Setlan

Sencillo lenguaje para manejo de conjuntos. La extensión del lenguaje es `.stl`

Estructura de un programa

```
program <instrucción>
```

Ejemplo:

```
program
    print "hello world!"
```

Un ejemplo más complejo:

```
program {
    using
        int n;
    in
        scan n;
        print n * n;
}
```

Identificadores

Un identificador de variable es una cadena de caracteres de cualquier longitud compuesta únicamente de las letras desde la `A` hasta la `Z` (mayúsculas o minúsculas), los dígitos del `0` al `9`, y el caracter `_`.

Los identificadores no pueden comenzar por un dígito y son sensibles a mayúsculas: la variable `var` es diferente a la variable `Var`, que a su vez son distintas a la variable `VAR`. Para este proyecto no es necesario el reconocimiento de caracteres acentuados (e.g. *á*, *é*, *í*, *ó*, *ú*) ni la letra eñe (*ñ*).

Tipos de datos

Se dispone de tres tipos de datos en el lenguaje:

- `int`: representan números enteros con signo de `32 bits`.
- `bool`: representa un valor booleano, es decir, `true` o `false`.
- `set`: representa un conjunto de números enteros sin un orden específico. Dichos números van separados por una coma `,`, de la siguiente forma:
    - `{}` representa un conjunto vacío.
    - `{a,b,c}` representa el conjunto con los elementos `a`, `b` y `c` del tipo `int`, `a`, `b` y `c` deben ser distintos entre sí.
    - Ejemplo:
        - `{0,1,2}` es el conjunto de los números enteros `0`, `1` y `2`.
        - El conjunto `{0,1,2}` es equivalente al conjunto `{1,2,0,2}`; es decir, en *Setlan*, la expresión `{0,1,2} == {1,2,0,2}` debe generar el valor `true`.

Las palabras `int`, `bool` y `set` están reservadas por el lenguaje para la declaración de variables, al indicar su tipo.

Instrucciones

Asignación

```
<variable> = <expresión>
```

Ejecutar esta instrucción tiene el efecto de evaluar la expresión del lado derecho y almacenarla en la variable del lado izquierdo. La variable tiene que haber sido declarada previamente y su tipo debe coincidir con el tipo de la expresión, en caso contrario debe mostrarse un error en pantalla y abortar la ejecución.

Ejemplo válido:

```
a = 10
```

Bloque

```
{
    <declaración de variables>
    <instrucción 1>;
    <instrucción 2>;
    ...
    <instrucción n>;
}
```

El bloque es una instrucción que consiste de una sección de declaración de variables, la cual es opcional, y una secuencia de instrucciones finalizadas por `;`.

La sintaxis de la declaración de variables es:

```
using
    <tipo> x1, x2, ... , xn;
    <tipo> y1, y2, ... , yn;
    ...
    <tipo> z1, z2, ... , zn;
in
```

Estas variables sólo serán visibles a las instrucciones y expresiones del bloque. Se considera un error declarar más de una vez la misma variable en el mismo bloque.

Las variables declaradas tienen un valor por defecto dependiendo del tipo del que son:

- Las variables del tipo `int` tiene por defecto el valor cero (`0`).
- Las variables del tipo `bool` tiene por defecto el valor `false`.
- Las variables del tipo `set` tiene por defecto el valor del conjunto vació, `{}`.

Ejemplo válido:

```
program {
    using int age; in
    print "how old are you?";
    scan age;
    print "you said ", age, ", right?";
}
```

Entrada

```
scan <variable>
```

Permite obtener datos escritos por el usuario vía entrada estándar. Al ejecutar la instrucción el interpretador debe solicitar al usuario que introduzca un valor que debe ser comparado con el tipo de la `<variable>` destino.

Si el valor suministrado por el usuario es inválido se debe repetir el proceso de lectura.

Puede exisitir cualquier cantidad de espacios en blanco antes o después del valor introducido.

Esta instrucción funciona únicamente con variables (y valores) de tipo `bool` e `int`.

Salida

```
print x1, x2, ... , xn
println x1, x2, ... , xn
```

Donde `x1, x2, ... , xn` pueden ser expresiones de cualquier tipo o cadenas de caracteres encerradas en comillas dobles.

El interpretador debe recorrer los elementos en orden e imprimirlos en pantalla. La instrucción `println` imprime automáticamente un salto de línea después de haber impreso la lista completa de argumentos.

