Universidad Nacional del Centro de la
Provincia de Buenos Aires

FACULTAD DE CIENCIAS EXACTAS

Ingeniería de Sistemas

Trabajos Prácticos 3 y 4

Diseño de Compiladores I

Docente: Mailén González

GRUPO 17
Abraham, Maria Belen: abraham.belen.99@gmail.com
Piu, Florencia Jesus: fpiu@alumnos.exa.unicen.edu.ar
Silvestri Ayala, Giuliana Estefania: gsilvestriayala@alumnos.exa.unicen.edu.ar

19/11/24

ÍNDICE

Introducción

El siguiente proyecto tiene como objetivo continuar el desarrollo del compilador, completando la etapa de generación de código. Se incorporó la funcionalidad de generación de código intermedio en una estructura de Árbol Sintactico, información semántica y chequeos, chequeos semánticos, generación de código assembler con el mecanismo de variables auxiliares en Pentium 32. Finalmente se incorporaron los controles en tiempo de ejecución correspondientes, control de error para división por cero, overflow en suma de datos de punto flotante y valor fuera de rango en variables declaradas de un subtipo de un tipo básico.

Temas Particulares

- **Tema 11**: **Subtipos**

  Cuando el compilador detecte la declaración de un nuevo tipo, definido como subrango de un tipo básico, se d registrar en la entrada correspondiente de la Tabla de Símbolos:
  - Información del nuevo tipo:
    - Nombre del tipo
    - Tipo base
    - Límite inferior
    - Límite superior
  - Otros atributos que el compilador requiera para permitir esta funcionalidad en el lenguaje.

  Por ejemplo, para:

  ```
  typedef enterito := integer[-10,10];
  ```

  se debe registrar que enterito es un tipo, que el tipo base es integer, y los límites inferior y superior del rango:

  Cuando se detecte una declaración de variables del nuevo tipo, por ejemplo:

  ```
  enterito a, b, c;
  ```

  se deberá indicar, para las entradas de las variables declaradas en la Tabla de Símbolos, que el tipo es enterito.

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

  ```
  WHILE ( <condicion> ) <bloque_de_sentencias_ejecutables> ;
  ```

  El bloque de sentencias ejecutables se ejecutará mientras la condición sea verdadera.

- **Tema 18**: ***Asignaciones múltiples***

  Se deberá generar código para asignar cada elemento de la derecha al correspondiente de la izquierda, en el orden en que se presenten.

  Por ejemplo, para:

  ```
  a,b,c := 1+d, e, f+5;    // Se deberán generar 3 asignaciones: a:=1+d, b:=e y c:=f+5
  ```

  En caso que del lado izquierdo haya menos elementos que del lado derecho, se descartarán las expresiones sobrantes del lado derecho, y se informará la situación con un Warning.

  En caso que del lado izquierdo haya más elementos que del lado derecho, se asignará a los elementos sobrantes el valor 0, y se informará la situación con un Warning.

  Por ejemplo, para:

  ```
  x,y := 1, f1(3), w;          // Se descartará el último elemento del lado derecho
  i, j , k := a*3 , b;         // Se deberán generar 3 asignaciones: i:=a*3, j:=b y k:=0
  // En ambos casos, se emitirá un Warning:
  ```

- **Tema 21**: ***pair***

  Cuando el compilador detecte la declaración de un nuevo tipo **pair**, se deberá registrar en la Tabla de Símbolos, la información del nuevo tipo:

  - Nombre del tipo
  - Tipo de los componentes
  - Otros atributos que el compilador requiera para permitir esta funcionalidad en el lenguaje.
  - Por ejemplo, para:
  - `typedef pair <integer> pint;` //declara un tipo par de enteros

  se debe registrar que pint es un tipo, cuyos componentes son de tipo integer.

  Cuando se detecte una declaración de variables del nuevo tipo, por ejemplo:

  ```
  pint p1,p2,p3;    // declaración de las variables p1 a p3 con el tipo pint
  ```

  se deberá indicar, para las entradas de las variables declaradas en la Tabla de Símbolos, que el tipo es pint.

