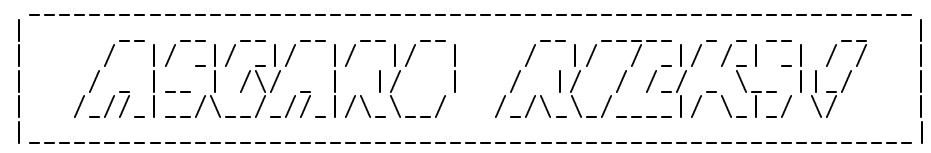
Universidad Simón Bolívar
Departamento de Computación y Tecnología de la Información
CI3725 – Traductores e Interpretadores
Abril – Julio 2012

Proyecto
Interpretador de AsGArD

AsGArD (Ascii Generated Art Dialect) es un lenguaje de programación imperativo, diseñado por los Dioses Nórdicos con la finalidad de crear y manipular arte ASCII. Este lenguaje será utilizado por los mismos como una prueba, solo aquellos que logren dominarlo tendrán permitido el paso hacia Valhalla. El siguiente es un ejemplo de lo que se podría lograr con este lenguaje:



Sabiendo que los estudiantes de Ingeniería en Computación de la U.S.B. ven un curso de Traductores e Interpretadores, los Dioses mismos les han encargado implementar un interpretador para su lenguaje, AsGArD. En el diseño del mismo, los Dioses obviaron muchos elementos comunes de los lenguajes de programación, como el manejo de estructuras de datos compuestas y procedimientos, pues habrían aumentado innecesariamente la complejidad del lenguaje a implementar por encima del tiempo disponible para su desarrollo; sin embargo se les pedirá, durante el transcurso del proyecto, analizar cómo podrían incluirse algunas de éstas en su implementación (Después de todo, quieren asegurar tu paso a Valhalla).

A continuación se describe entonces el lenguaje AsGArD, para el cual Ud. creará un interpretador. El desarrollo se realizará en 3 etapas: (I) Análisis léxico; (II) Análisis sintáctico, incluyendo la construcción del árbol sintáctico abstracto; (III) Análisis de contexto e interpretador final del lenguaje.

0 Estructura de un Programa en AsGArD

Un programa en AsGArD tiene la siguiente estructura:

[`using` ⟨*Lista de Declaraciones*⟩] `begin`
  ⟨*Instrucción*⟩
`end`

donde las palabras claves `begin` y `end` indican el principio y el fin del programa respectivamente. Note que los corchetes "[" y "]" no son parte del programa, sino que son utilizados para indicar que lo que encierran es opcional, que en este caso corresponde a la declaración de las variables del programa, precedida por la palabra clave `using`. Por otra parte, los signos "⟨" y "⟩" son utilizados

para indicar que lo que encierran es un componente del programa cuya sintaxis será explicada más adelante.

La ⟨*Lista de Declaraciones*⟩ es una lista no vacía que enumera las declaraciones de variables y sus tipos respectivos. Estas definiciones serán utilizadas luego en la ⟨*Instrucción*⟩. Las definiciones de las variables estarán separadas por punto–y–comas (“ `;`”) en la lista en cuestión. Cada definición de variables tiene la forma siguiente:

⟨*Lista de Identificadores*⟩ `of type` ⟨*Tipo*⟩

La ⟨*Lista de Identificadores*⟩, es una lista no vacía de identificadores (nombres) de variables. Todas las variables declaradas en este punto compartiran el mismo ⟨*Tipo*⟩. Cada identificador estará formado por una letra seguida de cualquier cantidad de letras y dígitos decimales. No se aceptará como identificador de variable secuencias alfabéticas que correspondan a palabras claves utilizadas en la sintaxis de AsGArD (por ejemplo: `using`, `begin`, `if` (ver Sección 1), etc.) En AsGArD se hace distinción entre mayúsculas y minúsculas, por lo que los identificadores `abcde` y `aBcdE` son diferentes; igualmente, el identificador `Begin` no es palabra clave.

El lenguaje manejará solamente variables de tipo entero (representados por la palabra clave `integer`), booleano (representados por la palabra clave `boolean`) y lienzos (representados por la palabra clave `canvas`). Las variables declaradas en este punto toman valores solamente a través de la instrucción de asignación e inicialmente no tienen valor. Se considerará un error de ejecución el tratar de usar el valor de una variable que no haya sido inicializada aún.

La sintaxis de las instrucciones (i.e. de ⟨*Instrucción*⟩) se describe en la Sección 1. Como adelanto, mostramos a continuación un ejemplo de programa escrito en AsGArD:

```
using x, y of type integer; c of type canvas begin
    x := 10 ;
    y := 15 ;
    c := <empty> ;
    from 1 to x repeat
        using d of type canvas begin
            d := <empty> ;
            with j from 1 to y repeat
                if j % 2 = 0 then
                    d := d : </>
                else
                    d := d : <\>
                done
            done ;
            c := c | d
        end
    done ;
    print c
end
```

Las partes involucradas en este ejemplo serán explicadas en la siguiente sección.