Las cadenas de caracteres deben estar encerradas entre comillas dobles (`"`) y sólo debe contener caracteres imprimibles. No se permite que tenga saltos de línea, comillas dobles o backslashes (`\`) a menos que sean escapados. Las secuencias de escape correspondientes son `\n`, `\"` y `\\`, respectivamente.

Ejemplo válido:

```
print "¡Hola, mundo! \nEsto es una comilla escapada \" y un backslash \\"
```

Que generaría la siguiente impresión en salida estándar:

```
$ ./setlan print.stl
¡Hola, mundo!
Esto es una comilla escapada " y un backslash \\
```

Condicional `if then else`

```
if (<condición>) <instrucción 1> else <instrucción 2>
if (<condición>) <instrucción 1>
```

La condición debe ser una expresión de tipo `bool`, de lo contrario debe mostrarse un error en pantalla y abortar la ejecución.

Ejecutar está instrucción tiene el efecto de evaluar la condición y si su valor es `true` se ejecuta la instrucción 1; si su valor es `false` se ejecuta la instrucción 2. Es posible omitir la palabra clave `else` y la instrucción 2 asociada, de manera que si la expresión es `false` no se ejecuta ninguna instrucción.

Note que esta especificación del condicional es **muy parecida** a la del lenguaje de programación *C*.

Ejempo válido:

```
program {
    using int x; in

    scan x;

    if (x < 0)
        print "less"
    else if (x > 0)
        print "greater"
    else
        print "equal"
    ;
}
```

En este ejemplo hay dos `if`, el primero consta de la condición `x < y`, tiene como `<instrucción 1>` a `print "less"`, y como `<instrucción 2>` a la segunda instrucción `if` con el condicional de `x > y`.

Ejemplo equivalente normalmente usado:

```
program {
    using int x; in

    scan x;

    if (x < 0) {
        print "less";
    } else if (x > y) {
        print "greater";
    } else {
        print "equal";
    };
}
```

Ejemplo equivalente:

```
program {
    using int x; in

    scan x;

    if (x < 0) {
        print "less";
    } else {
        if (x > y) {
            print "greater";
        } else {
            print "equal";
        };
    };

}
```

Obviamente es más *estético* usar el primer o segundo ejemplo, pero quiere hacerse obvio que la instrucción del `else` es un `if` más, y no es una construcción compleja del lenguaje, así evitamos escribir casos especiales para `else if`.

Iteración `for`

```
for <variable> <dirección> <expresión set> do <instrucción>
```

Para ejecutar un `for` se evalúa la expresión de tipo `set`, y a la variable se le asigna los valores de ésta. Se ejecutará la instrucción con cada valor del conjunto en orden ascendente (de menor a mayor) si `<dirección>` es la palabra `min`; si `<dirección>` es la palabra `max`, se ejecutará con los valores del conjunto de manera descendiente (de mayor a menor).

El `for` declara automáticamente a la variable `<variable>` de tipo `int` y local al cuerpo de la iteración. Esta variable es sólo de lectura y no puede ser modificada.

La `<expresión set>` debe ser evaluada como valor una sola vez antes de comenzar la ejecución de la iteración, es decir, si `<expresión set>` hace uso de una variable que es modificada dentro del cuerpo de la iteración, el nuevo valor del conjunto no es tomado en cuenta para el resto de las iteraciones pero sí para el valor de la variable al finalizar la iteración.

Ejemplo válido que hace evidente el último párrafo:

```
program {
    using
        set s;
    in
    s = {2, 3, 1};
    for i min s do {
        println i;
        s = s + {i * 2}; # unión de conjuntos
    };

    print s;
}
```

Este ejemplo imprimiría lo siguiente por salida estandar:

```
$ ./setlan for.stl
1
2
3
{1,2,3,4,6}
```

Ya que en el `for` se hizo *unión de conjuntos* (`+`) con los conjuntos `{2}`, `{4}` y `{6}`, y el valor `{2}` ya estaba, así que no se agrega de nuevo; quedando `s` con los valores `{1,2,3,4,6}`, pero no siendo considerados estos valores para el `for`, ya que generaría un ciclo infinito. Note que se recorrieron los elementos de menor a mayor.

