LI5 : Rapport Technique du compilateur pascal

Nicolas LASSALLE Maxime GRYL

15/05/2004

Table des matières

Première partie

Interpréteur Pcode

1 Fonctionnement Global

Le fonctionnement de l'interpréteur se découpe en deux grandes phases :
- lecture et chargement en mémoire des instructions du fichier pcode
- exécution de toutes les instructions

1.1 Description des modules principaux

- **L'interface d'entrée / sortie**
  Elle permet la communication avec l'utilisateur ! Il fournit une interface de bas niveau qui écrit ou lit caractère par caractère.

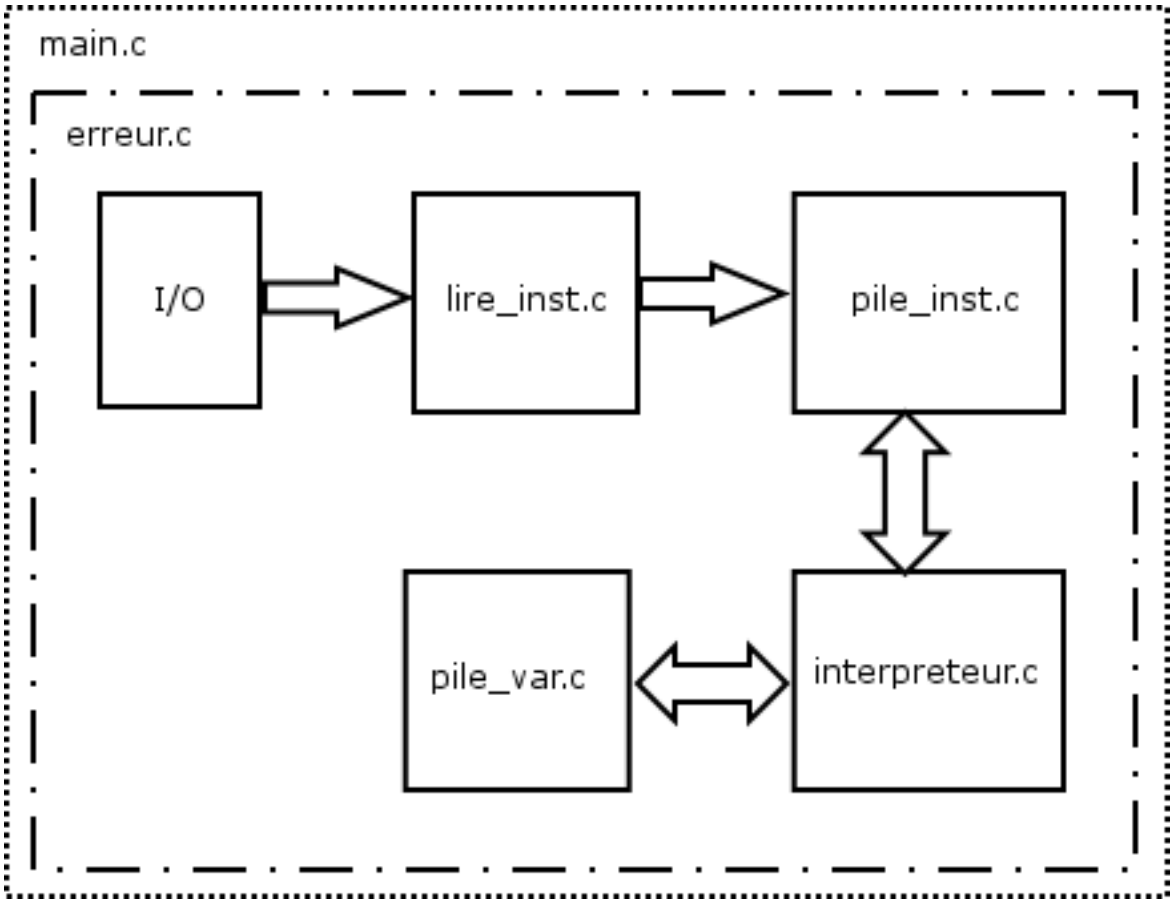
- **Le chargeur d'instruction**
  Ce module se sert de l'interface d'entrée / sortie pour lire le fichier pcode et charger toutes les instructions en mémoire. Il affiche des messages d'erreurs si la syntaxe/grammaire du programme pcode est non valide. Même en cas d'erreurs il continu à lire tout le fichier afin d'indiquer toutes les erreurs contenues.

- **La pile d'instructions**
  C'est une pile qui contient toutes les instructions à exécuter. Elle est remplie par le chargeur d'instruction, et c'est l'interpréteur qui va accéder à toutes les valeurs qu'elle contient.

- **La pile d'exécution**
  C'est une pile qui sert à simuler une machine à entier.

- **L'interpréteur / débugeur**
  C'est le moteur central de l'interpréteur. Il va lire les instructions à exécuter, se charge de gérer la pile d'exécution.

1.2 Schéma des modules



2 Le module d'entrées sorties

Le module d'entrées sorties s'occupe de l'ouverture et de la fermeture des fichiers et du traitement de son flux de données. Il bufférise le fichier pour diminuer les accès au disque et donc accélérer le programme. Les données du fichier sont envoyées caractère par caractère.

2.1 Interface d'utilisation

L'utilisation de ce module se fait par le biais de ces fonctions :
- Les fonctions **ouvre_fichier_lecture (char*)** et **ferme_fichier (Fichier)** qui permettent respectivement l'ouverture en lecture ainsi que le remplissage du buffer et la fermeture du fichier.
- La fonction **remplit_buffer (Fichier)** qui remplit le buffer. Elle est apellée quand tout le buffer est lu.
- La fonction **lire_car (Fichier *)** qui renvoi le caractère suivant.
- La fonction **insere_car (Fichier , char)** qui insère un caractère dans le flux de caractères.

2.2 Fonctionnement interne du module

La structure Fichier se compose d'un entier qui est le *File Descriptor* du fichier réel *fd*, d'un buffer de caractères *buffer[]*, et de 2 autres entiers qui indiquent la position actuelle dans le buffer et la position actuelle dans le fichier *pos_buffer* et *pos_fichier*. Un dernier caractère est rajouté *insere*, il sert pour l'insertion d'un entier dans le flux.
Deux autres entiers sont présent dans la structure : *unsigned int ligne ;* et *unsigned int colone*, elles sont utilisées durant la lecture dans le fichier (buffer) afin de connaitre la ligne et la colonne correspondante lors de la rencontre d'une erreur.

L'ouverture d'un fichier *toto* par ouvre_fichier_lecture("toto") retourne un Fichier avec le buffer de caractères remplit. On lira alors le fichier par l'appel à lire_car(Fichier), qui va renvoyer une à une toutes les lettres du buffer jusqu'à la fin de celui ci où elle le remplira de nouveau. Si le caractère *insere* est présent, alors il est prioritaire sur le buffer, puisqu'il doit etre inséré au flux. La fin du fichier est signalée par l'envoi de FIN_FICHIER.

3 Le jeu d'instruction pcode

3.1 Opérateurs

- **ADD** : Additionne le sous-sommet de pile et le sommet, laisse le résultat au sommet (idem pour SUB, MUL, DIV).
- **EQL** : Laisse 1 au sommet de pile si sous-sommet = sommet, 0 sinon (idem pour NEQ, GTR, LSS, GEQ, LEQ).

3.2 Ecriture

- **PRN** : Imprime la valeur entière au sommet, dépile.
- **PRD** : Imprime le caractère qui est au sommet de pile, dépile.
- **PRB** : Imprime 'vrai' si le sommet et != 0, 'faux' sinon, dépile.
- **PRI** : Imprime le sommet, ne dépile pas (utile pour débuguage manuel).
- **PRC 'c'** : Imprime le caractère passé en paramètre de l'instruction.
- **PRS "chaine"** : Imprime la chaîne de caractères passée en paramètre de l'instruction.
- **PRL** : Imprime le caractère "retour à la ligne".

Remarque : L'instruction *PRS* n'est pas implémentée en tant que fonction à part entière : elle est traduite à la lecture du fichier Pcode par une suite d'instruction **PRC 'c'** pour chaque caractère de la chaîne.

3.3 Lecture

- **INN** : Lit un entier, le stocke à l'adresse trouvée au sommet de pile, dépile.
- **INC** : Lit un caractère, le stocke à l'adresse trouvée au sommet de pile, dépile.
- **INB** : Lit un booléen, le stocke à l'adresse trouvée au sommet de pile, dépile. Booléen : **'1'** pour VRAI, **'0'** pour FAUX.

Remarque : Les instructions de lectures sont munies de tests ne permettant pas l'enregistrement de valeurs saisies ne correspondant pas au type souhaité.
Le dépassement de capacité n'est pas géré. (exemple : l'entier 1111111111111 saisi ne correspondra pas à la valeur enregistrée).

3.4 Insts de base

- **INT c** : Incrémente de la constante c le pointeur de pile (la constante c peut être négative).
- **LDI v** : Empile la valeur v.
- **LDA a** : Empile l'adresse a.
- **LDV** : Remplace le sommet par la valeur trouvée à l'adresse indiquée par le sommet (déréférence)
- **STO** : Stocke la valeur au sommet à l'adresse indiquée par le sous-sommet, dépile 2 fois.

3.5 Sauts

- **BRN i** : Branchement inconditionnel à l'instruction i. Adressage absolu.
- **BRR** : Branchement inconditionnel à l'instruction i. Adressage relatif.

- **BZE i** : Branchement à l'instruction i si le sommet = 0, dépile. Adressage absolu.
- **BZR i** : Branchement à l'instruction i si le sommet = 0, dépile. Adressage relatif.

Attention : La gestion des sauts imcomplets dans le pcode .

3.6 Procédures / fonctions

- **CAL i** : Empile le compteur d'instructions et réalise le branchement à l'instruction i.
- **RET i** : Néttoie la pile des emplacements pour les paramètres des procédures.

3.7 Autres

- **CPA / CPI** : Copie la valeur de l'adresse/entier en sommet de pile sur le sur-sommet.
- **CPJ** : Copie la valeur de l'entier en sommet de pile sur le sur-sommet.
- **DEL** : Supprime l'entier du dessus de la pile (sert rarement (cf cas())).
- **HLT** : Termine le programme

4 Lecture des instructions

Ce module permet de lire les instructions contenues dans le fichier pcode. Les commentaires contenus dans le code source pcode sont supprimés. Les instructions sont vérifées, ainsi que les paramètres de ces instructions. Chacune de ses intructions sont ajoutées dans la pile d'éxécution. C'est donc ce module qui remplit la pile d'éxécution. Dès qu'une erreur est rencontrée, le remplissage de la pile s'arrête, mais la lecture des insctructions continue normalement. Celà permet d'indiquer toutes les erreurs contenues dans le code source pcode.

4.1 Interface d'utilisation

L'interface d'utilisation est simplifiée au maximum. Pour lancer la lecture des instructions, il faut apeller la fonction **int remplir_pile (Fichier fic, Pile_exe *p_exe)**. Attention, le fichier passé doit être ouvert en écriture.

4.2 Fonctionnement interne du module

C'est la fonction remplir_pile(...) qui effectue une grande partie du traitement.

Voici les étapes suivies :
- Suppression des caractères vides en début de ligne
- Lecture des trois caractères composant un mnémonique
- Appel à la fonction code_correct(...) qui retourne plusieurs cas possibles :
  1. MNEMO_OK : le mnémonique est correct et ne prend pas de paramètre
  2. MNEMO_PARAM : le mnémonique est correct et prend un paramètre de type entier
     On lit alors le paramètre, qui peut être positif ou négatif et qui est constitué d'une suite de chiffres.
  3. MNEMO_CAR : le mnémonique est correct et prend un paramètre de type caractère
     On lit alors le paramètre. Le caractère lu doit être encadré par des quotes ' afin que l'interpréteur puisse pouvoir afficher des caractères blancs.
  4. MNEMO_STRING : le mnémonique est correct et prend un paramètre de type string
     On lit alors le paramètres. Le paramètre est une suite de caractères encadrés par des guillemets ".
  5. MNEMO_ERREUR : le mnémonique est incorrect

  Pour les mnémoniques prenant un paramètre, on vérifie l'existance d'un caractère blanc entre le mnémonique et son paramètre.
- Si le caractère indiquant un commentaire # est rencontré, on apelle la fonction suppr_commentaire(...) qui supprime tous les caractères jusqu'au saut de ligne suivant.
- On ajoute alors le mnémonique et son paramètre dans la pile des instructions.

Si au cours de la lecture des instructions on rencontre le symbole **$j**, on apelle la fonction remplir_sauts(...) qui se charge de compléter les sauts incomplets laissés par notre compilateur pascal (ou bien par un programmeur pcode complètement fou :)).

5 La pile des instructions

Elle contient toutes les instructions pcode du programme. Ces instructions sont fournies par le chargeur d'instructions et seront utilisées par l'interpréteur. Certaines instructions étant des sauts, il est nécesaire d'avoir tout le programme en mémoire ainsi qu'un pointeur sur l'instruction en cours qui soit modifiable.

5.1 Interface d'utilisation

Création/Destruction

- *Pile_exe creer_pile_exe()* renvoie une pile vide avec un premier buffer (vide) alloué.
- *void pile_exe_ajust (Pile_exe*)* permet d'avoir une pile donc la taille correspond exactement au nombre d'instructions total (l'augmentation se faisait par TAILLE_BUFFER et pas incrémentation simple), et ainsi mettre à jour la variable qui contient le sommet de la pile.
- *void pile_exe_free (Pile_exe*)* Vide la pile ; libére l'espace mémoire.

Accès Il n'y a que 3 sortes d'accès pour les valeurs dans la pile :

- *void push_exe (Pile_exe*, Instruction)* qui permet l'ajout en sommet de pile d'une instruction.
- *mnemoniques get_mn_exe (Pile_exe*)* qui retourne l'instruction courante.
- *int get_param_exe (Pile_exe*)* qui retourne le paramètre de l'intruction courante.

