Diseño de Compiladores I - 2023
Trabajo Práctico Nº 2
Fecha de entrega: 11-10-2023

OBJETIVO

Construir un Parser (Analizador Sintáctico) que invoque al Analizador Léxico creado en el Trabajo Práctico Nº 1, y que reconozca un lenguaje con las siguientes características:

SINTAXIS GENERAL:

Programa:

- Programa constituido por un conjunto de sentencias, que pueden ser declarativas o ejecutables.
  Las sentencias declarativas pueden aparecer en cualquier lugar del código fuente, exceptuando los bloques de las sentencias de control.
  Los elementos declarados sólo serán visibles a partir de su declaración (esto será chequeado en etapas posteriores).
- El programa estará delimitado por llaves '**{**' y '**}**'.
- Cada sentencia debe terminar con coma "**,**".

Sentencias declarativas:

- Sentencias de declaración de datos para los tipos de datos correspondientes a cada grupo según la consigna del Trabajo Práctico 1, con la siguiente sintaxis:

  <tipo> <lista_de_variables>,

  Donde <tipo> puede ser (Según tipos correspondientes a cada grupo): **SHORT, INT, LONG, USHORT, UINT, ULONG, FLOAT, DOUBLE**
  Las variables de la lista se separan con punto y coma ("**;**")
- Incluir declaración de funciones **VOID**, con la siguiente sintaxis:

  VOID ID (<parametro>)

  {

  <cuerpo_ de_la_funcion>

  }

  Donde:
  - <parametro> será un identificador precedido por un tipo.:

    <tipo> ID

    Se permite hasta un parámetro, y puede no haber parámetros. **Este chequeo debe efectuarse durante el Análisis Sintáctico**
  - <cuerpo_de_la_funcion> es un conjunto de sentencias declarativas (incluyendo declaración de otras funciones) y/o ejecutables, incluyendo sentencias de retorno con la siguiente estructura:

    RETURN,

  Ejemplos válidos:

```
VOID f1 (INT y)
{
        INT x,
        x = y,
        …
        RETURN,
}
```

```
VOID f2 (LONG x)
{
        INT x,
        x = y,
        …
        RETURN,
}
```

```
VOID f3 ()
{
        FLOAT x,
        x = 1.2,
        …
        IF (x > 0.0)
                RETURN,
        ELSE
           {
                x = 2.0,
                RETURN,
                }
        END_IF,
}
```

Sentencias ejecutables:

- Asignaciones donde el lado izquierdo puede ser un identificador, y el lado derecho una expresión aritmética. Los operandos de las expresiones aritméticas pueden ser variables, constantes, u otras expresiones aritméticas.
  No se deben permitir anidamientos de expresiones con paréntesis.
- invocación a una función, con el siguiente formato:

  ID(<parametro_real>), // Función con parámetro

  o

  ID ( ), // Función sin parámetro

  El parámetro real puede ser cualquier expresión aritmética, variable o constante.
  Ejemplos:

```
f1(a),          f2(),          f3(a+b),          f4(5_i),
```

- Cláusula de selección (**IF**). Cada rama de la selección será un bloque de sentencias. La estructura de la selección será, entonces:

  IF (<condicion>) <bloque_de_sent_ejecutables> **ELSE** <bloque_de_sent_ejecutables> **END_IF,**

  El bloque para el **ELSE** puede estar ausente.
  La condición será una comparación entre expresiones aritméticas, variables o constantes, y debe escribirse entre **"(" ")"**.
  El bloque de sentencias ejecutables puede estar constituido por una sola sentencia, o un conjunto de sentencias ejecutables delimitadas por llaves.
- Sentencia de salida de mensajes por pantalla. El formato será

  PRINT<cadena>,

  Ejemplos:

```
PRINT#Hola mundo#,          //Tema 34
PRINT %Hola                 //Tema 35
      Mundo%,
```

TEMAS PARTICULARES

Nota: La semántica de cada tema particular, se explicará y resolverá en las etapas 3 y 4 del trabajo práctico

Temas 9 a 12 - Operadores

- **Tema 9**: **++**

  En los lugares donde un identificador puede utilizarse como operando (expresiones aritméticas o comparaciones), considerar el uso del operador '**++**' luego del identificador. Por ejemplo:

```
a = b++ * 7_i,
z = a+++b++,
IF (a++ > 2_l) …
```

- **Tema 10**: **--**

  En los lugares donde un identificador puede utilizarse como operando (expresiones aritméticas o comparaciones), considerar el uso del operador '**--**' luego del identificador. Por ejemplo:

```
a = b-- * 7_i,
z = a---b--,
IF (a-- > 2_l) …
```

- **Tema 11**: **+=**

  En las sentencias de asignación, puede utilizarse el operador '**+=**' en lugar del operador de asignación '**=**'. Por ejemplo:

```
a += 10_i,
z += a * b,
```

- **Tema 12**: **-=**

  En las sentencias de asignación, puede utilizarse el operador '**-=**' en lugar del operador de asignación '**=**'. Por ejemplo:

```
a -= 10_i,
```

```
z -= a * b,
```

Temas 13 a 16: Sentencias de Control

- **Tema 13**: ***WHILE DO***

  WHILE (<condicion>) **DO** <bloque_de_sentencias_ejecutables> ,

  <condición> tendrá la misma definición que la condición de las sentencias de selección**.**

  <bloque_de_sentencias_ejecutables> podrá contener una sentencia, o un grupo de sentencias ejecutables delimitadas por llaves.

