Universidad Nacional del Centro de la
Provincia de Buenos Aires

FACULTAD DE CIENCIAS EXACTAS

Ingeniería en Sistemas

Entrega Final

Diseño de Compiladores I

Temas asignados: 1 6 7 10 11 13 19 22 23 27 28

9/12/2024

ÍNDICE

Temas

1. Enteros (16 bits)

6. Punto flotante de 64 bits

7. Constantes octales

10. Cadenas multilínea:

11. Subtipos

13. WHILE

19. Pattern Matching

22. TRIPLE

23. GOTO

27. Sin conversiones en expresiones / Conversiones explícitas de parámetros

28. Comentarios de 1 línea

Chequeos en Ejecución

a) División por cero para datos enteros y de punto flotante.

d) Overflow en productos de enteros.

h) Valor fuera de rango en variables declaradas de un subtipo de un tipo básico (tema 11) .

Correcciones

Errores cometidos

Generación de polaca cuando existían errores.

La anterior versión generaba la polaca a pesar de que existieran errores léxicos o sintácticos. En esta nueva versión al detectar un error léxico o semántico, detenemos todo lo relacionado a la generación de código intermedio a partir de ese momento.
De esta manera, la polaca se termina de generar mientras no existan errores léxicos o sintácticos, para así, poder detectar errores semánticos.

Manejo de bifurcaciones en tipos Doubles

En la anterior versión, se utilizaba la instrucción JAE en lugar de JGE. Con esta se realizaban los saltos condicionales, tanto en tipo enteros como en tipo double. Esto era incorrecto debido a que dicha instrucción fallaría en tipos enteros con signo negativo.
En la nueva versión, todas las bifurcaciones condicionales de operaciones Double utilizan operaciones de salto no signado, mientras que las operaciones Integer se mantienen con operaciones de salto signado.

La nueva versión del método "*operadorSaltoCondicional()*" es la siguiente:

```
public static void operadorSaltoCondicional(String elemento, StringBuilder codigo, String comparadorAnterior, String tipoOperandos) {
    String operador = pila.pop(); // Es la direccion a saltar
    switch (comparadorAnterior) {
    // Le paso el operador anterior al salto para saber que comparaciï¿½n era y asï¿½ usar el jump adecuado
        case ">":
            if(tipoOperandos.contains("INTEGER")||tipoOperandos.contains("OCTAL")) {
                codigo.append("JLE LABEL" + operador + "\n \n"); // Salto en el caso contrario al de la comparacion
            }else {
                codigo.append("JBE LABEL" + operador + "\n \n"); // Salto en el caso contrario al de la comparacion
            }

            break;
        case "<":
            if(tipoOperandos.contains("INTEGER")||tipoOperandos.contains("OCTAL")) {
                codigo.append("JGE LABEL" + operador + "\n \n"); // Salto en el caso contrario al de la comparacion
            }else {
                codigo.append("JAE LABEL" + operador + "\n \n"); // Salto en el caso contrario al de la comparacion
            }

            break;
        case ">=":
            if(tipoOperandos.contains("INTEGER")||tipoOperandos.contains("OCTAL")) {
                codigo.append("JL LABEL" + operador + "\n \n"); // Salto en el caso contrario al de la comparacion
            }else {
                codigo.append("JB LABEL" + operador + "\n \n"); // Salto en el caso contrario al de la comparacion
            }
            break;
```

```
        case "<=":
            if(tipoOperandos.contains("INTEGER")||tipoOperandos.contains("OCTAL")) {
                codigo.append("JG LABEL" + operador + "\n \n"); // Salto en el caso contrario al de la comparacion
            }else {
                codigo.append("JA LABEL" + operador + "\n \n"); // Salto en el caso contrario al de la comparacion
            }
            break;
        case "=":
            codigo.append("JNE LABEL" + operador + "\n \n"); // Salto en el caso contrario al de la comparacion
            break;
        case "!=":
            codigo.append("JE LABEL" + operador + "\n \n"); // Salto en el caso contrario al de la comparacion
            break;
        }
}
```

Correcciones de códigos con errores

Código 1

lala
begin
 	
integer varx, vary, varx
varx := 327
varx := 2.3d+34;
tipo_abc vary;
 	
end
	Linea 5 Error: Falta ',' entre variables
	Linea 5 Error: Falta ';' al final de la sentencia
		Chequen cómo les está reconociendo las sentencias. Si dice que falta , es porque está tomando el varx de la segunda línea como parte de la primera, lo que les debería arrastrar otros errores.

Este error era erróneamente identificado ya que la asignación era considerada parte de la declaración de variables. Como se ve en la salida del código 1, sólo detecta la falta de ‘,’ entre variables y el ‘;’ al final de la sentencia.
Al analizar otros casos similares, descubrimos que esto ocurría cuando le continuaba una asignación, por lo tanto, se creó la siguiente regla.

```
asignacion | ASIGNACION expresion_arit
{cargarErrorEImprimirloSintactico("Linea " + AnalizadorLexico.saltoDeLinea +
" Error: Falta el operando izquierdo de la asignacion. ");}
;
```

Esta regla se reduce cuando existe una asignación con el operando de la izquierda faltante. Con la nueva versión obtenemos el siguiente resultado.

```
Linea 5 Error: Falta ',' entre variables
Linea 5 Error: Falta ';' al final de la sentencia
Linea 6 Error: Falta el operando izquierdo de la asignacion.
Linea 6 Error: Falta ';' al final de la sentencia
```

Código 2

lala
begin
 	
integer FUN juancito (double a) BEGIN
if (e < 4) then
 	
IF (e<3) THEN
RET (x);
ELSE
RET (a);
END_IF;
end_if;
END;
end
		Falta reconocer un error.

