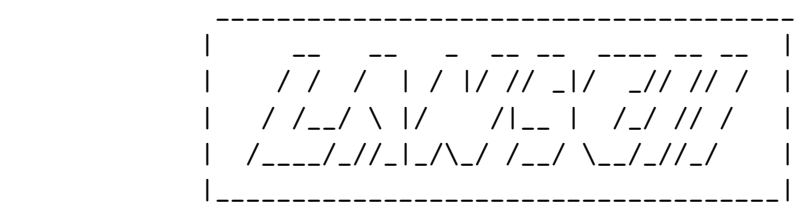
Universidad Simón Bolívar
Departamento de Computación y Tecnología de la Información
CI3725 – Traductores e Interpretadores
Abril – Julio 2015

Lanscii

Lanscii es un lenguaje de programación imperativo, diseñado con la finalidad de crear y manipular arte ascii. El siguiente es un ejemplo de lo que se podría lograr con este lenguaje:



A continuación se describe la definición del lenguaje de programación Lanscii .

1. Estructura de un programa en Lanscii

Un programa de Lanscii tiene la siguiente estructura:

```
{ <lista de declaraciones> | <instrucción> }
```

O la siguiente estructura:

```
{ <instrucción> }
```

donde las llaves { y } indican el principio y fin del programa. Note que hay dos formas de definir un programa, una declarando variables y otra sin declararlas.

La estructura de un programa tiene el mismo *comportamiento* que la instrcción de *incorporación de alcance*, que será explicada más adelante.

2. 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 con un dígito y son sensibles a mayúsculas, es decir, la variable `var` es diferente a la variable `Var`. Para este proyecto no es necesario el reconocimiento de letras acentuadas (como á, é, í, ó, ú), ni de la letra eñe (ñ).

3. Tipos de datos

Se dispone de tres tipos de datos en el lenguaje:

- Enteros: Representados por un símbolo de porcentaje (`%`), son números enteros con signo de treinta y dos (32) bits.
- Booleanos: Representados con un símbolo de exclamación (`!`), son un valor booleano, es decir, cierto `true` o falso `false`.
- Lienzos: Representados con un símbolo de arroba (`@`), son un lienzo para los caracteres ascii </>, <\>, <|>, <_>, <-> y < > (éste último es un espacio). Un lienzo vacío se representa con el caracter `#`.

4. Instrucciones

4.1. Asignación

`<ident> = <expr>`

Tiene el efecto de evaluar la expresión `<expr>` y almacenar el resultado en la variable identificada por `<ident>`. La variable `<ident>` debe haber sido declarada; en caso contrario se dará un mensaje de error. Análogamente, las variables utilizadas en `<expr>` deben haber sido declaradas y además haber sido inicializadas, i.e. se les debe haber asignado algún valor previamente; en caso contrario se dará un mensaje de error. Además, `<expr>` debe tener el mismo tipo que la variable `<ident>`; en caso contrario se dará un mensaje de error.

4.2. Secuenciación

`<instr 0> ; <instr 1>`

Esta instrucción corresponde a ejecutar la instrucción `<instr 0>` y, a continaución, la instrucción `<instr 1>`.

Note que la secuenciación de instrucciones genera **una** instrucción compuesta. La última instrucción de una secuenciación no tiene un punto y coma `;` después de ella.

4.3. Entrada

`read <ident>`

Permite obtener datos escritos por el usuario vía entrada estándar. Al ejecutar la instrucción el interpretador debe esperar a que el usuario introduzca un valor que debe ser comparado con el tipo de la variable con el identificador `<ident>`. Si el valor suministrado por el usuario es inválido, se debe repetir el proceso de lecutra. La variable puede ser solamente del tipo entero o booleano.

4.4. Salida

```
write <expr>
```

Donde `<expr>` debe ser una expresión de tipo lienzo. Esta instrucción imprime el lienzo evaluado en la expresión `<expr>`.

4.5. Condicional

```
( <expr> ? <intr 0> )
```

```
( <expr> ? <intr 0> : <instr 1> )
```

La expresión `<expr>` debe ser una expresión booleana, de lo contrario se dará un mensaje de error.

La semántica para esta instrucción es la del convencional *if-then-else*: Se evalúa la expresión `<expr>`; si ésta es verdadera, se ejecuta la instrucción `<instr 0>`, en caso contrario se ejecuta la instrucción `<instr 1>` (si la rama `:` está presente). En caso de que la expresión `<expr>` sea falsa y la rama `:` no esté presente, no se ejecuta ninguna acción.

4.6. Iteración indeterminada

```
[ <expr> | <instr> ]
```

La semántica para esta instrucción es la del convencional *while-do*: Se evalúa la expresión <`expr`>; si ésta es verdadera, se ejecuta la instrucción `<instr>` y se repite el proceso (evaluando la condición `<expr>` de nuevo); o si ésta es falsa, se abandona la ejecución de la iteración.

