;  
        % contient la valeur de la base du prochain bloc à fournir %  
  
    ext procédure initialiser;  
        begin base_initiale [1] := &bas + 128; % la base repère le 128e mot  
            du bloc %  
            for i := 1 to &maxpr  
                do base_initiale [i+1] := base_initiale [i] + taille;  
            end;  
  
    ext procédure préparer (p : nom_processus);  
        begin base_future [p] := base_initiale [p] end;  
  
    ext function fournir (n : integer) : nom_bloc;  
        var p : nom_processus;  
            b : nom_bloc;  
        begin p := scheduler.aktif;  
            b := base_future [p];  
            b := b+n;  
            if b >= base_initiale [p+1] then trap ("dépassement")  
                else begin fournir := base_future [p];  
                    base_future [p] := b;  
                end;  
            end;  
  
    ext procédure rendre (b : nom_bloc);  
        var p : nom_processus;  
        begin p := scheduler.aktif;  
            if (b < base_initiale [p]) or (b >= base_initiale [p+1])  
                then trap ("incompatible")  
                else base_future [p] := b;  
            end;  
  
    ext procédure enlever (p : nom_processus);  
        begin end;  
  
end.
```

```

pattern allocateur_pile_2 (&nb_morceaux, &maxpr : integer; &bas, &haut : address);
    % &bas et &haut délimitent l'espace réservé aux piles %
    type nom_processus = 1 .. &maxpr;
    nom_morceau = 0 .. &nb_morceaux; % 0 correspond au morceau indéfini %
    nom_bloc = address;
    ref function scheduler.actif : nom_processus;
    const taille : integer = (&haut-&bas)/&nb_morceaux; % taille d'un
        morceau %
    var base_initiale : array [1 .. &nb_morceaux+1] of nom_bloc;
        % contient la valeur de la base du 1er bloc de chacun des morceaux %
    base_future : array [0.. &nb_morceaux] of nom_bloc;
        % contient la valeur de la base du prochain bloc à fournir dans
        un morceau %
    pile : array [1.. &maxpr] of nom_morceau;
        % repère le dernier morceau alloué pour cette pile %
    libre : set of 1.. &nb_morceaux; % contient le nom des morceaux
        libres %
    précédent : array [1.. &nb_morceaux] of nom_morceau;
        % pour chaque morceau indique le morceau précédent dans la pile.
        0 indique qu'il s'agit du dernier (fond) %

ext procédure initialiser;
    begin libre := [1 .. &nb_morceaux];
        base_initiale [1] := &bas+128; % la base repère le 128e mot du bloc %
        for i := 1 to &nb_morceaux
            do base_initiale [i+1] := base_initiale [i] + taille;
        base_future [0] := base_initiale [1] -1; % le morceau indéfini est
        plein %
    end;

ext procédure préparer (p : nom_processus);
    begin pile [p] := 0 end;
        % le prochain fournir allouera le vrai premier morceau %

```

Figure A2.8 (début)

```
ext function fournir (n : integer) : nom_bloc;  
  var p : nom_processus;  
    m1, m2 : nom_morceau;  
    b : nom_bloc;  
  begin if n > taille then trap ("dépassement")  
    else begin  
      p := scheduler.actif;  
      m1 := pile [p]  
      b := base_future [m1] + n;  
      if b < base_initiale [m1+1] % même morceau ? %  
        then begin fournir := base_future [m1];  
          base_future [m1] := b;  
        end  
      else if libre = [ ] then trap ("saturé")  
        else begin % on change de morceau %  
          m2 := one of libre;  
          libre := libre - [m2];  
          pile [p] := m2;  
          précédent [m2] := m1;  
          base_future [m2] := base_initiale [m2] + n;  
          fournir := base_initiale [m2];  
        end;  
      end;  
    end;  
end;  
  
ext procédure rendre (b : nom_bloc);  
  var p : nom_processus;  
    m : nom_morceau;  
  begin p := scheduler.actif;  
    m := pile [p];  
    if (b < base_initiale [m]) or (b >= base_initiale [m+1])  
      then trap ("incompatible")  
    else if b = base_initiale [m] % on change de morceau %  
      then begin libre := libreU [m];  
        pile [p] := précédent [m];  
      end;  
    else base_future [m] := b;  
  end
```

Figure A2.8 (continuation)

```
ext procédure enlever (p : nom_processus);  
  var m : nom_morceau;  
  begin m := pile [p];  
    while m ≠ 0 do begin libre := libre U [m];  
      m := précédent [m];  
    end;  
  end;  
  
  synchro % il faudrait mettre les opérations sur libre en exclusion  
    mutuelle %  
  
end.
```

Figure A2.8 (fin)

```

pattern allocateur_pile_3 (&maxpr, &maxblocs : integer; &bas, &haut : address);
    % &bas et &haut délimitent l'espace réservé aux piles %
    type nom_processus = 1.. &maxpr;
    nom_bloc = address;
    numéro = -1... &maxblocs; % -1 correspond à un bloc libre
                                0 correspond au bloc indéfini %
    descr = record précédent : numéro; base : nom_bloc end;
    ref function scheduler.actif : nom_processus;
    var pile : array [1 .. &maxpr] of numéro;
    % contient le numéro du dernier bloc empilé pour un processus %
    région : array [1.. &maxblocs] of descr;
    % contient les descripteurs des blocs existants %
    dernier : numéro; % repère le dernier bloc empilé %
    base_future : nom_bloc; % base du prochain bloc à fournir %

ext procédure initialiser;
    begin dernier := 0;
        base_future := &bas + 128; % la base repère le 128e mot du bloc %
    end;

ext procédure préparer (p : nom_processus);
    begin pile [p] := 0      end; % pile où aucun bloc n'est empilé %

ext function fournir (n : integer) : nom_bloc;
    var p : nom_processus;
    b : nom_bloc;
    begin p := scheduler.actif;
    b := base_future+n;
    if b > &haut then trap ("saturé")
    else begin dernier := dernier+1;
        base of region [dernier] := base_future;
        précédent of region [dernier] := pile [p];
        pile [p] := dernier;
        fournir := base_future;
        base_future := b;
    end;
end;

```

Figure A2.9 (début)

```

ext procédure rendre (b : nom_bloc);
    var p : nom_processus;
        i : numéro;
    begin p := scheduler.actif;
        i := pile [p];
        if (i=0) or (b ≠ base of region [i]) then trap ("incompatible")
        else begin pile [p] := précédent of région [i];
            précédent of région [i] := -1; % bloc libéré %
            if i = dernier % dépile tous les blocs libres %
                then begin
                    while précédent of région [dernier] = -1
                        do dernier := dernier -1;
                    base_future := base of région [dernier+1];
                end;
            end;
        end;

end;

ext procédure enlever (p : nom_processus);
    var i1, i2 : numéro;
    begin i1 := pile [p];
        if i1 ≠ 0 % il reste des blocs dans la pile %
            then begin while i1 ≠ 0 do % libération %
                begin i2 := précédent of région [i1];
                    précédent of région [i1] := -1;
                    i1 := i2;
                end;
            if pile [p] = dernier % dépiler %
                then begin
                    while précédent of région [dernier] = -1
                        do dernier := dernier-1;
                    base_future := base of région [dernier+1];
                end;
            end;
        end;

    end;

synchro % fournir et la dernière instruction conditionnelle de rendre et
    d'enlever sont en exclusion mutuelle %

end.

```

Figure A2.9 (fin)

