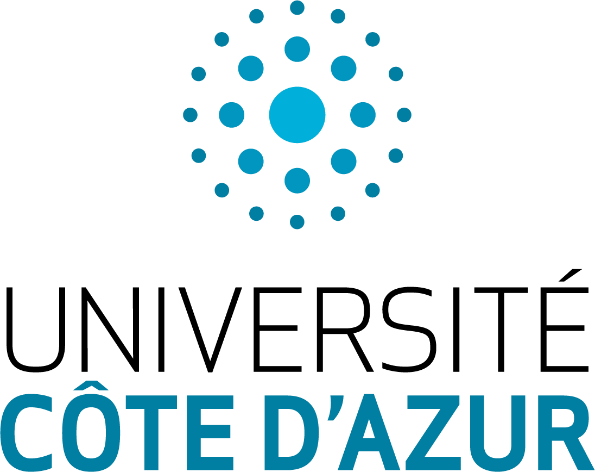
**Rapport de Projet de Compilation**

Compilateur d’un langage miniC vers DOT



**Licence 3 Informatique 2025**

**Université Côte d’Azur**

**Réalisé par :**

**Daniel CARRIBA NOSRATI et Bence DI PLACIDO**

**SOMMAIRE**

**Table des matières**

[**1. INTRODUCTION 3**](#_u5vanh2fbmks)

[**2. MÉTHODE DE TRAVAIL 4**](#_qw2lqnakprcr)

[**3. ARBORESCENCE DU PROJET 5**](#_n449dcrv6cu9)

[**4. DÉVELOPPEMENT 7**](#_8gk2z5v2o0we)

[4.1. ANALYSE LEXICALE 7](#_lfov27kl7u7h)

[4.2. ANALYSE SYNTAXIQUE 7](#_au1sy8fevsjt)

[4.2.1 Problèmes rencontrés lors de l’analyse syntaxique 9](#_2gpeb41jb6jb)

[4.3. ANALYSE SÉMANTIQUE 9](#_7nj3kl7amrzc)

[4.3.1 Problèmes rencontrés lors de l’analyse sémantique 10](#_wntr739jphm)

[4.4. GÉNÉRATION DE CODE 10](#_7v9a16jc7zn6)

[**5. ERREURS DÉTECTÉS PAR L’ANALYSE SÉMANTIQUE 12**](#_3mpypztvb4dh)

[5.1. ERREURS 12](#_e26ocsc4sc8)

[5.2. WARNINGS 12](#_q2fa1hcz682)

[**6. CONCLUSION 13**](#_7a6w4szhnvk)

[**7. ANNEXES 14**](#_866lpfxlsyhw)

[7.1 EXEMPLES DE SCHÉMAS DE LA GÉNÉRATION DE L’ARBRE ABSTRAIT 14](#_dxybixuz21b1)

[7.1.1 Génération de l’arbre abstrait pour la règle expression 14](#_86ln2io7u8qo)

[7.1.2 Génération de l’arbre abstrait pour la règle selection 15](#_9x20xlgwds18)

[7.1.3 Génération de l’arbre abstrait pour la règle condition 17](#_fh3ernw89uua)

[7.1.4 Génération de l’arbre abstrait pour la règle affectation 18](#_drt9wzm7l1xp)

[7.1.5 Génération de l’arbre abstrait pour la règle variable 19](#_csx421w2oh9p)

# 

# 1. INTRODUCTION

Dans le cadre de l’UE Compilation, nous avons réalisé un compilateur de langage miniC, un sous-langage du langage C, capable de traduire ce dernier en langage intermédiaire DOT, un langage intermédiaire structuré.

En réalisant ce projet, nous avons pu mettre en pratique les connaissances acquises durant le cours et les TD. Nous avons essayé d’avancer dans le rythme du cours afin de pouvoir mettre en œuvre les connaissances acquises.

Dans un premier temps nous avons complété l’analyse lexicale (ANSI-C.l) qui génère les tokens. Par la suite, nous avons réalisé l’analyse syntaxique à l'aide des définitions données dans le fichier “miniC.y”. Durant l’évolution du projet, nous avons modifié les règles de la grammaire (voir [4.2 ANALYSE SYNTAXIQUE](#_au1sy8fevsjt)). De plus, nous avons implémenter l’analyse sémantique pour respecter au maximum les attentes du langage miniC et générer le code DOT conformément aux consignes.

