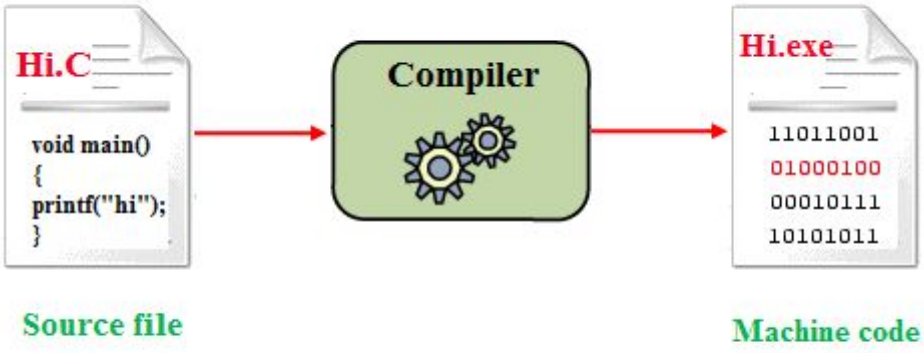
POLYTECH

SORBONNE UNIVERSITÉ
CRÉATEURS DE FUTURS DEPUIS 1257

Rapport du projet Compilation



Lisa Hennebelle et Diego Moreno Villanueva

Professeur encadrant: Quentin Meunier

Année 2019/2020

SOMMAIRE

Introduction

Le but de ce projet est de coder un compilateur d'une version du langage C réduite : le MiniC. Ce langage se décompose en plusieurs règles et une grammaire associées. Le compilateur va devoir interpréter le fichier qui lui est donné en argument donc passer par des étapes de vérification lexicale, syntaxique et sémantique avant de générer le code assembleur associé.

Pour ce faire, on a eu recours à des outils qui nous permettent de générer des analyseurs lexicaux et syntaxique à partir du langage associé à notre compilateur: lex pour l'analyse lexicale et yacc pour l'analyse syntaxique. En spécifiant les caractères qui constituent le langage dans lex et les règles de grammaire dans yacc, on peut analyser simplement un fichier MiniC. Il va comme cela créer un arbre du programme pendant l'analyse syntaxique puis le modifier pendant l'analyse sémantique (aussi appelée passe 1). Ensuite, durant la passe 2, il génère le code assembleur associé à cet arbre.

Notre compilateur doit aussi pouvoir prendre plusieurs options en arguments, pour pouvoir afficher ces mêmes options, arrêter le compilateur à une certaine étape de vérification (syntaxique ou sémantique) et autres. Ces options seront développées dans la partie II concernant les modules.

Il nous a aussi fallu manipuler les contextes et environnements liés au différentes parties du programme.

I/ Implémentation de lex et yacc

I.a) LEX

Le premier pas dans l'analyse lexicale est de définir les caractères et lexèmes de notre langage. On distingue donc les lettres, les chiffres et l'ensemble des caractères imprimables. A partir de ces trois distinctions, on est capables de décrire la forme qui auront les nombres entiers, les noms identifiants, les chaînes de caractères et les commentaires.

```
LETTRE          [a-zA-Z]
CHIFFRE         [0-9]
ASCII           [\040-\176]
ENTIER          ("-")?[1-9]{CHIFFRE}*
IDF             {LETTRE}({LETTRE}|{CHIFFRE}|'_')*
CHAINE_CAR      ({ASCII})+
CHAINE          '"'({CHAINE_CAR}|'\"'|'\n')*'"'
COMMENTAIRE     "//"({ASCII})*[\n]
```

Après, il est nécessaire de générer des valeurs qui seront reconnues par le fichier yacc pour vérifier les règles de grammaire et construire l'arbre : ce sont les tokens. Les tokens sont définis dans le fichier yacc mais sont généré par lex. En fonction des caractères ou sous ensembles de caractères reconnus par lex, un token correspondant va être généré.

De la même manière, et afin de communiquer avec le fichier yacc, une variable externe nommée `yylval` va contenir des informations particulières (valeur d'un identifiant, d'une variable, chaîne de caractères…) dans ses champs intval et strval. Un exemple d'utilisation de ces champs est donné ci après pour un identifiant :

```
{ENTIER} {
          yylval.intval = atoi(yytext);
          printf("Detecté entier %s, TOK_INTVAL généré\n",yytext);
          return TOK_INTVAL;
}
```

Ici, lorsqu'un entier, tel qu'il est défini par la règle précédente, est détecté, on assigne sa valeur à yylval.intval et on retourne un token `TOK_INTVAL` à yacc. La fonction qui renvoie des tokens s'appelle `yylex`. Elle est appelée par `yyparse` pour générer chaque token l'un après l'autre. Ainsi un simple appel à cette fonction dans le main de lex permet de faire la vérification lexicale d'un fichier `infile` passé en argument de notre compilateur :

```
node_t program_root;
yyin = fopen(infile, "r");
yyparse(&program_root);
```

I.b) Yacc

Dans le fichier yacc sont définis les tokens ainsi que leur associativité et priorités. Par exemple, comme dans une expression mathématique telle que A + B * C, le * prévaut sur le +.

L'associativité, quant à elle, se fait toujours par la gauche pour des tokens de même priorité : l'expression A + B + C sera traitée comme (A + B) + C pour éviter les ambiguïtés.

Le fichier yacc permet aussi de définir les règles de grammaire du langage compilé. A gauche de ces règles, se trouvent les non-terminaux comme vus dans le cours, et à droite les terminaux. En même temps que de définir ces règles, nous construisons l'arbre du programme : les noeuds sont créés au fur et à mesure que les token sont analysés par yacc et les règles réduites.

Ces noeuds sont décrits par la structure `node_s` dans un fichier attenant au projet, `defs.h` comme suivant :

```
typedef struct _node_s {
        node_nature nature;
        node_type type;
        int64_t value;
        int32_t offset;
        bool global_decl;
        int32_t lineno;
        int32_t stack_size;
        int32_t nops;
        struct _node_s * opr[100];
        struct _node_s * decl_node;
        char * ident;
        char * str;
        // pour l'affichage du graphe
        int32_t node_num;
} node_s;
```

fonction make_node:

Pour créer l'arbre au fur et à mesure de l'analyse syntaxique, nous avons implémenté la fonction `make_node (node_nature nature, int32_t nops ,...)` qui retourne un `node_t` (pointeur sur `node_s`) pour qu'elle crée un noeud de l'arbre et qu'elle modifie les champs de ce noeud en fonction des arguments de la fonction.

Au niveau des règles, un exemple de la bonne façon d'appeler cette fonction est la suivante :

```
vardecl : type listteypedecl TOK_SEMICOL
        {
                $$ = make_node(nature, nops, $1, $2);
        };
```

Il faut noter que dans cette notation, `$$` correspond à la partie gauche de la règle en question, donc dans ce cas c'est le noeud que l'on veut créer. De la même manière, les arguments `$1`, `$2`, etc., font référence aux arguments de la partie droite de la règle. Il n'est pas nécessaire, comme montré ici, d'utiliser tous les arguments de la règle.

Cette fonction peut accepter un nombre d'arguments variable, de n'importe quel type, en plus de `nature` et `nops`. C'est cette possibilité que nous avons exploitée afin de passer en argument les valeurs des champs spécifiques lorsqu'ils étaient récupérables avant de créer le noeud. On a convenu d'une convention pour passer les arguments spécifiques dans un ordre précis afin de pouvoir les assigner comme il le faut dans la fonction.