```

pattern gestion-processus (&maxpr, &ancêtre : integer);
  type contexte = record
    regp : address; % compteur ordinal %
    regc : address; % base des remanents %
    regl : address; % base des locaux %
  end;
  descrp = record contex : contexte; prior : integer end;
  nom_processus : 1 .. &maxpr;
  renseignements = record
    nbfils : integer; % nbre de fils %
    père, fils, frère : 0.. &maxpr;
    % liens de parenté, (0 : lien indéfini) %
  end;
  ref function scheduler.actif : nom_processus;
  ref function scheduler.créer (d : descrp) : nom_processus;
  ref procédure scheduler.débloquer (n : nom_processus; p : integer);
  ref procédure scheduler.attendre;
  ref procédure scheduler.élire
  var PR : array [1 .. &maxpr] of renseignements;

ext procédure initialiser;
  % &ancêtre est le nom prédéfini du processus initial %
  begin with PR [&ancêtre] do begin nbfils := 0;
    fils := 0;
  end;
end;

ext procédure créer (début : address; p : integer);
  var p_créateur, p_crée : nom_processus;
  d_crée : descrp;
  begin p_créateur := scheduler.actif;
    with context of d_crée do begin
      regp := début;
      regc := C; % registre C du créateur %
      regl := L; % registre L du créateur %
    end;
    prior of d_crée := p;
    p_crée := scheduler.créer (d_crée);

```

Figure A2.10 (début)

