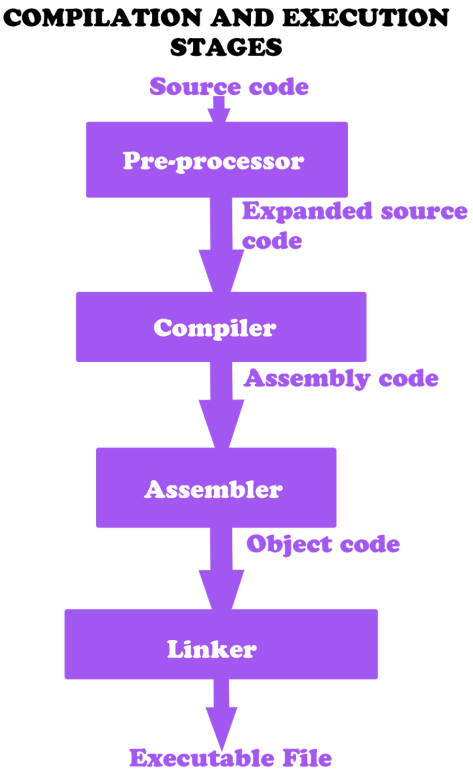
HARIAN Elyoth
CALVINO Enzo

EISE 4

Rapport du projet Compilateur Minic



Cours dispensé par Mr. Quentin MEUNIER.

Table des Matières

I. Lexicographie du langage

1. Le langage Lex

Notre fichier Lex (lexico.l) comporte 3 parties.
Une première partie qui contient toutes nos déclarations, c'est à dire tous nos includes et nos variables globales.
Dans la partie suivante, nous parlerons de la reconnaissance des Lexèmes.
Enfin, la dernière partie qui contient le main de toute nos architecture, c'est dans ce main que nous effectuerons la parse de notre code, l'analyse syntaxique, l'analyse sémantique (la passe de vérification) et enfin la génération du code assembleur.

2. La reconnaissance de Lexèmes (TOKEN)

Pour effectuer la reconnaissance des Lexèmes (TOKEN), c'est à dire des différents mots-clés et caractères réservés à notre langage, il a d'abord fallu définir certains Identificateurs pour faciliter l'analyse lexical de notre code. Nous avons donc définit les identificateurs suivants :

LETTRE ; CHIFFRE ; CHIFFRE_NON_NUL ; ENTIER_DEC ; LETTER_HEXA ; ENTIER_HEX ; IDF ; ENTIER ; CHAINE_CAR ; CHAINE ; COMMENTAIRE ; SPACE ; SAUT_DE_LIGNE

Tous ces identificateurs ont donc été associés à leur représentation dans Lex comme par exemple LETTRE : [a-zA-Z] qui représente toutes les lettres de l'alphabet minuscule et majuscule.
Nous avon ensuite définis nos mots et caractères réservés du langage comme par exemple : int , bool, main, void , if , +, ==, /, ...

Une fois ces identificateurs définis, nous avons donc écrit nos règles lexicales, c'est à dire les actions à réaliser lorsque l'analyse lexicale détecte l'un de ces mots réservés dans le code source. Ainsi, pour chaque caractère réservé ou mot-clés nous avons renvoyé le Lexème associé comme par exemple pour le mot-clés int, on a retourné le lexème TOK_INT etc.

Nous avons également réalisé des actions particulières pour les identificateurs suivants :

- IDF : on récupère dans une chaine de caractère l'identifiant puis on renvoie le TOK_IDENT

- ENTIER : on récupère dans une variable la valeur de l'entier puis on renvoie le TOK_INTVAL

- CHAÎNE : on récupère dans une variable la chaîne de caractère puis on renvoie le TOK_STRING

- COMMENTAIRE : On ne réalise aucune action

- SAUT_DE_LIGNE | SPACE : On ne réalise aucune action

II. Analyse syntaxique et construction de l'arbre du programme

1. Le langage Yacc

Notre fichier Yacc (grammar.y) comporte 3 parties. Une première partie qui contient toutes nos spécifications, c'est à dire, tous nos includes et nos variables globales, mais aussi le prototype des fonctions que nous définissons dans la 3eme partie. Cette première partie comporte également la déclaration de nos terminaux (lexèmes avec %token), la structure de ces derniers (%union) et l'associativité et la priorité des opérateurs (%left, %right etc).

