PARSEO Y GENERACIÓN DE CÓDIGO – 2DO SEMESTRE 2016
LICENCIATURA EN INFORMÁTICA CON ORIENTACIÓN EN DESARROLLO DE SOFTWARE
UNIVERSIDAD NACIONAL DE QUILMES

Trabajo práctico 1

Analizador sintáctico y semántico para Cucaracha

Fecha de entrega: 21 de septiembre

Índice

1. Introducción

Este trabajo consiste en programar el *front-end* para un compilador del lenguaje Cucaracha. El trabajo consta de tres partes:

1. En primer lugar, se debe programar un parser. Dado el código fuente de un programa escrito en Cucaracha, el analizador sintáctico debe dar como resultado un árbol de sintaxis abstracta (AST). El parser se puede implementar manualmente (por ejemplo, utilizando la técnica de análisis sintáctico por descenso recursivo) o utilizando cualquier generador de parsers disponible para el lenguaje de implementación elegido (por ejemplo `bison` para C, `ply` para Python, `happy` para Haskell, `ANTLR` para Java, etc.).

2. En segundo lugar, se debe implementar funcionalidad para serializar el AST convirtiéndolo a una representación textual. Esto servirá para validar que el parser construido en la primera etapa es correcto. Esta etapa se puede implementar mediante un simple recorrido *preorder* sobre el AST.

3. Por último, se debe programar un analizador semántico para Cucaracha. En esta etapa se deben identificar errores semánticos. Nos interesan esencialmente los errores de tipos. La implementación del analizador se puede hacer recorriendo el AST para validar que el programa esté correctamente formado. Por ejemplo, la condición de un `if` debe ser una expresión de tipo booleano.

2. Descripción informal del lenguaje Cucaracha

El siguiente programa en Cucaracha imprime "HOLA" en la salida:

```
fun main() {
  putChar(72)
  putChar(79)
  putChar(76)
  putChar(65)
}
```

Un programa en Cucaracha es una lista de declaraciones de funciones. Las funciones en Cucaracha pueden no devolver nada (procedimientos a los que se invoca exclusivamente por su efecto) o devolver un valor. Las funciones que no devuelven nada se declaran de la siguiente manera:

```
fun nombre(parámetros) bloque
```

Las funciones que devuelven algo deben venir acompañadas del tipo del valor que devuelven. En ese caso la sintaxis es:

```
fun nombre(parámetros) : tipo_de_retorno bloque
```

Los nombres de todos los identificadores (nombres de funciones, variables y parámetros) están dados por una secuencia de caracteres alfanuméricos (`a..z`, `A..Z`, `0..9`) y el guión bajo (`_`), pero no pueden empezar con un número.

2.1. Instrucciones

El cuerpo de una función en Cucaracha es un bloque. Un bloque es una lista de instrucciones delimitadas por llaves.

Las instrucciones en Cucaracha son las siguientes. Cada una se acompaña del nombre que tendrá asociado en el árbol de sintaxis (`EscritoEnCamelCase`) y de un ejemplo.

- **Asignación** (`StmtAssign`).
  `variable := expresión`

  ```
  fun main() {
    x := 65
    putChar(x)
  }
  ```

- **Asignación a un vector** (`StmtVecAssign`).
  `variable[expresión] := expresión`

  ```
  fun main() {
    x := [1,2,3]
    x[0] := 65
    putChar(x[0])
  }
  ```

- **Condicional con o sin else** (`StmtIf`, `StmtIfElse`).
  `if expresión bloque`
  `if expresión bloque else bloque`

  ```
  fun main() {
    if 1 > 0 {
      putChar(65)
    }
  }
  ```

```
fun main() {
  if 1 > 0 {
    putChar(65)
  } else {
    putChar(66)
  }
}
```

- **While** (StmtWhile).
  `while expresión bloque`

```
fun main() {
  x := 10
  while x > 0 {
    x := x - 1
    putChar(65)
  }
}
```

- **Return** (StmtReturn).
  `return expresión`

```
fun letraA() : Int {
  return 65
}

fun main() {
  putChar(letraA())
}
```

- **Invocación a una función** (StmtCall).
  `función(argumentos)`

```
fun decirHola() {
  putChar(72)
  putChar(79)
  putChar(76)
  putChar(65)
}

fun main() {
  decirHola()
}
```

2.2. Tipos de datos

El lenguaje Cucaracha maneja tres tipos de datos: *números*, *valores lógicos* y *vectores de números*. Las funciones pueden recibir como parámetros números, valores lógicos y vectores de números, y pueden devolver números y valores lógicos, pero **no pueden devolver vectores**[1]. En las declaraciones de tipos, los tipos se escriben de la siguiente manera:

- `Int` – el tipo de los números.
- `Bool` – el tipo de los valores lógicos.
- `Vec` – el tipo de los vectores de números.

