SORBONNE UNIVERSITÉ

Sorbonne Université – École Polytech' Sorbonne

Projet Compilation

Année 2025-2026

I. Spécifications
II. Ressources et environnement de développement
III. Annexes

Responsable d'UE : Quentin Meunier

I. Specifications

1 Introduction

Ce projet a pour but l'écriture d'un compilateur, c'est-à-dire l'écriture d'un programme qui transforme un programme source en programme assembleur.

Le langage utilisé pour les programmes source est appelé MiniC : il s'agit d'un sous-ensemble du langage C (avec quelques différences), ayant notamment les restrictions suivantes par rapport au C :

- les expressions et variables n'ont que deux types : `int` et `bool`
- il y a un typage fort des expressions (pas de conversions implicite `int` → `bool`)
- l'évaluation des expressions est faite de manière non-paresseuse
- il n'y a pas de fonctions (hormis le `main`), pointeurs, tableaux
- les mots-clés suivants et fonctionnalités associées ne sont pas supportés : `switch`, `case`, `break`, `continue`, `goto`, `typedef`, `struct`, `union`, `volatile`, `register`, `packed`, `inline`, `static`, `extern`, `unsigned`, `signed`, `long`, `long long`, `short`, `char`, `size_t`, `float`, `double`
- les opérateurs suivants ne sont pas supportés : `++`, `--`, `-=`, `+=`, `*=`, `/=`, `«=`, `»=`, `&=`, `|=`, ...
- il n'y a pas de cast

Le langage cible est le langage assembleur Mips.

1.1 Analyse lexicale

L'analyse lexicale est la phase de transformation d'une suite de caractères (d'un fichier) en une suite de lexèmes (ou tokens). Par exemple, la suite de lettres `for`, quand elle est entourée de caractères autres que des chiffres, des lettres, et du caractère `_`, est un mot réservé du langage : on lui associe donc un token représentant le for. Comme la machine d'état qui fait cette transformation est très pénible à écrire manuellement, on utilise en général un outil pour décrire les tokens avec un plus haut niveau d'abstraction, l'outil se chargeant de la génération du fichier C contenant la machine d'état correspondante. L'outil utilisé dans ce projet est Lex.

1.2 Analyse syntaxique

L'analyse syntaxique est la phase au cours de laquelle on vérifie que la suite de tokens en sortie de l'analyse lexicale est valide. Par exemple, une succession de deux tokens associés au mot-clé `for` n'est pas valide syntaxiquement. Pour faire cette analyse, on décrit les langages valides à l'aide de règles de grammaire (en général de type hors-contexte). C'est au cours de cette analyse qu'est construit l'arbre du programme : à chaque fois, ou presque, qu'une règle de grammaire est reconnue (par exemple une suite de token en partie droite d'une règle), on effectue la construction de la partie correspondante de l'arbre du programme ; dans la suite de l'analyse syntaxique, le non-terminal en partie gauche de cette règle remplacera la suite des tokens pour la reconnaissance de la prochaine partie droite de règle.

De la même manière que pour l'analyse lexicale, on utilise un outil dans lequel on a simplement à écrire les règles de la grammaire, et qui génère le code C associé. L'outil utilisé pour ce projet est Yacc.

Dans ce projet, le totalité des règles de la grammaire hors-contexte du langage sont données. Il vous faut compléter les actions associées pour construire l'arbre du programme.

Les analyses lexicales et syntaxiques sont effectuées conjointement. La figure 1 résume le processus de compilation du compilateur.

1.3 Analyse sémantique (ou de vérifications contextuelles)

L'analyse sémantique est faite au cours de la première passe, ou "passe 1". Une passe ici désigne une exploration de l'arbre (en profondeur). Même si un programme est syntaxiquement correct, il n'est pas forcément correct : en effet, un nom de variable peut être utilisé sans avoir été déclaré, ou une variable booléenne additionnée avec une variable entière. Ces vérifications sont faites lors de cette passe.

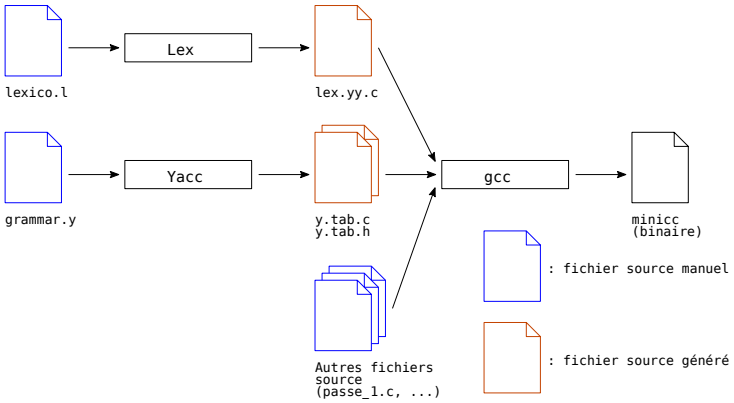


FIGURE 1 – Processus de compilation utilisant lex et yacc

Les programmes sémantiquement corrects sont spécifiés dans ce document à l'aide d'une grammaire attribuée. La passe 1 doit donc implémenter toutes ces vérifications. De plus, c'est au cours de cette passe que l'on fait les liens entre les occurrences des variables et leur déclaration, de manière à pouvoir avoir directement la position en mémoire d'une variable lors de la génération de code.

1.4 Génération de code

La passe de génération de code, ou passe 2, effectue un parcours de l'arbre au cours duquel sont générées les instructions assembleur du programme. Il n'y a plus de vérification à effectuer au cours de cette passe, sauf éventuellement à l'aide d'asserts.

2 Exemple introductif illustrant les différentes étapes de la compilation

Cette section illustre les résultats produits à l'issue de chaque analyse et passe, en considérant le programme MiniC suivant :

```
1  // Un exemple de programme MiniC
2  int start = 0;
3  int end = 100;
4
5  void main() {
6     int i, s = start, e = end;
7     int sum = 0;
8     for (i = s; i < e; i = i + 1) {
9        sum = sum + i;
10    }
11    print("sum: ", sum, "\n");
12 }
```

2.1 Étape d'analyse lexicale

Au cours de l'analyse lexicale, le programme est transformé en une séquence des lexèmes (ou *tokens*). La séquence de lexèmes pour le programme d'exemple est donnée ci-après, avec les numéros de ligne, les noms des identificateurs et les valeurs des littéraux.

TOK_INT 2	TOK_IDENT 2 'start'	TOK_AFFECT 2	TOK_INTVAL 2 '0'	TOK_SEMICOL 2	TOK_INT 3	TOK_IDENT 3 'end'	TOK_AFFECT 3

TOK_INTVAL 3 '100'	TOK_SEMICOL 3	TOK_VOID 5	TOK_IDENT 5 'main'	TOK_LPAR 5	TOK_RPAR 5	TOK_LACC 5	TOK_INT 6

TOK_IDENT 6 'i'	TOK_COMMA 6	TOK_IDENT 6 's'	TOK_AFFECT 6	TOK_IDENT 6 'start'	TOK_COMMA 6	TOK_IDENT 6 'e'	TOK_AFFECT 6

TOK_IDENT 6 'end'	TOK_SEMICOL 6	TOK_INT 7	TOK_IDENT 7 'sum'	TOK_AFFECT 7	TOK_INTVAL 7 '0'	TOK_SEMICOL 7

TOK_FOR 8	TOK_LPAR 8	TOK_IDENT 8 'i'	TOK_AFFECT 8	TOK_IDENT 8 's'	TOK_SEMICOL 8	TOK_IDENT 8 'i'	TOK_LT 8

TOK_IDENT 8 'e'	TOK_SEMICOL 8	TOK_IDENT 8 'i'	TOK_AFFECT 8	TOK_IDENT 8 'i'	TOK_PLUS 8	TOK_INTVAL 8 '1'	TOK_RPAR 8

TOK_LACC 8	TOK_IDENT 9 'sum'	TOK_AFFECT 9	TOK_IDENT 9 'sum'	TOK_PLUS 9	TOK_IDENT 9 'i'	TOK_SEMICOL 9

TOK_RACC 10	TOK_PRINT 11	TOK_LPAR 11	TOK_STRING 11 "sum :"	TOK_COMMA 11	TOK_IDENT 9 'sum'	TOK_COMMA 11	TOK_STRING 11 "\n"

TOK_RPAR 11	TOK_RACC 12

2.2 Étape d'analyse syntaxique

Lors de la phase d'analyse syntaxique, l'arbre correspondant au programme est construit à partir de la séquence de lexèmes. Les champs `ident` des noeuds de nature `IDENT` ainsi que les champs `value` des noeuds

de nature `INTVAL`, `BOOLVAL` et `STRINGVAL` sont initialisés aux valeurs correspondantes. L'arbre du programme d'exemple est représenté figure 2.

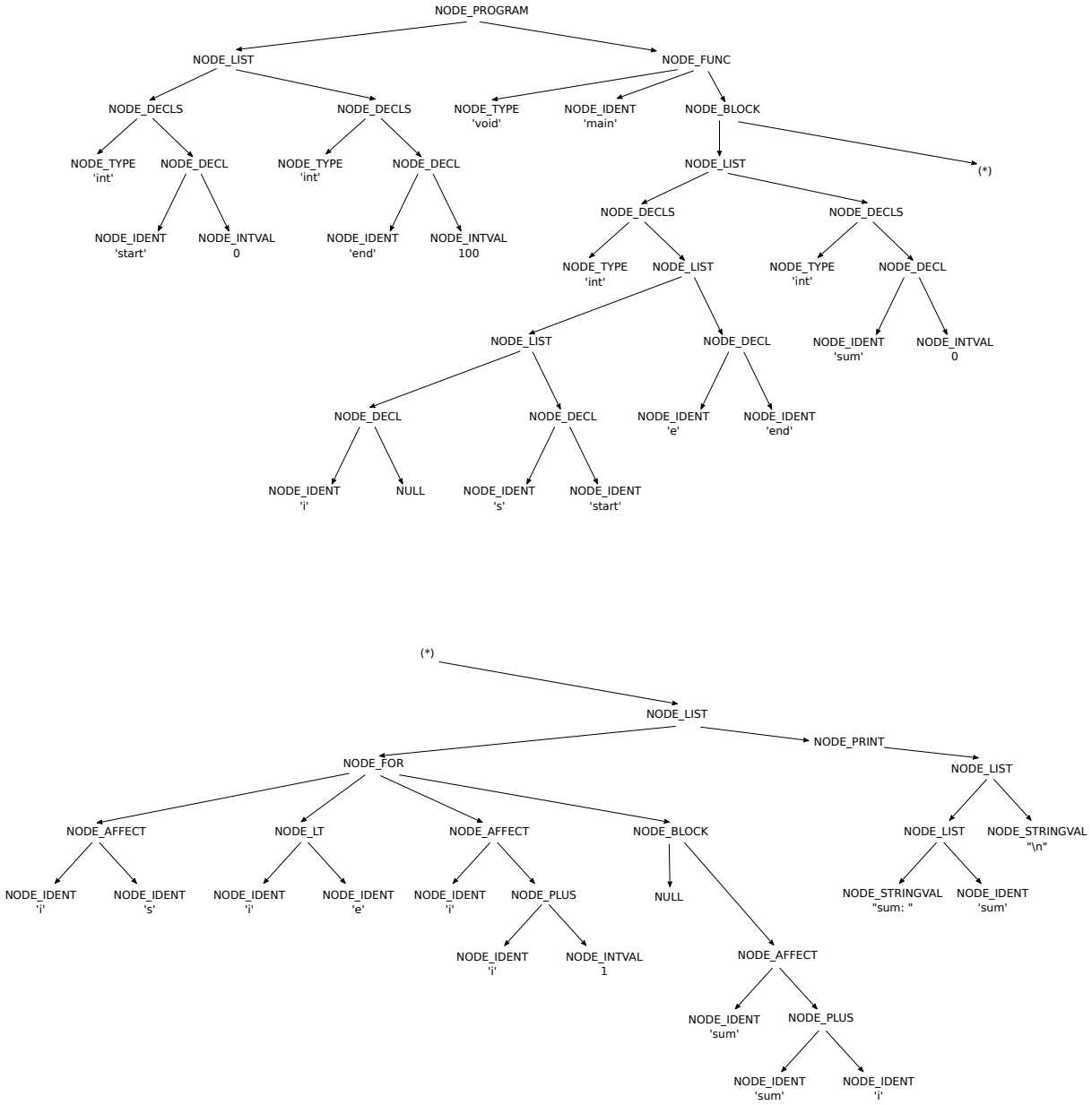


FIGURE 2 – Arbre du programme après analyse syntaxique

2.3 Étape de vérifications contextuelles et décorations

Lors de l'analyse contextuelle, on vérifie que le programme écrit respecte la spécification du langage et que l'on peut générer un code assembleur correspondant. De plus, les champs suivants sont mis à jour :

- Le champ `type` de tous les noeuds pouvant se trouver à la racine d'une expression (exemple : `IDENT`, `PLUS`, `BXOR`). Ce champ est mis à jour au fur et à mesure de la vérification dans la passe 1 mais n'a pas d'utilité dans la passe 2.
- Le champ `global_decl` des noeuds de nature `IDENT` correspondant à une déclaration, qui est mis à jour pour indiquer si la variable est globale ou locale.
- Le champ `decl_node` des noeuds de nature `IDENT` (autres que les noeuds de déclaration), qui est mis à jour avec l'adresse du noeud contenant la déclaration de la variable référencée.

- Le champ `offset` des noeuds de nature `IDENT` correspondant à une déclaration, qui est mis à jour pour refléter l'emplacement en mémoire de la variable :
  - Pour les variables locales, il s'agit de l'offset de pile (en octets)
  - Pour les variables globales, il s'agit de l'offset dans la section `.data` (en octets)
- Le champ `offset` du noeud de nature `FUNC`, qui est mis à jour avec la taille (en octets) en pile réservée pour les variables locales ; il s'agit également de l'offset de départ pour les temporaires

L'arbre du programme d'exemple à la fin de l'analyse contextuelle, avec les champs `offset` et `decl_node` mis à jour, est représenté figure 3.