- **Tema 14**: ***DO UNTIL***

  DO <bloque_de_sentencias_ejecutables> **UNTIL** (<condicion>),

  <condición> tendrá la misma definición que la condición de las sentencias de selección**.**

  <bloque_de_sentencias_ejecutables> podrá contener una sentencia, o un grupo de sentencias ejecutables delimitadas por llaves.

- **Tema 15**: ***DO WHILE***

  DO <bloque_de_sentencias_ejecutables> **WHILE** (<condicion>),

  <condición> tendrá la misma definición que la condición de las sentencias de selección**.**

  <bloque_de_sentencias_ejecutables> podrá contener una sentencia, o un grupo de sentencias ejecutables delimitadas por llaves.

- **Tema 16**: ***FOR***

  FOR i **IN RANGE** (m ; n ; j) < bloque_de_sentencias_ejecutables > ,
  i debe ser una variable.
  m, n y j serán constantes

  <bloque_de_sentencias_ejecutables> podrá contener una sentencia, o un grupo de sentencias ejecutables delimitadas por llaves.

  Ejemplos:

  ```
  FOR i IN RANGE (0;10;2) …
  FOR j IN RANGE (15;5;-1) …
  ```

  Nota: Las restricciones de tipo de la variable de control y los componentes del rango serán chequeadas en la etapa 3 del trabajo práctico.

Temas 17 a 24: Objetos

- **Tema 17**: ***Herencia por Composición - Uso anónimo***

Incorporar, como sentencia declarativa, la declaración de clases con la estructura que se muestra en los siguientes ejemplos:

```
CLASS ca {
   INT a,       // declaración de atributo
   VOID m() {  // declaración de método
     …
   },
}

CLASS cb{
   FLOAT b,     // declaración de atributo
   FLOAT a,     // declaración de atributo
   VOID n() {  // declaración de método
      …
   },
   ca,          // nombre de clase
}
…
```

Incorporar, como sentencia declarativa, la declaración de objetos de una clase determinada:

```
ca a1; a2,
cb b1; b2; b3,
```

Incorporar, dentro de las sentencias ejecutables, la posibilidad de utilizar referencias a métodos y atributos de objetos utilizando el '.' como se indica en los ejemplos:

```
a1.a = 3_i,
b1.b = 1.2,
b2.b = b1.a,
b1.m(),
b1.n(),
```

(todos los chequeos semánticos para estas referencias, se efectuarán en la etapa 3 del TP)

- **Tema 18**: ***Herencia por Composición - Uso con nombre***

Incorporar, como sentencia declarativa, la declaración de clases con la estructura que se muestra en los siguientes ejemplos:

```
CLASS ca {
   INT a;c,     // declaración de atributos
   VOID m() {  // declaración de método
     …
   },
}

CLASS cb{
   FLOAT b,     // declaración de atributo
   FLOAT a,     // declaración de atributo
   VOID n() {  // declaración de método
      …
   },
   ca,          // nombre de clase
}
…
```

Incorporar, como sentencia declarativa, la declaración de objetos de una clase determinada:

```
ca a1; a2,
cb b1; b2; b3,
```

Incorporar, dentro de las sentencias ejecutables, la posibilidad de utilizar referencias a métodos y atributos de objetos indicando explícitamente cuando el atributo o método corresponden a la clase heredada por composición.

Ejemplos:

```
a1.a = 3_i,
b1.ca.a = 3_i,
b1.a = 2.3,
b1.b = 1.2,
b1.ca.c = 1_i,
b1.ca.m(),
b1.n(),
```

(todos los chequeos semánticos para estas referencias, se efectuarán en la etapa 3 del TP)

- **Tema 19**: ***Declaración de métodos concentrada***

La semántica de este tema se explicará en la etapa 3 del Trabajo Práctico

- **Tema 20**: ***Declaración de métodos distribuida***

La implementación de los métodos de una clase puede efectuarse mediante una cláusula IMPL como se indica en el siguiente ejemplo

```
CLASS ca {
   INT a;c,     // declaración de atributos
   VOID m() {  // declaración de método
     …
   },
```

```
   VOID p(),    // prototipo de método
}

IMPL FOR ca: {
   VOID p() {  // implementación del método p de la clase ca
      ….
    },
   …
}
```

// **LÉXICO**: Incorporar a los símbolos detectados, el símbolo '**:**'.

- **Tema 21**: ***Forward declaration***

Incorporar la posibilidad de declarar las clases con posterioridad a su uso.

