Bitiondo

Lenguaje para operaciones en conjuntos de bits. Los archivos que contenga programas en *Bitiondo* tendrán como extensión `bto`.

Estructura de un programa

Un programa en *Bitiondo* está delimitado por las palabras `begin` y `end`, seguido de un conjunto de 0 o más declaraciones de variables (con o sin inicializar) y finalmente un conjunto de 0 o más instrucciones finalizadas por `;`. Es importante notar que el orden entre el conjunto de declaraciones de variables y el conjunto de instrucciones es estricto. Es decir, no puede existir una instrucción previa a una declaración de variable.

Un programa entonces tiene la forma general de:

```
begin
    <conjunto de declaraciones>
    <conjunto de instrucciones>
end
```

Comentarios y espacios en blanco

En *Bitiondo* 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 espacios en blanco entre los elementos sintácticos del lenguaje.

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 o por `_` 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`.
  - Almacenar enteros de más de `32 bits` no debe arrojar error, es decir, es responsabilidad del programador mantener la consistencia de su programa en el uso de enteros.
- `bool`: representa un valor booleano, es decir, `true` o `false`.
- `bits`: representa una estructura tipo arreglo que almacena bits.
  - Un bit está representado por un `1` o un `0`.
  - El tamaño del bitset se decide en declaración o inicialización y no es posible cambiarlo.
  - El operador `[]` permite el acceso directo a un bit. Es decir, dado un `bits` llamado `foo`, `foo[i]` accede al valor de su i-ésimo bit de **derecha a izquierda**, comenzando a contar desde el cero (0).

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

Expresiones y operadores

`int`

Los enteros cuentan con los operadores tradicionales sobre enteros: 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`.

Adicionalmente, se define el operador unario `@` que convierte un `int` en su representación en un elemento `bits` de tamaño 32. Este operador sólo puede ser aplicado sobre enteros positivos. Tratar de aplicarlo sobre un entero negativo debe mostrar un error y abortar ejecución.

Ejemplo:

```
begin
    int a = 2;
    outputln @a;
end
```

Que generaría:

```
$ ./bitiondo int2bits.bto
0b00000000000000000000000000000010
```

`bool`

Los `bool` cuentan con los operadores tradicionales sobre tipos booleanos: La conjunción `&&`, disyunción `||` y negación `!`. Los operandos de estos operadores deben ser de tipo `bool`, y su resultado también será de tipo `bool`.

Además se 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`, del tipo `int` y del tipo `bits`, en todo caso obteniendo un resultado de tipo `bool`.

`bits`

Una expresión de tipo `bits` tiene la forma de `0b<cadena de ceros y unos>`. Ejemplo: `0b101010`. En estas expresiones, el bit más significativo está más a la izquierda.

Existen operadores tradicionales para operaciones sobre bits. Los operandos de estos operadores deben ser de tipo `bits` y el resultado es de tipo `bits`. Estos operadores son:

- Negación (`~`): `~0b101 = 0b010`
- Conjunción (`&`): `0b101 & 0b110 = 0b100`
- Disyunción (`|`): `0b101 | 0b110 = 0b111`
- Disyunción exclusiva (`^`): `0b101 ^ 0b111 = 0b010`

Cada uno de estos operadores binarios requiere que las expresiones de lado izquiero y de lado derecho tengan la misma cantidad de elementos. En caso contrario, debe mostrarse un error y abortar la ejecución.

Adicionalmente, se proveen los operadores de desplazamiento:

- Desplazamiento hacia la derecha (`>>`): Desplaza todos los bits en el operando izquierdo la cantidad de posiciones del operando derecho hacia la derecha. Los espacios en blanco del lado izquierdo son llenados con ceros (`0`). El operando derecho debe ser un entero positivo menor al tamaño del `bits`. Ejemplo: `0b10011001 >> 2 = 0b00100110`.
- Desplazamiento hacia la izquierda (`<<`): Desplaza todos los bits en el operando izquierdo la cantidad de posiciones del operando derecho hacia la izquierda. Los espacios en blanco del lado derecho son llenados con ceros (`0`). El operando derecho debe ser un entero positivo menor al tamaño del `bits`. Ejemplo: `0b10011001 << 2 = 0b01100100`.

Por último, están definidos los operador unarios `[]` y `$`.