  Las variables declaradas de un tipo **pair** sólo podrán utilizarse en asignaciones donde el lado izquierdo y derecho sean variables de este tipo. No debe permitirse su uso en otro tipo de sentencias / expresiones.

  Los componentes de las variables declaradas con el nuevo tipo podrán ser utilizados en cualquier lugar donde pueda utilizarse una variable, con las mismas reglas y chequeos que se consideran para variables.

- **Tema 23**: ***goto***

  Cuando el compilador detecte un goto a una etiqueta, deberá generar en el código intermedio una bifurcación incondicional a la posición donde se encuentre la etiqueta correspondiente.

  Al detectar la etiqueta, se debe incorporar la etiqueta al código intermedio, ya sea como un nodo del árbol, un terceto tipo etiqueta, o un elemento de la Polaca Inversa.

  Se debe chequear la existencia de la etiqueta a la que se pretende bifurcar.

Tema 26: ***Conversiones Implícitas***

- El compilador debe incorporar las conversiones en forma implícita, cuando se intente operar entre operandos de diferentes tipos (uno entero y otro flotante). En todos los casos, la conversión a incorporar será del tipo entero (1-2-3-4)

  al tipo de punto flotante asignado al grupo. Por ejemplo, el compilador incorporará una conversión del tipo **INTEGER** a **SINGLE**, si se intenta efectuar una operación entre dos operandos de dichos tipos.
- En el caso de asignaciones, si el lado izquierdo es de uno de tipo entero (1-2-3-4), y la expresión del lado derecho es de tipo diferente, se deberá informar error por incompatibilidad de tipos. En cambio, si el lado izquierdo es del tipo de punto flotante asignado al grupo, y el lado derecho es de tipo entero (1-2-3-4), el compilador deberá incorporar la conversión que corresponda al lado derecho de la asignación.

Generación de Código Intermedio

Estructura utilizada: Árbol Sintáctico

El árbol sintáctico utilizado para almacenar el código intermedio se compone de nodos comunes, nodos de control, nodos de asignación y nodos hojas.

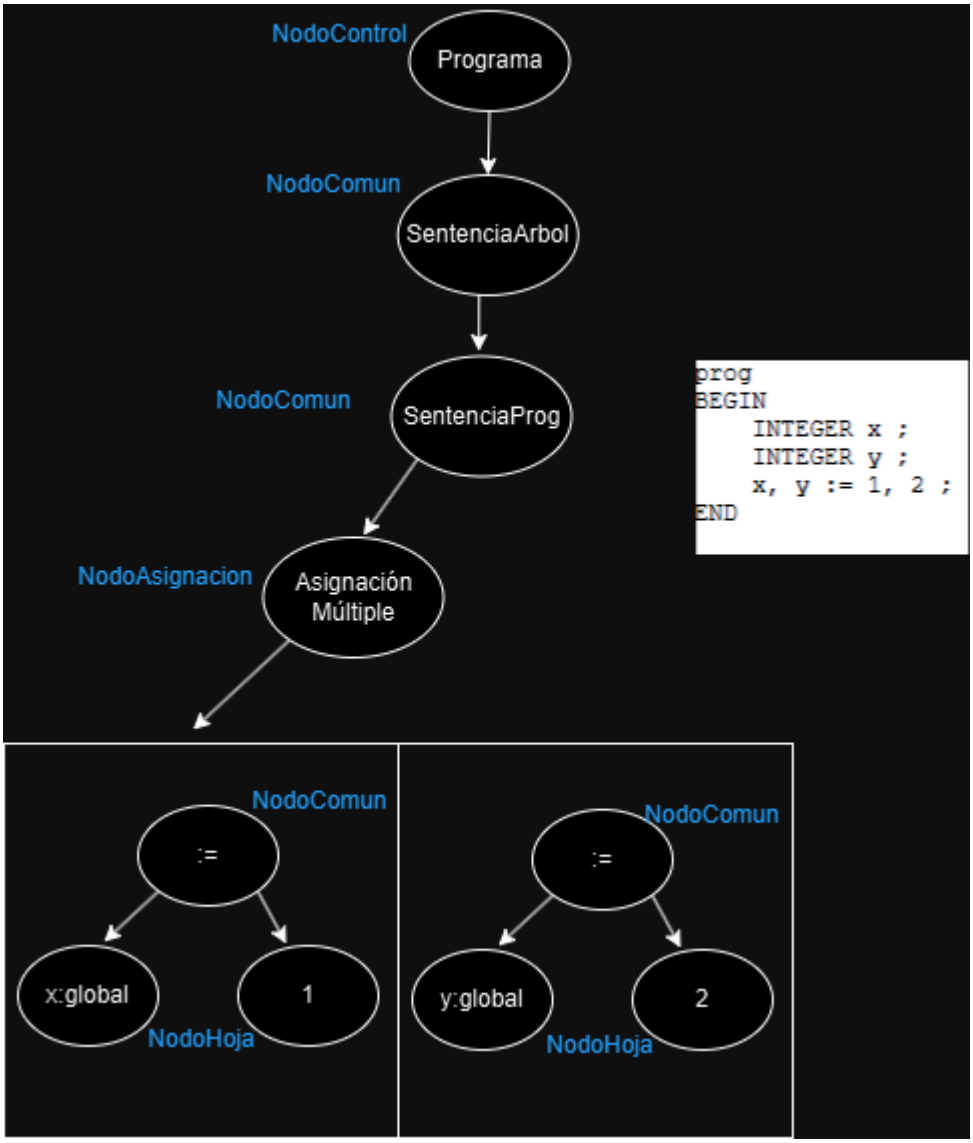
- Los **nodos comunes** representan a las sentencias ejecutables.