```

with PR [p_crée] do begin
    nb_fils := 0; % pas de fils %
    fils := 0;
    père := p_créateur; % lien vers le père %
    frère := fils of PR [p_créateur];
    end;
with PR [p_créateur] do begin
    nb_fils := nb_fils+1; % un fils de plus %
    fils := p_crée;
    end;
end;
ext procédure place_aux_jeunes;
    var courant, x : nom_processus;
    begin courant := scheduler.aktif;
        x := fils of PR [courant];
        entrex ;
        while x ≠ 0 do begin scheduler.débloquer (x, 0);
            x := frère of PR [x];
        end;
        scheduler.attendre;
        sortix;
        scheduler.élire;
        fils of PR [courant] := 0; % plus de fils %
    end;
ext procédure terminer;
    var courant, créateur : nom_processus;
    begin courant := scheduler.aktif;
        créateur := père of PR [courant];
        with PR [créateur] do
            begin entrex;
                nbfils := nbfils-1;
                if nbfils = 0 then scheduler.débloquer (créateur, 0);
            end;
            scheduler.mourir;
            sortix;
            scheduler.élire;
        end;
    end.

```

Figure A2.10 (fin)

ANNEXE 3

STRUCTURE DE CONTROLE ET D'ADRESSAGE DU T1600

Le T1600 est un miniordinateur : c'est un monoprocesseur accédant à une mémoire centrale de 4 à 64 K mots de 16 positions binaires.

3.1 L'Unité centrale

3.1.1 Les registres

Les registres rapides (16 positions binaires) peuvent se partager en 3 catégories :

i) Les registres de travail

- A est l'accumulateur (pour des opérations arithmétiques ou logiques).
- B est l'extension de cet accumulateur (pour les instructions travaillant sur 32 positions binaires : décalage, multiplication, division).
- X est le registre d'index (adressage)
- Y est un registre auxiliaire (très peu d'instructions le considèrent comme opérande implicite).

A part leur spécialisation propre, ces quatre registres peuvent être utilisés comme opérandes dans les instructions usuelles de chargement et rangement (avec un emplacement mémoire).

ii) Les registres d'adressage

- C , L et W sont les registres de base.
- K est un pointeur de pile manipulé par certaines instructions.
- P est le compteur ordinal

A part P ces registres ne peuvent être opérandes (explicites) que dans des instructions ne portant que sur des registres.

iii) Les registres internes

S regroupe :

- les indicateurs de condition (code condition),
- le masque des interruptions,
- le mode d'exécution : maître ou esclave.

SLO et *SLE* sont des registres limites pour la protection et la translation dynamique de la zone de mémoire qu'ils délimitent. Ils ne sont pas directement programmables. Ils acquièrent une valeur lorsque les registres sont chargés avec le vecteur d'état d'une tâche.*

