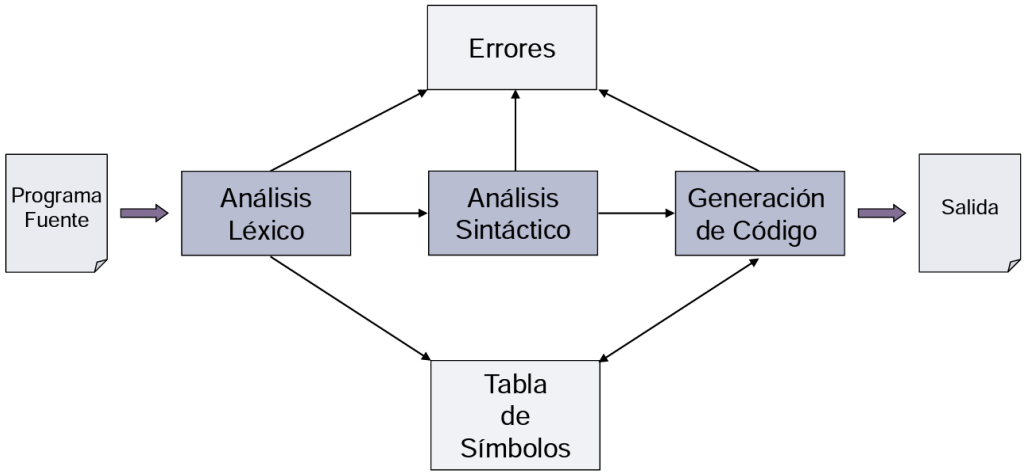
Trabajo práctico Especial
Diseño de Compiladores

Integrantes : Alvarez, Mauricio Ezequiel(maurialva.ma@gmail.com)
Fleba, Agustin (agusfleba75@gmail.com)
Cuthill, Malcom (malcolmcuthill98@gmail.com)
Ayudante:Mailen Gonzales
N° Grupo : 19

Primera Entrega AL

El objetivo de la primera entrega fue analizar las estrategias que se deben utilizar para desarrollar el compilador que reconozca un lenguaje propuesto por la cátedra. Del mismo modo, para la segunda entrega del Trabajo práctico, se desarrolló el parser que luego va a invocar al analizador léxico desarrollado anteriormente.
En esta primera entrega se adjuntan los archivos del desarrollo del analizador léxico y el analizador sintáctico.

Primera parte - Analizador Léxico



Lo primero que se llevo a cabo fue un analizador léxico. Este analizador que desarrollamos tenía que ser capaz de reconocer:

- *Identificadores cuyos nombres pueden tener hasta 22 caracteres de longitud. El primer carácter puede ser una letra, o "_", y el resto pueden ser letras, dígitos y "_". Los identificadores con longitud mayor serán truncados y esto se informará como Warning. Las letras utilizadas en los nombres de identificador pueden ser minúsculas y/o mayúsculas.*
- *Constantes correspondientes al tema particular asignado a cada grupo. Nota: Para aquellos tipos de datos que pueden llevar signo, la distinción del uso del símbolo „-„ como operador aritmético o signo de una constante, se postergará hasta el trabajo práctico Nro. 2.*
- ***Operadores aritméticos:** "+", "-" , "*", "/" agregando lo que corresponda al tema particular.*
- ***Operador de asignación:** ":="*
- ***Comparadores:** ">=", "<=", ">", "<", "==", "<>"*
- ***Operadores lógicos:** "&&", "||" "(", ")", "," y ";"*
- *Cadenas de caracteres correspondientes al tema particular de cada grupo.*
- *Palabras reservadas (en mayúsculas):*

IF, THEN, ELSE, ENDIF, PRINT, FUNC, RETURN, BEGIN, END, BREAK *y demás símbolos / tokens indicados en los temas particulares asignados al grupo.*

- *El Analizador Léxico debe eliminar de la entrada (reconocer, pero no informar como tokens al Analizador*

Sintáctico), los siguientes elementos.

- ***Comentarios*** *correspondientes al tema particular de cada grupo.*
- ***Caracteres en blanco, tabulaciones y saltos de línea****, que pueden aparecer en cualquier lugar de una sentencia.*

Temas asignados particularmente a nuestro grupo para el desarrollo :

4. Enteros largos sin signo: Constantes enteras con valores entre 0 y 232 – 1. Se debe incorporar a la lista de palabras reservadas la palabra ULONG.

6. Dobles: Números reales con signo y parte exponencial. El exponente comienza con la letra E(mayúscula) y el signo es opcional. La ausencia de signo, implica un exponente positivo. La parte exponencial puede estar ausente. Puede estar ausente la parte entera, o la parte decimal, pero no ambas. El „.‟ es obligatorio.
Ejemplos válidos: 1. .6 -1.2 3.E–5 2.E+34 2.5E-1 13. 0. 1.2E10
Considerar el rango 2.2250738585072014E-308 < x < 1.7976931348623157E+308
-1.7976931348623157E+308 < x < -2.2250738585072014E-308 0.0
Se debe incorporar a la lista de palabras reservadas la palabra DOUBLE.