2.4 Programme assembleur

```
# Declaration des variables globales
.data

start: .word 0
end:   .word 100
.asciiz "sum: "
.asciiz "\n"


# Programme
.text


main:
    # Prologue : allocation en pile pour les variables locales
    # i se trouve a l'emplacement 0($29)
    # s se trouve a l'emplacement 4($29)
    # e se trouve a l'emplacement 8($29)
    # sum se trouve a l'emplacement 12($29)
    addiu $29, $29, -16
    # s = start
    lui   $8, 0x1001
    lw    $8, 0($8)
    sw    $8, 4($29)
    # e = end
    lui   $8, 0x1001
    lw    $8, 4($8)
    sw    $8, 8($29)
    # sum = 0
    ori   $8, $0, 0
    sw    $8, 12($29)
    # for (i = s; i < e; i = i + 1)
    # i = s
    lw    $8, 4($29)
    sw    $8, 0($29)
    # i < e ?
_L1:
    lw    $8, 0($29)
    lw    $9, 8($29)
    slt   $8, $8, $9
    beq   $8, $0, _L2
    # sum = sum + i
    lw    $8, 12($29)
    lw    $9, 0($29)
    addu  $8, $8, $9
```

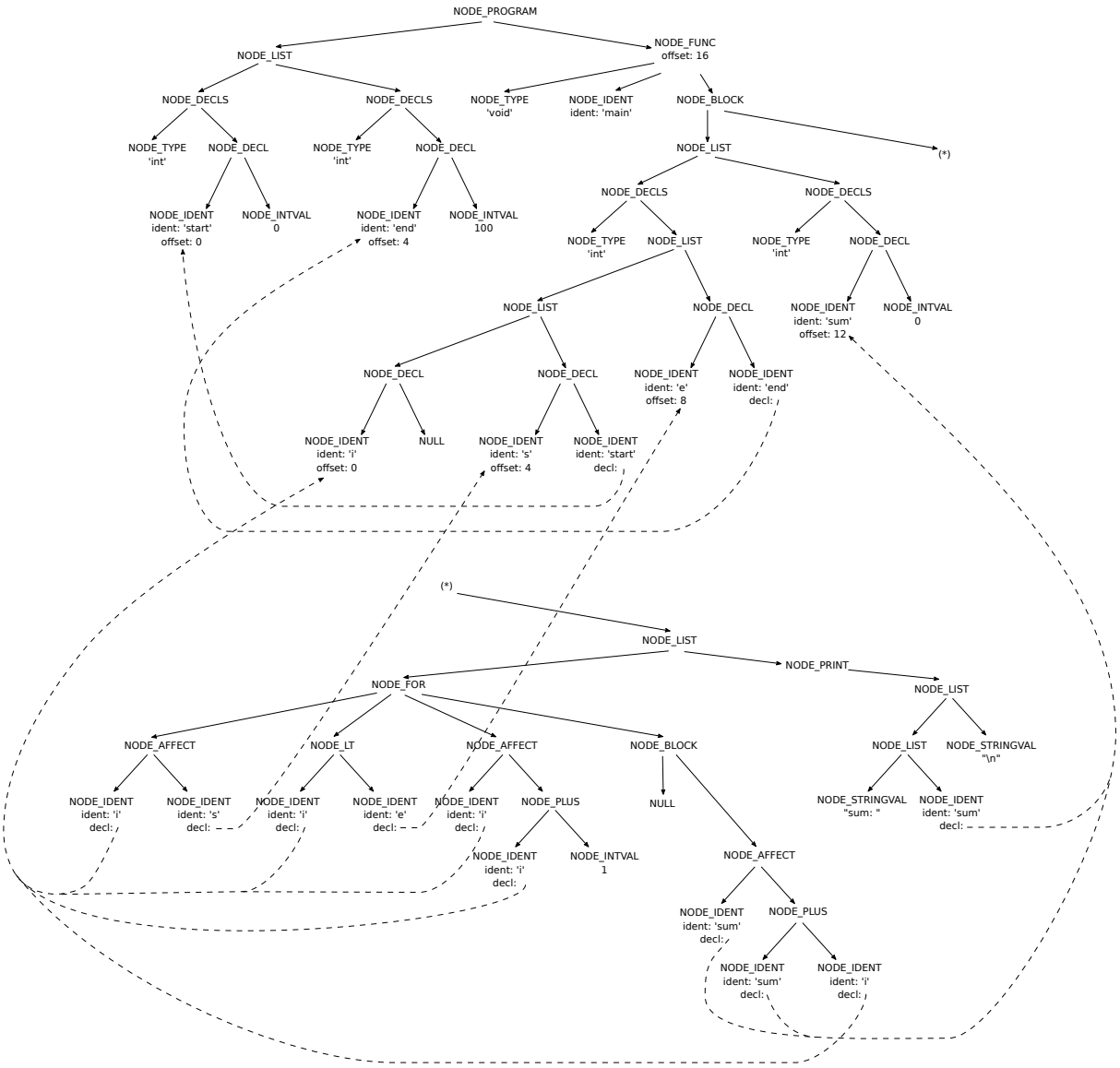


FIGURE 3 – Arbre du programme après vérifications contextuelles et décorations

```
sw      $8, 12($29)
# i = i + 1
lw      $8, 0($29)
ori     $9, $0, 1
addu    $8, $8, $9
sw      $8, 0($29)
```

```
    # Retour au test de boucle
    j _L1
_L2:
    # print("sum :")
    lui   $4, 0x1001
    ori   $4, $4, 8
    ori   $2, $0, 4
    syscall
    # print(sum)
    lw    $4, 12($29)
    ori   $2, $0, 1
    syscall
    # print("\n");
    lui   $4, 0x1001
    ori   $4, $4, 14
    ori   $2, $0, 4
    syscall
    # Desallocation des variables locales en pile
    addiu $29, $29, 16
    # exit
    ori   $2, $0, 10
    syscall
```

3 Lexicographie de MiniC

3.1 Conventions de notations

- Les éléments entre simple quotes (comme `'0'`, `','`) désignent les caractères ou séquences de caractères correspondants ;
- Les mots notés en majuscules (comme `LETTRE`, `CHIFFRE`) désignent des langages.
- Les opérateurs sur les langages utilisés sont les notations habituelles d'expressions régulières.
- On appelle *caractère de formatage* :
  - la tabulation horizontale
  - la fin de ligne
- On appelle *caractère imprimable* tout caractère dont le code ASCII est dans l'intervalle [0x20-0x7E]. N.B. Les code des caractères `' '` (espace), `'"'` et `'\'` sont respectivement 0x20, 0x22 et 0x5c. La tabulation horizontale et la fin de ligne ne sont pas des caractères imprimables.

Remarque : la spécification donnée ici utilise des notations usuelles. Il ne faut pas la recopier telle quelle mais l'adapter à la syntaxe de lex.

3.2 Unités lexicales

Les unités lexicales de MiniC sont les mots réservés, les symboles spéciaux, ainsi que les langages `ENTIER`, `IDF`, et `CHAINE`.

3.3 Mots réservés

Les séquences de lettres suivantes sont des mots réservés :

```
void      int      bool     true     false      if
else      while    for      do       print
```

3.4 Identificateurs

```
LETTRE   =  {'a', ..., 'z', 'A', ..., 'Z'}
CHIFFRE  =  {'0', ..., '9'}
IDF      =  (LETTRE)(LETTRE | CHIFFRE | '_')*
```

Exception : les mots réservés ne sont pas des identificateurs.

3.5 Symboles spéciaux

Les caractères suivants, ainsi que les associations suivantes de deux caractères ont un sens particulier en MiniC :

```
'+'  '-'  '*'  '/'  '%'  '>'  '<'  '!'  '~'  '&'
'|'  '^'  '='  ';'  ','  '('  ')'  '{'  '}'
'>>' '>>>', '<<' '>=' '<=' '==' '!=' '&&' '||'
```

3.6 Littéraux entiers

```
CHIFFRE_NON_NUL = {'1', ..., '9'}
ENTIER_DEC      = '0' | CHIFFRE_NON_NUL CHIFFRE*
LETTRE_HEXA     = {'a', ..., 'f', 'A', ..., 'F'}
ENTIER_HEXA     = '0x' (CHIFFRE | LETTRE_HEXA)+
ENTIER          = ENTIER_DEC | ENTIER_HEXA
```

3.7 Chaines de caractères

`CHAINE_CAR` est l'ensemble de tous les caractères imprimables, à l'exception des caractères `'"'` et `'\'`.

```
CHAINE = '"' (CHAINE_CAR | '\"' | '\n')*  '"'
```

3.8 Commentaires

Un commentaire est une suite de caractères imprimables et de tabulations qui commence par `'//'` et s'étend jusqu'à la fin de la ligne.

3.9 Séparateurs

Les séparateurs de MiniC sont `' '` (caractère d'espace) et les caractères de formatage (tabulation horizontale et fin de ligne).

4 Syntaxe

Ce document présente la syntaxe hors-contexte du langage MiniC. Les lexèmes (ou tokens) sont les éléments retournés à l'issue de l'analyse lexicale sous la forme d'une suite.

4.1 Définition des lexèmes

```
%token TOK_VOID TOK_INT TOK_BOOL TOK_TRUE TOK_FALSE TOK_IF TOK_DO TOK_WHILE TOK_FOR
%token TOK_PRINT TOK_SEMICOL TOK_COMMA TOK_LPAR TOK_RPAR TOK_LACC TOK_RACC
```

Les lexèmes suivants ont une associativité et une priorité donnée. Les opérateurs sont dans l'ordre de priorité croissante. Le lexème `TOK_THEN` n'est jamais retourné et est là pour résoudre le problème classique de positionnement du `else` dans le cas d'une expression `if (a) if (b) c; else d;`. De même, le lexème `TOK_UMINUS` sert à changer la priorité du `TOK_MINUS` lorsque le `'-'` rencontré est un moins unaire.

```
%nonassoc TOK_THEN
%nonassoc TOK_ELSE

/* a = b = c + d <=> b = c + d; a = b; */
%right TOK_AFFECT

%left TOK_OR
%left TOK_AND
%left TOK_BOR
%left TOK_BXOR
%left TOK_BAND
%nonassoc TOK_EQ TOK_NE
%nonassoc TOK_GT TOK_LT TOK_GE TOK_LE
%nonassoc TOK_SRL TOK_SRA TOK_SLL

/* a / b / c = (a / b) / c et a - b - c = (a - b) - c */
%left TOK_PLUS TOK_MINUS
%left TOK_MUL TOK_DIV TOK_MOD

%nonassoc TOK_UMINUS TOK_NOT TOK_BNOT
```

Pour les lexèmes qui retournent une information en plus, on doit spécifier le type de cette information.

```
%token <intval> TOK_INTVAL;
%token <strval> TOK_IDENT TOK_STRING;

%type <ptr> program listdecl listdeclnonnull vardecl ident type listtypedecl decl maindecl
%type <ptr> listinst listinstnonnull inst block expr listparamprint paramprint
```

4.2 Règles syntaxiques de MiniC

Certaines listes utilisent des non-terminaux différents pour le cas vide et non-vide, afin d'éviter des conflits de type *shift-reduce* dans yacc.

```
program         : listdeclnonnull maindecl
                | maindecl
                ;

listdecl        : listdeclnonnull
                |
                ;

listdeclnonnull : vardecl
```

```
                | listdeclnonnull vardecl
                ;

vardecl         : type listtypedecl TOK_SEMICOL
                ;

type            : TOK_INT
                | TOK_BOOL
                | TOK_VOID
                ;

listtypedecl    : decl
                | listtypedecl TOK_COMMA decl
                ;

decl            : ident
                | ident TOK_AFFECT expr
                ;

maindecl        : type ident TOK_LPAR TOK_RPAR block
                ;

listinst        : listinstnonnull
                |
                ;

listinstnonnull : inst
                | listinstnonnull inst
                ;

inst            : expr TOK_SEMICOL
                | TOK_IF TOK_LPAR expr TOK_RPAR inst TOK_ELSE inst
                | TOK_IF TOK_LPAR expr TOK_RPAR inst %prec TOK_THEN
                | TOK_WHILE TOK_LPAR expr TOK_RPAR inst
                | TOK_FOR TOK_LPAR expr TOK_SEMICOL expr TOK_SEMICOL expr TOK_RPAR inst
                | TOK_DO inst TOK_WHILE TOK_LPAR expr TOK_RPAR TOK_SEMICOL
                | block
                | TOK_SEMICOL
                | TOK_PRINT TOK_LPAR listparamprint TOK_RPAR TOK_SEMICOL
                ;

block           : TOK_LACC listdecl listinst TOK_RACC
                ;

expr            : expr TOK_MUL expr
                | expr TOK_DIV expr
                | expr TOK_PLUS expr
                | expr TOK_MINUS expr
                | expr TOK_MOD expr
                | expr TOK_LT expr
                | expr TOK_GT expr
                | TOK_MINUS expr %prec TOK_UMINUS
                | expr TOK_GE expr
                | expr TOK_LE expr
                | expr TOK_EQ expr
                | expr TOK_NE expr
                | expr TOK_AND expr
                | expr TOK_OR expr
                | expr TOK_BAND expr
```

```
                | expr TOK_BOR expr
                | expr TOK_BXOR expr
                | expr TOK_SRL expr
                | expr TOK_SRA expr
                | expr TOK_SLL expr
                | TOK_NOT expr
                | TOK_BNOT expr
                | TOK_LPAR expr TOK_RPAR
                | ident TOK_AFFECT expr
                | TOK_INTVAL
                | TOK_TRUE
                | TOK_FALSE
                | ident
                ;

listparamprint  : listparamprint TOK_COMMA paramprint
                | paramprint
                ;

paramprint      : ident
                | TOK_STRING
                ;

ident           : TOK_IDENT
                ;
```

5 Grammaire d'arbres

5.1 Généralités

Les arbres construits lors de l'analyse syntaxique sont décrits à l'aide d'une grammaire hors-contexte. Les non terminaux sont en gras minuscule ; ils définissent des "classes d'arbres", ensemble des arbres qui en dérivent. L'axiome est le premier non terminal, ici **program**. La classe d'arbres **program** est donc l'ensemble des arbres des programmes MiniC syntaxiquement corrects.

Les règles de la grammaire sont de la forme :

- **G → D1 | D2 | ... | Dn**

(**n** ≥ 1) où **G** est le non terminal partie gauche (définissant une classe d'arbres), et les **Di** sont les alternatives de partie droite. Un **Di** est :

- soit un non terminal A, auquel cas la classe d'arbres définie par A est incluse dans celle définie par **G** ;
- soit de la forme `NODE_XXX` ou `NODE_YYY`(**F1**, **F2**, ..., **Fp**), auquel cas `NODE_XXX` est un noeud sans enfant (une feuille) de nature `XXX`, et `NODE_YYY` est un noeud interne de nature `YYY` ayant `p` enfants, dans l'ordre **F1, ..., Fp**. Un **Fi** est un non terminal A, arbre de la classe définie par A.

5.2 Champs des noeuds de l'arbre du programme

Aux noeuds de l'arbre sont associées des informations supplémentaires (des "champs") : tous les noeuds de l'arbre possèdent un champ `lineno` (numéro de ligne du texte correspondant, dans le fichier source, à initialiser avec `yylineno`), un champ `opr` qui est un tableau de pointeurs vers les noeuds enfants, et un champ `nops` (nombre d'enfants, i.e. taille du tableau `opr`). Certains noeuds ont aussi un champ spécifique initialisé lors de la création du noeud. D'autres champs sont également définis et utilisés lors des étapes de vérification contextuelle et de génération de code.