- `[]` permite el acceso a un bit dentro de una expresión de tipo `bits` dada una posición de derecha a izquierda, comenzando a contar desde cero (0). La expresión entre los corchetes debe evaluar a un entero positivo menor a la máxima posición posbile del `bits`. Ejemplo: `0b100[0] = 0`, `0b100[1] = 0`, `0b100[2] = 1`.
- `$` permite obtener la representación en `int` de un valor de tipo `bits`. Es importante notar que un `int` es de 32 bits, por lo que cualquier intento de representar numéricamente un valor de tipo `bits` con más o menos de 32 bits, debe producir un error y abortar la ejecución. Adicionalmente, por otras restricciones del lenguaje, es imposible que `$` produzca un valor `int` negativo.

Ejemplo:

```
begin
    bits foo[32];
    foo[1] = 1;

    outputln foo;   # Imprime 0b00000000000000000000000000000010
    outputln $foo;  # Imprime 2
end

begin
    bits foo[30];
    foo[2] = 1;

    outputln foo;   # Imprime 0b00000000000000000000000000000100
    outputln $foo;  # *** ERROR: No se puede convertir de 30 bits a int
end
```

Instrucciones

Declaración de variables

Una variable es declarada asignándole un tipo. Para las variables que han de almacenar valores de tipo `int` y `bool`, la instrucción de declaración tiene la forma de:

```
<type> <identificador>;
```

Una variable de estos dos tipos puede ser inicializada. Esto consiste en declararla y asignarle un valor en la misma instrucción:

```
<tipo> <identificador> = <valor>;
```

Una variable de tipo `bits` es declarada asignando un tamaño para el arreglo de bits de la siguiente manera:

```
<tipo> <identificador>[<tamaño>];
```

Para inicializar una variable de este tipo se usa del lado derecho de la asignación una cadena de unos y ceros precedidos por un `0b`:

```
<tipo> <identificador>[<tamaño>] = 0b<cadena de unos y ceros>;
```

La expresión del lado derecho de la asignación debe contener la misma cantidad de ceros y unos que el tamaño especificado para la variable. Si estos valores difieren, debe mostrarse un error y abortar ejecución.

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 `bits` tiene por defecto la cadena de bits del tamaño declarado donde cada bit es `0`.

Ejemplo:

```
begin
    # Variables declaradas sin inicialización
    int a;
    bool b;
    bits c[2];

    # Variables declaradas con inicialización
    int d = 1;
    bool e = true;
    bits f[2] = 0b01;
end
```

Asignación

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.

```
    <variable> = <expresión>;
```

Al igual que en la declaración con inicialización, en el caso específico de una variable de tipo `bits`, la expresión del lado derecho de la asignación debe contener la misma cantida de ceros y unos que el tamaño especificado para la variable. Si estos valores difieren, debe mostrarse un error y abortar ejecución.

Bloque

El bloque es una instrucción que consiste de una sección de declaración de variables, la cuál es opcional, y una secuencia de instrucciones finalizadas por `;`. La secuencia de instrucciones dentro de un bloque puede estar compuesta por cero (0) o más instrucciones. Tiene la siguiente forma:

```
begin
    <declaración de variables>
    <instrucción 0>;
    <instrucción 1>;
    ...
    <instrucción n>;
end
```

Las variables declaradas al principio de un bloque sólo serán visibles por las instrucciones y expresiones del bloque. Se considera un error declarar más de una vez la misma variable en el mismo bloque.

Ejemplo válido:

```
begin
    int age;
    output "how old are you?";
    input age;
    output "you said ", age, ", right?";
end
```

Entrada

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.

```
input <variable>;
```

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. Especifícamente, al hacer `input` sobre una variable de tipo `bits`, la entrada válida debe ser una cadena de ceros y unos precedida de un `0b`, además de ser del tamaño especificado para la variable correspondiente.

Por ejemplo:

```
begin
    bits b[3];
    input b;    # Una entrada válida es 0b000.
end
```

Salida

Permite la impresión en pantalla de expresiones de cualquier tipo o cadenas de caracteres.

```
output x1, x2, ... , xn;
outputln x1, x2, ... , xn;
```

El interpretador debe recorrer los elementos en orden e imprimirlos en pantalla. La instrucción `outputln` 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 deben contener caracteres válidos para impresión. 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.

NOTA: Al imprimir un valor de tipo `bits`, se deben imprimir los bits almacenados precedidos por `0b`.

Ejemplo válido:

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

Que generaría la siguiente impresión en salida estándar:

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

Ejemplo con valores:

```
begin
    bits bs[4];
    bool b;

    outputln bs, " es como ", b;

    bs = 0b1111;
    b = true;

    outputln bs, " es como ", b;
end
```

Que generaría:

```
$ ./bitiondo equivalence.bto
0b0000 es como false
0b1111 es como true
```

Condicional `if-else`

La instrucción tradicional de selección tiene la siguiente forma:

```
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.

Iteración determinada `for`

Esta instrucción consiste en una iteración hasta el cumplimiento de una condición.

