Universidad Simón Bolívar
Dpto. de Computación y Tecnología de la Información
CI3725 - Traductores e Interpretadores
Abril-Julio 2009

Proyecto Unico
Interpretador de `Yisiel`

A continuación se describe a grandes rasgos el lenguaje `YISIEL`, para el cual Ud. creará un interpretador. El lenguaje es muy similar al lenguaje `GCL` [1], con algunas extensiones interesantes para este curso. Se espera de Ud. que consulte aquellos elementos que le parezcan ambigüos o que necesiten una descripción más clara, puesto que en esta definición se han dejado de lado varios detalles en el interés que sean los estudiantes los que encuentren tales aspectos.

El desarrollo se realizará en tres partes:

1. Análisis lexicográfico.
2. Análisis sintáctico y alimentación de la Tabla de Símbolos.
3. Construcción del árbol abstracto, análisis de contexto e interpretación.

Estructura de un Programa

Un programa consiste de instrucciones para definir variables globales, definiciones de procedimientos adicionales y las instrucciones para definir el programa principal. Las instrucciones de definición de variables globales al programa, si existen, siempre estarán al principio del mismo. Las instrucciones para definir procedimientos, si existen, estarán después de las variables globales. Las instrucciones para definir variables locales a un procedimiento siempre estarán al principio del cuerpo del procedimiento.

Definiendo variables

El lenguaje opera con variables enteras o con arreglos de enteros. Para definir una o más variables se escribe

`var` $< id_0 >$`,` $< id_1 >$`, ...,` $< id_n >$ `:` $< tipo >$

pudiendo haber tantos $< id >$ como sean necesarios, mientras que $< tipo >$ puede ser

- `value`, indicando que cada una de las variables contiene un valor entero.
- `array of` $< entero >$, indicando que cada una de las variables contiene un arreglo de valores enteros con subíndices entre 0 y $< entero > -1$

Pueden usarse tantas definiciones de variable como se desee, siempre y cuando estén todas juntas al inicio del programa o del procedimiento en el cual aparezcan. Por ejemplo:

```
var foo, bar, baz is : value
var qux : value
var grok, bleh : array of 42
```

Definiendo Procedimientos

La definición de procedimientos permite asociar una instrucción con un nuevo nombre, que luego puede ser utilizado como cualquier otra instrucción. Esto es, para definir una nueva instrucción se escribe

`proc` $< id >$ `(` $< modo >< id_0 >$`, ... ,` $< modo >< id_n >$ `)`
  `as` $< locales >< instruccion >$

donde $< id >$es cualquier identificador alfanumérico para denotar el procedimiento. En el cuerpo del procedimiento $< locales >$ es la lista de definición de variables locales al procedimiento, que puede estar vacía, mientras que $< instruccion >$ es cualquier instrucción válida en el lenguaje. La lista de parámetros formales puede estar vacía. El $< modo >$ puede ser

- `in`, para indicar que el parámetro es de entrada. En este caso el parámetro se pasa por *valor*.
- `out`, para indicar que el parámetro es de salida. En este caso el parámetro se pasa por *copia-resultado*.

Los parámetros *siempre* serán valores simples, en otras palabras no pueden pasarse arreglos como parámetros. Los nombres de los parámetros formales se consideran variables locales accesibles solamente dentro de la $< instruccion >$. Los procedimientos pueden ser recursivos.

Instrucciones Permitidas

Vacía: La instrucción vacía se utiliza cuando la sintaxis requiere una instrucción, pero se desea que el programa no haga nada en ese caso.

```
skip
```

Asignación: permiten almacenar el valor de una expresión en una variable. El lenguaje permite realizar asignaciones simples o múltiples tanto en valores como en posiciones particulares de arreglos, e.g.

`foo <-` $< expr >$
`grok [`$< expr_0 >$`] <-` $< expr_1 >$
`foo, bar <-` $< expr_0 >$`,` $< expr_1 >$

en el caso de las asignaciones múltiples, primero se evalúan *todas* las expresiones del lado derecho para luego ser asignadas en las variables correspondientes del lado izquierdo.

Selección: Es una lista de comandos con guardias, de los cuales se selecciona exactamente uno para su ejecución, e.g.

`if` $< guard_0 >$ `->` $< instruccion_0 >$
   $< guard_1 >$ `->` $< instruccion_1 >$
$\vdots$
   $< guard_n >$ `->` $< instruccion_n >$
`fi`

Para ejecutar una selección, *todas* las guardias son evaluadas. Si ninguna es cierta, termina la ejecución del programa con un error a tiempo de ejecución. Si una o más son ciertas, se selecciona la instrucción correspondiente a una de ellas y se ejecuta.

Repetición: Ejecuta los comandos con guardias hasta que *ninguno* sea cierto, e.g.

```
do < guard_0 > -> < instruccion_0 >
   < guard_1 > -> < instruccion_1 >
⋮
   < guard_n > -> < instruccion_n >
od
```

Para ejecutar una repetición, *todas* las guardias son evaluadas. Si ninguna es cierta, termina la repetición. Si una o más son ciertas, se selecciona la instrucción correspondiente de una de ellas y se ejecuta, para luego continuar con la ejecución de la repetición.

