Modificación de la Práctica de CL

14 de Abril de 2010

A continuación se presenta la modificación de la práctica que debéis realizar, junto con una guía de ayuda, que no es obligatorio seguir, y que quizás requiera de adaptaciones en vuestra práctica dependiendo de cómo la hayáis programado. Esta guía contiene también la información de cómo se realizará la evaluación, en base a unos juegos de prueba que podéis encontrar en el racó, en `examens.fib.upc.edu` accediendo a la entrega correspondiente, incluidos en el fichero `infoexamen.tar`, cuyo contenido podéis extraer con `tar xf infoexamen.tar`, para ejecutar a continuación `tar xf jp.tar`. El fichero `infoexamen.tar` contiene además otros ficheros que os permitirán realizar una autoevaluación. Al final se indica cómo realizar dicha autoevaluación y la entrega vía el racó. Es recomendable ir realizando entregas a medida que vuestra práctica vaya superando más juegos de pruebas.

3 puntos de la nota vienen dados por pasar los juegos de pruebas básicos de la práctica. El resto depende de lo que hagáis a continuación.

Queremos introducir, en el lenguaje CL, un par de modificadores a las declaraciones de funciones y procedimientos, con las siguientes funcionalidades:

- *Shadow* fuerza a que la función con este modificador oculte una función ya existente en un ámbito superior.
- *Overload* permite que dos o más funciones con este modificador tengan el mismo identificador siempre y cuando ambas secuencias de parámetros formales sean distintas (es decir, permite la sobrecarga de funciones).

Estos modificadores los colocaremos después de la lista de parámetros formales en el caso de declaraciones de procedimientos, y después del tipo del valor de retorno para las declaraciones de funciones. Por ejemplo:

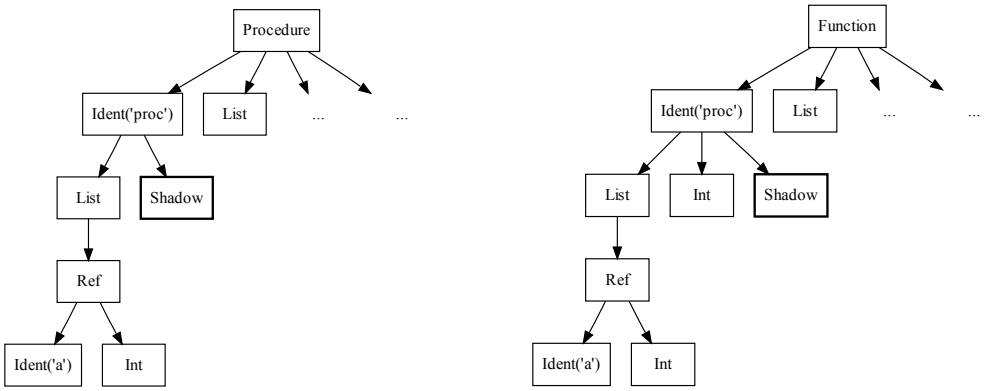
```
Program
  Vars x Int EndVars
  Procedure p1()
  EndProcedure
  Procedure p2()
    Procedure p1() Shadow // Se acepta: se oculta p1 en el ámbito padre
    EndProcedure
    Function f1() Return Int Shadow // Error: no hay ningún f1 que ocultar
      Return 1
    EndFunction
  EndProcedure
  Procedure p3(Val a Int) Overload
  EndProcedure
  Procedure p3(Val b Bool) Overload // Se acepta esta sobrecarga de p3
  EndProcedure
EndProgram
```

Por simplicidad, no vamos a permitir que una función pueda estar etiquetada con los dos modificadores a la vez. Por lo tanto, podéis empezar por cualquiera de los dos modificadores, pero os sugerimos que lo hagáis en el orden indicado por los juegos de prueba.

SHADOW

El primer modificador, `shadow`, en una declaración de una función o procedimiento *f*, indica de manera explícita que esta *f* oculta otra función o procedimiento ya existente en un ámbito superior (con el mismo nombre). Si no existiera otra función *f* en un ámbito superior al que contiene la declaración de *f*, se dará un mensaje de error.

