PRÁCTICA DE PROCESADORES DE LENGUAJES

Curso 2008 – 2009

Entrega de Septiembre

NOMBRE Y APELLIDOS: **DAVID MARTINEZ PEÑA**

DNI: **73.774.849-A**

CENTRO ASOCIADO: **ALZIRA-VALENCIA**

REALIZÓ SESIÓN DE CONTROL (Sí / No): **SI**

MAIL DE CONTACTO: **david@martinezpenya.es**

TELÉFONO DE CONTACTO: **629 047 152**

GRUPO (A ó B): **B**

Cambios realizados en los analizadores Léxico y Sintáctico

En el analizador Léxico creo que no he cambiado ninguna cosa, quizá algún detalle menor. En cuanto al sintáctico, he cambiado algunas reglas, que a nivel sintáctico eran iguales, pero no me permitían realizar un buen análisis semántico. Son un ejemplo de este caso las definiciones de campos de un registro, ya que a nivel sintáctico son muy parecidas (o iguales) a una declaración de variables, pero a nivel semántico hay que hacer otras tareas con los registros que no tienen nada que ver con las tareas que hay que hacer cuando se trata de variables.

El analizador Semántico

Aquí comenzó a complicarse la cosa, y me costó bastante arrancar. Al final aglutine las clases de los no terminales en las siguientes:

- **BloqueSentencias** (útil para la generación de código intermedio)
- **DefCamposRegistro** (según voy añadiendo campos a un registro, los voy metiendo en esta clase, y finalmente recorro los campos de esta clase para crear el Tipo Registro. Al principio esto lo hacía con una variable global, pero finalmente me pareció más elegante esta otra opción).
- **Expresión** (la uso para hacer comprobaciones de tipos)
- **ListaObjetos** (Es como una clase auxiliary que uso para juntar objetos del mismo tipos y hacerlos subir en el árbol sintáctico)
- **Par** (Parámetros de funciones)
- **ProgramModule** (Facilitada por el equipo docente)
- **Var** (Variables, clase auxiliar donde voy almacenando el tipo, nombre, linea y columna de la variable que estoy definiendo)

Descripción de la Tabla de Símbolos

Sobre la arquitectura que proporciona el equipo docente, mi tabla de símbolos contiene un scope (ya que cada uno de ellos tiene su propia tabla de símbolos) y un HashMap que almacena todos los símbolos para ese scope. Implementé todas las funciones que necesitaba en esta clase.

Cabe mencionar que tengo una clase de utilidades, en donde implementé funciones para devolver una Lista a partir de un HashMap (*`getListFromHashMap`*). Dos funciones para buscar Tipos en toda la pila de scopes (y no solo en el actual) (`containsType` y `searchType`). Así como un método para calcular el tamaño que ocupa en memoria un tipo de datos pasado por parámetro (`BuscaTamanyoTipo`).

Generación de Código Intermedio

En esta fase se genera código intermedio, independiente del hardware. El tipo de código intermedio es el de tres direcciones, formando cuadruplas con el formato:

OPERACION, RESULTADO, OPERADO1, OPERADOR2

Las palabras clave usadas para las operaciones han sido:

ADD, para las sumas

ASIG, para las asginaciones

BN, salto si negativo

BR, salto incondicional

BZ, salto si es cero

BNZ, salto si no es cero

CMP, comparaciones

DEV, sentencias devuelve

DIV, divisiones

ESCRIBE, muestra por pantalla texto

ESCRIBEENT, muestra por pantalla enteros

ESCRIBESAL, muestra por pantalla saltos de linea

ETIQUETA, etiquetas para los saltos

FIN, fin del programa

FINFUNCION, fin de la declaracion de funciones

FUNCION, comienzo de la declaracion de funciones

INICIO, inicio del programa

LLAMA, llamada a función

NOP, ninguna operación

PAR, parámetro para llamada a funcion

PARAMETROS, comienzo de parámetros para la función

Ejemplo de un programa en ñ:

```
/*  8.n
// Precedencia de operadores
*/

vacio principal () {

entero a;
entero b;
b = 19;
a = b++;
a= a+1+6/2;

escribe("a(24):");
escribeEnt(a);

a = ((3+3)/2);

escribe("asociacion_parentesis(3):");
escribeEnt(a);

devuelve;
}
```

El mismo programa en código intermedio:

```
****************************************
**           CODIGO INTERMEDIO            **
****************************************
** Cuadruple - [INICIO 0, null, null]
** Cuadruple - [FUNCION .Lprincipal0, null, null]
** Cuadruple - [ASIG b, 19, null]
** Cuadruple - [ADD .T0, b, 1]
** Cuadruple - [ASIG b, .T0, null]
** Cuadruple - [ASIG a, b, null]
** Cuadruple - [ADD .T1, a, 1]
** Cuadruple - [DIV .T2, 6, 2]
** Cuadruple - [ADD .T3, .T1, .T2]
** Cuadruple - [ASIG a, .T3, null]
** Cuadruple - [ESCRIBE "a(24):", null, null]
** Cuadruple - [ESCRIBEENT a, null, null]
** Cuadruple - [ADD .T4, 3, 3]
** Cuadruple - [DIV .T5, .T4, 2]
** Cuadruple - [ASIG a, .T5, null]
** Cuadruple - [ESCRIBE "asociacion_parentesis(3):", null, null]
** Cuadruple - [ESCRIBEENT a, null, null]
** Cuadruple - [FINFUNCION .Lprincipal0, null, null]
** Cuadruple - [FIN null, null, null]
****************************************
```

Descripción de la estructura utilizada

La estructura utilizada es la proporcionada por el equipo docente, en cada BloqueDeSentencias tengo un atributo que se llama codigointermedio, donde voy acumulando el código y pasándolo hacia arriba en el árbol sintáctico.