Ces arguments supplémentaires sont récupérés par l'appel à la fonction `va_arg(ap, nature_arg)` dans laquelle on spécifie la liste des arguments supplémentaires récupérée dans `ap`, en faisant appel à `va_list ap` , et la nature attendue de cet argument dans `nature_arg`.
Pour commencer à lire les arguments, avant l'appel à `va_arg`, on utilise la fonction `va_start()`.

```
node_t make_node(node_nature nature, int32_t nops, ...) {
        va_list ap;
        va_start(ap,nops);
```

On commence la fonction par l'allocation mémoire du `node_t retour`, puis on récupère les arguments systématiquement présents `nature` et `nops` pour les attribuer aux champs associés de `retour`.
L'argument `nature` est la nature du noeud (`NODE_FUNC`, `NODE_DECL`, etc) et `nops` le nombre d'enfants de ce noeud. Nous avons décidé de toujours passer en argument supplémentaire directement après `nops,` les enfants du noeud, lorsqu'il y en avait. Pour les affecter au champ correspondant de `retour` (`opr`, le tableau contenant les enfants du noeud), nous avons effectué une boucle `for` sur le nombre d'enfants et ajouté chaque argument suivant de make_node à `opr`, jusqu'à ce qu'on est parcouru tous les enfants. Nous avons également alloué de la mémoire pour les autres attributs pointeurs de `node_s`.

Code correspondant :

```
retour = (node_s *) malloc(sizeof(node_s));
retour->nature = nature;
retour->nops = nops;
for (int i = 0; i < nops; i++)
{
	retour->opr[i] = va_arg(ap,node_t);
}
retour->opr = (node_s **)malloc(sizeof(node_s*) * nops);
retour->ident = malloc(sizeof(char)*30);
retour->str = malloc(sizeof(char)*30);
retour->decl_node = NULL;
```

Par la suite, des noeuds de certaines nature doivent recueillir des informations supplémentaires : c'est là que nous avons effectué un switch sur la nature du noeud créé.

```
case NODE_IDENT :
	retour->type   =va_arg(ap,node_type); // argument supp à la position nops + 1 = type de noeud
	retour->decl_node=va_arg(ap,node_t); // argument supp à la position nops + 2 = declaration de noeud
	retour->offset =va_arg(ap,int32_t ); // argument supp à la position nops + 3 = declaration de l'emplacement mémoire de la variable int32_t
	retour->global_decl=(bool)va_arg(ap,int); // argument supp à la position nops + 4 = declaration globale?
	strcpy(retour->ident,va_arg(ap, char*)); // argument supp à la position nops + 5 = identifiant
	break;
```

Un nœud de type `NODE_IDENT` devra récupérer les informations suivantes :

- le `type` de nœud (`TYPE_BOOL`, `TYPE_INT` ou `TYPE_VOID` dans le cas d'un identifiant), qui va être le premier argument supplémentaire de l'appel à `make_node()`,
- la déclaration du noeud `decl_node`, qui va être le deuxième argument supplémentaire de l'appel à `make_node()`,
- l'emplacement mémoire `offset` associé à la déclaration de l'identifiant, qui va être le troisième argument supplémentaire de l'appel à `make_node()`,
- le booléen associé à l'information "est ce que l'identifiant correspond à une variable globale" nommé `global_decl`. Il est ici converti depuis un `int` car `va_arg` n'accepte pas de booléen, qui va être le quatrième argument supplémentaire de l'appel à `make_node()`,

- le nom de l'identifiant , récupéré depuis `yylval.strval`, pour initialiser le champ `ident`, qui va être le cinquième et dernier argument supplémentaire de l'appel à `make_node()`,

  Pour déterminer si un identifiant correspond à une variable globale, nous avons effectué le test suivant dans lex: si jamais l'identifiant 'main' avait été attribué, on assignait à un entier nommé `mainflag`. A partir de ce moment, toutes les variables déclarées sont des variables locales. La règle `ident` fonctionne ainsi:

```
ident : TOK_IDENT
	{
		node_t nident;
		if (mainflag != 1)
		{
			$$ = make_node(NODE_IDENT, 0, typetemp,nident , -1, 1,
yylval.strval);
		}
		else
		{
			if (strcmp (yylval.strval, "main") == 0)
		{
				$$ = make_node(NODE_IDENT, 0, typetemp, nident ,-1, 1,
yylval.strval);
		}
		else
		{
				$$ = make_node(NODE_IDENT, 0, typetemp, nident , -1, 0,
yylval.strval);
		}
		}
	}
```

Le `mainflag` est donc testé avant l'appel à `make_node` dans le cas d'un identifiant :

- s'il vaut 0, le main n'a pas encore été déclaré, c'est une variable globale,
- s'il vaut 1, deux cas se présentent, soit on est dans la déclaration de 'main' auquel cas on considère que c'est encore une variable globale, soit une variable déclarée à l'intérieur du `main`, donc locale.

Il faut aussi déterminer le type associé à l'identifiant: pour cela à chaque appel à la règle `type`, on modifie le `type_node typetemp` qui va être passé en argument de `make_node` pour créer un noeud `NODE_IDENT`.

```
case NODE_AFFECT :
	retour->type  =va_arg(ap,node_type); // argument supp à la position nops + 1 = type
de noeud
	break;
```

Un nœud de type `NODE_AFFECT` correspond à l'attribution d'une valeur à une variable déclarée. Pour cela, on attribue le `node_type` passé en argument au nouveau nœud à créer.

```
case NODE_FUNC :
       retour->offset   =8; // argument supp à la position nops + 1 = declaration de
l'emplacement mémoire de la variable
       retour->stack_size=8; // argument supp à la position nops + 2 = declaration de
       break;
```

Un nœud de type NODE_FUNC correspond à la fonction main du programme. Pour le rappel, MiniC n'accepte pas de fonctions autres que celle-là. Pour la création de ce nœud, il faudra récupérer les informations suivantes :

- `offset`, mentionné précédemment, aura la valeur fixe de 8.
- `stack_size`, la taille en pile à allouer, a une valeur fixe de 8 avant d'être mise à jour lors de la passe 1.

```
case NODE_TYPE:
       retour->type = va_arg(ap,node_type);
       break;
```

Dans le cas d'un `NODE_TYPE`, on doit spécifier son type, passé en argument supplémentaire.

```
case NODE_INTVAL:
       retour->type = TYPE_INT;
       retour->value = va_arg(ap,int64_t);
       break;
```

Dans le cas d'un `NODE_INTVAL`, on doit spécifier en plus de sa nature le `type (TYPE_INT)` du nœud et la valeur entière associée `value`. Cette valeur est spécifiée dans l'argument supplémentaire de la fonction.

```
case NODE_BOOLVAL:
       retour->type = TYPE_BOOL;
       retour->value = va_arg(ap, int);
       break;
```

Dans le cas d'un `NODE_BOOLVAL`, comme pour un `NODE_INTVAL`, on doit spécifier le `type (TYPE_BOOL)` du nœud et la valeur associée dans le champ `value`. Comme cette valeur est de type `int`, on a adopté la convention 0 = false et 1 = true.

```
case NODE_STRINGVAL :
       retour->type = TYPE_STRING;
       strcpy(retour->str, va_arg(ap,char*));
       retour->offset = va_arg(ap,int32_t);
       break;
```