4.7. Iteración determinada

```
[ <expr 0> .. <expr 1> | <intr> ]
```

```
[ <ident> : <expr 0> .. <expr 1> | <intr> ]
```

Donde `<ident>` es un identificador, `<expr 0>` (límite inferior) y `<expr 1>` (límite superior) expresiones aritméticas, e `<instr>` una instrucción.

La ejecución de esta instrucción consiste en, inicialmente, evaluar las expresiones aritméticas `<expr 0>` y `<expr 1>`, lo cual determina la cantidad de veces que a continuación se ejecuta el cuerpo `<instr>` (Lo cual sería `maximo(<expr 0> - <expr 1> + 1, 0)`). En cada iteración, la variable que corresponde a `<ident>` (de estar presente) cumplirá la función de contador del ciclo obteniendo como valor, al inicio de cada iteración, la cantidad de estas iteraciones cumplidas hasta el momento (en condiciones normales) sumado al límite inferior. Así, a dicha variable le será asignado el resultado de evaluar `<expr 0>` antes de comenzar la primera ejecución de `<instr>`, el valor de evaluar (`<expr 0> + 1`) antes de la siguiente y así en adelante, hasta llegar a la evaluación de `<expr 1>` antes de la última iteración. El alcance para la variable definida en `<ident>` es únicamente la intrucción de iteración determinada a la que pertenece, por lo que el valor de la misma al momento de salir del ciclo es indiferente.

Note que dentro de `<instr>` estarán prohibidas asignaciones a la variable representada por `<ident>`. Esto, ya que si el valor de dicha variable pudiese modificarse dentro de `<instr>`, entonces la misma podría perder su rol como contador de la iteración original.

4.8. Incorporación de alcance

```
{ <lista de declaraciones> | <instr> }
```

```
{ <instr> }
```

La `<lista de declaraciones>` es una lista no vacía que enumera las declaraciones de variables y sus tipos respectivos. Estas variables pueden ser luego utilizadas únicamente en la instruccion `<instr>`. Cada declaración estará separada por espacios. Cada declaración tiene la siguiente forma:

```
<tipo> <lista de identificadores>
```

La `<lista de identificadores>` es una lista no vacía de identificadores (nombres) de variables, separados por espacios ( ). Todas las variables en esta lista compartirán el mismo tipo `<tipo>`. Estos identificadores no pueden ser palabras reservadas del lenguaje, como `true` o `false`, note que `TrUe` no es una palabra reservada.

A continuación mostramos un programa correcto

```
{ %x y @lie !boo |
    x   = 10 ;
    y   = 15 ;
    boo = false ;
    lie = # ;

    { @boo |
        boo = #
    } ;

    {
        {
            x = 42
        }
    }
}
```

5. Expresiones

5.1. Expresiones aritméticas

Una expresión aritmética estará formada por números naturales, identificadores de variables, y operadores convencionales de aritmética entera. Los operadores a ser considerados serán suma (`+`), resta (`-` binario), multiplicación (`*`), división entera (`/`), resto de división entera o módulo (`%`), y negación (`-` unario). Tal como se acostumbra, las expresiones serán construidas con notación infija para los operadores binarios, por ejemplo `1+2`, y con notación prefija para el operador unario, por ejemplo `-3`. La tabla de precedencia es también la convencional (donde los operadores más fuertes

están hacia abajo):

- `+`, `-` binario
- `*`, `/`, `%`
- `-` unario

y, por supuesto, se puede utilizar paréntesis para forzar un determinado orden de evaluación. Por tanto, evaluar `2+3/2` da como resultado `3`, mientras que evaluar `(2+3)/2` da `2`. Los operadores con igual precedencia se evalúan de izquierda a derecha. Por lo tanto, evaluar `60/2*3` da `90`, mientras que evaluar `60/(2*3)` da 10.

Expresiones con variables serán evaluadas de acuerdo al valor que éstas tengan en ese momento, para lo cual se requiere que tales variables hayan sido declaradas y previamente inicializadas. Por ejemplo, la evaluación de `x+2` da `5`, si `x` fue declarada y en su última asignación tomó valor `3`. Si `x` no fue declarada o es de un tipo no compatible para la operación, se da un mensaje de error; a este tipo de errores se les llama *estáticos*, pues pueden ser detectados antes de la ejecución del programa. Si `x` fue declarada pero no ha sido inicializada previamente, también debe darse un mensaje de error; este error sería *dinámico*, pues sólo puede ser detectado durante la ejecución del programa.

5.2. Expresiones booleanas

Análogamente a las expresiones aritméticas, una expresión booleana estará formada por las constantes `true` y `false`, identificadores de variables, y operadores convencionales de lógica booleana. Los operadores a ser considerados serán conjunción (/\), disyunción (\/), y negación (`^`). Tal como se acostumbra, las expresiones serán construidas con notación infija para los operadores binarios, por ejemplo `true` /\ `false`. Sin embargo, el operador unario (negación) será construido con notación postfija, por ejemplo `true^`. La tabla de precedencia es también la convencional (donde las operadores más fuertes están hacia abajo):

- \/
- /\

- ^

y, por supuesto, se puede utilizar paréntesis para forzar un determinado orden de evaluación. Por tanto, evaluar `true \/ true /\ false` da como resultado `true`, mientras que evaluar `(true \/ true) /\ false` da como resultado `false`. Los operadores con igual precedencia se evalúan de izquierda a derecha. Sin embargo, note que en este caso, en realidad dicho orden es irrelevante.