3.2.2 La multiprogrammation

A tout instant on considère que le calculateur est en train d'exécuter une tâche. A chaque tâche est associé un vecteur d'état (*PST*) qui sera chargé dans les registres de l'unité centrale, lorsque la tâche est activée, et qui en sauvegardera la valeur lorsque le processeur lui sera retiré.** Certaines tâches sont réservées au traitement des interruptions : elles sont qualifiées par *hard* (les autres par *soft*).

3.2.3 Les interruptions

Les interruptions sont réparties en 8 niveaux hiérarchisés pouvant être masqués séparément. Chaque niveau est divisé en sous-niveaux. Une seule interruption est mémorisée par sous niveau. A chaque niveau est associé une tâche *hard* particulière qui est activée lorsque le niveau n'est pas masqué, qu'il n'y a pas d'autres tâches *hard* plus prioritaires en cours d'activation et qu'une interruption est apparue à l'un de ses sous-niveaux.

L'instruction *ACQ* termine tout traitement de tâche *hard* : elle désactive le sous niveau où est apparue l'interruption. La tâche *hard* ou *soft* la plus prioritaire est alors réactivée.

3.2.4 L'option "scheduler et sémaphores"

L'option de T1600 intitulée "Scheduler et Sémaphores" propose un moniteur microprogrammé permettant de gérer jusqu'à 128 tâches *soft*. Elles sont numérotées de 0 à 127 avec un ordre de priorités décroissantes. Elles sont moins prioritaires que les tâches *hard*. Chacune est identifiée par son numéro (priorité fixe) et possède un vecteur d'état propre.

* Lorsqu'il s'agit de T1600 nous continuerons à employer le mot tâche au lieu du terme processus.

** Seul le masque des interruptions n'est pas modifié par les changements de tâches.

Pour résoudre les problèmes d'exclusion mutuelle et de synchronisation, des opérations sur sémaphores sont proposées en distinguant les deux types : sémaphore d'exclusion mutuelle et sémaphore privé (structure de données différente). Le contenu des sémaphores est initialisé directement.

Pour la gestion des tâches, le moniteur microprogrammé utilise certains mots situés en mémoire débanalisée (adresses basses) selon le schéma (simplifié) de la figure A 3.1. Les valeurs de ces emplacements peuvent être modifiées directement (l'instruction *ACQ* permet d'appeler le microprogramme pour qu'il effectue une commutation de tâches si nécessaire). Elles peuvent aussi être modifiées par des instructions spéciales :

ARM(i) signale qu'une nouvelle tâche de numéro *i* est à prendre en compte, désormais, lors des choix qu'aura à effectuer le microprogramme (met à 1 un booléen de l'*ASTF*).