Les champs spécifiques aux noeuds de certaines natures sont les suivants :

- champ `ident` : identifiant, chaine de caractères
  - `NODE_IDENT` : initialisé à la création
- champ `type` : type de l'expression, type énuméré
  - `NODE_TYPE` : initialisé à la création
  - `NODE_IDENT` (occurrence de déclaration) : mis à jour au cours de la passe 1
  - `NODE_IDENT` (occurrence d'utilisation) : mis à jour au cours de la passe 1, à partir du type enregistré dans le `NODE_IDENT` correspondant à la déclaration
  - Noeuds correspondant à des expressions : mis à jour au cours de la passe 1
- champ `value` : entier, valeur du littéral
  - `NODE_INTVAL, NODE_BOOLVAL` : initialisé à la création
- champ `str` : chaine de caractères, valeur du littéral
  - `NODE_STRINGVAL` : initialisé à la création
- champ `global_decl` : variable globale, booléen
  - `NODE_IDENT` (occurrence de déclaration) : mis à jour au cours de la passe 1
- champ `decl_node` : pointeur vers un `NODE_IDENT`, correspondant à la déclaration de la variable
  - `NODE_IDENT` (occurrence d'utilisation) : mis à jour au cours de la passe 1
- champ `offset` : entier, position de la variable en mémoire (en section `.data` ou en pile) pour les `NODE_IDENT` et les `NODE_STRINGVAL` ; taille en pile correspondant à toutes les variables locales pour les `NODE_FUNC`
  - `NODE_IDENT` (occurrence de déclaration) : mis à jour au cours de la passe 1
  - `NODE_STRINGVAL` : mis à jour au cours de la passe 1
  - `NODE_FUNC` : mis à jour au cours de la passe 1, après l'analyse de la fonction

Remarque : dans l'implémentation, il n'y a qu'une sorte de noeud, qui possède donc tous les attributs. La nature d'un noeud est définie par son champ `nature`. Il s'agira de n'utiliser que les attributs pertinents d'un noeud en fonction de sa nature.

5.3 Règles de la grammaire d'arbres

program	→	NODE_PROGRAM(**vardecls**, **main**)	(0.1)
vardecls	→	**decls_list**	(0.2)
	→	NULL	(0.3)
decls_list	→	NODE_LIST(**decls_list**, **decls**)	(0.4)
	→	**decls**	(0.5)
decls	→	NODE_DECLS(**type**, **decl_list**)	(0.6)
decl_list	→	NODE_LIST(**decl_list**, **decl**)	(0.7)
	→	**decl**	(0.8)
decl	→	NODE_DECL(**ident**, **expinit**)	(0.9)
main	→	NODE_FUNC(**type**, **ident**, **block**)	(0.10)
type	→	NODE_TYPE	(0.11)
ident	→	NODE_IDENT	(0.12)
block	→	NODE_BLOCK(**vardecls**, **insts**)	(0.13)
insts	→	**inst_list**	(0.14)
	→	NULL	(0.15)
inst_list	→	NODE_LIST(**inst_list**, **inst**)	(0.16)
	→	**inst**	(0.17)
inst	→	**block**	(0.18)
	→	**exp**	(0.19)
	→	NODE_IF(**exp**, **inst**)	(0.20)
	→	NODE_IF(**exp**, **inst**, **inst**)	(0.21)
	→	NODE_WHILE(**exp**, **inst**)	(0.22)
	→	NODE_DOWHILE(**inst**, **exp**)	(0.23)
	→	NODE_FOR(**exp**, **exp**, **exp**, **inst**)	(0.24)
	→	NODE_PRINT(**printparams_list**)	(0.25)
	→	NULL	(0.26)
printparam_list			(0.27)
	→	NODE_LIST(**printparam_list**, **printparam**)	

	→	**printparam**	(0.28)
printparam	→	**ident**	(0.29)
	→	NODE_STRINGVAL	(0.30)
expinit	→	**exp**	(0.31)
	→	NULL	(0.32)
exp	→	NODE_PLUS(**exp**, **exp**)	(0.33)
	→	NODE_MINUS(**exp**, **exp**)	(0.34)
	→	NODE_MUL(**exp**, **exp**)	(0.35)
	→	NODE_DIV(**exp**, **exp**)	(0.36)
	→	NODE_MOD(**exp**, **exp**)	(0.37)
	→	NODE_UMINUS(**exp**)	(0.38)
	→	NODE_LT(**exp**, **exp**)	(0.39)
	→	NODE_GT(**exp**, **exp**)	(0.40)
	→	NODE_LE(**exp**, **exp**)	(0.41)
	→	NODE_GE(**exp**, **exp**)	(0.42)
	→	NODE_EQ(**exp**, **exp**)	(0.43)
	→	NODE_NE(**exp**, **exp**)	(0.44)
	→	NODE_AND(**exp**, **exp**)	(0.45)
	→	NODE_OR(**exp**, **exp**)	(0.46)
	→	NODE_BAND(**exp**, **exp**)	(0.47)
	→	NODE_BOR(**exp**, **exp**)	(0.48)
	→	NODE_BXOR(**exp**, **exp**)	(0.49)
	→	NODE_SLL(**exp**, **exp**)	(0.50)
	→	NODE_SRL(**exp**, **exp**)	(0.51)
	→	NODE_SRA(**exp**, **exp**)	(0.52)
	→	NODE_NOT(**exp**)	(0.53)
	→	NODE_BNOT(**exp**)	(0.54)
	→	NODE_AFFECT(**ident**, **exp**)	(0.55)
	→	NODE_INTVAL	(0.56)
	→	NODE_BOOLVAL	(0.57)
	→	**ident**	(0.58)

6 Sémantique de MiniC

6.1 Introduction

La sémantique de MiniC n'est pas formellement définie : on se référera à la sémantique des langages de programmation usuels, en particulier du C, pour les constructions non évoquées dans les paragraphes qui suivent.

Un programme *sémantiquement correct* (ou simplement *correct*) est un programme qui respecte les règles de la grammaire attribuée, c'est-à-dire pour lequel la passe de vérification se déroule sans erreur. Un programme non correct est dit *incorrect*.

Un programme correct est dit *erroné* si une erreur peut survenir lors de son exécution, par exemple en cas de division par 0 ou d'accès à une variable non initialisée. Le compilateur est tenu, en l'absence d'options spécifiques, de produire du code assembleur d'un programme correct erroné, même s'il arrive à déterminer qu'une erreur va arriver à l'exécution.

6.2 Initialisation des variables

Une variable globale non initialisée doit être initialisée à la valeur 0 pour un entier, et `false` pour un booléen.

Une variable locale non initialisée ne doit pas être initialisée. Avant qu'elle soit affectée, sa valeur est indéterminée.

Les initialisations doivent avoir lieu dans l'ordre de déclaration des variables.

6.3 Terminaison d'un programme

Pour des raisons de limitation du simulateur, tous les programmes doivent se terminer par l'appel système `exit()` (appel système numéro 10 en mips). Cet appel système doit être effectué au niveau de l'accolade fermante de la fonction `main()`, après l'épilogue. La fonction `main()` doit toujours retourner le type `void`.

6.4 Ordre d'évaluation

Les opérandes des opérations arithmétiques binaires, de comparaison et d'affectation sont évalués *de gauche à droite*.

Attention : cela est différent du C : il n'y a pas de point de séquence, ni de comportement indéfini, puisque l'ordre d'évaluation est parfaitement défini. La sémantique des expressions est donc la même que celle des expressions en Java (hormis point suivant).

Les expressions booléennes sont évaluées non-paresseusement de gauche à droite. Cela signifie que lorsqu'on évalue `C1 && C2`, on évalue `C1`, puis `C2` même si `C1` est fausse. De même, lorsque l'on évalue `C1 || C2`, on évalue d'abord `C1`, puis `C2` même si `C1` est vraie.

6.5 Taille des entiers et débordements lors de l'évaluation des expressions

Les entiers réprésentables du langage sont ceux codables sur 32 bits. Lors de l'analyse lexicale, seuls des entiers positifs peuvent être retournés. De ce fait, l'intervalle des entiers pouvant être reconnus sans erreurs lors de cette analyse est l'intervalle $[0\,;\,2^{32}\text{ - }1 = 4294967295]$. En principe, l'analyse lexicale ne permet pas de discriminer entre un entier représentable et un entier non représentable. Cependant, comme la conversion des caractères en entier est faite lors de cette analyse, l'erreur correspondante – sémantique – sera malgré tout levée durant cette phase.

Remarque : chaque mot ayant le bit de poids fort à 1 représente deux nombres ; par exemple, le mot 0x80000000 représente les nombres -2^{31} et 2^{31}, et le mot 0xFFFFFFFF les valeurs -1 et 2^{31} - 1. De plus, comme toutes les comparaisons sont signées, on a par exemple que $-2 > 3000000000$.

Une division entière par 0 ou un calcul modulo 0 doit provoquer une erreur. L'instruction mips `div` ne générant pas d'exception, celle-ci doit être testée logiciellement ("à la main") à l'aide de l'instruction `teq` (*trap if equal*).

Il n'y a pas de débordement pour les opérations d'addition, de soustraction et de décalage sur les entiers : les calculs sont fait modulo 2^{32} ; les décalages à droite sont arithmétiques avec l'opérateur `>>` (la valeur des bits injectés est la valeur du bit de poids fort avant injection – instruction mips `sra`) et logiques avec l'opérateur `>>>` (la valeur des bits injectés est 0 – instruction mips `srl`).

6.6 Procédures d'affichage

Un appel à `print(e);` écrit sur la sortie standard :

- la valeur de la variable `e` si `e` est une variable ; pour les variables booléennes, la valeur affichée doit être 0 pour `false` et 1 pour `true`
- la chaine de caractères `e` s'il s'agit d'une chaine de caractères littérale

`print(e1, e2, ..., en);` est équivalent à `print(e1); print(e2); ... ; print(en);`.

6.7 Catégories des erreurs à l'exécution

A priori, les seules erreurs qui peuvent survenir à l'exécution (c'est-à-dire lors de la simulation du programme assembleur) sont les suivantes :

- Programmes corrects erronés dont le code est généré :
  - Divison par 0 ou calcul modulo 0 : exception logicielle avec test dynamique (utiliser l'instruction mips `teq`)
  - Accès à des variables locales non initialisées : comportement indéfini
- Programmes corrects non erronés mais pour lesquels le code généré comporte une erreur (erreur dans le code généré par le compilateur) :
  - Lecture ou écriture non alignée
  - Lecture ou écriture dans un segment non autorisé
  - Format de l'instruction incorrect ou instruction inexistante (devrait être limité avec l'utilisation de la bilbiothèque fournie)
  - ...
- Programmes corrects non erronés qui dépassent les capacités de la machine :
  - Débordement de pile (se traduit par un accès dans un segment non autorisé)

7 Grammaire attribuée de MiniC

7.1 Introduction

La vérification contextuelle d'un programme MiniC peut être faite en une seule passe. En effet, ce langage (tout comme le C) ne contient pas, à un endroit donné d'un programme, de référence à un identificateur qui est défini plus loin dans le programme, ce qui nécéssiterait plusieurs passes. En C, si une fonction f() appelle une fonction g() définie plus tard, la fonction g() doit être pré-déclarée avant f().

Les vérifications à effectuer lors de la passe de vérification sont spécifiées formellement à l'aide d'une grammaire attribuée.

Remarque : certaines règles diffèrent de celles de la grammaire hors-contexte du langage (règles récursives notamment) car il s'agit uniquement d'une spécification, et non d'une grammaire qui doit être implantée.

7.2 Domaines d'attributs

Dans cette partie sont définis les domaines d'attributs et les opérations sur les attributs.

7.2.1 Définition des domaines

Soit Nom le domaine des identificateurs, et Type le domaine des types du langage MiniC. Les types du langage MiniC sont void, bool et int.

$$\text{Type} = \{\underline{\text{void}}, \underline{\text{bool}}, \underline{\text{int}}\}$$

Dans le langage MiniC, les identificateurs sont tous des identificateurs de variables. Cela est une spécificité du langage, car dans un langage comme Java, il y a des identificateurs de type (enum), de champ ou attribut de classe, de paramètre, de variable, de classe et de méthode.
Opérateur est l'ensemble des opérateurs du langage.

$$\begin{aligned}\text{Opérateur} = \{&\underline{\text{plus}}, \underline{\text{minus}}, \underline{\text{mul}}, \underline{\text{div}}, \underline{\text{mod}}, \underline{\text{eq}}, \underline{\text{ne}}, \underline{\text{lt}}, \underline{\text{gt}}, \underline{\text{le}}, \underline{\text{ge}}, \underline{\text{and}}, \underline{\text{or}}, \underline{\text{bxor}}, \underline{\text{band}}, \underline{\text{bor}},\\ &\underline{\text{not}}, \underline{\text{bnot}}, \underline{\text{sll}}, \underline{\text{srl}}, \underline{\text{sra}}\}\end{aligned}$$

7.2.2 Opérations sur les domaines d'attributs

Compatibilité pour l'affectation

Contrairement au C, le langage MiniC fait une distinction nette entre le type entier et le type booléen. Ainsi, il n'est pas possible d'affecter une expression de type booléenne dans une variable de type entier, et une expression de type entière dans une variable de type booléenne. De même, les conditions doivent retourner une expression de type booléenne.

Signature des opérateurs

On définit deux opérations : type_op_unaire et type_op_binaire, qui permettent de calculer respectivement le type du résultat d'un opérateur unaire et d'un opérateur arithmétique binaire.

type_op_unaire : Opérateur × Type → Type

$$\begin{aligned}\text{type\_op\_unaire}(\underline{\text{minus}}, \underline{\text{int}}) &= \underline{\text{int}}\\ \text{type\_op\_unaire}(\underline{\text{bnot}}, \underline{\text{int}}) &= \underline{\text{int}}\\ \text{type\_op\_unaire}(\underline{\text{not}}, \underline{\text{bool}}) &= \underline{\text{bool}}\end{aligned}$$

type_op_binaire : Opérateur × Type × Type → Type

$$\text{type\_op\_binaire}(op, \underline{\text{int}}, \underline{\text{int}}) = \underline{\text{int}},$$
$$\text{si } op \in \{\underline{\text{plus}}, \underline{\text{minus}}, \underline{\text{mul}}, \underline{\text{div}}, \underline{\text{mod}}, \underline{\text{band}}, \underline{\text{bor}}, \underline{\text{bxor}}, \underline{\text{sll}}, \underline{\text{srl}}, \underline{\text{sra}}\}$$

type_op_binaire(op, $\underline{\text{int}}$, $\underline{\text{int}}$) = $\underline{\text{bool}}$,
si $op \in \{\underline{\text{eq}}, \underline{\text{ne}}, \underline{\text{lt}}, \underline{\text{gt}}, \underline{\text{le}}, \underline{\text{ge}}\}$

type_op_binaire(op, $\underline{\text{bool}}$, $\underline{\text{bool}}$) = $\underline{\text{bool}}$,
si $op \in \{\underline{\text{and}}, \underline{\text{or}}, \underline{\text{eq}}, \underline{\text{ne}}\}$

7.2.3 Contextes et environnements

Un contexte associe à un identificateur la déclaration de la variable correspondante. Dans le cadre de la vérification contextuelle, la seule information pertinente associée à une variable est son type, c'est pourquoi un contexte associe à un nom de variable un type. Au sein d'un contexte, il ne peut donc pas y avoir deux variables avec le même nom. Un environnement correspond à l'ensemble des variables accessibles depuis un endroit du programme. Un environnement est créé par un empilement de contextes, noté /, au sein duquel la définition la plus récente d'une variable masque les définitions plus anciennes. L'empilement est défini formellement de la façon suivante :

- / : Contexte × Environnement → Environnement
  $\forall x \in \text{Nom},\ (ctx/env)(x) = ctx(x),$ si $x \in \text{dom}(ctx)$,
  $= env(x),$ si $x \notin \text{dom}(ctx)$ et $x \in \text{dom}(env)$.

L'environnement constitué d'un seul contexte ctx est noté $Env(ctx)$.