[1] Es difícil apreciar la importancia de este hecho en el TP1, pero va a ser de suma importancia para el TP2.

La lista de parámetros formales en la declaración de una función es una lista de nombres de parámetros separados por comas. Cada parámetro viene acompañado de su tipo. Por ejemplo:

```
fun f(numero : Int, vector : Vec) : Bool {
  return numero > vector[0]
}
```

2.3. Expresiones

- **Uso de una variable** (ExprVar). Simplemente un identificador (p.ej. `x`).
- **Constantes numéricas** (ExprConstNum). Se escriben siempre en notación decimal como una secuencia de dígitos (p.ej. `54321`).
- **Constantes lógicas** (ExprConstBool). Se escriben `True` y `False`.
- **Constructor de vectores** (ExprVecMake). Los vectores se construyen con una lista de expresiones separadas por comas y delimitadas por corchetes.
  `[expresión, expresión, ..., expresión]`

  ```
  fun main() {
      x := [64 + 1, 65, 66]
      putChar(x[0])
  }
  ```

- **Tamaño de un vector** (ExprVecLength). Un cardinal seguido de un identificador (que puede ser una variable o parámetro).
  `#identificador`

  ```
  fun mostrarTodo(v : Vec) {
      i := 0
      while i < #v {
          putChar(v[i])
          i := i + 1
      }
  }
  ```

- **Acceso al i-ésimo elemento de un vector** (ExprVecDeref). Un identificador seguido de una expresión entre corchetes.
  `identificador[expresión]`

  ```
  fun enesimo(v : Vec, n : Int) : Int {
      return v[n]
  }
  ```

- **Invocación a una función (ExprCall)**
  `función(argumentos)`
- **Operadores lógicos** Son los siguientes:
  - `and` (ExprAnd): operador binario.
  - `or` (ExprOr): operador binario.
  - `not` (ExprNot): operador unario.
- **Operadores relacionales** Son los siguientes:
  - `<=` (ExprLe): operador binario.
  - `>=` (ExprGe): operador binario.
  - `<` (ExprLt): operador binario.

- `>` (`ExprGt`): operador binario.
- `==` (`ExprEq`): operador binario.
- `!=` (`ExprNe`): operador binario.

- **Operadores aritméticos** Son los siguientes:
  - `+` (`ExprAdd`): operador binario.
  - `-` (`ExprSub`): operador binario.
  - `*` (`ExprMul`): operador binario.

  Nota: el lenguaje Cucaracha no cuenta con el operador "menos" unario (p.ej. `-2` no es una expresión válida).

2.4. Precedencia y asociatividad de los operadores

Todos los operadores lógicos y aritméticos asocian a izquierda. Los operadores relacionales no se pueden asociar (p.ej. $a == b == c$ es una expresión inválida). Hay cuatro niveles de precedencia, de menor a mayor:

- **Operadores lógicos binarios:** `and or`
- **Negación lógica:** `not`
- **Todos los operadores relacionales:** `<= >= < > == !=`
- **Operadores aritméticos aditivos:** `+ -`
- **Multiplicación:** `*`

Se pueden utilizar paréntesis para forzar la asociatividad deseada. Por ejemplo:

```
not 5 < 2 + 3 * 2 and 2 * (3 + 3) == 12
```

asocia así:

```
(not (5 < (2 + (3 * 2)))) and ((2 * (3 + 3)) == 12)
```

2.5. Comentarios

Se pueden incluir comentarios en el programa. Los comentarios comienzan con `//` y se extienden hasta el final de la línea. Por ejemplo:

```
fun main() {
    resolverMisterioDelUniverso()
}

fun resolverMisterioDelUniverso() {
    // No implementado.
}
```

3. Serialización del AST

El analizador sintáctico debe construir un AST. Para validar que esté construido correctamente, se pide contar con funcionalidad para serializar el AST, escribiéndolo como un texto. Para serializar un nodo del AST, se debe imprimir, en el siguiente orden:

- Un paréntesis izquierdo "(".
- El nombre del nodo, por ejemplo `StmtAssign`, tal como figura en el Apéndice B.
- Un caracter de nueva línea (es decir un "enter", el caracter `'\n'`).