Dans le cas d'un `NODE_STRINGVAL`, comme les deux cas ci-dessus, on doit spécifier le type (`TYPE_STRING`) du nœud et la valeur associée dans le champ `str`. On doit aussi assigner la valeur `offset` qui correspond au type `int32_t` dans les arguments de la fonction `make_node`.

```
case NODE_PLUS :
case NODE_MINUS:
case NODE_MUL:
case NODE_DIV:
case NODE_MOD:
case NODE_LT:
case NODE_GT:
case NODE_LE:
case NODE_GE:
case NODE_EQ:
case NODE_NE:
case NODE_AND:
case NODE_OR:
case NODE_BAND:
case NODE_BOR:
case NODE_BXOR:
case NODE_SRA:
case NODE_SRL:
case NODE_SLL:
case NODE_NOT:
case NODE_BNOT:
case NODE_UMINUS:
       retour->type = TYPE_NONE;
       break;
```

Pour tous les noeuds de types listés ci dessus, nous avons spécifié un type `TYPE_NONE`.

fonction free_tree:

Comme vu précédemment, les noeuds ont des attributs auxquels on a alloué de la mémoire. Le pas suivant est donc de libérer cette espace pour éviter les fuites. Ce sera fait avec la fonction récursive `free_tree` qui prendra en arguments le noeud `root` et un entier qui sera utilisé pour montrer l'arbre du programme au fur et à mesure de la libération de mémoire avec le niveau de trace 4 (expliqué plus tard dans la fonction `parse_args`).

Les attributs à libérer seront seulement ceux dont nous avons alloué de la mémoire, donc le pointeur `node_decl` et d'autres attributs ne sont pas considérés, sinon on aura des erreurs de segmentation. De la même manière, on fait attention à ne pas utiliser `free` pour les noeuds enfants dans `opr`, car ils sont déjà libérés récursivement.

```
void free_tree(node_t node, int tab)
{
       if(nbtraces == 4)
       {
              for(int i = 0; i < tab; i++) printf("    ");
              printf("freeing node %s\n", node_nature2string(node->nature));
              for(int i = 0; i < tab; i++) printf("    ");
              printf("nops = %d\n", node->nops);
              tab++;
       }
       for (int32_t i = 0; i < node->nops; i += 1) {
              free_tree(node->opr[i], tab);
       }
       free(node->opr);
       free(node->ident);
       free(node->str);
       free(node);
}
```

fonction analyse_tree:

Le rôle de cette fonction est diverse : d'une part elle lance l'appel à la passe 1 avec `lancer_parcours(root)`, et elle est chargée aussi de faire l'appel à la passe 2, tout en vérifiant les variables `stop_after_syntax` et `stop_after_verif` qui sont déclarées lors de la compilation de `./minicc` avec `-s -v` (expliqué plus tard dans `parse_args`). Finalement, elle commence la libération de mémoire avec `free_global_strings` et `free_tree`.

```
void analyse_tree(node_t root) {
	if (!stop_after_syntax) {
		// Appeler la passe 1
		lancer_parcours(root);
		if(errorflag)
		{	// si détection d'erreur au cours de la passe 1
			exit(1);
		}
		if(nbtraces == 3)
		{
			char * txtname = "tree.dot";
			dump_tree(root,txtname); // afficher le tree
		}
		if (!stop_after_verif) {
			create_program();
			// Appeler la passe 2

			//dump_mips_program(outfile);
			free_program();
		}
		free_global_strings();
		free_tree(root, 0);
	}
}
```

II/ Implémentation de modules

II.a) Module implémentant l'analyse des arguments de la ligne de commande

Ce module prend la forme d'une fonction, implémentée dans le fichier `common.c` et appelée au tout début du `main` de lex. Elle s'appelle `parse_args` et prend en argument les `argv` et `argc` du `main`. C'est grâce à cette fonction qu'on récupère les options que l'utilisateur a choisies et qu'on affecte la valeur de `infile` si jamais il est passé en argument du compilateur.

fonction parse_args

Pour ce faire, nous nous sommes appuyés sur le fonctionnement de la fonction Unix `getops()` qui permet de dresser une liste des options qu'attend notre compilateur et de tester la valeur des options choisies pour effectuer le comportement associé.

```
void parse_args(int argc, char ** argv) {
	int c = 0;
	int indice;
	char * filename;
	extern char * optarg;
	extern int optind;
	nbtraces = 0;
	while (c != -1)
	{
		c = getopt(argc, argv,"bo:t:r:svh" );
		switch (c)
```

Au début de la fonction, on effectue une boucle while sur les options du compilateur : tant qu'il y a des options, on vérifie la valeur de l'option et on effectue le comportement associé.

La liste des options attendues est spécifiée en troisième argument de getopt : ici il s'agit des options `-b`, `-o`, `-t`, `-r`, `-s`, `-v`, `-h`, dont certaines qui attendent des arguments obligatoirement, ce qui est spécifié par les '`:`' après la lettre.

Chaque implémentation du comportement lié à une option est explicitée ci-dessous:

```
	case 'h':
printf("Appels a ./minicc avec des options :\n"
"• -b : Affiche une bannière indiquant le nom du compilateur et des membres du binôme\n"
"• -o <filename> : Définit le nom du fichier assembleur produit (défaut : out.s).\n"
"• -t <int> : Définit le niveau de trace à utiliser entre 0 et 5 (0 = pas de trace ; 5 =
toutes les traces; defaut = 0).\n"
"• -r <int> : Définit le nombre maximum de registres à utiliser, entre 4 et 8 (défaut :
8).\n"
"• -s : Arrêter la compilation après l'analyse syntaxique (défaut = non).\n"
"• -v : Arrêter la compilation après la passe de vérifications (défaut = non).\n"
"• -h : Afficher la liste des options (fonction d'usage) et arrêter le parsing des
arguments.\n");
exit(0);
```

La première option considérée est `-h`. Cette option sert à imprimer toutes les options que le compilateur peut fournir. Une fois cette option appelée, on n'exécute pas la compilation, d'où l'appel à `exit(0)`.

```
case 'b':
	printf("##################COMPILATEUR##################\n
	Binome: Lisa HENNEBELLE & Diego MORENO VILLANUEVA\n");
	exit(0);
```

La deuxième option considérée est l'option `-b`, qui imprime une bannière contenant le nom du compilateur et les membres de l'équipe. Une fois cette option appelée, on n'exécute pas la compilation, d'où l'appel à `exit(0)`.

```
case 'o':
	filename = optarg;
	printf("nom du fichier : %s\n", filename);
	break;
```

L'option `-o <filename>` sert à spécifier le nom du fichier assembleur généré. Ce nom doit être écrit juste à la suite de l'option pour être validé.

```
case 't':
       nbtraces = atoi(optarg);
       if (nbtraces > 5)
       {
              printf("Le nombre que vous avez indiqué est trop grand, remettez en un
inférieur ou égal à 5\n");
              exit(EXIT_FAILURE);
       }
       break;
```

L'option `-t <int>` définit le niveau de trace demandé à l'appel du compilateur. Ce niveau est compris entre 0 (: aucune trace) et 5 (: toutes les traces)

Niveaux de traces :
-niveau 1 : affichage des tokens générés
-niveau 2 : affichage des noeuds créés
-niveau 3 : affichage de la libération récursive de mémoire
-niveau 4 : affichage de l'arbre dans fichier
-niveau 5 : affichage des opérations sur les contextes lors de la passe 1

Si jamais le numéro entré après "-t" est supérieur à 5, on affiche un message d'erreur et on sort du compilateur en `exit(EXIT_FAILURE)`.

```
case 'r':
       nbregistres = atoi(optarg);
       if (nbregistres < 4 || nbregistres > 8)
       {
              printf("Le nombre que vous avez indiqué est incorrect, remettez en un entre
4 et 8\n");
              exit(EXIT_FAILURE);
       }
       break;
```

