CC41A - Control 1
Tiempo: 2 horas
Con Apuntes

José M. Piquer
Sem: 00/1

Pregunta 1 (Compiladores)

Queremos escribir un evaluador de expresiones, agregándole la capacidad de tener expresiones condicionales, con la gramática:

```
Inst -> Expr;
      | if Expr { Inst* }
```

Donde `Expr` son las expresiones habituales de la tarea 2, y el `if` no lleva `else`.

a.-

Escriba un programa para flex que detecte los tokens necesarios.

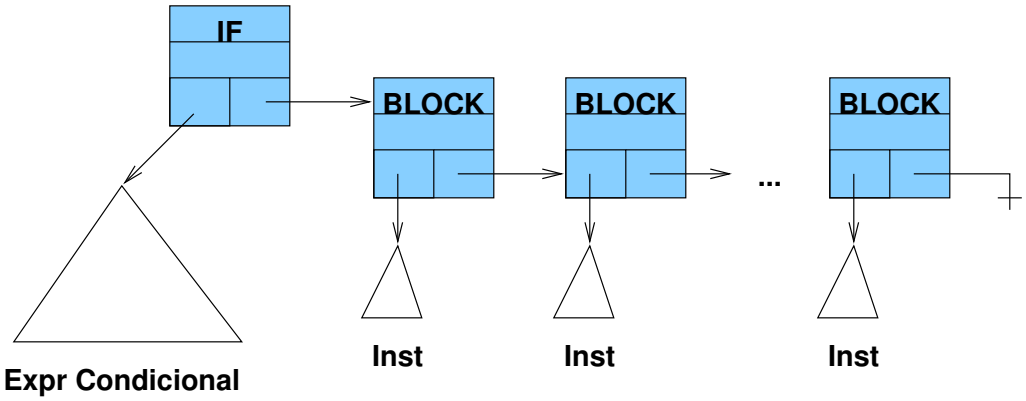
Lo importante es poner el if antes de los identificadores, puesto que no debe confundirse con uno.

```
"if" {return IF;}
"{"  {return ABRE_CRESPO;}
"}"  {return CIERRE_CRESPO;}
[a-z][0-9a-z]*   {return ID;}
```

b.-

Dibuje el árbol de la expresión `if` y luego escriba un programa en descenso recursivo que lo construya. Suponga que `Expr` ya existe y funciona, usted solo escribe `Inst()`.

Para el if, usaremos un árbol con hijo izquierdo la expresión e hijo derecho un operador nuevo que llamaremos BLOCK. Ese operador tiene como hijo izquierdo una instrucción y derecho una operación BLOCK con el resto del bloque de instrucciones. La estructura es:



Para hacer más simple el código, supongamos una función `maketree` *que recibe el operador, el árbol izquierdo y el árbol derecho, pide memoria para un nodo e inicializa sus campos con esos valores.*

No se consideran los casos de error.

```
TREE *Inst()
{
    TREE *tree, *p;

    if(Nexttok() == IF) {
       Gettok();
       tree = maketree(IF, Expr(), NULL);
       Gettok();   /* { */
       p = tree;
       while(Nexttok() != CIERRE_CRESPO) {
           p->right = maketree(BLOCK, Instr(), NULL);
           p = p->right;
       }
       Gettok();   /* } */
    }

    return tree;
}
```

c.-

¿Porqué cree usted que los analizadores sintácticos típicamente generan un árbol en vez de evaluar la expresión inmediatamente?

La principal razón es cuando se requiere guardar el código para futuras ejecuciones, por ejemplo código de funciones o ciclos, en cuyo caso no queremos volver a analizar sintácticamente el código original.

Pregunta 2 (Evaluadores)

Escriba un programa en C (o Java) que evalúe el árbol de la pregunta anterior. Suponga que solo hay constantes enteras y la condición del if se interpreta como: 0 es false y el resto es true.

```
int Eval(TREE *tree)
{
    int ret;

    if(tree == NULL)
        return 0;

    switch(tree->op)
      {
    case INT:
        return tree->val;
    break;
    case PLUS:
        return Eval(tree->left) + Eval(tree->right);
    break;
    ... etc ...
    case IF:
        if( Eval(tree->left) != 0 ) return Eval(tree->right);
        else return 0;
    break;
    case BLOCK:
        ret = Eval(tree->left);
        if( tree->right == NULL ) return ret;
        else return Eval(tree->right);
    break;
      }
}
```

Pregunta 3 (Scheme)

a.-

Escriba una función que "suma" una lista en Scheme. Recibe de parámetro una lista con sublistas y retorna el resultado de sumar todos los átomos. Por ejemplo:

```
(suma '(1 2 (3 4 (5)) ()))
-> 15

(define (suma l)
```

```
(if (null? l)
    0
    (if (pair? l)
        (+ (suma (car l)) (suma (cdr l)))
        l )))
```

b.-

¿Porqué en Scheme la instrucción `if` no puede ser simplemente una función scheme?

El problema es si alguna de las expresiones del código condicional ocasiona efectos de borde (modificación de variables, entrada/salida, etc). Por ejemplo:

```
(if (< a b) (print a) (print b))
```

Imprimiría ambos valores si fuese una función.