Este error era generado por las reglas gramaticales de las sentencias IF sin ELSE.
Al reconocer un IF simple con una sentencia de retorno dentro, se consideraba erróneamente que existía un retorno dentro de la función. Esto es incorrecto y debe existir una sentencia de retorno fuera del IF en estos casos.

Código 3

lala begin
 	
double a, b, c;
a := -2.0;
b := -3.0;
c := 4.0;
 	
outf(a);
outf(b);
 	
if (a < b) then
OUTF([a < b]);
else
OUTF([a >= b]);
end_if;
 	
if (a < b) then
OUTF([a < b]);
else
OUTF([a >= b]);
end_if;
 	
if (a = b) then
OUTF([a = b]);
else
OUTF([a != b]);
end_if;
 	
if (a != b) then
OUTF([a != b]);
else
OUTF([a = b]);
end_if;
 	
if (a >= b) then
OUTF([a >= b]);
else
OUTF([a < b]);
end_if;
 	
end
	2.00000000000000000000
	3.00000000000000000000
	a < b
	a < b
	a != b
	a != b
	a >= b
		Revisar.
		
		Revisar el caso también donde a y b son enteros.

Código 4

lala begin
 	
integer a, b, c;
a := -2;
b := 10;
c := 4;
 	
WHILE (a < b)BEGIN
a:=a+1;
outf(a);
END;
end
	3
	4
	5
	6
	7
	8
	9
	10
		Revisar.

A partir del código 3 y 4, se detectaron dos errores que eran generados por el mal manejo de las constantes negativas.
A la hora de convertir una constante a su equivalente negativo (método *getCopiaNeg()* de la clase Simbolo) , no era asignado el caracter '-' delante del Id del nuevo símbolo. Debido a esto, el operando que se extraía de la polaca no poseía el "signo" y era erróneamente interpretado como positivo.
Además del cambio anterior, fue necesario reemplazar el caracter '-' por '_' para evitar conflictos assembler.

Código 5

lala
begin
integer a,z;
 	
typedef triple <double> tsing;
 	
tsing s1, s2;
 	
tsing fun f(tsing b)
begin
 	
b{0} := 4.0;
b{1} := 40.0;
ret (b);
 	
end;
 	
s2 := f(s1);
 	
outf(s2);
end
	error A2006: undefined symbol : operando	

En la generación de código assembler existía un error en la sentencia responsable de cargar en la pila del coprocesador el operando leído.

Sentencia que generaba el error:

```
codigo.append("FLD  operando \n");
```

Sentencia corregida:

```
codigo.append("FLD " + operando2 + "\n");
```

Junto a esta solución, se descubrió que al usar la posición '0' de una tripla no era notificado el error indicando que estaba fuera de rango. Esto se debía a que se verificaba que la posición sea <= 3 pero no > 0.
En la nueva versión este código no compila y notifica el siguiente error.

```
Linea 11 Error:  La posicion 0 es invalida, se espera un valor entre 1 y 3.
```

Código 6

lala
begin
integer a, b, c, d;
a := 2;
 	
integer fun hola(integer f) begin
OUTF([En "hola" recibimos:]);
OUTF(f);
 	
integer e;
e := f * 3;
 	
integer fun chau(integer a) begin
OUTF([En "chau" recibimos:]);
OUTF(a);
 	
integer b;
b := a + 2;
 	
OUTF([En "chau" retornamos:]);
OUTF(b);
 	
ret(b);
end;
 	
OUTF([Retornamos el llamado a "chau" pasando:]);
OUTF(e);
ret(chau(e));
end;
 	
OUTF([Llamo a "hola" y me retorna:]);
OUTF(hola(a));
OUTF({FIN});
End
	Llamo a "hola" y me retorna:
	En "hola" recibimos:
	2
	Retornamos el llamado a "chau" pasando:
	6
	En "chau" recibimos:
	6
	En "chau" retornamos:
	8
	8
		Falta un print.

En este caso, la última sentencia OUTF({FIN}) no era impresa debido a que la cadena que se intentaba imprimir estaba contenida entre ‘{‘ ‘}’ , en lugar de ‘[‘ ‘]’.

Si bien es correcto que no haya sido impresa, fue necesario incluir una regla dentro de las sentencias OUTF para considerar estos casos particulares y notificar el error.

A continuación se mostrará la nueva regla gramatical.
outf_rule: OUTF '(' '{' error '}' ')' {*Error sintáctico*}

Al ejecutar nuevamente el código de prueba, se carga el siguiente error en el archivo de errores sintácticos:

```
Linea :33 Error : Cadena mal escrita en sentencia OUTF.
```