El primer paso será extender la gramática y añadir las definiciones de tokens necesarias para el nuevo modificador. Esto os modificará el AST resultante para declaraciones de funciones y procedimientos, y por lo tanto tenéis que asegurar que vuestro `TypeCheck` recorre correctamente los AST de declaraciones que contengan el modificador `shadow`. Os sugerimos que uséis un AST como el siguiente para procedimientos y funciones con este modificador:



(1,5 puntos) Para el siguiente juego de pruebas, sólo será necesario reconocer correctamente el token `shadow` y que las declaraciones de funciones y procedimientos sigan funcionado correctamente. De hecho, no hay ningún error generado por `shadow`.

```
 1: Program
 2:   Procedure proc(Ref a Int)
 3:   EndProcedure
 4:   Function func(Val a Int) Return Int
 5:     Return a
 6:   EndFunction
 7:   Procedure sub()
 8:     Procedure proc(Ref a Int) Shadow
 9:       proc(2)
10:     EndProcedure
11:     Function func(Val a Int) Return Int Shadow
12:       a := True
13:       Return True
14:     EndFunction
15:     Function func(Val a Bool) Return Int Shadow
16:       Return 2
17:     EndFunction
18:   EndProcedure
19: EndProgram

Type Checking:
L. 9: Parameter 1 is expected to be referenceable but it is not.
L. 12: Assignment with incompatible types.
L. 13: Return with incompatible type.
L. 15: Identifier func already declared.
```

Cuando se analice la cabecera de una función decorada con `shadow`, se deberá buscar en la tabla de símbolos si ya existe una función con el mismo identificador, y si **no** existe, dar un error. Para simplificar, no hace falta comprobar que el identificador ocultado es en efecto una función o procedimiento.

En el archivo *newsemanticerrors.cc* que se facilita con este examen está disponible la función que genera el nuevo mensaje de error. Debéis añadir esta función al fichero *semantic.cc* e invocarla adecuadamente.

(1 punto) Así podréis pasar el segundo juego de pruebas.

```
 1: Program
 2:   Vars
 3:   EndVars
 4:   Procedure p0(Val a Int, Val b Int)
 5:     Procedure p1(Val a Int) Shadow
 6:     EndProcedure
 7:   EndProcedure
 8:   Procedure p1(Val a Int)
 9:     Procedure p1() Shadow
10:     EndProcedure
11:     Procedure p10()
12:     EndProcedure
13:   EndProcedure
14:   Procedure p2(Val a Int)
15:     Procedure p3(Val a Int) Shadow
16:       Procedure p3(Val a Int) Shadow
17:       EndProcedure
18:     EndProcedure
19:   EndProcedure
20:   Procedure p4(Val a Int) Shadow
21:     Procedure p10() Shadow
22:     EndProcedure
23:   EndProcedure
24: EndProgram

Type Checking:
L. 15: Procedure or function p3 does not hide any existing one.
L. 20: Procedure or function p4 does not hide any existing one.
L. 21: Procedure or function p10 does not hide any existing one.
```

OVERLOAD

El segundo modificador permite la sobrecarga de procedimientos o funciones, que será implementada mediante una estrategia conocida como la decoración de identificadores.

En esta estrategia, dos funciones f con parámetros formales diferentes en realidad tienen entradas diferentes en la tabla de símbolos, ya que cada entrada es previamente decorada con información sobre los parámetros de la función correspondiente. Por ejemplo, si la primera función toma un parámetro de tipo entero y la segunda toma un parámetro de tipo booleano, la primera función se insertaría en la tabla de símbolos con el identificador *f_i* y la segunda con el identificador *f_b*.

