Procesadores de lenguajes

Trabajo de curso

Creación de un lenguaje y su compilador a código Q. Lenguaje creado: “Atomic Kitty”.

Gabriel Jiménez Perera - 44745858W

Alberto Casado Garfia - 45352480E

CONTENIDO

DEFINICIÓN DEL LENGUAJE

INTRODUCCIÓN

El lenguaje que hemos pensado está basado en Python y en C, principalmente. Pretendemos la sencillez del C puro con la comodidad de Python, obligando a que el código quede bien organizado mediante el uso del tabulado para especificar bloques de instrucciones. También queremos incorporar nuestra versión de las tuplas de Python, dejando de lado los vectores de C. En cuanto al tipado, será de carácter fuerte.

Tipos primitivos del lenguaje: int, long, float, double, char, string, bool.

Las funciones pueden retornar void y se pueden crear nuevos tipos con tuplas.

Otras palabras reservadas: and, or, if, elif, else, in, not, is

NOTAS SOBRE TIPOS

CONVERSIÓN DE TIPOS

Entre distintos tipos no se permiten conversiones implícitas y las conversiones explícitas se realizarán mediante el cast de C, solo que utilizando llaves en lugar de paréntesis *{tipo-destino} variable*.

STRING

Las ristras de caracteres no tendrán un tamaño máximo y se escribirán entre comillas dobles. Se permitirá la concatenación de cadenas de caracteres mediante el operador '+'.

TUPLAS

Las tuplas son elementos híbridos entre las tuplas de Python y los vectores de C. Pueden tener varios tipos predefinidos en su interior. No se permite modificar sus elementos.

ACCESO A SUS ELEMENTOS

Por índice como los vectores de C. El índice deberá expresarse como un número entero entre corchetes después del identificador de la tupla. El índice no podrá ser una expresión sino un número directamente.

```
(int, int) tupla1 = (1,2)
(int, char)tupla2 = (1,'a')
(char, int) tupla3 = (tupla2[1], tupla2[0])
int m = 1
int n1 = tupla1[0]    #Permitido
int n2 = tupla2[0]    #Permitido
int n3 = tupla1[m]   #No Permitido
¿? n4 = tupla2[m]    #No permitido
```

DECLARACIÓN Y ASIGNACIÓN

```
(tipo1, tipo2,...) nombre = (expresión que devuelve tipo1,...)
(int, int) tupla = (0,1)
(int, string) tupla2 = (3, "hola") #Strings no implementadas
(string, int, float) tupla3 = ("mundo", 4, 0.5) #Strings no implementadas
(string, (int, float)) tupla4 = (tupla3[0], (4, 0.5))

tupla[0] = 3 #No permitido, las tuplas no pueden modificarse
```

OPERADORES NUMÉRICOS

Se utilizarán los operadores básicos para número de C: +, -, *, /. También se utilizarán los comparativos <, >, <=, >=. Para comprobar la igualdad se utilizará la palabra reservada "is".

Precedencia de operadores de forma parecida a C. Los que se encuentren en el mismo nivel de precedencia se interpretarán por orden de izquierda a derecha. De mayor a menor, los órdenes de precedencia serían:

- Expresiones entre paréntesis.
- Multiplicaciones y divisiones.
- Sumas y restas.
- Comparaciones (>, <, >=, <=).
- Operadores de igualdad (is, not is).
- Operadores de contenido (in, not in).
- Operadores lógicos (and, or).

De forma especial, el operador de cast (que utilizan llaves), tendrá la precedencia máxima.

DECLARACIÓN DE VARIABLES

La declaración de variables se realizará de manera muy parecida a Python, solo que indicando el tipo de la variable:

```
tipo nombre = expresión
nombre = expresión
int i = 0
i = 1
float n = 2.0
double d = 2.0
long l = 20
string s = "string"
bool b  = true
char c = 'c'
```

ASIGNACIÓN DE EXPRESIONES

```
tipo nombre = expresion
int i = 1+2+(3 + 4)*5/7          (i: 8)
```

FUNCIONES DEL LENGUAJE

RANGOS

i..j -> Representa un rango de valores similar a [i, j).

i..j,k -> Similar a la función anterior, pero el incremento sería de k en k. i y j pueden ser expresiones numéricas o caracteres, pero k solo puede ser una expresión numérica.

IMPRESIÓN

print(char) -> Función que imprime por pantalla un carácter.

print(int) -> Función que imprime por pantalla un número entero.

ESTRUCTURAS DE CONTROL

ESTRUCTURAS CONDICIONALES

IF ELIF ELSE

Utilizaremos estructuras condicionales simples muy parecidas a C, pero sin la necesidad de paréntesis para la condición.

```
if condicion1:
    cosas1()
elif condicion2 or condicion3:
    cosas2()
else:
    cosas3()
```

WHEN

Para la estructura tipo switch, implementaremos nuestra versión del when de Kotlin. Para cada caso solo se permitiría una instrucción y los casos pueden ser negados con el operador not.

when expresión:

x -> instrucción

…

else -> instrucción

Se ejecutaría la primera instrucción que cumpla x, donde x puede ser:

- Una lista de valores (con al menos un elemento) que se compara con el resultado de la expresión. Si alguno de los valores de la lista es igual *(is)* al resultado de la expresión, se ejecuta la instrucción indicada.
- Una estructura tipo *in* para indicar un rango de valores de tipo *m..n*, por lo que si el *resultado de la expresión >= m and resultado de la expresión < n*, se ejecuta la instrucción correspondiente.

```
when 2+3:
    0, 1 -> cosas1()
    in 2..10 -> cosas2()
    not in 10..20 -> cosas3()
    else -> cosas4()
```