Si se ejecutase el programa anterior, la salida sería la siguiente:

```
/\/\/\/\/\/\/\/\/
/\/\/\/\/\/\/\/\/
/\/\/\/\/\/\/\/\/
/\/\/\/\/\/\/\/\/
/\/\/\/\/\/\/\/\/
/\/\/\/\/\/\/\/\/
/\/\/\/\/\/\/\/\/
/\/\/\/\/\/\/\/\/
/\/\/\/\/\/\/\/\/
/\/\/\/\/\/\/\/\/
```

1 Instrucciones

Las instrucciones permitidas en AsGArD son:

Asignación: Una asignación "⟨*Ident*⟩ `:=` ⟨*Expr*⟩" tiene el efecto de evaluar la expresión ⟨*Expr*⟩ (ver Secciones 2, 3 y 4) y almacenar el resultado en la variable ⟨*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. Además ⟨*Ident*⟩ no puede ser la variable ligada a una iteración determinada (explicada más adelante), de ocurrir este caso se dará un mensaje de error.

Secuenciación: La composición secuencial de las instrucciones ⟨*Instr0*⟩ e ⟨*Instr1*⟩ es la instrucción compuesta "⟨*Instr0*⟩ `;` ⟨*Instr1*⟩". Ésta corresponde a ejecutar la instrucción ⟨*Instr0*⟩ y, a continuación, la instrucción ⟨*Instr1*⟩.

Note que "⟨*Instr0*⟩ `;` ⟨*Instr1*⟩" es *una* instrucción compuesta. La secuenciación permite combinar varias instrucciones en una sola, que puede entonces ser, por ejemplo, *la* instrucción del cuerpo del programa principal.

Condicional: Las instrucciones condicionales de AsGArD son de la forma

"`if` ⟨*Bool*⟩ `then` ⟨*Instr0*⟩ [`else` ⟨*Instr1*⟩] `done`"

donde de nuevo es importante notar que hemos usado corchetes, "[" y "]", para indicar que lo que éstos encierran es opcional (la rama "`else`"). ⟨*Bool*⟩ es una expresión booleana (ver Sección 3), e ⟨*Instr0*⟩ e ⟨*Instr1*⟩ son instrucciones cualesquiera.

La semántica para esta instrucción es la convencional: Se evalúa la expresión booleana ⟨*Bool*⟩; si ésta es verdadera, se ejecuta ⟨*Instr0*⟩ y, en caso contrario, se ejecuta ⟨*Instr1*⟩ (si la rama "`else`" está presente). En caso de que la expresión booleana sea falsa y la rama del "`else`" no se encuentre presente, la instrucción no tendrá efecto alguno. Es decir, no se ejecutará ninguna acción.

Repetición Indeterminada: Las instrucciones de repetición indeterminada (esto es, con condiciones generales de salida) de AsGArD son de la forma

"`while` ⟨*Bool*⟩ `repeat` ⟨*Instr*⟩ `done`"

con ⟨*Bool*⟩ una expresión booleana e ⟨*Instr*⟩ una instrucción cualquiera.

La semántica para esta instrucción es la convencional: Se evalúa la expresión ⟨*Bool*⟩; si ésta es verdadera, se ejecuta el cuerpo ⟨*Instr*⟩ y se vuelve al inicio de la ejecución (preguntando nuevamente por la condición anterior) o, en caso contrario, se abandona la ejecución de la repetición.

Visto de otra forma, sea `I` una instrucción de repetición indeterminada, con forma: `while B repeat I0 done`, esta instrucción es equivalente a la secuenciación `I0 ; I` si `B` es cierto y a la instrucción vacía (una instrucción que no tiene efecto alguno) si `B` es falso.

Como ejemplo, el siguiente programa calcula el máximo común divisor entre dos números:

```
using x, y of type integer begin
    read x ;
    read y ;
    while x /= y repeat
        if x > y then
            x := x - y
        else
            y := y - x
    done
    {- El maximo común divisor ahora está en 'x' y en 'y'. -}
end
```

Repetición Determinada: Las instrucciones de repetición determinada (esto es, con cantidad fija de repeticiones) de AsGArD son de la forma

"[`with` ⟨*Ident*⟩] `from` ⟨*Aritm–Inf*⟩ `to` ⟨*Aritm–Sup*⟩ `repeat` ⟨*Instr*⟩ `done`"

con ⟨*Ident*⟩ un identificador, ⟨*Aritm–Inf*⟩ (límite inferior) y ⟨*Aritm–Sup*⟩ (limite superior) expresiones aritméticas e ⟨*Instr*⟩ una instrucción cualquiera.

La ejecución de esta instrucción consiste en, inicialmente, evaluar las expresiones aritméticas ⟨*Aritm–Inf*⟩ y ⟨*Aritm–Sup*⟩, lo cual determina la cantidad de veces que a continuación se ejecuta el cuerpo ⟨*Instr*⟩ (Lo cual sería $maximo(\langle Aritm\text{–}Inf\rangle - \langle Aritm\text{–}Sup\rangle + 1, 0)$). 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 ⟨*Aritm–Inf*⟩ antes de comenzar la primera ejecución de ⟨*Instr*⟩, el valor de evaluar (⟨*Aritm–Inf*⟩ + 1) antes de la siguiente y así en adelante, hasta llegar a la evaluación de ⟨*Aritm–Sup*⟩ antes de la última iteración. El alcance para la variable definida

en ⟨*Ident*⟩ es únicamente la intrucción de repetición determinada a la que pertenece, por lo que el valor de la misma al momento de salir del ciclo es indiferente.

