Université des Sciences et de la Technologie Houari Boumediene
Faculté d'Electronique et d'Informatique
Département d'Informatique

Master 1 : IL-IV

Année Universitaire : 2019/2020
Module Compilation

Projet de compilation du langage R
Avec les Outils FLEX et BISON

1. Introduction :

Le but de ce projet est de réaliser un mini-compilateur en langage C passant par les différentes phases de la compilation à savoir l'analyse lexicale en utilisant l'outil FLEX et l'analyse syntaxico-sémantique en utilisant l'outil BISON, la génération du code intermédiaire, l'optimisation ainsi que la génération du code machine.
Les traitements parallèles concernant la gestion de la table des symboles ainsi que le traitement des différentes erreurs doivent être également réalisés lors des phases d'analyse du processus de compilation.

2. Description du langage R :

Un programme en R est composé d'une suite de déclarations et d'instructions. Chaque instruction doit être sur une seule ligne (et elle ne se termine pas par un « ; »).

Exemple :

```
#Le programme calcule le produit des N premiers nombres entiers
INTEGER N ← 50
P ← 0
FOR (i IN 1:10)
{
P ← P * i
}
```

2.1. Commentaire :

Un commentaire peut être écrit sur une ou plusieurs lignes. Chaque ligne du commentaire doit être précédée par un « # ».

Exemple : # ceci est un commentaire

2.2. Déclarations :

Une déclaration peut être de variables simples (entiers, réels, booléens et caractères) ou bien de variables structurées (tableaux).
Le type peut ne pas être spécifié mais dans ce cas, la variable doit être initialisée à une valeur (avec laquelle on devinera le type) lors de la déclaration.

2.3. Types :

TYPE	Description	Exemples de VALEURs
INTEGER	Une **variable** de type INTEGER représente un entier. Une **valeur** de type INTEGER peut être signée ou non signée tel que sa valeur est entre **-32768** et **32767**. Si la valeur entière est signée, elle doit être mise entre parenthèses.	(-3520), 0, 5210, …
NUMERIC	Une **variable** de type NUMERIC représente un réel. Une **valeur** de type NUMERIC peut être signée ou non signée. Si la valeur réelle est signée, elle doit être mise entre parenthèses. La partie entière et la partie décimale sont séparées par un point.	(-3520.34), 5210.17, …
CHARACTER	Une **variable** de type CHARACTER représente un caractère. Une **valeur** de type CHARACTER doit être entre deux apostrophes.	'a', 'b', …, 'z', 'A', 'B', 'C' ... 'Z', '#', ' ', '%', ...
LOGICAL	Une **variable** de type LOGICAL représente un booléen. Une **valeur** de type LOGICAL à deux valeurs possibles : TRUE ou FALSE.	TRUE, FALSE

2.4. Identificateur :

Un identificateur (IDF) est une suite alpha-numérique qui commence par une lettre majuscule suivie d'une suite de chiffres et lettres minuscules. Un IDF ne doit pas contenir plus de 10 caractères. Les noms du programme principal, des variables et des constantes sont des identificateurs.

2.4.1. Déclaration des variables de type simple :

La déclaration de variables a les trois formes suivantes :

Syntaxe :

Forme 1 : **TYPE** *Liste_IDFs*
Forme 2 : **TYPE** <*IDF*> ← **VALEUR**
Forme 3 : <*IDF*> ← **VALEUR**

Liste_IDFs : ensemble d'identificateurs séparés par une virgule.

Exemple :

```
# Forme 1
NUMERIC x
INTEGERy, z

# Forme 2
INTEGER x ← 2020

# Forme 3
x ← 19.5
```

2.4.2. Déclaration des tableaux :

La déclaration d'un tableau a la forme suivante :

Syntaxe :

TYPE *<IDF> [<Taille>] ;*

2.5. Expressions :

Les expressions peuvent être arithmétiques, de comparaison, ou logiques.

. Une expression arithmétique est une combinaison de variables (IDFs) ou de valeurs de type INTEGER ou NUMERIC et d'opérateurs arithmétiques.

. Une expression de comparaison (condition) est une combinaison de variables, valeurs ou d'expressions arithmétiques et d'opérateurs de comparaison. Une expression de comparaison doit être mise entre parenthèses.

. Une expression logique est une combinaison de conditions et d'opérateurs logiques.