7.3 Conventions d'écriture

On utilise les notations suivantes :
- les parties hors-contexte des règles sont en gras ;
- les terminaux de la grammaire, autres que les symboles spéciaux, sont soulignés ;
- les attributs synthétisés sont préfixés par ↑ ;
- les attributs hérités sont préfixés par ↓.

7.3.1 Affectation des attributs

Pour toute règle, les attributs synthétisés du non terminal en partie gauche et les attributs hérités des non terminaux en partie droite doivent être affectés. Ces affectations peuvent être effectuées de deux manières différentes : 1. explicitement en utilisant une clause ***affectation*** ; 2. implicitement par une expression fonctionnelle.

- Affectation explicite de la forme ***affectation*** $v := exp$. Par exemple, la règle (1.61)

  ident ↓env ↑$type$ → **idf** ↑nom
  affectation $type := env(nom)$

  signifie qu'à l'attribut synthétisé ↑$type$ du non terminal **ident** est affecté la valeur $env(nom)$.

- Affectation implicite par une expression fonctionnelle. Par exemple, dans la règle (1.5) :

  main_declaration ↓env
  → **type** ↑$type$ **idf** ↑nom `'(' ')'` **bloc** ↓env

  L'attribut hérité du non terminal **bloc** est directement affecté avec la valeur de l'attribut hérité du non terminal **main_declaration** : env.

7.3.2 Conditions sur les attributs

Les valeurs d'attributs, pour une règle de grammaire, peuvent être contraintes. Ces contraintes peuvent être exprimées de 2 manières différentes : 1. explicitement par une condition logique sur les valeurs d'attributs ; 2. implicitement en contraignant par filtrage les valeurs possibles d'attributs.

- Utilisation d'une clause ***condition*** P, où P est une condition logique. Si P est faux, la clause n'est pas respectée. Par exemple, dans la règle (1.5)

main_declaration $\downarrow env$
$\rightarrow$ **type** $\uparrow type$ **idf** $\uparrow nom$ `'(' ')'` **bloc** $\downarrow env$
condition $nom =$ "*main*"
condition $type =$ void

les deux conditions imposent que le nom de l'identifiant de l'unique fonction soit `main`, et que le type de retour de l'unique fonction soit void.

- Par filtrage : on impose une forme particulière pour un attribut hérité dans une partie gauche de règle, ou pour un attribut synthétisé pour une partie droite de règle. Par exemple, la règle (1.20)

  inst $\downarrow env$ $\rightarrow$ **while** `'('` **exp** $\downarrow env$ $\uparrow$bool `')'` **bloc** $\downarrow env$

  impose que la valeur de l'attribut synthétisé de **exp** soit le type boolean.

7.3.3 Abréviation pour les valeurs de domaines

Dans certaines règles, certaines valeurs de domaines ne sont pas contraintes et n'ont pas d'utilité pour la règle (elles ne servent ni au calcul de valeur d'attribut hérité en partie droite ou synthétisé en partie gauche, ni dans l'expression d'une affectation ou d'une contrainte portant sur une autre valeur). Dans ce cas, on remplace ce nom par un "tiret bas" '__', de façon à bien mettre en évidence que la valeur correspondante n'est pas utilisée ni contrainte.
Par exemple, la règle (1.29) pourrait s'écrire :

param_print $\downarrow env$
$\rightarrow$ **ident** $\downarrow env$ $\uparrow type$

en introduisant un nom inutile (*type*).

7.4 Grammaire attribuée spécifiant la passe de vérification

7.4.1 Type

Le type void n'est utilisé que pour le type de retour de la fonction main().

type $\uparrow$int $\rightarrow$ **int** (1.1)

type $\uparrow$bool $\rightarrow$ **bool** (1.2)

type $\uparrow$void $\rightarrow$ **void** (1.3)

7.4.2 Programme

programme $\rightarrow$ **liste_decl_vars** $\downarrow$__ $\downarrow\{\}$ $\downarrow true$ $\uparrow ctx$ (1.4)
main_declaration $\downarrow Env(ctx)$

Le premier attribut hérité de **liste_decl_vars**, l'environnement des variables pour l'analyse des expressions, n'est pas affecté car les variables globales ne peuvent pas être initialisées à partir d'expressions. Il y a seulement besoin d'initialiser un contexte global vide.

main_declaration $\downarrow env$ (1.5)
$\rightarrow$ **type** $\uparrow type$ **idf** $\uparrow nom$ `'(' ')'` **bloc** $\downarrow env$
condition $nom =$ "*main*"
condition $type =$ void

7.4.3 Déclaration de variables

Les variables globales ne peuvent être initialisées qu'avec des constantes littérales, tandis que les variables locales aux blocs peuvent être initialisées avec des expressions. L'attribut *global* dans les différentes règles se réfère au fait qu'il s'agisse d'une déclaration de variable globale. Cet attribut est passé avec la valeur *true* pour les déclarations de variables globales (règle (1.4)), et à $false$ pour les variables locales aux blocs (règle (1.14)). L'attribut *env* passé dans les différentes règles correspond à l'empilement des contextes à l'entrée du contexte courant. Il est vide pour la déclaration des variables globales. L'attribut *ctx* correspond quant à lui au contexte courant, créé à l'entrée du bloc.

liste_decl_vars $\downarrow env\ \downarrow ctx_0\ \downarrow global\ \uparrow ctx$ (1.6)
$\rightarrow$ **liste_decl_vars** $\downarrow env\ \downarrow ctx_0\ \downarrow global\ \uparrow ctx_1$
decl_vars $\downarrow env\ \downarrow ctx_1\ \downarrow global\ \uparrow ctx$

liste_decl_vars $\downarrow env\ \downarrow ctx\ \downarrow global\ \uparrow ctx$ (1.7)
$\rightarrow\ \varepsilon$

decl_vars $\downarrow env\ \downarrow ctx_0\ \downarrow global\ \uparrow ctx$ (1.8)
$\rightarrow$ **type** $\uparrow type$
liste_decl_type $\downarrow env\ \downarrow ctx_0\ \downarrow type\ \downarrow global\ \uparrow ctx$ `';'`
condition $type \neq \underline{\text{void}}$

liste_decl_type $\downarrow env\ \downarrow ctx_0\ \downarrow type\ \downarrow global\ \uparrow ctx$ (1.9)
$\rightarrow$ **liste_decl_type** $\downarrow env\ \downarrow ctx_0\ \downarrow type\ \downarrow global\ \uparrow ctx_1$ `','`
decl_var $\downarrow env\ \downarrow ctx_1\ \downarrow type\ \downarrow global\ \uparrow ctx$
$\rightarrow$ **decl_var** $\downarrow env\ \downarrow ctx_0\ \downarrow type\ \downarrow global\ \uparrow ctx$ (1.10)

decl_var $\downarrow env\ \downarrow ctx_0\ \downarrow type\ \downarrow global\ \uparrow ctx$ (1.11)
$\rightarrow$ **idf** $\uparrow nom$
condition $nom \notin \text{dom}(ctx_0)$
affectation $ctx := ctx_0 \cup \{nom \mapsto type\}$

$\rightarrow$ **idf** $\uparrow nom$ `'='` **litteral** $\uparrow type_1$ (1.12)
condition $global = true$ et $nom \notin \text{dom}(ctx_0)$ et $type = type_1$
affectation $ctx := ctx_0 \cup \{nom \mapsto type\}$

$\rightarrow$ **idf** $\uparrow nom$ `'='` **exp** $\downarrow ctx_0/env\ \uparrow type_1$ (1.13)
condition $global = false$ et $nom \notin \text{dom}(ctx_0)$ et $type = type_1$
affectation $ctx := ctx_0 \cup \{nom \mapsto type\}$

Pour analyser l'expression d'initialisation, l'environnement affecté à l'attribut est l'empilement du contexte courant avec l'environnement englobant. En effet, l'expression d'initialisation peut référencer des variables déclarées précédemment dans le même bloc et des variables déclarées dans un bloc englobant.

7.4.4 Bloc

bloc $\downarrow env$ $\rightarrow$ `'{'` **liste_decl_vars** $\downarrow env\ \downarrow\{\}\ \downarrow false\ \uparrow ctx$ **liste_inst** $\downarrow ctx/env$ `'}'` (1.14)

L'environnement considéré pour analyser les expressions du bloc est l'empilement du contexte du bloc sur l'environnement englobant.

7.4.5 Instructions

liste_inst $\downarrow env$ $\rightarrow$ **liste_inst** $\downarrow env$ **inst** $\downarrow env$ (1.15)
$\rightarrow\ \varepsilon$ (1.16)

Toutes les expressions apparaissant dans des conditions doivent avoir le type $\underline{\text{bool}}$.

inst $\downarrow env$ $\rightarrow$ **exp** $\downarrow env\ \uparrow$__ `';'` (1.17)
$\rightarrow$ **if** `'('` **exp** $\downarrow env\ \uparrow\underline{\text{bool}}$ `')'` **inst** $\downarrow env$ (1.18)
$\rightarrow$ **if** `'('` **exp** $\downarrow env\ \uparrow\underline{\text{bool}}$ `')'` **inst** $\downarrow env$ **else** **inst** $\downarrow env$ (1.19)
$\rightarrow$ **while** `'('` **exp** $\downarrow env\ \uparrow\underline{\text{bool}}$ `')'` **inst** $\downarrow env$ (1.20)
$\rightarrow$ **for** `'('` **exp** $\downarrow env\ \uparrow$__ `';'` **exp** $\downarrow env\ \uparrow\underline{\text{bool}}$ `';'` **exp** $\downarrow env\ \uparrow$__ `')'` (1.21)
inst $\downarrow env$
$\rightarrow$ **do** **inst** $\downarrow env$ **while** `'('` **exp** $\downarrow env\ \uparrow\underline{\text{bool}}$ `')'` `';'` (1.22)

→ **bloc** ↓*env* (1.23)

→ **print** `'('` **liste_param_print** ↓*env* `')'` (1.24)

→ `';'` (1.25)

liste_param_print ↓*env* (1.26)

→ **liste_param_print** ↓*env* `','` **param_print** ↓*env*

→ **param_print** ↓*env* (1.27)

param_print ↓*env* (1.28)

→ **chaine**

→ **ident** ↓*env* ↑__ (1.29)

7.4.6 Expressions

exp ↓*env* ↑*type* → **exp** ↓*env* ↑$type_0$ **op_bin** ↑*op* **exp** ↓*env* ↑$type_1$ (1.30)
affectation *type* := type_op_binaire($op, type_0, type_1$)

→ **op_un** ↑*op* **exp** ↓*env* ↑*type* (1.31)
affectation *type* := type_op_unaire(*op*, *type*)

→ **ident** ↓*env* ↑$type_0$ `'='` **exp** ↓*env* ↑$type_1$ (1.32)
condition $type_0 = type_1$
affectation *type* := $type_0$

→ **ident** ↓*env* ↑$type_0$ (1.33)
affectation *type* := $type_0$

→ `'('` **exp** ↓*env* ↑$type_0$ `')'` (1.34)
affectation *type* := $type_0$

→ **litteral** ↑$type_0$ (1.35)
affectation *type* := $type_0$

op_bin ↑plus → `'+'` (1.36)

op_bin ↑minus → `'-'` (1.37)

op_bin ↑mul → `'*'` (1.38)

op_bin ↑div → `'/'` (1.39)

op_bin ↑mod → `'%'` (1.40)

op_bin ↑sll → `'<<'` (1.41)

op_bin ↑srl → `'>>>'` (1.42)

op_bin ↑sra → `'>>'` (1.43)

op_bin ↑gt → `'>'` (1.44)

op_bin ↑lt → `'<'` (1.45)

op_bin ↑ge → `'>='` (1.46)

op_bin ↑le → `'<='` (1.47)

op_bin ↑band → `'&'` (1.48)

op_bin ↑bor → `'|'` (1.49)

op_bin ↑bxor → `'^'` (1.50)

op_bin ↑eq → `'=='` (1.51)

op_bin ↑ne → `'!='` (1.52)

op_bin ↑and → `'&&'` (1.53)

op_bin ↑or → `'||'` (1.54)

op_un ↑uminus → `'-'` (1.55)

op_un ↑bnot → `'~'` (1.56)

op_un ↑not → `'!'` (1.57)

litteral ↑int → **entier** (1.58)

litteral ↑bool → **true** (1.59)

litteral ↑bool → **false** (1.60)

7.4.7 Identificateur

ident ↓*env* ↑*type* → **idf** ↑*nom* (1.61)
condition $nom \in \mathrm{dom}(env)$
affectation $type := env(nom)$

On doit trouver une définition associée au nom *nom* dans l'environnement *env*.

7.5 Profils d'attributs des symboles non terminaux et terminaux

7.5.1 Type

type ↑Type

7.5.2 Programme

liste_decl_vars ↓Environnement ↓Contexte ↓Bool ↑Contexte
main_declaration ↓Environnement

7.5.3 Déclaration de variables

liste_decl_vars ↓Environnement ↓Contexte ↓Bool ↑Contexte
decl_vars ↓Environnement ↓Contexte ↓Bool ↑Contexte
liste_decl_type ↓Environnement ↓Contexte ↓Type ↓Bool ↑Contexte
decl_var ↓Environnement ↓Contexte ↓Type ↓Bool ↑Contexte

7.5.4 Instructions

bloc ↓Environnement

7.5.5 Instructions

liste_inst ↓Environnement
inst ↓Environnement
liste_param_print ↓Environnement
param_print ↓Environnement

7.5.6 Expressions

exp ↓Environnement ↑Type
op_bin ↑Opérateur
op_un ↑Opérateur
litteral ↑Type

7.5.7 Identificateur

ident ↓Environnement ↑Type
idf ↑Nom

7.6 Implémentation de l'environnement

Dans cette partie, on montre sur un exemple comment les environnements peuvent être implémentés.

Un environnement est une liste chaînée de contextes, qui sont des tables d'associations identificateur ↦ définition. La définition correspond au noeud de nature `NODE_IDENT` associé à la déclaration dans l'arbre du programme. Considérons le programme MiniC suivant :

```
1     int a = 0;
2     int b = 0;
3
4     void main() {
5        int a = 1;
6        int c = 2;
7
8        if (true) {
9           int a = 5;
10          int d = 6;
11          a = a + b + c + d;
12       }
13       else {
14          int d;
15          int e;
16          e = d = 1;
17       }
18    }
```

La figure 4 montre l'environnement d'analyse de différentes parties du programme.
(a) Le contexte global `C0` est empilé dans l'environnement. Les variables `a` et `b` y sont ajoutées lorsque l'on rencontre leur déclaration durant le parcours de l'arbre du programme : les définitions associées aux noms enregistrées dans le contexte sont les noeuds de l'arbre de nature `IDENT` correspondant à la déclaration de ces variables.

(b) Au début de l'analyse du `main()` le contexte `C1` est empilé dans l'environnement au dessus de `C0`. Les définitions des variables locales `a` et `c` y sont ajoutées. On remarque que la définition de la variable locale a masque la définition de la variable globale `a`.

