Université Mohammed V - Rabat
École Nationale Supérieure d'Informatique
et d'Analyse des Systèmes

Rapport de Projet de Compilation

Filière

Génie Logiciel

Conception d'un langage et réalisation d'un compilateur avec le langage C

Encadré par :

Pr. Youness TABII

Pr. Rachid OULAD HAJ THAMI

Réalisé par :

ELHAYYANI Haitam

LAKHDACHI Yassine

ROUAI Chaimaa

SAMAD Houda

Année universitaire : 2022/2023

Remerciements

C'est avec un grand plaisir que nous réservons ces quelques lignes en signe de gratitude et de profonde reconnaissance à tous ceux qui, de près ou de loin, ont contribué à la réalisation et laboutissement de ce travail. Tout dabord nous remercions nos encadrants **Mr OULADY HAJ THAMI Rachid** et **Mr TABII Youness** pour leur soutient et leur aide tout au long de ce travail.

Table des matières

Table des figures

Introduction générale

Après l'étape de l'écriture d'un code en un langage de programmation donné, l'étape de sa compilation s'impose. Afin de vérifier les erreurs de la syntaxe et traduire le code en un langage de bas niveau compréhensible par notre machine, nous pouvons se servir d'un compilateur dédié. En vrai sens du mot, un compilateur est un programme informatique composé généralement de trois majeurs composants : analyseur lexical, syntaxique et sémantique.

A travers ce projet de compilation, nous aurons l'occasion de développer notre propre langage ainsi que son compilateur composé de l'analyseur lexical, syntaxique.

Nous présentons en première partie la grammaire de notre langage, les différents tokens, en deuxième partie l'analyseur lexical puis en dernière partie l'analyseur syntaxique.

Description de la grammaire

1.1 La grammaire

Terminaux	Règles
Prog	Lib { Const | Var | Func } Main
Biblio	Bibliotheques : | [lib : ID{,ID} ;] | [h : ID{,ID} ;] | e
Const	Constants : { ID = Value ; } | e
Var	Variables : { ID [= Value] | e ; } | e
Func	Fonctions : [{ ID ({ ID | ID,}) { Insts | Return } }] | e
Prince	Principale ({ID | ID,}) { Insts }
Value	int | long | float | double | string | char | List | File |
Int	Num { Num }
Long	Num { Num } L
Float	Num { Num } [, Num { Num }] | e F
Double	Num { Num } [, Num { Num }] | e D
String	" { Alpha | Num | Special } "
Char	'Alpha'
Liste	Liste < { ID | value | ID, |value , | } >
Fichier	FICHIER (string , 'AccessType') ;

AccessType	r | w
Insts	Inst { Inst | Return }
Inst	CallFunction ; | Decision | Loop | Affectation
Affectation	ID { [num] } = { expr | value} ;
CallFunction	ID ({ ID | ID, })
Condition	(expr Comp_Op expr) {logic$_o p(exprcom_opexpr)$}
Decision	si (condition) { Insts } {sinonSi { Insts } } { Sinon {insts}}
Loop	for ID in Value { Insts } | for ID in (Num, Num) { Insts } | while (condition){ Insts } | do{ Ints }while (condition) ;
Expr {TERM [+|-] TERM || callFunction || value	
FACT	ID{ [num] } | NUM | (EXPR) | CallFunction
Return	return expr ;
Operator	Comp_Op | Arith_Op | Assign_Op | Logic_Op
Comp_Op	== | < | <= | > | >= |!=
Arith _Op	+ | - | / | * | ^ | %
Assign_Op	=
Logic_Op	& ||
ID	_| lettre { _| lettre | chiffre }
Alpha	a | .. | z | A | .. | Z
Num	0 | .. | 9
Special	Tous les caractères spéciaux

1.2 Tokens et symboles

1.2.1 Tokens de la structure de programme

Nom	Symbole
LIBRARIES_TOKEN	Libraries
CONST_TOKEN	Constants
VAR_TOKEN	Variables
FUNC_TOKEN	Functions
MAIN_TOKEN	Main

1.2.2 Tokens de types des données

INT_TOKEN	Int
FLOAT_TOKEN	Float
STRING_TOKEN	String
DOUBLE_TOKEN	Double
CHAR_TOKEN	Char
LONG_TOKEN	Long
LIST_TOKEN	List
FILE_TOKEN	File

1.2.3 Tokens des opérations arithmétiques

Nom	Symbole
PLUS_TOKEN	+
MOINS_TOKEN	-
DIV_TOKEN	/
MOD_TOKEN	%
MULT_TOKEN	*
EG_TOKEN	==
DIFF_TOKEN	!=
INFEG_TOKEN	<=
INF_TOKEN	<
SUPEG_TOKEN	>=
SUP_TOKEN	>
AFF_TOKEN	=