- Todos y cada uno de los hijos del nodo, identados con dos espacios más que el padre, recursivamente, en el orden en el que figuran en el Apéndice B.
- Un paréntesis derecho ")".
- Un caracter de nueva línea.

Los identificadores se serializan escribiendo su nombre. Los números se serializan escribiendo sus dígitos decimales. Las constantes booleanas se serializan escribiendo `True` y `False`. Los tipos se serializan escribiendo `Int`, `Bool`, `Vec` y `Unit`.

Ejemplo. El siguiente programa:

```
fun main() {
  x := 65
  putChar(x)
}
```

se serializa así:

```
(Program
  (Function
    main
    Unit
    (Block
      (StmtAssign
        x
        (ExprConstNum
          65
        )
      )
      (StmtCall
        putChar
        (ExprVar
          x
        )
      )
    )
  )
)
```

4. Análisis semántico y chequeo de tipos

Una vez construido el AST, se debe hacer un recorrido por el árbol verificando que el programa esté bien formado desde el punto de vista semántico. Para ello se sugiere proceder en el orden siguiente:

1. Visitar todas las definiciones de funciones y armar una **tabla de funciones** que dado el nombre de una función indique los tipos de sus parámetros y su tipo de retorno. En este recorrido es debe:

   a) Verificar que no haya dos funciones con el mismo nombre.

   b) Verificar que el tipo de retorno de las funciones nunca sea `Vec`.

   c) Verificar que haya una función `main` sin parámetros y con tipo de retorno `Unit`.

   La tabla de funciones se debe inicializar con dos funciones que provee el lenguaje para imprimir texto por la salida, `putChar` y `putNum`. Ambas reciben un `Int` y devuelven `Unit`.

2. Visitar el cuerpo de todas y cada una de las funciones. El alcance de las variables es local a cada función. Al momento de visitar el cuerpo de cada función, se sugiere crear una **tabla de nombres locales** que a cada nombre de variable o parámetro le asocia su tipo.

3. En un momento dado de este recorrido por el programa, podemos decir que una expresión está bien formada de acuerdo con la siguiente definición inductiva:
   - Una variable `x` está bien formada si está registrada en la tabla de nombres locales y tiene el tipo indicado en la tabla.
   - Una constante numérica o lógica siempre está bien formada y tiene tipo `Int` o `Bool` respectivamente.
   - La construcción de un vector `[`e_1`, ..., `e_n`]` está bien formada y tiene tipo `Vec` si cada expresión e_i tiene tipo `Int`.
   - La longitud de un vector `#x` tiene tipo `Int` si `x` es un nombre local de tipo `Vec`.
   - El acceso a un vector e_1`[`e_2`]` está bien formado si e_1 es una expresión de tipo `Vec` y e_2 es una expresión de tipo `Int`.
   - Una expresión formada a partir de una invocación a función `f(`e_1`, ..., `e_n`)` está bien formada si `f` es una función definida en la tabla de funciones que tiene n parámetros, cada expresión e_i está bien formada y su tipo coincide con el del i-ésimo parámetro de f. Además, f debe devolver `Int` o `Bool` (no debe devolver `Unit`).
   - Los operadores tienen los tipos esperados:
     - `and`, `or`, `not`: reciben booleanos y devuelven booleanos.
     - `<=`, `>=`, `<`, `>`, `==`, `!=`: reciben números y devuelven booleanos.
     - `+`, `-`, `*`: reciben números y devuelven números.
4. El cuerpo de cada función es una lista de instrucciones. Para que la función esté bien formada, cada instrucción debe estar bien formada.
   - Una asignación `x := e` está bien formada si `e` es una expresión bien formada. En este caso, si `x` aún no figura en la tabla de nombres locales, agregar `x` a la tabla de nombres locales con el tipo correspondiente. Si `x` ya figuraba en la tabla de nombres locales, verificar que el tipo que tenía coincida con el de `e`.
   - Una asignación a un vector `x[`e_1`] := `e_2 está bien formada si `x` es un nombre local de tipo `Vec`, y tanto e_1 como e_2 son expresiones bien formadas de tipo `Int`.
   - Un condicional con o sin else `if e bloque` / `if e bloque else bloque` está bien formado si la condición `e` es una expresión bien formada de tipo `Bool` y los bloques están bien formados.
   - Un ciclo `while e bloque` está bien formado si la condición `e` es una expresión bien formada de tipo `Bool` y el bloque está bien formado.
   - La instrucción `return e` está bien formada si `e` es una expresión bien formada y su tipo coincide con el tipo de retorno de la función. Además, en Cucaracha el `return` solamente puede aparecer como la última instrucción en el cuerpo de una función.
   - Una instrucción formada a partir de una invocación a función `f(`e_1`, ..., `e_n`)` está bien formada si `f` es una función definida en la tabla de funciones que tiene n parámetros, cada expresión e_i está bien formada y su tipo coincide con el del i-ésimo parámetro de f. Además, f debe devolver `Unit` (no debe devolver `Int` ni `Bool`).
5. Por último, si una función tiene tipo de retorno `Int` o `Bool`, se debe verificar que la última instrucción sea un `return`, mientras que si el tipo de retorno es `Unit` se debe verificar que **no** tenga un `return`.