La seconde partie contient toutes nos règles de grammaires, c'est à dire les actions que nous allons réaliser lors de la reconnaissance d'une suite de non terminaux et de terminaux que nous aurons spécifié dans la règle.

Enfin, la 3eme et dernière partie contient tout le code C que nous voulons y implémenter comme nos fonctions dont le prototype a été écrit en première partie. Ce code sera recopié à l'identique dans le fichier .c généré par la compilation de notre fichier Yacc.

2. L'implémentation des règles de grammaire

Pour l'implémentation des règles de grammaire, nous nous sommes référés à la partie sur les règles syntaxique du MiniC du sujet du projet.
Puis nous avons fait les liens avec les règles de grammaire d'arbre afin de construire notre arbre du programme. En recoupant ces informations et en faisant le lien entre arbre du programme et arbre de dérivation, nous avons pu déterminer la nature du node qu'il fallait créer selon la règle dans laquelle nous étions. Pour créer ce node, nous avons utilisé la syntaxe de Yacc avec le caractère réservé $$ qui représente le non-terminal de la règle. Et nous avons attribué ce non-terminal à un node à l'aide de la fonction make_node que nous détaillerons juste après.

Par exemple, pour la règle qui crée un NODE IDENT dans l'arbre du programme, nous écrivons ceci :

```
ident:
    TOK_IDENT
    {
       $$ = make_node(NODE_IDENT, 1, yylval.strval);
    }
    ;
```

Le $$ représente ici la notion de la règle qui est ident et la dépendance de la règle ici est un non-terminale (le lexème TOK_IDENT).

Un autre exemple, pour la règle qui définit les paramètres d'un print s'écrit comme ceci :

```
listparamprint:
    listparamprint TOK_COMMA paramprint
    {
       $$   = make_node(NODE_LIST, 2 , $1, $3);
    }
    |
    paramprint
    {
       $$ = $1;
    }
    ;
```

Ici, nous pouvons voir que dans le cas où le print contient plusieurs paramètre, nous créons un NODE_LIST en donnant notamment à notre fonction les paramètres $1 et $3 qui représentent les dépendances non-terminales de la règle ici listparamprint et paramprint. Et dans le cas ou, le print ne comporte qu'un paramètre, nous ne créons aucun noeud et nous nous redirigeons seulement vers la règle paramprint.

3. La fonction make_node

La fonction make_node possède le prototype suivant :

node_t make_node(node_nature nature, int nops, ...);

Il s'agit d'une fonction à liste d'arguments variables, ce qui signifie que le nombre et le type d'arguments que l'on donne à cette fonction peut changer d'un appel à l'autre. Le but de cette fonction est de créer une structure de type node_t qui est un pointeur sur une autre structure node_s et de la renvoyer. Ces structures qui sont centrales dans notre architecture de code et qui vont nous servir tout au long de la réalisation de notre compilateur sont définies comme ceci :

```
typedef struct _node_s {
   node_nature nature;
   node_type type;
   int64_t value;
   int32_t offset;
   bool global_decl;
   bool is_ini;
   int32_t lineno;
   int32_t stack_size;
   int32_t nops;
   struct _node_s ** opr;
   struct _node_s * decl_node;
   char * ident;
   char * str;
   int32_t node_num;
} node_s;

typedef node_s * node_t;
```

Cette structure comporte beaucoup d'attributs dont la plupart ont un nom explicite et sont expliqués dans le sujet du projet. Nous avons simplement ajouté un attribut à la structure initiale qui est le *bool is_ini;*

Il s'agit d'un booléen qui nous permet de savoir si une variable locale est initialisée ou pas.
Pour en revenir à notre fonction make_node, nous avons réalisé son rôle en une seule fonction. Elle commence par allouer de la mémoire pour la structure node puis elle réalise des allocation mémoires pour les nodes enfants de ce node.
Ensuite nous faisons un switch sur la nature du node qui est créé et on initialise certains attributs de ce node en construction selon sa nature (INTVAL, BOOLVAL, STRINGVAL ect). Ces attributs sont transmis à la fonction à l'aide des paramètres variables ce qui nous permet une plus grande liberté dans l'appel de cette fonction.

III. Analyse des arguments de la ligne de commande

1. La Fonction set_trace_level

Cette fonction sert uniquement pour l'affichage d'une trace lors de l'exécution du programme. Elle comporte un switch sur la valeur du niveau de trace demandé entre 0 et 4, 0 représente le niveau le plus faible, c'est à dire aucun affichage et 4 le niveau de trace maximum. Cette fonction permet ainsi de gérer les variables de debug de Lex et Yacc pour pouvoir afficher les lexème détectés et les règles acceptées lors de l'analyse lexicale et syntaxique de notre code source.