Para simplificar, asumiremos que si una función o procedimiento está declarado como `overload`, no podrá haber una declaración sin `overload` de la misma función o procedimiento. Además, también supondremos que todos los `overload`s están en el mismo ámbito

Las reglas que usaremos para decorar los símbolos serán:

- Por cada parámetro de tipo *int*, se añadirá la letra `i` al identificador.
- Por cada parámetro *bool*, se añadirá la letra `b`.
- Por cada parámetro de tipo *array*, se añadirá la letra `a`, seguida del número de elementos del array, y el tipo de los elementos siguiendo estas mismas reglas.
- Para cada *struct*, se añadirá la letra `s`, seguida del nombre y tipo de cada elemento del struct en orden (con el formato `<nombre,tipo>`), y la letra `e` para indicar el fin del struct.
- Ignoraremos si el parámetro es por valor o por referencia.
- Las funciones y procedimientos que no tengan el modificador Overload seguirán como hasta ahora.
- En el caso de funciones, se ignorará el tipo de retorno.

Por ejemplo:

```
Procedure  p(Val a Int,
             Val b Bool,
             Val c Array[10] of Array[5] Of Bool,
             Val d Struct z Int EndStruct) Overload
EndProcedure
```

Se convertirá en el símbolo: `p_iba10a5bs<z,i>e`

En el momento de verificar las llamadas, será necesario deducir a partir del nombre de la función llamada, las reglas anteriores y los parámetros reales, qué identificador decorado sería compatible con esos parámetros, y entonces comprobar si ése identificador existe en la tabla de símbolos. Si no existe, daremos el error de función no existente como hasta ahora.

El primer paso será extender la gramática y añadir las definiciones de tokens necesarias para el nuevo modificador. Recordad que en esta modificación no se permitirá que una función esté decorada con dos o más modificadores. Os sugerimos un AST similar al del modificador `shadow` para funciones etiquetadas con `overload`.

(1 punto) Para el siguiente juego de pruebas, sólo será necesario reconocer correctamente el token `overload` y que las declaraciones de funciones y procedimientos sigan funcionado correctamente.

```
 1: Program
 2:   Vars
 3:     x Int
 4:     z Bool
 5:   EndVars
 6:   Procedure proc1(Ref a Int)
 7:     a := a + 1
 8:   EndProcedure
 9:   Procedure proc2(Val a Int) Overload
10:     x := a
11:     z := a
12:   EndProcedure
13:   Function func1(Val a Int) Return Int Overload
14:     x := a
15:     z := a
16:     Return a
17:   EndFunction
18:   Function func2(Val a Int) Return Bool Overload
19:     x := a
20:     Return a
21:   EndFunction
```

```
 22: EndProgram

Type Checking:
L. 11: Assignment with incompatible types.
L. 15: Assignment with incompatible types.
L. 20: Return with incompatible type.
```

Ahora tenéis que implementar una función que, teniendo como entrada un `ptype` genere el *string* correspondiente a la decoración detallada en las anteriores normas (por ejemplo, dado un `ptype` de kind *bool* devuelva `"b"`). Usaréis esta función para decorar los identificadores de las declaraciones de funciones y procedimientos marcados con `overload` justo antes de llamar a `InsertIntoST` para insertarlos en la tabla de símbolos.

Salvo para los dos últimos juegos de prueba, sólo será necesario implementar los casos de parámetros de tipos *bool* y *int*; podéis devolver una cadena vacía para el resto.

(1,5 puntos) Con eso, podréis pasar el siguiente juego de pruebas.

```
  1: Program
  2:   Procedure proc(Val a Int, Val b Int) Overload
  3:   EndProcedure
  4:   Procedure proc(Val a Int) Overload
  5:   EndProcedure
  6:   Procedure proc(Val a Int, Val b Int, Val c Int) Overload
  7:   EndProcedure
  8:   Procedure proc(Val a Int, Val b Bool) Overload
  9:   EndProcedure
 10:   Procedure proc(Val a Int, Val b Int) Overload
 11:   EndProcedure
 12:   Function func(Val a Int, Val b Bool) Return Int Overload
 13:     Return 1
 14:   EndFunction
 15:   Function func(Val a Bool) Return Int Overload
 16:     Return 2
 17:   EndFunction
 18:   Function func(Val a Bool) Return Int Overload
 19:     Return 3
 20:   EndFunction
 21:   Function func(Val a Bool) Return Bool Overload
 22:     Return 4
 23:   EndFunction
 24: EndProgram

Type Checking:
L. 10: Identifier proc_ii already declared.
L. 18: Identifier func_b already declared.
L. 21: Identifier func_b already declared.
L. 22: Return with incompatible type.
```

Ahora debéis aplicar el cambio anterior para verificar las llamadas a funciones `overload`.

