Universidad Simón Bolívar
Departamento de Computación y Tecnología de la Información
CI3725 – Traductores e Interpretadores
Octubre 2013 – Enero 2014

Proyecto
Intérprete de Brainiac

Brainiac es un lenguaje de programación imperativo, diseñado por el temible y astuto profesor Moriarty para comunicar en secreto sus intenciones a sus subordinados. Armados con un intérprete para el mismo, los subordinados de Moriarty pueden ejecutar los programas y llevar a cabo los siniestros deseos de su jefe.

Con la cooperación del Dragón Morado, informante de confianza en asuntos de lenguajes y compiladores, el persipicaz Sherlock Holmes logra interceptar los programas escritos en Brainiac por el profesor. Usando su conocimiento y habilidades de deducción, logra inferir la definición del lenguaje con absoluta precisión y detalle. Sin embargo, ni Holmes ni Whatson en todo su conocimiento tienen idea de como implementar un intérprete para un lenguaje. Por ello, sin perder tiempo, se ponen en contacto con el CEIC (*Coordinación Estratégica Internacional de Computistas*) para que los ayuden con esta crucial tarea.

Sabiendo que los capaces estudiantes de Ingeniería en Computación de la U.S.B. ven un curso de Traductores e Interpretadores, el CEIC ha delegado esta importante labor a ellos. Es conocido que el ataque del profesor será a finales del trimestre en curso, por lo que el intérprete debe estar listo a tiempo para sabotear sus planes. Ustedes son los únicos con las capacidades y entrenamiento necesarios para tener éxito en el tiempo disponible. El destino de la sociedad está en sus manos.

El diseño original del lenguaje obvia muchos elementos usuales de los lenguajes de programación, como el manejo de estructuras de datos compuestas y procedimientos. Sin embargo, el profesor Moriarty no se arriesgará al fracaso y posiblemente cambie la definición del lenguaje para futuras comunicaciones. Siendo así y previendo tales cambios, se les pedirá durante el transcurso del proyecto analizar cómo podrían incluirse algunas de estas características adicionales en su implementación.

A continuación se describe entonces el lenguaje Brainiac, para el cual Ud. creará un interpretador. El desarrollo se realizará en 4 etapas: (I) Análisis lexicográfico; (II) Análisis sintáctico y construcción del árbol sintáctico abstracto; (III) Análisis de contexto e (IV) Intérprete final del lenguaje.

0. Estructura de un Programa en Brainiac

[`declare` ⟨*Lista de Declaraciones*⟩] `execute`
  ⟨*Instrucción*⟩
`done`

donde las palabras claves `execute` y `done` 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 `declare`. 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 utilizadas en la ⟨*Instrucción*⟩. Las definiciones de las variables estarán separadas por punto–y–comas ("`;`") en la lista en cuestión. Cada definición de variable tiene la forma siguiente:

⟨*Lista de Identificadores*⟩ `::` ⟨*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 Brainiac (por ejemplo: `declare`, `execute`, `if` (ver Sección 1.), etc.) En Brainiac se hace distinción entre mayúsculas y minúsculas, por lo que los identificadores `abcde` y `aBcdE` son diferentes; igualmente, el identificador `Execute` 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 cinta (representados por la palabra clase `tape`). Las variables 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 Brainiac que imprime "Hello World!" (sin las comillas):

```
declare x, y :: integer; t :: tape execute
    x := 3 ;
    y := 15 ;
    from 1 to x do
        t := [y] ;
        {++++++++++} at t;
        while #t /= 0 do
            {>+++++++>++++++++++>+++>+<<<<-} at t
        done :
        {>++.>+.+++++++..+++.>++.} at t;
        {<<++++++++++++++.>.+++.} at t;
        {------.--------.>+.>.} at t
    done
done
```

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

1. Instrucciones

Las instrucciones permitidas en BRAINIAC 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.

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 BRAINIAC 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.

Iteración Indeterminada: Las instrucciones de iteración indeterminada (esto es, con condiciones generales de salida) de BRAINIAC son de la forma

"`while` ⟨*Bool*⟩ `do` ⟨*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 iteración.