2. La Fonction parse_arg

La fonction parse_arg effectue l'analyse des arguments que l'on donne sur la ligne de commande lorsque nous faisons appelle au compilateur. Elle permet d'analyser et de gérer les options à l'aide de la bibliothèque *getopt* de Linux. Ainsi, nous pouvons alors choisir le nom du fichier de sortie dans lequel sera généré le code assembleur. Sélectionner le niveau de trace lors de l'exécution à des fin de debug notamment. Afficher des informations sur le compilateur et ses options etc. Pour réaliser tout cela, on introduit un certains nombre de variables globales qui seront partagées par plusieurs fichiers pour pouvoir appliquer les demandes effectuées sur la ligne de commande.

IV. Implémentation du module des contextes

1. La structure des contextes.

La structure context_s contient un noeud root qui va représenter la racine de l'arbre de nos déclarations dans le context actuel.

```
typedef struct _context_s {
        noeud_t root;
} context_s;

typedef context_s * context_t;
```

La structure noeud_s est composée d'un tableau de pointeurs sur noeuds. Ce tableau contient 63 pointeurs représentants les 63 prochaines lettres de nos identificateurs.

insérer schéma

la structure noeud_s est alors composée d'un attribut idf_existant. Cet attribut permet de savoir si la ramification allant jusqu'au noeud actuel présente un identificateur, ainsi que des attributs *char lettre* représentant la lettre associée. D'un pointeur sur void *void * data* correspondant à la valeur de la variable, ainsi que d'un *node_t node*, qui est un pointeur vers le noeud ident fait à la déclaration.

2. Fonctions d'implémentations

a. create_context

La fonction create_context a pour but d'allouer en mémoire un context et implémente tous ses attributs. Elle a pour prototype *context_t create_context();*
Ainsi on alloue le noeud root de base en lui mettant la lettre '/' et son attribut idf_existant à true. Le principal concept est ici de partir du noeud root pour stocker tous les identifiants du context actuel, on fait donc pointer tous les 63 pointeurs de suite_idf à NULL, en effet les pointeurs non initialisés sont indéterminés mais ne valent pas null, cette méthode nous permet ainsi de pouvoir éviter des erreurs de segmentation.

b. free_context

Dans la même logique que create_context, free_context a pour but de désallouer en mémoire un context en désallouant de même tous ses attributs.
Son prototype est *void free_context(context_t context);*
pour ce faire nous appelons la fonction free_noeud explicité dans la prochaine section.

c. free_noeud

Le free_noeud est une fonction qui agit récursivement afin de pouvoir désallouer tous les noeuds fils alloués. Son prototype est *void free_noeud(noeud_t noeud);*

Le principe est simple, si l'attribut data du noeud est alloué, on le désalloue et pour les 63 éléments de suite_idf, si un pointeur fils est alloué on appelle free_noeud sur le pointeur alloué. Puis en remontant dans la récursivité, on free le noeud.

d. idf_in_context

La fonction idf_in_context vérifie sur un idf est présent dans un contexte.
Son prototype est bool *idf_in_context(context_t context, char * idf);*
Cette fonction renvoie true si l'identifier se trouve dans le context et false sinon.
Cette fonction nous est utile dans de nombreux cas que l'on verra prochainement. Voila son fonctionnement :
On définit un noeud_actuel qui part du noeud_root. ainsi qu'une chaîne. Pour chaque lettre de l'identifier, on vérifie premièrement si la case correspondante à la lettre est allouée en effet si au bout d'une certaine lettre celle-ci n'est pas allouée on conclut que l'identité n'est pas dans le context actuel. Nous passons par la suite au noeud fils suivant. Si nous arrivons au bout de l'identifier, à la dernière lettre, nous retournons la valeur de idf_existant.

e. context_add_element

La fonction context_add_element ajoute l'association entre le nom idf et le noeud data dans le contexte context. Si le nom idf est déjà présent, l'ajout échoue et la fonction retourne false Sinon elle retourne true.