El analizador semántico debe aceptar los programas que cumplan estas verificaciones y rechazar los que no las cumplan. En caso de rechazarlos, se debe proveer un mensaje de error indicando su causa lo mejor que se pueda.

Nota: el siguiente programa no se rechaza:

```
fun main() {
  if (False) {
    x := 1
  }
  putChar(x)
}
```

porque no lo consideramos un error de tipos. Es un error de naturaleza dinámica (como dividir por 0, acceder a un vector fuera de rango o escribir un while que se cuelga). Estos errores se pueden identificar potencialmente con técnicas más sofisticadas de análisis estático que veremos más adelante.

5. Pautas de entrega

Para entregar el TP se debe enviar el código fuente por e-mail a la casilla `foones@gmail.com` hasta las 23:59:59 del día estipulado para la entrega, incluyendo `[TP lds-est-parse]` en el asunto y el nombre de los integrantes del grupo en el cuerpo del e-mail. No es necesario hacer un informe sobre el TP, pero se espera que el código sea razonablemente legible. Se debe incluir un README indicando las dependencias y el mecanismo de ejecución recomendado para que el programa provea la funcionalidad pedida.

A. Gramática del lenguaje Cucaracha

A.1. Lista de símbolos terminales

Todos los símbolos terminales se acompañan de su nombre EN_MAYUSCULAS.

Identificadores y constantes

`[_a-zA-Z][_a-zA-Z0-9]*`	ID	Identificador (nombres de variables y funciones).
`[0-9]+`	NUM	Constante numérica.

- Se acepta que el parser se limite a identificadores de hasta 64 caracteres.
- Se acepta que el parser se limite a constantes numéricas hasta $2^{31} - 1$ (que es el máximo entero con signo que se puede escribir con 32 bits, es decir 2147483647).

Símbolos

`(`	LPAREN	Paréntesis izquierdo (para delimitar de parámetros y asociar).
`)`	RPAREN	Paréntesis derecho.
`,`	COMMA	Coma (para separar listas de parámetros y expresiones).
`[`	LBRACK	Corchete izquierdo (para construir y referenciar vectores).
`]`	RBRACK	Corchete derecho.
`{`	LBRACE	Llave izquierda (para abrir bloques).
`}`	RBRACE	Llave derecha.
`:=`	ASSIGN	Operador de asignación (para asignar a una variable o vector).
`:`	COLON	Dos puntos (para especificar tipos de parámetros y el tipo de retorno).
`#`	HASH	Cardinal (para obtener la longitud de un vector).
`<=`	LE	Operador relacional (menor o igual).
`>=`	GE	Operador relacional (mayor o igual).
`<`	LT	Operador relacional (menor estricto).
`>`	GT	Operador relacional (mayor estricto).
`==`	EQ	Operador relacional (igual).
`!=`	NE	Operador relacional (distinto).
`+`	PLUS	Operador aritmético (suma).
`-`	MINUS	Operador aritmético (resta).
`*`	TIMES	Operador aritmético (multiplicación).

