Projet de compilation

Avec les Outils FLEX et BISON
Master I

1. Introduction

Le but de ce projet est de réaliser un mini-compilateur en effectuant 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, du langage « Lang». Les traitements parallèles concernant la gestion de la table des symboles ainsi que le traitement des différentes erreurs doivent être également réalisés lors des différentes phases d'analyse du processus de compilation.

2. Description du Langage MiniLang

La structure générale d'un programme écrit en langage Lang se décompose de deux parties : (1) une partie pour déclarer les variables utilisées et (2) une partie instructions qui contiennent elles-mêmes une ou plusieurs instructions. La structure générale est présentée comme suit:

```
< ! docprogram  ID_program>
< SUB  VARIABLE >
     Partie declaration
</ SUB  VARIABLE >
<body>
      Partie instruction
</body>
</docprogram>
```

La partie déclaration : La partie déclaration permet de **déclarer** toutes les variables et les constantes utilisées par le programme.

```
< SUB  VARIABLE >
     Déclaration de la première variable
     Déclaration de la deuxième variable
       …
     Déclaration de la nième variable
< /SUB  VARIABLE >
```

3. Déclarations des variables

Le bloc déclaration comporte des variables et des constantes.

3.1 Déclaration des variables de type simple

La déclaration d'une variable a la forme suivante :

```
< nom_variable AS TYPE /> ;
< Liste_variable  AS TYPE /> ;
```

Note : aucun ordre n'est défini entre les déclarations :
On déclarer une variable en suite un (ou plusieurs) instruction (s) permettant de déclarer plusieurs variables ou l'inverse.

3.2 La déclaration de plusieurs variable par une seul instruction

Il s'agit d'un ensemble d'identificateurs séparés par une barre.

```
<nom_variable | nom_variable2 AS TYPE />;
```

3.3 Déclaration des tableaux

La déclaration d'un tableau a la forme suivante :

```
<ARRAY AS TYPE >
    <nom_tableau1: taille1/>
    <nom_tableau2: taille2/>
    ……
</ARRAY>
```

3.4 Les types des variables

Les types des variables doivent être : INT, CHR,STR,FLT, BOL.

- **INT :** Une constante entière est une suite de chiffres. Elle peut être signée ou non signée tel que sa valeur est entre **-32768** et **32767**. Si la constante entière est signée, elle doit être mise entre parenthèses.
- **FLT :** Une constante réelle est une suite de chiffres contenant le point décimal. Elle peut être signée ou non signée. Sa taille maximale est de 11 caractères (le point décimale,7 caractères au maximum pour la partie entière et 3 caractères aux maximum pour la partie décimale). Si la constante réelle est signée, elle doit être mise entre parenthèses.
- **BOL :** Une constante booléenne peut avoir les deux valeurs : **TRUE** et **FALSE**.
- **CHR :** Une variable de type CHR représente un caractère.
- **STR :** Une variable de type STR représente une chaîne de caractères.

3.5 Déclaration des Constantes :

La déclaration d'une constante se fait comme suit :

```
<SUB CONSTANTE>
   <nom_constante1 = valeur/>;
   <nom_constante1 = valeur/>;
</SUB CONSTANTE>
```

ou bien :

```
<SUB CONSTANTE>
   <nom_constante1  AS TYPE /> ;
   <Liste_constante2  AS TYPE/> ;
</SUB CONSTANTE>
```

3.6 Opérateurs arithmétiques, logiques et de comparaison

a) **Opérateurs arithmétiques :** +, - , * , /

b) **Opérateurs logiques**

- **AND(expression1, expression2,….) :** le **'ET'** logique.
- **OR(expression1, expression2,….) :** le **'OU'** logique.
- **NOT(expression) :** la négation.

c) Opérateurs de comparaison

- **SUP(expression1, expression2) :** expression1 > expression2.
- **INF(expression1, expression2) :** expression 1< expression2.
- **SUPE(expression1, expression2) :** expression1 ≥ expression2.
- **INFE(expression1, expression2) :** expression1 ≤ expression2.
- **EGA(expression1, expression2) :** expression1 = expression2.
- **DIF(expression1, expression2) :** expression1 ≠ expression2.

d) **Les conditions :** Une condition est une expression qui renvoie une valeur booléenne. Elle peut prendre la forme d'une expression logique, de comparaison ou une valeur booléenne

Affectation	
Description	Exemple
<AFF: idf, expression />	**<AFF:** a, (x+7+b)/ (5,3-(-2))**/>** **Equivalent à:** a=(x+7+b)/ (5,3-(-2)) **<AFF:** a, NOT(AND(TRUE,a))**/>** **Equivalent à:** !((TRUE & a)) **<AFF:** a, 'c'**/>** **<AFF:** a, "hello" **/>** **<AFF: T1[3]**, (x+7+T2[1])/ (5,3-(-2))**/>** **Equivalent à:** T1[3]=(x+7+T2[1])/ (5,3-(-2))

Entrées / Sorties	
Description	Exemple
Entrée : **<INPUT: idf** "…. **sign_formatage….**"**/>** Sortie : **<OUTPUT :**"…"+nom_varaible+"…signe_formatage "**/>**	**<INPUT : a** "donner la valeur de : **$**"**>** **<OUTPUT:** "a est $:"+a+ "b est: $"+b **/>**

Les signes de formatages sont :