(c) Au début de l'analyse du bloc `then`, le contexte `C2` est empilé dans l'environnement au dessus de `C1`. Les définitions des variables locales `a` et `d` y sont ajoutées. Lors de l'analyse de l'expression `a = a + b + c + d;`, on commence par chercher dans l'environnement la définition de l'occurrence de `a` à droite du `=`. Une définition pour `a` est trouvée dans l'environnement `C2`, et permet de générer l'instruction de lecture de `a` à partir de l'offset enregistré dans le noeud. On cherche ensuite la définition de `b` dans l'environnement. Puisqu'une aucune définition de `b` n'est trouvée dans `C2`, on en recherche une dans `C1`. Puisqu'aucune définition de `b` n'est trouvée dans `C1`, on la recherche dans `C0`, où elle est trouvée. De même, les variables `c` et `d` sont trouvées respectivement dans `C1` et `C2`, puis la définition de l'occurrence de `a` à gauche du `=` est trouvée dans `C2`. À la fin du bloc, le contexte `C2` est dépilé de l'environnement.

(d) Au début de l'analyse du bloc `else`, le contexte `C3` est empilé dans l'environnement au dessus de `C1`. Les définitions des variables locales `d` et `e` y sont ajoutées. Lors de l'analyse de l'expression `e = d = 1;` les définitions de `d` et `e` sont trouvées dans `C3`.

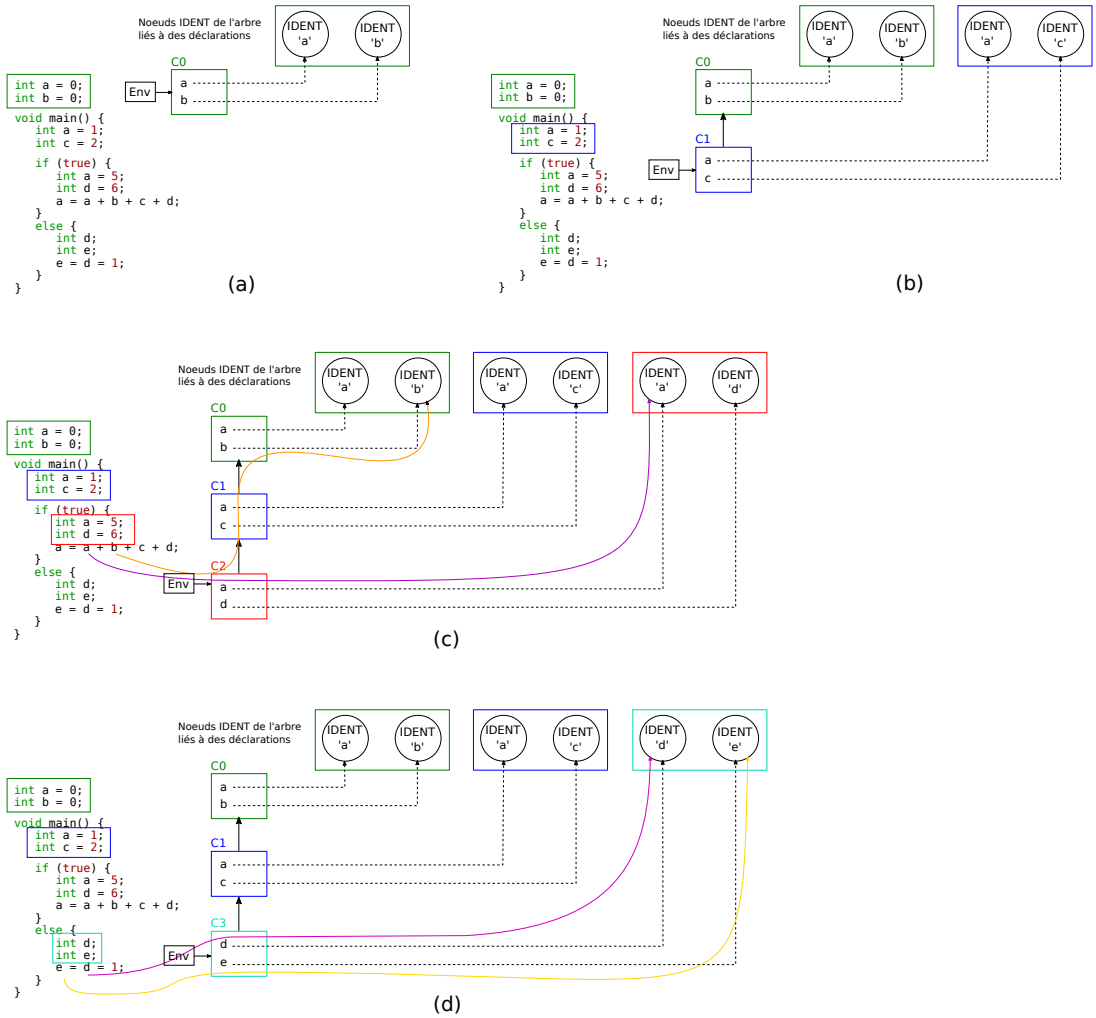


FIGURE 4 – Environnement d'analyse des différents blocs du programme

8 MiniCC : Spécification du compilateur

8.1 Ligne de commande

Le programme principal, `minicc`, est un compilateur MiniC complet. Cette section décrit les arguments de la ligne de commande qui doivent être supportés par `minicc`. Les arguments de la ligne de commande feront l'objet de tests spécifiques pour l'évaluation.

On permettra de désigner le fichier d'entrée par des chemins de la forme `<répertoires/nom.c>`. Le nom du fichier à compiler doit être compris comme le premier argument de la ligne de commande qui ne soit ni une option, ni une valeur d'option. Cet argument ne doit être défini qu'une seule fois. Sauf erreur dans le programme d'entrée, le résultat doit être par défaut dans un fichier `<out.s>` situé dans le répertoire courant (et non pas dans le répertoire du fichier source).

La commande `minicc`, sans argument, affichera les options disponibles. On définira les options suivantes à la commande `minicc` [1].

- `-b` : Affiche une bannière indiquant le nom du compilateur et des membres du binôme
- `-o <filename>` : Définit le nom du fichier assembleur produit (défaut : `out.s`).

1. Pour l'implémentation des options, il est conseillé de ne pas utiliser la fonction `getopt()`, qui n'est pas très adaptée à cette spécification

- `-t <int>` : Définit le niveau de trace à utiliser entre 0 et 5 (0 = pas de trace ; 5 = toutes les traces. defaut = 0).
- `-r <int>` : Définit le nombre maximum de registres à utiliser, entre 4 et 8 (défaut : 8).
- `-s` : Arrêter la compilation après l'analyse syntaxique (défaut = non).
- `-v` : Arrêter la compilation après la passe de vérifications (défaut = non).
- `-h` : Afficher la liste des options (fonction d'usage) et arrêter le parsing des arguments.

Remarque : les options `'-s'` et `'-v'` sont incompatibles.

En l'absence des options `'-b'`, `'-h'`, et `'-t <n>'` avec `n` $\neq$ `0`, une exécution de `minicc` ne doit produire aucun affichage si la compilation réussit. Il est impératif de respecter les conventions sur les arguments de la ligne de commande, car les compilateurs rendus seront testés automatiquement à l'aide de scripts à la fin du projet.

L'option `-b` ne peut être utilisée que sans autre option, et sans fichier source. Dans ce cas, `minicc` termine après avoir affiché la bannière.

En cas d'erreur dans la ligne de commande, le programme devra retourner un code d'erreur (valeur de retour ou du paramètre de `exit()` différente de 0), sauf si l'option `-h` est rencontrée avant que l'erreur ne soit détectée. Si la ligne de commande est correcte, la valeur de retour de `minicc` devra être 0. En bash, la valeur de retour du dernier programme lancé est stockée dans la variable `$?`.

Exemples de lignes de commandes correctes :

- `./minicc -h`
- `./minicc -b`
- `./minicc fichier.c`
- `./minicc fichier.c -o fichier.s`
- `./minicc -o fichier.s fichier.c`
- `./minicc -o fichier.s -t 0 fichier.c -r 6`
- `./minicc -o fichier.s -v test.c`
- `./minicc -s test.c`

Exemples de lignes de commandes incorrectes :

- `./minicc -b fichier.c`
- `./minicc fichier_1.c fichier_2.c`
- `./minicc -s -v fichier.c`
- `./minicc -t -r 4 fichier.c`
- `./minicc fichier_1.c -o fichier.s fichier_2.c`
- `./minicc -r 2 fichier.c`
- `./minicc -t 6 fichier.c`
- `./minicc -t 0 -r 8 -o fichier.s`

8.2 Formattage des messages d'erreur

Les messages d'erreur (lexicales, syntaxiques, contextuelles, et éventuelles limitations du compilateur) doivent être formatées de la manière suivante (cette règle est également indispensable pour l'évaluation automatique de votre compilateur par les enseignants) :

```
Error line <numéro de ligne>: <description informelle du problème>
```

Comme par exemple :

```
Error line 12: variable "foobar" undeclared (rule 1.4)
```

ou bien :

```
Error line 3: Syntax error
```

Il est indispensable d'afficher un numéro de ligne correct et selon ce format car les scripts d'évaluation vérifieront ce numéro.

II. Ressources et environnement de développement

I Philosophie générale et vue globale du travail à réaliser

L'objectif de ce projet est de toucher à tous les aspects d'un compilateur. En ce sens, il vous est demandé de réaliser la quasi-totalité du code. Néanmoins, suite à des retours faisant état d'une longueur trop importante pour ce projet, un certain nombre de tâches sont annexes. Pour ces tâches, une interface ainsi qu'une version compilée de son implémentation vous seront fournies et pourront être utilisées sans pénalité. Néanmoins, toute tâche annexe correctement réalisée sera prise en compte au niveau de la notation. Dans ce dernier cas, notez que les interfaces fournies ne sont qu'une façon possible de faire et que vous n'êtes obligés de conserver le même découpage en fonctions. Enfin, les binaires fournis seront compilés pour Linux.

Le travail à réaliser peut se décomposer grossièrement selon les tâches/modules suivants :

- Compléter l'écriture du fichier `lexico.l` décrivant la lexicographie du langage
- Compléter l'écriture du fichier `grammar.y` réalisant l'analyse syntaxique du langage et la construction de l'arbre du programme
- Module implémentant l'analyse des arguments de la ligne de commande
- Module implémentant un contexte, c'est-à-dire l'association entre un nom de symbole et un noeud de l'arbre (annexe)
- Module implémentant un environnement, réalisant l'empilement et le dépilement des contextes en fonction des blocs du programme (annexe)
- Module implémentant un allocateur de registres, définissant les registres source et destination à utiliser pour une instruction (annexe)
- Première passe réalisant les vérifications contextuelles
- Deuxième passe réalisant la génération du code assembleur

Note : il n'est normalement pas nécessaire de faire plus de 2 passes sur le programme (c'est-à-dire 2 parcours de l'arbre). Si vous souhaitez en faire plus, il conviendra alors de justifier chaque passe.

La librairie fournie, nommée `miniccutils` (fichiers `libminiccutils.a` et `miniccutils.h`) regroupe toutes les fonctions pour les modules de contexte, d'environnement, et d'allocation de registre. Elle contient également les fonctions relatives à la création des instruction mips et du programme assembleur, qui ne sont pas à refaire. Enfin, pour vous aider, elle contient une fonction permettant de vérifier qu'un arbre de programme construit au cours de l'analyse syntaxique est valide.

II Ressources et code fourni

1 Fichier `defs.h`

Le fichier `defs.h` contient la définition du type `node_t`, ainsi que les enums `node_nature` et `node_type`. Le type `node_t` est expliqué dans le document de spécifications, dans la partie décrivant la grammaire d'arbre.

Les enums `node_nature` et `node_type` définissent respectivement les différentes nature possibles pour un noeud, et le type des expressions possibles pour un programme.

2 Fichiers `arch.h` et `arch.c`

Les fichiers `arch.[ch]` contiennent des fonctions implémentant des constantes de l'architecture Mips concernant certains registres ou adresses. Les registres disponibles dans l'architecture sont considérés être les registres $8 à $15, de manière à ne pas avoir à gérer la sauvegarde et la restauration des registres persistants. Cela laisse donc 8 registres au maximum (`num_arch_registers`). La fonction `get_num_registers()` retourne le nombre de registres disponibles pour le code à générer,

qui peut être différent si l'option `-r` a été utilisée sur la ligne de commande (dans ce cas, c'est à vous d'appeler la fonction `set_max_registers()` lors de l'analyse des arguments).

3 Bibliothèque de création des programmes mips

Les fonctions de la librairie miniccutils servent à la création des différents type d'instruction mips, mais aussi des directives utiles pour ce projet. Chaque instruction ou directive créée est automatiquement ajoutée à la fin du programme courant.

Voici la liste de toutes les fonctions fournies pour la création de directives et d'instructions :

- `void create_data_sec_inst();`
- `void create_text_sec_inst();`
- `void create_word_inst(char * label, int32_t init_value);`
- `void create_asciiz_inst(char * label_str, char * str);`
- `void create_label_inst(int32_t label);`
- `void create_comment_inst(char * comment);`
- `void create_lui_inst(int32_t r_dest, int32_t imm);`
- `void create_addu_inst(int32_t r_dest, int32_t r_src_1, int32_t r_src_2);`
- `void create_subu_inst(int32_t r_dest, int32_t r_src_1, int32_t r_src_2);`
- `void create_slt_inst(int32_t r_dest, int32_t r_src_1, int32_t r_src_2);`
- `void create_sltu_inst(int32_t r_dest, int32_t r_src_1, int32_t r_src_2);`
- `void create_and_inst(int32_t r_dest, int32_t r_src_1, int32_t r_src_2);`
- `void create_or_inst(int32_t r_dest, int32_t r_src_1, int32_t r_src_2);`
- `void create_xor_inst(int32_t r_dest, int32_t r_src_1, int32_t r_src_2);`
- `void create_nor_inst(int32_t r_dest, int32_t r_src_1, int32_t r_src_2);`
- `void create_mult_inst(int32_t r_src_1, int32_t r_src_2);`
- `void create_div_inst(int32_t r_src_1, int32_t r_src_2);`
- `void create_sllv_inst(int32_t r_dest, int32_t r_src_1, int32_t r_src_2);`
- `void create_srlv_inst(int32_t r_dest, int32_t r_src_1, int32_t r_src_2);`
- `void create_srav_inst(int32_t r_dest, int32_t r_src_1, int32_t r_src_2);`
- `void create_addiu_inst(int32_t r_dest, int32_t r_src_1, int32_t imm);`
- `void create_andi_inst(int32_t r_dest, int32_t r_src_1, int32_t imm);`
- `void create_ori_inst(int32_t r_dest, int32_t r_src_1, int32_t imm);`
- `void create_xori_inst(int32_t r_dest, int32_t r_src_1, int32_t imm);`
- `void create_slti_inst(int32_t r_dest, int32_t r_src_1, int32_t imm);`
- `void create_sltiu_inst(int32_t r_dest, int32_t r_src_1, int32_t imm);`
- `void create_lw_inst(int32_t r_dest, int32_t imm, int32_t r_src_1);`
- `void create_sw_inst(int32_t r_src_1, int32_t imm, int32_t r_src_2);`
- `void create_beq_inst(int32_t r_src_1, int32_t r_src_2, int32_t label);`
- `void create_bne_inst(int32_t r_src_1, int32_t r_src_2, int32_t label);`
- `void create_mflo_inst(int32_t r_dest);`
- `void create_mfhi_inst(int32_t r_dest);`
- `void create_j_inst(int32_t label);`
- `void create_teq_inst(int32_t r_src_1, int32_t r_src_2);`
- `void create_syscall_inst();`
- `void create_stack_allocation_inst();`
- `void create_stack_deallocation_inst(int32_t val);`

Remarques :

- Pour toutes ces fonctions, les paramètres sont les opérandes de l'instruction dans l'ordre. Quand l'opérande est un registre, le paramètre est le numéro entier du registre (exemple : 8 pour r8).
- Pour les labels correspondant à un point du programme, un entier identifiant le label de manière unique doit être passé (c'est à vous de gérer les numéros de ces labels). Un nom

de label générique est généré à partir du numéro (`'_L<num>'`), et ne peut pas entrer en conflit avec des noms de variables ou de fonctions (car celles-ci ne peuvent pas commencer par `'_'`.