Pour composer les expressions arithmétiques et logiques, on utilise les opérateurs indiqués dans le tableau suivant :

Type d'expression	Opérateurs	Exemple d'expression
Expression arithmétique	Addition : + Soustraction : – Multiplication : * Division : / Reste : %	$x * x + 4 * (y/z)$ $y * 2.51 - x$ $x\%5 - (3.54/y) * z$
Expression de comparaison (condition)	Supérieur : > Inférieur : < Egal : == Différent : != Supérieur ou égal : >= Inférieur ou égal : <=	$(x > 5)$ $(x * 8\%y \leq z)$ $(z < x/2.5)$ $(6! = x - 7)$
Expression logique	Le « et » logique : **and** Le « ou » logique : **or**	$(x/5 > 1) \vee (y + z \leq 8.3)$ $(x + 1 < y) \wedge (y\%5! = 1) \vee (z \geq y)$

2.6. Instructions :

Instruction	Description	Exemple
Affectation	<IDF> ← <expression> # Une <expression> peut être une valeur, une expression # arithmétique ou logique	$f \leftarrow 'L'$ $d \leftarrow b * b - 4 * a * c$ $e \leftarrow (x > 8)$

Instruction	Description	Exemple
Incrémentation & Décrémentation	<IDF>+ ← <Valeur> <IDF> - ← <Valeur> # La <Valeur> doit être de type INTEGER positif # <IDF>+ ← <Valeur> égale à <IDF> ← <IDF> + <Valeur> # <IDF> - ← <Valeur> égale à <IDF> ← <IDF> - <Valeur>	$i+\leftarrow 2$ $i-\leftarrow 1$

Instruction	Description	Exemple

Condition
(IF …)
	IF (Expression_Logique)
	{
	<Bloc_Instructions>
	}
	
	# Le <Bloc_Instructions> est exécuté si l'<Expression_Logique>
	# est vérifiée.
		IF $(promo = 2020)$
		{
		Best← $TRUE$
		}

Instruction	Description	Exemple

Condition
(IF … ELSE …)
	IF (Expression_Logique)
	{
	<Bloc_Instructions_1>
	}
	ELSE
	{
	<Bloc_Instructions_2>
	}
	
	# Le <Bloc_Instructions_1> est exécuté si l'<Expression_Logique>
	# est vérifiée.
		IF $(promo = 2020)$
		{
		Best← $TRUE$
		}
		ELSE
		{
		Best← $FALSE$
		}

Condition
(IF … ELSE IF … ELSE …)
	IF (Expression_Logique_1)
	{
	<Bloc_Instructions_1>
	}
	ELSE IF (Expression_Logique_2)
	{
	<Bloc_Instructions_2>
	}
	ELSE
	{
	<Bloc_Instructions_3>
	}
		IF $(x > 0)$
		{
		$y \leftarrow z/x$
		}
		ELSE IF $(x < 0)$
		{
		$y \leftarrow z$
		}
		ELSE
		{
		$y \leftarrow 0$
		}

Instruction	Description	Exemple
Affectation avec condition (IFELSE)	IDF ← IFELSE(<Expression_Logique>, <expression_1>, <expression_2>) # IDF reçoit <expression_1> si l'<Expression_Logique> est # vérifiée.	$y \leftarrow \big((z > 0), x, 1\big)$

Instruction	Description	Exemple

Boucle
(FOR)
	FOR (IDF **IN** <borne_Inf>**:** <borne_Sup>)
	{
	<Bloc_instructions>
	}
		FOR $(i$ **IN** $1{:}10)$
		{
		$P \leftarrow P * i$
		}

Instruction	Description	Exemple

Boucle (WHILE)	**WHILE** (<Expression_Logique>):
	{
	<Bloc_instructions>
	}
		WHILE $(i < N)$
		{
		$i \leftarrow i + 2$
		}

2.7. Associativité et priorité des opérateurs :

Les associativités et les priorités des opérateurs sont données par la table :

	Opérateur(s)	Associativité	Ordre de priorité
Parenthèses	()	Gauche	**-**
Opérateurs logiques	and		
	or		
Opérateurs de comparaison	$\geq = ! =<$		
Opérateurs arithmétiques	$+ -$		
	$*/\%$		**+**

3. Travail à réaliser :

Ci-dessous les différentes phases à effectuer afin de réaliser le compilateur demandé.

3.1. Analyse Lexicale avec l'outil FLEX :

Son but est d'associer à chaque mot du programme source la catégorie lexicale à laquelle il appartient et de générer la table des symboles. Pour cela, il est demandé de définir les différentes entités lexicales à l'aide d'expressions régulières et de générer le programme FLEX correspondant.

3.2. Analyse syntaxico-sémantique avec l'outil BISON :

Pour implémenter l'analyseur syntaxico-sémantique, il va falloir écrire la grammaire qui génère le langage défini ci-dessus. La grammaire associée doit être LALR. En effet l'outil BISON est un analyseur ascendant qui opère sur des grammaires LALR. Il faudra spécifier dans le fichier BISON les différentes règles de la grammaire ainsi que les règles de priorité pour les opérateurs afin de résoudre les conflits. Les routines sémantiques doivent être associées aux règles dans le fichier BISON.