Dada una llamada, deberéis seguir las normas de decoración para decorar el nombre de la función/procedimiento llamada en base a los parámetros reales. Tras haber hecho el `TypeCheck` de los parámetros reales, tendréis el AST decorado con el `ptype` de cada parámetro, que podréis usar como entrada a la función hecha en el apartado anterior.

Una vez obtenido el nombre decorado, será necesario buscarlo en la tabla de símbolos. Si existe, la búsqueda ha terminado, y se hará la comprobación de la coincidencia de los parámetros reales con los formales del `ptype` obtenido. Si no existe, y para poder seguir funcionando con funciones y procedimientos que no tuvieran el modificador `overload` en la declaración, será necesario buscar

el identificador no decorado como hasta ahora. El mensaje de error en el caso que ambos identificadores no existan será el habitual.

Como recordatorio, `symboltable.find(ident)` busca un identificador en la tabla de símbolos, devolviendo cierto si lo ha encontrado; en cuyo caso, `symboltable[ident].tp` devuelve el `ptype` asociado.

(1 punto) Y entonces podréis pasar el quinto juego de pruebas.

```
 1: Program
 2:   Vars
 3:     i Int
 4:     j Bool
 5:   EndVars
 6:   Procedure proc(Val a Int) Overload
 7:     proc(a, False)
 8:     proc(a, 3)
 9:   EndProcedure
10:   Procedure proc(Val a Int, Val b Bool) Overload
11:     proc(4)
12:     proc(i, True)
13:   EndProcedure
14:
15:   Function func(Val a Int, Val b Int, Val c Int) Return Int Overload
16:     Return a + b + c
17:   EndFunction
18:   Function func(Val a Int, Val b Int) Return Bool Overload
19:     Return a > b
20:   EndFunction
21:   Function func(Val a Int, Val b Int) Return Int Overload
22:     Return a + b
23:   EndFunction
24:   Function func(Val a Int, Val b Int, Val c Int, Val d Int) Return Bool Overload
25:     Return a + b + c + d > 100
26:   EndFunction
27:
28:   i := func(3, 4, 5)
29:   i := func(3, 4)
30:   j := func(3, 4)
31:   i := func(3, 4, 5, 6)
32:   j := func(3, 4, 5, 6)
33: EndProgram

Type Checking:
L. 8: Identifier proc is undeclared.
L. 21: Identifier func_ii already declared.
L. 29: Assignment with incompatible types.
L. 31: Assignment with incompatible types.
```

(0,5 puntos) El siguiente juego de pruebas verifica que hayáis extendido la función de decoración para tener en cuenta *arrays*. Para hacer la conversión de un entero a una *string*, podéis usar la función ya implementada `itostring`.

```
 1: Program
 2:   Vars
 3:     ai1 Array [5] Of Int
 4:     ai2 Array [7] Of Int
 5:     ab1 Array [5] Of Bool
 6:     i Int
 7:     j Bool
 8:   EndVars
 9:
```

```
 10:   Function f(Val a Array[5] Of Int) Return Int Overload
 11:     Return a[0]
 12:   EndFunction
 13:   Function f(Val a Array[7] Of Int) Return Bool Overload
 14:     Return a[0] > a[5]
 15:   EndFunction
 16:   Function f(Val a Array[5] Of Bool) Return Bool Overload
 17:     Return False
 18:   EndFunction
 19:
 20:   Function f(Val a Array[5] Of Int, Val b Array[5] Of Int)
 21:             Return Array[5] Of Int Overload
 22:     Return a
 23:   EndFunction
 24:   Function f(Val a Array[5] Of Int, Val b Array[7] Of Int)
 25:             Return Array[7] Of Int Overload
 26:     Return b
 27:   EndFunction
 28:   Function f(Val b Array[5] Of Int, Val z Array[7] Of Int)
 29:             Return Array[5] Of Int Overload
 30:     Return b
 31:   EndFunction
 32:
 33:   i := f(ai1)
 34:   i := f(ai2)
 35:   i := f(ab1)
 36:   j := f(ai1)
 37:   j := f(ai2)
 38:   j := f(ab1)
 39:
 40:   ai1 := f(ai1, ai1)
 41:   ai1 := f(ai1, ai2)
 42:   ai2 := f(ai1, ai1)
 43:   ai2 := f(ai1, ai2)
 44: EndProgram

Type Checking:
L. 28: Identifier f_a5ia7i already declared.
L. 34: Assignment with incompatible types.
L. 35: Assignment with incompatible types.
L. 36: Assignment with incompatible types.
L. 41: Assignment with incompatible types.
L. 42: Assignment with incompatible types.
```