Nótese 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. Si estarán permitidos cambios a posibles variables que aparezcan en ⟨*Aritm–Inf*⟩ y ⟨*Aritm–Sup*⟩, sin embargo no afectarán el rango de la repetición (Se evaluarán los límites al momento de entrar a la misma).

Incorporación de alcance: Una instrucción de incorporación de alcance en AsGArD tiene la siguiente estructura:

[`using` ⟨*Lista de Declaraciones*⟩] `begin`
⟨*Instrucción*⟩
`end`

Así es, extactamente la misma estructura que la de un programa. Esta instrucciones incorpora las nuevas declaraciones de variables (de existir) y las hace visibles/usables únicamente en la ⟨*Instrucción*⟩.

Entrada y Salida: AsGArD cuenta con instrucciones que le permiten interactuar con un usuario a través de la entrada/salida estándar del sistema de operación (indistinto para muchos sistemas de operación conocidos). Para leer un valor de la entrada las instrucciones serán de la forma

"`read` ⟨*Ident*⟩"

donde ⟨*Ident*⟩ es un identificador para una de las variables del programa. Esta variable puede ser solamente de tipo entero o booleano. La instrucción debe saber manejar la entrada en ambos casos. Para escribir en la salida las instrucciones serán de la forma

"`print` ⟨*Lienzo*⟩"

donde ⟨*Lienzo*⟩ debe ser una expresión de tipo `canvas` (Ver Sección 4).

2 Expresiones Aritméticas

Una expresión aritmética está 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 (`%`), e inverso (`-` 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 tanto, evaluar `60/2*3` da `90`, mientras que evaluar `60/(2*3)` da `10`.

El valor de expresiones con variables es calculado de acuerdo al valor que estas últimas tengan al momento de ser evaluadas, 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.

3 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`. Como no existen operadores de igual precedencia, la asociatividad de los mismos es irrelevante.

El valor de expresiones con variables es calculado de acuerdo al valor que estas últimas tengan al momento de ser evaluadas, 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.

Además AsGArD también contará con operadores relacionales que comparan expresiones entre sí. Éstas serán de la forma "⟨*Aritm*⟩ ⟨*Rel*⟩ ⟨*Aritm*⟩", donde ambas ⟨*Aritm*⟩ son expresiones aritméticas y ⟨*Rel*⟩ 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*⟩ ⟨*Rel*⟩ ⟨*Bool*⟩", u expresiones sobre lienzos bajo la forma "⟨*Lienzo*⟩ ⟨*Rel*⟩ ⟨*Lienzo*⟩", pero con ⟨*Rel*⟩ pudiendo ser únicamente igualdad (`=`) y desigualdad (`/=`).

4 Expresiones sobre Lienzos

Análogamente a las expresiones aritméticas y booleanas, una expresión sobre lienzos estará formada por las constantes `<empty>`, `</>`, `<\>`, `<|>`, `<_>`, `<->` y `< >` (espacio en blanco), identificadores de variables, y algunos operadores sobre lienzos. Los operadores a ser considerados serán concatencación horizontal (`:`), concatenación vertical (`|`), rotación (`$`) y trasposición (`'`). Las expresiones serán construidas con notación infija para los operadores binarios, por ejemplo `<empty> : </>`. El operador unario de rotación será construido con notación infija, por ejemplo `$<->`, mientras que el operador unario de trasposición será construido con notación postfija, por ejemplo `<->'`. 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.

El valor de expresiones con variables es calculado de acuerdo al valor que estas últimas tengan al momento de ser evaluadas, 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 `<empty>`. 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 `<empty>` 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.

Por ejemplo:

```
x = //          y = |-|          z = y' = |-
    \\              -|-                   -|
                                          |-

    x : y = //|-|            x | z = //
            \\-|-                    \\
                                     |-
                                     -|
                                     |-

$y = -|          $$y = -|-         $$$y = |-
     |-                 |-|               -|
     -|                                    |-
```

Notemos que aplicar tres veces el operador de rotación sobre el lienzo `y`, resulta en lo mismo que aplicar el operador de trasposición sobre el mismo. Este no es el caso general.

5 Comentarios

En AsGArD es posible comentar secciones completas del programa, para que sean ignorados por el interpretador del lenguaje, encerrando dicho código entre los simbolos ‘‘{-’’ y ‘‘-}’’. Estos comentarios no permiten anidamiento (comentarios dentro de comentarios) por lo que dentro de una sección comentada debe prohibirse el símbolo que cierra comentarios (‘‘-}’’). El símbolo que los abre (‘‘{-’’) puede reaparecer en el comentario, sin embargo no tendrá efecto alguno (será ignorado como el resto de la sección comentada).