CS Games 2016

Compilation Compétition

Participants	**2**
Workstations	**1**
Value	**7%**
Duration	**3 hours**

MPIRE

... BEGIN TRANSMISSION ...
Greetings, humans. The Dome is a technological prowess, however it is still not perfect. For the moment, all its infrastructure is managed via a pre-alpha era language, the sources of this language could not be recovered. Therefore, we are quite limited when it comes to controlling the language and its evolutions. The goal of this challenge is for you to develop a new language. We could then use the knowledge of the best amongst you to make one which can control the Dome in its entirety.
... END TRANSMISSION ...

Problem Description

Your objective is to develop an interpreter for the MPIRE (eMpirical Procedural Idiom for Robust Execution) language.

An interpreter is a program which interprets source code from a program, line after line. Contrary to a compiler, it does not produce machine-readable code into an executable file. The program is read and is executed by a virtual machine which may produce machine code on-the-fly via a JIT system for performance.

A *parser* is a syntactic analyser; it produces a syntax tree from lexical tokens.

A *lexer* is a lexical analyser; it produces tokens from raw text sequences.

You can generate both *lexer* and *parser* using SableCC (version 4b2). Examples of *grammar* and *visitor* will be given to you.

A *grammar* is a structured definition of the syntactic tokens for a language. It uses regular expressions for the lexical part and defines syntactic structures recursively to determine the validity of a program.

A *visitor* is a software construction permitting the visit of a syntax tree to analyse or execute a program or part of a program.

Specifications of the **MPIRE** language will be discussed in this document.

Technical Description

The **MPIRE** interpreter that you must produce has to be developed using Java. You must produce a `Interpret.class` file in the `bin` folder for this challenge.

Example of file hierarchy

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

Any failure to comply will result in a 5% penalty.

The `tests.sh` file will allow proper testing of your solution on example files and inputs, assuming your solution respects the naming scheme.

Correction

Functionality 90%

How many tests is your interpreter capable of running adequately amongst the public and private test.

Grammar Quality 5%

Code Quality 10%

- ❏ Explicit naming scheme for classes, methods and variables
- ❏ Proper spacing between blocks
- ❏ WTFPM (What-The-Fuck-Per-Minute) metric

Language Specifications

Syntax

Programs are exclusively encoded using ASCII-7bits.

Keywords

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

Identifiers

Any sequence of letters and underscores (except those defined as keywords) is considered an identifier. Numeric digits are not a valid part of identifier.

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

Sequences of numerical digits are literal numbers.

Particularities

The character = is used as assignation operator as much as a comparison operator.

Comparisons can be chained and are propagated from left to right. For example, the expression:

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

means

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

There is no need for any delimitation symbols (constructions like `{...}` or indentation).

Programs and [instructions|statement]

An **MPIRE** program is a sequence of instructions.

Assignations

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

Example:

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

Printing

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

NOTE : *println* does not take any argument, this is purely voluntary.

NOTE : `print` can have two behaviours with the value passed as parameter: if `byte` is used, it will print the character associated with the value in ASCII, else, it will print the numerical value of the parameter.

Example:

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

Output:

```
hello42*
```

Sequences

Sequences of instructions are executed one after another. Spaces have no impact on the execution flow.

Examples:

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

Several instructions can be grouped into a block through parentheses.

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

Output:

```
a=1
```

While

Syntax:

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

Example:

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

Output:

```
*****
```

If

Syntax:

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

Example:

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

Output:

```
okok
```

Values

Values signed integers with at least 16 bits of precision.

Literals

Literal values can either be regular values or ASCII 7-bits characters.
A string is delimited by the use of " (double quote).
A single character is preceded by the character ' (simple quote).

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

Output:

```
*=42
```

Variables

Variable are necessarily global. Use of an undefined variable results in undefined behaviour.

Basic arithmetic operations

- Infix binary operators
  `+ - * /`
- Unary negation operator
  `-`
- Parenteses can be used for grouping and priority management
  `()`
- Operations priority is standard

Reading values

Syntax

```
read
read byte
```

`read` will read a line and transform it to a numeric value as defined in **MPIRE**.

`read byte` reads the next byte from stdin and interprets it as an integer.

Conditions

Comparison

Syntax

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

Where *op* is either one of the following comparison operators:

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

Examples:

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

Output:

```
ok
```

Boolean operations

There are three boolean operations permitted in **MPIRE**:

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

Priority order is, from most prioritary to least prioritary: not, &&, ll. Parentheses can be used for grouping and priority.

Examples:

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

Output:

```
okok
```