- Los **nodos de control** representan a las sentencias ejecutables de selección, flujo de control(goto), y funciones.

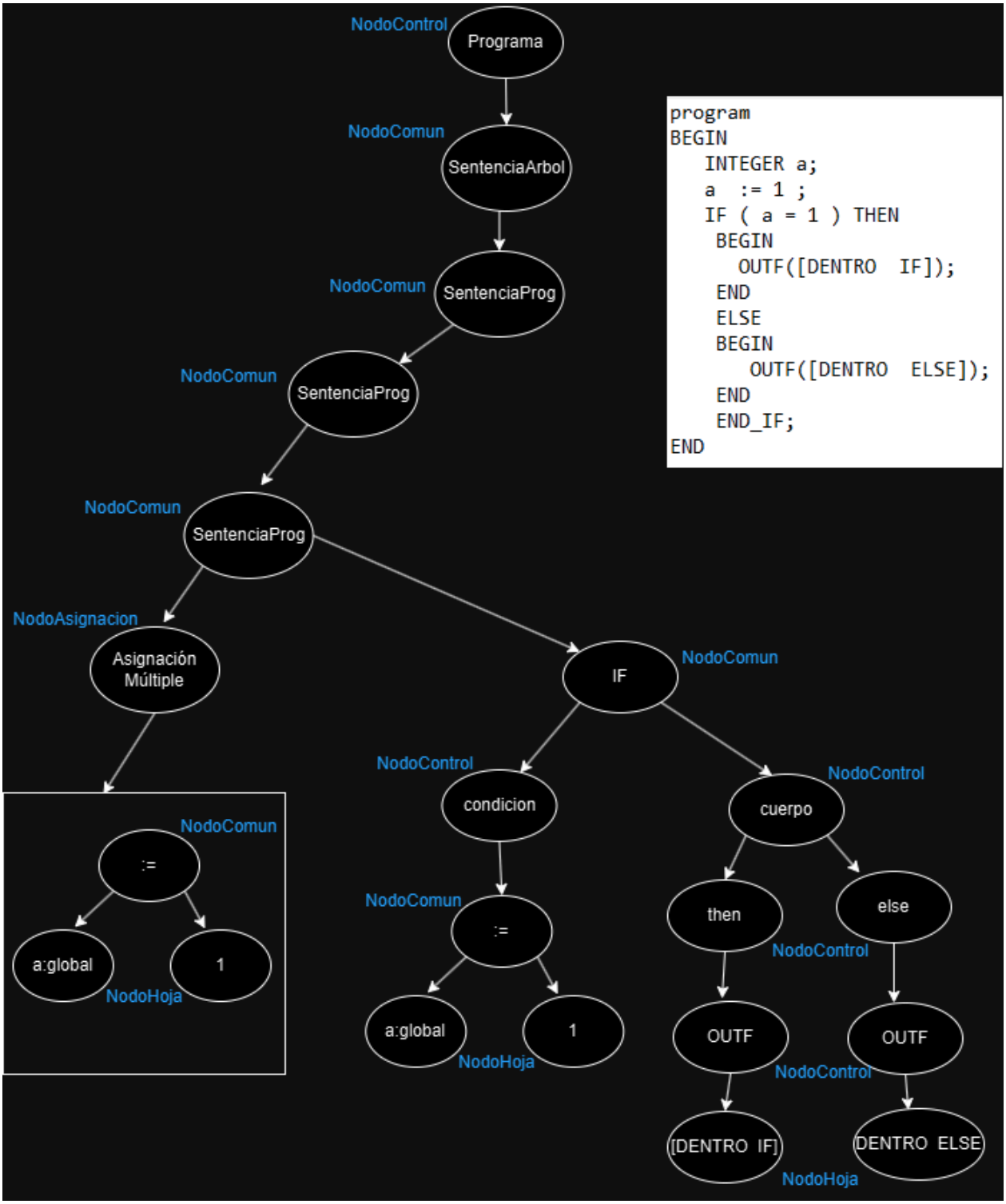
- Los **nodos hojas** representan constantes o identificadores.

