Hassler Florian 16.06.2018

Lizzi Dimitri

Technique de compilation

Compilateur Hepial

Table des matières

Introduction

Le but de ce travail pratique est de créer un compilateur pour le le langage de programmation fictif hepial.
Le lexique du langage est défini avec l'analyseur lexical JFlex. Les règles de grammaires sont ensuite définies avec l'analyseur syntaxique CUP. Enfin du code assembleur Jasmin est généré pour exécuter les instructions du fichier hepial.

Analyse lexicale

Analyseur JFlex

L'analyse lexical se fait avec JFlex. Son but est de définir le lexique du langage. En d'autres termes, il permet de définir les "mots" ou symboles terminaux utilisés par le langage à l'aide d'expression régulière. Lors de l'analyse du fichier, il va repérer si les mots utilisés dans le fichier hepial correspondent à ceux défini dans son lexique.
Cette analyse lexical va permettre à l'analyseur syntaxique de pouvoir reconnaître chaque symbole reconnu par le langage hepial. Ainsi, tout mot non reconnu par le lexique produira une erreure de syntaxe.

Lexique

Le lexique défini tous les mots clés et symboles reconnus par le langage hepial.

symbole	expression régulière	description
PRG	"programme"	Entête du programme
STARTPRG	"debutprg"	début de programme
ENDPRG	"finprg"	fin de programme
STARTFUNC	"debutfonc"	début de fonction
ENDFUNC	"finfonc"	fin de fonction
CONSTANT	"constante"	constante
SEMICOLON	"\;"	fin d'instruction
OPENPARENT	"\("	parenthèse ouvrante
CLOSEPARENT	"\)"	parenthèse fermante
EQUAL	"\="	affectation
COMMA	"\,"	séparateur d'identifiant de variable
DOUBLEPOINTS)	".."	range d'un tableau
OPENBRACK	"\["	crochet ouvrant
CLOSEBRACK	"\]"	crochet fermant
PLUS	"\+"	addition
MINUS	"\-"	soustraction
TILDA	"\~"	non bit à bit
TIMES	"*"	multiplication
NOT	"non"	non logique
AND	"et"	et logique
OR	"ou"	ou logique
DIVIDE	"\/"	division
EQUALS	"=="	égal à
DIFF	"<>"	différent de

INF	"<"	inférieur à
SUP	">"	supérieur à
INFEQUAL	"<="	inférieur ou égal
SUPEQUAL	">="	supérieur ou égal
TRUESYM	"vrai"	valeur booléenne vrai
FALSESYM	"faux"	valeur booléenne faux
WHILESYM	"tantque"	début boucle while
DOSYM	"faire"	symbole do
ENDWHILE	"fintantque"	fin de boucle while
IFSYM	"si"	symbole if
THEN	"alors"	symbole then
ELSESYM	"sinon"	symbole else
ENDIF	"finsi"	symbole de fin if
FORSYM	"pour"	début de boucle for
FROM	"allantde"	limite inférieure de boucle for
TO	"a"	limite supérieure de boucle for
ENDFOR	"finpour"	fin de boucle for
READ	"lire"	instruction de lecture
WRITE	"ecrire"	instruction écriture
RETURNSYM	"retourne"	symbole return
TINTEGER	"entier"	type entier (signé)
TBOOLEAN	"booleen"	type booléen
INTEGERCONST	[0-9]+	nombre entier: un ou plusieurs chiffres
STRINGCONST	[\"]([^\"]|\"\")*[\"]	chaine de caractère: n'importe quel caractère ou rien entre deux guillemets
IDENT	[a-zA-Z][a-zA-Z0-9]*	identifiant de variable: doit commencer par une lettre et peut d'autres lettres ou chiffres
COMMENT	\/\/([^\n]|[^\r\n])*	commentaire: deux slash puis n'importe quel caractère ou vide

Analyse Syntaxique

Analyseur CUP

L'analyse syntaxique se fait avec CUP dont le but est de définir les symboles non terminaux et les règles de grammaire du langage afin que celui-ci aie du sens. Il permet donc de repérer si les mots utilisés dans le fichier à analyser sont utilisés dans un ordre cohérent.
Lorsque CUP reconnaît qu'une instruction hepial a été écrite en utilisant les bons mots avec la bonne grammaire, on peut exécuter du code JAVA qui va permettre de gérer cette instruction afin qu'elle produise le résultat attendu.