Pour les différentes parties du projet, nous avons utilisé les technologies respectives :

* Lex pour l’analyse lexicale
* Yacc pour l’analyse syntaxique
* C pour la sémantique et la génération de code

Pour ce projet, nous avons réalisé un compilateur à deux passes. Lors de la première passe le compilateur détecte les potentielles erreurs lexicales, syntaxiques et sémantiques et génère une structure de données (arbre abstrait) qui, si aucune erreur n’a été détectée, va être utilisée lors de la seconde passe pour faire la génération de code DOT.

Vous trouverez des exemples de schémas pour la génération de l’arbre abstrait dans la partie [7. ANNEXES](#_866lpfxlsyhw).

# 

# 2. MÉTHODE DE TRAVAIL

Afin d’avancer à un bon rythme et de ne pas nous égarer dans le projet, nous avons établi différents objectifs à réaliser chaque semaine. Nous nous sommes régulièrement consultés, sur place à Valrose ou à distance via Discord, afin de discuter de nos avancées ou des problèmes rencontrés.

Nous avons utilisé Git et la plateforme GitHub pour la gestion de versions, afin de mener à bien le projet. Nous avons travaillé ensemble sur toutes les parties du projet.

# 

# 3. ARBORESCENCE DU PROJET

├── ANSI-C.l // fichier contenant l’analyse lexicale

├── exempleminiC.c

├── Grammaire-miniC.pdf

├── Grille-Evaluation-Projet-Compilation.pdf

├── Makefile // fichier Makefile pour compiler le code, exécuter le code ou encore supprimer l’exécutable et les fichiers dot générés

├── miniC.y // fichier contenant l’analyse syntaxique et sémantique et la génération de code

├── Projet-Compilation-2025.pdf

├── Rapport

│ └── Rapport-CARRIBA\_NOSRATI\_Daniel-DI\_PLACIDO\_Bence.docx // rapport du projet au format word

├── README.md // readme au format md pour github

├── README.txt // readme au format txt

├── Tests // fichiers tests fournis par le professeur

│ ├── add.c

│ [...]

│ └── variables.c

├── TestsError // fichiers tests supplémentaires produisants différentes erreurs

│ ├── division\_par\_zero.c

│ ├── fonction\_deja\_declaree.c

│ ├── lexical\_error.c

│ ├── switch\_plusieurs\_case\_identiques.c

│ ├── switch\_plusieurs\_default.c

│ ├── syntax\_error.c

│ ├── utilisation\_variable\_dimension\_superieur.c

│ ├── variable\_deja\_declaree\_bloc.c

│ └── variable\_void.c

├── TestsWarning // fichiers tests supplémentaires produisants différents warnings

│ ├── absence\_return\_fonction\_int.c

│ ├── acces\_mauvais\_indice\_tableau.c

│ ├── appel\_fonction\_avec\_mauvais\_nb\_params.c

│ ├── presence\_return\_fonction\_void.c

│ ├── utilisation\_fonction\_non\_declaree.c

│ ├── utilisation\_fonction\_void\_dans\_expr.c

│ └── utilisation\_variable\_non\_declaree.c

└── utils // fichiers c et h utilisés par le compilateur

├── arbres.c

├── arbres.h // structures et fonctions pour les arbres

├── couleurs\_terminal.h

├── dot.c

├── dot.h // fonctions pour générer du code au format dot

├── extras.c

├── extras.h // fonctions supplémentaires, utiles

├── noeuds.c

├── noeuds.h // structures et fonctions pour les nœuds

├── tables\_symboles.c

└── tables\_symboles.h // structures et fonctions pour les tables de symboles

# 

# 4. DÉVELOPPEMENT

## 4.1. ANALYSE LEXICALE

L’analyse lexicale est la première étape de la compilation, consistant à lire le code en entrée pour le découper en unités significatives appelées lexèmes (ou *tokens*), comme les mots-clés, les identificateurs, les opérateurs, etc. Elle permet de simplifier l’analyse syntaxique en préparant une représentation plus structurée du code. Dans notre projet, nous avons implémenté l’analyseur lexical à l’aide de l’outil Lex, qui utilise des expressions régulières associées à des actions en C pour reconnaître et catégoriser les différents éléments du code. Les *tokens* ainsi générés sont ensuite transmis à l’analyseur syntaxique construit avec Yacc, afin de permettre l’analyse grammaticale du programme.

Pour réaliser notre compilateur, nous avons supprimé dans ANSI-C.l les tokens inutiles pour le langage miniC et nous avons ajouté les commentaires qui sont ignorés par notre compilateur.