(0,5 puntos) Y *structs*. Podéis observar cómo recorrer los campos de un struct en la función `equivalent_types` dentro de *ptype.cc*.

```
  1: Program
  2:   Vars
  3:     s1 Struct
  4:       a Int  s Struct a Array[2] Of Int EndStruct
  5:     EndStruct
  6:     s2 Struct
  7:       s Struct a Array[2] Of Int EndStruct  a Int
  8:     EndStruct
  9:     i Int
 10:     j Bool
 11:   EndVars
 12:
 13:   Procedure p(Val a Int, Val b Bool) Overload
 14:   EndProcedure
 15:   Procedure p(Val a Struct a Int b Bool EndStruct) Overload
```

```
 16:   EndProcedure
 17:   Procedure p(Val a Struct c Int d Bool EndStruct) Overload
 18:   EndProcedure
 19:
 20:   Function f(Val a Struct
 21:                   a Int s Struct a Array[2] Of Int EndStruct
 22:                 EndStruct,
 23:             Val b Struct a Array[2] Of Int EndStruct)
 24:           Return Int Overload
 25:     Return 3
 26:   EndFunction
 27:   Function f(Val a Struct
 28:                   a Int s Struct a Array[2] Of Int EndStruct
 29:                 EndStruct,
 30:             Val b Struct a Array[2] Of Int EndStruct)
 31:            Return Int Overload
 32:     Return 30
 33:   EndFunction
 34:   Function f(Val a Struct
 35:                   s Struct a Array[2] Of Int EndStruct a Int
 36:                 EndStruct,
 37:             Val b Struct a Array[2] Of Int EndStruct)
 38:            Return Bool Overload
 39:     Return False
 40:   EndFunction
 41:
 42:   i := f(s1, s1.s)
 43:   i := f(s2, s2.s)
 44:   j := f(s1, s1.s)
 45:   j := f(s2, s2.s)
 46: EndProgram
```

```
Type Checking:
L. 27: Identifier f_s<a,i><s,s<a,a2i>e>es<a,a2i>e already declared.
L. 43: Assignment with incompatible types.
L. 44: Assignment with incompatible types.
```

Autoevaluación: Basta con ejecutar `./fesentrega.sh` para crear automáticamente un fichero llamado `entrega.tar`.

Si ejecutáis entonces `./checker.sh`, al cabo de un rato os indicará qué juegos de pruebas habéis pasado junto con la correspondiente nota final. La evaluación definitiva se hará con ligeras modificaciones de dichos juegos de pruebas. De hecho, el `checker` es una ayuda bastante fidedigna para comprobar que vuestras modificaciones funcionan, pero queda bajo vuestra responsabilidad el comprobar que emitís los errores esperados y que no dais errores absurdos de más.

Entrega: Tenéis dos opciones. Podéis ejecutar `perl entregador.pl`, que realiza la entrega del fichero `entrega.tar` del mismo directorio. También podéis conectaros en prácticas vía web a `examens.fib.upc.edu`. Debéis entregar el fichero `entrega.tar` creado tal y como se indica en la autoevaluación.