7. Incorporar a la lista de palabras reservadas la palabra REPEAT.

13. Comentarios multilínea: Comentarios que comienzan con “//” y terminan con “//” (estos comentarios pueden ocupar más de una línea).

16. Cadenas de 1 línea: Cadenas de caracteres que comiencen y terminen con “ % ” (estas cadenas no pueden ocupar más de una línea). Ejemplo: %¡Hola mundo !% .

Desiciones de diseño o implementacion:

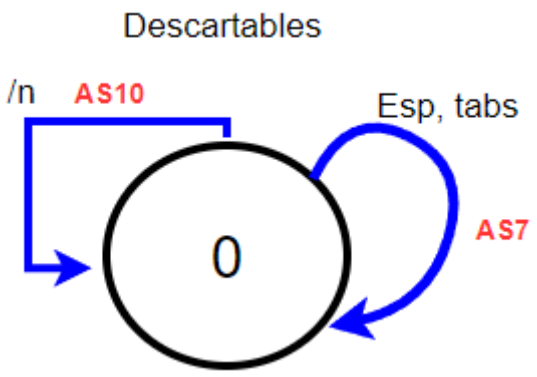
Las decisiones que fueron tomadas para la implementación son :

- Analizador léxico :
  - ☐ El analizador léxico finaliza cuando llega un EOF ya que no hay más nada que leer.De no ser asi transiciona hasta generar los todos los tokens y en caso de fallar genera los errores correspondientes que se imprimen al final de la compilación,en letra roja.Con los warnings pasa lo mismo pero estos son generados con letra amarilla.
- Ante cualquier situacion en la que no se genera un token, se genera el error y se siguen con la compilación.
- El exponente del DOUBLE en caso de no ser definido es 0.
- Cada vez que hay una falla, se reinicia la maquina de estados para evitar generar un token duplicado leído de la iteración anterior.
- Cuando el tamaño del string es superior al permitido, queda solo con las 22 primeras letras que son las permitidas.
- Para que el token sea considerado como CTE_ULONG el número deberá ser escrito seguido de una ul minuscula, por ejemplo : 25ul. Si fuera mal escrito, por ejemplo 25u, es tomado como si fuera una CTE_DOUBLE.

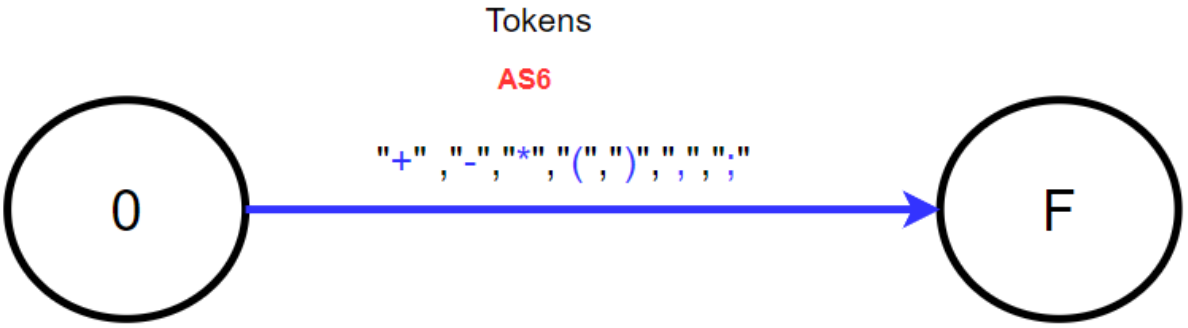
Diagrama de transición de estados:

Para que el grafo tenga una mejor legibilidad se dividió la máquina de estados en subgrafos que representan la obtención de cada token.
Los diagramas con los que se representó la máquina de estados son:

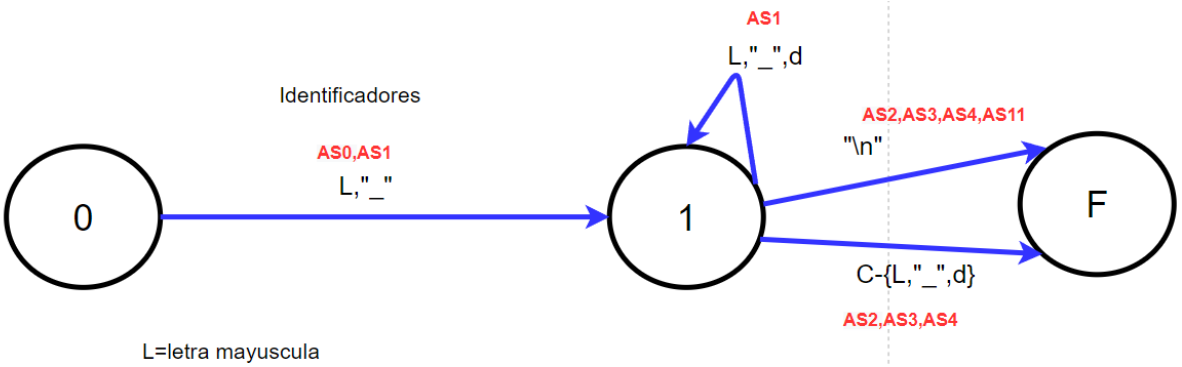
Descartables



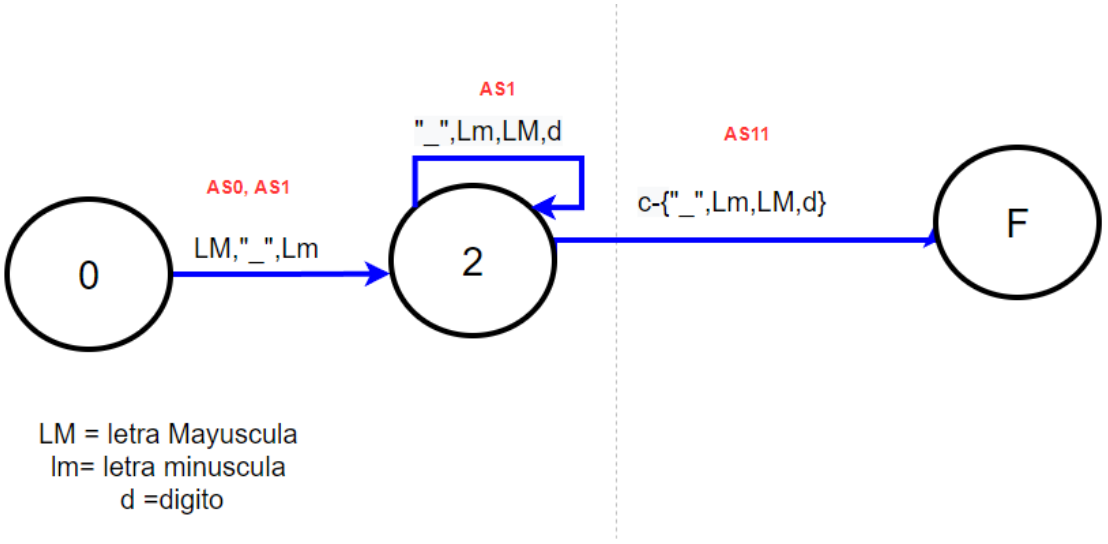
Tokens



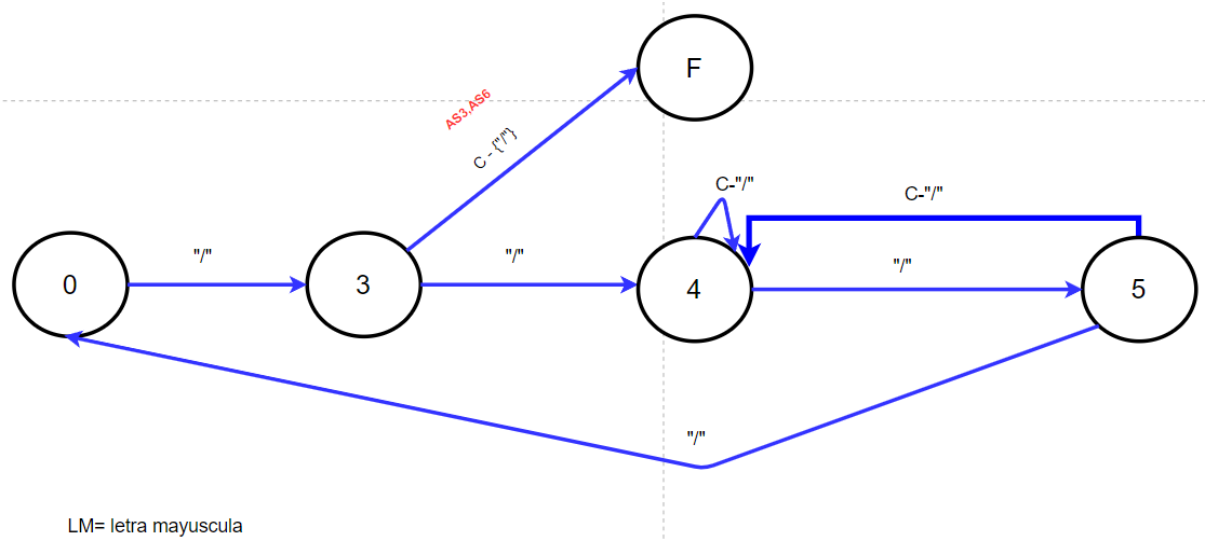
Identificadores



Reservadas

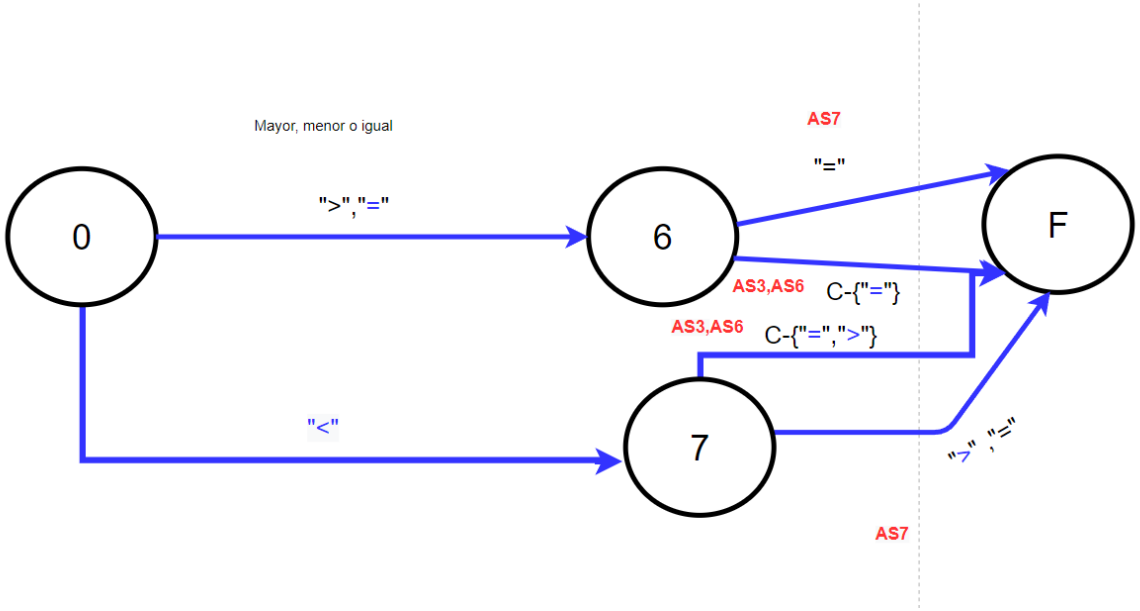


Comentario

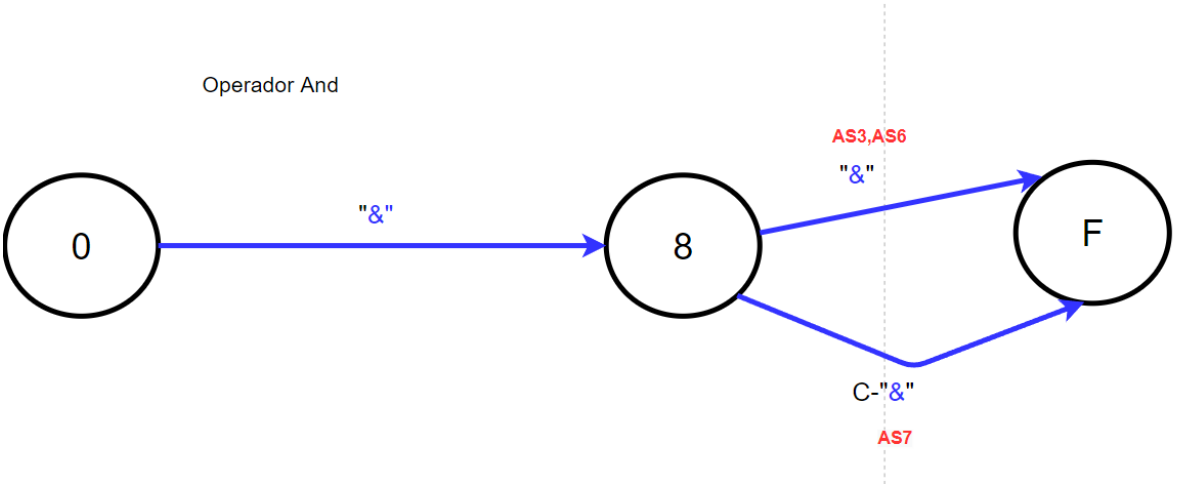


LM= letra mayuscula

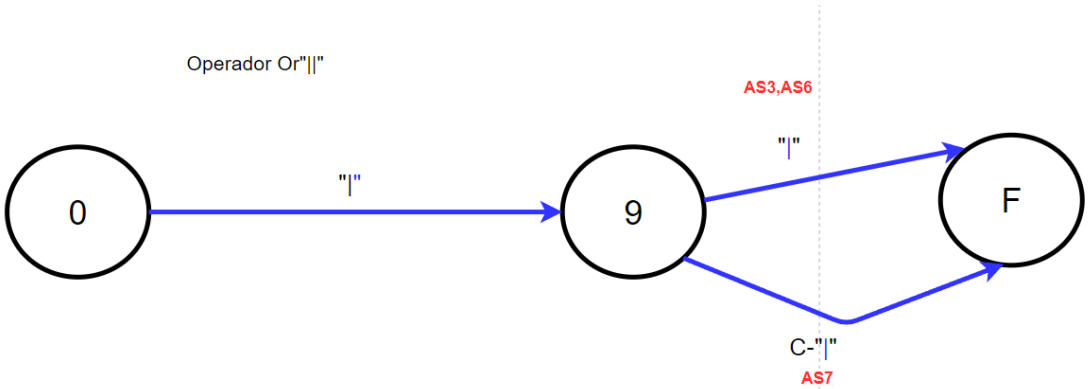
Comparadores



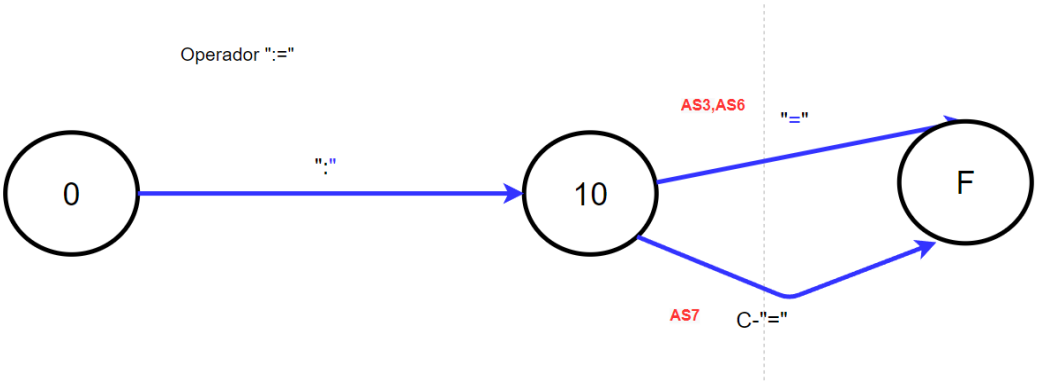
And



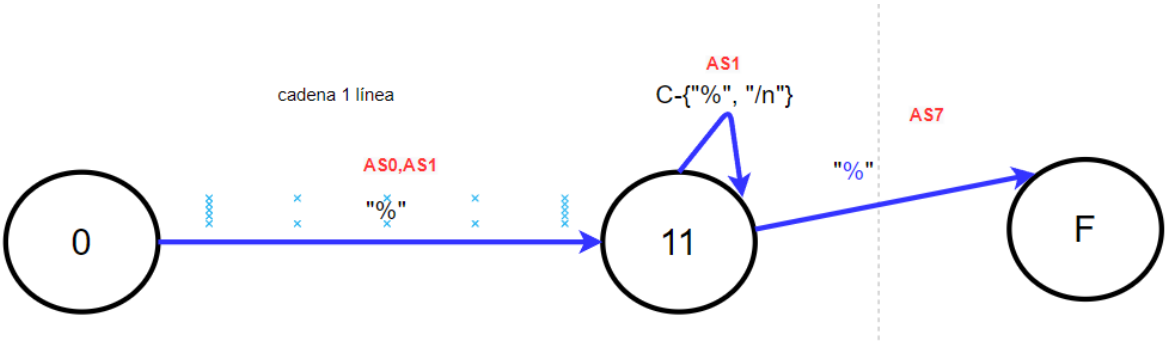
Or



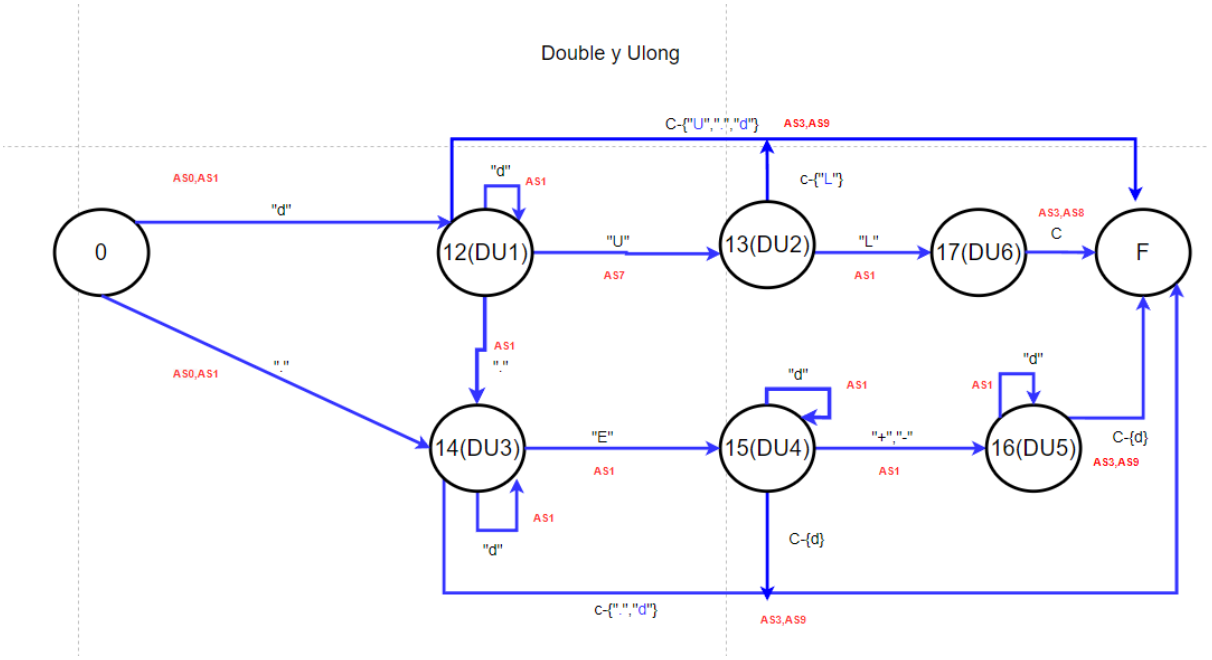
Asignación



1 línea



Double y ULONG



Acciones semánticas tenidas en cuenta

En esta sección se describen brevemente las acciones semánticas definidas en las acciones semánticas que se implementan, según la acción que se realiza.

- ☐ ***AS0 - InicStringVacio:***Inicia un string vacío para almacenar los símbolos leídos por el analizador léxico. Este string, se utiliza por las clases ConcatenaChar, TruncaId, GeneraTokenTS, GeneraTokenPR y detección de constantes numéricas.