Déplacement Le pointeur d'instruction courante a besoin de se déplacer dans la pile, pour les boucles par exemple.

- *void goto_instruction_suivante (Pile_exe*)* pour passer à l'instruction suivante.
- *goto_abs_exe (Pile_exe*, int)* pour un saut relatif (positif ou négatif) à partir de l'instruction courante.
- *goto_relatif_exe (Pile_exe*, int)* pour un saut direct à un numéro de ligne.

Autre Les sauts ne pouvant toujours être complets, les paramètres de ces sauts sont stoqués en fin de fichier pcode avec le numéro de la ligne où se trouve le saut associé (voir section correspondante). Pour mettre à jour les sauts avec leurs paramètres il faut apeller la fontion *void pile_exe_maj_saut (Pile_exe *, int , int)*

5.2 Fonctionnement interne du module

Structure de la pile La Pile_exe possède 3 éléments :

- **Une instruction**
- **Un pointeur sur l'instruction courante** entier long
- **La longueur de la pile** entier long

La pile est doit etre créée, remplie puis ajustée. Ensuite il faut mettre à jour les sauts et enfin on peut commencer à lire les instructions (il n'y a plus d'ajout a faire). Selon les instructions la progression dans cette pile se fera croissante (incrémentation du pointeur d'instruction courante), ou par saut ; relatif (ajout ou soustraction avec le pointeur) ou en valeur absolue (remplacement du pointeur). L'interpreteur pourra, pour chaque instruction, accéder à sa mnémonique, et/ou son paramètre (s'il existe). Des tests sont effectués pour les sauts qui dépasseraient la pile (supérieurs ou inférieurs), et quittent l'exécution de l'interpréteur par des erreurs de type *SEGMENTATION FAULT* ou *STACK OVERFLOW*.

6 La pile d'exécution (FIFO)

La pile d'excution sert à stoquer les valeurs des variables et sera utilisée pour les calculs, mise à jour des variables...

La première partie de la pile (bas) va contenir les valeurs des variables globales, les élements du reste de la pile pourront être des adresses, des variables locales de fonctions/procédures, des valeurs entières ou des caractères. Tout ces élements sont stoqués sur cette pile sous forme d'entiers (ex : les char sont en code ASCII). Pour l'utilisation de nombres à virgule il serait intérressant de ne pas changer la pile d'entier en pile de double (ou float), mais d'utiliser 2 entier pour 1 réel (ex : représentation sous forme de struture ; 1 entier pour le chiffre avant la virgule, 1 entier pour le reste).

6.1 Interface d'utilisation

- *Pile_var creer_pile_var ()* Créer la pile à vide avec allocation d'un premier buffer.
- *pile_var_free (Pile_var*)* Libère l'espace mémoire occupé par la pile.

- *push_var (Pile_var*, int)* Ajoute une valeur en sommet de pile.
- *push_to_var(Pile_var*, int, int)* Ajoute une valeur à l'adresse passée en paramètre de pile.
- *push_sommet_var (Pile_var*, int)* REMPLACE la valeur au sommet. Utilisé pour les résultats (des comparaisons /opérations).

- *int pop_var (Pile_var*)* Retourne la valeur de sommet de pile, dépile.
- *int get_sommet_var (Pile_var)* Retourne la valeur de sommet de pile, NE dépile PAS.
- *int get_sous_sommet_var (Pile_var)* Retourne la valeur de SOUS-sommet de pile, NE dépile PAS.
- *int get_from_var(Pile_var, int)* Retourne la valeur placée à l'adresse passée en paramètre, NE dépile PAS.

- *inc_sp_var(Pile_var*, int)* Incrémente le pointeur de pile (allocation si nécessaire).

6.2 Fonctionnement interne du module

Ici on simule une pile de variables. La pile est représentée sous forme d'un tableau dynamique. Ce tableau est alloué par buffers afin d'optimiser la rapidité d'exécution du programme.

Toutes les opérations de base d'une pile telle que le pop, le push ont été écrites. Celà permet un accès plus rapide à des opérations répétitives lors de l'interprétation. Ces fonctions n'excluent pas l'accès direct aux champs de la structure. Cependant il faudra faire attention aux dépassements de tableaux.

Ces dépassements de tableaux étaientt gérés selon les accès à la section variables ou à la section de calcul, avec l'implémentation des fonctions, et donc la mise en place de variables locales en dehors de la section des variables, ces protections n'ont plus toutes un sens et se sont réduites.

Voila la structure de la pile :

```
typedef struct
{
      int *mem;
      int sp;
      int segment;
}Pile_var;
```

- **mem** contient toute les données de la pile.
- **sp** est le sommet de pile.
- **segment** correspond à la limite entre la zone des variable et la zone de calcul (utilité trés réduite avec l'implémentation des procédures/fonctions à variables locales).

Le fontionnement de la pile est assez simple : pile fifo normale.
Pour chaque ajout dans la pile on teste si la pile est pleine, dans ce cas on réalloue la taille d'un buffer.

7 L'interpréteur

Ici on simule l'éxécution du fichier de pcode. Pour celà on utilise deux piles : la pile des instructions et la pile des variables.

7.1 Interface d'utilisation

La fonction *execute* est la fonction principale du module. C'est une boucle qui avance dans la pile d'instruction jusqu'à HLT ou tant que la pile d'instruction n'est pas vide. Il va associer à chaque instruction Pcode une action correspondante (calcul sur la pile des variables, saut dans la pile d'instruction,...). Après chaque instruction effectuée on passe à la suivante. Cette procédure est très simple ; elle appelle selon l'instruction courante la procédure correspondante.

7.2 Fonctionnement interne du module

Les procédures associées aux instructions sont elles même assez basiques, elles font appel aux fonctions d'accès des piles des variables et instructions. Elles portent généralement le nom de leur mnémomiques, *cf sources*.

La fonction de débuguage *affiche_ debug (Pile_ var *p_ var, Pile_ exe *p_ exe)* se trouve également dans ce module. Elle affiche les informations suivantes en mode DEBUG (variable *debug*) :

- l'instruction courante
- l'instruction précédente
- l'instruction suivante
- le segment de données
- le sommet et le sous somment de la pile

8 La gestion d'erreurs

Notre module de gestion d'erreurs possède deux fonctionalités importantes. Il permet une gestion centralisée des erreurs pouvant survenir lors de l'exécution d'une fonction du programme (ex : erreur d'ouverture de fichier). Il permet également de gérer les erreurs pouvant survenir dans la grammaire/syntaxe/sémantique du programme que l'on compile.

On distingue dans ce module deux types d'erreurs :
- **les erreurs fatales** : quand une erreur de ce type est rencontrée il est impossible de continuer l'exécution du programme. Il faut alors libérer la mémoire allouée et retourner un code d'erreur unique.
- **les messages d'avertissement** : ce sont les erreurs ne nécessitant pas l'arrêt du programme.

Il y a deux avantages à utiliser cette méthode :
- les messages d'erreurs sont tous regroupés dans un même fichier. Il sera donc plus facile d'envisager une traduction de l'application.
- la gestion de la mémoire est optimale, ainsi tout bloc alloué sera systématiquement désaoullé.

8.1 Interface d'utilisation

L'utilisation de ce module se fait au travers de trois fonctions et d'un type énuméré contenant la liste des erreurs possibles :
- la fonction **free_all (Erreurs code_retour)** : elle déclenche une erreur fatale. Elle affiche donc le message d'erreur associé au code_erreur passé en paramètre, elle libére toute la mémoire allouée et retourne un code d'erreur unique.
- la fonction **erreur_mess (Erreurs code_retour)** : elle affiche un message d'erreur simple, l'erreur rencontrée n'étant pas génante pour la fin de l'exécution du programme.
- la fonction **usage (...)** : elle est apellée si les paramètres du programme sont incorrects. Elle affiche alors la liste des paramètres possibles du programme. Elle termine le programme, et ne libère pas la mémoire, car à ce stade aucune allocation mémoire n'a été effectuée.
- le type énuméré **Erreurs** qui contient la liste des erreurs que l'on peut rencontrer. Par convention, le nom des erreurs fatales commence par EXIT_ et celui des erreurs non fatales par ERR_ .

8.2 Fonctionnement interne du module

Le fonctionement interne n'est pas très compliqué. A chacune des entrées du type énuméré on associe un message d'erreur. Ces messages d'erreurs sont stockés dans un tableau de chaines de caractères : ce tableau est apellé ErreursMsg. Commes les valeurs d'un type énuméré sont numérotées de 0 à N-1, on accède à un message d'erreur de la façon suivante : ErreursMsg [ERR_... | EXIT_...] .

Il faut cependant faire attention à la fonction **free_all(...)** car c'est elle qui libère la mémoire allouée. Donc si on rajoute des allocations mémoires dans un module, il faut faire le free correspondant dans cette fonction afin d'avoir une gestion de la mémoire optimale.

Deuxième partie

Compilateur Pascal

9 Fonctionnement Global

9.1 Description des modules principaux

- **L'interface d'entrée / sortie**
  Elle permet la communication avec l'utilisateur ! Il fournit une interface de bas niveau qui écrit ou lit caractère par caractère.

- **L'analyseur lexical**
  Son rôle est de fournir un flot de token à partir du module d'entrée sortie. Il lit caractère par caractère jusqu'à trouver un caractère blanc (retour chariot, espace ou tabulation).

- **L'analyseur syntaxique**
  Alors que l'analyseur lexicale reconnait les mots du langage, l'analyseur syntaxique reconnait les phrases du langage. Il récupère donc les tokens fournis par l'analyseur lexical.

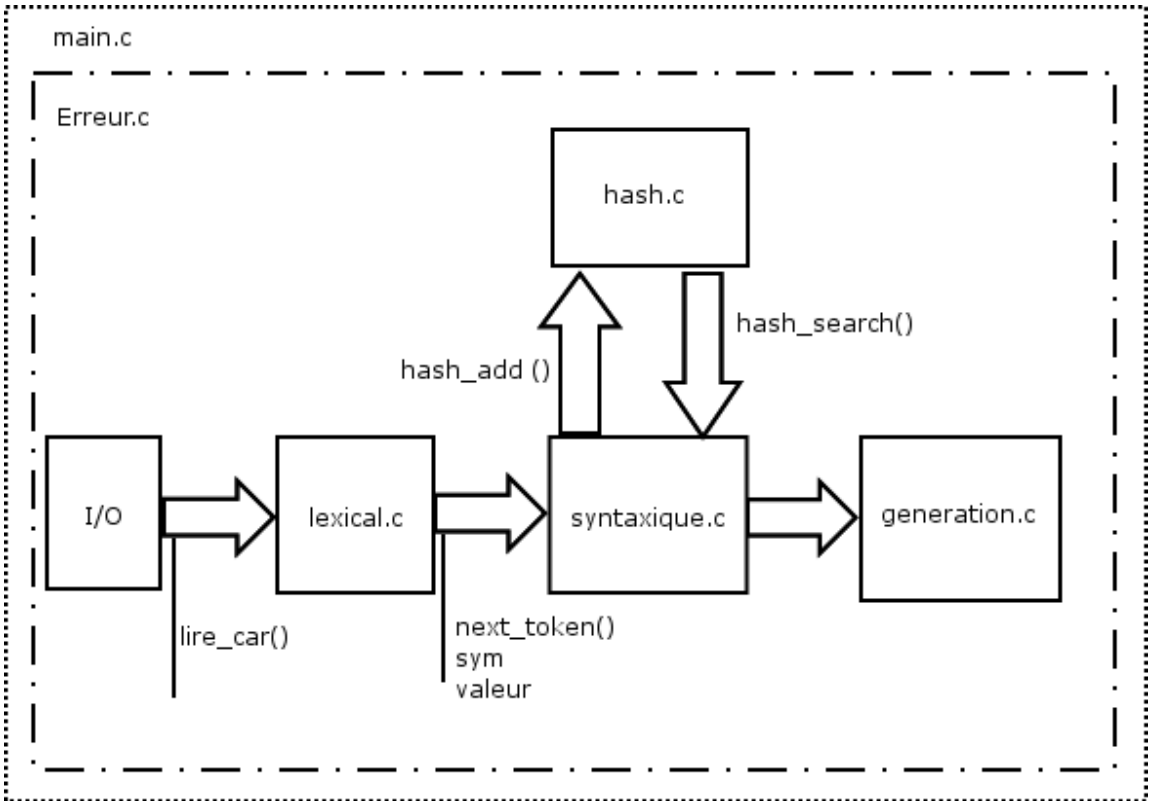
- **L'analyseur sémantique**
  Une fois que la syntaxe du programme que l'on compile est correcte, il faut vérifier certaines règles sémantiques telles que :
  - les identifiants utilisés sont bien déclarés ?
  - les opérandes ont ils le type attendu ?
  - les opérandes sont ils compatible ?
  - les paramètres de fontions sont ils correct ?

  Ainsi il sera nécessaire de construire une table des symboles.

- **Gestion des erreurs**
  En cas d'erreur dans la syntaxe du programme pascal, une erreur est déclenchée. Cela permet au programmeur de pouvoir corriger ses erreurs. Il corrige également le programme afin que le compilateur puisse continuer à analyser le programme pascal, et ainsi indiquer au programmeur plusieurs erreurs à la fois. A la première erreur, la génération de code est arrêtée. Des erreurs fatales peuvent aussi être déclenchées. Dans ce cas un message d'erreur est affiché, la mémoire est libérée et un code de retour unique est renvoyé.

- **Génération de code**
  Ce module est utilisé par le module syntaxique qui génère le code pcode correspondant au programme pascal que l'on compile. Le générateur de code se sert du module d'entrée / sortie afin d'écrire sur le disque le fichier pcode.