Palabras clave

`Bool`	BOOL	Tipo de los valores lógicos.
`Int`	INT	Tipo de los valores numéricos.
`Vec`	VEC	Tipo de los vectores de números.
`True`	TRUE	Constante lógica: verdadero.
`False`	FALSE	Constante lógica: falso.
`and`	AND	Operador lógico (“y”).
`else`	ELSE	Else.
`fun`	FUN	Encabeza las definiciones de funciones.
`if`	IF	If.
`not`	NOT	Operador lógico (“no”).
`or`	OR	Operador lógico (“o”).
`return`	RETURN	Return.
`while`	WHILE	While.

A.2. Producciones de la gramática

Los símbolos no terminales se describen con su nombre ⟨*en_cursiva*⟩.

⟨*programa*⟩ ⟶ ϵ
| ⟨*declaración_de_función*⟩ ⟨*programa*⟩

⟨*declaración_de_función*⟩ ⟶ FUN ID ⟨*parámetros*⟩ ⟨*bloque*⟩ Procedimiento.
| FUN ID ⟨*parámetros*⟩ COLON ⟨*tipo*⟩ ⟨*bloque*⟩ Función que devuelve un valor.

⟨*parámetros*⟩ ⟶ LPAREN ⟨*lista_parámetros*⟩ RPAREN

⟨*lista_parámetros*⟩ ⟶ ϵ
| ⟨*lista_parámetros_no_vacía*⟩

⟨*lista_parámetros_no_vacía*⟩ ⟶ ⟨*parámetro*⟩
| ⟨*parámetro*⟩ COMMA ⟨*lista_parámetros_no_vacía*⟩

⟨*parámetro*⟩ ⟶ ID COLON ⟨*tipo*⟩

⟨*tipo*⟩ ⟶ INT Tipo de los números.
| BOOL Tipo de los valores lógicos.
| VEC Tipo de los vectores de números.

⟨*bloque*⟩ ⟶ LBRACE ⟨*lista_instrucciones*⟩ RBRACE Lista de instrucciones delimitadas por llaves.

⟨*lista_instrucciones*⟩ ⟶ ϵ
| ⟨*instrucción*⟩ ⟨*lista_instrucciones*⟩

⟨*instrucción*⟩ ⟶ ID ASSIGN ⟨*expresión*⟩ Asignación a una variable.
| ID LBRACK ⟨*expresión*⟩ RBRACK ASSIGN ⟨*expresión*⟩ Asignación a un vector.
| IF ⟨*expresión*⟩ ⟨*bloque*⟩ Condicional sin else.
| IF ⟨*expresión*⟩ ⟨*bloque*⟩ ELSE ⟨*bloque*⟩ Condicional con else.
| WHILE ⟨*expresión*⟩ ⟨*bloque*⟩ While.
| RETURN ⟨*expresión*⟩ Return.
| ID LPAREN ⟨*lista_expresiones*⟩ RPAREN Invocación a una función.

⟨*lista_expresiones*⟩ ⟶ ϵ
| ⟨*lista_expresiones_no_vacía*⟩

⟨*lista_expresiones_no_vacía*⟩ ⟶ ⟨*expresión*⟩
| ⟨*expresión*⟩ COMMA ⟨*lista_expresiones_no_vacía*⟩

⟨*expresión*⟩ ⟶ ⟨*expresión_lógica*⟩

⟨*expresión_lógica*⟩ ⟶ ⟨*expresión_lógica*⟩ AND ⟨*expresión_lógica_atómica*⟩ Conjunción lógica.
| ⟨*expresión_lógica*⟩ OR ⟨*expresión_lógica_atómica*⟩ Disyunción lógica.
| ⟨*expresión_lógica_atómica*⟩

⟨*expresión_lógica_atómica*⟩ ⟶ NOT ⟨*expresión_lógica_atómica*⟩ Negación lógica.
| ⟨*expresión_relacional*⟩