- ☐ ***AS1 - ConcChar:*** Agrega el último símbolo leído a un string temporal.
- ☐ ***AS2 - TruncId***: extrae el string temporal en caso de que se supere un límite impuesto.
- ☐ **AS3 - Retrocede_Fuente**: Retrocede en el código fuente, permitiendo volver a leer el último símbolo leído.
- ☐ **AS4 - GeneraTS(tabla de simbolos):** agrega una nueva entrada a la tablada de símbolos, utilizando como lexema el contenido de un string temporal y agrega el token correspondiente a la lista de tokens.
- ☐ **AS5 - GeneraPR:** Corrobora que el contenido del string temporal matchee con una palabra reservada. si es así, agrega el token correspondiente a la lista de tokens. De lo contrario, notifica el error.
- ☐ **AS6 - GenerareTokenparticular:** Agrega un token específico a la lista de tokens.
- ☐ **AS7 - IgnoraC:** Ignora el último símbolo leído.
- ☐ **AS8 - GeneraUL:** Parsea el string temporal, chequea el rango del valor almacenado esté entre 0 y 2^32 -1.Si este esta dentro de este rango, lo inserta en la TS.
- ☐ ***AS9 - GeneraTokenDouble:*** Parsea el string temporal, luego asigna a un atributo asociado al exponente de un double. Además, construye y normaliza el double a partir de la base y exponentes almacenados. Si cumple con los requisitos lo agrega a la TS.
- ☐ **AS10 -Cuentasaltolinea :** Incrementa un contador asociado a la cantidad de líneas del fuente.
- ☐ **AS11 - ChequeoTipo** : Chequea si es una palabra reservada si lo es genera la AS5, si no, genera un token de la tabla de símbolos.

Además, dentro de las acciones semánticas tenemos las notificaciones de los warnings y los errores que realizan estas acciones en los casos que sea necesario .

Matriz de transición de estados :

El diagrama de transición de estados también se puede representar en forma de matriz, donde las columnas representan el último símbolo leído, las filas los estados(que se encuentran en la clase Estados del compilador) desde los que partimos y la casilla al símbolo(Inputs en el compilador) a la cual nos dirigimos junto con la acción semántica o las acciones semánticas que se realizarán en caso de que corresponda(en los casos que fueron escritos solo las AS correspondientes, estas van a estado final). Además, los errores que pueden ocurrir fueron marcados con -1.
La matriz de transición de estados se encuentra en el siguiente link de Google Sheet:
MATRIZ DE TRANSICIÓN DE ESTADOS Y ACCIONES SEMÁNTICAS

Errores léxicos considerados:

Dentro de los errores léxicos consideramos :

1. El exponente por defecto si no se encontrara es 0.
2. Cada vez que se lee una cte literal, el código la genera como si fuera una DOUBLE, si esta constante tiene solo la U, por ejemplo 2u, será tomada como CTE, no como ULONG.
3. Si encontramos un EOF en medio de una cadena es marcado como error.

Segunda Entrega:AS

Consigna:

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 (INT, LONG, 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 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.