## 4.2. ANALYSE SYNTAXIQUE

L’analyse syntaxique est la deuxième étape du processus de compilation, juste après l’analyse lexicale. Elle sert à vérifier que la suite des *tokens* fournie par Lex respecte bien les règles de la grammaire du langage, c’est-à-dire qu’elle est correctement structurée. Cette étape permet de repérer les potentielles erreurs syntaxiques. Lors de cette analyse syntaxique nous construisons également un arbre abstrait, une forme intermédiaire structurée, représentant le code miniC donnée en entrée, qui sera utilisée pour l’étape finale, la génération de code, de notre compilateur . Pour cette partie, nous avons utilisé Yacc, un outil qui permet de décrire la grammaire du langage et d’associer du code C à chaque règle afin de pouvoir créer cet arbre abstrait.

Nous avons apporté plusieurs modifications sur la grammaire initiale mis à disposition dans “miniC.y”.   
 Dans un premier temps, nous avons ajouté les règles *push*, *pop*, *pushf* et *popf* à chaque début et fin de bloc. Elles sont utilisées seulement pour gérer les piles des tables de symbolespour l’analyse sémantique. Cette modification n’impacte pas la syntaxe.

Nous avons également séparé la règle *fonction* en deux partie mais cela ne modifie pas la grammaire, cette modification a pour but de pouvoir déclarer la fonction dans une table de symbole avant de traiter le corps de la fonction (utile pour les fonctions récursives).

Nous avons ajusté la règle *liste\_parms* car il n’était possible d’avoir un seul paramètre que sous la forme “fonction(,param)”. Nous avons donc ajouté la règle *l\_parm* qui permet d’avoir un unique paramètre sans avoir la virgule avant. Cette modification permet également de simplifier la génération de l’arbre abstrait.

La règle *liste\_instructions* a également été modifiée pour simplifier la génération de l’arbre abstrait. Elle renvoie soit vers rien, soit vers le non terminal *l\_instrucions*, qui peut être dérivée en une unique instruction ou en plusieurs instructions.

Dans la règle *selection*, nous avons supprimé les productions

selection -> CASE CONSTANTE ':' instruction | DEFAULT ':'

et nous avons modifié la production

SWITCH '(' expression ')' instruction

en

SWITCH ‘(‘ expression ‘)’ ‘{‘ push liste\_switch\_case pop ‘}’

et nous avons ajouté les nouvelles règles *liste\_switch\_case*, *switch\_case*, *case\_constante* et *default* qui gèrent les case et default avec leurs listes d’instructions. Cela permet dans la syntaxe d’avoir des case et default seulement en présence d’un switch, et permet également une génération correcte de l’arbre abstrait pour les switch.

Pour la règle *variable* nous avons séparé les variables et les tableaux en deux règles différentes pour faciliter la génération de l’arbre abstrait.

Nous avons traité les opérateurs directement dans la règle *expression* pour faciliter la gestion des priorités dans la construction de l’arbre abstrait, et avons donc supprimé la règle *binary\_op*.

La règle *liste\_expression* n’était pas adaptée pour avoir une seule expression à cause d’un ‘,’ obligatoire devant (liste\_expressions ‘,’ expression). Nous avons donc également ajouté la règle *l\_expr* qui prend en compte une expression sans virgule ou plusieurs expressions.  
 Nous avons traité les opérateurs logiques directement dans la règle *condition* pour faciliter la gestion des priorités dans la construction de l’arbre abstrait, et avons donc supprimé la règle *binary\_rel*.

Afin d’assurer les priorités des oérateurs logiques et binaires (“et logique” prioritaire sur “ou logique” ainsi que “et binaire” prioritaire sur “ou logique”) nous avons modifié les déclarations yacc en début de fichier. Nous avons également ajouté la déclaration yacc *%nonassoc MOINSUNAIRE* pour gérer l’opérateur “moins unaire”, et avons supprimé les déclaration yacc *%left OP* et *%left REL* plus utilisées.

Pour générer l’arbre abstrait nous avons implémenté les structures et suivantes :

* La structure tree\_list qui représente une liste chaînée de tree.
* La structure tree qui comprend la racine de l’arbre qui est de type node
* La structure node qui représente un neoud et qui comprend les propriétés nécessaire pour la génération du dot ainsi qu'une node\_list contenant ses fils.
* La structure node\_list qui représente une liste chaînée de node.