Type	Signe de formatage
INT	$
FLT	%
STR	#
CHR	&
BOL	@

Condition IF (Si … Alors … Sinon … Fin Si**)**

Description	Exemple
<IF: condition> **<THEN>** instruction 1 instruction2 **</THEN>** **<ELSE>** instruction 3 instruction4 **</ELSE>** **</IF>** **Notes :** • Le bloc «THEN »est exécuté ssi la condition est vérifiée. Sinon le bloc « ELSE » sera exécuté s'il existe. • On peut avoir des conditions imbriquées. • Le bloc ELSE est facultatif	**<IF: AND(EGA(a.3),SUP(7,b)) >** **<THEN>** **<OUTPUT:** "a = 3 et 7> b " **/>** **</THEN>** **<ELSE>** **<OUTPUT:** "condition non vérifiée" **/>** **</ELSE>** **</IF>**

Boucle (Tant que … Fin Tant que**)**

Description	Exemple
<DO> instruction 1 instruction2 **<WHILE : condition/>** **</DO>** **Notes :** • le bloc d'instructions est exécuté ssi la condition est vérifiée. On peut avoir des boucles imbriquées.	**<DO>** **<OUTPUT:** "condition vérifiée" **/>** **<AFF:** b, ADD(b,1) **/>** **<WHILE : AND(EGA(a.3),SUP(7,b)) />** **</DO>**

Boucle (Tant que … Fin Tant que**)**

Description	Exemple
<FOR initialisation UNTIL IDF **ou bien** Constante **>** instruction 1 instruction2 **< /FOR>** **Notes :** • le bloc d'instructions est exécuté ssi la condition est vérifiée. • On peut avoir des boucles imbriquées.	**<FOR i=1 Until 14>** **<OUTPUT:** "condition vérifiée" **/>** **<AFF:** b, ADD(b,1) **/>** **</FOR>**

Remarques:

1- On ne peut lire qu'une seule variable à la fois.

2- L'instruction « INPUT » ne peut contenir que le signe de formatage de la variable à lire.

3- Une chaine de caractères est une suite de caractères entre deux guillemets.

4- Un idf de type CHAR est un seul caractère situé entre deux apostrophes.

3.7 Associativité et priorité des opérateurs :

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

Opérateur		Associativité
Opérateurs Logiques	OR (ou)	Gauche
	AND (et)	Gauche
Opérateurs de comparaison	SUP (>) SUPE (>=) EGA (==) DIF (!=) INFE (<=) INF (<)	Gauche
Opérateurs Arithmétiques	+ -	Gauche
	* /	Gauche

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. Pour cela, il est demandé de définir les différentes entités lexicales à l'aide d'expressions régulières et de générer le programme FLEX correspondant.

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 au-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 priorités pour les opérateurs afin de résoudre les conflits. Les routines sémantiques doivent être associées aux règles dans le fichier BISON.

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. Cette table 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 **recherche** et d'**insérer** des éléments dans la table des symboles. Les variables structurées de type tableau doivent aussi figurer dans la table de symboles.

4. Génération du code intermédiaire

Le code intermédiaire doit être généré-sous forme des quadruplets.

5. 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 :

File "Test", line 4, character 56: syntax error

6. Optimisation

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

• **Propagation de copie** (e.g. remplacer t1=t2 ; t3=4*t1 par t1=t2 ; t3=4*t2).

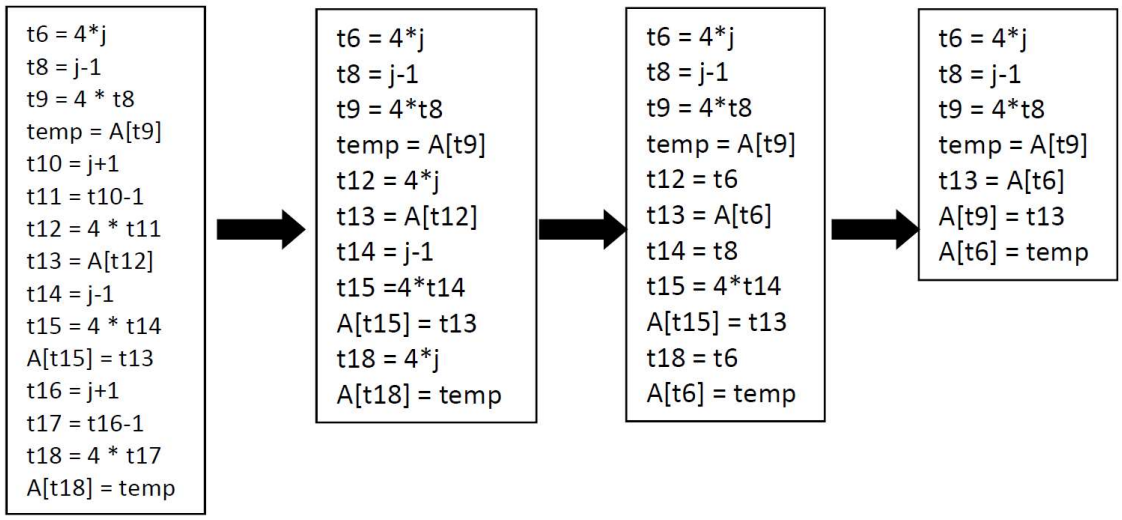
• **Propagation d'expression** (e.g. remplacer t1=expr; t3=4*t1 par t1=expr; t3=4*expr).

• **Élimination d'expressions redondantes** (communes) (e.g. remplacer t6=4*j ; t12=4*j par t6=4*j ; t12=t6).

• **Simplification algébrique** (e.g. remplacer t1+1-1 par t1).

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

Exemple :



7. Génération du code machine

Enfin le code objet doit être généré selon la syntaxe de **l'assembleur 8086**.