- Pour les labels correspondant à des directives de déclaration de variables (`.word`) et des chaines de caractères (`.asciiz`), une valeur non nulle du paramètre `label` génère le label correspondant suivi de `':'` avant la directive. À titre d'exemple :
  - l'appel `create_word_inst("a", 5);` génère le code `a: .word 5`
  - l'appel `create_word_inst(null, 5);` génère le code `.word 5`
  - l'appel `create_asciiz_inst("chaine", "hello");` génère le code `chaine: .asciiz "hello"`
  - l'appel `create_asciiz_inst(null, "hello");` génère le code `.asciiz "hello"`

  Néanmoins, le paramètre de nom (premier paramètre) ne peut être utilisé qu'à des fins de débug, car la simulation du code sur mars doit être faite avec les macros instructions désactivées, et il n'est pas possible d'autoriser seulement certaines macros comme `la`. Il faut donc stocker l'offset de la variable dans la section (dans le champ `offset` du noeud de l'arbre correspondant) afin de pouvoir calculer l'adresse à chaque lecture.
- La fonction `void create_stack_allocation_inst()` crée une instruction d'allocation en pile, qui sera mise à jour par la suite : en effet, au début de l'analyse d'une fonction MiniC lors de la passe 2, on ne sait pas encore de combien de temporaires on aura besoin en pile. Cette valeur sera connue à la fin de l'analyse de la fonction MiniC, et mise à jour à partir de la valeur passée en paramètre à la fonction `void create_stack_deallocation_inst(int32_t val)`, qui doit donc être appelée pour désallouer la place en pile nécessaire.

Il y a enfin les 3 fonctions suivantes :

- `void create_program()` : à appeler au début pour créer un programme
- `void dump_mips_program(char * filename)` : pour écrire le programme au format texte dans le fichier `filename`
- `void free_program()` : à appeler à la fin de la passe 2 pour libérer les structures allouées à la création

4 Implémentation du module de Contexte

Le module de contexte définit le type `context_t` qui fait l'association entre un nom d'identificateur et la définition de la variable correspondante.

Les fonctions de l'interface fournie sont les suivantes :

- `context_t create_context()` : alloue un objet de type `context_t` et le retourne
- `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`. Ce noeud doit être le `NODE_IDENT` associé à la déclaration de la variable. Si le nom `idf` est déjà présent, l'ajout échoue et la fonction retourne `false`. Sinon, la fonction retourne `true`.
- `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`.
- `void free_context(context_t context)` : libère la mémoire allouée pour `context`.

Remarque : Ce module n'est utilisé que par le module d'environnement. Cette interface n'est donc utile que dans le cas où vous implémentez votre propre module d'environnement mais pas votre module de contexte.

5 Implémentation du module d'Environnement

Le module d'environnement réalise la gestion de l'empilement et du dépilement des contextes, et permet d'associer un nom de variable à sa définition dans le contexte le plus proche (du bloc le plus interne vers le bloc le plus externe puis variable globale). Pour cela, le module chaine entre eux des contextes à l'aide de plusieurs variables statiques en interne, et en particulier une variable statique contenant l'environnement courant.

La difficulté de ce module est de gérer les offsets des différentes variables (globales, chaines de caractère, locales). En effet, le module gère un offset de contexte, réinitialisé au début de chaque fonction, qui est incrémenté à chaque déclaration de variable. À la fin de l'analyse d'une fonction, l'offset courant correspond à la place à allouer en pile pour les variables locales de la fonction. De plus, si l'environnement global est optionnel, les cas les chaines de caractère doivent dans tous les cas être placées en section `.data` après la déclaration de la dernière variable globale. Il faut donc pouvoir se souvenir de la valeur de l'offset courant à la fin de l'analyse des variables globales.

Dans l'absolu, il est possible d'utiliser le même emplacement en pile pour les variables locales de deux blocs consécutifs (non imbriqués). Néanmoins, il est conseillé d'utiliser des emplacements distincts pour toutes les variables locales de la fonction. C'est l'approche adoptée par l'implémentation fournie ; c'est aussi l'approche utilisée par gcc.

Pour des raisons de simplicité, on pourra considérer que toutes les variables se voient réservés 4 octets en mémoire quelque soit leur type et l'endroit de leur allocation (pile ou section `.data`). Les chaines de caractères doivent faire l'objet d'un traitement particulier.

L'interface fournie est décrite ci-après. Les fonctions `push_global_context()`, `push_context()`, `pop_context()`, `get_decl_node()`, `env_add_element()`, `reset_env_current_offset()`, `get_env_current_offset()` et `add_string()` sont à appeler lors de la passe 1, tandis que les fonctions `get_global_strings_number()` et `get_global_string()` sont à appeler lors de la passe 2. La fonction `free_global_strings()` est à appeler à la fin de la passe 2.

- `void push_global_context()` : est à appeler avant l'analyse de la déclaration des variables globales. Elle initialise un contexte pour les variables globales et en fait le contexte courant.
- `void push_context()` : est à appeler avant l'analyse de la déclaration des variables d'un bloc. Elle initialise un contexte pour les variables locales et en fait le contexte courant.
- `void pop_context()` : est à appeler à la fin de l'analyse d'un bloc déclarant des variables. Cette fonction dépile et libère le contexte courant.
- `int32_t env_add_element(char * ident, void * node)` : ajoute dans le contexte courant l'association entre le nom `ident` et le noeud `node`. Si la valeur retournée est positive ou nulle, il s'agit de l'offset de la variable dans l'environnement et l'offset courant du contexte courant est mis à jour ; si la valeur retournée est négative, cela signifie qu'une variable du même nom existe déjà dans le contexte courant.
- `void * get_decl_node(char * ident)` : retourne la définition de la variable `ident` rencontrée en premier dans l'empilement des contextes, en commençant par le contexte courant.
- `void reset_env_current_offset()` : réinitialise l'offset courant du contexte à 0 ; cette fonction doit être appelée au début de l'analyse d'une fonction dans la passe 1.
- `int32_t get_env_current_offset()` : retourne l'offset courant du contexte ; cette fonction est à appeler à la fin de l'analyse d'une fonction lors la passe 1 pour connaitre la place en pile qu'il est nécessaire d'allouer pour les variables locales de cette fonction. Cette valeur est à sauvegarder dans le champ `offset` du noeud de nature `NODE_FUNC` de la fonction analysée.
- `int32_t add_string(char * str)` : ajoute la déclaration en section `.data` d'une chaine de caractères littérale et retourne l'offset correspondant
- `int32_t get_global_strings_number()` : retourne le nombre de chaines de caractères littérales. Cette fonction devrait être utilisée pour la déclaration des chaines littérales en section `.data`.
- `char * get_global_string(int32_t index)` : retourne la chaine de caractères littérale

d'index `index`, qui doit être strictement inférieur à la valeur retournée par `get_global_strings_number()`. Cette fonction devrait être utilisée pour la déclaration des chaines littérales en section `.data`.

- `free_global_strings()` : libère la mémoire allouée pour les chaines littérales.

La valeur de retour des fonctions `env_add_element()` et `add_string()` devrait être stockée dans le champ `offset` des noeuds adéquats.

6 Implémentation de l'allocateur de registres

Le but de ce module est de fournir les numéros des registres pour les instructions du programme assembleur, et de gérer correctement le cas où il n'y a plus de registre disponible. Dans ce dernier cas, une expression dite *temporaire*, dans le sens où il ne s'agit pas d'une expression correspondant à la valeur d'une variable, doit être stockée en pile pour libérer un registre et restaurée plus tard.

Exemple

Par exemple, pour traduire en assembleur l'expression suivante :
`a = 1 + (2 + (3 + (4 + 5)))`
La sémantique de MiniC oblige l'évaluation dans l'ordre des expressions `1`, puis `2`, puis `3`, puis `4` et enfin `5`, mais aussi de respecter l'ordre des opérations spécifié par les parenthèses. L'arbre obtenu pour cette expression est représenté figure 1.

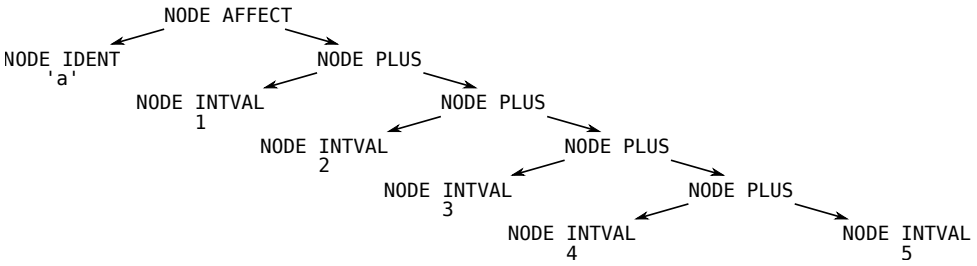


FIGURE 1 – Arbre de l'expression a = 1 + (2 + (3 + (4 + 5)))

Un code assembleur correct serait donc (en supposant que `a` se trouve à l'adresse `4(r29)`) :

```
addiu r8, r0, 1
addiu r9, r0, 2
addiu r10, r0, 3
addiu r11, r0, 4
addiu r12, r0, 5
addu r11, r11, r12
addu r10, r10, r11
addu r9, r9, r10
addu r8, r8, r9
sw r8, 4(r29)
```

Cette implémentation utilise 5 registres. Si on suppose que l'on ne dispose maintenant que de 4 registres pour implémenter cette expression, il faut utiliser la pile pour stocker des résultats intermédiaires du calcul. Un code assembleur est le suivant :

```
addiu r8, r0, 1
addiu r9, r0, 2
```

```
addiu r10, r0, 3
sw r10, 8(r29)
addiu r10, r0, 4
sw r10, 12(r29)
addiu r10, r0, 5
lw r11, 12(r29)
addu r10, r11, r10
lw r11, 8(r29)
addu r10, r11, r10
addu r9, r9, r10
addu r8, r8, r9
sw r8, 4(r29)
```

On observe ici que l'on a besoin de deux mots en pile pour stocker des valeurs temporaires. Ces deux mots en pile utilisée doivent être alloués au début de la fonction en même temps que la place pour les variables locales.

Remarque : Si une passe d'optimisation basée sur la propagation des constantes permettrait de résoudre ce cas précis (charger directement la valeur 15 dans un registre), le problème se pose dans tous les cas avec des variables et des expressions plus complètes, en particulier dans le cas des expressions qui ont des effets de bord. Par ailleurs, il ne vous est pas demandé de réaliser des passes d'optimisation.

Si maintenant on enlève les parenthèses de l'expression, l'arbre construit est illustré figure 2, et seuls deux registres suffisent :

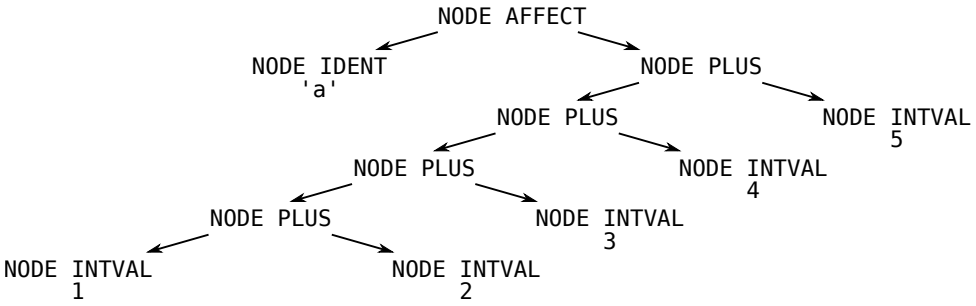


FIGURE 2 – Arbre de l'expression 1 + 2 + 3 + 4 + 5

```
addiu r8, r0, 1
addiu r9, r0, 2
addu r8, r8, r9
addiu r9, r0, 3
addu r8, r8, r9
addiu r9, r0, 4
addu r8, r8, r9
addiu r9, r0, 5
addu r8, r8, r9
sw r8, 4(r29)
```

Remarque : Dans le code précédent, il s'agit de l'implémentation la plus naïve et la plus automatique (celle qui vous est demandée). Il est bien sûr possible d'utiliser un seul registre et moins d'instructions en utilisant directement des instructions `addiu r8, r8, x` (pour `x` de 2 à 5).

Implémentation fournie

Toutes les fonctions de ce module sont à appeler au cours de la passe 2. Ainsi, au cours de cette passe, les instructions correspondant à la sauvegarde et à la restauration en pile des expressions temporaires doivent être générées, *via* les fonctions `push_temporary()` et `pop_temporary()`. De plus, la taille maximale allouée à un instant donné pour la sauvegarde des temporaires est utilisée à la fin de l'analyse d'une fonction pour connaitre la quantité de mémoire qu'il faut allouer en pile au début de la fonction.

- `bool reg_available()` : teste s'il reste un registre disponible pour stocker un résultat d'expression. Si la fonction retourne `false`, cela signifie qu'il faudra stocker un résultat intermédiaire en pile.
- `void push_temporary(int32_t reg)` : génère une instruction de sauvegarde du registre `reg` en pile (contenant une expression temporaire) et met à jour l'offset de sauvegarde des temporaires.
- `void pop_temporary(int32_t reg)` : génère une instruction de restauration du registre `reg` à partir de la pile, et met à jour l'offset de sauvegarde des temporaires.
- `int32_t get_current_reg()` : retourne le numéro du registre courant, c'est-à-dire du dernier registre alloué.
- `int32_t get_restore_reg()` : retourne le numéro du registre réservé pour la restauration des valeurs en pile. Ce numéro est dépendant du nombre de registres utilisables (option `-r`).
- `void allocate_reg()` : alloue un registre pour y stocker le résultat d'une expression ; il faut pour cela qu'il y ait au moins un registre disponible. L'effet de cette fonction sera visible lors du prochain appel à `get_current_reg()`, qui retournera un nouveau numéro.
- `void release_reg()` : libère le registre courant.
- `int32_t get_new_label()` : retourne un numéro unique de label
- `void set_temporary_start_offset(int32_t offset)` : définit l'offset de début pour les temporaires. Cet offset de début correspond à la place en pile réservée pour les variables locales. Cette fonction doit être appelée au début de l'analyse d'une fonction, pour que les offsets générés dans les instruction de sauvegarde et restauration des temporaires soient corrects.
- `void reset_temporary_max_offset()` : réinitialise l'offset maximum pour les temporaires. Cette fonction doit être appelée au début de l'analyse d'une fonction.
- `int32_t get_temporary_max_offset()` : retourne l'offset maximum atteint pour les temporaires lors de l'analyse de la fonction courante. Cette fonction doit être appelée à la fin de l'analyse d'une fonction, pour calculer la place requise en pile par cette fonction : il faut pour cela ajouter à cette valeur la place occupée par les variables locales.
- `int32_t get_temporary_curr_offset()` : retourne l'offset courant pour les temporaires. Cette fonction n'est utile qu'à des fins de débug.