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

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

```
declare x, y :: integer execute
    read x ;
    read y ;
    while x /= y do
        if x > y then
            x := x - y
        else
            y := y - x
        done
    done
    $- El maximo común divisor ahora está en 'x' y en 'y'. -$
end
```

Iteración Determinada: Las instrucciones de iteración determinada (esto es, con cantidad prefijada de repeticiones) de BRAINIAC son de la forma

"`for` [⟨*Ident*⟩ `from`] ⟨*Aritm–Inf*⟩ `to` ⟨*Aritm–Sup*⟩ `do` ⟨*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 el máximo entre (⟨*Aritm–Inf*⟩ - ⟨*Aritm–Sup*⟩ + 1) y 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 valor de ⟨*Ident*⟩ al momento de salir del ciclo será (⟨*Aritm–Sup*⟩ + 1) en caso de que se ejecute al menos una iteración y 0 de lo contrario.

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. La variable ⟨*Ident*⟩ debe haber sido previamente declarada o se debe darse un mensaje de error.

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

[`declare` ⟨*Lista de Declaraciones*⟩] `execute`
⟨*Instrucción*⟩
`done`

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

Entrada y Salida: BRAINIAC 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

"⟨*write*⟩ ⟨*Expr*⟩"

donde ⟨*Expr*⟩ puede ser una expresión de cualquier tipo. La instrucción debe saber manejar la entrada en todos los casos.

En el caso de cintas, esta instrucción imprimirá el contenido entero de la misma interpretando los valores en cada posición como un caracter codificado en ASCII (los valores no imprimibles serán ignorados).

2. 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 (`%`), 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` resulta en `90`, mientras que evaluar `60/(2*3)` resulta en `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` resulta en `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 prefija, 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` resulta en `true`, mientras que la evaluación de `(true` \/ `true)` /\ `false` resulta en `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` resulta en `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 Brainiac 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*⟩", con ⟨*Rel*⟩ pudiendo ser únicamente igualdad (`=`) y desigualdad (`/=`).

4. Expresiones sobre Cintas

A diferencia de las expresiones artiméticas y booleanas, las cintas no cuentan con constantes literales. A cambio, tienen un constructor `[`⟨*Num*⟩`]`, donde ⟨*Num*⟩ es una expresión entera que dictará el tamaño de la cinta. Si el valor de ⟨*Num*⟩ no es positivo se deberá dar un mensaje de error; este error sería *dinámico*, pues sólo puede ser detectado durante la ejecución del programa. Inicialmente, todas las posiciones de la cinta tendrán el valor cero (0). Además, pueden ser identificadores de variables y la aplicaciones de algunos operadores sobre cintas.

Los operadores a ser considerados serán concatencación (`&`), inspección (`#`) y ejecución (`at`). Las concatenación será aplicada con notación infija, por ejemplo `[3] & y`. El operador de inspección será aplicado con notación infija, por ejemplo `#x`. Igualmente, el operador de ejecución será construido con notación prefija, tomando la cadena de *B-instrucciones* a ser ejecutadas, por ejemplo `{+>-<.} at [3]`. La tabla de precedencia será la siguiente (donde las operadores más fuertes están hacia abajo):

```
&
#
at
```

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 `[3] & x` resulta en `[5]` si `x` fue declarada y en su última asignación tomó valor `[2]`. 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.

La operación de inspección toma una cinta y devuelve el valor que se encuentra almacenado en la primera posición de la misma.

La operación de ejecución toma una cadena de *B-instrucciones* y las aplica de izquierda a derecha sobre la cinta propuesta. Dichas cadenas están encerradas entre llaves '`{`' y '`}`'.

B-instrucciones: Cada *B-instrucción* válida se describe a continuación:

- (+) Suma 1 al valor almacenado en la primera posición de la cinta.
- (-) Resta 1 al valor almacenado en la primera posición de la cinta.
- (>) Rota la cinta una posición hacia la derecha (por ejemplo, la primera posición original ahora será la segunda).
- (<) Rota la cinta una posición hacia la izquierda (por ejemplo, la segunda posición original ahora será la primera).
- (.) Imprime el valor almacenado en la primera posición de la cinta (interpretado como un valor ASCII).
- (,) Lee un valor de la entrada estándar (interpretado como un valor ASCII) y lo almacena en la primera posición de la cinta.

5. Comentarios

En Brainiac 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).

Además, es posible comentar una línea completa utilizando el símbolo **$$**.

Referencias adicionales:

- La definición de este lenguaje está inspirada en el lenguaje de programación esotérico Brainf**k, que pueden revisar en el siguiente enlace: http://esolangs.org/wiki/Brainf**k
- La historia asociada al proyecto está inspirada en la serie de novelas por Sir Arthur Conan Doyle, protagonizadas por Sherlock Holmes. Mas información en el siguiente enlace: http://en.wikipedia.org/wiki/Sherlock_Holmes