L'option `-r <int>` permet de définir le nombre maximum de registres utilisés dans le code assembleur respectif. Ce nombre doit être entre 4 et 8 et la valeur par défaut est 8.

```
case 's':
       sflag = 1;
       if (vflag == 1)
       {
              printf("les options -v et -s ne sont pas compatibles\n");
              exit(EXIT_FAILURE);
       }
       break;
```

L'option `-s` permet d'arrêter la compilation après l'analyse syntaxique. On associe à son appel un flag nommé `sflag` pour spécifier qu'elle a été saisie. Comme elle n'est pas compatible avec l'option `-v`, un flag similaire nommé `vflag` associé à cette option est testé. Si jamais il est égal à 1, les deux options ont été appelées, alors on affiche un message d'erreur et on sort du compilateur en `exit(EXIT_FAILURE)`.

```
case 'v':
       vflag = 1;
       if (sflag == 1)
       {
              printf("les options -v et -s ne sont pas compatibles\n");
              exit(EXIT_FAILURE);
       }
       break;
```

L'option `-v` permet d'arrêter la compilation après la passe de vérifications. Pour la même raison que l'option précédente, on lui associe un flag nommé `vflag`. La vérification pour l'incompatibilité avec l'option `-s` est présentée ici aussi.

```
infile = argv[optind];
```

Après le switch, si jamais on n'est pas sorti du compilateur par l'appel à la fonction exit(), on peut récupérer le nom du fichier `infile` à compiler dans `argv`, à la position `optind` (après les différentes options saisies).

Module de contextes

Le deuxième module que nous avons implémenté est celui de contextes. Il permet de définir une structure `context_t` et des fonctions destinées à la création et à des manipulations de contextes durant la passe 1. Un module similaire est disponible pour créer et manipuler des environnements. Nous avons choisi de manipuler celui-ci pour bien prendre en main les contextes. Nous nous sommes donc penchés sur le fichier context.h dans le dossier de librairies utils. Ce fichier définit la structure `context_t` ou plutôt la structure `context_s`, puisqu'un `context_t` est un pointeur sur `context_s`. Celle-ci contient un attribut `root` de nature `noeud_t`. Un `noeud_t` est un pointeur vers une autre structure `noeud_s` qui possède plusieurs attributs listés ci-dessous :

```
typedef struct _noeud_s {
   bool idf_existant;
   struct _noeud_s * suite_idf[NB_ELEM_ALPHABET];
   char lettre;
   void * data; // node_t corresponding to the variable decl
} noeud_s;
```

Ces modules vont nous permettre de créer des contextes liés à des identifiants avec cette structure de type liste chaînée : à chaque `noeud_s` on va associer une lettre de l'identifiant nommée `lettre`, et un tableau de noeuds suivants `suite_idf` qui contient un noeud pour chaque lettre de l'alphabet. Une fois qu'on a créé tous les noeuds correspondant à chaque lettre de l'identifiant dans l'ordre, on va pouvoir mettre à jour le dernier noeud en changeant son attribut `idf_existant` à true. Ensuite, on associe la première déclaration de l'identifiant dans le code dans le champ `data`. Cela nous permettra d'accéder à d'autres informations utiles pour la passe 1.

Une fois ce fonctionnement assimilé, on a décidé d'implémenter par nous mêmes les fonctions associées listées dans le fichier header :

```
context_t create_context();
int caractere_indice(char lettre);
noeud_t init_noeud_context(int lettre);
bool context_add_element(context_t context, char * idf, void * data);
void * get_data(context_t context, char * idf);
void free_noeud(noeud_t noeud);
void free_context(context_t context);
```

Pour cela, nous avons créé un fichier associé context.c qui inclut context.h.
Voici les différentes implémentations réalisées :

fonction create_context() :

```
context_t create_context() // cree un contexte et lui alloue de la mémoire
{
	context_t cont = malloc(sizeof(context_s));
	if (cont == NULL)
	{
	printf("problème dans la création d'un contexte\n");
	}
	printf("create_context\n");
	cont->root = malloc(sizeof(noeud_s));
	(cont->root)->idf_existant = false;
	for(int i = 0; i < NB_ELEM_ALPHABET; i++)
	{
		cont->root->suite_idf[i] = NULL;
	}
	(cont->root)->lettre = 0;
	(cont->root)->data = NULL;
	return cont;
}
```

Cette fonction sert à créer un contexte en lui allouant de la mémoire, à créer un `noeud_t root` en lui allouant aussi de la mémoire et à initialiser les champs de `root`. La fonction retourne le contexte ainsi initialisé.

fonction init_noeud_context():

```
noeud_t init_noeud_context(int lettre)
{
  noeud_t noeud = malloc(sizeof(noeud_s));
  noeud->idf_existant = false;
  for(int i = 0; i < NB_ELEM_ALPHABET; i++)
  {
	noeud->suite_idf[i] = NULL;
  }
  noeud->lettre = lettre;
  noeud->data = NULL;
  return noeud;
}
```

Cette fonction initialise un noeud en lui allouant de la mémoire et en associant une lettre à son champ `lettre.` Elle est utilisée par la suite pour l'association d'une identifiant avec un contexte.

fonction free_noeud():

```
void free_noeud(noeud_t noeud) //free recursif d'un noeud_t
{
	for(int i = 0; i < NB_ELEM_ALPHABET; i++)
	{
		if(noeud->suite_idf[i])
		{
			free_noeud(noeud->suite_idf[i]);
		}
	}
	noeud->data = NULL;
	free(noeud);
}
```

Cette fonction permet de libérer l'espace alloué à un noeud associé à un contexte en opérant de manière itérative sur tous les noeuds suivants d'un contexte si jamais ils existent.

fonction free_context() :

```
void free_context(context_t context) // liberer la memoire allouée pour le contexte
{
	// parcourir le contexte pour tout liberer
	free_noeud(context->root);
	free(context);
}
```

On utilise dans cette fonction la fonction précédente pour libérer tous les identifiants contenus dans le contexte à partir de `root`. Ensuite, on libère l'espace alloué au contexte lui même.

fonction caractere_indice(char lettre)

```
int caractere_indice(char lettre)
{
    int car;
    if(lettre >= 'a' && lettre <= 'z') {
        car = lettre - 'a';
    }
    else if(lettre >= 'A' && lettre <= 'Z') {
        car = lettre - 'A' + 26;
    }
    else if(lettre >= '0' && lettre <= '9') {
        car = lettre - '0' + 52;
    }
    else if(lettre == '_') {
        car = 62;
    }
    else {
        printf("ERROR dans l'évaluation d'un caractère %d\n", lettre);
    }
    return car;
}
```

Cette fonction est utilisée pour traduire un caractère d'un identifiant (les lettres minuscules ou majuscules, les nombres et le caractère '_') en un nombre qui représente son indice dans le tableau `suite_idf[]`, et imprimera un message d'erreur si ce caractère n'est pas admissible. La fonction sera utilisée dans quelques fonctions de ce fichier.

fonction context_add_element() :

```
bool context_add_element(context_t context, char * idf, void * data)// 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, la
fonction retourne true.
{

        noeud_t actuel= context->root;
        int idf_len = strlen(idf);
        int lettre, let_indice;

        for(int i = 0; i < idf_len; i++)
        {
                lettre = idf[i];
                let_indice = caractere_indice(lettre);

                if(!(actuel->suite_idf[let_indice]))
                {
                        actuel->suite_idf[let_indice] = init_noeud_context(lettre);
        }
                actuel = actuel->suite_idf[let_indice];
        }
        if( actuel->idf_existant == true)
        {
                return false;
        }
        else
        {
                actuel->data = data;
                actuel->idf_existant = true;
                return true;
        }

}
```