- Los **nodos de asignación** están compuestos por una lista de árboles comunes que representan a cada asignación.

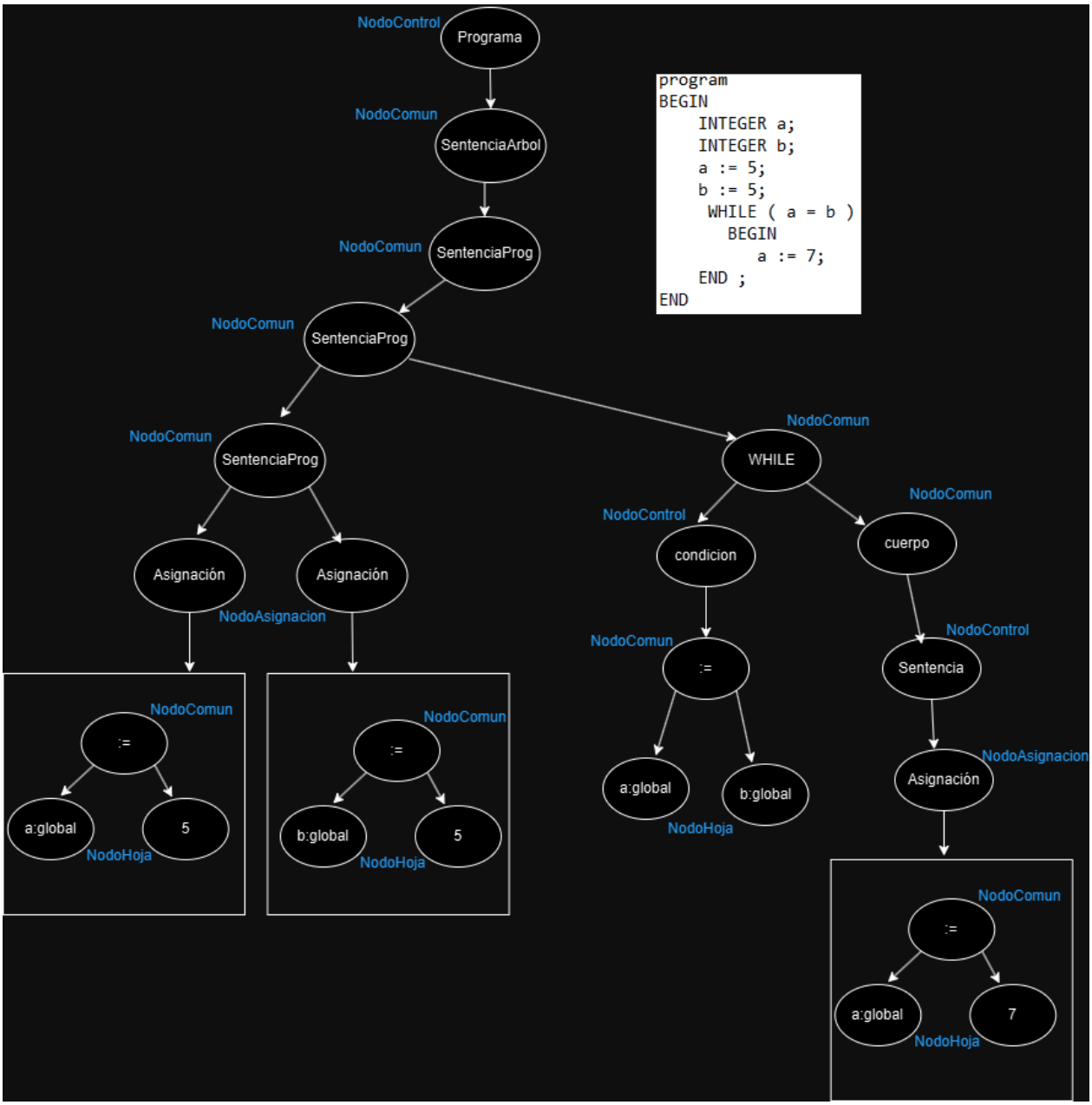
Ejemplo del árbol de asignación múltiple