9.2 Schéma des modules


main.c
Erreur.c
hash.c
hash_search()
hash_add ()
I/O
lexical.c
syntaxique.c
generation.c
lire_car()
next_token()
sym
valeur

10 Le module d'entrées sorties

La partie ENTREES est exactement la même que celle de l'interpréteur. La partie SORTIE permet de créer le fichier Pcode pour être interprété. Elle comporte comme pour l'entrée un buffer, qui va stoquer plusieurs caractères avant de les écrire dans le fichier. Ce module de sortie est utilisé par le module de génération (de Pcode).

La Partie entrées : *cf interpréteur* La Partie sorties :

10.1 Interface d'utilisation

Pour facilité l'insertion des données dans le fichier (buffer avant), il y a 3 façons d'enregistrer :

- *void ecrire_car (char, Fichier *)* qui permet d'écrire un caractère seul.
- *void ecrire_string (char *, Fichier *)* qui permet d'écrire une chaîne de caractère d'un coup.
- *void ecrire_entier (int, Fichier *)* qui permet d'écrire les caractères composant un entier.

La fonction d'ouverture *Fichier ouvre_fichier_ecriture (char *)* permet un accès en écriture, et la fonction *void supprime_fichier (Fichier *)* permet de supprimer le fichier du disque. L'ouverture en écriture du fichier implique la destruction du fichier de même nom s'il existe déjà sur le disque. Il est possible d'implémenter un test qui demandera l'autorisation de l'utilisateur pour supprimer, et/ou peut-être une option d'exécution (ex : -f) qui forcera l'écrasement du fichier.

Le buffer sera écrit dans le fichier par la fonction *void ecrit_buffer (Fichier *)* s'il est plein, et par *void vide_buffer (Fichier *)* sinon.

10.2 Fonctionnement interne du module

La structure d'un fichier pour son écriture possède un seul élément qui la différencie de la structure du fichier en lecture de l'interpréteur : **char *nom_fichier ;** Et le seul élément qui différencie cette structure de celle de l'interpréteur, ce champ est nécessaire à la suppression du fichier en cas de compilation incomplète.

Quand le buffer est plein, alors les données sont écrites dans le fichier et le buffer remis à 0 (pos_buffer=0). Si le programme pascal est compilé sans erreur, alors le buffer non-remplit est vidé dans le fichier. Si le programme Pascal est erroné, alors il ne faut avoir de fichier Pcode incomplet qui soit généré : la fonction de sortie utilisera le nom du fichier pour le supprimer. Ces deux appels (*void vide_buffer (Fichier *)* et *void supprime_fichier (Fichier *)*) sont effectué par *freeAll*.

11 L'analyseur lexical

L'analyse lexicale est la première phase de la compilation. Dans le fichier source, l'analyse lexicale reconnait des tokens, qui sont les mots avec lesquels les phrases sont construites. Ces mots sont envoyés à l'analyseur syntaxique. On distingue les unités lexicales suivantes :

- les caractères spéciaux simples : *=, +, -, ...*
- les caractères spéciaux doubles : *<=, <>, >=, ...*
- les mots clés : *begin, end, while, ...*
- les litteraux : *+25, 3, 78, ...*
- les identificateurs : *i, toto1, toto2, ...*

L'analyseur lexical respecte les méta règles suivantes :

- Un *commentaire* est une suite de caractères encadrés par des parenthèses (* et *) et aussi de { et } .
- Un *séparateur* est un *caractère séparateur* (espace, tabulation, retour chariot) ou un commentaire.
- Deux *identificateur* ou *mots clés* qui se suivent doivent être séparés par au moins un séparateur.
- Des séparateurs peuvent être insérés partout.

Outre la reconnaissance des tokens du langage, l'analyseur lexical se charge donc de supprimer les séparateurs (tabulations, espaces, retour à la ligne) et les commentaires.

11.1 Tokens reconnus

- Opérateurs :
  PLUS_TOKEN (+), MOINS_TOKEN (-), MUL_TOKEN (*), DIV_TOKEN (/), EGAL_TOKEN (=), DIFF_TOKEN (<>), INF_TOKEN (<), SUP_TOKEN (>), INF_EGAL_TOKEN (<=), SUP_EGAL_TOKEN (>=)

- Délimiteurs :
  PAR_OUV_TOKEN (), PAR_FER_TOKEN ()), VIRG_TOKEN (,), PT_VIRG_TOKEN (;), POINT_TOKEN (.), PT_PT_TOKEN (..), DEUX_PT_TOKEN (:), QUOTE_TOKEN ('), CRO_OUV_TOKEN ([), CRO_FER_TOKEN (])

- Mots clés du langage pascal :
  AFFEC_TOKEN (:=), BEGIN_TOKEN (begin), END_TOKEN (end), IF_TOKEN (if), WHILE_TOKEN (while), THEN_TOKEN (then), DO_TOKEN (do), WRITE_TOKEN (write), WRITELN_TOKEN (writeln), READ_TOKEN (read), CONST_TOKEN (const), VAR_TOKEN (var), PROGRAM_TOKEN (program), FOR_TOKEN (for), TO_TOKEN (to), DOWNTO_TOKEN (downto), REPEAT_TOKEN (repeat), UNTIL_TOKEN (until), STEP_TOKEN (step), CASE_TOKEN (case), OF_TOKEN (of), ELSE_TOKEN (else), OTHERWISE_TOKEN (otherwise), TYPE_TOKEN (type), ARRAY_TOKEN (array), RECORD_TOKEN (record), TRUE_TOKEN (true), FALSE_TOKEN (else)

- Variables :
  ID_TOKEN (*suite de lettres ou de chiffres commençant par une lettre ou par '_'*), NUM_TOKEN (*constantes numériques*)

11.2 Interface d'utilisation

Le module lexical fournit l'accès aux types et variables suivantes :
- Le type énuméré **Tokens** qui contient la liste de tous les tokens reconnaissables.
- La variable **token** de type Tokens qui contient le dernier token reconnu
- La variable **sym** de type char* qui contient la forme textuelle du dernier token lu
- La variable **valeur** de type int qui contient la valeur du dernier token lu

Ces 3 variables sont mises à jour par l'appel à la fonction **next_token()**.

Donc l'utilisation de ce module est très simple : il suffit d'apeller la fonction **next_token()** afin d'obtenir le token correspondant au mot reconnu dans le fichier.

11.3 Fonctionnement interne du module

11.3.1 Lecture des caractères dans le fichier

Elle est réalisée grâce à la fonction lire_car_min() qui est un niveau au dessus de lire_car() du module d'entrée sorties. En effet, cette fonction ne retourne que des caractères en minuscules.

11.3.2 Suppression des séparateurs

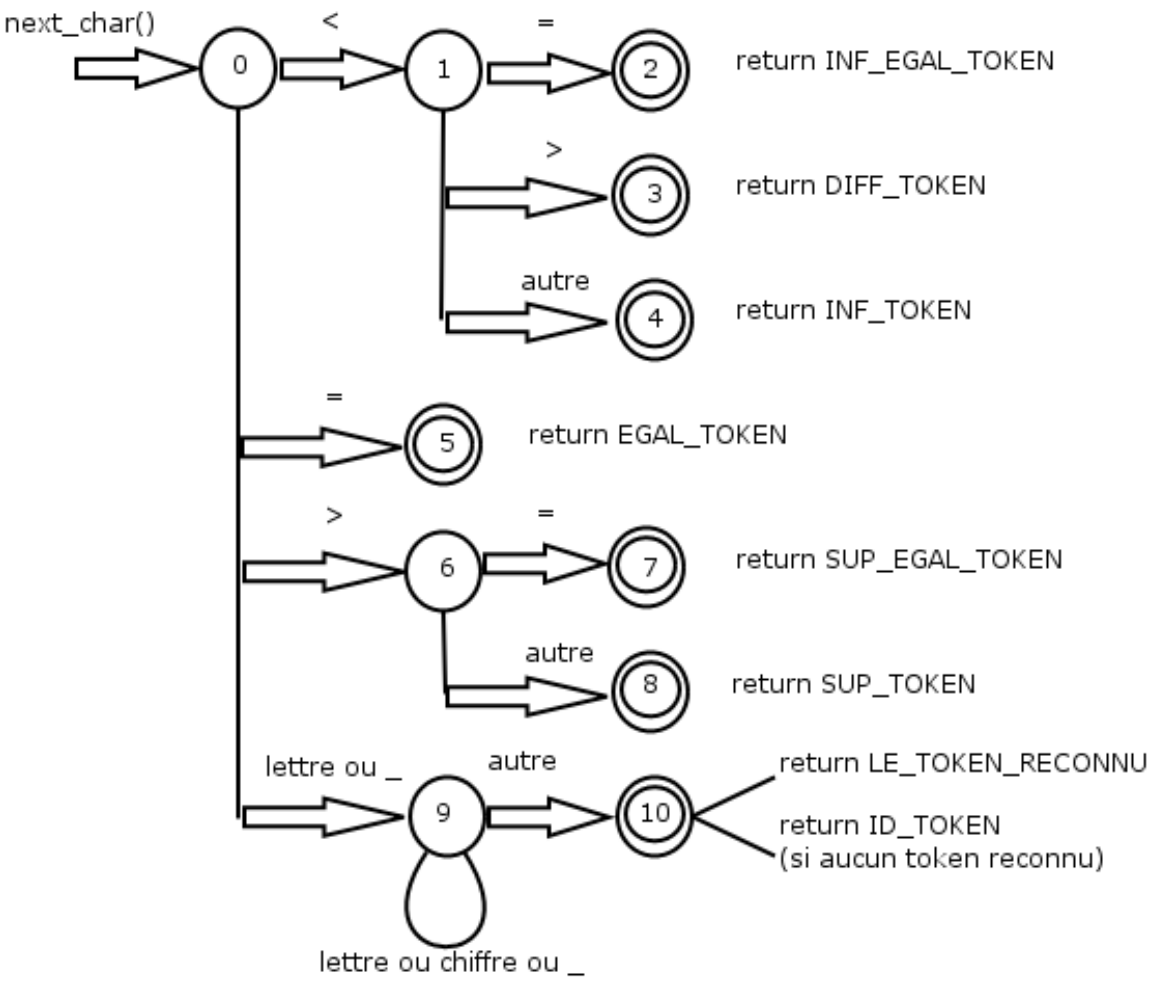
Elle est réalisée par la fonction next_char(). Elle supprime n'importe quelle séquence de caractères séparateurs. Elle impose de rencontrer au moins un caractère séparateur afin d'imposer la séparation des tokens. Cette fonction retourne 1 si au moins un caractère séparateur est rencontré, 0 sinon.

11.3.3 Reconnaissance des mot clés et des caractères spéciaux du langage

C'est la partie la plus complexe de l'analyseur lexicale. Il a fallut trouver une méthode de reconnaissance rapide des tokens et des caratères, car l'utilisation de la fonction strcmp est très couteuse.

La reconnaissance se déroule comme suit : on apelle la fonction next_char() puis on lit un premier caractère. On a alors plusieurs possibilités :
- Le caractère lu est un caractère simple (ex : '+'). On renvoi le token correspondant (PLUS_TOKEN).
- Le caractère lu est est le début d'un caractère double (ex : '<'), on essaie alors de reconnaitre un caractère double (ex : '<='). Si un caractère double est rencontré, on renvoi le token correspondant (INF_EGAL_TOKEN), sinon on renvoi le token correspondant au caractère simple (INF_TOKEN)..
- Si le caractère simple rencontré est un chiffre, on lit le chiffre correspondant. Le token renvoyé est alors NUM_TOKEN. La variable valeur est également mise à jour.
- Si le caractère simple rencontré est une lettre, on essaie de reconnaitre un des mots du langage. Si un mot est reconnu on renvoi le token correspondant. Autrement, nous venons de rencontrer un identificateur. On renvoi alors ID_TOKEN et la variable sym est mise à jour.

La reconnaissance suit donc l'automate suivant :



11.3.4 Reconnaissance des tokens : explications détaillées

La méthode de reconnaissance utilisée est similaire à celle employée pour la reconnaissance des mnémoniques dans l'interpréteur pcode. On code les lettres dans un entier en effectuant des décalages. Chaque lettre peut être codée sur 8bits. En décallant ces 8 bits de 8bits vers la gauche on peut stocker 4 lettres dans un entier qui fait 32bits.

Le premier problème de cette méthode est que les tokens reconnaissables ne font pas tous 4 lettres ou moins. Ce problème a été résolu en stockant les tokens en plusieurs fois dans plusieurs entiers. Cependant il y a une exception pour les tokens dont la taille modulo 4 est égale à un : dans ce cas il n'y a plus qu'une seule lettre a reconnaitre et la coder sur un entier est plus couteux que de faire directement un test d'égalité avec la lettre attendue pour finir le token. Chaque token est donc rajouté au cas pas cas, ce qui est le deuxième inconvénient de cette méthode, car le programme n'est pas très facilement maintenable.

Plus concrêtement voici quelques exemples :

1. Reconnaissance des tokens de taille $\leq$. Exemle : ELSE_TOKEN
   (a) Appel de la fonction reco_identifiant(...) qui code les 4 premières lettres maximum sur un entier.

   (b) La fonction teste le code obtenu avec les codes existants correspondant aux tokens reconnaissables.

   (c) Le code de 'else' est le même que celui de MN_ELSE. On renvoit donc ELSE_TOKEN.

   (d) Le code ne correspont au début d'aucun token, donc on apelle la procédure fin_identifiant(...) qui se charge de finir de lire l'identifiant et qui renvoit ID_TOKEN.

2. Reconnaissance des tokens de taille 5. Exemle : BEGIN_TOKEN

   (a) Appel de la fonction reco_identifiant(...) qui code les 4 premières lettres sur un entier.

   (b) La fonction teste le code obtenu avec les codes existants correspondant aux tokens reconnaissables.

   (c) Le code de 'begi' est le même que celui de MN_D_BEGIN. On apelle la procédure fin_token_unique (...) avec la lettre 'n' (la fin de begin !) et qui vérifie que le caractère suivant est bien 'n' et qu'il y a un blanc ensuite. Cette procédure renvoit soit BEGIN_TOKEN si le test est OK, soit elle passe à l'étape (d).