⟨*expresión_relacional*⟩ ⟶ ⟨*expresión_aditiva*⟩ LE ⟨*expresión_aditiva*⟩ Menor o igual.
| ⟨*expresión_aditiva*⟩ GE ⟨*expresión_aditiva*⟩ Mayor o igual.
| ⟨*expresión_aditiva*⟩ LT ⟨*expresión_aditiva*⟩ Menor estricto.
| ⟨*expresión_aditiva*⟩ GT ⟨*expresión_aditiva*⟩ Mayor estricto.
| ⟨*expresión_aditiva*⟩ EQ ⟨*expresión_aditiva*⟩ Igual.
| ⟨*expresión_aditiva*⟩ NE ⟨*expresión_aditiva*⟩ Distinto.
| ⟨*expresión_aditiva*⟩

⟨*expresión_aditiva*⟩ ⟶ ⟨*expresión_aditiva*⟩ PLUS ⟨*expresión_multiplicativa*⟩ Suma.
| ⟨*expresión_aditiva*⟩ MINUS ⟨*expresión_multiplicativa*⟩ Resta.
| ⟨*expresión_multiplicativa*⟩

⟨*expresión_multiplicativa*⟩ ⟶ ⟨*expresión_multiplicativa*⟩ TIMES ⟨*expresión_atómica*⟩ Multiplicación.
| ⟨*expresión_atómica*⟩

⟨*expresión_atómica*⟩ ⟶ ID Uso de una variable.
| NUM Constante numérica.
| TRUE La constante lógica verdadero.
| FALSE La constante lógica falso.
| LBRACK ⟨*lista_expresiones*⟩ RBRACK Construcción de un vector.
| HASH ID Longitud de un vector.
| ID LBRACK ⟨*expresión*⟩ RBRACK Acceso al i-ésimo de un vector.
| ID LPAREN ⟨*lista_expresiones*⟩ RPAREN Invocación a una función.
| LPAREN ⟨*expresión*⟩ RPAREN Uso de paréntesis para asociar.

B. Árbol de sintaxis abstracta para Cucaracha

Para referencia sobre cómo se construyen los árboles de sintaxis abstracta y cómo se llama cada uno de los nodos se utiliza notación estilo Haskell. El árbol puede estar implementado internamente de la manera que prefieran, pero se debe respetar el formato de serialización indicado en la Sección 3.

```
type Id    ::=  String
data Start   ::=  [Definition]
data Type    ::=  Int                                          Tipo de los números.
                | Bool                                         Tipo de los valores lógicos.
                | Vec                                          Tipo de los vectores de números.
                | Unit    Tipo de retorno de las funciones que no devuelven nada.

data ProgramT   ::=  Program [FunctionT]
data FunctionT   ::=  Function Id Type [ParameterT] BlockT
        Nota: las funciones que no devuelven nada se anotan con el tipo Unit.
data ParameterT   ::=  Parameter Id Type      Parámetro con su anotación de tipos.
data BlockT   ::=  Block [StmtT]
data StmtT   ::=  StmtAssign Id ExprT                          Asignación a una variable.
                | StmtVecAssign Id ExprT ExprT                 Asignación a un vector.
                | StmtIf ExprT BlockT                          Condicional sin else.
                | StmtIfElse ExprT BlockT BlockT               Condicional con else.
                | StmtWhile ExprT BlockT                       While.
                | StmtReturn ExprT                             Return.
                | StmtCall Id [ExprT]                          Invocación a una función.
data ExprT   ::=  ExprVar Id                                   Uso de una variable.
                | ExprConstNum Integer                         Constante numérica.
                | ExprConstBool Bool                           Constante booleana.
                | ExprVecMake [ExprT]                          Construcción de un vector.
                | ExprVecLength Id                             Longitud de un vector.
                | ExprVecDeref Id ExprT                        Acceso al i-ésimo de un vector.
                | ExprCall Id [ExprT]                          Invocación a una función.
                | ExprAnd ExprT ExprT                          Conjunción lógica.
                | ExprOr ExprT ExprT                           Disyunción lógica.
                | ExprNot ExprT                                Negación lógica.
                | ExprLe ExprT ExprT                           Menor o igual.
                | ExprGe ExprT ExprT                           Mayor o igual.
                | ExprLt ExprT ExprT                           Menor estricto.
                | ExprGt ExprT ExprT                           Mayor estricto.
                | ExprEq ExprT ExprT                           Igual.
                | ExprNe ExprT ExprT                           Distinto.
                | ExprAdd ExprT ExprT                          Suma.
                | ExprSub ExprT ExprT                          Resta.
                | ExprMul ExprT ExprT                          Multiplicación.
```