Pour manipuler ces structure nous avons également implémenté plusieurs fonctions utiles comme :

* La fonction create\_node() qui crée et renvoie un nœud. Elle prend en paramètre les propriétés utiles pour la génération de code DOT ainsi qu’une node\_list pour ses fils.
* La fonction create\_node\_list() qui permet de créer une liste chaînée d’un nombre quelconque de nœuds.
* etc.

Ces fonctions utiles pour manipuler ces structures sont dans utils/arbres.c et utils/noeuds.c pour tree et tree\_list ainsi que node et node\_list respectivement.

### 4.2.1 Problèmes rencontrés lors de l’analyse syntaxique

Nous avons rencontré quelques difficultés lors de l’analyse syntaxique pour créer l’arbre abstrait, tel que les priorités des opérateurs ou encore la construction des switch. Nous avons finalement réussi, en modifiant la grammaire comme expliquée ci-dessus.

Nous avons également rencontré des problèmes concernant des fuites mémoires, mais avons su les résoudre en analysant les erreurs retournés par l’outil valgrind.

## 4.3. ANALYSE SÉMANTIQUE

L’analyse sémantique a pour rôle de vérifier que le programme reste logique et respecte les règles du langage au-delà de la syntaxe. Par exemple, elle permet de détecter les erreurs comme l’utilisation de variables non déclarées, les incompatibilités de types ou encore des appels fonction pas cohérents. Dans notre projet, cette analyse se produit directement pendant l’analyse syntaxique, grâce aux actions C associées aux règles Yacc.

Nous avons implémenté en C une structure de données sous la forme d’une pile de tables de symboles. Cette structure nous sert à stocker les variables et leurs informations nécessaires pour chaque blocs dans le code. Pour chaque bloc une nouvelle table de symbole est empilée sur la pile, et à la fin de celui-ci la table de symbole est dépilée.

Cette pile de tables de symboles permet de vérifier si les variables sont bien déclarées, pas déclarer plusieurs fois dans un même bloc, le type des variables, la dimension des variables (tableaux) et si l'accès à un tableau ne dépasse pas les limites.

Nous avons également une pile de tables de symboles différentes pour les fonctions, qui permet de vérifier le type d’une fonction, si elle comporte un return ou non, si besoin, si elle n’est pas déclarée une seconde fois plus loin dans le code ou encore si elle comporte bon nombre de paramètres lors d’un appel.

Pour connaitres les différentes variables et fonctions déclarées, nous avons implémenter les structures suivantes :

* La structure table\_t représentant la pile de table de symboles contenant les symboles de la table ainsi que la référence vers la prochaine table dans la pile.
* La structure symbole\_t représentant une liste chaînée de symboles avec les caractéristiques nécessaire à l’analyse sémantique

Pour la gestion de ces structures, nous avons implémenter les fonctions suivantes :

* push\_table() qui empile une nouvelle table de symbole.
* pop\_table() qui dépile la table de symbole au sommet
* declarer() pour ajouter le symbole avec ses caractéristiques à la table de symbole.
* rechercher() pour rechercher un symbole dans table au sommet de la pile
* rechercher\_dans\_pile() pour rechercher un symbole dans la pile.
* ect.

### 4.3.1 Problèmes rencontrés lors de l’analyse sémantique

Au cours du développement de l’analyse sémantique nous avons rencontré des difficultés pour implémenter entièrement la division par zéro. En vue de la difficulté pour déterminer la valeur d’une expression, qui peut être un calcul complexe contenant des appels de fonctions, nous avons jugé ne pas avoir le temps nécessaire pour l’implémenter pleinement. Ainsi nous avons implémenté uniquement une erreur sémantique pour les divisions par la constante zéro.

Nous avons également eu des difficultés à libérer proprement toute la mémoire allouée pour les différentes structures. Nous avons réglé ce problème en analysant les fuites retournées par valgrind.

Pour finir, nous avons eu des problèmes à gérer les “switch”. Nous avons eu des difficultés à vérifier l’unicité de chaque case ainsi que la présence d’un unique default. Dans un premier temps nous avons implémenté cette vérification à l’aide d’une variable globale qui comptait le nombre de default, mais cela ne fonctionne pas avec des switch imbriqués. Ces derniers ont également posé problème pour vérifier l’unicité des case, où au début, on avait décidé de stocker les constantes de chaque case dans une liste pour vérifier si un case contenant la même constante existait déjà. Finalement, nous avons réussi à résoudre ce problème à l'aide des piles de tables de symbole. Nous avons ajouté dans la table de symbole un symbole par case ainsi que le default, ce qui nous a permis de tout vérifier pour chaque blocs.