   (d) Le code ne correspont au début d'aucun token, donc on apelle la procédure fin_identifiant(...) qui se charge de finir de lire l'identifiant et qui renvoit ID_TOKEN.

3. Reconnaissance des tokens de taille comprise entre 5 et 8. Exemle : RECORD_TOKEN

   (a) Appel de la fonction reco_identifiant(...) qui code les 4 premières lettres sur un entier.

   (b) La fonction teste le code obtenu avec les codes existants correspondant aux tokens reconnaissables.

   (c) Le code de 'reco' est le même que celui de MN_D_RECORD. On apelle la procédure fin_token (...) avec le paramètre MN_D_RECORD (le code fin de begin !) et qui vérifie que le caractère suivant est bien 'n' et qu'il y a un blanc ensuite. Cette procédure renvoit soit RECORD_TOKEN si le test est OK, soit elle passe à l'étape (d).

   (d) Le code ne correspont au début d'aucun token, donc on apelle la procédure fin_identifiant(...) qui se charge de finir de lire l'identifiant et qui renvoit ID_TOKEN.

12 L'analyseur syntaxique

La syntaxe d'un langage est décrite par une grammaire constituée par un sensemble de règles de productions. Chaque règle indique en partie gauche un non terminal qui peut être dérivé en une des formes décrites en partie droite séparées par des " | ". Une forme est composées de terminaux notés en minuscule et de non terminaux en majuscule. Une forme entre accolade signifie qu'elle peut être présente autant de fois que possible (dont zéro). Ainsi le but de l'analyse syntaxique est de montrer que l'axiome de la grammaire , ici PROGRAM, peut être dérivé en une suite de terminaux formant le texte à analyser.

12.1 Grammaire reconnue

```
PROGRAM     ::= program ID ; BLOCK .
BLOCK       ::= {{CONSTS} {TYPES} {VARS}} {{PROCS} {FUNCS}} INSTS
PROCS       ::= procedure ID PARAMLIST ; CONSTS VARS INSTS;
FUNCS       ::= function ID PARAMLIST : TYPE_RETOUR; CONSTS VARS INSTS;
PARAMLIST   ::= (PARAMDECL {,PARAMDECL}) | ()
PARAMDECL   ::= var ID : TYPE | ID : TYPE
TYPE_RETOUR ::= integer | char | boolean
APPEL       ::= ID PARAMEFFLIST
PARAMEFFLIST::= (PARAMEFF {,PARAMEFF} ) | ()
PARAMEFF    ::= true | false | 'un char' | ID | EXPR
CONSTS      ::= const ID = NUM ; { ID = NUM ; } | E
VARS        ::= var DECL { ; DECL } ; | $E
DECL        ::= ID {, ID} : TYPE
TYPE        ::= ID | TYPE_ARRAY | TYPE_RECORD
TYPES       ::= type TYPEDECL ; {TYPEDECL ;} | E
TYPEDECL    ::= ID = TYPE_ARRAY | ID = TYPE_RECORD
TYPE_ARRAY  ::= array[DIM {, DIM}]
TYPE_RECORD ::= record CHAMPS {; CHAMPS} end
CHAMPS      ::= ID {, ID} : TYPE
DIM         ::= EXPR | EXPR .. EXPR
INSTS       ::= begin INST { ; INST } {end}
INST        ::= INSTS | AFFEC | SI | TANTQUE | ECRIRE | LIRE | FOR | REPEAT
                | APPEL | E
AFFEC       ::= VAR := EXPR
VAR         ::= ID | VAR . ID | VAR [EXPR {,EXPR}]
SI          ::= if COND {then} INST
TANTQUE     ::= while COND do INST
ECRIRE      ::= write ( MES { , MES } )
MES         ::= '~tous les carac~' | EXPR | VAR
LIRE        ::= read ( VAR { , VAR } )
COND        ::= EXPR RELOP EXPR | CHAREXPR RELOP CHAREXPR
                | BOOLEXPR BOOLOP BOOLEXPR
CHAREXPR    ::= 'un char' | VAR
BOOLEXPR    ::= true | false | VAR
BOOLOP      ::= = | <>
RELOP       ::= = | <> | < | > | <= | >=
EXPR        ::= TERM { ADDOP TERM }
```

```
ADDOP       ::= + | -
TERM        ::= FACT { MULOP FACT }
MULOP       ::= * | /
FACT        ::= VAR | NUM | APPEL | ( EXPR )
FOR         ::= for VAR := EXPR TO EXPR PAS do INSTS
PAS         ::= E | step EXPR
TO          ::= to | downto
REPEAT      ::= repeat INST {, INST} until COND
CAS         ::= {case} VAR {of} LABEL
LABEL       ::= NUMS | NUMS .. NUMS | NUMS {, NUMS}
ELSECASE    :: else | otherwise | E
NUMS        ::= ID | EXPR
```

12.2 Interface d'utilisation

Comme l'analyse syntaxique est un automate, il suffit d'apeller la fonction correspondant à l'axiome de la grammaire. Si aucune erreur est rencontré, le programme est syntaxiquement correct.
La fonction à apeller est **program()** ;

12.3 Fonctionnement interne du module

12.3.1 La fonction teste

Cette fonction "teste" teste si le prochain token (variable token) est bien celui passé en paramètre à la fonction ; on s'arrête sur une erreur sinon (procédure erreur définie dans le module de gestion d'erreurs) :

12.3.2 Codage de l'automate

Comme l'automate a de bonne propriétés, il est possible d'écrire un analyseur récursif descendant simple (sans retour arrière). Li'dée est que chaque règle de grammaire est associée à une procédure qui vérifie la concordance du texte à analyser avec une de ses parties droites.

13 L'analyseur sémantique

Après l'analyse lexicale et syntaxique, l'étape suivante est l'analyse sémantique. Voici des exemples de vérifications sémantiques :

- construire et mémoriser des représentations de types définis par l'utilisateur.
- traiter les déclarations de variables, de constantes, de fonctions et de mémoriser les types associés.
- vérifier que toutes les variables/constantes utilisées, fonctions apellées, types utilisés ont bien été préalablement déclarés.

Pour stocker toutes ces informations, on cré une table des symboles. Cette table des symboles est représentée en mémoire sous la forme d'une table de hashage. Les collisions sont gérées par sauts et par des arbres binaires de recherche.

13.1 Représentation des informations dans la table des symboles

13.1.1 La table de hashage

La table de hashage possède une taille fixe définit par la constante MAX_TABLE. On est obligé de fixer la taille de la table car une taille dynamique imposerait de devoir hasher à nouveau tous les symboles déjà rentrés.
Le premier élément d'un emplacement dans la table est la racine de l'arbre qui gère des collisions.
La table de hashage possède également un champ offset qui indique la taille totale qui l'interpréteur va devoir réserver dans la pile pour les déclarations des variables.
Voici le type utilisé par la table de hashage :

```
#define MAX_TABLE  XXXXU
typedef Noeud *Arbre;
typedef struct
{
      int offset;
      Arbre table[MAX_TABLE];
} Tablesym;
```

13.1.2 Arbre de gestion des collisions

Ici la structure est plus complexe car toutes les informations relatives aux déclarations du programme pascal doivent être stockées dans les noeuds de l'arbre.
Le principe de base utilisé est celui des arbres binaires de recherche qui permettent une recherche rapide dans l'arbre.

Chaque noeud possède les informations suivantes :

- **char *nom** : Le nom du symbole
- **int adresse** : L'adresse que la variable va utiliser dans la pile de pcode. Si le symbole est une constante, ce champ sert à stocker la valeur de la constante.
- **int taille** : Si le symbole est une variable, ce champ stocke la taille de la variable (1 pour un integer, 2 un tableau de deux integer, etc...). Si le symbole est un type, on stocke la taille qu'une variable de ce type va utiliser.
- **Classes classe** : Permet de stocker la classe du symbole.
- **Types type** : Permet de stocker le type du symbole.

- **struct _noeud *type_de_base** : C'est un pointeur vers le symbole sur lequel le type est basé. Par exemple si le symbole est une variable de type integer, ce pointeur va pointer vers le symbole integer.
- **union Type_infos type_info** : Ce champ sert à stocker les différentes informations concernant le type. Ce champ ne sert donc que si le symbole est un type complexe. (Par exemple pour un tableau, il faut stocker le nombre de dimensions, tout ceci va être détaillé).
- **struct _noeud *localite** : Ce champ est pointeur permettant de gérer les localités des variables et des constantes. Si le symbole est global, ce pointeur est NULL. Si le symbole appartient à une fonction/procédure, ce pointeur pointe vers le symbole de la fonction/procédure correspondante.
- **struct _noeud *fg, *fd** : Pointeur vers le fils gauche et droit. N'oublions que c'est un noeud d'un arbre binaire de recherche.

Voici le type utilisé par un noeud :

```
typedef struct _noeud
{
      char *nom;
      int adresse;                /* Place occupée dans la pile de var pcode */
      int taille;                 /* Taille occupée en mémoire */

      Classes classe;             /* La classe (var | const | prog ...) */

      struct _noeud *type_de_base;/* Le type sur lequel le noeud se base
                                     par exemple un tableau d'entier */
      Types type;                 /* Le type (scalaire | tableau | record) */
      union Type_infos type_info; /* Représentation d'un tableau ou d'un champ de record

      struct _noeud *localite;    /* Localité de la variable, NULL = globale */

      struct _noeud *fg, *fd;
} Noeud;
```

Tout simplement, un arbre est définit comme étant un pointeur vers un noeud :

```
typedef Noeud *Arbre;
```

13.1.3 Les classes des symboles

Les classes des symboles sont représentées par un type énuméré. Les valeurs ont été fixées à des puissances de deux ce qui permet d'effectuer des masques et des recherches de plusieurs classes à la fois. En fait cette technique se base sur la représentation binaire, car chaque valeur du type énuméré n'aura qu'un seul bit à 1 dans la forme binaire. Ainsi celà permet de les combiner facilement. La recherche est également accélérée. Exemple sur les masques :

```
CLASS_VAR   = 1  -> 0001
CLASS_CONST = 2  -> 0010
CLASS_PROG  = 3  -> 0100
```

```
CLASS_VAR | CLASS_CONST -> 0011

recherche de CLASS_VAR | CLASS_CONST dans CLASS_VAR :
0001 & 0011 = 0010   != 0  -> test OK

recherche de CLASS_VAR | CLASS_CONST dans CLASS_CONST :
0001 & 0011 = 0001   != 0  -> test OK

recherche de CLASS_VAR | CLASS_CONST dans CLASS_PROG :
0001 & 0011 = 0100   == 0  -> test KO
```

Voici le type utilisé pour les classes :

```
typedef enum
{
   CLASS_PROG      = 1,
   CLASS_CONST     = 2,
   CLASS_VAR       = 4,
   CLASS_TYPE      = 8,
   CLASS_CHAMP     = 16,
   CLASS_ENS       = 32,
   CLASS_PROCEDURE = 64,
   CLASS_FONCTION  = 128
} Classes;
```

Par convention on fixe le début de chaque élément du type énuméré par "CLASS_".

13.1.4 Les types des symboles

Les types des symboles sont représentées par un type énuméré. Les valeurs sont également fixées à des puissances de deux, le fonctionement des masques est exactement le même que celui des classes de symboles. Voici le type utilisé pour les types :

```
typedef enum
{
   TYPE_SCALAIRE    = 1,
   TYPE_TABLEAU     = 2,
   TYPE_RECORD      = 4,
   TYPE_CHAMP       = 8,
   TYPE_INTER       = 16,
   TYPE_BOOLEAN     = 32,
   TYPE_CHAR        = 64,
   TYPE_PARAMETRE   = 128
} Types;
```

Par convention on fixe le début de chaque élément du type énuméré par "TYPE_".

13.1.5 Stockage des informations sur les types complexes et les fonctions/procédures

Les types complexes et les fonctions/procédures ont besoin d'informations supplémentaires qui ne peuvent pas être stockées directement dans les noeud des arbres.