Cette fonction permet d'associer le nom `idf` et le noeud `data` à un certain contexte. Ainsi on part de la racine `root` pour parcourir les noeuds correspondants à l'identifiant. Si jamais le dernier noeud parcouru a son attribut `idf_existant` à true, on retourne false. Sinon on fait l'association de data à ce noeud et on met `idf_existant` à true, puis on retourne true. On se servira de cette fonction dans la passe pour détecter des erreurs de redéfinition dans des contextes communs.

fonction get_data() :

```
void * get_data(context_t context, char * idf)//retourne le noeud précédemment associé à
idf dans context, ou null si idf n'existe pas dans context.
{
        noeud_t noeud = context->root;
        int i = 0, idf_len = strlen(idf);
        int car = caractere_indice(idf[i]);

        // parcours de contextes à la chaine
        while(i < idf_len && noeud->suite_idf[car])
        {
                noeud = noeud->suite_idf[car];
                if(idf[++i])
                        car = caractere_indice(idf[i]);
        }
        if(i < idf_len)
        {
                printf("IDF non trouve\n");
                return NULL;
        }
        if (noeud->idf_existant == true)
        {
                printf("IDF trouve !\n");
                return (context->root)->data;
        }
        return NULL;
}
```

Cette fonction permet de récupérer le champ `data` associé à un identifiant `idf` dans un certain contexte. Elle retourne NULL si cet identifiant n'est pas présent ou la valeur du champ `data` associé si il est bien présent. De la même manière que précédemment, cette fonction va opérer de manière de façon itérative sur tous les noeuds depuis `root` qui correspondent aux lettres de l'identifiant.

III/ Script de vérification

Le script de vérification permet de tester le compilateur de manière séquentielle sur plusieurs fichiers sources de test afin de détecter des potentielles erreurs à certains points du compilateur. Il permet aussi de tester la détection d'erreurs sur des fichiers comportant volontairement des fautes.

Les dossiers sources des fichiers de tests sont divisés en 3 grandes catégories : le dossier Syntaxe, le dossier Vérif et le dossier Gencode. Chacun de ces dossiers possède un sous dossier OK et un sous dossier KO. En fonction de son emplacement, on peut prédire le comportement attendu du compilateur:

Syntaxe

- Les fichiers présents dans le sous dossier Syntaxe/KO doivent provoquer une erreur syntaxique, soit durant la vérification syntaxique.

- Les fichiers présents dans le sous dossier Syntaxe/OK doivent passer la vérification syntaxique sans provoquer d'erreur. Comme le compilateur est appelé avec l'option -s pour les faire compiler, ces fichiers ne doivent pas nécessairement compiler parfaitemement, seulement être syntaxiquement corrects.

Vérif

- Les fichiers présents dans le sous dossier Vérif/KO doivent provoquer une erreur durant la passe 1, soit durant la vérification sémantique. Ces erreurs peuvent être provoquées par des déclarations multiples d'un identifiant, par l'assignation d'une valeur booléenne à un entier etc.

- Les fichiers présents dans le sous dossier Vérif/OK doivent passer la vérification sémantique sans provoquer d'erreurs. Ils doivent donc compiler parfaitement, mais peuvent provoquer des erreurs à l'exécution. Dans tous les cas, ils sont appelés avec l'option -v.

Gencode

- Les fichiers présents dans le sous dossier Gencode/KO doivent provoquer une erreur à l'exécution, cela peut être dû à une division par 0 ou bien à un dépassement de segment.

- Les fichiers présents dans le sous dossier Gencode/OK doivent passer les différentes vérification et passes sans provoquer d'erreurs, donc compiler parfaitement, mais aussi ne pas provoquer d'erreurs à l'exécution.

Explication du script

Nous avons choisi le langage bash afin d'automatiser les manipulations entre dossiers, les créations de fichiers, et le lancement du compilateur avec différentes options sur lesdits fichiers.

Le script se découpe donc en deux grandes parties : la partie création des fichiers de test et la partie exécution du compilateur sur ces fichiers. Pour automatiser les manipulations de fichiers, nous avons déclaré un tableau contenant les noms des fichiers de tests (`minictest0.c`, `minictest1.c`, `minictest2.c` …). Ces noms sont identiques pour chaque sous dossier mais leur nombre varie. Il suffit néanmoins comme cela d'une seule déclaration du tableau `tableau_indi` regroupant tous leurs noms.

```
declare -a tableau_indi=("minictest0.c" "minictest1.c" "minictest2.c" "minictest3.c"
"minictest4.c" "minictest5.c" "minictest6.c" "minictest7.c" "minictest8.c" "minictest9.c"
"minictest10.c" "minictest11.c" "minictest12.c" "minictest13.c"
"minictest14.c""minictest15.c" "minictest16.c" "minictest17.c" "minictest18.c"
"minictest19.c" "minictest20.c" "minictest21.c" "minictest22.c" "minictest23.c"
"minictest24.c")
```

Ensuite nous naviguons dans le premier sous-dossier où nous allons écrire nos fichiers de tests :

```
cd Tests/Syntaxe/OK
```

Par la suite, nous déclarons deux autres tableaux : `tab_carac` qui nous sert à recenser les changements que nous allons effectuer par rapport à un fichier de référence pour générer un nouveau fichier et `fichierref` qui contient le nom du fichier de référence.

```
#declaration des fichiers de tests et des changements de caractères associes
declare -a tab_carac=(       "s/==/!=/g"
                                    "s/var1/carotte/g"
                                    "s/==/||/g"
                                    "s/var1+1/var1%1/g"
                                    "s/var1+1/var1-1/g"
                                    "s/15/-15/g"
                                    "s/2/5/g"
                                    "s/i;/i=0;/g"
                                    "s/+1;/*45;/g"
                                    "s/>>/<</g"
                                    "s/>>/>>>/g"
        "s/while(nb<=5){nb=nb+1;}/do{nb=nb+1;}while(nb<=5);/g"
                                    "s/|/&/g"
"s/nb<=5){nb=nb+1;}/nb>=5){nb=nb-1;}/g"
                                                )
    #declaration et ecriture d'un fichier de reference
declare -a fichierref=("minictestOriginal.c")
```

Ici, toutes les modifications sont destinées à créer des fichiers qui sont différents du premier, mais qui doivent tout de même passer la phase de vérification syntaxique.

Par la suite, nous écrivons le fichier de référence `minictestOriginal.c` , qui va servir de référence globale à tous les fichiers de tests:

```
echo "void main() {"                                >  ${fichierref[0]}
echo "    int var1 = 15, i, nb = 0;"                      >> ${fichierref[0]}
echo "    int bin1= 230; bin2 = 12, bin3;"          >> ${fichierref[0]}
echo "    bool var2 = (true && false), booleen;"    >> ${fichierref[0]}
echo "    if (var2 == true) {"                      >> ${fichierref[0]}
echo "      var1 = 1;"                              >> ${fichierref[0]}
echo "    }"                                        >> ${fichierref[0]}
echo "    else {"                                   >> ${fichierref[0]}
echo "      var1 = 2;"                              >> ${fichierref[0]}
echo "    }"                                        >> ${fichierref[0]}
echo "    while(nb<=5){nb=nb+1;}"                   >> ${fichierref[0]}
echo "    bin1 = bin1 >> 2;"                        >> ${fichierref[0]}
echo "    bin3 = bin1 | bin2;"                      >> ${fichierref[0]}
echo "    bin2 = bin1 ^ bin3;"                      >> ${fichierref[0]}
echo "    booleen = !var2;"                         >> ${fichierref[0]}
echo "    for (i = 1; i < 5; i = i + 1)"            >> ${fichierref[0]}
echo "    {var1 = var1+1;}"                         >> ${fichierref[0]}
echo "    print(var1);"                             >> ${fichierref[0]}
echo "}"                                            >> ${fichierref[0]}
```

On a essayé d'y intégrer le plus de mots réservés au miniC (void, if, else, int, bool …) afin de tous les tester d'un coup.