Lien entre variables locales et temporaires

Les variables locales, comme les expressions temporaires, utilisent toutes les deux la pile. Ce sont néanmoins deux choses différentes : les variables locales sont rencontrées au cours de la passe 1, et à la fin de celle-ci, on est donc en mesure de dire le nombre de variables locales que comporte une fonction. Comme les offsets des variables locales retournés par l'environnement servent directement à adresser la pile depuis son sommet (si l'appel `env_add_element(...)` retourne 4 pour l'ajout d'une variable locale, alors il faudra accéder à la variable par `4($29)`), ces offsets ne peuvent pas être modifiés. Pour cette raison et pour ne pas introduire davantage de complexité, les temporaires sont alloués en pile "sous" les variables locales, contrairement à ce qui est normalement fait. À la fin de la passe 2, quand on connait le nombre de mots à allouer en pile pour les temporaires, on ajoute cette valeur à la place à allouer en pile pour les variables locales pour obtenir la place totale en pile à allouer pour la fonction.

Ceci est illustré sur l'exemple suivant, qui reprend l'expression de l'exemple précédent, en

supposant que l'on n'ait que 4 registres disponibles :

```
1       void main() {
2           int a = 0;
3           {
4               int b = 1;
5               a = 1 + (2 + (3 + (4 + 5)));
6           }
7       }
```

- Lors de la passe 1, on analyse les déclarations de `a` et `b`, et on les ajoute dans l'environnement courant avec la fonction `env_add_element()`. Ces variables obtiennent respectivement les offsets 0 et 4, valeurs qui sont écrites dans le champ `offset` des noeuds de nature `NODE_IDENT` correspondant à la déclaration de ces variables.
- À la fin de l'analyse de la fonction `main`, toujours dans la passe 1, le champ `offset` du noeud de nature `NODE_FUNC` est mis à jour avec la valeur renvoyée par la fonction `get_env_current_offset()`, ici 8 car ces 2 variables occupent 8 octets en pile.
- Au début de la passe 2, il faut informer le module de gestion des registres que les temporaires doivent commencer à l'offset 8 avec la fonction `set_temporary_start_offset()`. Cela est nécessaire car la passe 2 va générer des instructions `sw` et `lw` pour les temporaires, sans écraser les variables locales `a` et `b`.
- L'appel à la fonction `create_stack_allocation_inst()` au début du main crée une instruction `addiu $29, $29, 0` dont l'immédiat sera modifié plus tard.
- Lors de la passe 2, lors de l'analyse de l'expression `1 + (2 + (3 + (4 + 5)))`, 2 mots en pile pour stocker des expressions temporaires sont requis. Lors du stockage du 2e temporaire, l'offset courant des temporaires atteint la valeur 8. On ne sait néanmoins toujours pas la valeur qu'il faudra allouer en pile car une expression qui suit pourrait avoir besoin de plus de temporaires.
- À la fin de l'analyse du `main` dans la passe 2, on connait enfin la place à allouer en pile : il s'agit de la somme entre le champ `offset` du noeud `NODE_FUNC` mis à jour à la fin de la passe 1 et contenant la valeur 8 (pour les variables locales), et la valeur 8 renvoyée par la fonction `get_temporary_max_offset()`. Cette valeur est passée à la fonction `create_stack_deallocation_inst()` qui va créer l'instruction `addiu` pour désallouer cette place en pile, et mettre à jour le champ immédiat de l'instruction `addiu` utilisée pour l'allocation.

La pile peut être représentée comme sur la figure 3.

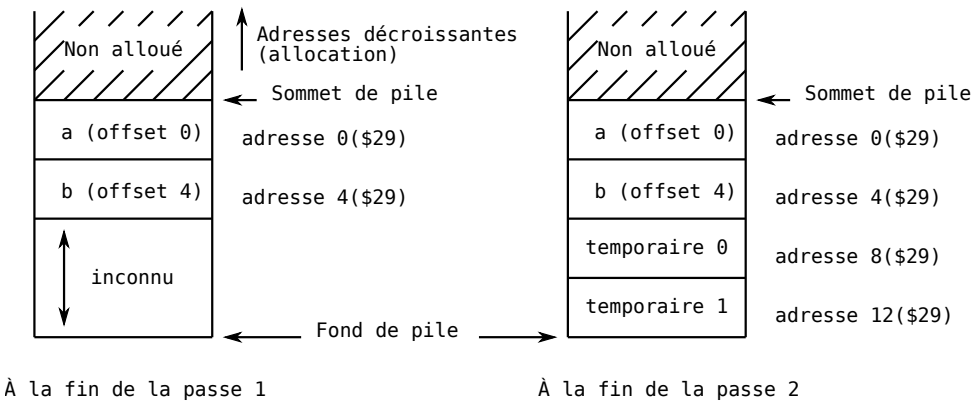


FIGURE 3 – Vue de la pile à la fin des passes 1 et 2

7 Vérification de l'arbre du programme

La fonction `check_program_tree(node_t n)` fournie dans la librairie prend en paramètre le noeud racine de l'arbre d'un programme et vérifie que l'arbre est conforme à la grammaire d'arbre. Si c'est le cas, cette fonction n'affiche rien et retourne `true`. Si une erreur est rencontrée, la fonction affiche un message décrivant le problème et retourne `false` (remarque : il est possible que cette fonction ne soit pas exempte de bug, notamment pour les arbres incorrects).

8 Affichage de l'arbre du programme

La fonction `dump_tree` qui vous est fournie dans le fichier `common.c` permet de générer un graphe de l'arbre du programme au format dot. Ce graphe peut être visualisé à l'aide de l'outil `xdot` ou `graphviz`. On peut par exemple faire appel à cette fonction à la fin de la construction de l'arbre pour vérifier que l'arbre construit respecte bien la grammaire d'arbre, ou à la fin de la passe 1 pour vérifier que les décorations ajoutées lors de la passe 1 sont correctes.

Sur les distributions récentes de linux, installer cet outil se fait de la manière suivante :

```
sudo apt install python3-pip # pour installer pip3, le gestionnaire de packages
                             # de python3
pip3 install xdot            # pour installer le module xdot de python3
sudo apt install xdot        # pour installer l'utilitaire xdot
xdot apres_syntaxe.dot &     # Pour visualiser un arbre
```

Attention : l'ordre affiché entre les différents fils d'un noeud ne correspond pas forcément à l'ordre des fils dans le tableau `opr`.

9 Allocations et désallocations mémoire

Dans ce projet, un certain nombre d'allocations mémoire sont à réaliser. Dans un esprit de programmation durable, il est attendu (et il sera vérifié) que votre programme ne comporte pas de fuite mémoire. Vous pouvez bien sûr utiliser `valgrind` pour traquer de telles fuites, et il est fortement recommandé de le faire (`valgrind` est également utile pour le débug, pour voir par exemple les accès aux variables non initialisées). Pour ne pas avoir de fuites mémoire avec l'utilisation de `lex` et `yacc`, il faut appeler la fonction `yylex_destroy()` à la fin de votre `main()`, et compiler le fichier produit par `yacc` avec l'option `-DYY_NO_LEAKS`. Une exception à ceci est lorsque le programme d'entrée comporte une erreur détectée dans la passe 1. Dans ce cas, on ne cherchera pas à désallouer toutes les structures.

10 Code de référence, simulateur et fichiers fournis

Les différentes ressources numériques se trouvent dans une archive `projet_compilation_src.tar` dont l'emplacement vous sera communiqué ultérieurement. Elle contient notamment les fichiers source fournis, ainsi que les implémentations des différentes fonctions fournies dans la librairie `libminiccutils.a`. L'arborescence de l'archive est la suivante :

```
arch.c
arch.h
common.c
common.h
defs.h
grammar.y
lexico.l
```

```
Makefile
minicc_ref
passe_1.c
passe_1.h
passe_2.c
passe_2.h
Tests/
  Syntaxe/
    KO/
    OK/
  Verif/
    KO/
    OK/
  Gencode/
    KO/
    OK/
utils/
   libminiccutils.a
   miniccutils.h
```

Remarques

- `minicc_ref` est un binaire du compilateur de référence
- Le projet est à faire sous linux, car les binaires ne seront pas portés sur windows.
- Il est conseillé d'utiliser le simulateur `mars` pour simuler le code assembleur produit. L'archive java de ce simulateur, `Mars_4_2.jar` est disponible sur mooodle. Elle est utilisable en ligne de commande ou avec une interface graphique (utile pour débugger les codes assembleur générés). Cette archive est exécutable et se lance de la manière suivante : `java -jar Mars_4_2.jar` (il est conseillé de créer un alias).
- Vos noms de fichiers et répertoires, si vous en créez, ne doivent pas comporter d'espace.

III Organisation du travail

1 Gestion du projet

Le travail est à réaliser par binôme, et c'est à vous de vous répartir le travail au sein du binôme. Néanmoins, à l'issue du projet, les deux membres du binôme devraient avoir une connaissance précise du projet, y compris sur le code qu'ils n'ont pas écrit.

La partie concernant `lex` et `yacc` est à réaliser en priorité, puisqu'elle conditionne tout le reste du projet. Ensuite, il vous est conseillé de faire marcher au plus vite l'affichage des chaines de caractère, afin de pouvoir tester vos programmes. À ce sujet, il est généralement observé que les programmes sont largement sous-testés. L'écriture des tests devrait être faite en parallèle, sinon avant, l'écriture du programme à tester. Il vous est par ailleurs fortement recommandé d'écrire des scripts de test qui permettent de lancer tous vos tests en d'en vérifier le résultat (à titre personnel, je recommande python pour cela, mais d'autres langages sont possibles). Cela permet d'avoir des tests dits de "non régression", et de s'assurer ainsi qu'un ajout ou une modification dans une passe ne “casse” pas une fonctionnalité qui marchait.

Concernant les optimisations (par exemple : propagation des constantes, mises de certaines variables locales en registres, etc.), cet aspect ne sera pas pris en compte pour la notation, aussi il vous est déconseillé d'essayer d'optimiser le code assembleur produit. Seule la fonctionnalité et le respect de la spécification seront évalués. Bien sûr, il n'y aura pas de pénalité pour la génération d'un code optimisé mais le temps devrait plutôt être passé sur les tests.

Enfin, les types fournis, tels que le type `node_t` ne doivent pas être modifiés, car la librairie `libminiccutils.a` ne serait pllus compatible.

2 Livrables

Il vous est demandé de fournir à la fin du projet votre code, vos tests, vos scripts de tests et un rapport d'une dizaine de pages au format **pdf** décrivant tous les éléments qui vous semblent pertinents, comme par exemple :

- Les choix de conception réalisés
- Les modules et fonctionnalités implémentés et non implémentés, les fonctionnalités qui ne marchent pas, les bugs connus
- Une description concernant l'utilisation de vos scripts de test
- L'architecture logicielle de votre compilateur ou de vos scripts

L'archive contenant votre code et le rapport devra être de type `.tar.gz` et ne devra contenir aucun fichier binaire (autre que le rapport).

Si cela est possible, une soutenance sera organisée à la fin du projet, au cours de laquelle des questions vous seront posées sur des aspects d'implémentation aussi bien que sur des aspects plus généraux.

Concernant les tests, ceux-ci comptent pour une part conséquente de la note (20%) et seront évalués de manière automatique à la fin du projet. Ils peuvent être écrits en parallèle ou même avant le compilateur, et il est conseillé de les démarrer au plus tôt. La structure donnée pour l'arborescence des tests devrait être respectée, à savoir :

- Les tests présents dans le dossier `Tests/Syntaxe/OK` ne doivent pas provoquer d'erreur – et donc ne rien afficher – quand ils sont compilés avec l'option `-s`.
- Les tests présents dans le dossier `Tests/Syntaxe/KO` doivent provoquer une erreur – et donc afficher un message avec le numéro de ligne correct – quand ils sont compilés avec l'option `-s`.
- Les tests présents dans le dossier `Tests/Verif/OK` ne doivent pas provoquer d'erreur – et donc ne rien afficher – quand ils sont compilés avec l'option `-v`.
- Les tests présents dans le dossier `Tests/Verif/KO` doivent provoquer une erreur – et donc afficher un message avec le numéro de ligne correct – quand ils sont compilés avec l'option `-v`.
- Les tests présents dans le dossier `Tests/Gencode/OK` ne doivent pas provoquer d'erreur – et donc ne rien afficher et produire un fichier assembleur – quand ils sont compilés.
- Les tests présents dans le dossier `Tests/Gencode/KO` ne doivent pas provoquer d'erreur – et donc ne rien afficher et produire un fichier assembleur – quand ils sont compilés, mais provoquer une erreur à l'exécution.

Remarques :

- Les tests présents dans les deux sous-dossiers de `Syntaxe` ne seront appelés qu'avec l'option `-s`, et ceux dans les deux sous-dossiers de `Verif` ne seront appelés qu'avec l'option `-v`
- Les tests de `Gencode` devraient effectuer des affichages avec `print` ; en effet, le code assembleur ne peut pas être testé autrement que par le résultat de son exécution, donc les résultats des conditions et calculs faits dans le programme de test devraient être affichés pour permettre de discriminer entre un compilateur buggé et un compilateur sain.
- Certains tests peuvent être réutilisés entre deux parties. Par exemple, tous les tests dans `Verif/OK` produisent un fichier assembleur s'ils sont compilés sans `-v`, et peuvent donc être copiés dans `Gencode`, modulo l'ajout d'une trace pertinente (cf. point du dessus).
- Concernant les scripts de tests, ceux-ci ne doivent pas reposer sur l'utilisation de `minicc_ref` : par exemple, il ne faut pas faire un `diff` entre la sortie de votre compilateur et celle de `minicc_ref`. En effet, dans un vrai projet, vous n'aurez pas de programme de référence.

Vous pouvez bien sûr vous servir de `minicc_ref` pour savoir le résultat attendu de la compilation d'un fichier (par exemple, voir qu'il y a une erreur ligne 24), mais pas de manière automatique dans un script (c'est à vous de stocker quelque part l'information que l'erreur doit se produire ligne 24).

3 Évaluation

Votre projet sera évalué sur les aspects suivants :

- 40% : Le passage de votre compilateur sur un ensemble de tests de manière automatique. Le score obtenu constituera la note.
- 20% : Le passage de l'ensemble de vos tests sur des compilateurs buggés ("mutants") de manière automatique. Le score obtenu constituera la note.
- 10% : L'automatisation de vos tests.
- 10% : La qualité d'écriture de votre code (indentation, respect d'un style, découpage en fonctions pertinent, etc.). Cette note sera aussi abaissée si la quantité de code écrite est faible (par exemple, rendre 50 lignes parfaitement indentées ne permet pas d'avoir 20 à ce critère).
- 10% : Le rapport décrivant l'architecture logicielle.
- 5% : Fuites mémoire, si nombre d'allocation suffisamment conséquent.
- 5% : Erreurs visibles ou non à l'exécution (typiquement, erreurs détectées par valgrind), si le code écrit est suffisamment conséquent.