Ce type union permet de faire un embranchement selon le type du symbole. Les symboles représentés par ce type sont les champs de record, les tableaux, les intervalles ainsi que les fonctions/procédures. Voici la déclaration du type :

```
union Type_infos
{
      Type_tableau tableau;
      Type_champ champ;
      Type_intervalle intervalle;
      Type_fonction fonction;
};
```

Union sur les tableaux Pour un tableau, on a besoin de stocker le nombre de dimensions. Pour chaque dimension, on doit stocker l'indice de début (car un tableau ne commence pas forcément par 0) ainsi que la taille de la dimension. Voici les types utilisés pour les déclarations de tableaux :

```
/* Représentation d'une dimension d'un tableau */
typedef struct
{
      int indice_debut;
      int taille;
} Dim_tableau;

/* Représentation d'un tableau */
typedef struct
{
      int nb_dim;
      Dim_tableau *tab;
} Type_tableau;
```

Union sur les champs Pour un champ d'un record, on a besoin de stocker le décalage par rapport au premier champ du record. Le premier champ du record a pour décalage zéro. La taille des champs suivants est la taille cumulée des champs précédents. Pour chaque champ, on doit également stocker à quel record le champ appartient. Voici les types utilisés pour les déclarations de records :

```
typedef struct
{
      int decalage;
      struct _noeud *from_record;
} Type_champ;
```

Union sur les intervalles Pour un intervalle, il suffit de stocker l'indice de début et de fin. Voici les types utilisés pour les déclarations des intervalles :

```
typedef struct
{
      int indice_debut;
      int taille;
} Dim_tableau;
```

Union sur les fonctions/procédures Pour les fonctions/procédures il faut stocker le nombre de paramètres. Pour chaque paramètre il faut stocker le type du paramètre. Il faut également stocker si le paramètre est en passage par valeur ou en passage données/résultat. Voici les types utilisés pour les déclarations des fonctions/procédures :

```
/* Représentation d'un paramètre des fonctions/procédures */
typedef struct
{
      struct _noeud *param;  /* Tableau de pointeur vers les types des paramètres */
      char var;              /* 0 si passage par valeur */
} Param;

/* Représentation des fonctions/procédures */
typedef struct
{
      int nb_param;    /* Nb de paramètres */
      Param *params;   /* Liste des paramètres */
} Type_fonction;
```

13.2 Fonctionnement de la table des symboles

13.2.1 Calcul de la clé de hashage

La fonction de hashage doit retourner une clé qui est comprise entre 0 et MAX_TABLE. Elle doit permettre une dispertion maximale des symboles afin d'essayer de conserver un accès direct aux symboles. La fonction utilisée est tirée du livre "Maitrise des algorithmes en C", Edition O'Reilly. Cependant, elle a été adaptée car par momment la fonction retournait des clés négatives. Donc tout a été passé en unsigned (y comprit le define MAX_TABLE en ajoutant U à la fin de la valeur). Voici la fonction de hashage :

```
unsigned int hash (char *s)
{
   unsigned char *p = (unsigned char *) s;
   unsigned int res = 0;
   unsigned int tmp;

   while (*p)
   {
      res = (res << 4) + (*p);
      if ( (tmp = (res & 0xf0000000U)))
      {
         res = res ^ (tmp >> 24);
         res = res ^ tmp;
      }
      p++;
   }

   return res % MAX_TABLE;
}
```

13.2.2 Gestion des collisions

Parfois, deux symboles peuvent avoir la même clé de hashage. Il y a alors une collision que l'on doit gérer. Afin d'améliorer les performances de la table des symboles, les collisions sont gérées de deux manières. La première consiste à recalculer une seconde clé de hashage, et ainsi effectuer des sauts dans la table. La seconde est d'insérer le symbole dans l'arbre associé à la case de la table de hashage désignée par la première clé.

13.2.3 Ajout d'un symbole

Il y a deux fonctions qui permettent d'ajouter un symbole dans la table des symboles. La première insère tous les symboles sauf les champs de record. En effet les champs de records sont particuliers, car pour deux records différents, on peut avoir un champ de même nom. Pour plus de simplicité, des fonctions dédiées aux champs de records ont étés ajoutées.

Le principe de base de l'ajout est le suivant :
- Calcul de la clé de hashage
- Si la case est libre, on insère en tête de l'arbre
- Si la case est occupée, on effectue des sauts succéssifs afin de trouver de trouver une case libre.
- Si une case libre est rencontrée lors des sauts, on insère en tête de l'arbre.
- Sinon on insère dans l'arbre de la clé initiale.

L'ajout dans les arbres autorise deux symboles identiques par leur nom pour deux localitées différentes.

Pour les champs de records, on autorise deux noms égaux si les deux symboles appartiennent à deux records différents.

13.2.4 Recherche d'un symbole

La recherche d'un symbole se fait grâce à trois fonctions :
- La première permet de rechercher un symbole de localité globale.
- La deuxième permet de rechercher un symbole pour une localité donnée.
- La troisième permet de rechercher un champ de record.

Le principe de base de la recherche est similaire à celui de l'ajout d'un symbole. Voici le déroulement de l'ajout :
- Calcul de la clé de hashage
- Si le symbole en racine d'arbre est égal à celui recherché, on renvoi un pointeur vers cet emplacement
- Si le symbole est différent, on effectue des sauts succéssifs afin de rechercher un symbole égal dans les racines des arbres des sauts effectués.
- Si un symbole en racine d'arbre est identique à celui recherché, on revoi un pointeur vers cet emplacement
- Sinon on recherche dans l'arbre de la clé initiale.

13.3 Interface d'utilisation

Il faut commencer par déclarer une table des symboles en utilisant le type **Tablesym**. Il faut ensuite initialiser cette table. Quand on a finit de se servir de la table déclarée, il faut la supprimer afin de libérer tout l'espace mémoire. Ensuite l'utilisation se résume aux recherches et ajouts.

Par contre l'ajout des informations concernant les types complexes doit se faire en attaquant directement les structures des noeuds retournés lors des ajouts. Quelques macros d'accès ont cependant été créées.

13.3.1 Initialisation de la table

Il faut apeller la procédure **void hash_init (Tablesym *t)**

13.3.2 Suppression de la table

Il faut apeller la procédure **void hash_free (Tablesym *t)**

13.3.3 Les ajouts

Voici les fonctions d'ajout :
- **Arbre hash_add (Tablesym *t, Classes c, Arbre localite)** : permet d'ajouter n'importe quel symbole, le tout étant de préciser sa classe et sa localité
- **Arbre hash_add_champ (Tablesym *t, struct _noeud *pere)** : elle permet d'ajouter un champ de record pour le record désigné par ***pere**

Ces deux fonctions retournent NULL si l'ajout n'est pas possible, ou bien un pointeur vers le noeud créé.

13.3.4 Les recherches

Voici les fonctions de recherche :
- **Arbre hash_search (Tablesym *t, char *s, Classes c)** : permet de rechercher un symbole ***s** de localité globale et de classe **c**
- **Arbre hash_search_localite (Tablesym *t, char *s, Classes c, Arbre localite)** : permet de rechercher un symbole ***s** de localité **localite** et de classe **c**
- **Arbre hash_search_champ (Tablesym *t, char *s, struct _noeud *pere)** : permet de rechercher un champ ***s** du record ***pere**

Ces trois fonctions retournent NULL si le symbole recherché n'existe pas, ou bien un pointeur vers le noeud représentant le symbole recherché.

13.3.5 Les macros d'accès

Ces macros ne couvrent pas la totalité des informations, mais seulement les plus utiles. Voici les macros utilisables pour l'accès aux informations des types :

```
/* Accès aux infos des tableaux */
#define TAB_TAB type_info.tableau.tab
#define TAB_NB_DIM type_info.tableau.nb_dim
#define TAB_INDICE_DEBUT(dim) type_info.tableau.tab[(dim)].indice_debut
```

```
#define TAB_TAILLE(dim) type_info.tableau.tab[(dim)].taille

/* Accès aux infos des intervalles */
#define INTER_DEBUT type_info.intervalle.debut
#define INTER_FIN type_info.intervalle.fin

/* Accès aux infos des fonctions/procédures */
#define FUNC_RETOUR type_info.fonction.retour
#define FUNC_NB_PARAM type_info.fonction.nb_param
#define FUNC_TAB_PARAM type_info.fonction.params
#define FUNC_PARAM(dim) type_info.fonction.params[(dim)].param
#define FUNC_VAR(dim) type_info.fonction.params[(dim)].var
```

14 Les types prédéfinis

Les types prédéfinis utilisables dans la grammaire de notre pascal sont les suivants : char, integer et boolean. Ces trois types sont hachés dans la table des symboles. A l'initialisation de la table, les types sont rajoutés dans la table. Les pointeurs lors des ajouts dans la table sont stockés en mémoire, ceci permet d'avoir un accès encore plus rapide à ces types, car ils sont très fréquemment utilisés. En effet des fonctions telles que l'affichage ou la saisie par exemple vérifient si la variable utilisée est bien de type primitif. On évite donc les appels aux fonctions de la table de symboles.

14.1 Interface d'utilisation

On accède aux pointeurs sotckés via le tableau **Arbre type_predef[3]** définit dans le module. L'accès aux cases du tableau se fait grâce au type énuméré **Predefs** qui associe un identificateur textuel à chacune des cases. Celà permet d'utiliser plus facilement ce tableau.

Voici le type énuméré :

```
typedef enum
{
   PREDEF_INTEGER = 0,
   PREDEF_CHAR    = 1,
   PREDEF_BOOLEAN = 2
}Predefs;
```

Une macro est également définie, elle sert à savoir si un type est primitif ou non : **EST_PRIMITIF(id)**

15 La gestion d'erreurs

Le module utilisé est presque le même que celui de l'interpréteur, seuls les codes de retour et messages d'erreurs ont été changés. Les fonction free_all(...) et usage(...) ont été adaptées.

16 Rattrapage et Synchronisation d'erreurs

Ces modules ont étés traités mais non (ou peu) implémentés. En effet, implémenter entièrement de tels modules prend beaucoup de temps et malgrès les réflexions, les tests et les "squelettes" faits sur le rattrapage et la correction des erreurs, nous avons décidé de ne pas continuer.

16.1 Rattrapage

Le rattrapage s'effectue dans la fonction *teste* du module syntaxique. Il se place dans le switch, c'est à dire dans le cas où le token attendu n'est pas trouvé. Il faut donc chercher si l'erreur est "grave", ou s'il peut s'agir d'une faute de frappe. Le rattrapage s'implémente donc au cas par cas selon les tokens.

exemple 1 Voici l'implémentation dans le cas de l'affectation.
Si ce cas est actif, cela veux dire que l'on attendait un AFFEC_TOKEN mais qu'un autre token est trouvé. Il faut donc afficher une erreur pour la signalée à l'utilisateur, puis ensuite essayer de voir l'erreur comise.

```
case AFFEC_TOKEN:
        erreur (ERR_AFFEC );
```

si on trouve un NUM_ TOKEN, alors l'erreur est peut-être un oublit

```
        if (token == NUM_TOKEN )
        {
           printf ("insertion affec_token \n");
           return TESTE_CORRIGE;
        }
```

si le token trouvé est '=', alors le problème est peut-être clavier (on teste alors si le caractère trouvé n'est pas une touche proche du caractère recherché

```
        if (token == EGAL_TOKEN ||
            sym[0] == ':' ||
            sym[0] == '!' ||
            sym[0] == 'm' ||
            sym[0] == 'l' ||
            sym[0] == ')' ||
            sym[0] == '$' )
        {
           next_token();
           /* faire attention : si on a 2 ou 3 char. next_token donne le
           char suivant et pas l'expr à affecter */
           if (token == TOKEN_INCONNU)
              next_token();

           printf ("corrigé \n");
           return TESTE_CORRIGE;
        }
        else
```

```
            printf ("non corrigé \n");

         break;
```

.

exemple 2 Plus facilement pour le 'point', une seule lettre testée.

```
      case POINT_TOKEN:
         erreur (ERR_POINT );
         if (token ==  PT_VIRG_TOKEN ||
             token == VIRG_TOKEN ||
             sym[0] == ':' )
         {
            next_token();
            printf( "=point=corrigé\n");
            return TESTE_CORRIGE;
         }
         else
            printf( "=point=non corrigé\n");

         break;
```

16.2 Syncronisation

La syncronisation se situe aussi dans la fonction *teste* du module syntaxique, mais en plus dans les appels de cette fonction dans les procédures des grammaires.
La nouvelle signature de teste est : **char teste (Tokens t, Tokens synchro_token)**. En effet, la syncronisation dépend moins d'une erreur de frappe que d'une grosse erreur syntaxique : il faut donc que le token de syncronisation, c'est a dire la "fin de phrase" syntaxique soit donnée par les procédures des grammaires. La syncronisation ne peut être effectuée pour chaque grammaire, une syncronisation sur le token final "end." peut être envisagée.

Fonctionnement Si un token attendu n'est pas trouvé, et si aucune correction n'a pu être effectuée, alors il faut syncroniser ; aprés le switch de la fonction *teste*, il faut faire défiler tout les tokens afin de trouver le token de synchronisation.

```
while (token != synchro_token)
   next_token();
printf(" synchronisation effectuée \n");
return TESTE_SYNCHRO;
```

La fonction renvoie TESTE_SYNCHRO pour signaler qu'elle n'a ni trouvée ni rectifiée le token, alors la fonction appellante se terminera directement, sans chercher les tokens qui ont étés sautés par la synchronisation.

exemple Pour les constantes on décide de syncroniser sur le point virgule final de la grammaire.

```
/**
 * CONST ::= const ID = NUM; { ID = NUM; } | e
 * synchronisation sur ";"
 */
void consts ()
{
   /* ici on est déja sûr que c'est CONST_TOKEN */
   teste (CONST_TOKEN, CONST_TOKEN);
   do
   {
      teste_et_entre (ID_TOKEN, CLASS_CONST);
      if ( teste (EGAL_TOKEN, PT_VIRG_TOKEN) != TESTE_SYNCHRO )
         if ( teste (NUM_TOKEN, PT_VIRG_TOKEN) != TESTE_SYNCHRO )
            teste (PT_VIRG_TOKEN, PT_VIRG_TOKEN);
   } while (token == ID_TOKEN);
}
```

17 Le générateur de code

Le générateur est assez simple si on ne considère pas les sauts imcomplets dans le pcode. En effet, les fonctions importantes sont des fonctions de transcription de mnémoniques en caractères et leurs écritures vers le fichier (via le module d'entrée/sortie).

17.1 Interface d'utilisation

17.1.1 Ecriture des instructions

Pour simplifier l'appel aux générations il y a 3 façons d'insérer une mnémonique, selon si elle a un paramètre et selon le type de ce paramètre. *generer1 (mnemoniques)* Ecrit dans le fichier une instruction et passe à la ligne *generer2i (mnemoniques , int)* Ecrit dans le fichier une instruction, un espace, son paramètre (un entier) et passe à la ligne *generer2c (mnemoniques , char)* Idem pour les paramètres de type caractère.

La création de commentaires dans le fichier Pcode est aussi possible grâce à : *generer_ com (char*)* Qui écrit dans le fichier un commentaire et passe à la ligne

17.1.2 Gestions des sauts incomplets

Précision : Pour la gestion des sauts incomplets l'instruction *BZE* est un peu détournée. Malgrès cela elle reste utilisable avec son utilisation originale.