QUIT signale que la tâche en cours d'exécution est désarmée.

RQST(mutex) et *RLSE(mutex)* sont les opérations classiques *P* et *V* sur le sémaphore d'exclusion *mutex* (peut modifier un booléen de *ESTF*).

WAIT(priv) et *ACT(priv)* sont les opérations *P* et *V* sur le sémaphore privé *priv*.

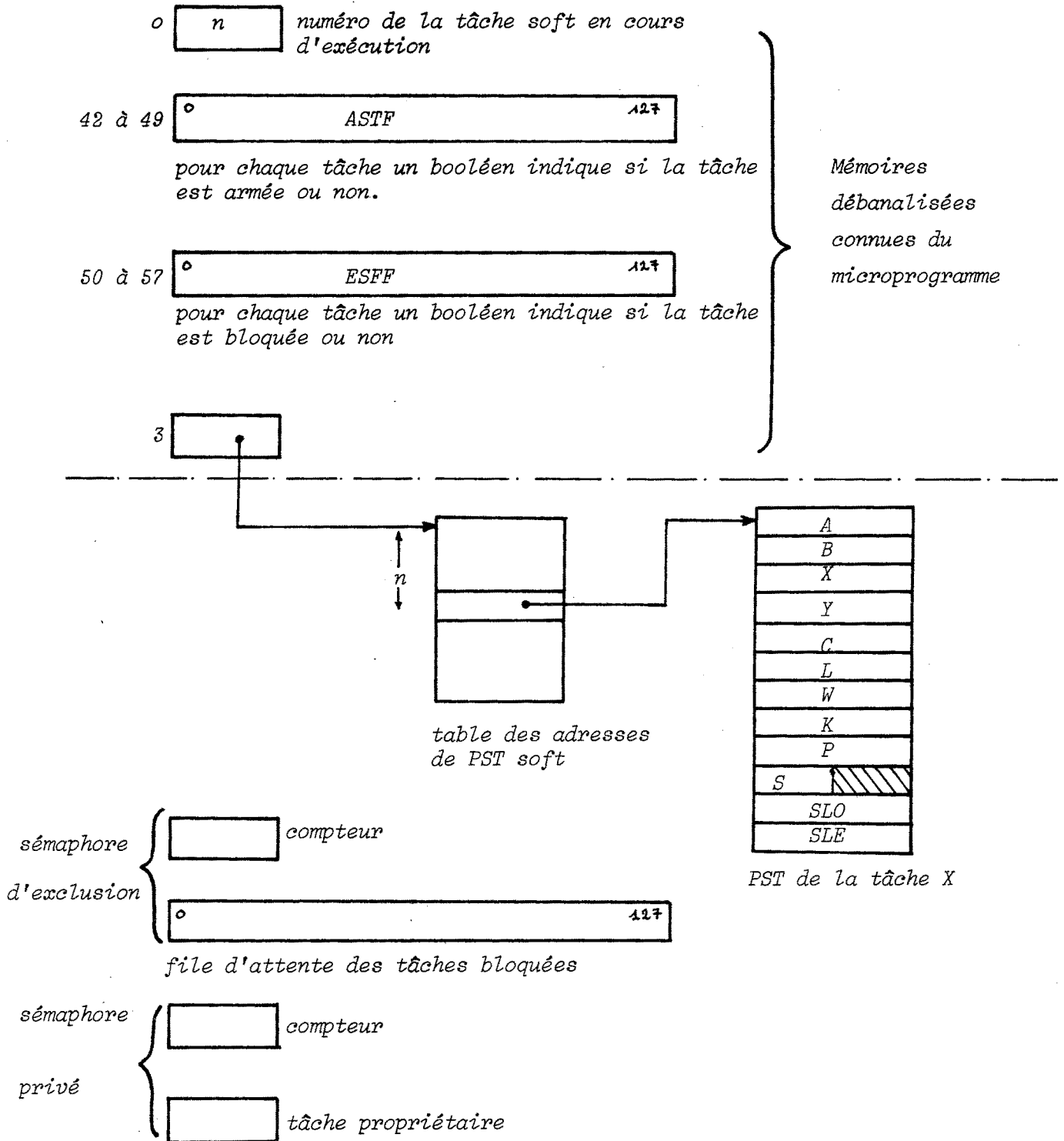
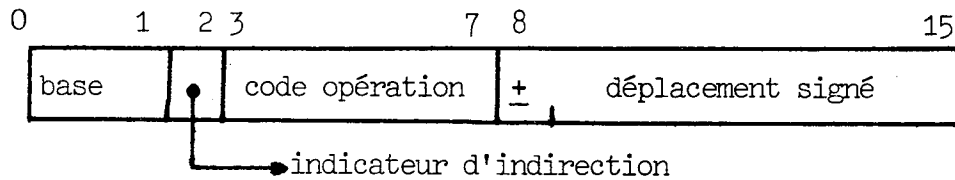


Figure A3.1

3.2. Adressage

3.2.1 Mode d'adressage

Les instructions sont codées sur un mot. Les instructions avec référence mémoire sont découpées en quatre parties :



L'adressage se fait par base+déplacement. Les registres *C*, *L* ou *W* peuvent servir de registres de base.

C est primitivement prévu pour baser les données communes à plusieurs segments de programmes,

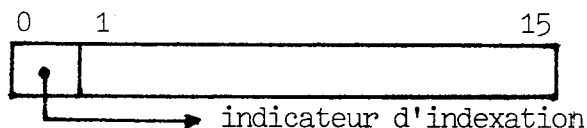
L pour baser les données locales à un programme,

W comme base de travail (pour les autres accès).

Le déplacement est signé ($-128 \leq \text{déplacement} \leq 127$).

Les modes d'adressage possibles sont :

- Adressage direct : adresse de l'opérande = [base]+déplacement.
- Adressage indirect : l'adresse figurant dans l'instruction est celle d'un relais ayant le format suivant :



sans indexation : adresse de l'opérande = [[base]+déplacement]

avec indexation : adresse de l'opérande = [[base]+déplacement]+[X]

L'adressage est un adressage mot. Des instructions spéciales permettent de sélectionner des octets.

Enfin, certaines instructions de rupture de séquence (par exemple tous les sauts conditionnels) se font par déplacement relativement au compteur ordinal *P*.

D'autres instructions permettent des manipulations de piles. Il s'agit des instructions d'empilement et de dépilement utilisant un registre spécial (*K*) comme pointeur de sommet de pile.

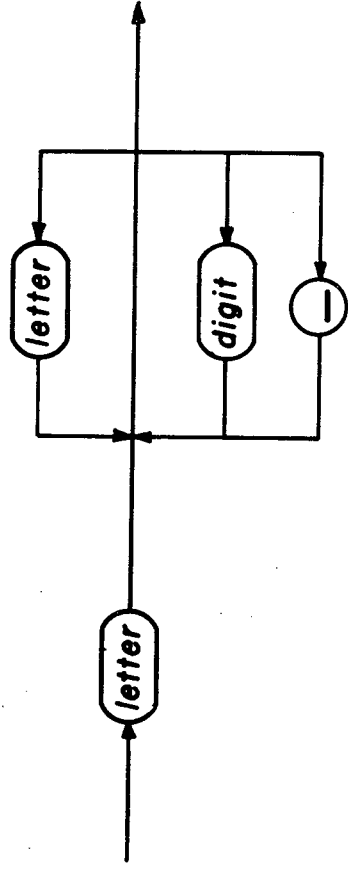
3.2.2 Protection

Une option de T1600 intitulée "translation et protection dynamique de mémoire" autorise l'emploi des deux registres *SLO* et *SLE*. En présence de cette option, toute adresse référencée en mode esclave est considérée comme relative au contenu du registre *SLO* qui lui est donc ajouté. Si l'adresse obtenue est supérieure au contenu de *SLE* un déroutement est effectué.

ANNEXE 4

CARTES SYNTAXIQUES DU LANGAGE D'ECRITURE
DE SESAME.

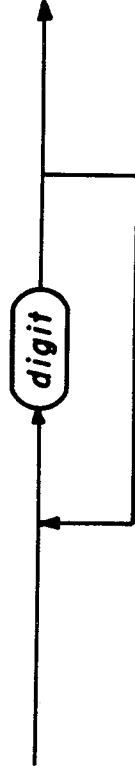
identifier



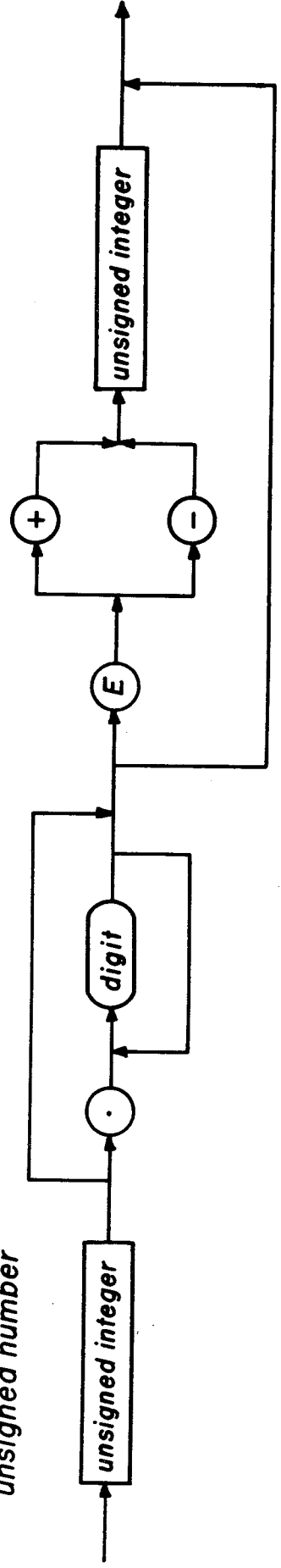
meta idf



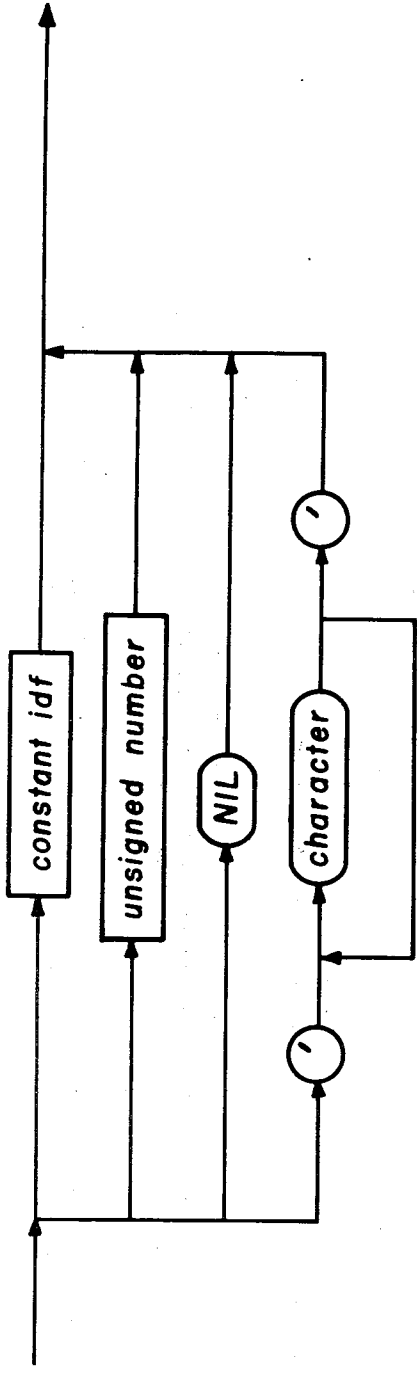
unsigned integer



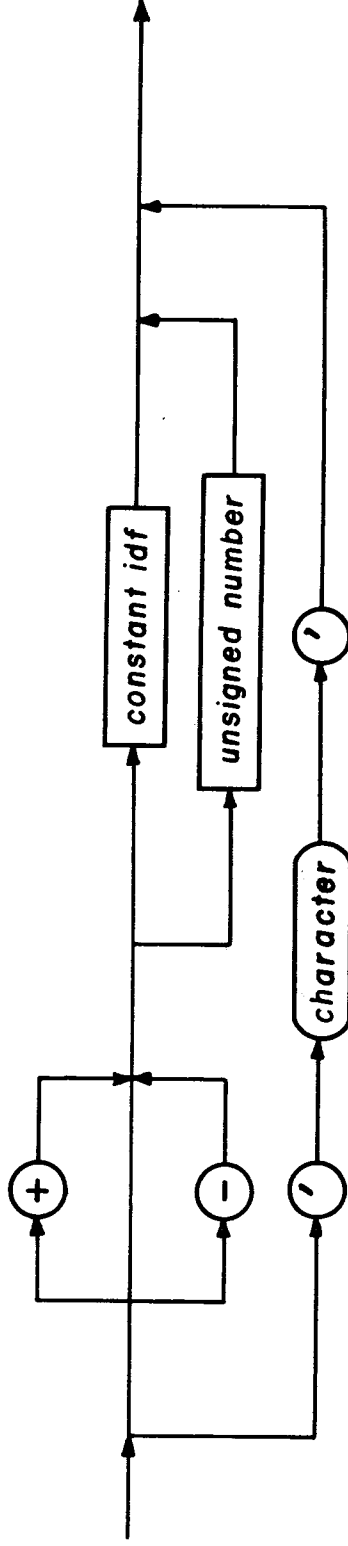
unsigned number



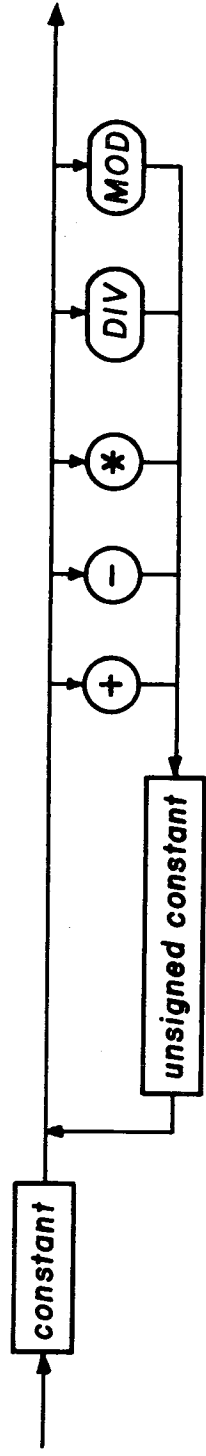
unsigned constant



constant



constexpr

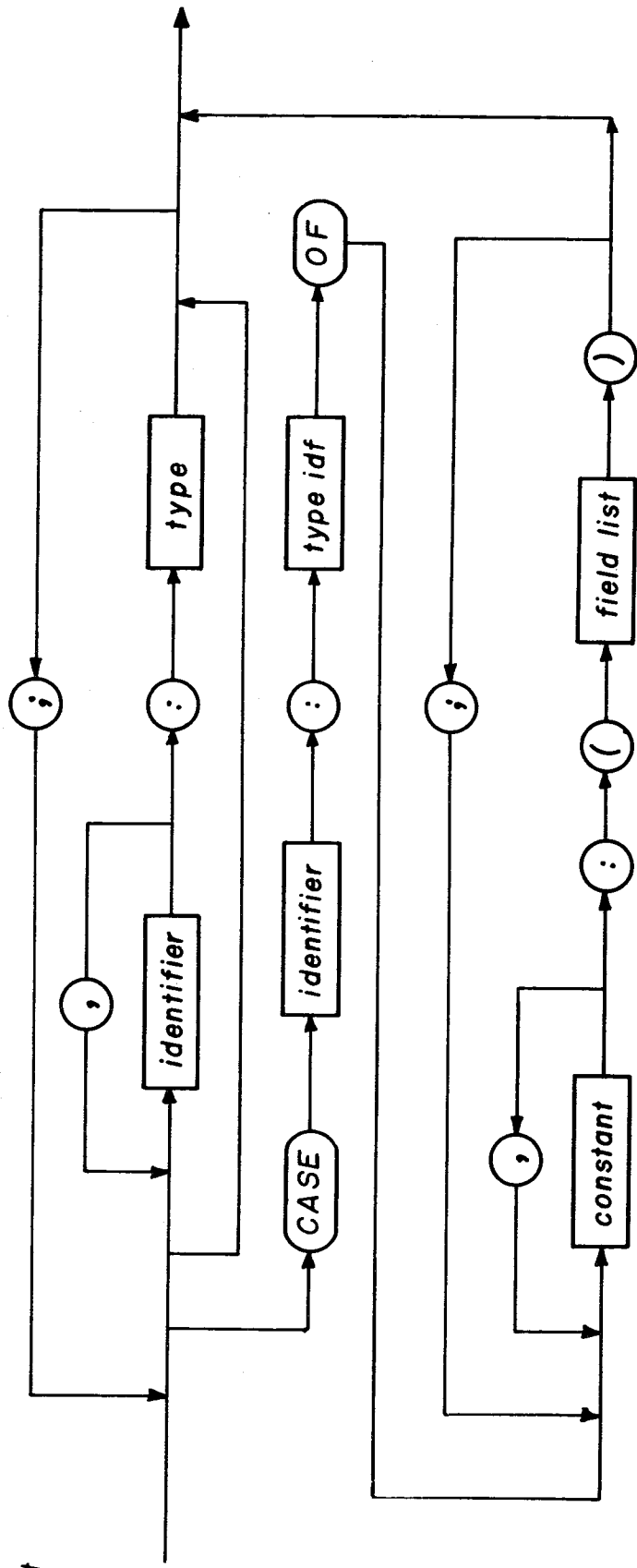


[illegible]

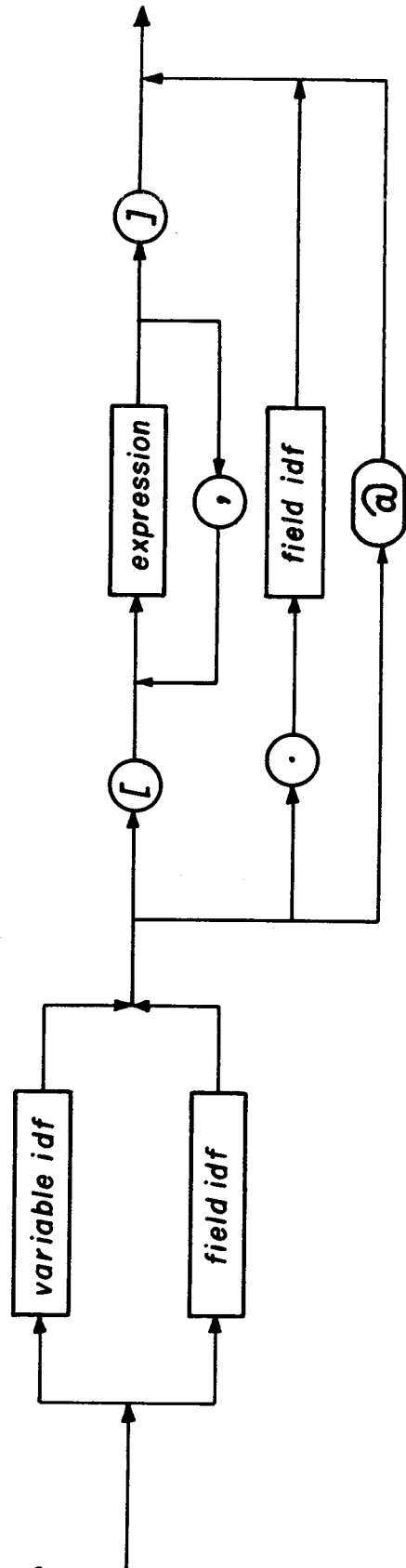
```