Table des symboles

La tables des symboles permet de gérer toutes les variables ou constantes créées dans un programme hepial.
La classe de la table des symbole est un singleton. Cela permet de s'assurer qu'une seule instance de cette classe sera utilisée.
La classe est composée principalement d'une hashtable qui permet d'enregistrer les variables ou constantes créées dans le programme hepial.
Ainsi, lorsque le fichier est parcouru et qu'une variable est déclarée, l'analyseur syntaxique va ajouter cette nouvelle variable à la table des symbole. Chaque entrée de la table des symbole possède un identifiant et un type.
Les tableaux n'ont malheureusement pas pu être implémentés par manque de temps.

Type

Les variables ou constantes peuvent avoir deux types différents:

- Entier (signé)
- Booléen

Un type supplémentaire String (chaîne de caractère) a également été créé afin de pouvoir gérer les chaînes de caractères qui peuvent être affichée par la commande write.

Arbre abstrait

L'arbre abstrait permet de gérer toutes les instructions et expressions qui sont reconnues par le langage hepial. L'arbre abstrait est construit comme suit:

- AbtractTree
    - Instruction
        - ReturnInstr (instruction de retour de fonction)
        - ReadInstr (instruction de lecture)
        - WriteInstr (instruction d'écriture)
        - WhileLoop (boucle while)
        - ForLoop (boucle for)
        - Condition (if statement)
        - Call (appel de fonction)
        - Affectation (affectation de variable ou constante)
        - Block (liste d'instruction)
    - Expression
        - Unary (opérations unaires)
            - Not (non logique)
            - Tilda (non bit à bit)
        - Binary (opérations binaires)
            - Arithmetic (opérations arithmétiques)
                - Division (division entière)
                - Addition (addition entière
                - Substraction (soustraction entière)
                - Multiplication (multiplication entière)
            - Comparison (comparaisons de deux valeurs)
                - Superior (supérieur à)
                - SupEqual (supérieur ou égal à)
                - Equal (égale à)
                - NotEqual (différent de)
                - Inferior (inférieur à)
                - InfEqual (inférieur ou égal à)
            - And (et logique)
            - Or (ou logique)
        - BooleanValue (valeur booléenne)
        - IntNumber (nombre entier signé)
        - Idf (identifiant de variable ou constante)
        - StringValue (chaine de caractère)

Chaque branche ou feuille de cet arbre est une classe qui gère son instruction ou expression. La programmation objet permet de créer réellement cet arbre en héritant les bonnes classes entre elles.
Au tout début de l'analyse syntaxique, on crée une pile d'arbre abstrait. Cette pile va donc pouvoir contenir n'importe quelle instruction ou expression car toutes sont héritée d'arbre abstrait.
Ainsi lorsqu'une instruction ou expression est reconnue avec succès par l'analyseur syntaxique, elle est ajoutée à la pile.

Priorité des opérations

Le langage hepial permet d'effectuer des opérations arithmétique ou logique. L'ordre des opération doit être scrupuleusement respecté et par conséquent chaque opérateur a une priorité. De manière générales les opérations s'effectuent de gauche à droite. Les opérateur d'égalité (==), différence (<>), supériorité (>), supérieur ou égal (>=), infériorité (<), inférieur ou égal (<=) ont la priorité la plus haute. Viennent ensuite les opérateurs de multiplication (*), division (/) et "et logique" (et). Enfin les opérateurs d'addition (+), soustraction (-) et "ou logique" (ou) ont la priorité la plus faible.

Analyse sémantique

L'analyse sémantique de chaque instruction ou expression est faite dans la classe *SemanticAnalyser.*
Le design pattern *visitor* est utilisé pour en faciliter son implémentation. Chaque classe de l'arbre abstrait implémente une fonction *accept* qui reçoit en paramètre un visiteur et qui se retourne elle même.

```
@Override
public Object accept(Visitor visitor) {
   return visitor.visit(this);
}
```

Cela permet à la classe *SemanticAnalyser* d'obtenir les informations sur l'expression ou l'instruction voulue et de faire les test afin de vérifier que toutes les conditions sont réunies au bon fonctionnement de l'instruction. Ce sont principalement des tests sur la concordance des types qui sont effectués. L'analyseur sémantique vérifie également les éventuelles affectations sur une constante ou les division ne sont par zéro.

Génération du code assembleur

Lorsque l'analyse lexicale et syntaxique est concluante, le compilateur va générer un code Jasmin, un langage assembleur pour la machine virtuelle JAVA. Le langage Jasmin permet de décrire des classes dans un langage simple, permettant d'utiliser les instructions du set d'instruction JVM. Les fichiers *.j* contenant le code jasmin sont ensuite compilés en *bytecode* binaire dans un fichier *.class,* comme une classe java compilée standard.