Fonctionnement des sauts La prise en compte de sauts imcomplets a été faite avec la création d'une pile des sauts. Les instructions de sauts pour lesquelles le paramètre ne peut être placé directement (puisque l'adresse d'arrivée ne sera connue qu'après la génération d'autres instruction, et que le buffer ne permet que l'écriture d'un flux **continu**) ne correspondent plus aux instructions d'origine du Pcode quand elles sont générées : Le paramètre est placé à 0 et n'a plus d'importance, le saut est stoqué en fin de fichier, après la balise **$J**. C'est dans l'interpréteur, après la lecture du fichier pcode, lorsque toutes les instructions sont empilées, que la mise à jour des valeurs des sauts sera effectuée. Pour chaque ligne suivant la balise **$J**, on remplacera la valeur 0 du paramètre de l'instruction à la ligne enregistrée par la valeur enregistrée du saut.

Exemple : le IF

```
void si ()
{
/* entier qui va contenir le numéro du saut */
   int sauv;

   teste (IF_TOKEN);
   cond ();

/* génération de l'instruction de saut, param=0
   car ne correspond à rien à ce moment */
   generer2i (BZE, 0);

/* enregistrememement de l'endroit où se trouve
   l'inst de saut */
/* ligne_pcode = Départ */
```

```
   sauv = pile_saut_push (&sauts, ligne_pcode);
   teste (THEN_TOKEN);
/* les instructions sont codées (on ne connait pas le
   nombre de lignes utilisées) */
   inst ();

/* Enregistrement de la ligne d'arrivée du saut et association
   avec le numéro du saut */
/* ligne_pcode = Arrivée */
   pile_saut_update (&sauts, sauv, ligne_pcode+1);
}
```

Ainsi, le fichier ressemblera à : ...
LDI 2
EQL *la condition du if*
BZE 0 #ligne 22 (départ) *le saut incomplet*
... les instructions si condition=VRAI
... #ligne 44 (arrivée) *suite du programe*
HLT
$J
22 44

Utilisation de la pile des sauts :

- Initialisation par *pile_saut_init (Pile_saut *)*. A effectuer avant toute génération.
- Pour sauvegarder l'adresse de l'instruction (ligne) et récupérer le numéro correspondant : *int pile_saut_push (Pile_saut *, int)*. A effectuer après la génération de l'instruction de saut.
- Pour indiquer l'adresse d'arrivée du saut : *pile_saut_update (Pile_saut *, unsigned int , int)*. En entrée la pile des sauts, le numéro du saut à mettre à jour (retourné par *pile_saut_push(...)*) et enfin la valeur du saut. A effectuer après la dernière instruction des instruction à sauter. (généralement après inst() ;)
- Pour écrire la pile des sauts dans le fichier de sortie : *pile_saut_to_file (Pile_saut *)*. A effectuer à la fin de la génération des instructions (après le HLT)
- Pour libérer la mémoire ocupée par la pile des sauts : *pile_saut_free (Pile_saut *)*. A effectuer après l'écriture de la pile dans le fichier.

17.2 Fonctionnement interne du module

La génération des instructions dans le fichier ne s'effectue que si la variable *corrige*, qui indique si une erreur de compilation est survenue, est fausse. Ainsi, il n'y a pas de Pcode "faux" généré.

Pour une meilleure lisibilitée du fichier Pcode la fonction *numeroter (int)*, appellée par les procédures de génération, permet de numéroter le fichier Pcode, en tant que commentaire après l'instruction. Le char (booléen) *numerotation* permet d'activer ou de désactiver cette numérotation.

Il est aussi possible d'interdire ou de permettre l'enregistrement des commentaires dans le fichier par le booléen *commentaire*.

Ces 2 options n'ont pas d'accès à partir de la ligne de commande. (facilement implémentable)

Troisième partie

Réponses aux questions

18 Chapitre 1 : Traducteur et interpréteur

1. ```
   # 0 : A
   # 1 : SMALLEST
   # 2 : LARGEST

   int 3  # réserve 3 emplacements en mémoire

   lda 0  # Empile &0 pr lire A
   inn    # Lecture de A

   lda 1  # Empile &1 pr affecter SMALLEST
   lda 0  # Empile l'adresse de A
   ldv    # Remplace l'adresse de A par sa valeur
   sto    # Remplace &1 par la valeur de A

   lda 2  # Empile &2 pr affecter LARGEST
   lda 0  # Empile l'adresse de A
   ldv    # Remplace l'adresse de A par sa valeur
   sto    # Remplace &2 par la valeur de A

   brr 23 # "lda 0  # Empile l'adresse de A" (après while)

   lda 0  # Empile l'adresse de A
   ldv    # Remplace l'adresse de A par sa valeur
   lda 2  # Empile l'adresse de LARGEST
   ldv    # Remplace l'adresse de LARGEST par sa valeur
   gtr    # A > LARGEST
   bzr 5  # Si test faux, on saute l'affectation
   lda 2  # Empile &2 pr affecter LARGEST
   lda 0  # Empile l'adresse de A
   ldv    # Remplace l'adresse de A par sa valeur
   sto    # Remplace &2 par la valeur de A

   lda 0  # Empile l'adresse de A
   ldv    # Remplace l'adresse de A par sa valeur
   lda 1  # Empile l'adresse de SMALLEST
   ldv    # Remplace l'adresse de SMALLEST par sa valeur
   lss    # A > SMALLEST
   bzr 5  # Si test faux, on saute 2 instructions pr la lecture de A
   lda 1  # Empile &1 pr affecter SMALLEST
   lda 0  # Empile l'adresse de A
   ldv    # Remplace l'adresse de A par sa valeur
   sto    # Remplace &1 par la valeur de A

   lda 0  # Empile &0 pr lire A
   inn    # Lecture de A

   lda 0  # Empile l'adresse de A
   ```
   ```

```
ldv    # Remplace l'adresse de A par la valeur
ldi 0  # Empile la valeur 0
eql
bzr -26# Début de la boucle

lda 1  # Empile l'adresse de SMALLEST
ldv    # Remplace l'adresse de SMALLEST par sa valeu
prn    # Affiche la valeur de SMALLEST
prl
lda 2  # Empile l'adresse de LARGEST
ldv    # Remplace l'adresse de LARGEST par sa valeu
prn    # Affiche la valeur de LARGEST
prl
```

2. Oui, le jeu d'instructions est suffisant. Il permet tout à fait de gérer une "machine à entier" car il contient toutes les opérations nécessaires : les opérateurs de base (+ - / *), les opérateurs de comparaisons ($< \leq > \geq = <>$).
3. *CF le code source de l'application.*
4. *CF le code source de l'application.*
5.
6. Oui, un programme pcode peut être incorrect. Comme tout programme il y a les erreurs de syntaxe détectables dès la compilation. Par exemple un mnémonique peut être erroné ou bien il peut manquer un paramètre. Il y a aussi les erreurs qui seront détectées à l'exécution, par exemple la division par zéro ou bien l'accès à une zone mémoire interdite ou inexistante.
7. On peut créer un débugeur de pcode. Le débugeur peut être activé en passant l'option "-d" à l'interpréteur. On a alors un mode pas à pas où chaque instruction est éxécutée après l'appui de la touche entrée. Il y a également une vue sur les instructions précédentes et suivantes, sur le sommet de la pile d'exécution ainsi qu'un affichage du segment de données.
8. L'avantage des branchements relatifs est que l'on peut faire des sauts en positif et négatif. Ce mode est plus proche du mode de fonctionnement de la machine. Il permet aussi de générer un code plus petit/simple et est surtout beaucoup plus lisible. Les deux méthodes de sauts ont étés implémentées.
9. Non réalisé.

19 Chapitre 2 : Analyse syntaxique

1. Les points virgules ont le rôle de séparateur d'instructions. On peut les supprimer si on les remplace par un autre séparateur. En fait le ; n'est pas important en soi, il faut juste avoir un séparateur d'instructions. On peut aussi mettre une instruction par ligne.
2. C'est une instruction qui dit au CPU de ne rien faire. Par exemple 'nop' (no operation) en pcode. Ça peut servir pour faire baisser la température du processeur. C'est utile sur une architecture parallèle, par exemple pour dire à un processeur de laisser travailler les autres processeurs. C'est donc utile pour l'architecture, mais pas forcément pour un langage de programmation.

3. En théorie les const peuvent être utiles dans les var. Mais pas dans notre langage. Le but est d'avoir une analyse vraiment descendante. On peut annuler cette règle en mettant des boucles tant que.
4. Le problème est celui des "dandling else" : savoir à quel if se raccroche le else trouvé. Pour résoudre ce problème il suffit d'introduire les begin/end qui encadrent les if et les else.
5. La méta règle est ambiguë pour l'imbrication des commentaires. L'imbrication des commentaires est souhaitable mais est trop dure à gérer. La règle qui sera utilisée ici est la suivante : dès que l'on rencontre un marqueur de début de commentaire ("(*" ou "{") on passe tout le flot de caractères jusqu'à ce que l'on rencontre le marqueur de fin de commentaire correspondant ("*)" ou "}" respectivement).
6. *CF le code source de l'application.*
7. *CF le code source de l'application.*
8. *CF le code source de l'application.*
9. *CF le code source de l'application.*
10. Pour pouvoir inclure des blancs dans les identificateurs il faudrait mettre un terminateur d'identificateur.

20 Chapitre 3 : Analyse sémantique

1. Au début on insère sans chercher car il n'y a pas de vérification d'existance dans la table des symboles. Un façon de palier à ce problème est de faire un teste_et_cherche avec un teste_et_entre. On peut donc imaginer une fonction teste_et_cherche_et_entre. Nous n'autorisons pas les doubles déclarations (const A=3 ; var A ;). Une variable et le programme pourraient avoir le même nom, ça ne poserait pas de problèmes de sémantique. Seulement ce n'est pas autorisé en pascal.
2. Oui, cette règle est nécessaire, car si on a une constante et une variable de même nom, comment savoir quelle valeur de ces deux symboles utiliser ?
3. *CF le code source de l'application.*
4. L'avantage de la méthode d'un paramètre au compilateur est que ça affiche l'ensemble des symboles. L'utilisation des directives de compilation permet un affichage ciblé des symboles intéressants. Cela nécessite aussi la modification du code source pascal. Seule la première méthode a été implémentée. Ces deux méthodes ne sont pas incompatibles.
5. Ce n'est pas une bonne idée.
6. C'est une très bonne idée car ça augmentera les performances du compilateur. Seulement il faut gérer les problèmes de colision dans la table de hashage.

21 Chapitre 4 : Gestion des erreurs

1.
2. La fonction UNGET_TOKEN n'a pas été implémentée. Le buffer de lecture ne le permet pas, ou rend cette tâche difficile.
3. *CF le code source de l'application.*
4. *CF le code source de l'application.*
5.

22 Chapitre 5 : Génération de code

1. *CF le code source de l'application.*
2. *CF le code source de l'application.*
3. *CF le code source de l'application.*
   Les procédures sont : generer1 (mnemoniques), qui écrit une instruction generer2i (mnemoniques, int), qui écrit une instruction paramétrée par un entier generer2c (mnemoniques, char), qui écrit une instruction paramétrée par un char generer_com (char*), (facultative) qui écrit un commentaire Les mnémoniques sont codées par des entiers pour leurs utilisations dans le programme. Ce codage pourrait être facilement utilisé pour générer du code non lisible, mais le code est ici reconvertit avant sont écriture dans le fichier pcode. Le code générer est donc compréhensible.
4. Comme le programme pascal est erroné, le pcode généré le sera aussi. Il y aura des erreurs à l'interprétation car le programme pcode ne sera pas valide.
5. *CF le code source de l'application.*
6. *CF le code source de l'application.*
7. *CF le code source de l'application.*
8. *CF le code source de l'application.*
9. Il faudrait vérouiller la valeur en écriture. Par exemple on pourrait passer temporairement la variable en constante afin d'interdire sa modification.
   Pour considérer cete variable en tant que locale à la boucle on peut utiliser les instructions créées pour les procédures/fonctions. (exemple : faire un INT 1 pour la variable, la stoquer et utiliser cette copie). Le for implémenté n'utilise pas cette méthode : Une nouvelle instruction CPA permet de faire une copie de l'adresse en sommet de pile en sur-sommet de pile. Ainsi les comparaisons et l'incrémentation de la variable de boucle s'effectue directement sur la variable. Elle contient donc une nouvelle valeur en sortie de boucle.
10. Pas à faire.

23 Chapitre 6 : Variables de type tableau

1. *Non réalisé car implémenté dans le chapitre 8.*
2. *CF le code source de l'application.*
   Les tailles peuvent être des expressions simples et évaluables. Pour implémenter l'évaluation, nous avons créé une fonction expr() spéciale "expr_const()" qui calcule le résultat d'une expression en ne prenant comme opérandes que des constantes déjà déclarées ou des literraux.
3. Pour gérer cette fonctionnalité, il faudrait une fonction expr() qui calcule tant que l'expression est évaluable et qui en même temps générerait le code corespondant dans un "tampon". Le calcul "direct" s'arréterait dès qu'il y a accès à une variable, le pcode dans le tampon serait indispensable et copié dans le fichier. On générerait alors directement dans le fichier la fin du code. Si le calcul "direct" est faisable, alors le pcode dans le Tampon serait supprimé.
4. (a) Dans la table des symboles on stocke l'indice de début ainsi que la taille. Tout ceci va être implémenté dans le chapitre 8.
   (b) *CF le code source de l'application.*

5. (a) DECL : := ID | ID [DIM {,DIM}]
   DIM : := EXPR | EXPR..EXPR
   (b) En mémoire la structure des tableaux multi-dimentionnels est linéaire. On y accède grâce à la formule suivante :

$$\sum_{i=0}^{n} x_i \ \Pi_{j=0}^{i-1} X_j$$

   (c) Il faut stocker le nombre de dimensions. Pour chaque dimension il faut stocker l'indice de départ ainsi que la taille de la dimension.
6. Il faut créer une nouvelle instruction pcode qui vérifie que le sous-sommet est ≥ 0 et $<$ au sommet et qui dépile une fois.
7. *Non implémenté.*

24 Chapitre 7 : Variables de type enregistrement

Non réalisé car implémenté dans le chapitre 8.