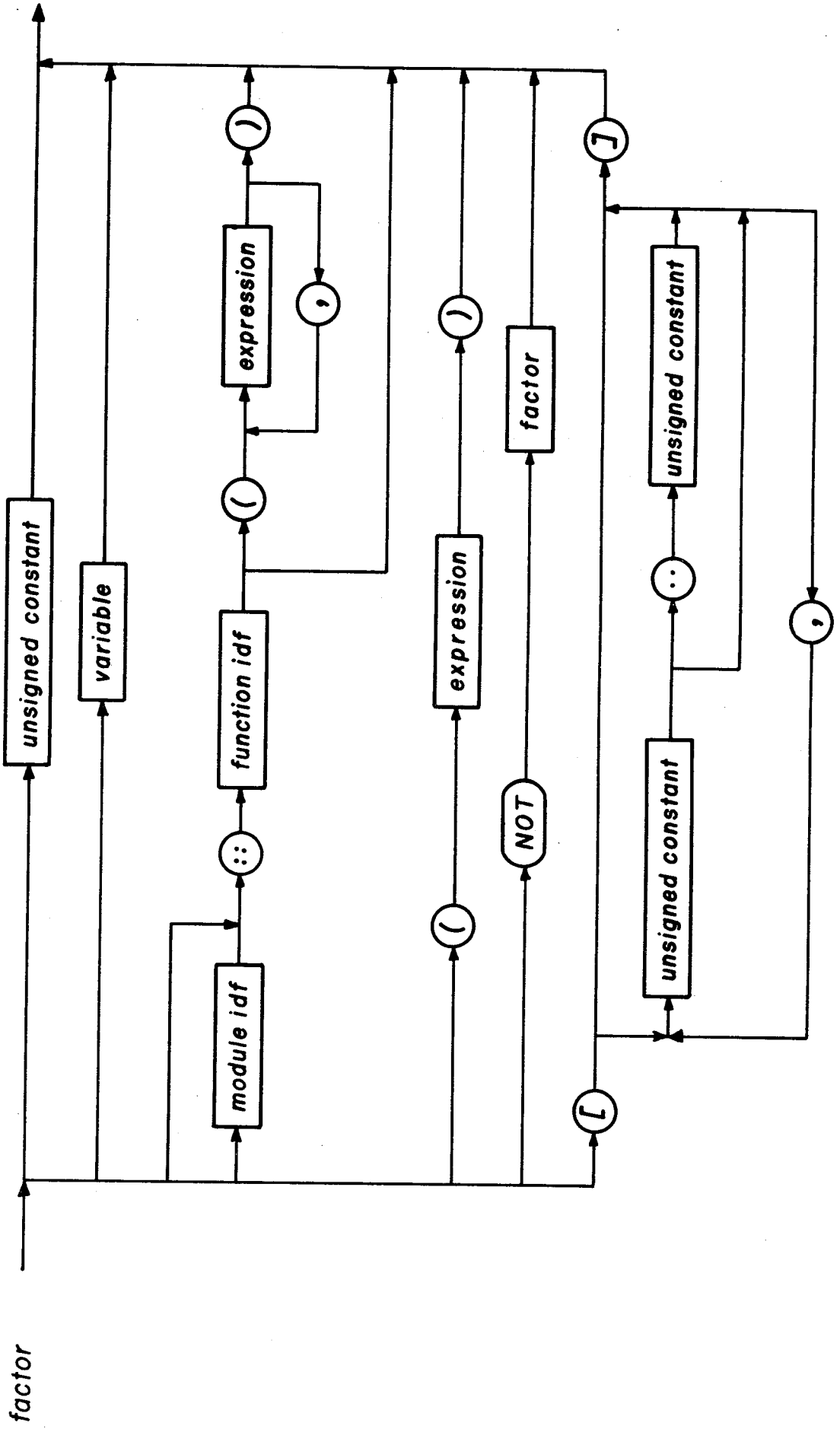
graph LR
    S1[simple type] -- id --> S2[type idf]
    S2 -- type --> S3[simple type]
    S3 -- '[' --> S4[array]
    S3 -- type --> S5[type]
    S4 -- ']' --> S6[simple type]
    S4 -- SET --> S7[SET]
    S6 -- ']' --> S8[simple type]
    S6 -- OF --> S9[OF]
    S8 -- ']' --> S10[simple type]
