Procesadores de lenguajes.

Manuel Montero

Besay Montesdeoca

DEFINICIÓN DEL LENGUAJE

Características generales

- Para delimitar las estructuras se utilizan las llaves y los puntos y comas con el objetivo de conocer el fin de una sentencia.
  - `F b = 1.2;`
- Debe existir una función principal con el nombre "ZOP" la cual será el punto de inicio de la ejecución del programa. Dicha función debe estar al final de todas, puesto que ZOP solo puede hacer llamadas a funciones que estén previamente declaradas.

- Las funciones deben estar declaradas de la siguiente forma:
  - `#nombreFuncion:tipo() { ... };`
- Las variables declaradas fuera de cualquier función y al inicio del código serán tomadas como variables globales.

Codificación de las variables

Las variables tienen tipado.

Las variables pueden contener números reales y enteros utilizando las etiquetas "I" y "F". Adicionalmente existen otros tipos embebidos en el lenguaje, como los booleanos y las string, que pueden ser usados directamente en los prints o en las condiciones de las estructuras de control.

```
I a = 8;
F b = 1.2;
```

Comentarios

Comentario en una línea o al final de una línea: se introduce con el símbolo //

Estructuras condicionales.

- If → Para construirla basta con escribir la condicion despues de la palabra "if" separada por un espacio y con o sin paréntesis.
- else → Cubre todas las condiciones restantes.

```
if 5<2{
   code... }
else{
   code… }
```

Estructuras repetitivas.

- Loop → La estructura loop cumple las funciones de "while" y "for" de otros lenguajes.
    1. La estructura loop tiene 3 parámetros:
        a. Inicialización de iterador (Opcional).
            i. Para inicializar una variable se declara su nombre seguida de su valor al inicio del bucle (Ejem: i=0)
        b. Condición (Obligatoria).
            i. Condición de parada.
        c. Incremento de variables (Opcional).

```
loop I i=2, i < 1000, i += 3  { code… }
loop ,n<10,  { code… }
```

Impresión por pantalla

Para mostrar el contenido de una variable o para imprimir cualquier cadena de texto simplemente basta con escribir "print "seguido del nombre de la variable o la cadena de texto sin asignarle ninguna operación de asignación ni aritmética.

```
I a = 8;                          Consola:
print a;                           8
print "Fin de la ejecución";      Fin de la ejecución
print "He \"terminado\"";         He "terminado"
```

Funciones

Para distinguir una función se utiliza el símbolo "#", seguida del nombre de la función y a continuación el símbolo ":" y el tipo, a continuación entre paréntesis los parámetros separados por coma que necesita. Las funciones podrán devolver ningún o un dato. Si es una función que no devuelve nada:

```
#sumaResta:I(I numero1, I numero2){
	I suma = numero1 + numero2;
	return suma;
}
```

EJEMPLO

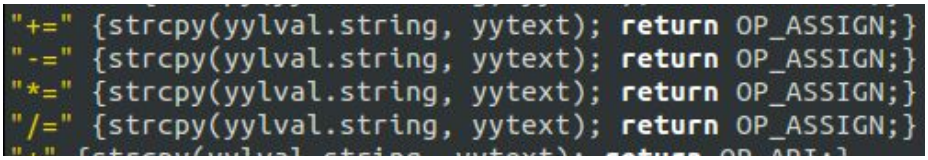
A continuación se muestra el código de la sucesión de fibonacci hecha en ZOP.

```
I max = 20;
#ZOP(){
	I num1 = 1;
	I num2 = 1;
	print num1;
	print num2;
	//Fibbo
	I result = 0;
	loop , max > 1, {
		result = num1 + num2;
		print result;
		num1 = num2;
		num2 = result;
		max -= 1;
	}
	print "end\n";
}
```

ANALIZADOR LÉXICO

En primer lugar fue necesario desarrollar un analizador léxico, un programa que recibe como entrada recibe el código fuente de otro programa y produce una salida compuesta de tokens o símbolos. Estos tokens servirán para una posterior etapa del proceso de traducción, siendo la entrada para el analizador sintáctico. La herramienta que se utilizó para esta labor fue Flex, esta herramienta lee los ficheros de entrada dados, o la entrada estándar si no se le ha indicado ningún nombre de fichero, con la descripción de un escáner a generar. La descripción se encuentra en forma de parejas de expresiones regulares y código C, denominadas reglas. Flex genera como salida un fichero fuente en C, lex.yy.c, que define una rutina yylex(). Este fichero se compila y se enlaza con la librería `-lfl' para producir un ejecutable. Cuando se arranca el fichero ejecutable, este analiza su entrada en busca de casos de las expresiones regulares. Siempre que encuentra uno, ejecuta el código C correspondiente.