25 Chapitre 8 : Définition de types

1. *Non implémenté comme dans le poly.*
2. *CF le code source de l'application.*
3. Il suffit de modifier la grammaire de la règle TYPE comme suit :
   TYPE : := ID | integer | char | boolean | TYPE_RECORD | TYPE_ARRAY
4. (a) *Non implémenté.*
   (b) Ça ne pose de problème car les deux variables sont déclarées avec le même type anonyme. Les pointeurs "type_de_base" pointent tous deux vers le même enregistrement dans la table des symboles.
   (c) *Non implémenté.*
5. Non, c'est très simple à implémenter.
6. *Non implémenté.*
7. *Non implémenté.*
8. Ce n'est pas obligatoire dans le cadre des déclarations semi croisées, par exemple pour une liste chainée il y a un pointeur qui pointe vers le type lui même.
   Il faut juste s'assurer que les déclarations incomplètes ont étés complétées.
9. Pour implémenter le with il faut instaurer une recherche prioritaire dans la table des symboles.
10.

26 Chapitre 9 : Procédures simples

1. *CF le code source de l'application.*
2. *CF le code source de l'application.*
3. *CF le code source de l'application.*

4. La récursivité ne pose pas de problèmes à ce stade du compilateur.
5. La récursivité est implémentable, mais pour cela il faut faire des déclarations incomplètes de procédures.

27 Chapitre 10 : Déclarations locales

1. Une allocation statique des variables ne permet pas de récursivitée.
2.
3.
4. *CF le code source de l'application.*
5. *CF le code source de l'application.*
6.

28 Chapitre 11 : Procédures imbriquées

Non implémenté.

29 Chapitre 12 : Procédures paramétrées

1.
2.
3. Le passage de paramètre par valeur est le seul passage implémenté.
4.

30 Chapitre 13 : Fonctions

1.
2.
3.

31 Chapitre 14 : Edition de liens

Non implémenté.

Quatrième partie
Annexes

A Super conversation

(16 :00 :36) -=Nico=- : rha putain, ça me les casse les localités, ttes les fonctions de hash changent un peu !
(16 :00 :39) -=Nico=- : GRRRRRRRRRRRRRRRRRRRRRRRRRr
(16 :01 :05) =Zeitoun= : pourquoi ? a cause des bases ?
(16 :01 :24) -=Nico=- : à cause du paramètre "localite" que je doi passer à chaque fois
(16 :01 :40) =Zeitoun= : c koi ce param ? un boolean ?
(16 :01 :43) -=Nico=- : pr savoir si on cherche en global ou pr cette fonction là
(16 :01 :52) -=Nico=- : un pointeur
(16 :02 :07) -=Nico=- : qui pointe vers le champ du nom d'une fonction
(16 :02 :17) -=Nico=- : si la variable appartient à cette fonction
(16 :02 :22) =Zeitoun= : faudrai quon mette au point nos 2 iédées, jcroi que ca differe
(16 :02 :25) -=Nico=- : ou vers NULL si la valeur est globale
(16 :02 :52) =Zeitoun= : ya moyen que jte parle encore un peu des baseS ?
(16 :03 :03) -=Nico=- : si c'est compréhensible, oé :)
(16 :03 :39) =Zeitoun= : bon, alors on va commencé par prendre le program principal come une fonction.
et tout est un ensemble de fonction imbriquée.
(16 :03 :47) -=Nico=- : (moi la localité c juste pr savoir que cette variable X appartient à la fction Y)
(16 :03 :47) =Zeitoun= : (des ue ca suit pas tu dit)
(16 :04 :12) -=Nico=- : déjà je fait pas les fonctions imbriquées, tro le bordel à gérer !
(16 :04 :13) =Zeitoun= : atten, jvai te dire comment jpensé géré ca. (et que lapartenance on sen fou)
(16 :04 :31) -=Nico=- : bah nan, c to important l'apartenance, sinon ds la fonction
(16 :04 :44) =Zeitoun= : oui, important, mais gérable autrement
(16 :04 :48) -=Nico=- : XA on déclarer une var toto, et elle serai accéssible aussi dans la fcontion XB
(16 :04 :49) =Zeitoun= : sans avoir de pointeur
(16 :05 :03) =Zeitoun= : atten que jesplique.
(16 :05 :21) =Zeitoun= : heuu, ducoup, on peu appelé qune seule fonction a la fois ?
(16 :05 :21) -=Nico=- : (je veu bien que c'est de localité de niveau 1, ms faut savoir à quelle fonction la var appartient !)
(16 :05 :23) -=Nico=- : oki
(16 :05 :59) -=Nico=- : à moins de créer un fork, je sais pas comment on pourrai apeller 2fctions en même tps !
(16 :06 :28) =Zeitoun= : et tu compte viré les variables locales a la fin de chaque fonction, ou les laissé dans la tab de hash ?
(16 :06 :34) -=Nico=- : je laisse !
(16 :06 :40) =Zeitoun= : haaaaaaaa
(16 :06 :57) -=Nico=- : prkoi virer ? ça sert à rien !
(16 :07 :15) =Zeitoun= : jveu dire pour les fonction en meme temp : une proc qui appelle une autre proc
(16 :07 :20) -=Nico=- : si tu les vire une fois que tu a fait l'appel de fonction, ça veu dire que c'est une fonction kleenex
(16 :07 :36) -=Nico=- : ça pose pas de pbs ça !
(16 :07 :55) =Zeitoun= : bin, qd ta fonction est fini tu peux pu te servir des variables locales !
(16 :08 :34) -=Nico=- : bah je pourai pa m'en servir ailleurs vu que j'ai mon pointeur vers le nom de la fonction

(16 :08 :52) =Zeitoun= : oui, bin c la quon differe. donc je texplique ce que je pensé :
(16 :08 :59) -=Nico=- : c tro la merde à commencer à virer les variables !
(16 :09 :18) -=Nico=- : le hashage ça risque de foirer et tou et tou !
(16 :09 :57) =Zeitoun= : (au pire on "désactive" non ? avec un booean, ou on vire le nom) ?
(16 :10 :09) -=Nico=- : mais ça reste pas actif !
(16 :10 :23) =Zeitoun= : bin tantmis, on n'en veux plus !
(16 :10 :26) =Zeitoun= : tant mieux
(16 :10 :34) -=Nico=- : ça reste ds la table de hash, ms on peu plu s'en servir
(16 :11 :16) =Zeitoun= : oui, bin c bien comme ca
(16 :12 :04) -=Nico=- : au début de ma decl de focntion j'ai cur_localite = pointeur_ver_ma_fonction ; et juska la fin de la déclaration de la fonction, les ajouts se font pr cette fonction là, et la recherche est prioritaire pr cette fonction là !
(16 :12 :57) =Zeitoun= : et sil y a une onction entre les 2 ?
(16 :13 :12) -=Nico=- : j'ai di pas de declaration de fonction imbriquée
(16 :13 :38) -=Nico=- : ou bien tu voulai dire appel de fction ?
(16 :13 :43) =Zeitoun= : oui ! !
(16 :13 :54) -=Nico=- : (soi précis stp, TP tro gro pr etre flou)
(16 :13 :59) -=Nico=- : j'y ai pensé aussi !
(16 :14 :20) =Zeitoun= : si on a A dans le prig principal, A dans la 1° fonction appellé, qui elle-meme epellle une 2eme fonction qui a une var A
(16 :14 :58) -=Nico=- : dc ds la fonction qui gère les appels de fonctions y fau faire un truc du genre :
sav_localite = cur_localite ;
cur_localite = la fonction_que_j'apelle ;
les insts ;
cur_localite = sav_localite ;
(16 :15 :25) -=Nico=- : ainsi on se positionne tjs ds la localité de la bonne fonction !
(16 :15 :42) -=Nico=- : et on la restaure à la fin
(16 :19 :38) =Zeitoun= : Les bases ca sert a faire un correspondance entre les valeur relatives des variables globales et les vraies valeur dans la pile.
Base 0 pour le programme principale : base(0) = 0 .
donc adresse de A dans le prog principale = adresse(A) + base(0) = adresse(A)

appel dune procedure : on passe a la base 1. base(1) = adresse actuelle dans la pile (ex :212)

Dans la premeier fonction apellée on a une var locale A. On fait un INT 1

l'adresse de A est 1 .

Quand on fait A := 8 ; on cherche A dans le hash, pi on recupere son adresse ET sa BASE (ici 1)

on a donc son adresse : adresse(A) + base(1) = 1 + 212 = 213. => en absolu dans la pile des var.

Et quand on a plusieurs procédure apéllée déja, on ca cherché le A avec la plus grande base. (cad, le plus loin dans l'abre de hash)
(16 :20 :13) -=Nico=- : ur dans la pile. (pile pcode ?)

(16 :20 :50) =Zeitoun= : valeur dans la pile des variables
(16 :21 :43) -=Nico=- : dc en fait la base c le niveau de localité, 0 = global, 1 = ds une fonction, 2 ds une fonction qui est ds une fonction
(16 :21 :48) =Zeitoun= : l'adresse dune variable c'est sa position de stoquage de valeur dans la pile de pcode
(16 :21 :51) -=Nico=- : etc etc ..?
(16 :21 :56) =Zeitoun= : oui
(16 :22 :09) -=Nico=- : bah ça marche avec mon poineur
(16 :22 :14) =Zeitoun= : ouffffffff
(16 :22 :15) -=Nico=- : base = 0 si localite) NULL
(16 :22 :24) -=Nico=- : base = 1 si lovalite !=N ULL
(16 :22 :24) =Zeitoun= : mais bon, ta compris le rpincipe ?
(16 :22 :41) =Zeitoun= : enfin, moi ce que je veux c pouvoir récup la bonne vleur
(16 :22 :48) =Zeitoun= : enfin, la bonne adresse.
(16 :23 :14) =Zeitoun= : ducoup, toi, quesque tu me donne qd je demande une adresse de variable ?
(16 :23 :32) =Zeitoun= : ladrese relative + un pointeur vers la fonction ?
ladrese directe ?
(16 :23 :49) -=Nico=- : pr l'instant rien j'arrete pa de dire que je me galère rien qu'avec le sémantique !
(16 :24 :02) -=Nico=- : que j'ai pa core géré pr ttes ces adresses !
(16 :24 :33) -=Nico=- : déjà l'instant je veu que l'on puisse utiliser les var locales ds les fonctions !
(16 :24 :53) -=Nico=- : elle se déclarent, ms je suis entrain de faire ttes les modifs qui faut !
(16 :26 :10) =Zeitoun= : oki, moi aussi. Mais tes truc de pointeurs vers la fonction, ca fera que l'adresse stoquée sera relative ou absolue ?
(16 :26 :23) =Zeitoun= : bin non, c imposible absolue, jsuis con
(16 :26 :25) -=Nico=- : JEN SAIS RIEEEEEEEEEEEENNNNNNNNNNNNNNNNNNNNNNNNNNNNNNN
(16 :26 :35) -=Nico=- : :)
(16 :26 :54) =Zeitoun= : bon, donc faudrai faire le truc des bases....
(16 :27 :10) -=Nico=- : int getBase () {
(16 :27 :26) -=Nico=- : return ptr->localite == cur_localite ;
(16 :27 :27) -=Nico=- : }
(16 :27 :48) =Zeitoun= : jlavais fait les bases avec une pile_var
(16 :28 :03) -=Nico=- : ?
(16 :28 :38) =Zeitoun= : bin, les bases ca se geré par une pile en fait.
ou no empilé les bases roissantes
(16 :28 :50) -=Nico=- : j'compren rien
(16 :29 :03) -=Nico=- : lol
(16 :30 :12) =Zeitoun= : imagine ya pas tout tes truc de pointeur sur les fonction (juste pour texpliké) :
Appell dune procedure : on est en base 1, tu rentre une variable locale A, adresse 1
(16 :31 :01) =Zeitoun= : la base 1 est "crée" a lappel de cette procedure. ele corespond au pointuer de la pile au moment de lappel
(16 :31 :46) =Zeitoun= : (puisque les variable qui vont eter crée par le INT se feront au somet de pile)
donc au dessus de la valeur e base(1)
(16 :31 :51) -=Nico=- : (pile des vairalbes ds l'interpréteur ?)
(16 :32 :02) =Zeitoun= : oui