    S8 -- OF --> S11[OF]
    S10 -- ']' --> S12[field list]
    S10 -- END --> S13[END]
    S12 -- RECORD --> S14[RECORD]
    S14 -- ']' --> S15[field list]
    S14 -- END --> S16[END]
    S15 -- END --> S17[END]
  
```

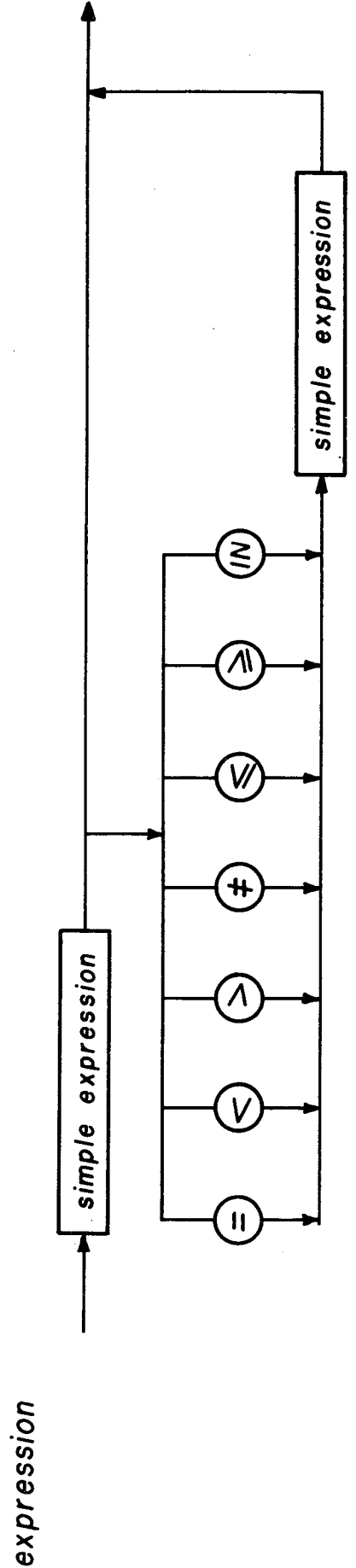
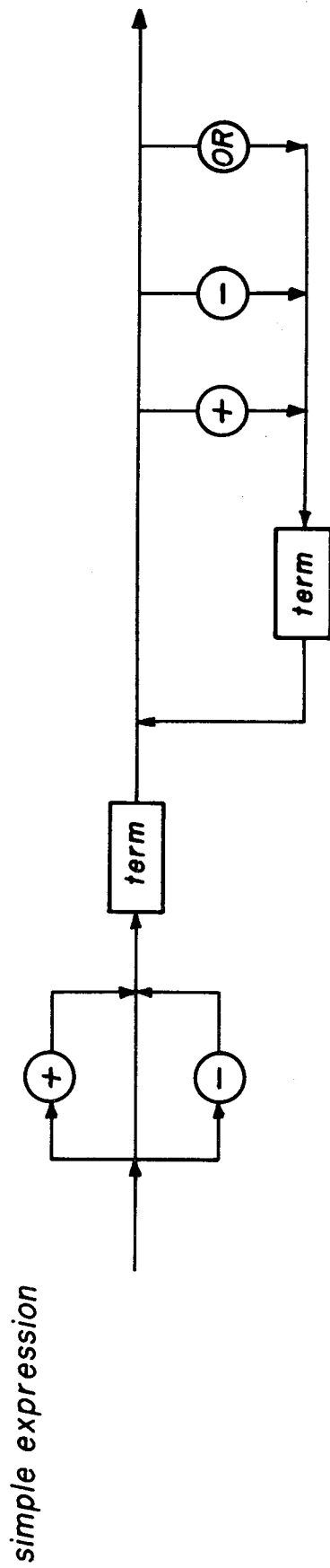
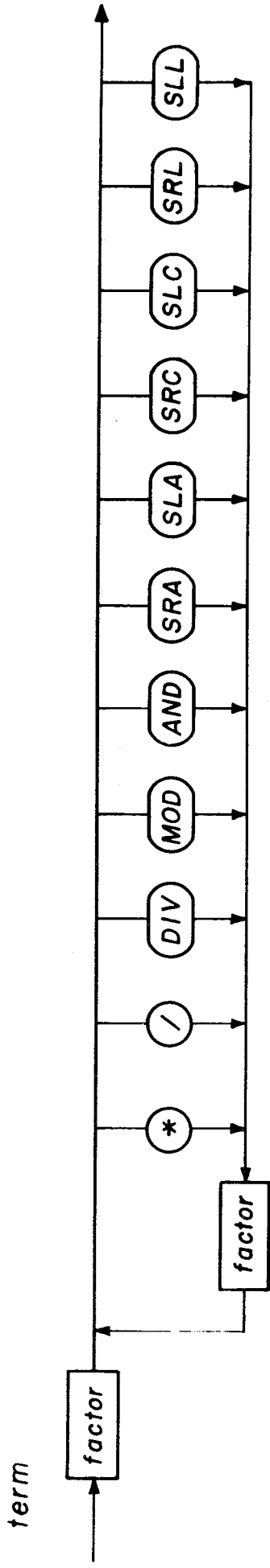
field list