Son prototype est :
*bool context_add_element(context_t context, node_t node, char * idf, void * data);*

Le principe de fonctionnement est assez simple, en premier lieu, on appelle la fonction idf_in_context afin de pouvoir nous assurer que l'élément n'est pas déjà dans le contexte. Puis nous allons pour chaque lettre, vérifier si le pointeur de la lettre est déjà allouée, c'est à dire si il y a déjà un identifier qui possède les mêmes premières lettres, sinon allouée la lettre et passer à la lettre/noeud suivant. Arrivé à la dernière lettre, nous allons allouée la data, lui attribuer le node correspondant à l'identifier (au moment de la déclaration) et mettre l'idf_existant à true.

f. get_data

La fonction get_data renvoie le la data du nom idf dans le contexte context. elle a pour prototype :
*void * get_data(context_t context, char * idf);*
Elle va donc prendre le contexte actuel et parcourir l'arbre du context correspondant à notre identifiant. Une fois sur la dernière lettre, elle renvoie l'attribut data du noeud.

g. get_node

La fonction get_node est une copie de la fonction get_data, elle renvoie le node de l'identifier idf dans le contexte context. elle a les mêmes paramètres que la fonction get_data précédemment explicitée.

V. Implémentation du module des environnements

1. La structure des environnements

a. Les variables globales

L'implémentation du module des environnements nécessite de nombreuses variables globales. Premièrement, nous utilisons la variable *flag_global* (qui nous servira aussi dans le fichier pass1.c décrit par la suite), cette variable nous permet de nous situer dans l'arbre du programme, en effet les variables globales ont un traitement spécifique notamment l'initialisation à 0 si jamais elle ne sont pas initialisées manuellement.

Nous avons également la variable *int32_t global_offset* qui définit l'offset global du programme. Elle permet d'initialiser les offsets des variables globales et d'instancier les offsets des environnements afin de déterminer les offsets locaux.
La variable *int32_t global_strings_number*, elle, détermine le nombre de chaînes de caractères que contient notre programme. Cette variable est utilisée dans la passe 2 pour pouvoir mettre les chaînes de caractères dans le segment .data du code assembleur.
La variable *char ** global_string* est un tableau de string contenant toutes les chaînes de caractères du programme.

b. La structures env_s

La structure env_s contient un context environnant ainsi qu'un pointeur vers l'environnement supérieur. Elle contient aussi un offset permettant alors de déterminer les offsets des variables environnantes avec la relation offset = global_offset - env_offset.

2. Fonctions d'Implémentations

a. push_global_context

La fonction *push_global_context* a pour objectif de créer le contexte du programme, c'est à dire le contexte global de notre programme. Dans cette fonction, nous créons un contexte grâce à la fonction *create_context* et nous allouons notre variable globale *env_actuel*. Enfin nous attribuons le context créer dans l'attribut contexte de notre variable global et faisons pointer l'attribut next à NULL car c'est notre plus haut context dans la hiérarchie de notre programme.

b. push_context

La fonction *push_context* est appelée à chaque nouveau bloc, elle crée un nouveau context et le fait pointer vers l'environnement supérieur, celui qui était précédemment pointé par la variable *env_actuel*. A la fin de l'allocation, *env_actuel* pointe alors vers ce nouvel environnement.

c. pop_context

La fonction *pop_context* s'appelle a chaque fin de block, elle désalloue le context actuel, et fait pointer la variable env_actuel vers l'environnement supérieur, celui pointer dans l'attribut next de l'ancien contexte.

d. get_decl_node

La fonction *get_decl_node* est la fonction permettant d'implémenter facilement le champ node de la structure noeud_s. Cette fonction va ainsi appeler la fonction qui va vérifier dans l'environnement courant que l'identifiant est dans le contexte actuel, si oui elle va retourner la fonction get_node de ce contexte, sinon nous allons retester la condition sur l'environnement englobant.

e. env_add_element

La fonction *env_add_element* ajoute un élément dans l'environnement actuel. Son prototype est :
*int32_t env_add_element(char * ident, void * node, int32_t size);*

Cette fonction est appelée uniquement avec un node de nature NODE_DECL. Le principe est assez simple, si la variable est globale, on incrémente l'offset des variables globales.
Puis, on test si il s'agit d'une déclaration avec initialisation ou sans. Si il n'y a pas d'initialisation et que nous sommes dans un cas global nous mettons 0 dans l'attribut *value*. Si il n'y a pas d'initialisation et que nous sommes en local, nous mettons une valeur aléatoire. Dans le cas ou la valeur de l'initialisation est une valeur négative, nous initialisons l'attribut *value* par la valeur entière récupérée dans le NODE_INTVAL et nous la multiplions par -1.
A la fin de ces traitements, nous appelons *context_add_element* pour ajouter notre variable au context actuel. Et l'offset de cette variable locale vaut alors l'offset global du programme soustraite par l'offset de l'environnement actuel.

