Diseño de Compiladores I - 2018
Trabajo Práctico Nº 2
Fecha de entrega: 27/09/2018

Objetivo

Construir un parser que invoque al Analizador Léxico creado en el Trabajo Práctico Nº 1, y que reconozca un lenguaje que incluya:

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.
- El programa no tendrá ningún delimitador.
- Cada sentencia debe terminar con " **,** ".

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>**, //o la estructura que corresponda al tema asignado (11 a 13)**
  Donde <tipo> puede ser (Según tipos correspondientes a cada grupo):
  integer
  usinteger
  linteger
  uslinteger
  single
  double
  Las variables de la lista se separan con punto y coma ("**;**")

Sentencias ejecutables:

- Cláusula de selección (**if**). Cada rama de la selección será un bloque de sentencias. La condición será una comparación entre expresiones aritméticas, variables o constantes, y debe escribirse entre **( )**. La estructura de la selección será, entonces:
  if (<condicion>**)** <bloque_de_sentencias> **else** <bloque_de_sentencias> **end_if**
  El bloque para el **else** puede estar ausente.
- Un bloque de sentencias puede estar constituido por una sola sentencia, o un conjunto de sentencias delimitadas por **'{'** y **'}'.**
- Sentencia de control según tipo especial asignado al grupo. (Temas 7 al 10 del Trabajo práctico 1)
  Debe permitirse anidamiento de sentencias de control. Por ejemplo, puede haber una iteración dentro de una rama de una selección.
- Sentencia de salida de mensajes por pantalla. El formato será **print**(cadena). Las cadenas de caracteres sólo podrán ser usadas en esta sentencia, y tendrán el formato asignado al grupo en el Trabajo Práctico 1.
- Los operandos de las expresiones aritméticas pueden ser variables, constantes, u otras expresiones aritméticas.
  No se permiten anidamientos de expresiones con paréntesis.

Temas particulares

Temas 7 a 10: Sentencias de Control

- ***while*** (tema 7 en TP1)
  while (<condicion> **)** <bloque_de_sentencias> **,**
- ***loop until*** (tema 8 en TP1)
  loop <bloque_de_sentencias> **until (** <condicion> **) ,**
- ***for*** (tema 9 en TP1)
  for (i **:=** n; <condición>; j) < bloque_de_sentencias > **,**
  i debe ser una variable de tipo entero (1-2-3-4).
  n y j serán constantes de tipo entero (1-2-3-4).
  <condición> será una comparación de i con m. Por ejemplo: i < m
  Donde m puede ser una variable, constante o expresión aritmética de tipo entero (1-2-3-4).
  Aclaración: Para los grupos que tengan asignados 2 tipos enteros, considerar el "más chico" (por ej. **integer**, para grupos con **integer** y **linteger**)
  Nota: Las restricciones de tipo serán chequeadas en la etapa 3 del trabajo práctico.
- ***case do*** (tema 10 en TP1)
  case (variable){
  valor1**: do** <bloque> **,**
  valor2**: do** <bloque> **,**
  ...
  valorN**: do** <bloque> **,**
  } ,
  Nota**:** Los valores valor1, valor2, etc., sólo podrán ser constantes del mismo tipo que la variable.
  La restricción de tipo será chequeada en la etapa 3 del trabajo práctico.

Temas 11 a 13

- ***Mutabilidad*** (Tema 11 del TP1)

  Sentencias declarativas:

  - Para este tema, las sentencias declarativas tendrán la siguiente estructura:
    let mut <tipo> <lista_de_variables>,
    donde la lista de variables, podrá incluir punteros, que se indicarán con un '*' precediendo al nombre.
    Por ejemplo:
    let mut integer a; b; *c; d; *e, // a, b y d son variables **integer**, mientras que c y e son punteros a **integer**.
    o
    let <tipo> <asignación>
    donde el valor asignado será un valor constante.
    Por ejemplo:
    let integer x := 5_i,

  Sentencias ejecutables:

  - Incorporar a la sintaxis del lenguaje, la posibilidad de usar, del lado derecho de una asignación la estructura &ID.
  - Incorporar a la sintaxis del lenguaje, la posibilidad de usar, del lado izquierdo de una asignación la estructura *ID.