Ejemplo del árbol de sentencia IF



Ejemplo del árbol de sentencia WHILE



Uso de notación posicional de Yacc

La notación posicional de Yacc fue utilizada para crear una representación de la estructura del programa. De esta manera almacenamos el código intermedio en una estructura de árbol sintáctico. También se utilizó para asignar los valores a los nodos hoja y para realizar modificaciones en la tabla de símbolos.

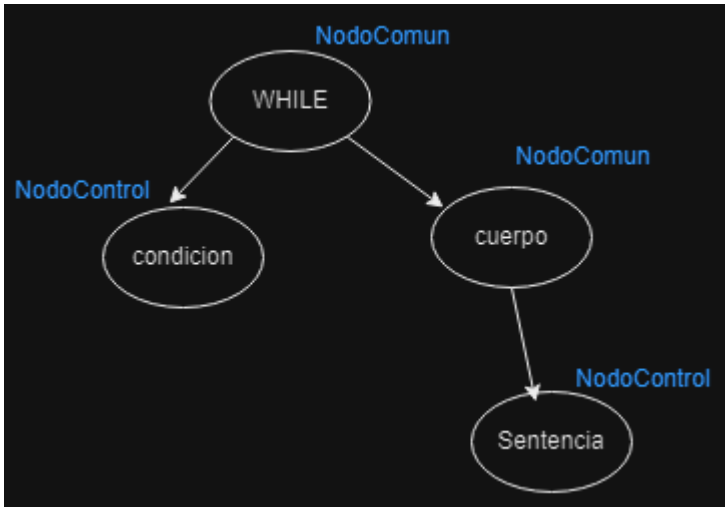
$$ fue utilizado en la mayoría de las reglas gramaticales para ir generando y retornando los nodos del árbol hasta que la estructura del programa queda completamente representada en un árbol sintáctico.

$n fue utilizado para acceder a los valores de los componentes de las reglas de producción.
Para acceder a los valores de constantes o identificadores utilizamos $n.sval y $n.obj para acceder al árbol.

Generación de las bifurcaciones en sentencias de control

Descripción tipo pseudocódigo del algoritmo

```
if (nodoCondicion != nodoError) {

    nodoBloque = obtenerNodoBloqueSentencias()

    bloque = crearNodoControl("sentencia", nodoBloque)

    cuerpo = crearNodoComun("cuerpo", bloque, null)

    nodoCondicion = obtenerNodoCondicion()

    condicion = crearNodoControl("Condicion", nodoCondicion)

    arbol = crearNodoComun("WHILE", condicion, cuerpo))

}else{
    arbol =  obtenerNodoCondicion()
  }
```



Errores semánticos

En esta etapa del compilador detectamos errores semánticos durante el procesamiento de las reglas. En caso de encontrar un error semántico o sintáctico, decidimos utilizar un “nodoError” constante, el cual se agrega al árbol. Esta acción genera que no se creen nodos innecesarios en el árbol ya que el mismo es constante y siempre que se detecta un error, se hace referencia al mismo nodo. El compilador sigue ejecutando aunque encuentre un error y estos se van almacenando en una lista para luego imprimirlos en la salida.