En nuestro caso algunos patrones devuelven el mismo identificador por lo que para un correcto entendimiento con el futuro analizador sintáctico es necesario guardar el patrón de forma literal como “string”, variable que el analizador utiliza para distinguirlos.

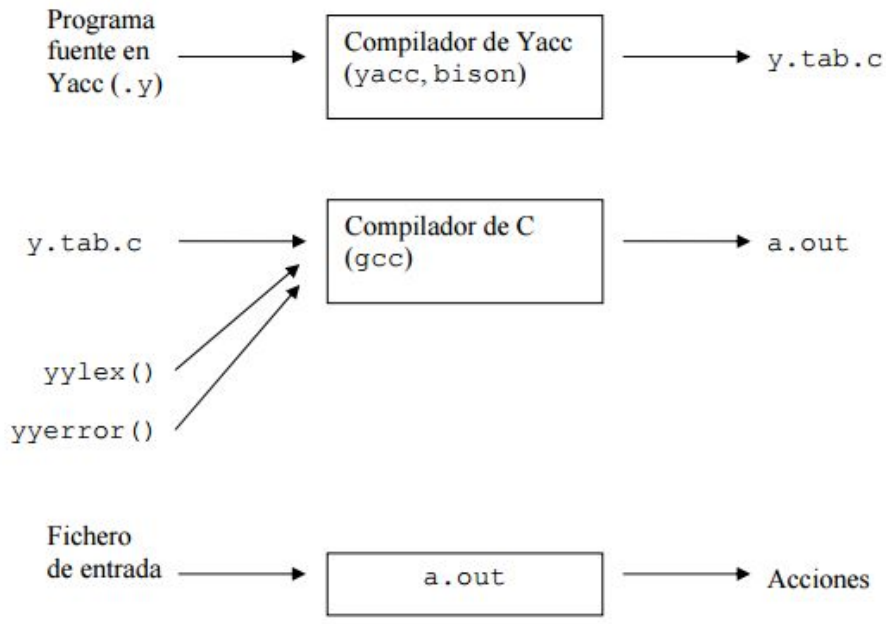


ANALIZADOR SINTÁCTICO

Hemos visto cómo el análisis léxico facilita la tarea de reconocer los elementos de un lenguaje uno a uno. En segundo lugar no centramos en el análisis sintáctico, que nos permitirá averiguar si un fichero de entrada cualquiera respeta las reglas de la gramática que hemos construido. Para el análisis sintáctico hemos utilizado la herramienta yacc (Yet Another Compiler Compiler).

Igual que sucedía con lex, yacc no es directamente un analizador sino un generador de analizadores. A partir de un fichero fuente en yacc, se genera un fichero fuente en C que contiene el analizador sintáctico. Sin embargo, un analizador sintáctico

de yacc no puede funcionar por sí solo, sino que necesita un analizador léxico externo para funcionar. Dicho de otra manera, el fuente en C que genera yacc contiene llamadas a una función yylex() que debe estar definida y debe devolver el tipo de lexema encontrado en la etapa anterior. Además, es necesario incorporar también una función yyerror(), que será invocada cuando el analizador sintáctico encuentre un símbolo que no encaja en la gramática.