Ensuite nous avons créé une fonction qui prend en argument un int et qui fait la boucle while suivante :

```
function bouclewhile {
   declare -i i=0
   while [ $i -lt $1 ]
   do
        # step 0
        cp --copy-contents ${fichierref[0]} ${tableau_indi[$i]}

        #step 1
       echo ${tableau_indi[$i]}
        sed -i ${tab_carac[$i]} ${tableau_indi[$i]}
        i=$[$i+1]
   done
}
```

Ainsi, tant que i est inférieur à l'argument de la fonction (désigné par `$1`), on effectue deux étapes.

`step 0 :` On prend le nom d'un fichier dans le tableau `tableau_indi` associé et on crée un fichier à ce nom avec le même contenu que le fichier de référence.

`step 1 :`On affiche le nom du fichier qu'on modifie avec la commande `echo` puis on utilise la commande `sed` pour remplacer des caractères par d'autres dans ce fichier.

La syntaxe de la commande est la suivante :

`sed -i s/carac0/carac1/g [nom_fichier]`

Cela permet de remplacer le caractère `carac0` par le caractère `carac1` dans tout le fichier `nom_fichier`. Cela justifie aussi la façon d'écrire les remplacements de caractère dans le tableau `tab_carac` explicité juste avant.

Comme il y a 14 modifications listées dans `tab_carac`, on lance cette fonction sur 14 boucles pour créer 14 fichiers différents :

```
#lancer la fonction sur 14 boucles :
declare -i nbboucleSO=${#tab_carac[@]}
bouclewhile $nbboucleSO
```

On a aussi pensé à faire d'autre type de modifications : l'ajout de lignes. On a donc créé deux fichiers supplémentaires en ajoutant des lignes telles qu'une déclaration globale avant le main dans un fichier et une boucle if dans un autre.

```
cp --copy-contents ${fichierref[0]} ${tableau_indi[$nbboucleSO]}
sed -i '7i\if ((var1%2)== 0){ var2 = false; }'  ${tableau_indi [$nbboucleSO]}
nbboucleSO=$[$nbboucleSO+1]
cp --copy-contents ${fichierref[0]} ${tableau_indi[$nbboucleSO]}
sed -i '1i\int var3 = 15;' ${tableau_indi[$nbboucleSO]}
nbboucleSO=$[$nbboucleSO+1]
```

Cet ajout se fait grâce à la commande `sed` avec la syntaxe suivante :

`sed -i 'ni\ ligne_a_rajouter' [nom_fichier]`

où on rajoute la ligne `ligne_a_rajouter` à la `n`ième ligne du fichier `nom_fichier`.

Ensuite on se déplace encore, pour se placer dans le dossier Syntaxe/KO et opérer une nouvelle boucle pour créer des fichiers. Ces fichiers comportent des erreurs de syntaxe et doivent provoquer des erreurs aux lignes concernées.

```
cd ../KO
```

On crée un nouveau fichier de référence qui est une copie conforme du fichier source provenant de `Syntaxe/OK` puis on effectue la boucle sur 7 occurrences :

```
declare -a tab_carac=(       "s/==/eq/g"
                                    "s/var1/35/g"
                                    "s/if/&&/g"
                                    "s/2;/2/g"
                                    "s/5;/5/g"
                                    "s/15,/15/g"
                                    "s/;}/}/g"
                                    "s/+1;/}/g"
                                    "s/>>/></g"
                                    "s/else/els/g"
                                    "s/</<</g"
                                    "s/print/printf/g"
                                    "s/{nb=nb+1;}/do{nb=nb+1;}/g")
declare -a fichierref=("minictestOriginal.c")
declare -a fichiersrc=("../OK/minictestOriginal.c")

cp --copy-contents ${fichiersrc[0]}  ${fichierref[0]} #copie du fichier de référence
bouclewhile ${#tab_carac[@]}
```

Ensuite, on navigue de nouveau pour rejoindre le dossier Verif/OK :

```
cd ../../Verif/OK
sudo cp -r ../../Syntaxe/OK/* .
```

Ici on ne fait que copier les fichiers préalablement générés dans Syntaxe/OK.

Ensuite on se rend dans Verif/KO

```
cd ../KO
```

Ici on va aussi procéder deux deux manières différentes: en remplaçant des caractères par d'autres pour générer des erreurs ou en rajoutant des lignes qui sont syntaxiquement correctes mais qui faussent le code.

```
declare -a fichierref=("minictestVerif.c")
declare -a fichiersrc=("../OK/minictestOriginal.c")
declare -a tab_carac=("s/true/tru/g"
                          "s/true/'true'/g"
                          "s/true/1/g" )
declare -i nbboucleVK=${#tab_carac[@]}

#copie du fichier de référence de Syntaxe ok à syntaxe ko
cp --copy-contents ${fichiersrc[0]}  ${fichierref[0]}
#remplacement de caractères
bouclewhile $nbboucleVK

cp --copy-contents ${fichierref[0]}  ${tableau_indi[$nbboucleVK]}
#creation de fichiers par ajout de lignes
sed -i '7i\    if ((var1%2)== 0){ bin4 = false; }' ${tableau_indi[$nbboucleVK]}
#redeclaration de var2
nbboucleVK=$[$nbboucleVK+1]

cp --copy-contents ${fichierref[0]}  ${tableau_indi[$nbboucleVK]}
sed -i '7i\if ((var1%2)== 0){ var2 = 1; }' ${tableau_indi[$nbboucleVK]}
nbboucleVK=$[$nbboucleVK+1]

cp --copy-contents ${fichierref[0]}  ${tableau_indi[$nbboucleVK]}
sed -i '7i\if ((var1%2)== 0){ bool var3 = var2 + 1; }' ${tableau_indi[$nbboucleVK]}
nbboucleVK=$[$nbboucleVK+1]
```

Par la suite, on bouge dans Gencode/OK :

```
cd ../../Gencode/OK
```

Ici, comme dans Verif/OK, on ne fait que copier les fichiers corrects de départ :

```
sudo cp -r ../../Syntaxe/OK/* .
declare -i nbboucleGO=$nbboucleSO
```