Iteración indeterminada (`repeat while do`)

```
repeat <instrucción 1> while (<condición>) do <instrucción 2>
                       while (<condición>) do <instrucción 2>
repeat <instrucción 1> while (<condición>)
```

La condición debe ser una expresión de tipo `bool`. Para ejecutar la instrucción se ejecuta la instrucción 1, luego se evalúa la condición, si es `false` termina la iteración; si es `true` se ejecuta la instrucción del cuerpo y se repite el proceso (comenzando en la instrucción 1 de nuevo).

Nótese que tiene tres formas de usarse:

- No colocando el `repeat <instrucción 1>`, parecido a una instrucción `while (<condición>) <instrucción 2>` típica.
- No colocando la `<instrucción 2>`, parecido a una instrucción `do <instrucción 1> while` de `C`.
- Usando ambos, como fue explicado anteriormente.

Ejemplo válido:

```
program {
    using int x; in

    repeat              # primer caso, atípico pero cómodo
        scan x
    while (x > 0) do
        print x
    ;

    scan x;
    while (x > 0) do {     # segundo caso, un `while do {...}` típico
        print x;
        scan x;
    };

    scan x;
    repeat {            # tercer caso, parecido a un `do {...} while` de C
        print x;
        scan x;
    } while (x > 0);
}
```

Nótese que las tres iteraciones del ejemplo hacen cosas muy parecidas, pero dos de ellas tienen un `scan` más para lograrlo.

Reglas de alcance

Para utilizar una variable primero debe ser declarada al comienzo de un bloque o como parte de la variable de iteración de una instrucción `for`. Es posible anidar bloques e instrucciones `for` y también es posible declarar variables con el mismo nombre que otra variable en un bloque o `for` exterior. En este caso se dice que la variable interior *esconde* a la variable exterior y cualquier instrucción del bloque será incapaz de acceder a la variable exterior.

Dada una instrucción o expresión en un punto particular del programa, para determinar si existe una variable y a qué bloque pertenece, el interpretador debe partir del bloque o `for` más cercano que contenga a la instrucción y revisar las variables que haya declarado, si no la encuentra debe proceder a revisar el siguiente bloque que lo contenga, y así sucesivamente hasta encontrar un acierto o llegar al tope.

Si se llega al tope sin encontrar un acierto debe mostrarse un error en pantalla y abortar la ejecución.

El siguiente ejemplo pone en evidencia estas reglas:

```
program {
    using
        int x;              # incializado en 0
        set y;              # incializado en {}
        bool z;             # inicalizado en false
    in

    println "start";

    {
        using
            set x;          # inicializado en {}
        in
        x = {0,1}
        y = x + {2,3,4};
        println 1, x;       # x es de tipo `set`
        println 2, y;       # y es de tipo `set`
        println 3, z;       # b es de tipo `bool`
    };

    {
        using
            bool y;         # inicializado en false;
            int z;          # inicializado en 0
        in
        x = 10;
        println 1, x;       # x es de tipo `int`
        println 2, y;       # y es de tipo `bool`
        println 3, z;       # z es de tipo `int`
    };

    for i max y do
        print (i+x), " ";   # usa `i` del for y `x` del bloque principal
    println "";             # sólo para el salto de línea

    for i min {7, 5, 8, 3, 9, 6, 4, 2, 1, 0} do {
        using
            bool i;         # esconde la variable `i` del for
        in
        print i, " ";       # siempre imprime `false`
        i = false;
    };
    println "";             # sólo para el salto de línea
}
```

Expresiones

`int`

Los enteros cuentan con los típicos operadores. La suma `+`, resta `-`, multiplicación `*`, división de enteros `/`, resto de la división `%` y negación de enteros `-` (menos unario). Los operandos deben ser de tipo `int`, y su resultado es de tipo `int`.

A continuación una lista de precedencias, de menor a mayor, para estos operadores:

- `+`, `-` *(binario)*

- `*`, `/`, `%`
- `-` *(unario)*

`set`

Hay operadores para las operaciones típicas sobre conjuntos. La unión `++`, diferencia `\`, intersección `><`. Estos operadores deben tener ambos operandos de tipo `set`, y su resultado es de tipo `set`.