## 4.4. GÉNÉRATION DE CODE

Une fois l’analyse syntaxique terminée et l’arbre abstrait construit, nous effectuons une seconde passe qui parcourt cet arbre abstrait. Cette étape a pour but de générer le code DOT, un langage utilisé pour représenter des graphes. En visitant chaque nœud de l’arbre abstrait, nous produisons automatiquement le code DOT. Ce code DOT est une forme intermédiaire sous forme de graphe, qui peut être visualiser à l’aide d’outils comme Graphviz.

Nous avons respecté les différents attributs donnés dans les consignes :

* invtrapezium bleu pour les fonctions.
* trapezium bleu pour les return.
* diamond noir pour les IF.
* box noir pour les BREAK.
* septagon noir pour les appels de fonctions.
* ellipse pour le reste.

Le code généré est écrit dans un fichier au format dot du même nom que le code à compiler. En utilisant le Makefile pour exécuter le compilateur sur le fichiers tests, un fichier pdf est également créé pour visualiser chaque fichier dot à l'aide de l’outil graphviz. Tout fichier dot et pdf est généré dans le même répertoire que le fichier test compilé.

# 5. ERREURS DÉTECTÉS PAR L’ANALYSE SÉMANTIQUE

## 5.1. ERREURS

Notre compilateur reconnait les erreurs sémantiques suivantes:

* Division directe par la constante 0
* Variables déjà déclarés dans le même bloc.
* Fonctions déjà déclarées.
* Deux “default” dans un “switch”
* Deux “case n” identiques dans un “switch”
* Utilisation d’une variable ou d’un tableau avec un nombre de dimensions supérieur à celui défini.
* Déclaration d’une variable de type void

## 5.2. WARNINGS

Notre compilateur génère les warnings sémantiques suivants:

* Absence de “return” dans les fonctions de type int.
* Présence de “return expr;” dans les fonctions de type void.
* Utilisation de variables non déclarées.
* Utilisation de fonctions non déclarées.
* Taille d’un tableau non respecté.
* Appel a une fonction void dans une expression de type int
* Fonction appelée avec le mauvais nombre de paramètres.

Nous n’avons pas implémenter entièrement l’erreur pour la division par 0 pour des expressions à valeur égale à zéro, seulement la division par la constante 0, car il aurait fallu interpréter les valeurs des expressions, qui peuvent comprendre des appels de fonctions ou encore des calculs conséquents.

# 

# 6. CONCLUSION

Ce projet nous a permis de mettre en pratique un maximum de notions sur les compilateurs vus en cours. Travailler avec Lex et Yacc nous a demandé un peu de temps afin de bien comprendre la logique qui se cache derrière ces outils. A l’aide des TDs, nous avons pu comprendre les différentes étapes du compilateur tel que l’analyse lexicale, l’analyse syntaxique, sémantique ou encore la génération de code qui restait jusque-là assez abstraites.

Ce travail nous a également permis de mieux comprendre ce qui se cache réellement derrière le fonctionnement d’un compilateur. En passant par chaque étape de la chaîne de compilation, nous avons pris conscience de la complexité et de la précision nécessaires pour transformer un simple fichier de code source en un programme exécutable. Ce qui nous paraissait auparavant automatique comme la détection d’erreurs de syntaxe ou la traduction du code s’est révélé être un processus rigoureux reposant sur des mécanismes bien définis. Cela nous a donné une toute nouvelle perspective sur les outils de programmation que nous utilisons quotidiennement.

# 

# 7. ANNEXES

## 7.1 EXEMPLES DE SCHÉMAS DE LA GÉNÉRATION DE L’ARBRE ABSTRAIT

Ci-dessous se trouvent quelques exemples de schémas (squelette) de la génération de l’arbre abstrait.