La classe *JasminGenerator* se charge d'écrire un code Jasmin en parcourant l'arbre abstrait qui a préalablement été construit avec Cup et validé sémantiquement par l'analyseur sémantique. Tout comme ce dernier, elle utilise le patron de conception *Visiteur* pour parcourir les différents noeuds de l'arbre abstrait.

Le point d'entrée du générateur est la méthode `public String generate(Block mainBlock)`. Elle va instancier des champs utilitaires qui seront utilisé pendant le parcours de l'arbre abstrait:

Type	Nom	Description
StringBuilder	jasminStringBuilder	Le code est écrit petit à petit à travers cette instance de stringBuilder. A la fin de la génération, un string contenant le code complet est généré en appelant sa méthode toString.
HashMap<String, Integer>	localsIndices	Permet de se souvenir de l'index d'une variable dans la table des variables locales de la fonction main.
int	nextLocalIndex	Compteur permettant de savoir quel est le prochain index de variable locale à utiliser à la prochaine déclaration de variable. Ce compteur est incrémenté à chaque fois qu'une nouvelle variable est stockée dans le table des variables locales.
int	nextIfNumber	Compteur permettant d'obtenir un numéro unique utilisé pour différencier les labels utilisés lors d'un embranchement (condition ou boucle). Il est utilisé comme suffixe aux labels pour éviter que deux labels du même nom se retrouvent dans le code généré. Ce numéro est incrémenté à chaque embranchement.

Le début du code jasmin est ensuite généré. Celui-ci définit une classe *HepialProgram* héritant de Object, avec un constructeur par défaut qui appelle celui de la classe Object.

Une méthode main est ensuite générée. Celle ci est le point d'entrée de l'application. Une fois dans le corps de cette méthodes, la génération du code à partir de l'arbre abstrait commence, en visitant le bloc principal de l'application, se trouvant au sommet de l'arbre.

Visite d'un bloc

Lors de la visite d'un bloc, on visite tour à tour chacune des instructions dont il est composé.

Visite d'une constante

Lorsqu'une constante est visitée, sa valeur est ajoutée au sommet de la pile avec l'instruction *ldc.*

Visite d'un identifieur

Lors de la visite d'un identifieur, on vérifie si on lui a déjà donné un index dans la table des variables locales. Si oui, on retourne son index. Sinon, on lui assigne un nouvel index et on le retourne.

Visite d'une affectation

On commence par visiter la destination de l'affectation, qui est forcément un identifieur, pour récupérer son index dans la table des variables locales. On visite ensuite l'expression source. S'il s'agit d'un identifieur, on reçoit son index dans la table des variables locales et on le charge sur la pile à l'aide de l'instruction *iload*. S'il s'agit d'autre chose qu'un identifieur, on sait qu'une valeur aura été ajoutée dans la pile lors de la visite.

On affecte ensuite la valeur au sommet de la pile à l'index de l'identifieur source dans la table des symboles en utilisant l'instruction *istore*.

Opération

Lors de la visite d'une opération mathématique (addition, soustraction, multiplication, division), on visite l'opérande de gauche puis celle de droite. S'il s'agit d'un identifiant, sa valeur est chargée sur la pile. On appelle ensuite l'instruction relative à l'opérateur, qui dépile les deux valeurs des opérandes et empile le résultat.

Les opérations logiques sont effectuées de la même manière, car un booléen est en fait stocké dans une variable entière, en utilisant la valeur 0 pour faux et 1 pour vrai.

Condition

On visite d'abord l'expression de la condition pour que sa valeur soit empilée. Si cette valeur est 0, elle est considérée comme fausse, si elle est différente, elle est considérée comme vraie. L'instruction *ifeq* va sauter vers un label délimitant le bloc d'instruction *sinon* si la valeur au sommet de la pile est 0, ou alors exécuter les instructions qui suivent si la valeur est différente de 0. Le pseudo-code ci-dessous illustre ce comportement:

```
ifeq label_sinon
    instructions si vrai
    goto label_fin
label_sinon:
    instructions si faux
label_fin:
```

Boucle while

Une boucle while utilise à peu près la même logique qu'une condition, sauf que les labels sont positionnés différemment:

```
label_debut:
ifeq label_fin:
    instructions à répéter à chaque itération
goto label_debut
label_fin
```

Boucle for

Une boucle for définit une variable initialisée à une valeur de départ. Au début de chaque itération de la boucle, on teste si cette valeur est plus grande que la valeur de fin. Si c'est le cas, on saute à un label de fin de boucle. Sinon, on exécute le bloc d'instructions de l'itération, on incrémente la variable et on saute avant la condition pour l'exécuter à nouveau.