A continuación una lista de precedencias, de menor a mayor, para estos operadores:

- `++`, `\`
- `><`

También hay operadores que sirven para *mapear* una operación sobre todos los elementos de un conjunto, es decir, modificar todos los elementos de éste. Está la suma `<+>`, la resta `<->`, la multiplicación `<*>`, la división de enteros `</>` y el resto de la divisón `<%>`. Los operadores deben tener un operando de tipo `int` y uno de tipo `set`, y su resultado es de tipo `set`.

A continuación una lista de precedencias, de menor a mayor, para estos operadores:

- `<+>`, `<->`
- `<*>`, `</>`, `<%>`

El orden de los operandos es importante, aquí hay ejemplos ilustrativos:

- Suma `<+>`
  - `{1,2,5} <+> 2 == {3,4,7}`
  - `2 <+> {1,2,5} == {3,4,7}`
- Resta `<->`
  - `{1,2,5} <-> 2 == {-1,0,3}`
  - `2 <-> {1,2,5} == {1,0,-3}`
- Multiplicación `<*>`
  - `{1,2,5} <*> 2 == {2,4,10}`
  - `2 <*> {1,2,5} == {2,4,10}`
- División `</>`
  - `{1,2,5} </> 2 == {0,1,2}`
  - `2 </> {1,2,5} == {2,1,0}`
- Resto de la división `<%>`
  - `{1,2,5} <%> 2 == {1,0,1}`
  - `2 <%> {1,2,5} == {0,0,2}`

También hay operadores unarios sobre conjuntos, uno para conocer el máximo valor del conjunto `>?`, otro para el mínimo valor del conjunto `<?`, y otro para conocer el número de elementos del conjunto `$?` (`$` parace a una *S* de *size*, ¿verdad?... ¿no?, bueno...).

Ejemplo de uso de operadores:

- `>? {4,1,2} == 4`
- `<? {4,1,2} == 1`
- `$? {4,1,2} == 3`

`bool`

Los booleanos cuentan con los típicos operadores, la conjunción `and`, disyunción `or` y negación `not`. Los operandos de estos operadores deben ser de tipo `bool`, y su resultado también será de tipo `bool`.

A continuación una lista de precedencias, de menor a mayor, para estos operadores:

- `or`
- `and`
- `not`

Además *Setlan* cuenta con operadores relacionales capaces de hacer comparaciones entre enteros. Menor `<`, mayor `>`, menor o igual `<=`, mayor o igual `>=`, igual `==` y desigual `/=`. Ambos operandos deben ser de tipo `int`, y el resultado será de tipo `bool`.

Los operadores `==` y `/=` también se pueden usar con operandos del tipo `bool` y del tipo `set`, en ambos casos obteniendo un resultado de tipo `bool`.

Además hay un operador para la contención de conjuntos `@`, el operador debe tener un operando izquierdo de tipo `int`, un operando derecho de tipo `set`, y su resultado es de tipo `bool`.

El operador `@` en uso:

- `2 @ {1,2} == true`
- `3 @ {1,2} == false`

A continuación una lista de precedencias, de menor a mayor, para estos operadores:

- `<`, `<=`, `>`, `>=`
- `==`, `/=`
- `@`

Comentarios y espacios en blanco

En *Setlan* se pueden escribir comentarios de una línea al estilo de *Python* o *Ruby*. Al escribir `#` se ignorarán todos los caracteres hasta el siguiente salto de línea.

Los espacios en blanco son ignorados de manera similar a otros lenguajes de programación (no como *Python*), es decir, el programador es libre de colocar cualquier cantidad de espacioes en blanco entre los elementos sintácticos del lenguaje.

Ejemplos

Hello world

```
program
    println "Hello world!"
```

Programa que calcula todos los números de Fibonacci desde cero (`0`) hasta el número introducido por el usuario:

```
program {
    using
        int n;
    in

    print "input: ";
    scan n;

    if (n < 0)
        println "no negative fibonaccis";
    else if (n == 0)
        println "fib(0) = 1";
    else {
        using
            int low, high, count, tmp;
        in

        count = 0;
        low = 0;
        high = 1;
        while (count <= n) do {
            tmp = low + high;
            low = high;
            high = tmp;
            println "fib(", count, ") = ", low;
            count = count + 1;
        };
    }
    ; # del `if` exterior
}
```

Corrida:

```
$ ./setlan fib.stl
input: 5
fib(0) = 1
fib(1) = 1
fib(2) = 2
fib(3) = 3
fib(4) = 5
fib(5) = 8
```