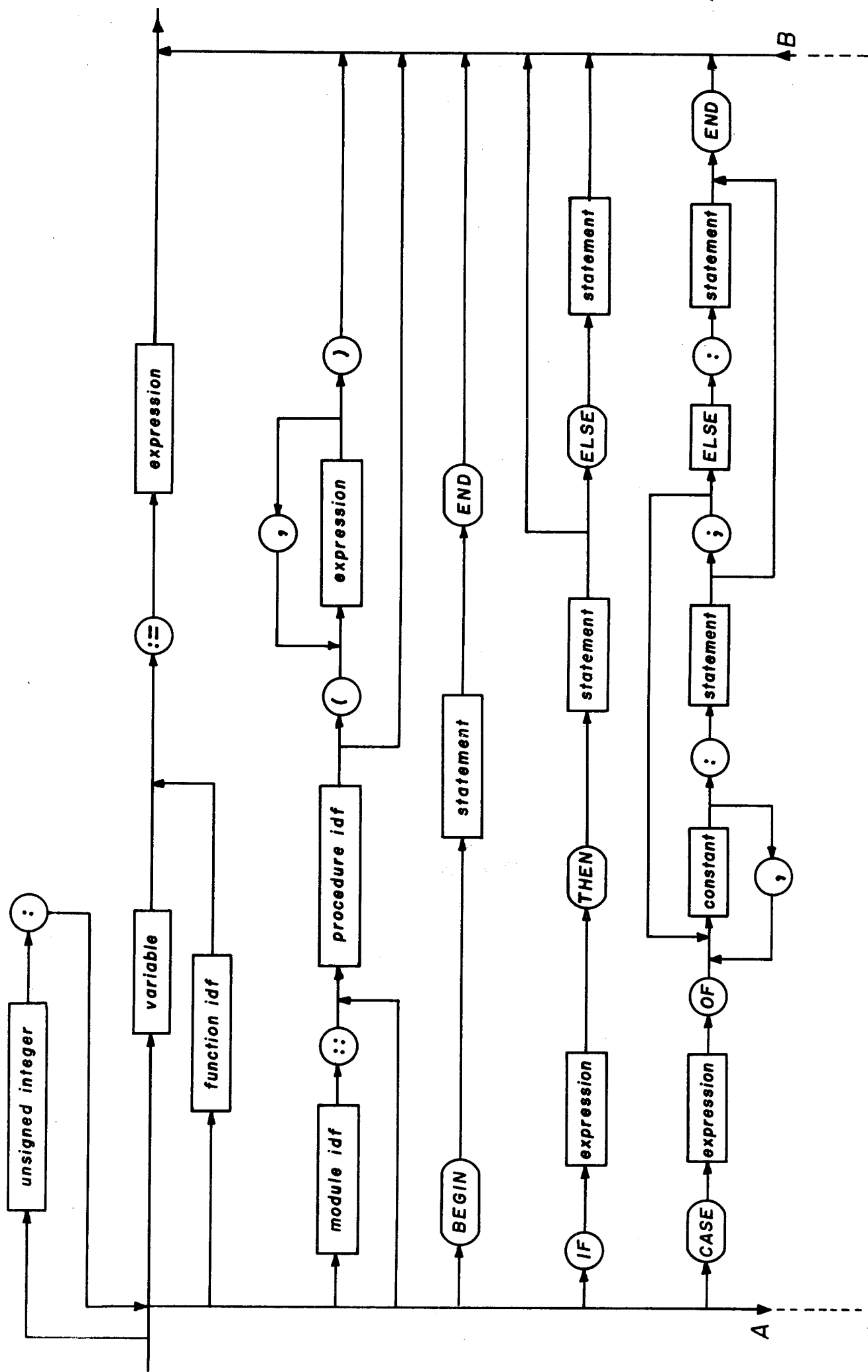
variable

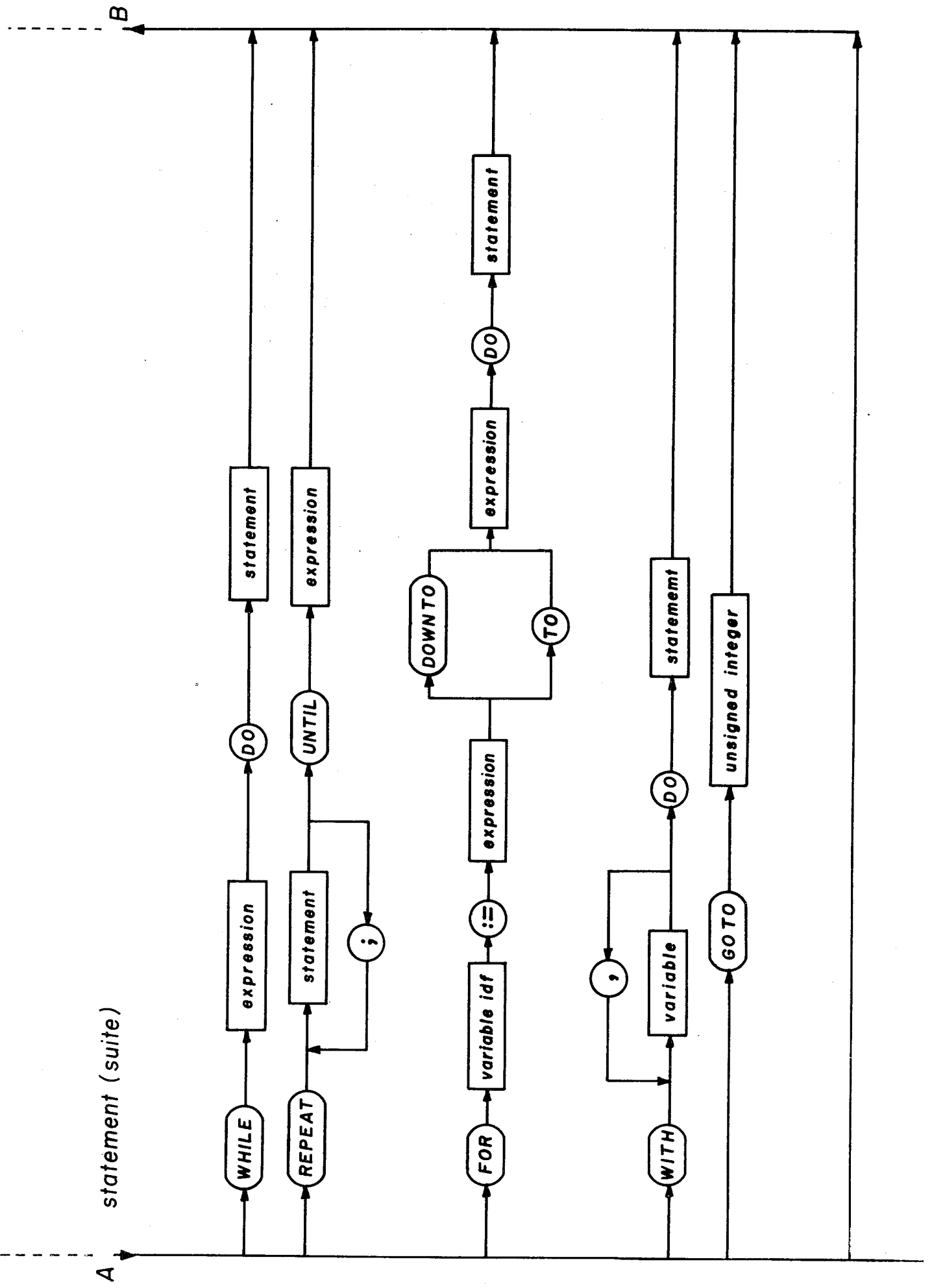




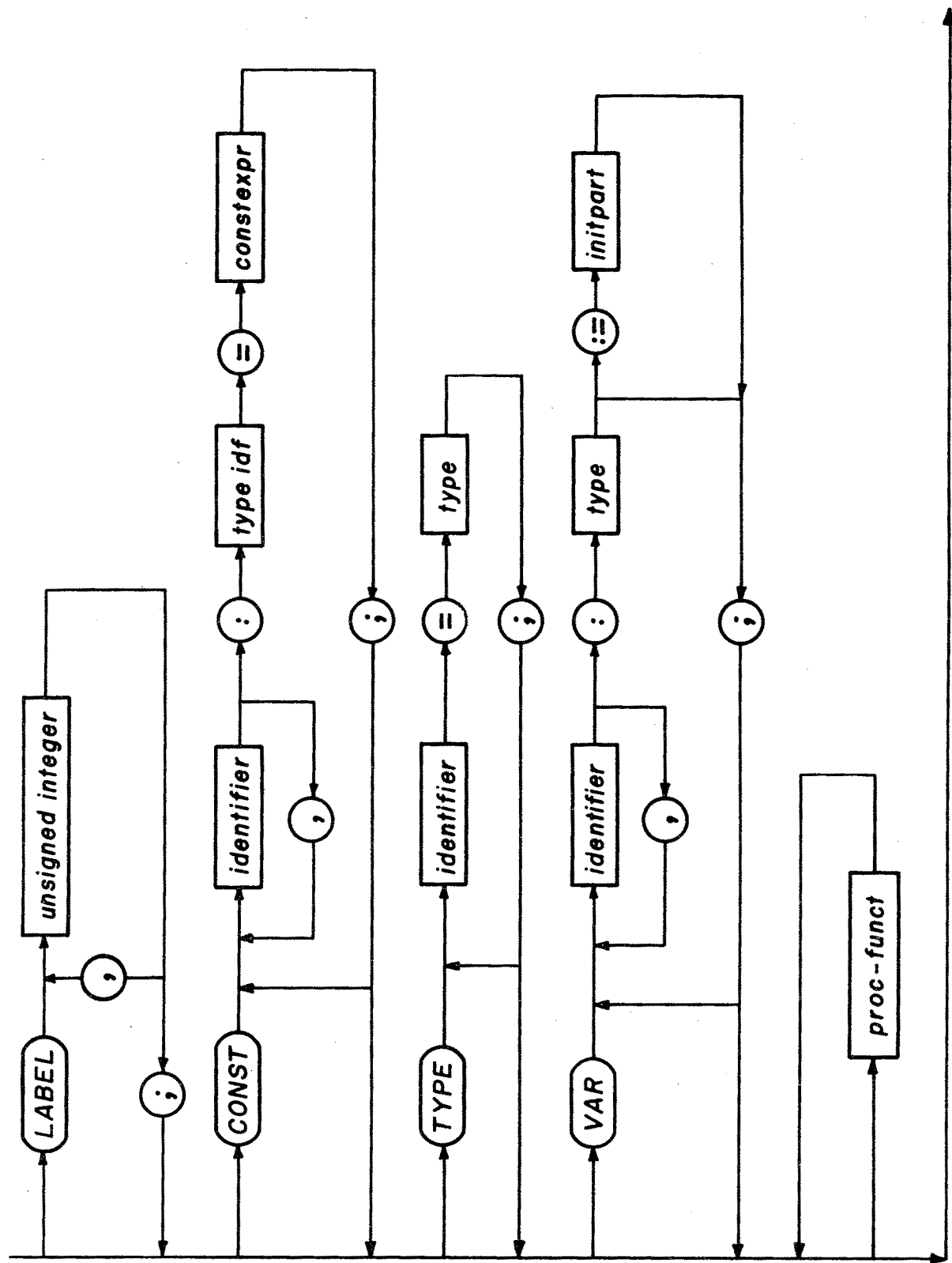


statement

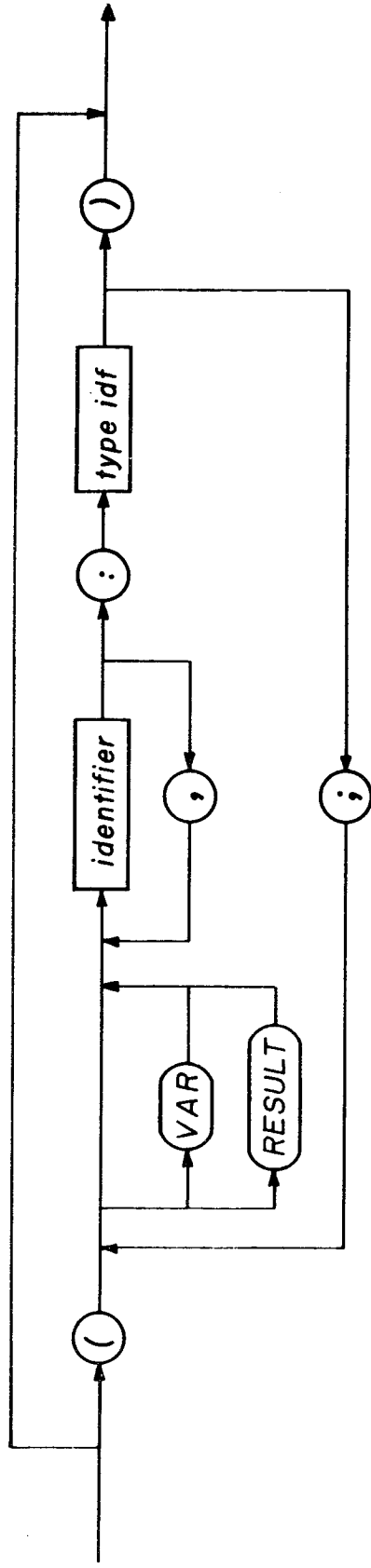




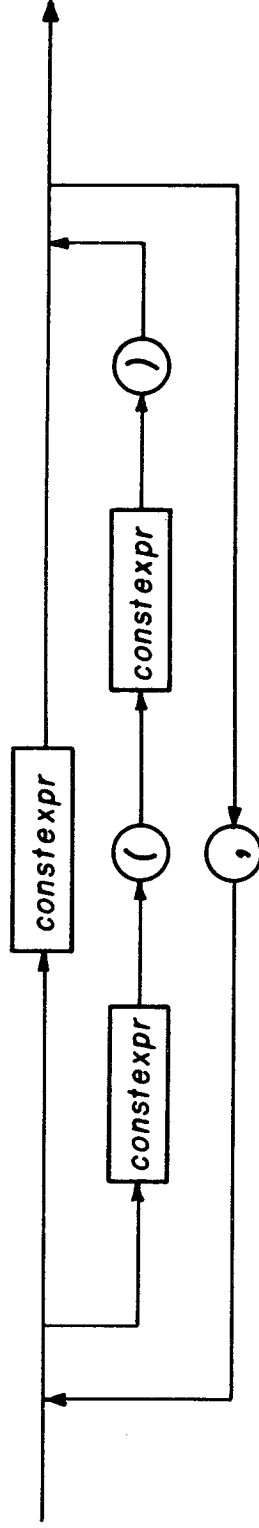
block



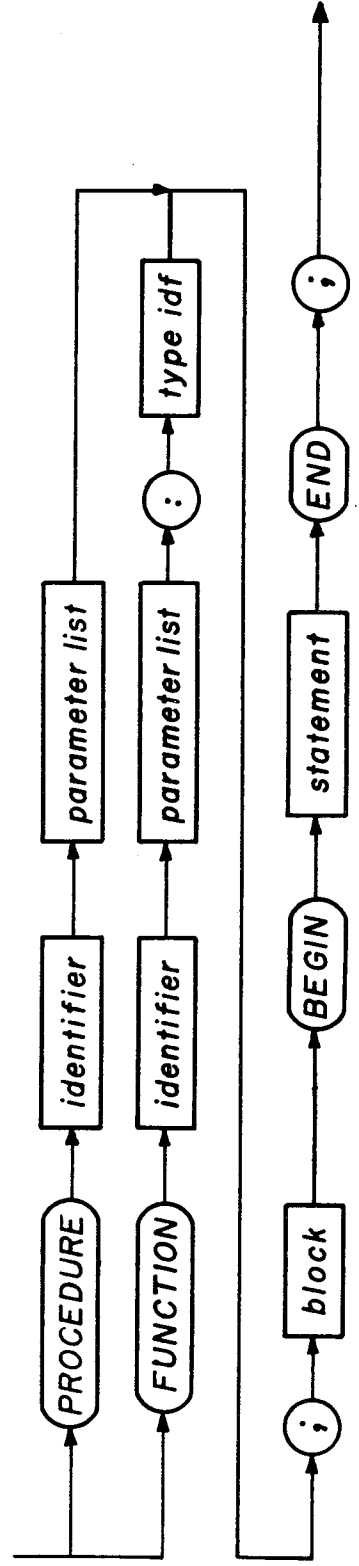
parameter list



init part



proc-funct



pattern