### 7.1.1 Génération de l’arbre abstrait pour la règle *expression*

expression:

‘(‘ expression ‘)’

|  |
| --- |
| <$2.code> |
| $$.noeud = $2.node |

expression:

expression OP expression

|  |
| --- |
| <$1.code> |
| <$3.code> |
| fils = create\_node\_list(2,$1.node,$3.node)  $$.noeud = create\_node(OP,fils) |

Afin de ne pas répéter les schémas nous considérons que OP peux remplacer n’importe quelle opérateur tel que +,-,/,\*,<<,>>,& et |.

expression:

MOINS expression %prec MOINSUNAIRE

|  |
| --- |
| <$2.code> |
| fils = create\_node\_list(1,$2.node)  $$.noeud = create\_node(“-”,fils) |

expression:

CONSTANTE

|  |
| --- |
| $$.noeud = create\_node($1,NULL) |

expression:

variable

|  |
| --- |
| <$1.code> |
| $$.noeud = $1.node |

expression:

IDENTIFICATEUR ‘(‘ liste\_expressions ‘)’

|  |
| --- |
| <$3.code> |
| $$.noeud = create\_node($1,$3) |

### 7.1.2 Génération de l’arbre abstrait pour la règle *selection*

selection:

IF ‘(‘ condition ‘)’ instruction %prec THEN

|  |
| --- |
| <$3.code> |
| <$5.code> |
| fils = create\_node\_list(2,$3,$5)  $$.noeud = create\_node(“IF”,fils) |

selection:

IF ‘(‘ condition ‘)’ instruction ELSE instruction

|  |
| --- |
| <$3.code> |
| <$5.code> |
| <$7.code> |
| fils = create\_node\_list(3,$3,$5,$7)  $$.noeud = create\_node(“IF”,fils) |

selection:

SWITCH ‘(‘ expression ‘)’ ‘{‘ push liste\_swtitch\_case pop ‘}’

|  |
| --- |
| <$3.code> |
| <$7.code> |
| fils = new\_empty\_node\_list()  fils.item = $3  fils.suivant = $7  $$.noeud = create\_node(“SWITCH’,fils) |

liste\_switch\_case:

switch\_case

|  |
| --- |
| <$1.code> |
| $$.liste\_noeuds = create\_node\_list(1,$1) |

liste\_switch\_case:

liste\_switch\_case switch\_case

|  |
| --- |
| <$1.code> |
| <$2.code> |
| $$.liste\_noeuds = add\_node\_to\_list($1,$2) |

switch\_case:

case\_constante ‘:’ liste\_instructions

|  |
| --- |
| <$1.code> |
| <$3.code> |
| $$.noeud = create\_node($1.var,$3) |

switch\_case:

default ‘:’ liste\_instructions

|  |
| --- |
| <$1.code> |
| <$3.code> |
| $$.noeud = create\_node(“case default”,$3) |

case\_constante:

CASE CONSTANTE

|  |
| --- |
| $$.var = concat(2,”case ”, $2 ) |

### 7.1.3 Génération de l’arbre abstrait pour la règle *condition*

condition :

NOT ‘(‘ condition ‘)’

|  |
| --- |
| <$3.code> |
| fils = create\_node\_list(1,$3)  $$.noeud = create\_node(“NOT”,fils) |

condition:

condition LAND condition

|  |
| --- |
| <$1.code> |
| <$3.code> |
| fils = create\_node\_list(2,$1,$3)  $$.noeud = create\_node(“&&”,fils) |

condition:

condition LOR condition

|  |
| --- |
| <$1.code> |
| <$3.code> |
| fils = create\_node\_list(2,$1,$3)  $$.noeud = create\_node(“||”,fils) |

condition:

‘(‘ condition ‘)’

|  |
| --- |
| <$1.code> |
| $$.noeud = $2.noeud |

condition:

condition binary\_comp condition

|  |
| --- |
| <$1.code> |
| <$3.code> |
| fils = create\_node\_list(2,$1,$3)  $$.noeud = create\_node($2.var,fils) |

### 7.1.4 Génération de l’arbre abstrait pour la règle *affectation*

affectation :

variable ‘=’ expression

|  |
| --- |
| <$1.code> |
| <$3.code> |
| fils = create\_node\_list(2,$1,$3)  $$.noeud = create\_node(“:=”,fils) |

### 

### 7.1.5 Génération de l’arbre abstrait pour la règle *variable*

variable :

IDENTIFICATEUR

|  |
| --- |
| $$.noeud = create\_node($1,NULL) |

variable :

tableau

|  |
| --- |
| <$1.code> |
| $$.noeud = $1.noeud |

tableau :

IDENTIFICATEUR ‘[‘ expression ‘]’

|  |
| --- |
| <$3.code> |
| n = create\_node($1,NULL)  $$.liste\_noeuds = create\_node\_list(2,n,$3) |

tableau :

tableau ‘[‘ expression ‘]’

|  |
| --- |
| <$1.code> |
| <$3.code> |
| $$.liste\_noeuds = add\_node\_to\_list($1,$3) |