Generación de Código Final

Aquí se produce la traducción de las cuádruplas generadas en el paso anterior a código ensamblador para ENS2001.

Lo primero es colocar todo el código fuente y las variables globales en la memoria. Y a partir de aquí generar el registro de activación de principal, y a continuación todos los que se vayan llamando desde el código fuente.

Todas las cadenas de texto se colocan al final de código ensamblado.

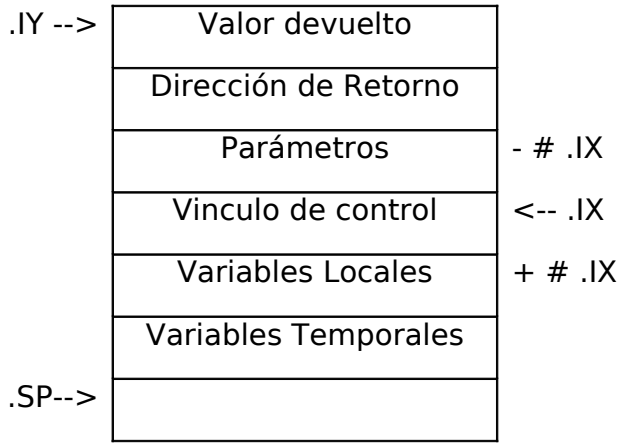
El mismo programa en código intermedio:

```
; Código ensamblado de ñ para ens2001
      MOVE #65535, .SP
      MOVE .SP, .IY
      SUB .SP, #1
      MOVE .A, .SP
; Asignamos #19 a #-3[.IY]
      MOVE #19, #-3[.IY]
; Suma #-3[.IY] a #1 y lo guarda en #-0[.IY]
      ADD #-3[.IY], #1
      MOVE .A, #-0[.IY]
; Asignamos #-0[.IY] a #-3[.IY]
      MOVE #-0[.IY], #-3[.IY]
; Asignamos #-3[.IY] a #-2[.IY]
      MOVE #-3[.IY], #-2[.IY]
; Suma #-2[.IY] a #1 y lo guarda en #-5[.IY]
      ADD #-2[.IY], #1
      MOVE .A, #-5[.IY]
; Divide #6 entre #2 y lo guarda en #-6[.IY]
      DIV #6, #2
      MOVE .A, #-6[.IY]
; Suma #-5[.IY] a #-6[.IY] y lo guarda en #-7[.IY]
      ADD #-5[.IY], #-6[.IY]
      MOVE .A, #-7[.IY]
; Asignamos #-7[.IY] a #-2[.IY]
      MOVE #-7[.IY], #-2[.IY]
; Escribimos en pantalla la cadena txt1
      WRSTR /txt1
; Escribimos en pantalla #-2[.IY]
      WRINT #-2[.IY]
; Suma #3 a #3 y lo guarda en #-8[.IY]
      ADD #3, #3
      MOVE .A, #-8[.IY]
; Divide #-8[.IY] entre #2 y lo guarda en #-9[.IY]
      DIV #-8[.IY], #2
      MOVE .A, #-9[.IY]
; Asignamos #-9[.IY] a #-2[.IY]
      MOVE #-9[.IY], #-2[.IY]
; Escribimos en pantalla la cadena txt2
      WRSTR /txt2
; Escribimos en pantalla #-2[.IY]
      WRINT #-2[.IY]
; Fin del Código ensamblado de ñ para ens2001
      HALT
```

```
; Cadenas empleadas para mostrar por consola
      txt0:          DATA "\n"
      txt1:          DATA "a(24):"
      txt2:          DATA "asociacion_parentesis(3):"
```

Descripción del Registro de Activación

El registro de activación tiene los siguientes campos:



Conclusiones

Este es el tercer curso que realizo esta asignatura. Es el primer año que llego al segundo semestre. Tal y como se advierte en el documento de la práctica, esta segunda parte es todavía más larga que la primera. Complicándose sobremanera hasta que no tienes asentados en la cabeza los conceptos teóricos.

También me gustaría agradecer desde aquí a los profesores el esfuerzo que han realizado para ofrecernos un documento de directrices, que se echó mucho de menos hace dos años. Además de todas las funciones que nos han ahorrado (para mostrar mensajes de debug, errores, etc.) y que hace dos años corrían por nuestra cuenta y la verdad es que nos apartaban un poco del verdadero sentido de la práctica que era construir un compilador. Este año todos estos conceptos estaban mucho más claros que el año pasado.