1.2.4 Tokens des opérateurs logiques

Nom	Symbole
AND_TOKEN	&
OR_TOKEN	|

1.2.5 Tokens des symboles spéciaux

Nom	Symbole
BO_TOKEN	[
BF_TOKEN	]
PO_TOKEN	(
PF_TOKEN	)
PV_TOKEN	;
DP_TOKEN	:
VIR_TOKEN	,
CBO_TOKEN	{
CBF_TOKEN	}
ST_TOKEN	"

1.2.6 Tokens des conditions

Nom	Symbole
IF_TOKEN	if
ELIF_TOKEN	elif
ELSE_TOKEN	else

1.2.7 Tokens des loops

Nom	Symbole
WHILE_TOKEN	while
DO_TOKEN	do
FOR_TOKEN	for
IN_TOKEN	in

1.2.8 D'autres tokens

Nom	Symbole
ID_TOKEN	
NUM_TOKEN	
EOF_TOKEN	
LIB_TOKEN	lib
H_TOKEN	h
RETURN_TOKEN	return

Réalisation et mise en oeuvre

Ce chapitre aborde la mise en oeuvre et la réalisation, en présentant les outils de réalisation ainsi que le travail réalisé.

2.1 Outils de développement

Lune des raisons très fortes pour lesquelles le langage de programmation C est si populaire et utilisé si largement est la flexibilité de son utilisation pour la gestion de la mémoire. Les programmeurs ont la possibilité de contrôler comment, quand et où allouer et désallouer la mémoire. La mémoire est allouée de manière statique, automatique ou dynamique dans la programmation C à laide des fonctions malloc et calloc. Initialement, C a été conçu pour implémenter le système dexploitation Unix. La plupart du noyau Unix, et tous ses outils et bibliothèques de support, ont été écrits en C. Plus tard, dautres personnes lont trouvé utile pour leurs programmes sans aucune entrave, et ils ont commencé à lutiliser. Une autre bonne raison dutiliser le langage de programmation C est quil se trouve à proximité du système dexploitation. Cette fonctionnalité en fait un langage efficace car les ressources au niveau du système, telles que la mémoire, sont facilement accessibles, ce qui nous a facilité de créer un compilateur.

2.2 Fonctionnement du Compilateur

Le compilateur récupère en entrée le trajet complet du fichier où se trouve le programme écrit dans le language cbl, ce fichier aura l'extension .cbl.

Avant d'entamer les étapes de compilation, le compilateur s'assure que le fichier est non vide, puis passe à la première étape qui est l'analyse lexicale. Le compilateur recourt à l'exécutable de cet analyseur et lui fournit le trajet du fichier programme. A la fin de cette étape on aura comme sortie un fichier contenant les erreurs, ainsi qu'un autre contenant les tokens générés par l'analyseur lexicale.

Dans le cas où le fichier erreur soit non vide, le compilateur procède comme suit : il renvoie à l'écran les erreurs trouvés ainsi que la ligne où se trouve, puis termine la compilation. Dans l'autre cas, le compilateur procède à la deuxième étape qui est l'analyse syntaxique, et de la même manière qu'auparavant, le compilateur fait appel à l'éxécutable de l'analyseur syntaxique en lui fournissant cette fois deux fichiers celui du programme et celui des tokens.

Le compilateur attend la fin de l'exécution pour vérifier si des erreurs syntaxiques existent, dans ce cas ils les affichents et se termine. Dans l'autre cas on aura une compilation réussi.

2.3 Développement du compilateur

2.3.1 Analyseur Lexicale

- **Char NextChar()** : Lit le prochain charactère et le retourne.
- **void SaveToken()** : Enregistre dans un fichier, le token actuel sous forme de texte.

- **void LexError(const char* message)** : Enregistre l'erreur dans un fichier, et ignore la ligne actuel du programme.
- **void ignoreWhiteSpaces()** : Ignore les séparateurs comme les tabulations, les espaces et les nouvelles lignes.
- **void ignoreComment()** : Ignore un seul commentaire.
- **void getCurrentWord()** : Lit tout un mot depuis le charactère actuel.
- **bool SearchForToken()** : Reçoit une liste de tokens et un mot et cherche sa correspondence dans cette liste
- **bool isNumber()** : Construit le nombre et vérifie si cest un INT, LONG, FLOAT, DOUBLE. Sinon elle le traite comme un invalide Nombre.
- **bool isBloc()** : Vérifie si le token correspond aux différents étages du programme : Libraries, Constants, Variables, Functions, Main
- **bool isReturn()** : Vérifie si le token correspond au mot clé "return"
- **bool isString()** Construit le mot, et vérifie si respecte la forme suivante <" texte ">
- **bool isIdentifier()** : Il n'y a pas de comparaison dans cette fonction, parceque dans ce cas tous les tests sur le token qui commence avec alphabet ont échoué, donc il ne reste que le déclaré comme Identificateur.
- **bool isDataType()** : Deux types de données qui se déclare explicitement et qui sont <list, FILE>
- **bool isConditionOrLoop()** : Vérifie si le Token appartient aux tokens réservés pour les conditions et les boucles.
- **bool isCharacter()** : Un token de type charactère respecte la forme suivante <'c'> avec c un charactère quelconque.
- **bool isCharacter()** : Vérifie si cest opérateur correspond à la division, la

multiplication,la soustraction, laddition etc...

