Resultados del Examen de Prácticas de Procesadores de Lenguajes

(Convocatoria Ordinaria) celebrado el 30 de Junio de 2006.

1. Dado el lenguaje asignado para la realización de las prácticas, se va a añadir funcionalidad en dos aspectos:

(a) ***División entera*: Consiste en la división de dos expresiones de tipo entero por medio del operador binario // devolviendo un valor entero (ejemplo: $a - b//c + d$, donde todas las variables son de tipo entero). La prioridad del operador división entera es igual que la división.**

Solución:

Análisis Léxico

Dado que posee la misma misión sintáctica y precedencia que la división, se agrupa con dicho operador. Sea `OP_MULDIV` el token que identifica aquellos lexemas que son operadores binarios exclusivamente y que poseen la misma precedencia (normalmente se agrupa con el de multiplicación y otros que tengan asignada la misma precedencia).

Según lo dicho, sería necesario añadir las siguientes líneas al fuente LEX

```
...
%%
.....

 "/"    { yylval.atrib= ATR_DIV_FLO ; return OP_MULDIV ; }
 "//"   { yylval.atrib= ATR_DIV_ENT ; return OP_MULDIV ; }

.....
%%
...
```

donde `ATR_DIV_FLO` y `ATR_DIV_ENT` son dos constantes enteras de distinto valor que sirven para dar valor al atributo del token `OP_MULDIV` e `yylval` es la variable de LEX/YACC de tipo `YYSTYPE`, definida con la misma estructura que la práctica de análisis semántico. En dicha variable existe un campo denominado "`atrib`"que es de tipo entero que almacena valores referentes al atributo del token (de cara a distinguir los lexemas en el análisis semántico).

Análisis Sintáctico

Para contemplar expresiones de división entera ya existe la regla que así lo expresa en sintaxis Yacc:

```
...
%%
...
Expresion : ...
          | Expresion OP_MULDIV Expresion
          | ...
;
...
%%
...
```

Por lo tanto, no habría que añadir ningún tipo de regla sintáctica para contemplar este tipo de expresiones.

Análisis Semántico

Dado que ya existe la regla sintáctica que puede representar expresiones binarias, incluyendo las de división entera, habría que añadir la comprobación semántica de que ambas expresiones de los operandos son de tipo entero. En sintaxis de Yacc quedaría así:

```
...
%%
...
Expresion : ...
          | Expresion OP_MULDIV Expresion
              {
                 ....
                 if ($2.atrib == ATR_DIV_ENT)     /* Lexema "//" */
                 {
                    if ($1.tipo==TIPO_ENTERO && $3.tipo==TIPO_ENTERO)
                       $$.tipo=TIPO_ENTERO ;
                    else
                    {
                       printf ("Error semántico:
                                Tipos incompatibles en división entera\n");
                       $$.tipo= incorrecto ;
                    }
                 }
                 ....
              }
          | ...
;
...
%%
...
```

aquí, `TIPO_ENTERO` es una constante entera que codifica el tipo entero.

(b) ***Operador condicional ternario*: Consiste en la evaluación de una expresión de tipo lógico y dependiendo del resultado devolverá una expresión u otra (ejemplo: $a < b?c + 10 : d - 10$, donde las expresiones $c + 10$ y $d - 10$ deben ser del mismo tipo básico y $a < b$ es la expresión de tipo lógico). La prioridad del operador ternario condicional es la mínima del resto de operadores.**

Solución:

Análisis Léxico

Son necesarios dos nuevos tokens `OPINTERR` (?) y `DOSPUNTOS` (:) (los alumnos que tengan como lenguaje base Pascal ya poseen este último token). El papel sintáctico de estos tokens es diferente del papel de los tokens correspondientes a cualquier otro operador. Por tanto, los nuevos tokens no se pueden agrupar con los ya existentes para otros operadores.

La inclusión de estos nuevos tokens puede mostrarse en la siguiente sintaxis Lex:

```
...
%%
 ....
 "?"   { return OPINTERR ; }
 ":"   { return DOSPUNTOS ; }
 ....
%%
...
```

Análisis Sintáctico

Habría que añadir una nueva regla para el nuevo tipo de expresiones. La declaración de los token debe hacerse de forma que nos aseguremos que se asigna a este operador menor precedencia que a los demás, así que deberíamos de escribirlos antes que la declaración de los tokens correspondientes a otros operadores de expresiones. En YACC, junto a la nueva regla sintáctica que surge, se escribiría así:

```
%left OPINTERR
%left DOSPUNTOS
// resto de tokens de operadores de expresiones
........
%%
...
Expresion : ....
          | Expresion INTERR Expresion DOSPUNTOS Expresion
          | ....
;
...
%%
...
```

Análisis Semántico

A partir de la nueva regla gramatical introducida, debemos añadir aquellas comprobaciones semánticas necesarias, es decir, que la primera expresión sea de tipo lógico y que las otras dos sean del mismo tipo. Si todo es correcto, se sintetiza el tipo de cualquiera de las dos últimas expresiones.

En sintaxis de Yacc quedaría así:

```
...
%%
...
Expresion : ...
          | Expresion INTERR Expresion DOSPUNTOS Expresion
             {
                 if ($1.tipo==TIPO_LOGICO)
                 {
                    if ($3.tipo==$5.tipo && EsTipoBasico($3.tipo) )
                       $$.tipo= $3.tipo ;
                    else
                    {
                       $$.tipo= incorrecto ;
                       printf ("Error semántico en el operador condicional:
                                tipos diferentes o no básicos tras '?'\n");
                    }
                 }
                 else
                 {
                    $$.tipo= incorrecto ;
                    printf ("Error semántico en el operador condicional:
                             se esperaba una expresión lógica antes
                             de '?'\n");
                 }
             }
          | ...
;
...
%%
...
```

Donde:

- `TIPO_LOGICO` es una constante entera que codifica el tipo lógico.
- `EsTipoBasico(tipo)` es una función lógica que devuelve `true` si el tipo (que es un entero que se pasa como parámetro) es un entero, flotante o carácter, y `false` en caso contrario (arrays, listas, etc.) (no es necesario implementar esta función).