3.3. Gestion de la table de symboles :

La table de symboles doit être créée lors de la phase de l'analyse lexicale. Elle doit regrouper l'ensemble des variables et constantes définies par le programmeur avec toutes les informations nécessaires pour le processus de compilation, tels que la **nature** (variable, valeur ou tableau), le **type**, la **taille du tableau** (la taille est égale à 1 pour les variables simples et les valeurs). Cette table sera implémentée sous forme d'une table de hachage. Elle sera mise à jour au fur et à mesure de l'avancement de la compilation. Il est demandé de prévoir des procédures pour permettre de **rechercher** et d'**insérer** des éléments dans la table des symboles. Les variables structurées de type tableau doivent aussi figurer dans la table des symboles.

3.4. Optimisation :

On considère quatre types de transformations successives appliquées au code intermédiaire.

. **Propagation de copie** (ex. remplacer $t1 = t2; t3 = 4 * t1$ par $t1 = t2; t3 = 4 * t2$).

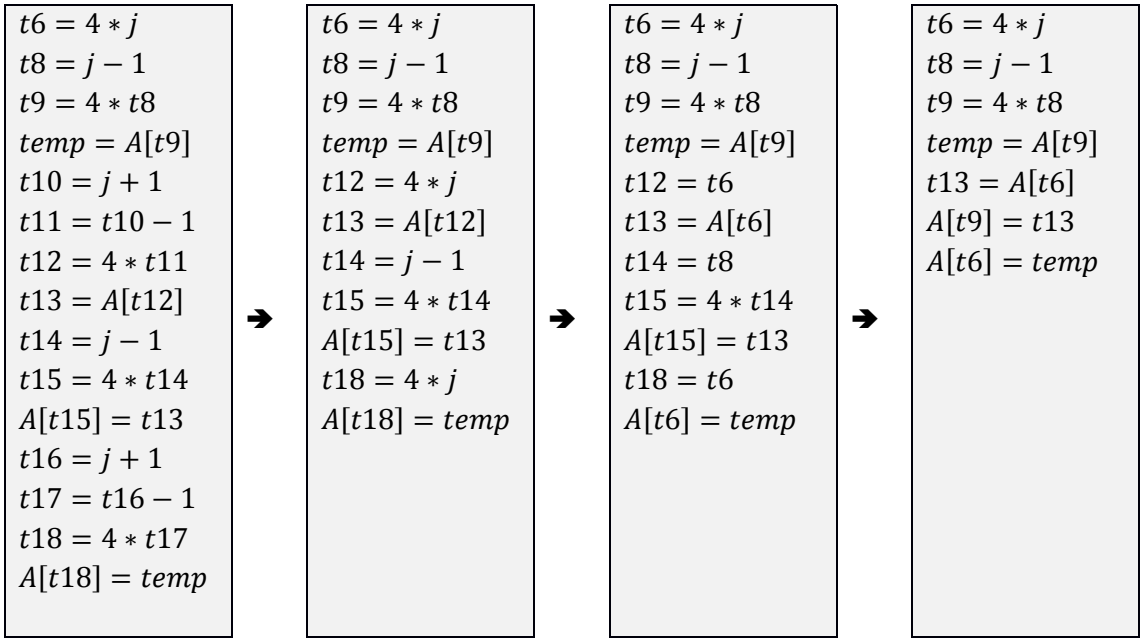
. **Propagation d'expression** (ex. remplacer $t1 = expr; t3 = 4 * t1$ par $t1 = expr; t3 = 4 * expr$).

. **Élimination d'expressions redondantes** (communes) (ex. remplacer $t6 = 4 * j; t12 = 4 * j$ par $t6 = 4 * j; t12 = t6$).

. **Simplification algébrique** (ex. remplacer $t1 + 1 - 1$ par $t1$).

. **Élimination de code inutile** (code mort).

Exemple :



3.5. Génération du code intermédiaire :

Le code intermédiaire doit être généré sous forme de quadruplets. Par exemple, l'instruction $x = y + z$ est traduite par le quadruplet $(+, y, z, x)$. Les quadruplets générés passent par la phase d'optimisation (voir section 3.4).

3.6. Génération du code machine :

Le code machine doit être généré en assembleur.

3.7. Traitement des erreurs :

Il est demandé d'afficher les messages d'erreurs adéquats à chaque étape du processus de compilation. Ainsi, lorsqu'une erreur lexicale ou syntaxique est détectée par votre compilateur, elle doit être signalée le plus précisément possible, par sa nature et sa localisation dans le fichier source. On adoptera le format suivant pour cette signalisation :

Type de l'erreur, line 4, colonne 56, entité qui a généré l'erreur.