Analizador Sintactico

En segundo lugar lo que se requirió fue desarrollar el parser para que posteriormente invoque al analizador léxico expuesto anteriormente. Este tenía que reconocer lo siguiente:

Programa:

- Programa constituido por un nombre de programa, seguido de un bloque de sentencias declarativas, y a continuación, un bloque de sentencias ejecutables.
- El bloque de sentencias ejecutables estará delimitado por **BEGIN** y **END**.
- 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>; **//incluyendo las declaraciones indicadas en temas particulares 17 a 24**

  Donde <tipo> puede ser (Según tipos correspondientes a cada grupo):

  INT
  UINT
  LONG
  ULONG
  SINGLE
  DOUBLE

  Las variables de la lista se separan con coma (",")
- Incluir declaración de funciones, con la siguiente sintaxis:

  <tipo> **FUNC** ID (<parametro>)
  <sentencias_declarativas_de_la_funcion> // conjunto de sentencias declarativas
  BEGIN
  <sentencias_ejecutables_de_la_funcion> // conjunto de sentencias ejecutables
  RETURN (<retorno>)
  END

  Donde:
  - <parametro> tendrá la siguiente estructura:
    - <tipo> ID
    - Sólo se permitirá declarar funciones con un parámetro
  - **RETURN** <retorno> será la última sentencia del cuerpo de la función
  - <retorno> podrá ser cualquier expresión aritmética

  Ejemplo:

```
INT FUNC f1 (INT y)
INT x,
BEGIN
        x := y;
        …
      RETURN (x);
END;
```

Sentencias ejecutables:

- Asignaciones donde el lado izquierdo es 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.
- Considerar como posible operando de una expresión, la invocación a una función, con el siguiente formato:
  ID(<parámetro>)
- Cláusula de selección (**IF**). Cada rama de la selección será un bloque de sentencias. La estructura de la selección será, entonces:

  IF (<condicion>) **THEN** <bloque_de_sentencias> **ELSE** <bloque_de_sentencias> **ENDIF;**

  El bloque para el **ELSE** puede estar ausente.
- La condición podrá ser una combinación lógica entre expresiones booleanas, que podrán ser comparaciones entre expresiones aritméticas. Una condición se escribe entre **"(" ")"**, y no puede incluir subexpresiones entre paréntesis dentro de la condición. Ejemplos de condiciones:

(a>3 && b <> 10 || j)
(a && b)
(j > z / 2)
(a + 2)
(w)

Las precedencias para los operadores, de mayor a menor, serán:

1) * , /
2) + , -
3) Comparadores >, <, >=, <=, ==, <>
4) && (and)
5) || (or)

- Un bloque de sentencias puede estar constituido por una sola sentencia, o un conjunto de sentencias delimitadas por **BEGIN** y **END.**
- Sentencia de salida de mensajes por pantalla. El formato será

```
PRINT(<cadena>);
```

Ejemplos:

```
PRINT(%Hola mundo %);            //Tema 16
PRINT(% Hola +                   //Tema 15
      +  Mundo%);
```

Además , a nuestro grupo se le asignaron los siguientes temas particulares:

- 7

- ***REPEAT*** (tema 7 en TP1)

  REPEAT (i = n; <condición_repeat>; 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_repeat> 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).

  Nota: Las restricciones de tipo serán chequeadas en la etapa 3 del trabajo práctico.

- ***WHILE DO*** (tema 8 en TP1)

  WHILE (<condicion>) **DO** <bloque_de_sentencias>

  <condición tendrá la estructura definida para sentencias de selección.

- ***REPEAT UNTIL*** (tema 9 en TP1)

  REPEAT <bloque_de_sentencias> **UNTIL** (<condicion>)

  <condición tendrá la estructura definida para sentencias de selección.

Para las 3 sentencias de control:

El bloque se sentencias puede contener una sentencia BREAK, que consistirá en la palabra reservada BREAK seguida de';'

- 11: Conversiones Implícitas: Se explicará y resolverá en trabajos prácticos 3 y 4.

- 18 :

Declaraciones de Funciones:

– A continuación de los encabezados de declaraciones de funciones, se podrá incluir una precondición con la siguiente estructura:

PRE: (<condición>);

Donde:

<condición> se definirá como las condiciones descriptas para la sentencia de selección.

Ejemplo:

```
FUNC INT f(INT x)
SINGLE y, z;
BEGIN
   PRE: (x > 3);
   RETURN (x + 1);
END;
```

Sentencias ejecutables:

– Incorporar la siguiente estructura:

TRY
<sentencia_ejecutable> /// No se permiten bloques TRY CATCH anidados
CATCH
BEGIN
<bloque de sentencias_ejecutables>
END

Ejemplo:

```
TRY
      x := f(z) * 4;
CATCH
BEGIN
      x := 10;
END;
```

Atención: Se debe incorporar al Análisis Léxico el reconocimiento de las palabras reservadas **PRE, TRY** y **CATCH** y el símbolo '**:**'.

- 25

▪ Tema 25 en TP1: Asignación de funciones a variables con inferencia de tipos.