- **bool isSpecialSymb()** : Vérifie si cest un charactère spécial.

2.3.2 Analyseur Syntaxique

- **Void getToken()** : A chaque appel, la fonction getToken donne le token suivant.
- **int error(char* message)** : Sassure quil ny a pas derreurs syntaxiques. Dans lautre cas elle enregistre un message derreur dans un fichier ń error.txt ż et saute les token reçu jusqua la fin de linstruction erronée.
- **void program()** : Vérifie le token courrant , appelle les fonctions correspondantes au bloc qui représente le token.
- **void libraries()** : Teste si lexpression représente soit la déclaration dune librairie ou bien dun header.
- **void constants()**: : Vérifie la syntaxe de déclaration des constants dans le bloc constants().
- **void variables()** : Assure que la déclaration des variables est bien faite.
- **void functions()** : Teste si lexpression actuel représente une déclaration de fonction sous la forme : Function(arguments)instructions.
- **void Insts()** : Appel la fonction Inst() , tanque cette dernière return true.
- **void Insts()** : vérifie tous sorte dinstructions qui peuvent se produire et fait appel aux fonctions correspondantes à chaque instruction.
- **void list()** : vérifie si laffectation dune donnée à un élément de la liste est valide.
- **void callFunction()** : Teste si lexpression dappel fonction avec ses arguments est valide.

- **void condition()** : Teste si lexpression actuel représente la syntaxe dune instruction conditionnelle qui est sous cette forme : (ID | CallFunction | Value) CompOp (ID | CallFunction | Value).
- **void decision()** : Teste si lexpression actuel représente une instruction conditionnelle sous la forme de : if (condition) expressions ou bien if (condition) expressions elif expressions.
- **void loop()** : Teste si linstruction courante représente lune des trois instructions de boucle (voir grammaire).
- **void expr()** : Vérifie si la syntaxe représentée dans la grammaire de lexpression est valide. Voir la grammaire de ń expr ż ;
- **void exprBegin()** : Vérifie si la syntaxe représentée dans le début de la grammaire de lexpression est valide. Si oui il passe le token suivant a la fonction expr() ;
- **void term()** : Vérifie si la syntaxe représentée dans la grammaire de Term est valide. Voir la grammaire de ń TERM ż ;
- **void fact()** : Teste si le token actuel représente soit Id , NUM ou bien une expression.
- **void is_value()** : Vérifie si le type de données est parmi les types suivants num , char, string,list ou bien FILE
- **void main()** : Teste si lexpression actuel représente la déclaration de la fonction main sous la forme : Main arguments instructions

Conclusion générale et Future Amélioration

Ce projet nous a permis de parfaitement comprendre la chaîne de compilation dun code. Et nous a donné loccasion de découvrir énormément de problèmes liés notamment à lanalyse, qui nétaient pas aussi évidents dans la théorie du cours.

Nous estimons avoir passé environ 3 semaines complètes pour lachèvement de ce compilateur. Les deux premiers jours on sest focalisé sur le regroupement de la grammaire, ensuite les tâches se sont divisées pour limplémentation des analyseurs.

Nous ne pensions pas passer autant de temps sur la conception de la grammaire, qui nous a valu de recommencer plusieurs fois et de passer de nombreuses heures à chercher la bonne méthode pour lever les ambiguïtés, sans en soulever d'autres.

La difficulté en général réside dans limplémentation de la grammaire lors la réalisation de lanalyseur syntaxique, ensuite la sémantique pourra être facilement vérifié en se basant sur la table de symbole généré par lanalyseur syntaxique.

Nous sommes ravis de vous annoncer nos perspectives et les nouvelles amé-

liorations qui seront apportées à notre compilateur :

— Ajout des qualificateurs qui détermine le champ de visibilité des variables et des fonctions ainsi que la durée de vie tel que les mots clés : public, private, static, automatic.

— Ajout de lanalyseur sémantique.

— Indiquer précisemment où se trouve l'erreur en indiquant la colonne et la ligne.

— Ajout des fonctions prédéfinies standard.

— Création du préprocesseur et ajout des mots clés des instructions macro tel que define, ifndef, ifdef, endif

— Créer une application qui installera notre compilateur sur le système d'exploitation.