Errores semánticos considerados:

● Variable no declarada en el ámbito
● Variable re declarada en el mismo ámbito
● Identificador re declarado en el mismo ámbito
● Función fuera de ámbito
● Función no declarada
● Tipo Parametro formal distinto Parametro Real
● Tipo no declarado
● Incompatibilidad de tipo en la asignación
● El tipo de la comparación es distinto
● No existe la etiqueta
● No existe el salto
● LimiteInf mayor que LimiteSup
● Indice Pair fuera de rango

Generación de Código Assembler

Mecanismo utilizado para la generación de código assembler

A partir del código intermedio generado, se genera el código assembler utilizando el mecanismo de variables auxiliares, si es que no ocurre ningún error léxico, sintáctico y semántico. Para generarlo, creamos la funcion “getAssembler” en los distintos nodos del árbol, la cual se encarga a partir del lexema, tipo y uso que contiene el nodo, de seleccionar la traducción correspondiente en assembler para añadirla en el archivo de salida. A su vez, creamos una clase “GeneradorAssBase” la cual se encarga de armar la estructura de la salida del compilador, convirtiendo los datos de la tabla de símbolos en formato assembler, agregando el encabezado y la parte final.

Mecanismo utilizado para efectuar operaciones aritméticas

El código assembler de las operaciones aritméticas se genera en la función ‘’getAssembler’’ de un NodoComun, la función se basa en un CASE que dependiendo de las características del nodo y de sus hijos agrega el código correspondiente a la salida Assembler. Por ejemplo, si en el nodo se detecta una suma en la cual ambos hijos son identificadores o constantes, el código assembler generado será:

```
MOV AX , $var1
ADD AX , $var2
MOV @aux , AX
```

Siguiendo la secuencia de instrucciones anteriores, en principio se mueve el valor de la $var1 al registro AX de 16 bits, luego se le suma el contenido de $var2 al registro AX. Y por último le asigna a @aux el valor resultante en AX.

Los cambios realizados a los identificadores de las variables para un correcto funcionamiento del assembler fueron las siguientes:

- Identificadores, constantes y cadenas: Se le agregó al principio de cada identificador el signo “$”, para diferenciarlo de las variables auxiliares. Reemplazamos “:” por “$”. También reemplazamos “.” por “_” ya qué punto es utilizado para otras funciones dentro del assembler. Por último si el identificador contiene un “+” o un “-” se reemplaza por “$”, para resolver conflictos.

Las variables auxiliares las armamos concatenando el string “@aux” y el entero “indiceAux” que lleva el índice del último auxiliar asignado dando por resultado “@aux+indiceAux”.

Mecanismo utilizado para la generación de las etiquetas destino de las bifurcaciones.

Para el manejo de las etiquetas, utilizamos una pila de labels.

Al momento de agregar una etiqueta se concatena el string “label” y el entero “indiceLabel” que se incrementa al apilar dando como resultado “label+indiceAux+’:’ ” esto se agrega en el tope de la pila.

Controles en tiempo de ejecución asignados

a) **División por cero para datos enteros y de punto flotante:**
El código Assembler deberá chequear que el divisor sea diferente de cero antes de efectuar una división. Este chequeo deberá efectuarse para los dos tipos de datos asignados al grupo.

c) **Overflow en sumas de datos de punto flotante:**
El código Assembler deberá controlar el resultado de la operación indicada, para el tipo de datos de punto flotante asignado al grupo. Si el mismo excede el rango del tipo del resultado, deberá emitir un mensaje de error y terminar.

h) **Valor fuera de rango en variables declaradas de un subtipo de un tipo básico (tema 11)**
El código Assembler deberá controlar que una variable declarada de un tipo definido por el usuario como subtipo de un tipo básico, no tome valores fuera del rango indicado en la definición del tipo.

