CS Games 2016

Compétition de Compilation

Nombre de participants	**2**
Nombre de postes	**1**
Valeur totale	**7%**
Durée Totale	**3 heures**

MPIRE

--- COMMUNICATION DU DÔME ---

Chers participants,
Le Dôme est une merveille technologique, mais il n'est pas encore parfait. Pour le moment, toute son infrastructure est gérée à l'aide d'un langage de l'ancienne ère, dont nous n'avons pas pu récupérer les sources. Notre évolutivité est désormais limitée par rapport au contrôle que nous avons sur ce langage. Cette épreuve a pour but d'évaluer votre capacité à développer le langage de la Nouvelle Ère.

--- FIN DE COMMUNICATION ---

Énoncé

Votre tâche consiste à développer un interpréteur de MPIRE (eMpirical Procedural Idiom for Robust Execution).

Un *interpréteur* est un programme qui interprète le code source d'un programme ligne après ligne. Contrairement à un compilateur, il ne produit pas de code exécutable sauvegardé dans un fichier. Le programme est exécuté par une machine virtuelle qui peut produire du code machine à la volée via un système de JIT pour de meilleures performances.

Un *parser* est analyseur syntaxique; il produit un arbre syntaxique à partir d'un ensemble de jetons lexicaux

Un *lexer* est un analyseur lexical; il produit des jetons lexicaux à partir de texte brut

Vous pouvez générer un ensemble de *parser* et *lexer* avec SableCC (version 4b2). Des exemples de *grammaire* et de *visiteur* sont disponibles.

Une *grammaire* est une définition structurée des jetons syntaxiques qui composent un langage. Celle-ci utilise la récursion et les expressions régulières afin de définir les chaînes de caractères valides.

Un *visiteur* est une construction logicielle permettant de visiter un arbre syntaxique à des fins d'analyse ou d'interprétation d'un programme ou d'une partie d'un programme.

Les spécifications du langage **MPIRE** sont décrites dans la suite de ce document.

Description Technique

L'interpréteur de **MPIRE** doit être codé en Java. Vous devez produire un fichier `Interpret.class` dans le dossier `bin`.

Exemple de hiérarchie de fichiers

```
./solution/
└──── bin
      └──── Interpret.class
```

Le non-respect de cette consigne entraîne une pénalité de 5%.

Le fichier `tests.sh` permet de tester rapidement votre interpréteur en exécutant un ensemble de programmes d'essais.

Correction

Fonctionnalité 90%

Correspond au nombre de tests, privés et public, que votre interpréteur est capable de passer.

Qualité de la grammaire 5%

Qualité du code 5%

- ❏ Nom explicite des classes, fonctions et variables
- ❏ Lisibilité du code
- ❏ Métrique WTFPM (What-The-Fuck-Per-Minute)

Spécification du Langage MPIRE

Syntaxe

Les fichiers de programmes sont encodés en ASCII sur 7 bits.

Mots-clés

- `print`
- `byte`
- `println`
- `while`
- `if`
- `else`
- `read`
- `not`

Tous les mots-clés sont sensibles à la casse et réservés, c'est à dire qu'ils ne peuvent être utilisés en tant qu'identifiant (nom de variable).

Identifiants

Sont valides toute séquence de lettres majuscules et minuscules, ainsi que le souligné (`_`). Les chiffres ne sont pas des caractères valides dans un identifiant.

`(a-z, A-Z,_)`

Les séquences de chiffres sont considérées comme des nombres littéraux.

Particularités Syntaxiques

Le caractère = sert autant à l'assignation qu'à la comparaison.

Les comparaisons peuvent être enchaînées et se propagent de gauche à droite. Par exemple, l'expression

```
a = b > 2 <= 3
```

signifie

```
a = b && b > 2 && 2 <= 3
```

Aucun délimiteur de portée n'est nécessaire (soit les construction de la forme `{...}` ainsi que l'indentation).

Programmes et Instructions

Un programme **MPIRE** est constitué d'une séquence *d'instructions*.

Assignations

Syntaxe :

```
<identifiant> = <valeur>
```

Exemples :

```
a = 5
foo = a + 5
```

Affichage

Syntaxe :

```
print <chaine>
print [byte] <valeur>
println
```

NOTE : l'instruction `println` ne prend pas d'arguments. Ce comportement est volontaire
NOTE : print peut avoir deux comportements lorsque l'on lui passe une valeur: si byte est présent, il affichera le caractère correspondant à la valeur passée en paramètre, sinon il affichera l'entier correspondant à la valeur passée en paramètre.

Exemple :

```
print "hello"
print 42
print byte 42
println
```

Sortie :

```
hello42*
```

Séquences

Les séquences d'instructions sont exécutées en ordre. Les espaces n'ont aucune importance sur le flux d'exécution du programme.

Exemple :

```
a = 1 print "a=" print a println
a=1print"a="print a println
```

Plusieurs instructions peuvent être groupées avec des parenthèses

```
(a = 1 print "a=" print a println)
```

Sortie :

```
a=1
```

Boucle Tant Que

Syntaxe :

```
while <condition> <instruction>
```

Exemple :

```
a = 5
while a>0 (print "*" a = a - 1)
print byte 10 # line feed
```

Sortie :

```
*****
```

Si

Syntaxe :

```
if <condition> <instruction> [else <instruction>]
```

Exemple :

```
a = 5
if a > 0 print "ok"
if a < 0 print "fail"
if a = 5 print "ok" else print "fail"
```

Sortie :

```
okok
```

Valeurs et Données

Les valeurs sont toutes représentées par des entiers signés avec au moins 16 bits de précision.

Littéraux

Les nombre sont représentés par leur valeur décimale, et les chaînes de caractères par une série de caractères ASCII sur 7 bits (avec le code ASCII).
Une chaîne de caractères est délimitée par des caractères " (double quote).
Un caractère unique est lui précédé du jeton ' (simple quote).

```
print byte 42
if 42 = '* print "="
print '*
```

Sortie

```
*=42
```

Variables

La portée des variables est globale. Le comportement lors de l'utilisation d'une variable non-définie est indéterminé.

Opérations arithmétiques de base

- Opérateurs binaires infixes
  `+ - * /`
- Opérateur de négation unaire
  `-`
- Parenthèses pour le regroupement d'instructions ainsi que la priorité des opérations
  `()`
- Priorité des opérations habituelle

Lecture

Syntaxe :

```
read
read byte
```

L'instruction `read` lit une ligne de l'entrée standard et l'interprète comme un entier, tel que défini en **MPIRE**.

L'instruction `read byte` lit le prochain octet de l'entrée standard et l'interprète comme un entier.

Conditions

Comparaisons

Syntaxe :

```
<valeur> [<op> <valeur>]+
```

Où *op* est l'un des symboles de comparaison suivant :

- `=`
- `!=`
- `<`
- `<=`
- `>`
- `>=`

Exemple :

```
a = 1
b = 2
if a = 1 != b < 4 print "ok"
if a = 1 != b = 4 print "fail"
```

Sortie :

```
ok
```

Opérations Booléennes

Il existe trois opérateurs booléens :

- `not`
- `&&`
- `||`

Le “ou” logique étant le moins prioritaire et la l'opérateur de négation “not” le plus prioritaire. Les parenthèses peuvent être utilisées pour le groupement ainsi que la priorité.

Exemple:

```
a = 5
if a > 2 && a < 7 print "ok"
if not (a < 2 || a > 7) print "ok"
print byte 10
```

Sortie :

```
okok
```