Comparaisons

Pour chaque type de comparaison, on charge une valeur dans la pile étant 0 ou 1 selon le résultat de la comparaison. On peut ensuite utiliser cette valeur dans une condition ou une boucle.

Instructions d'écriture

Pour écrire sur la sortie standard, on appelle la méthode println de l'instance statique java/lang/system/out de la classe java/io/Printstream. Selon le type de la valeur à afficher, on appellera la méthode avec le type correspondant. Par simplicité, les booléens sont affichés avec des valeurs numériques, 0 ou 1.

Instructions de lecture

La lecture s'effectue à l'aide de la classe java/util/Scanner utilisant le flux d'entrée java/lang/system/in. On instancie un nouveau scanner et on appelle sa méthode nextInt pour récupérer une valeur entière. On stocke ensuite cette valeur dans la table des variables locales, à un index correspondant à l'identifiant donné.

Limitations du code généré

Absence de fonctions

Les fonctions n'ont pas été implémentées dans ce projet par manque de temps. Pour les ajouter, il aurait fallu changer toute la mécanique des variables, pour les stocker dans des champ de la classe générée plutôt que de les déclarer comme des champs de la classe main. Il aurait aussi fallu pouvoir distinguer les symboles définis dans main et ceux définis dans les fonctions.

Taille de la pile

La taille de la pile da la méthode main est fixée à 100 éléments. Cette valeur est largement suffisante pour exécuter les codes sources d'exemple. Cependant, un code plus considérable pourrait arriver au bout de la pile et ne plus s'exécuter. Une solution à ce problème serait de compter à l'avance la taille nécessaire et d'allouer juste la taille nécessaire.

Nombre de variables

La table des variables locales de la fonction main est fixée à 100 variable. Ce problème aurait pu être être résolu en introduisant un compteur de variables. Cette valeur est cependant largement suffisante pour exécuter les codes sources de test que nous avons essayé.

Conclusion

Le compilateur du langage hepial est fonctionnel. L'analyseur lexical reconnais correctement les mots clés et symboles du langage hepial. L'analyseur syntaxique reconnais correctement toutes les instructions et expressions du langage. Le code assembleur Jasmin généré permet d'exécuter correctement les programmes hepial syntaxiquement corrects. Seuls les tableaux et les fonctions n'ont pas pu être correctement implémentés par manque de temps.

Exemple de programme

Pour chaque exemple de programme ci-dessous, il y a:

1. le programme en langage hepial
2. Le code Jasmin généré si les analyses lexicales, syntaxiques et sémantiques se déroulent sans erreur. Sinon, l'erreur retournée par un des analyseurs
3. L'éventuel output généré par le code Jasmin

Hello World

Programme affichant "hello world"

Code hepial

```
programme helloWorld
debutprg
  ecrire "hello world";
finprg
```

Code Jasmin

```
.class public HepialProgram
.super java/lang/Object
.method public <init>()V
aload_0
invokespecial java/lang/Object/<init>()V
return
.end method
.method public static main([Ljava/lang/String;)V
.limit stack 100
.limit locals 100
getstatic java/lang/System/out Ljava/io/PrintStream;
ldc "hello world"
invokevirtual java/io/PrintStream/println(Ljava/lang/String;)V
return
.end method
```

Output

```
hello world
```

CelsiusFarenheit

Programme de conversion de températures Celcius en Farenheit

Code hepial

```
programme celsiusfarenheit

entier c;

debutprg
   ecrire "Entrez une valeur en celcius:";
  lire c;
  ecrire "La valeur entrée est: ";
  ecrire c;
  ecrire "La valeur en farenheit est:";
  ecrire  c * 9 / 5  + 32;
finprg
```