Modificaciones en etapas anteriores

- Se agregó declaracion_especial como regla en la gramática, específica para las declaraciones de subtipos y de tipo pair.
- Se modificó la clase Símbolo:
    - Se agregó Rango1 y Rango2, donde se guardan los rangos definidos de los subtipos
    - Se agrego TipoParametro :
        - Si el símbolo tiene como uso identificador y el tipo es un id con uso Pair,se usa para guardar el tipo con el que fue declarado el Pair.
        - Si el símbolo tiene como uso identificador y el tipo es un id con uso Subtipo, se usa para guardar el tipo con el que fue declarado el Subtipo.

- Se modificó Tabla de Símbolo:
    - Se agregó la función ObtenerParametro: A partir del nombre de la función, devuelve el parámetro.
    - Se modificó la función imprimirTabla : dependiendo del uso y del tipo imprime los atributos correspondientes
        - Funcion: f1:global: ID 257: Tipo INTEGER Uso funcion Ambito global
        - Pair:

pint:global: ID 257: Tipo INTEGER Uso PAIR Ambito global
p1:global: ID 257: Tipo pint:global Uso identificador Ambito global Subtipo: INTEGER
p2:global: ID 257: Tipo pint:global Uso identificador Ambito global Subtipo: INTEGER

- Subtipo:

ent:global: ID 257: Tipo INTEGER Uso SUBTIPO Ambito global Indice Inf:-1 Indice Sup:10
n:global: ID 257: Tipo ent:global Uso identificador Ambito global Subtipo: INTEGER
Indice Inf:-1 Indice Sup:10

- Identificador:

a:global: ID 257: Tipo INTEGER Uso identificador Ambito global

Resolución de temas particulares

Tema 11: Subtipos

typedef enterito **:= integer**[-10,10]**;**

Reglas en la gramática:

- declaracion_tipo_subrango: TYPEDEF ID ASIG tipo rango
  - rango: '{' const ',' const '}':
  - En la gramática se controla que el rango superior no sea menor que el rango superior.

Decisiones:

- Declaración de variables: Si se declara una variable del nuevo tipo, se guarda en subtipo(atributo de la tabla de símbolo) en tipo del padre
  - Ejemplo:

    typedef enterito **:= integer**[-10,10]**;** →Se guarda enterito en una lista de subtipos

    enterito n; → En la tabla de Símbolos a n se le setea TipoParametro en Integer,además de setear los valores de los rangos.
- Para realizar el assembler se tiene en cuenta el tipo declarado en TipoParametro.

Tema 21: Pair

typedef pair <integer> pint;

Reglas en la gramática:

- declaracion_tipo_pair: TYPEDEF PAIR '<'tipo'>' ID
- var_pair: ID '{' CTE '}' →Se controla en la gramática que CTE sea 1 o 2.
  - En la gramática se controla que ID exista en la lista de Subtipos o en la lista de tipo Pair
- var_pair ASIG list_expresion
  - En la gramática se controla que lo que existe en list_expresion sea del mismo tipo que var_pair o que el subtipo.
  - Ejemplo:

    typedef pair <integer> pint; → Se agrega pint a la lista de topo pair.

    pint p1,p2; → se controla que pint exista en la lista de tipos Pair

    p1{1}:= 34; → Se controla que los tipos sean compatibles.

    p1 := p2; → En este caso se controla que p1 y p2 sean del mismo tipo Pair.

Decisiones:

- Declaración de variables:si se declara una variable del nuevo tipo, se guarda en TipoParametro(atributo de la tabla de símbolo) el tipo del padre

- Cuando se utiliza la regla var_pair se agrega en el árbol un nodo común índice pair,para poder traducirlo al assembler
- Para el assembler se controla si el tipo de la variable,se usa como pair,entonces se trata como un arreglo de 2 variables.Luego el subtipo de la variable para saber si es un arreglo de integer o double.

Para los tipos especiales se creó una regla nueva:

- Declaracion_especial: ID list_var
    - En la gramática se controla que ID exista en la lista de Subtipos o en la lista de tipo Pair
    - Si existe se cambia en TipoParametro el tipo del ID.
    - Si el ID es Subtipo se setean los rangos.