(16 :32 :18) =Zeitoun= : vu que c'est en fonvtion de la pile des var
(16 :33 :28) =Zeitoun= : Donc pour noter variable A, on va la mettre dans la table de hash a : (offset =0 (RAZ puor les var locales) et BASE = base_courante() = 1)
(16 :34 :22) -=Nico=- : (bon en gro, moi ce que je stocke : la taille des paramètres ds le champ du noeud correspondant à l'identificateur de la fonction, et ds le champ adresse de chacune des variables locales à cette fonction, je stocke la position par rapport à zéro ; si var a,b ;integer ; adresse de a = 0, adresse de b=1)
(16 :34 :30) =Zeitoun= : Pour trouvé la valeur de A, on va devoir allé voir a lendroit de la pile des var ou il est stoké :
hash me donne son adresse relative : 1 , et sa base : 1. je vais donc en 1 + base(1)
(16 :35 :22) =Zeitoun= : oui. je pense que c'est ca...
(16 :36 :01) =Zeitoun= : mais il me faut aussi la base correspondante.
donc surement récupérée avec le pointeur vers la fonction,non ?
(16 :36 :27) -=Nico=- : bah la base c soi 0 ou 1, c bien ça ?
(16 :36 :42) =Zeitoun= : non
(16 :36 :43) -=Nico=- : vu qu'on fai pa de déclarations de fonction imbriquée !
(16 :36 :53) =Zeitoun= : la base c croissant.
(16 :37 :07) -=Nico=- : oé, ms la base c géré ds la pile pcode ?
(16 :37 :26) -=Nico=- : c ds ta pile pcode que tu cré ttes les bases ?
(16 :38 :27) =Zeitoun= : base 0,
premier appel a Y : 1
2emem appel : apel a T dans Y : base =2
3eme appel : proc W appellée dans T : base = 3
(16 :38 :51) =Zeitoun= : non, c'est géré en C
(16 :39 :03) =Zeitoun= : pas dans le pcdoe, mais en fonction
(16 :39 :09) -=Nico=- : oé, ms je voi tjs pas à koi ça sert !
(16 :40 :21) =Zeitoun= : bin toi comment tu fait pour pécho une adrese dune var locale ? alors que tu ne sais pas son adresse dans la pile_var, puisque on ne sais pas ce qu'il y a déja dempilé
(16 :40 :52) =Zeitoun= : opur les var globale, on le sais car on comence a 0
(16 :41 :36) -=Nico=- : bah, qd on fai un apel de fcontion on fait un int, et le int y cré la place qui faut et il change le pointeur de sommet de pile c bien ça ?
(16 :42 :16) -=Nico=- : ?
(16 :42 :28) =Zeitoun= : oui
(16 :42 :48) -=Nico=- : bon, dc l'accès à la première variable locale c bien 0 ?
(16 :43 :00) -=Nico=- : comme si on été ds une pile normale ?
(16 :43 :24) =Zeitoun= : oui
(16 :43 :31) -=Nico=- : bon, bah pa besoin de base alors
(16 :43 :59) -=Nico=- : vu que moi ds le champ adresse de ma première variable locale d'une fonction, je stocke 0
(16 :44 :09) =Zeitoun= : mais si tu fait LDA 0 on ira cheché le segment 0 de la pile.
(16 :45 :46) =Zeitoun= : si ton appel de fonction se fait alrs que ta deja plein de calcul. donc ta pile de var est a 300.
tu fait un INT 2. pour a et b. Le int te reserve 2 place en sommet de pile : 301 et 302
ok ?
(16 :46 :05) -=Nico=- : oui
(16 :46 :21) =Zeitoun= : donc si tu ne met que adrese de a = 0. il ne saura jamais quil faut alé cherché A a l'adresse 301
(16 :46 :48) -=Nico=- : bah oé, ms en C on peu pa savoir qu'on é à l'adresse 300 !

(16 :47 :23) =Zeitoun= : mais linterpréteur si !
(16 :48 :19) =Zeitoun= : dans linterperteur, il va stoké les correspondance numéro de base / adresse de la pile
(16 :49 :07) -=Nico=- : dc en gro on se chamboule pr une variable qu'on fait ++ à chaque appel de fonction ?
(16 :49 :24) -=Nico=- : et – qd on a fini l'appel de fonction
(16 :49 :27) -=Nico=- : c bien ça ?
(16 :50 :11) -=Nico=- : function a(){ b () ; }
(16 :50 :48) -=Nico=- : function a(){ b () ; c() }
function b(){ echo "tot" }
function c(){ echo "titi" }
(16 :50 :56) -=Nico=- : si je fait a () ;
(16 :51 :19) -=Nico=- : base ++ ; a() apelle b () dc base++ ;
(16 :51 :28) -=Nico=- : appel à b(), fini, je fai base – ;
(16 :51 :37) -=Nico=- : c bien ça ?
(16 :51 :53) -=Nico=- : en gro la base ça sert à savoir à quel niveau d'appel on se situe ?
(16 :51 :55) =Zeitoun= : bin, les bases c codé yavai quasi rien afaire....
apres, c pouvoir stokée les bases dans la les variables de la table de hash, c ca que je veux.. (atta je mate)
(16 :52 :11) -=Nico=- : ms prkoi ds la table de hash ?
(16 :52 :13) =Zeitoun= : ouiiiiiiiiii
(16 :52 :14) -=Nico=- : ça sert à rien !
(16 :52 :17) =Zeitoun= : bin, dans les variables
(16 :52 :41) =Zeitoun= : si il y a 3 variables A, pour savoir a quelles bases elle appatiennes
(16 :52 :56) -=Nico=- : les variables la table de h elle peuvent pas te dire, cette variable X a été apellé pa b() qui a été apellé par c() qui a été apelle pa d()
(16 :53 :19) -=Nico=- : ds le hash on stocke pas une liste des apells de fonctions !
(16 :53 :57) =Zeitoun= : non, mais qd on met nue variable, on met la base !
(16 :54 :09) -=Nico=- : ms on la connai pa à l'avance la base
(16 :54 :30) =Zeitoun= : on est en base 4, donc si je met var A, ca sera adresse =0 et base = 4
comme ca, moi je retrouve son adresse
(16 :54 :38) -=Nico=- : si tu a 15fonctions qui s'apelle, tu le sais seulement ds le "main pascal" et non pas au momment des déclarations des entêtes de fonctions ! !
(16 :55 :42) -=Nico=- : c comme si tu me demandai de stocker ds la table de hash le nombre d'itération que va faire une boucle for, c pa possible :
(16 :55 :42) -=Nico=- : !
(16 :56 :22) =Zeitoun= : wé, aten, tu menbrouille. merde. lol
(16 :56 :23) -=Nico=- : imagine t'a un appel de fonction ds un for de 1000 itérations, tu va pa stocker ds la table de hash les 1000 même variable de la même fonction : !
(16 :56 :34) =Zeitoun= : 10.3 Allocation des variables
/pcp012.html de l"noncé en html
(16 :57 :38) -=Nico=- : bah oui, ns on fait de l'allocation dynamique hein !
(16 :57 :46) =Zeitoun= : oui
(16 :57 :50) -=Nico=- : " il devient impossible de connaître les adresseslors de la compilation (le nombre d'appels récursifs ne pouvantêtre connu). On va donc allouer dynamiquement sur la pile lesvariables locales."
(16 :58 :11) -=Nico=- : c pareil pr tes bases, vu que ça s'incrémente au fur et à mesure des appels

(16 :59 :18) -=Nico=- : capté ?
(16 :59 :31) =Zeitoun= : wé
(16 :59 :47) -=Nico=- : (16 :00 :36) -=Nico=- : rha putain, ça me les casse les localités, ttes les fonctions de hash changent un peu !
(17 :00 :02) =Zeitoun= : putin, ca fait 1h
(17 :00 :02) -=Nico=- : lol, une heure plus tard, on é enfin d'accord :)
(17 :00 :15) =Zeitoun= : mais bon, alors, jenfait koi de mes bases ?
(17 :00 :21) -=Nico=- : bah tu les gardes
(17 :00 :34) -=Nico=- : ds le syntaxique, il y a une fonction qui gère les appels de fonctions :
(17 :02 :20) -=Nico=- : int base=0 ; // var globale ds syntaxique
void appel_fonction ()
{
base ++ ;
/* Exécution des insts en prenant en compte le base ++ du dessus */
/* ds les insts on pourra avoir un appel à appel_fonction() qui fera un base ++ */
/* et la fonction var et var2 se servent de la base courante pr faire l'adressage */
base −; // fni d'appel de fnction, onrepart comme y fau ds les bases
(17 :02 :21) -=Nico=- : }
(17 :03 :45) -=Nico=- : bah nan, base c même po gérable en C !
(17 :04 :52) -=Nico=- : car si on a
for i :=4 to 20 do
function_toto ()
ds le syntaxique on passe qu'une seule fois ds l'appel function_toto()
(17 :05 :38) -=Nico=- : dc c l'interpréteur qui doit incrémenter base dès qu'il y a un appel à INT (== appel de fonction)
(17 :05 :41) =Zeitoun= : c pas un pb, on y passe et on FINI la fonction la
(17 :05 :59) =Zeitoun= : oui. ptete , je sais plus
(17 :06 :02) -=Nico=- : oé, ms function pourrai etre récursive
(17 :06 :04) =Zeitoun= : jen sais plus rien
(17 :06 :18) -=Nico=- : function_toto pourrai etre récursive !
(17 :06 :28) =Zeitoun= : bie fait pour sa guelle alors
(17 :06 :45) -=Nico=- : dc c le int qui fait un base ++, et ce, dans l'interpréteur !
(17 :07 :41) -=Nico=- : vu qu'on é en dynamique, le prog C ne peu faire que du statique !
(17 :08 :45) =Zeitoun= : moi javai mis ca dans l'instruction CAL je crois
(17 :08 :59) -=Nico=- : oé, ou ds cal, c pareil
(17 :09 :12) -=Nico=- : le tt ce que ça soit fait ds une fonction qui gère les fonctions !
(17 :10 :04) =Zeitoun= : dans le RET je viré toutes les cases réservée par INT avec un retour du pointeur a l'adresse de la base_courante.
pi on vire la base
(17 :10 :28) -=Nico=- : oui
(17 :10 :39) =Zeitoun= : et le LDL récupéré les adresses en fonction de l'adresse et de la base_courante
(17 :11 :20) -=Nico=- : oép
(17 :11 :26) =Zeitoun= : enfin, normalement, c pas forcément de la base courante.
c'est de la base de la variable
(17 :11 :58) =Zeitoun= : putin, heuresement que yavé mes 3 CAL RET et LDL, sinon je croyé plus aux bases
(17 :12 :09) =Zeitoun= : t'as faili mavoir sur ce coup la
(17 :12 :47) =Zeitoun= : (16 :56 :45) -=Nico=- : (16 :00 :36) -=Nico=- : rha putain, ça me

les casse les localités, ttes les fonctions de hash changent un peu !
(16 :56 :59) moi : putin, ca fait 1h
(16 :56 :59) -=Nico=- : lol, une heure plus tard, on é enfin d'accord :)
(17 :12 :50) =Zeitoun= : Bin non en fait :)
(17 :13 :04) -=Nico=- : lol
(17 :13 :22) -=Nico=- : tu veu qu'on prolonge un peu la conversation ?
(17 :13 :40) =Zeitoun= : as u want. moi jsuis casi revenu au point de départ
(17 :13 :52) -=Nico=- : qd on é en base X, ; on peu soi utiliser la variable locale de la fonction, soi la var globale
(17 :13 :59) -=Nico=- : commen qu'on accès à la var globale ?
(17 :14 :04) =Zeitoun= : jveu juste un stokage de mes adreses en relatif, et de la base
(17 :15 :01) -=Nico=- : jveu juste un stokage de mes adreses en relatif, -> juske là ok
et de la base -> peu pas,à la compilation on peu pa connaitre la base d'éxécution courante !
(17 :16 :00) =Zeitoun= : ra, merde
(17 :16 :08) =Zeitoun= : atten, réplachissement
(17 :16 :44) -=Nico=- : (bah oé, vu que la base est gérée ds l'interpréteur maintenant !)
(17 :18 :17) =Zeitoun= : wé, c bon. jai grillé le truc.
(17 :18 :28) =Zeitoun= : ca va mieux.
(17 :19 :08) =Zeitoun= : mais pas core ca. erde
(17 :20 :10) =Zeitoun= : on peut pas seulement dire si une var est globale ou pas par boolean : car si on a un A en globale, et un A dans une proc X, il faut que dans la proc Y il aile herché le globale, et pas le locale de la X
(17 :21 :09) -=Nico=- : bah ça c bon
(17 :21 :27) =Zeitoun= : hé ! ya moyen avec lappel de fonction de gardé en memoire son nom (enfin, num, pointeur..)
heuu, attan, c pas ce ke ta fait ?
(17 :21 :42) -=Nico=- : mon pointeur localite ?
(17 :21 :53) =Zeitoun= : ptete
(17 :22 :10) =Zeitoun= : opu pouvoir avoir les bonne locales, et pas les locales des autres ?
(17 :22 :36) -=Nico=- : pr chaq var le pointeur localite est NULL si globale, ou pointe vers la le nom de la fonction
(17 :23 :16) -=Nico=- : ds le prog on peu avoir 3 var A :
A global (localite = NULL)
A de la fonction X (lcalite pointe vers X)
A de la fonction Y (lcalite pointe vers Y)
(17 :24 :16) =Zeitoun= : wsé, donc, si on chope un A, on sais quoi prendre. OKI
(17 :25 :09) =Zeitoun= : donc tu me done le bon relatif de A ET un booeen qui me dit si c'est local ou global.
avec ca je rajoute la base_courante ou non
(17 :25 :40) -=Nico=- : bah comme ds tout programme
si on a une var glbale A
si ds une fonction X on a une var A, on utilise la var locale
si ds une fonction Y on a pa de var A, on utilise la var globale
(17 :25 :56) =Zeitoun= : wéwé
(17 :26 :14) -=Nico=- : bah oui, ça je te l'ai dit depuis un bail !
(17 :26 :33) =Zeitoun= : mais bon, faut qd meme que je puisse savoir si la var et en globle ou locale.
(17 :26 :36) -=Nico=- : (16 :34 :22) -=Nico=- : (bon en gro, moi ce que je stocke : la taille des paramètres ds le champ du noeud correspondant à l'identificateur de la fonction, et ds

le champ adresse de chacune des variables locales à cette fonction, je stocke la position par rapport à zéro ; si var a,b ;integer ; adresse de a = 0, adresse de b=1) (17 :26 :44) -=Nico=- : et aussi :
(17 :26 :50) =Zeitoun= : wé, ca c bon, javai conpris
(17 :27 :32) -=Nico=- : global si localite = NULL
local à une fonction si lovalite != NULL

(17 :29 :10) =Zeitoun= : wé, mais apres ya toute lhistoire des var() et var2(). ca sera la dedan de récup ce pointeur...
(17 :29 :15) =Zeitoun= : je pense
(17 :29 :32) -=Nico=- : juste ds var() en fait !
(17 :29 :37) =Zeitoun= : putin, jvai faire un log de la convers et le mettre dans le dossier moi
(17 :29 :49) -=Nico=- : MDR, chiche ?
(17 :30 :02) =Zeitoun= : en ANNEXE
(17 :30 :13) -=Nico=- : oé, faut carrément trop le faire !
(17 :30 :17) -=Nico=- : jle fait tt de suite :)