Les coefficients données sont indicatifs et sont susceptibles d'être modifiés. En cas de soutenance, ces coefficients seront réajustés. Les notes seront à priori les mêmes pour les deux membres d'un binôme. Néanmoins, en cas de déséquilibre significatif perçu entre les membres, les notes seront dissociées.

Remarques :

- Le non respect des consignes (telles que la gestion des arguments de la ligne de commande, la modification de l'arborescence des tests, ou l'affichage de messages ou traces pour un test correct) entrainera un malus sur la note. De même, un code qui ne compile pas sera sanctionné.
- Une grande partie des tests qui seront utilisés pour évaluer votre compilateur comportent l'affichage de chaines de caractères pour déterminer si le résultat est correct. Il est donc indispensable que votre compilateur gère correctement l'affichage d'une chaine de caractères littérale simple.
- Les pseudos-instructions (macros) mips ne sont pas autorisées ; l'évaluation avec Mars utilisera l'option `np` qui les désactive, et vous êtes encouragés à faire de même.

4 Fraude

Tous les projets seront analysés de manière automatique pour y détecter les cas de fraude. Voici ci-après un extrait du règlement de l'école au regard de la fraude et des sanctions associées.

> ***7.3 Infraction, plagiat, fraude***
>
> *Toute infraction aux instructions énoncées au §7.2 ou tentative de fraude dûment constatée entraîne l'application des articles R.712-9 à R 712-46 et R811-10 et R 811-11 du code de l'éducation relatifs à la procédure disciplinaire dans les établissements publics d'enseignement supérieur.*
>
> *Le plagiat consiste à présenter comme sien ce qui a été produit par un autre, quelle qu'en soit la source (ouvrage, documents sur internet, travail d'un autre élève...). Le plagiat est une fraude.*

En cas de fraude, l'élève est susceptible d'être déféré en section disciplinaire de l'établissement et s'expose aux sanctions suivantes :

- *l'avertissement ;*
- *le blâme ;*
- *l'exclusion de l'établissement pour une durée maximum de 5 ans - cette sanction peut être prononcée avec sursis si l'exclusion n'excède pas 2 ans ;*
- *l'exclusion définitive de l'établissement ;*
- *l'exclusion de tout établissement public d'enseignement supérieur pour une durée maximum de 5 ans ;*
- *l'exclusion définitive de tout établissement public d'enseignement supérieur.*

Toute sanction prévue ci-dessus et prononcée dans le cas d'une fraude ou d'une tentative de fraude commise à l'occasion d'une épreuve de contrôle continu, d'un examen ou d'un concours entraîne, pour l'intéressé, la nullité de l'épreuve correspondante ou du groupe d'épreuves ou de la session d'examen ou du concours.

En particulier, tout échange de code, y compris de tests ou de scripts, entre deux binômes différents constitue une fraude et entrainera la note de 0 pour les deux membres des deux binômes et/ou une procédure disciplinaire à l'encontre des personnes concernées.

Pour protéger vos données de toute tentative de copie de la part d'autres étudiants, vous devrez exécuter la commande suivante :

```
chmod -R go-rwx compilation/
```

sur le dossier contenant tous vos fichiers de projet (`compilation/` dans cet exemple). Enfin, si vous utilisez un dépôt git sur internet, pensez à empêcher les accès extérieurs.

IV Crédits

La présentation de ce projet s'inspire, en version réduite, du projet de génie logiciel de Grenoble INP - Ensimag. Avec l'aimable autorisation de Roland Groz et Catherine Oriat (Prenom.Nom@imag.fr)

III. Annexes

Instructions MIPS

	Assembleur		Opération		Effet	Format
Arithmétiques/logiques	Add	Rd, Rs, Rt	Add	*Overflow detection*	Rd<-Rs+Rt	R
	Sub	Rd, Rs, Rt	Substract	*Overflow detection*	Rd<-Rs-Rt	R
	Addu	Rd, Rs, Rt	Add	*No Overflow*	Rd<-Rs+Rt	R
	Subu	Rd, Rs, Rt	Substract	*No Overflow*	Rd<-Rs-Rt	R
	Addi	Rt, Rs, I	Add Immediate	*Overflow detection*	Rt<-Rs+I	I
	Addiu	Rt, Rs, I	Add Immediate	*No Overflow*	Rt<-Rs+I	I
	Or	Rd, Rs, Rt	Logical Or		Rd<-Rs or Rt	R
	And	Rd, Rs, Rt	Logical And		Rd<-Rs and Rt	R
	Xor	Rd, Rs, Rt	Logical Exclusive-Or		Rd<-Rs xor Rt	R
	Nor	Rd, Rs, Rt	Logical Not Or		Rd<-Rs nor Rt	R
	Ori	Rt, Rs, I	Or Immediate	*Unsigned immediate*	Rt<-Rs or I	I
	Andi	Rt, Rs, I	And Immediate	*Unsigned immediate*	Rt<-Rs and I	I
	Xori	Rt, Rs, I	Exclusive-Or Immediate	*Unsigned immediate*	Rt<-Rs xor I	I
	Sllv	Rd, Rt, Rs	Shitf Left Logical Variable	*5 lsb of Rs is significant*	Rd<-Rt<<Rs	R
	Srlv	Rd, Rt, Rs	Shitf Right Logical Variable	*5 lsb of Rs is significant*	Rd<-Rt>>Rs	R
	Srav	Rd, Rt, Rs	Shitf Right Arithmetical Variable	*5 lsb of Rs is significant* **with sign extension*	Rd<-Rt>>*Rs	R
	Sll	Rd, Rt, sh	Shitf Left Logical		Rd<-Rt<<sh	R
	Srl	Rd, Rt, sh	Shitf Right Logical		Rd<-Rt>>sh	R
	Sra	Rd, Rt, sh	Shitf Right Arithmetical	**with sign extension*	Rd<-Rt>>*sh	R
	Lui	Rt, I	Load Upper Immediate	*16 lowers bits of Rt are set to zero*	Rt<-I | | "0000"	I
	Slt	Rd, Rs, Rt	Set if Less Than		Rd<-1 if Rs<Rt else 0	R
	Sltu	Rd, Rs, Rt	Set if Less Than Unsigned		Rd<-1 if Rs<Rt else 0	R
	Slti	Rt, Rs, I	Set if Less Than Immediate	*Sign extended Immediate*	Rt<-1 if Rs<I else 0	I
	Sltiu	Rt, Rs, I	Set if Less Than Immediate	*Unsigned Immediate*	Rt<-1 if Rs<I else 0	I
	Mult	Rs, Rt	Multiply	*LO<-32 low significant bits* *HI<-32 high significant bits*	Rs*Rt	R
	Multu	Rs, Rt	Multiply Unsigned	*LO<-32 low significant bits* *HI<-32 high significant bits*	Rs*Rt	R
	Div	Rs, Rt	Divide	*LO<-Quotient* *HI<-Remainder*	Rs/Rt	R
	Divu	Rs, Rt	Divide Unsigned	*LO<-Quotient* *HI<-Remainder*	Rs/Rt	R
HI, LO	Mfhi	Rd	Move From HI		Rd<-HI	R
	Mflo	Rd	Move From LO		Rd<-LO	R
	Mthi	Rs	Move To HI		HI<-Rs	R
	Mtlo	Rs	Move To LO		LO<-Rs	R

Lecture/écriture mémoire	Lw	Rt, I(Rs)	Load Word	*Sign extended immediate*	Rt<-M(Rs+I)	I
	Sw	Rt, I(Rs)	Store Word	*Sign extended immediate*	M(Rs+I)<-Rt	I
	Lh	Rt, I(Rs)	Load Half Word	*Sign extended immediate. Two bytes from storage are located into the 2 less significant bytes of Rt. The sign of these 2 bytes is extended on the 2 most significant bytes.*	Rt<-M(Rs+I)	I
	Lhu	Rt, I(Rs)	Load Half Word Unsigned	*Sign extended immediate. Two bytes from storage are located into the 2 less significant bytes of Rt, others bytes are set to zero.*	Rt<-M(Rs+I)	I
	Sh	Rt, I(Rs)	Store Half Word	*Sign extended immediate/. The two less significant bytes of Rt are stored into the storage.*	M(Rs+I)<-Rt	I
	Lb	Rt, I(Rs)	Load Byte	*Sign extended immediate. One byte from storage is located into the less significant bytes of Rt. The sign of this byte is extended on the 3 most significant bytes.*	Rt<-M(Rs+I)	I
	Lbu	Rt, I(Rs)	Load Byte Unsigned	*Sign extended immediate. One byte from storage is located into the less significant bytes of Rt, others bytes are set to zero.*	Rt<-M(Rs+I)	I
	Sb	Rt, I(Rs)	Store Byte	*Sign extended immediate. The less significant byte of Rt is stored into the storage.*	M(Rs+I)<-Rt	I
Branchements	Beq	Rs, Rt, label	Branch if EQual		PC<-PC+4+(I*4) if Rs=Rt PC<-PC+4 if Rs!=Rt	I
	Bne	Rs, Rt, label	Branch if Not Equal		PC<-PC+4+(I*4) if Rs!=Rt PC<-PC+4 if Rs=Rt	I
	Bgez	Rs, label	Branch if Greater or Equal Zero		PC<-PC+4+(I*4) if Rs>=0 PC<-PC+4 if Rs<0	I
	Bgtz	Rs, label	Branch if Greater Than Zero		PC<-PC+4+(I*4) if Rs>0 PC<-PC+4 if Rs<=0	I
	Blez	Rs, label	Branch if Less or Equal Zero		PC<-PC+4+(I*4) if Rs<=0 PC<-PC+4 if Rs>0	I
	Bltz	Rs, label	Branch if Less Than Zero		PC<-PC+4+(I*4) if Rs<0 PC<-PC+4 if Rs>=0	I
	Bgezal	Rs, label	Branch if Greater or Equal Zero And Link		PC<-PC+4+(I*4) if Rs>=0 PC<-PC+4 if Rs<0 R31<-PC+4 in both cases	I
	Bltzal	Rs, label	Branch if Greater Than Zero And Link		PC<-PC+4+(I*4) if Rs<0 PC<-PC+4 if Rs>=0 R31<-PC+4 in both cases	I
	J	Label	Jump		PC<-PC 31:28 | | I*4	J
	Jal	Label	Jump and Link		R31<-PC+4 PC<-PC 31:28 | | I*4	J
	Jr	Rs	Jump Register		PC<-Rs	R
	Jalr	Rs	Jump and Link Register		R31<-PC+4 PC<-Rs	R
	Jalr	Rd, Rs	Jump and Link Register		Rd<-PC+4 PC<-Rs	R

Appels système

Avant de réaliser un appel système (avec syscall), il faut placer dans le registre $2 le numéro de l'appel système demandé. Il faut aussi donner les paramètres de l'appel quand il y en a. Le passage se fait par les registres, les registres $4 et/ou $5 sont utilisés. La valeur de retour (s'il y en a une) se trouve après l'appel dans le registre $2.

Écrire un entier en décimal sur la console :

- Appel système numéro 1.
- Un paramètre : l'entier à écrire sur la console qui doit être placé dans le registre $4.

Lire un entier sur la console :

- Appel système numéro 5.
- Valeur de retour (dans $2 après l'appel) : l'entier lu.

Écrire une chaîne de caractères sur la console :

- Appel système numéro 4.
- Un paramètre : l'adresse de la chaîne de caractères à écrire doit être placée dans le registre $4.

Lire une chaîne de caractères sur la console :

- Appel système numéro 8.
- 2 paramètres :
    1. l'adresse mémoire à partir de laquelle la chaîne de caractères lue sera sauvegardée doit être placée dans le registre $4.
    2. la taille maximale de la chaîne de caractères attendue doit être placée dans le registre $5 (en octet).

Attention, avec le simulateur MARS, la chaîne de caractères lue se termine par '\n' puis par '\0'.

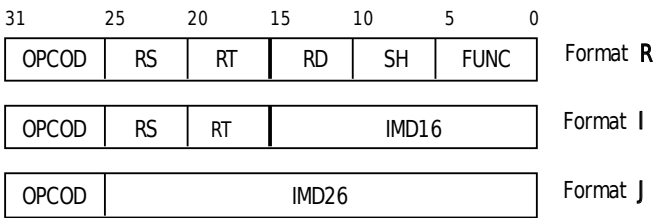
Terminer un programme :

- Appel système numéro 10.

Table des codes ASCII

Hex	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	NUL	SOH	STX	ETX	EOT	ENQ	ACK	BEL	BS	HT	LF	VT	FF	CR	SO	SI
1	DLE	DC1	DC2	DC3	DC4	NAK	SYN	ETB	CAN	EM	SUB	ESC	FS	GS	RS	US
2	SP	!	"	#	$	%	&	'	(	)	*	+	,	-	.	/
3	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
4	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
5	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
6	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
7	p	q	r	s	t	u	v	w	x	y	z	{	|	}	~	DEL

Format de codage des instructions



Directives assembleur

.align n : aligne le compteur d'adresse de la section concernée sur une adresse telle que les n bits de poids faible soient à zéro (c'est-à-dire une adresse multiple de 2^n).

.ascii chaîne [, autrechaîne, ...] : place à partir de l'adresse du compteur d'adresse de la section concernée la suite de caractères entre guillemets. S'il y a plusieurs chaînes, elles sont placées à la suite. Cette chaîne peut contenir des séquences d'échappement du langage.

.asciiz chaîne [, autrechaîne, ...] : identique à la précédente, la seule différence étant qu'elle ajoute un zéro binaire à la fin de chaque chaîne.

.byte n [, m, ...] : les valeurs de n [et m,...] représentées sur 1octet (tronquées sur 8 bits) sont placées à des adresses successives de la section, à partir de l'adresse du compteur d'adresse de cette section.

.half n [, m, ...] : les valeurs de n [et m,...] représentées sur 2 octets (tronquées sur 16 bits) sont placées à des adresses successives de la section, à partir de l'adresse du compteur d'adresse de la section.

.word n [, m, ...] : les valeurs de n [et m, ...] représentées sur 4 octets sont placées dans des adresses successives de la section, à partir de l'adresse du compteur d'adresse de la section.

.space n : un espace de n octets est réservé à partir du compteur d'adresse de la section concernée.

Codage des codes opération des instructions

DECODAGE OPCOD

INS 31:29 \ INS 28:26	000	001	010	011	100	101	110	111
000	SPECIAL	BCOND	J	JAL	BEQ	BNE	BLEZ	BGTZ
001	ADDI	ADDIU	SLTI	SLTIU	ANDI	ORI	XORI	LUI
010	COPRO							
011								
100	LB	LH		LW	LBU	LHU		
101	SB	SH		SW				
110								
111								

OPCOD = SPECIAL

INS 5:3 \ INS 2:0	000	001	010	011	100	101	110	111
000	SLL		SRL	SRA	SLLV		SRLV	SRAV
001	JR	JALR			SYSCALL	BREAK		
010	MFHI	MTHI	MFLO	MTLO				
011	MULT	MULTU	DIV	DIVU				
100	ADD	ADDU	SUB	SUBU	AND	OR	XOR	NOR
101			SLT	SLTU				
110								
111								