Expresiones con variables serán evaluadas de acuerdo al valor que éstas tengan en ese momento, para lo cual se requiere que tales variables hayan sido declaradas y previamente inicializadas. Por ejemplo, la evaluación de `x \/ false` da `true` si `x` fue declarada y en su última asignación tomó valor `true`. Si `x` no fue declarada o es de un tipo no compatible para la operación, se da un mensaje de error; a este tipo de errores se les llama *estáticos*, pues pueden ser detectados antes de la ejecución del programa. Si `x` fue declarada pero no ha sido inicializada previamente, también debe darse un mensaje de error; este error sería *dinámico*, pues sólo puede ser detectado durante la ejecución del programa.

5.3. Expresiones relacionales

Además Lanscii también contará con operadores relacionales que comparan expresiones entre sí. éstas serán de la forma `<arit> <op> <arit>`, donde ambas `<arit>` son expresiones aritméticas y `<op>` es un operador relacional. Los operadores relacionales a considerar son: menor (`<`), menor o igual (`<=`), mayor (`>`), mayor o igual (`>=`), igualdad (`=`) y desigualdad (`/=`). También será posible comparar expresiones booleanas bajo la forma `<bool> <op> <bool>`, u expresiones sobre lienzos bajo la forma `<lienzo> <op> <lienzo>`, pero con `<op>` pudiendo ser únicamente igualdad (`=`) y desigualdad (`/=`). Todas las expresiones relacionales devuelven un valor booleano al ser evaluadas.

5.4. Expresiones sobre lienzos

Análogamente a las expresiones aritméticas y booleanas, una expresión sobre lienzos estará formada por las constantes `#` (lienzo vacío), `</>`, `<\>`,

<|>, <_>, y <->, identificadores de variables, y algunos operadores sobre lienzos. Los operadores a ser considerados serán concatencación horizontal (`:`), concatenación vertical (|), rotación (`$`) y trasposición (`'`). Las expresiones serán construidas con notación infija para los operadores binarios, por ejemplo `# : </>`. El operador unario de rotación será construido con notación infija, por ejemplo `$<->`, mientras que el operador unario de traspuesta será construido con notación postfija. La tabla de precedencia será la siguiente (donde las operadores más fuertes están hacia abajo):

- |, `:`
- `$`
- `'`

y, por supuesto, se puede utilizar paréntesis para forzar un determinado orden de evaluación. Los operadores con igual precedencia se evalúan de izquierda a derecha.

Expresiones con variables serán evaluadas de acuerdo al valor que éstas tengan en ese momento, para lo cual se requiere que tales variables hayan sido declaradas y previamente inicializadas. Por ejemplo, la evaluación de `x : </>` da </> si `x` fue declarada y en su última asignación tomó valor `#`. Si `x` no fue declarada o es de un tipo no compatible para la operación, se da un mensaje de error; a este tipo de errores se les llama *estáticos*, pues pueden ser detectados antes de la ejecución del programa. Si `x` fue declarada pero no ha sido inicializada previamente, también debe darse un mensaje de error; este error sería *dinámico*, pues sólo puede ser detectado durante la ejecución del programa.

Las operaciones de concatenación de lienzos toman dos lienzos y producen un nuevo lienzo, que es la concatenación (horizontal o vertical) de los mismos. Para que dicha concatenación sea válida, se deben cumplir algunas condiciones:

- Si la concatenación es horizontal, ambos lienzos deben tener la misma dimensión vertical.
- Si la concatenación es vertical, ambos lienzos deben tener la misma dimensión horizontal.
- El lienzo `#` funciona como elemento neutro para ambas concatenaciones. (No debe tomarse como un lienzo con dimensión horizontal

y vertical igual a cero (0), sino mas bien como un lienzo de tamaño genérico, que está vacío en contenido).

La operación de rotación, toma un lienzo y lo rota `90` grados hacia la derecha (en sentido de las agujas del reloj). La operación de trasposición, toma un lienzo y reemplaza cada símbolo en la posición `(i, j)`, por el símbolo en la posición `(j, i)`, para cada `i` y `j` que quepan en dicho lienzo.

6. Comentarios

En Lanscii es posible comentar secciones completas del programa a ser ignoradas por el interpretador del lenguaje, encerrando dicho código entre los símbolos `{-` y `-}`. Estos comentarios no permiten anidamiendo (comentarios dentro de comentarios) por lo que una sección comentada termina en el primer cierre de comentario `-}` encontrado. El símbolo que los abre `{-` sí puede aparecer dentro de un comentario, pero éste será simplemente ignorado.