Por ejemplo:

```
CLASS ca {
   VOID m() {
      …
      }
   ...
   }
CLASS cc,
CLASS cd {
   cc c1,
   ca a1,
   VOID j() {
     c1.p(),   // será un error, pero se detectará en la siguiente etapa
     a1.m(),   // ok
   }
}
CLASS cc {     // declaración de clase cc
  INT z,
  VOID p() {
     …
     }
  …
}
```

- **Tema 22**: ***Interfaces***

Incorporar la declaración de interfaces, y su implementación.

Por ejemplo:

```
INTERFACE z {
   VOID q(...),
   VOID s(...),
  }

CLASS a IMPLEMENT z {
  VOID q(...) {
    …
    }
  VOID s(...) {
    …
    }
}
```

- **Tema 23**: ***Sobreescritura de métodos***
  La semántica de este tema se explicará en la etapa 3 del Trabajo Práctico

- **Tema 24**: ***Sobreescritura de atributos***
  La semántica de este tema se explicará en la etapa 3 del Trabajo Práctico

Temas 25 al 28 Comprobaciones de uso de variables

Se debe permitir la declaración de funciones locales dentro de funciones locales (método de clase), con un solo nivel de anidamiento. Esto será chequeado en la etapa 3 del TP.
Por ejemplo:

```
CLASS ca {
   INT a,
   VOID m() {
      INT xx,
```

```
        VOID funcion1() {
           FLOAT zz,
           FLOAT ww,
           ww = zz + xx + a,
           }
        …
        }
    }
```

- Tema 25 / 26 / 27 y 28:
  La semántica de estos temas se explicará en la etapa 3 del Trabajo Práctico

Temas 29 a 31: Conversiones

- Tema 29: **Conversiones Explícitas**: Se debe incorporar en todo lugar donde pueda aparecer una expresión, la posibilidad de utilizar la siguiente sintaxis:

  TOF (<expresión>) // para grupos que tienen asignado el tema 7

  TOD(<expresión>) // para grupos que tienen asignado el tema 8

// **LÉXICO**: Incorporar a la lista de palabras reservadas, la palabra **TOF** o **TOD** según corresponda.

- Tema 30: **Conversiones Implícitas**:
  Se explicará y resolverá en trabajos prácticos 3 y 4.

- Tema 31: **Sin conversiones:**
  Se explicará y resolverá en trabajos prácticos 3 y 4.

SALIDA DEL COMPILADOR

El programa deberá leer un código fuente escrito en el lenguaje descripto, y deberá generar como salida:

- Tokens detectados por el Analizador Léxico
- Estructuras sintácticas detectadas en el código fuente. Por ejemplo:
  Asignación
  Sentencia **WHILE**
  Sentencia **IF**
  etc.
  (Indicando nro. de línea para cada estructura)
- Errores léxicos y sintácticos presentes en el código fuente, indicando: nro. de línea y descripción del error. Por ejemplo:
  Línea 24: Constante de tipo **SHORT** fuera del rango permitido.
  Línea 43: Falta paréntesis de cierre para la condición de la sentencia **IF**.
- Contenidos de la Tabla de símbolos

CONSIDERACIONES GENERALES

a) Utilizar **YACC** u otra herramienta similar para construir el Parser.

b) Adaptar el Analizador Léxico del Trabajo Práctico 1 para convertirlo en el método o función ***int yylex()*** (o el nombre que el Parser generado requiera). Tener en cuenta que el léxico deberá devolver al parser, en cada invocación, un token. Para los identificadores, constantes y cadenas, deberá devolver además, la referencia a la entrada de la Tabla de Símbolos donde se ha registrado dicho símbolo, utilizando **yylval** para hacerlo.

c) Para aquellos tipos de datos que permitan valores negativos (**SHORT, INT, LONG, FLOAT, DOUBLE**) durante el Análisis Sintáctico se deberán detectar **constantes negativas**, modificando la tabla de símbolos según corresponda. Será necesario volver a controlar el rango de las constantes, ya que un valor aceptado para una constante por el Analizador Léxico, que desconoce su signo, podría estar fuera de rango si la constante es positiva.

- Ejemplo: Las constantes de tipo **INT** pueden tomar valores desde –32768 a 32767. El Léxico aceptará la constante 32768 como válida, pero si se trata de una constante positiva, estará fuera de rango.

d) Cuando se detecte un error, la compilación debe continuar.

e) Conflictos: Eliminar **TODOS LOS CONFLICTOS SHIFT-REDUCE Y REDUCE-REDUCE** que se presenten al generar el Parser.

FORMA DE ENTREGA

Se deberá presentar:

a) Código fuente completo y ejecutable, **incluyendo librerías del lenguaje** si fuera necesario para la ejecución
b) Informe
   - Contenidos indicados en el enunciado del Trabajo Práctico 1
   - Descripción del proceso de desarrollo del Analizador Sintáctico: problemas surgidos (y soluciones adoptadas) en el proceso de construcción de la gramática, manejo de errores, solución de conflictos shift-reduce y reduce-reduce, etc.
   - Lista de no terminales usados en la gramática con una breve descripción para cada uno.
   - Lista de errores léxicos y sintácticos considerados por el compilador.
   - Conclusiones.
c) Casos de prueba que contemplen **todas** las estructuras válidas del lenguaje. Incluir casos con errores sintácticos.