Pour le dernier dossier, Gencode/KO, on a généré des erreurs de type division par 0, modulo 0 ou encore dépassement de la limite de calcul en manipulant des entiers afin qu'ils contiennent des valeurs au delà de 2^{32} - 1. Pour se faire, avant d'appeler la fonction `bouclewhile`, on a ajouté une ligne dans le fichier de référence qui contient les opérations de division et modulo afin de les modifier avec la méthode habituelle.

```
cd ../KO
echo "/*/*/*/*/*/*/*/*/* Verification Gencode KO /*/*/*/*/*/*/*/*/*"

declare -a fichierref=("minictestOriginal.c")
declare -a fichiersrc=("../OK/minictestOriginal.c")
#dépassement de la mémoire car manipulé par la suite
declare -a tab_carac=(       "s/15/4294967295/g"
                      "s/+1/+4294967295/g"
                      "s/+1/*4294967295/g"
                      "s/13;/0;/g" #division par 0
                      "s/10;/0;/g") #modulo 0
declare -i nbboucleGK=${#tab_carac[@]}

#copie du fichier de référence de Syntaxe ok à gencode ko
cp --copy-contents ${fichiersrc[0]}  ${fichierref[0]}
#modification du fichier de référence pour le rendre incorrect
cp --copy-contents ${fichierref[0]}  ${tableau_indi[$nbboucleGK]}
sed -i '6 i\          int var4 =  var1 % 10; var1 = var1 / 13;' ${fichierref[0]}

#lancer la fonction sur 5 boucles
bouclewhile $nbboucleGK
```

Une fois tous les fichiers créés on se replace dans le dossier père de Tests/ et on commence la compilation. Nous avons aussi décidé de recompiler le compilateur pour être sûrs qu'ils soit à jour lors de nos tests.

```
cd ../../.. # retour au dossier source de Tests
make realclean
make # mettre à jour le compilateur minicc
```

Premièrement, afin de tester le bon fonctionnement du compilateur, on le lance avec des options qui vont le faire sortir à la fin du parsing des arguments :

```
#test des différentes options sans compilation
./minicc -b
./minicc -h
```

Ensuite, on va compiler chaque fichier de chaque sous dossier en traitant un sous dossier après l'autre.

On commence par Syntaxe/OK en appelant le compilateur avec l'option -s (syntaxe : `./minicc -s Tests/Syntaxe/OK/${tableau_indi[$i]})` et en affichant chaque fichier avant de le compiler. Ceci est fait dans une boucle while qui tourne sur le nombre de fichiers de test contenus dans ce dossier soit `$nbboucleSO`.

```
declare -i i=0
echo "Compilation des fichiers de syntaxe OK"
while [ $i -lt $nbboucleSO ]
  do
       echo "*********************************************************"
       echo ${tableau_indi[$i]}
       echo "syntaxe ok"
       more Tests/Syntaxe/OK/${tableau_indi[$i]}
     ./minicc -s Tests/Syntaxe/OK/${tableau_indi[$i]}
       echo "*********************************************************"
       i=$[$i+1]
    done
```

La suite des appels à `minicc` pour chaque sous dossier est similaire, seuls changent les options saisies et le nombre de fichier à compiler.

IV/ Implémentation des passes 1 et 2

Passe 1

Durant la passe 1, nous devons parcourir l'arbre du programme, créer les différents contextes et environnements reliés au programme et vérifier ou affecter les divers attributs des noeuds du programme. Certains de ses attributs sont hérités des noeud parents, d'autres sont synthétisés depuis les noeuds enfants. Dans chacun des cas, il faut mettre à jour des valeurs dans le noeud correspondant. Certaines règles possèdent des conditions de validation (par exemple toutes les conditions de boucles sont booléennes, la fonction main est obligatoirement de type void, etc) qu'il va falloir vérifier. Nous avons choisi de générer chaque erreur liée à la passe 1, puis de sortir avec un appel à exit(1) après tout le parcours si jamais une erreur avait été rencontrée. Pour cela, nous avons défini une fonction nommée yyerror_passe1.

Nous avons construit la passe 1 comme une fonction récursive nommée `parcours_rec` qui parcourt l'arbre de haut en bas comme la fonction `dump_tree`. Elle est lancée dans la fonction `lancer_parcours()`.

```
static void lancer_parcours(node_t root) {
	assert(root->nature == NODE_PROGRAM);
	parcours_rec(root);
}
```

La fonction récursive commence par un test pour s'assurer que le noeud n'est pas nul, sinon elle sort directement.

```
void parcours_rec(node_t n) {
	if (n == NULL)
	{
		return ;
	}
```

Ensuite, elle se divise en trois parties :

- Tout d'abord un switch pour déterminer les contextes à créer et vérifier les conditions des les noeuds terminaux dans le cas où ils sont sous conditions.
- Ensuite, on fait un appel récursif sur chaque enfant du noeud passé en argument.
- Pour finir, on fait de nouveau un switch sur la nature du noeud pour mettre à jour les attributs synthétisés au cas où les enfants auraient été modifiés par l'appel récursif, et clore les contextes.

1. Premier switch

```
switch (n->nature) {
case NODE_PROGRAM:
    if ( n->opr[0] != NULL ) // si il y a des déclarations globales
    {
       push_global_context();
       print_context("Pushing global context\n");
       globalpasse1 = true;
    }
    break;
```

Dans le cas du noeud de nature NODE_PROGRAM,on peut vérifier si il y a une liste de déclarations globales non nulles en regardant si son premier fils est nul. Si jamais il ne l'est pas, on doit créer un contexte global en utilisant la fonction à cet usage. On note aussi l'information dans le booléen `globalpasse1.`

```
case NODE_BLOCK:
       push_context();
       print_context("Pushing simple context\n");
       break;
```

Dans le cas d'un node de nature `NODE_BLOCK`, on entre dans un endroit où il faut générer un contexte et le pousser sur la pile.

```
case NODE_IDENT:
       if(!globalpasse1) // dans le main
       {
              previousdef = get_decl_node(n->ident);
              if ( previousdef == NULL || previousdef == n->decl_node)
              {
                     if( strcmp(n->ident, "main") != 0)
                     {
                            printf("Erreur près du signe %s\n", n->ident);
                            yyerror_passe1(&n,"pas de declaration précédente de l'ident");
                     }
              }
           else
           {
              n->type = previousdef->type;
           }
       }
       break;
```

Dans le cas d'un node `NODE_PROGRAM` , on vérifie si on est entré dans le main. Alors, on fait appel à `get_decl_node` pour trouver la dernière declaration en date de l'identifiant. Si jamais il n'a pas été déclaré ou que sa déclaration ne correspond pas au noeud de sa déclaration (stocké dans l'attribut `decl_node` de l'identifiant), alors il y a un appel à un identifiant qui n'a jamais été déclaré. Dans ce cas, excepté pour le main, on doit générer une erreur.

```
case NODE_STRINGVAL:
       offset = add_string(n->str);
       n->offset = offset;
     break;
```

Dans le cas d'une valeur de nature , il faut déterminer l'offset à lui accorder dans le .data. Pour cela, on fait appel à la fonction `add_string` de env.h et on stocke l'offset généré dans l'attribut `offset` du noeud.

```
case NODE_DECLS:
	if ((n->opr[0])->type == TYPE_VOID)
	{
		yyerror_passe1(&n, "Déclaration de variable de type void");
	}
	break;
```

Dans le cas d'un `NODE_DECLS` , on doit vérifier que la liste de déclarations n'est pas de type `TYPE_VOID`. Dans ce cas, on génère une erreur.

```
case NODE_DECL:
	offset = env_add_element(n->opr[0]->ident, n->opr[0], 4);
	if (offset < 0)
	{
		printf("Probleme offset\n");
	}
	else
	{
		n->opr[0]->offset = offset;
	}
	for (i = 0 ; i < n->nops; i++)
	{
		if ((n->opr[i]) && (n->opr[i])->nature == NODE_IDENT)
		{
			(n->opr[i])->decl_node = n;
		}
	}
     break;
```

Dans le cas d'un `NODE_DECL`, on ajoute le noeud à l'attribut `decl_node` de son enfant de type `NODE_IDENT.`

```
case NODE_FUNC:
	if (globalpasse1)
     {
		globalpasse1 = false;
     }
     reset_env_current_offset();
     print_context("Resetting current context\n");
     if ((n->opr[1])->type != TYPE_VOID)
     {
		yyerror_passe1(&(n->opr[1]), "Le main n'a pas le bon type, type void attendu
\n");
	}
     if (strcmp( ((n->opr[1])->ident), "main") != 0)
     {
		printf("nature du noeud: %s\n", node_nature2string((n->opr[1])->nature));
		printf("nom du main: %s\n", (n->opr[1])->ident);
		yyerror_passe1(&(n->opr[1]), "La fonction principale ne s'appelle pas main
\n");
	}
	break;
```

Dans le cas d'un `NODE_FUNC`, on remet globalpasse1 à false et on fait appel à la fonction `reset_env_current_offset()`. Ensuite, on vérifie d'une part si le type du noeud est bien `TYPE_VOID` et si son nom est bien "main". Dans chacun des cas, si la condition n'est pas respectée, on génère l'erreur associée.