f. reset_env_current_offset

La fonction reset *env_current_offset* met l'offset de la variable global env_offset à 0.

g. get_env_current_offset

La fonction *get_env_current_offset* renvoie l'offset global.

h. add_string

La fonction add_string, prend une chaîne de caractère et l'ajoute dans le tableau global *global_string*. De plus elle redéfinie l'offset global en fonction du nombre de caractère de la chaîne ajoutée.

i. get_global_string_number

La fonction *get_global_string_number* renvoi le nombre de chaînes de caractères contenues dans le tableau global global_string.

j. get_global_string

Cette fonction renvoi le tableau global contenant les chaînes de caractères de notre programme.

k. free_global_string

La fonction *free_global_string* désalloue le tableau global contenant les chaînes de caractères du programme.

VI. Première passe et vérifications contextuelles

1. Parcours de l'arbre du programme et vérification

Une fois l'arbre du programme généré par les modules de Lex et Yacc, il nous faut parcourir cet arbre pour implémenter certains attributs de noeuds, mais aussi vérifier si, notre code respecte certaines règles sémantiques propre au code Minic. Par exemple, la première passe doit vérifier si nous voulons accéder à une variable non déclarée, ou encore si nous faisons des opérations sur des types différents etc.
La première passe est donc essentielle et est la passe la plus importante à implémenter.

2. La fonction passe 1

La fonction *passe_1* à pour principe comme dit précédemment de parcourir l'arbre du programme implémenté plus tôt. Cette fonction est une fonction récursive ayant pour prototype :
void passe_1(node_t root);
La fonction est donc composée de plusieurs parties. D'abord, les traitements sur les noeuds lors de la descente de l'arbre puis l'appel à la fonction passe_1 sur les noeuds suivants et enfin les traitements sur les noeuds lors de la remontée de l'arbre.

a. Les opérations de descentes de l'arbre

Durant tout le parcours de l'arbre, les traitements vont varier selon la nature des noeuds. Nous commençons donc par une condition *switch(root->nature)*.

- Dans le cas du NODE_PROGRAM :

Il nous faut instancier le context global de notre programme en faisant appel à la fonction push_global_context.

- Dans le cas du NODE_IDENT :

Plusieurs cas de figure sont envisageables, il nous faut donc tous les traiter.

Dans un premier cas, nous somme dans le premier fils du *NODE_PROGRAM* et nous avons donc affaire à des déclarations de variable globales.
Pour faciliter le traitement nous gérons toutes les déclarations dans le *NODE_DECL*, pour reconnaître la déclaration, nous instancions une variable globales *flag_decl* à *false*.

En effet si nous nous situons sur un *NODE_IDENT* c'est que nous avons déjà effectué le traitement sur le *NODE_DECL* situé au dessus de lui.
Dans ce cas nous sommes dans une utilisation de la variable, nous vérifions donc si nous ne somme pas dans l'identificateur du main, puis nous instancions l'attribut *decl_node* avec la fonction *get_decl_node* sur l'attribut *ident* de notre noeud. Si il ne le trouve pas, cela signifie que la variable n'a pas été déclarée précédemment donc nous affichons un message d'erreur. Sinon tout c'est bien passé, nous attribuons le type de notre variable à celui de l'attribut *decl_node*.

- Dans le cas du NODE_DECLS :

Nous vérifions simplement si il n'y a pas de déclaration de type void.

- Dans le cas du NODE_DECL :

Nous mettons le *flag_decl* décrits précédemment à true et nous ajoutons la variable dans le contexte courant en instanciant son offset avec la fonction *env_add_element*.

- Dans le cas du NODE_TYPE :

Nous attribuons la variable global *type_actuel* à l'attribut type du noeud.

- Dans le cas du NODE_BLOCK :

Nous faisons la fonction *push_context*, En effet n'importe quel bloc produit la création d'un nouveau contexte et donc d'un nouvel environnement qui devient notre environnement courant.

- Dans le cas du NODE_FUNC :

Pour la passe 2, nous allons *reset_temporary_max_offset* et nous allons mettre le *flag_global* à *false* pour signaler que nous ne somme plus dans la partie "déclaration de variable globale" de notre arbre.