Tema 13: While

Reglas en la gramática:

- sentencia_while: WHILE '(' condicion ')' bloque_sentencias_del
- bloque_sentencias_del: BEGIN bloque_ejecutable END
- bloque_ejecutable: bloque_ejecutable sentencias_ejecutables
  | sentencias_ejecutables

Decisiones:

- Dentro de bloque_ejecutable se crea un nodo Común donde están anidadas las sentencias ejecutables.
- En sentencia_while: Se crean 2 nodos control, uno que se llama condición que es donde se anida la condición del while y otro que se llama sentencia donde se anidan las sentencias.Además se crea un nodo Común que se llama WHILE,y se agrega condición como hijo izq. y Sentencia como hijo der.

Tema 18: Asignaciones Múltiples

Reglas en la gramática:

- sentencia_asignacion: list_var ASIG list_expresion
- list_var: ID
  | ID ',' list_var ;
- list_expresion: expr_aritmetic
  | var_pair
  | list_expresion ',' expr_aritmetic;

Decisiones:

- List_var: los identificadores se guardan en una lista_identificadores de tipo string.
- List_expresion: las expresiones se guardan en una lista de arblores.
- En la regla de asignación se controla:
    - En caso que del lado izquierdo haya menos elementos que del lado derecho, se descartarán las expresiones sobrantes del lado derecho, y se informará la situación con un Warning.

- En caso que del lado izquierdo haya más elementos que del lado derecho, se asignará a los elementos sobrantes el valor 0, y se informará la situación con un Warning.
- Se controla la compatibilidad de tipos.

Tema 23: Goto

Reglas en la gramática:

- sentencia_goto: GOTO etiqueta
- etiqueta: ETIQUETA
- sentencia_programa: sentencias_declarativas
  |sentencias_ejecutables
  | etiqueta

Decisiones:

- En la gramática cuando se llega a la regla sentencia_goto se crea el nodo control llamado GOTO que tiene como hijo a la etiqueta.
- En la gramática cuando se llega a la regla sentencia_programa:etiqueta se crea el nodo control llamado Salto_Etiqueta que tiene como hijo a la etiqueta.
- Las decisiones anteriores son para poder generar el assembler del salto.

Tema 26: Conversiones Implícitas

Decisiones:

- Cuando se realizan operaciones aritméticas si alguno de los 2 lados es double,se convierte el de tipo integer a double.
- En asignación solo se permiten conversiones si el del lado izq es double y el del lado der integer,si es al revés da error.

Controles en Tiempo de Ejecución

a) División por 0 para datos enteros y de punto flotante:

- Se genera el código ensamblador para los operandos izquierdo y derecho de la división.
- Antes de realizar la división, se compara el divisor con cero.
- Si el divisor es cero, se salta a una rutina de manejo de errores (errroDivisionPorCero).

- Si el divisor no es cero, se realiza la división y se almacena el resultado en una variable auxiliar.

b) Overflow en suma de datos de punto flotante:

- Primero se realiza la suma, luego se compara el resultado con una constante que contiene el máximo valor permitido. Si el resultado excede el valor máximo, saltar a la rutina de manejo de errores (errorOverflowSumaDouble).

c) Valor fuera de rango en variables declaradas de un subtipo de un tipo básico (tema 11):

- Se compara el valor de la variable con el límite inferior, si es menor saltar a la rutina de manejo de errores (errorOverRangoSubtipoMenor).

- Se compara el valor con el límite superior, si es mayor saltar a la rutina de manejo de errores (errorOverRangoSubtipoMayor).

Conclusiones

Durante estos trabajos prácticos, hemos aprendido a realizar análisis léxico, sintáctico y semántico de un código, comprendiendo cómo un compilador genera tokens e identifica sentencias claves. En las etapas 3 y 4, implementamos la generación de código intermedio y ensamblador, asegurando que el código resultante sea ejecutable en el procesador. Además, abordamos la detección y corrección de errores semánticos y de tiempo de ejecución. Este proceso nos proporcionó una visión del funcionamiento de un compilador, desde el análisis inicial hasta la ejecución del código.