Code Jasmin

```
.class public HepialProgram
.super java/lang/Object
.method public <init>()V
aload_0
invokespecial java/lang/Object/<init>()V
return
.end method
.method public static main([Ljava/lang/String;)V
.limit stack 100
.limit locals 100
getstatic java/lang/System/out Ljava/io/PrintStream;
ldc "Entrez une valeur en celcius:"
invokevirtual java/io/PrintStream/println(Ljava/lang/String;)V
new java/util/Scanner
dup
getstatic java/lang/System/in Ljava/io/InputStream;
invokespecial java/util/Scanner/<init>(Ljava/io/InputStream;)V
invokevirtual java/util/Scanner/nextInt()I
istore 0
getstatic java/lang/System/out Ljava/io/PrintStream;
ldc "La valeur entrée est: "
invokevirtual java/io/PrintStream/println(Ljava/lang/String;)V
getstatic java/lang/System/out Ljava/io/PrintStream;
iload 0
invokevirtual java/io/PrintStream/println(I)V
getstatic java/lang/System/out Ljava/io/PrintStream;
ldc "La valeur en farenheit est:"
invokevirtual java/io/PrintStream/println(Ljava/lang/String;)V
```

```
getstatic java/lang/System/out Ljava/io/PrintStream;
iload 0
ldc 9
imul
ldc 5
idiv
ldc 32
iadd
invokevirtual java/io/PrintStream/println(I)V
return
.end method
```

Output

```
Entrez une valeur en celcius:
0
La valeur entrée est:
0
La valeur en farenheit est:
32
```

Boucles et conditions

Programme utilisant des boucles et des conditions

Code hepial

```
programme hepial4
entier x,y,z,p,d,i;
debutprg
  x = 1;
  y = 5;
  z = x + y;
  x = 5 - z;
  si x == 5 alors
     x = x+3;
     p = x*z;
  sinon
     x = x+5;
     y = 5;
  finsi
  d = 4/2;
  pour i allantde 1 a 10 faire
     x = x + i;
  finpour
  tantque z <= 5 faire
     z=z+1;
  fintantque
  ecrire "valeur de x :";
  ecrire x;
  ecrire "valeur de y :";
  ecrire y;
  ecrire "merci tchao !";
finprg
```

Code Jasmin

```
.class public HepialProgram
.super java/lang/Object
.method public <init>()V
aload_0
invokespecial java/lang/Object/<init>()V
return
.end method
.method public static main([Ljava/lang/String;)V
.limit stack 100
.limit locals 100
ldc 1
istore 0
ldc 5
```

```
istore 1
iload 0
iload 1
iadd
istore 2
ldc 5
iload 2
isub
istore 0
iload 0
ldc 5
if_icmpeq then_0
ldc 0
goto endif_0
then_0:
ldc 1
endif_0:
ifeq else_1
iload 0
ldc 3
iadd
istore 0
iload 0
iload 2
imul
istore 3
goto endif_1
else_1:
iload 0
ldc 5
iadd
istore 0
ldc 5
istore 1
endif_1:
ldc 4
ldc 2
idiv
istore 4
ldc 1
istore 5
for_2:
iload 5
ldc 10
if_icmpgt endfor_2
iload 0
iload 5
iadd
istore 0
iinc 5 1
```

```
goto for_2
endfor_2:
while_3:
iload 2
ldc 5
if_icmple then_4
ldc 0
goto endif_4
then_4:
ldc 1
endif_4:
ifeq endwhile_3
iload 2
ldc 1
iadd
istore 2
goto while_3
endwhile_3:
getstatic java/lang/System/out Ljava/io/PrintStream;
ldc "valeur de x :"
invokevirtual java/io/PrintStream/println(Ljava/lang/String;)V
getstatic java/lang/System/out Ljava/io/PrintStream;
iload 0
invokevirtual java/io/PrintStream/println(I)V
getstatic java/lang/System/out Ljava/io/PrintStream;
ldc "valeur de y :"
invokevirtual java/io/PrintStream/println(Ljava/lang/String;)V
getstatic java/lang/System/out Ljava/io/PrintStream;
iload 1
invokevirtual java/io/PrintStream/println(I)V
getstatic java/lang/System/out Ljava/io/PrintStream;
ldc "merci tchao !"
invokevirtual java/io/PrintStream/println(Ljava/lang/String;)V
return
.end method
```

Output

```
valeur de x :
59
valeur de y :
5
merci tchao !
```

Erreur sémantique

Code produisant une erreur sémantique due à l'affection d'une constante et en plus d'affectation d'une valeure integer dans une variable booléenne

Code hepial

```
programme hepial1

entier x;
constante booleen y = vrai;

debutprg
  y = 10 + 2;
finprg
```

Output

```
Semantic errors:
Affection error: destination is a constant.
Type error: source type of affectation is not conform to destination type.
```