Sentencias declarativas:

– Incorporar la declaración de variables de tipo FUNC, con la siguiente estructura:
FUNC <lista_de_variables>

Ejemplo:

```
FUNC x, y, z ;
```

Sentencias ejecutables:

– Sin cambios

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 **REPEAT UNTIL**
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 **INT** fuera del rango permitido.
Línea 43: Falta paréntesis de cierre para la condición de la sentencia **IF**.
– Contenidos de la Tabla de símbolos

LISTA DE NO TERMINALES:

A continuación se realiza una breve descripción de todos los NO TERMINALES utilizados en la gramática.
programa: Es la raíz del programa
conjunto sentencias declarativas: Compone al programa, describe conjuntos de sentencias declarativas de este.
conjunto_sentencias_ejecutables: Compone al programa, describe conjuntos de sentencias ejecutables de este.
sentencia_declarativa: Describe declaración de funciones o variables.
sentencia_control_repeat: Describe las sentencias de control del tipo *REPEAT*.
parametro: Describe los posibles tipos de parametros.
lista_variables: Describe un conjunto de *ID's*.
conjunto sentencias declarativas_funcion: Describe conjuntos de sentencias declarativas una función.
conjunto sentencias_ejecutables_funcion: Describe conjuntos de sentencias ejecutables de una función.
sentencia_ejecutable: Describe sentencias ejecutables: invocación, asignación, *TRY*, *CATCH*, *IF*, *WHILE*, *REPEAT* y *PRINT*.
ejecutable: Describe las alternativas de ejecución que pueden ocurrir dentro de una sentencia ejecutable.
tipo: Describe los distintos tipos posibles de un *ID*.
asignacion: Describe una asignación.
operador: Describe los operadores *AND* (&&) y *OR* (||)
expresion: Describe una expresión y sus operaciones.
termino: Describe un término y sus operaciones.
factor: Describe un factor y sus operaciones.
sentencia_if: Describe una sentencia *IF*.
rama_then_sola: Describe la rama *THEN* de una sentencia *IF*.
rama_else: Describe la rama *ELSE* de una sentencia *IF*.
condicion: Describe una condición.
condicion_repeat: Describe las condiciones posibles que se pueden dar en una sentencia que contiene *REPEAT*
comparador: Describe un comparador que se utiliza en una condición.
sentencia_salida: Describe la sintaxis de un *PRINT*.

Además de los mencionados, preferimos separar los no-terminales asociados a la gramática que se encarga de reconocer errores. Estos no-terminales son los siguientes:

error_sentencia_salida: Describe los errores que se pueden dar en la declaración de las sentencias PRINT.

error_sentencia_if : Describe los errores que se pueden dar en la declaración de las sentencias de control repeat
error_sentencia_ejecutables : Describe los errores que se pueden dar en la declaración de las sentencias ejecutables como el TRY-CATCH.
error_sentencia_declarativas : Describe los errores que se pueden dar en la declaración de las sentencias declarativas relacionadas a la declaración de funciones y tipos..

Para poder desarrollar el analizador sintáctico fue necesario la implementación de distintos aspectos. Lo primero fue definir la gramática asociada al lenguaje a reconocer, teniendo en cuenta los tipos de datos, tipos de sentencia, expresiones, etc. Luego, con la ayuda de la herramienta **YACC** (*yet another compiler compiler* o compilador de compiladores) generamos el parser. Esta herramienta nos facilita la creación de una máquina de estados de pila para poder reconocer los terminales y no terminales anteriormente definidos.
En cuanto a la gramática, esta cuenta con 3 elementos: terminales, reglas de inicio y reglas de la gramática. Los terminales (también denominados tokens), son los distintos elementos que nuestra gramática considera. Algunos de estos son identificadores, palabras reservadas, operadores aritméticos, comparadores, etc.
Una vez que fuimos siguiendo los pasos mencionados, fueron surgiendo algunos errores. Estos eran de 2 tipos: reduce/reduce o shift/reduce. Para que yacc nos entregue el parser era necesario resolverlos. Para eso primero era necesario identificar por que se daban estos errores, por lo que determinamos que: los errores shift/reduce se dan cuando el compilador no sabe si reducir por una regla, o hacer un shift y luego reducir por otra regla. En cuanto a los errores reduce/reducen, estos se dan por un claro error en la gramática, donde está es ambigua. Por lo tanto tuvimos que reestructurar nuestra gramática para que no se den más este tipo de errores.

ERRORES SEMÁNTICOS CONSIDERADOS:

error_sentencia_ejecutable: Se consideran como errores la ausencia del CATCH, la falta de la sentencia_ejecutable, la falta del BEGIN y faltas de “ ; ”.

error_sentencia_declarativa: Se consideraron error la falta de lista de variables, la ausencia de los ID
error_sentencia_if: Se consideraron como error la ausencia de la rama “THEN” y la falta del paréntesis de cierre
error_repeat: se consideró como error la ausencia del “;” final.
error_sentencia_salida: Se consideraron como errores la ausencia de cadena a imprimir, y la falta del primer paréntesis de la expresión.

CONCLUSIÓN:

Para concluir creemos que el proceso de realizar la gramática, con su correspondiente corrección de errores, fue muy útil para entender bien cómo funciona el analizador semántico de los compiladores que utilizamos habitualmente. Además, creemos que la búsqueda de cada error que nos iba surgiendo nos llevó a entender en profundidad cómo desarrollar una gramática .