    Es así que, dentro de las sentencias declarativas, se podrán encontrar declaraciones como la siguiente:
    let integer *p1 = &a,

  Nota: La semántica de estas sentencias, se explicará y resolverá en los trabajos prácticos 3 y 4.

- ***Closure*** (Tema 12 del TP1)

  Sentencias declarativas:

  - Incorporar, a la lista de tipos permitidos en una sentencia declarativa, el tipo **fun**
  - Incluir declaración de funciones, con la siguiente sintaxis:

    ```
    fun ID ( ) {
       <cuerpo_de_la_funcion>   // conjunto de sentencias declarativas y ejecutables
       return ( <retorno> )
     }
    ```

    Donde <retorno> será un identificador seguido de '(' ')' o el cuerpo de una función.

    Ejemplo:

    ```
    fun f1 ( ) {
        integer x,
        x := 5,
        void g ( ) {integer y, x := x + 1, print('suma'),}
        …
        return(g ( ) ),
    }
    ```

    o

    ```
    fun f2 ( ) {
        integer x,
        x := 5,
        …       …
        return({integer y, x := x + 1, print('suma'),} ),
    }
    ```

  Sentencias ejecutables:

  ~~- Incorporar la posibilidad de usar, del lado izquierdo de una asignación la estructura: **fun** ID.~~
  - Incorporar la posibilidad de usar, del lado derecho de una asignación la estructura: ID '(' ')'.
  - Incorporar como sentencia ejecutable, una invocación del tipo ID '(' ')'

  Nota: La semántica de estas sentencias, se explicará y resolverá en los trabajos prácticos 3 y 4.

- ***Borrowing*** (Tema 13 del TP1)
  - Eliminar la palabra **void** de la lista de palabras reservadas.
  - Incorporar, a la lista de palabras reservadas las palabras **readonly**, **write** y **pass**

  Sentencias declarativas:

  - Incluir declaración de funciones, con la siguiente sintaxis:

    ```
    <tipo> ID ( <tipo> ID) {
       <cuerpo_de_la_funcion>   // conjunto de sentencias declarativas y ejecutables
       return ( <retorno> )
     }
    ```

    Donde <retorno> será una expresión.

  Sentencias ejecutables:

  - Incorporar la invocación de funciones, con la siguiente sintaxis:
    ID(<nombre_parametro>;<lista_permisos>),
    Donde <lista_permisos> puede ser: **readonly / write / pass / write ; pass**
    La invocación puede aparecer en cualquier lugar donde pueda aparecer una expresión aritmética.

  Nota: La semántica de estas sentencias, se explicará y resolverá en los trabajos prácticos 3 y 4.

Temas 14 a 16: Conversiones

- Cada grupo debe incorporar un nuevo tema especial:

Grupo	Nuevo tema
1	14
2	16
3	16
4	15
5	16
6	14
7	16

Grupo	Nuevo tema
8	16
9	16
10	15
11	14
12	16
13	15
14	16

Grupo	Nuevo tema
15	16
16	16
17	16
18	14
19	16
20	16

La semántica de conversiones se explicará y resolverá en los trabajos prácticos 3 y 4.

- Tema 14: **Conversiones Explícitas**:
  Se debe incorporar en todo lugar donde pueda aparecer una expresión, la siguiente sintaxis:
  <tipo>(<expresión>)
  donde <tipo> será:

Grupo	tipo
1	linteger
6	uslinteger
11	uslinteger
18	linteger

- Tema 15: **Conversiones Implícitas**: Se explicará y resolverá en trabajos prácticos 3/4.
- Tema 16: **Sin conversiones:** Se explicará y resolverá en trabajos prácticos 3/4.

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 **integer** fuera del rango permitido.
  Línea 43: Falta paréntesis de cierre para la condición del **if**.
- Tabla de símbolos

Consignas

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.

c) Para aquellos tipos de datos que permitan valores negativos (**integer**, **linteger**, **single** y **double**) 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, puede estar fuera de rango si la constante es positiva.

- Ejemplo: Las constantes de tipo **integer** 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) Caso de prueba que contemple **todas** las estructuras válidas del lenguaje