Bloques: Permite combinar varias instrucciones simples para construir una instrucción compuesta. Se utiliza el punto y coma como *separador* de instrucciones en un bloque.

```
begin
  < instruccion_0 >;
  < instruccion_1 >;
   ... ;
  < instruccion_n >
end
```

Puede utilizarse un bloque en cualquier lugar donde iría una instrucción, incluso dentro de un bloque.

Retornar de Procedimiento: Permite finalizar la ejecución de un procedimiento,

```
return
```

retornando inmediatamente a la instrucción siguiente a la invocación del procedimiento. Note que la *única* forma de retornar valores resultantes de un procedimiento es mediante el uso de parámetros `out`.

Invocar Procedimiento: Permite invocar un procedimiento previamente definido, suministrando la lista de parámetros actuales para la llamada

$$< id > \texttt{(}\ < expr_0 >, < expr_1 >, \ \ldots\ , < expr_n > \texttt{)}$$

donde $< id >$ es el nombre del procedimiento a invocar. Se evalúan las expresiones de izquierda a derecha y sus resultados son utilizados para constituir los parámetros formales del procedimiento: los parámetros de entrada (con modo `in` en la definición del procedimiento) pueden ser exrpesiones arbitrarias, pero los parámetros de salida (con modo `out` en la definición del procedimiento) necesariamente tienen que ser variables.

Emisión de mensajes: Permite mostrar resultados por pantalla

`show` $< expr >$

Como un caso particular, pueden emitirse mensajes de texto por la pantalla utilizando

`show` $< string >$

donde $< string >$ es una cadena de caracteres literales.

Programa Principal

Solamente puede haber un bloque de programa principal, cuya presencia es por demás obligatoria y se reconoce con el bloque

```
main
  ... instrucciones ...
end
```

Tiene la misma estructura que un **Bloque**, salvo por la palabra reservada `main` en lugar de `begin`. La primera instrucción del bloque será la primera instrucción a ejecutar tan pronto el programa comience a ser interpretado.

Expresiones Aritméticas

- El lenguaje permite realizar sumas (`+`), restas (`-`), multiplicaciones (`*`), división entera (`/`) y residuo (`%`) utilizando las precedencias habituales de estos operadores, los cuales asocian hacia la izquierda. Si alguna operación produce un desbordamiento u operación indefinida, el programa abortará la ejecución con un mensaje de error descriptivo.
- Pueden utilizarse paréntesis libremente para indicar el orden de las operaciones.
- Pueden utilizarse números enteros literales en las expresiones.
- Puede utilizarse un $< id >$ dentro de una expresión para referirse al valor simple almacenado en la variable denotada.
- Puede utilizarse $< id >$`[` $< expr >$`]` dentro de una expresión para denotar el valor almacenado en la posición $< expr >$ dentro del arreglo $< id >$. Si la expresión $< expr >$ tiene un valor fuera del rango del arreglo, el programa abortará inmediatamente con un error a tiempo de ejecución.
- La expresión `$`$< id >$ solamente es válida si $< id >$ es un arreglo, y se obtiene su dimensión, e.g. la expresión `$grok` resultaría en `42`.

Expresiones para las Guardias

- Pueden utilizarse los operadores booleanos menor que (`<`), menor o igual que (`<=`), mayor que (`>`), mayor o igual que (`>=`), igual que (`=`) y diferente de (`!=`) entre dos expresiones cualesquiera. Estos operadores no son asociativos.
- Pueden utilizarse los conectores booleanos para conjunción (`&&`), disyunción (`||`) y negación (`~`) entre dos expresiones lógicas, con la precedencia y asociatividad habituales. Los conectores booleanos tienen precedencia sobre los operadores booleanos descritos en el punto anterior.

- Pueden utilizarse paréntesis libremente para indicar el orden de las operaciones.
- Pueden utilizarse las expresiones booleanas `true` y `false` explícitamente.

Elementos Lexicográficos

- Los identificadores ($< id >$) deben *comenzar* por una letra, y puede tener tantas letras, dígitos o caracteres *underscore* (_) como se desee. El lenguaje **es** sensible a la diferencia entre mayúscula y minúscula, i.e. `foo`, `Foo`, `FOO`, `FoO`, `fOO` y `fOo` son seis identificadores diferentes, más aún `VAR` y `PROC` **son** nombres válidos para variables o procedimientos pues **no son** lo mismo que las palabra reservadas `var` ni `proc`.
- Todos los números literales utilizados en el lenguaje son enteros con signo negativo opcional en base 10.
- Todos los caracteres desde el caracter numeral (#) hasta el final de la línea se consideran comentarios del programador y pueden ser ignorados. Así mismo, se consideran comentarios cualquier bloque de texto similar a

  ```
  {#
     Esto es un comentario
     de muchas líneas
  #}
  ```

  Nótese que en el texto del comentario no puede aparecer la secuencia `{#` nuevamente.
- Los espacios en blanco, tabuladores y saltos de línea han de ser ignorados.
- Las cadenas literales de caracteres comienzan con comilla simple (') y terminan con comilla simple, o bien comienzan con comilla doble (") y terminan con comilla doble. Esto permite que las primeras contengan la comilla doble y las segundas contengan la comilla simple. *Cualquier* caracter es válido dentro de una cadena literal *excepto* un salto de línea. La secuencia especial `\n` debe interpretarse como el salto de línea, mientras que la secuencia especial `\\` debe interpretarse como la barra invertida.

Referencias

[1] *Guarded Command Language*
`http://en.wikipedia.org/wiki/Guarded_Command_Language`