2. Deuxième switch

```
case NODE_BLOCK:
       pop_context();
       print_context("poping context\n");
       break;
```

Dans le cas d'un `NODE_BLOCK`, après l'appel récursif aux enfants, on réduit le contexte en faisant appel à la fonction `pop_context()`.

```
case NODE_AFFECT:
     n->type = (n->opr[0])->type;
     if (((n->opr[0])->type) != ((n->opr[1])->type))
     {
       yyerror_passe1(&n, "Affectation entre deux opérandes de type différents\n");
              printf(">Type et nature a gauche : %s et %s\n",
node_type2string((n->opr[0])->type), node_nature2string((n->opr[0])->nature));
              printf(">Type et nature a droite : %s et %s\n",
node_type2string((n->opr[1])->type), node_nature2string((n->opr[1])->nature));
       }
     break;
```

Dans le cas d'un `NODE_AFFECT`, on vérifie si les deux opérandes ont le même type. Sinon, on génère l'erreur correspondante.

```
case NODE_PLUS:
case NODE_MINUS:
case NODE_MUL:
case NODE_DIV:
case NODE_MOD:
case NODE_BAND:
case NODE_BOR:
case NODE_BXOR:
case NODE_SLL:
case NODE_SRL:
case NODE_SRA:
     /* type_op_binaire(int,int) = int */
     if(n->opr[0]->type != TYPE_INT)
     {
         yyerror_passe1(&n, "Le premier élément de l'opération n'est pas entier\n");
     }
     if(n->opr[1]->type != TYPE_INT)
     {
         yyerror_passe1(&n, "Le deuxième élément de l'opération n'est pas entier\n");
     }
     n->type = TYPE_INT;
     break;
```

Dans le cas des opérateurs binaires entre deux int, on vérifie si chaque opérande est du bon type, puis on assigne le `TYPE_INT` à l'expression.

```
case NODE_EQ:
case NODE_NE:
       /* type_op_binaire(int,int) = bool */
       /* type_op_binaire(bool,bool) = bool */
       if( (n->opr[0]->type == TYPE_INT && n->opr[1]->type != TYPE_INT) ||
              (n->opr[0]->type == TYPE_BOOL && n->opr[1]->type != TYPE_BOOL) )
     {
              if (nbtraces == 5)
              {
              printf("types de op1 : %s et op2 : %s \n",
node_type2string((n->opr[0])->type) , node_type2string((n->opr[1])->type));
              }

              yyerror_passe1(&n, "Les éléments ne sont pas de même type\n");
     }
     n->type = TYPE_BOOL;
     break;
```

Dans le cas des opérateurs binaires NODE_EQ et NODE_NE générant des expressions booléennes, on vérifie que les deux opérandes sont de même type (TYPE_BOOL ou TYPE_INT), et on assigne le type TYPE_BOOL au noeud.

```
case NODE_LT:
case NODE_GT:
case NODE_LE:
case NODE_GE:
       /* type_op_binaire(int,int) = bool */
       if (nbtraces == 5)
       {
              printf("types de op1 : %s et op2 : %s \n",
node_type2string((n->opr[0])->type), node_type2string((n->opr[1])->type));
       }
       if(n->opr[0]->type != TYPE_INT)
       {
              yyerror_passe1(&n, "Le premier élément de l'opération n'est pas entier\n");
       }
       if(n->opr[1]->type != TYPE_INT)
       {
              yyerror_passe1(&n, "Le deuxième élément de l'opération n'est pas entier\n");
       }
       n->type = TYPE_BOOL;
       break;
```

Dans le cas des opérateurs de comparaison NODE_LT, NODE_GT, NODE_LE et NODE_GE, on vérifie que les opérandes sont chacun de type TYPE_INT et on assigne le type TYPE_BOOL à l'expression.

```
case NODE_AND:
case NODE_OR:
        /* type_op_binaire(bool,bool) = bool */
        if(n->opr[0]->type != TYPE_BOOL)
        {
                yyerror_passe1(&n, "Le premier élément de l'opération n'est pas booléen\n");
        }
        if(n->opr[1]->type != TYPE_BOOL)
        {
                yyerror_passe1(&n, "Le deuxième élément de l'opération n'est pas
booléen\n");
        }
        n->type = TYPE_BOOL;
        break;
```

Dans le cas des opérateurs binaires NODE_AND et NODE_OR, on vérifie que les deux opérandes sont de type TYPE_BOOL, puis on assigne le type TYPE_BOOL à l'expression.

```
case NODE_UMINUS:
case NODE_BNOT:
 /* type_op_unaire(int) = int */
        if((n->opr[0])->type != TYPE_INT)
        {
                yyerror_passe1(&n ,"Essai de operation unaire sur un non int\n");
        }
        n->type= TYPE_INT;
        break;
```

Dans le cas des opérateurs unaires sur int NODE_UMINUS et NODE_BNOT, on vérifie que l'opérande est de type TYPE_INT puis on assigne le type TYPE_INT à l'expression.

```
case NODE_NOT:
        /* type_op_unaire(bool) = bool */
        if(n->opr[0]->type != TYPE_BOOL)
        {
                yyerror_passe1(&n, "Essai de operation unaire sur un non bool\n");
        }
        n->type = TYPE_BOOL;
        break;
```

Dans le cas de l'opérateur unaire NODE_NOT,on vérifie que l'opérande est de type TYPE_BOOL puis on assigne le type TYPE_BOOL à l'expression.

```
case NODE_IF:
case NODE_WHILE:
        if((n->opr[0])->type != TYPE_BOOL)
        {
                yyerror_passe1(&n, "Expression dans une boucle n'est pas booleenne\n");
        }
        break;
```

Dans les boucles NODE_IF et NODE_WHILE, on vérifie que le premier enfant du noeud est bien booléen sinon on génère une erreur.

```
case NODE_DOWHILE:
case NODE_FOR:
        if((n->opr[1])->type != TYPE_BOOL)
        {
                yyerror_passe1(&n, "Expression dans une boucle n'est pas booleenne\n");
        }
        break;
```

Dans les boucles `NODE_DOWHILE` et `NODE_FOR`, on vérifie que le deuxième enfant du noeud est bien booléen sinon on génère une erreur.

```
case NODE_DECL:
        if( n->opr[1] && (n->opr[0])->type != (n->opr[1])->type )
        {
                yyerror_passe1(&n,"Erreur de type dans la declaration\n");
        }
break;
```

Dans le cas d'un `NODE_DECL`, après assignation des attributs des enfants on vérifie que si la déclaration était de type (ident = expr), l'identifiant et l'expression ont le même type.