```
for (<variable> = <valor inicial>; <condicion>; <paso>) <instruccion>
```

El `for` declara automáticamente a la variable `<variable>` de **sólo lectura** de tipo `int` y local al cuerpo de la iteración. La iteración se ejecuta hasta el cumplimiento de la `<condición>`, que consiste en una expresión de tipo `bool`. El valor de `<variable>` crecerá o decrecerá en valores de `<paso>`, que consiste en una expresión de tipo `int`.

Ejemplo:

```
begin
    for (b = 0; b < 3; 1)
        outputln b;          # Imprime 0, 1 y 2
end

begin
    for (b = 2; b >= 0; -1)
        outputln b;          # Imprime 2, 1 y 0
end
```

Que generaría:

```
$ ./bitiondo foroutput.bto
0
1
2
2
1
0
```

Iteración determinada `forbits`

Esta instrucción recorre un elemento de tipo `bits`.

```
forbits <expresión bits> as <identificador> from <expresión int> going (higher|lower)
  <instrucción>
```

El `<identificador>` es la variable de **sólo lectura** a la cual se le asignará el valor del bit correspondiente al ciclo de la iteración, se comporta como un `int` y puede operarse con estos, pero siempre tendrá un valor de `0` o `1`. La `<expresión int>` es una expresión de tipo `int` que indica la posición del arreglo de bits donde se empezará a iterar. La dirección de iteración se hará hacia el último bit si la dirección es `higher` y hacia el primer bit si la dirección es `lower`.

Al tratar de iterar a partir de posiciones inválidas (mayores que el tamaño del `bits`, por ejemplo) se debe mostrar un error y abortar la ejecución.

Ejemplo:

```
begin
    bits bs[4];
    bs[2] = 1;

    forbits bs as b from 0 going higher
        output b, " ";

    outputln "";

    forbits bs as b from 3 going lower
        output b, " ";
end
```

Que generaría:

```
$ ./bitiondo forbitsoutout.bto
0 0 1 0
0 1 0 0
```

Iteración indeterminada `repeat-while-do`

La instrucción de iteración determinada se presenta en varias versiones, las cuales son:

```
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 2 y se repite el proceso comenzando en la instrucción 1 de nuevo.

Nótese que tiene tres formas de usarse:

- Omitiendo el `repeat <instrucción 1>`, como una instrucción `while (<condición>) <instrucción 2>` típica.
- Omitiendo la `do <instrucción 2>`, como una instrucción `do <instrucción 1> while` de `C` (cambiando `do` por `repeat`).
- Usando ambos, como fue explicado anteriormente.

Ejemplo válido:

```
begin
    int x;

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


    input x;
    while (x > 0) do      # segundo caso, un `while do {...}` típico
        begin
            output x;
            input x;
        end

    input x;
    repeat             # tercer caso, parecido a un `do {...} while` de C
        begin
            output x;
            input x;
        end
    while (x > 0);
end
```

Nótese que las tres iteraciones del ejemplo hacen cosas muy parecidas, pero dos de ellas tienen un `input` 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` o `forbits`. Es posible anidar bloques e instrucciones `for` o `forbits` y también es posible declarar variables con el mismo nombre que otra variable en un bloque o instrucciones

`for` o `forbits` 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 `forbits` que contenga inmediatamente a la instrucción y revisar las variables que haya declarado, si no la encuentra debe proceder a revisar el bloque superior 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:

```
begin
    int x;
    outputln x;          # Imprime 0

    begin
        bits x[2];
        outputln x;      # Imprime 0b00
    end

    x = 1;
    outputln x;          # Imprime 1

    begin
        bool x;
        outputln x;       # Imprime false
    end

    bits xs[2];
    fobits xs as x from 0 going higher
        outputln x;      # Imprime 0 y 0
end
```

Que generaría:

```
$ ./bitiondo scoperules.bto
0
0b00
1
false
0
0
```

Precedencia de operadores

La tabla general de precedencia de operadores del lenguaje es la siguiente, donde más arriba es mayor precedencia:

- Unarios:
  - `[]`
  - `!`, `~`, `$`, `@`, `-`
- Binarios:
  - `*`, `/`, `%`
  - `+`, `-`
  - `<<`, `>>`
  - `<`, `<=`, `>`, `>=`
  - `==`, `!=`
  - `&`
  - `^`
  - `|`
  - `&&`
  - `||`