- Dans le cas des NODE_INTVAL / BOOLVAL :

Nous attribuons le type du noeud actuel à *TYPE_INT* ou *TYPE_BOOL.*

- Dans le cas des NODE_STRINGVAL :

Nous attribuons l'offset et ajoutons la chaîne de caractères dans *global_string* avec la fonction *add_string.* Et nous ajoutons également le TYPE_STRING pour l'attribut *type* de notre NODE_STRINGVAL.

- Dans le cas d'un opérateur binaire (NODE_UMINUS / NODE_BNOT / NODE_NOT) :

Nous testons si les deux opérateurs sont bien du même type et testons de même la division et le modulo zéro de manière statique.

b. L'appel de passe_1 sur les noeuds fils de l'arbre.

Ici nous réalisons le parcours en profondeur de l'arbre en faisant un appel récursif sur les noeuds fils de notre noeud actuel en testant préalablement leur nombre de fils.

c. Les opérations sur la remontée de l'arbre.

Certaines opérations doivent être effectuées en remontant l'arbre du programme :

- Dans le cas du NODE_UMINUS :

On vérifie que le type de l'opérande est bien de type *TYPE_INT*.

- Dans le cas des NODE_FUNC :

On initialise l'attribut *stack_size* en faisant en lui attribuant la valeur de *(global_offset - env_actuel->env_offset)* du noeud.

- Au niveau des NODE_IF / FOR / WHILE / DOWHILE :

Nous testons si la valeure de retour des conditions de ces boucles est bien de type booléens.

- Dans le cas des NODE_BLOCK :

Pour la remontée nous utilisons la fonction *pop_context* afin de libérer l'environnement actuel et réactualiser l'environnement courant.

- Dans le cas du NODE_PROGRAM :

Nous refaisons un pop_context pour supprimer le context du programme.

VII. Deuxième passe et génération du code assembleur.

La deuxième passe a pour but de générer un code assembleur à partir de l'arbre du programme après la passe de vérification. Ainsi, comme dans la passe 1, nous allons faire un parcours en profondeur de notre arbre actualisé.

1. Les traitements descendants

- Dans le cas du NODE_PROGRAM :

Nous créons le segment *.data* afin de pouvoir par la suite mettre en mémoire nos variables globales.

- Dans le cas du NODE_FUNC :

Nous avons finis d'écrire les variables globales dans le *.data* nous passons donc à l'écriture des .asciiz dans notre *.data*.
Puis nous écrivons le segment *.text* et commençons à allouer la place en mémoire dans le registre 29 en récupérant l'attribut stack_size du noeud.

- Dans le cas du NODE_DECL :

Si nous sommes dans une déclaration d'une variable globale, nous regardons si la variable est initialisée à la déclaration. Si oui, nous créons un .word avec la valeur du littérale associé. Si non, nous initialisons la variable à 0.
Dans le cas d'une déclaration d'une variable locale, nous regardons si la variable est initialisée à la déclaration. Si elle ne l'est pas, nous ne faisons rien (la valeure est indéterminée). Si elle est initialisée, nous regardons comment. Si elle est initialisé avec un littéral, nous stockons en pile la valeur directement.

Si c'est avec une autre variable, on récupère la valeure de cet autre variable dans un registre et nous la stockons en pile. Enfin, si c'est avec une expression, nous effectuons le calcul de l'expression en utilisant le moins de registre possible puis nous récupérons le résultat dans un registre et nous le stockons en pile.

- Dans le cas du NODE_AFFECT :

Nous déterminons au préalable si, après la passe 1, cette affectation demandera plus de registre que disponible. Dans ce cas précis nous effectuons les traitement nécessaire pop_temporary.

- Dans le cas des NODE_WHILE / FOR / IF / DOWHILE :

Nous effectuons les traitements des conditions, c'est à dire les traitements relatifs aux opérateurs. puis selon la nature des noeuds, nous allons écrire différentes fonctions assembleurs.

Exemple :

```
WHILE :
loop :
        sltu $condition $operand1 $operand2        #On test la condition
        bne $condition $0 fin      #Si celle-ci est respectée on saute à la fin de la boucle
        ...
        instructions du block
        ...
        j loop
fin :
```

2. Les traitements en remontant

Nous aurons un unique traitement ici, qui sera celui du NODE_FUNC. Ce traitement aura pour perspective de terminer le fichier assembleur. Il se compose alors du code suivant :

```
ori $2 $0 10
syscall
```