Una analizador sintáctico está compuesto por reglas, las cuales definen cómo es una gramática válida en el lenguaje. En el caso de ZOP existen 24 reglas:

1. **program:** Define el como es el cuerpo del programa.
2. **otherFunc:** Es una regla recursiva, que añade reglas **function**.
3. **var_global:** Es una regla recursiva, que añade variables globales.
4. **main_function:** Define la estructura de la función de inicio.
5. **function:** Define la estructura de cualquier función.
6. **block:** Define un bloque de instrucciones.
7. **sentences:** Es una regla recursiva, que añade reglas **sentence**.
8. **args:** Define los parámetros de una función.
9. **sentence:** Define una instrucción.
10. **init:** Define una inicialización de variable.
11. **assign:** Define una asignación de variable.
12. **if:** Define la estructura "if".
13. **else:** Define la estructura "else".
14. **loop:** Define la estructura "loop".
15. **break_continue:** Define los "breaks" y "continues" de los bucles.
16. **return:** Define los "return" de las funciones.
17. **print:** Define la muestra por pantalla.
18. **expression:** Define expresiones aritméticas posibles en el lenguaje.
19. **expression_bin:** Define expresiones condicionales.
20. **const:** Define constantes, números, reales, strings …
21. **numeric:** Define constantes solo de números y reales.
22. **callFunction:** Define una llamada a función.
23. **param:** Define los parámetros de una llamada a función.
24. **var:** Define tipos de variables, números enteros o reales.

TABLA DE SÍMBOLOS

La tabla de símbolos es una estructura de datos que usa el proceso de traducción de un lenguaje de programación, por un compilador o un intérprete, donde cada símbolo en el código fuente de un programa está asociado con información tal como la ubicación, el tipo de datos y el ámbito de cada variable, constante o procedimiento.

Nuestra implementación de la tabla de símbolos se puede encontrar en los ficheros
"symtable.cpp" y "symtable.h", en donde están todas las funciones necesarias para introducir en la tabla los símbolos requeridos.

Nuestra tabla posee una estructura que guarda la siguiente información para cada variable y función que encuentre:

1. Nombre de la variable o función.
2. Tipo del símbolo (variable o función).
3. Tipo de la variable o tipo devuelto (entero o real)
4. Posición en memoria
5. Ámbito.

Además la tabla tiene funciones para la gestión de la misma:

```
int getHash(string);
symtable();
string generanameVar();
symrec * insertVar(string name, vartype typeVar, int amb, int mem);
symrec * insertFun(string name, vartype typeVar, int mem, int label);
void newScope();
void endScope();
bool existSymbol(string name);
symrec * getSymbol(string name);
bool checkEnum (string variable, string campo);
bool existParam(symrec * s, string name);
const list<symrec> * const tableHash() { return table; };
int scope() { return currentScope; };
int scopeSymbol(string name);
void printState();
void printSymbol(symrec * s);
```

Adicionalmente existe una variable global "DEBUGGING" que permite ver una traza por consola si esta está a "true".

GENERACIÓN DE CÓDIGO

La generación de código es una de las fases mediante el cual un compilador convierte un programa sintácticamente correcto en una serie de instrucciones a ser interpretadas por una máquina.
En este caso las instrucciones se codifican en lenguaje Q.
Para la generación de código se debe traducir cada regla y porción de código ZOP en lenguaje Q, para ello, a medida que vayamos recorriendo nuestro código iremos generando el código Q equivalente.
Para la generación del código Q tenemos dos ficheros con las funciones encargadas de hacerlo, “codeGenerationQ.c” y “codeGenerationQ.h”.

Básicamente cada función de “codeGenerationQ” transforma un trozo de código ZOP analizado por una regla en el codigo Q correspondiente.
A continuación se muestra de ejemplo cómo se traduce la función main:

Regla de la función main (con la llamada a generateCodeMain)

```
main_function :
FUNC MAIN {generateCodeMain(fout, &Q, nFunction);
table.insertFun("ZOP", type, 0, 0);} PARL PARR
{if (trace) printf("%d :: <main_function>  FUN MAIN PARL PARR block \n",numlin);}
{table.newScope();}
block
{table.endScope();};
```

Traducción a código Q.

```
void generateCodeMain(FILE* fout, qMachine *Q, int nFunction){
    fprintf(fout,"L 1:\t\t\t//Start Main\n");
    fprintf(fout,"\tR7 = %d;\t\t\t//Align memory\n", Q->Ztop);
    for(int i = 0; i < nFunction; i++){
        fprintf(fout,"\tR7 = R7 - 8;\t\t\t//reserve space function\n");
    }
}
```