Estructura equivalente en nuestro lenguaje:

```
if (variable is 0 or variable is 1){
     cosas1();
} else if (variable >= 2 and variable < 10){
     cosas2();
} else if (!(variable >= 10 and variable < 20)){
     cosas3();
} else {
     cosas4();
}
```

ESTRUCTURAS DE BUCLE

FOR

Bucle for, se utiliza declarando o reutilizando una variable tipo int a la que se le asignará los valores dados por la función de rango especificada, por tanto, si la variable utilizada existe se utilizará modificando su valor y, si no existe, se creará una variable nueva tipo entero que se eliminará al terminar el bucle. Además, podrán añadirse condiciones con un and:

```
for i in 3..7, 2:
  cosas()
```

WHILE

Bucle while tipo C con la notación del lenguaje:

```
while condicion:
    cosa1()
    cosa2()
```

FUNCIONES

DEFINICIÓN

Las funciones se definirán como en Python, pero con el tipo como en C:

```
tipoDevuelto nombreFuncion(int p1, int p2):
  cosas()
```

Como parámetros y como resultado, se pueden pasar y devolver cualquier tipo primitivo del lenguaje. Además, puede no devolverse nada si se indica como tipo de resultado *void*.

LLAMADA A FUNCIONES

Las llamadas a funciones se realizan indicando el identificador de la función, seguido de paréntesis, entre los que se podrán pasar los argumentos. El paso de parámetros y la devolución de resultados se realizarán por valor, pues se copiarán los valores de las variables o expresiones en ese momento.

Además, se permite la recursividad, pues desde dentro de una función se puede llamar a sí misma.

EJEMPLO

```
(float, float) funcion((int, int) p):
    cosas()

void función(int p):
    cosas()
```

CÓDIGO FUERA DE LAS FUNCIONES

Al igual que en Python se comenzará a ejecutar el código que no esté dentro de ninguna función. Las variables que no se declaren dentro de una función tendrán ámbito global. Las variables declaradas dentro de una función tendrán un ámbito solo dentro de dicha función. De igual forma, las variables declaradas dentro de un bloque (while, for…) tendrán ámbito solo dentro de dicho bloque.

Se permitirá intercalar código global y declaración de expresiones.

EJEMPLO DE FUNCIONAMIENTO DEL LENGUAJE

FIBONACCI

CÓDIGO

```
(int, int) fibo(int n):
    if n is 0:
        return (0, 0)
    elif n is 1:
        return (1, 0)
    else:
        (int, int) tupla = fibo(n-1)
        int i = tupla[0]
        int j = tupla[1]
        return (i+j, i)

print(fibo(10)[0])
```

EXPLICACIÓN

```
fibo(6)->(8,5)
fibo(5)->(5,3)
fibo(4)->(3,2)
fibo(3)->(2,1)
fibo(2)->(1,1)
fibo(1)->(1,0)
```

TRIÁNGULO DE TARTAGLIA

CÓDIGO (3 VERSIONES)

```
int tritar(int m, int n):
    if n is 0 or n is m:
        return 1
    else:
        return tritar(m-1, n-1) + tritar(m-1, n)

int tritar2((int, int)n):
    int n0 = n[0]
    int n1 = n[1]
    if n1 is 0 or n1 is n0:
        return 1
    else:
        return tritar(n0-1, n1-1) + tritar(n0-1, n1)

int tritarTupla((int, int)n):
    if n[1] is 0 or n[1] is n[0]:
        return 1
    else:
        return tritar(n[0]-1, n[1]-1) + tritar(n[0]-1, n[1])

for i in 0..11:
    print(tritar(10, i))
    print('\n')
    print(tritar2((10, i)))
    print('\n')
    print(tritarTupla((10, i)))
    print('\n')
    print('\n')
```

PROCESADOR LÉXICO

Como herramienta para crear el analizador léxico, se utilizó *Flex*, que permite generar analizadores léxicos que devuelven *tokens* a partir de expresiones regulares. Para ello se definen expresiones regulares y, mediante combinaciones de las mismas con otros caracteres, se devuelven los *tokens* correspondientes.

De manera especial, debido a que utilizamos el tabulado para indicar los bloques de instrucciones, hemos tenido que utilizar la opción *YY_USER_ACTION* de Flex, tal y como se muestra en el siguiente código:

```
#define YY_USER_ACTION                                          \
{                                                               \
  if (tabulado<numtabs){                                        \
    yyless(0);                                                  \
    numtabs--;                                                  \
    return CIERRABLOQUE;                                        \
  } else if (tabulado>numtabs){                                 \
      numtabs++;                                                \
    yyless(0);                                                  \
    return ABREBLOQUE;                                          \
  }                                                             \
}                                                               \
```

esto, combinado con el patrón de reconocimiento de tabulados:

```
^{TABULADOR}+    {
                     int i;
                     int tabsEncontrados = 0;
                     for (i = 0; yytext[i] != '\0'; i++){
                         if (yytext[i] == ' '){
                             i += 3;
                         }
                         tabsEncontrados++;
                     }
                     tabulado = tabsEncontrados;
                 }
```

nos permite devolver cuantos tokens de tipo *ABREBLOQUE* y *CIERRABLOQUE* correspondan. Esta dificultad añadida se debe a que *Flex* está construido con el propósito de devolver *tokens* según conjuntos de caracteres y su contexto, pero para abrir y cerrar bloques no contemplamos ningún carácter específico en la gramática, sino la diferencia de tabulados entre líneas.

ANALIZADOR SINTÁCTICO

Para la construcción del analizador sintáctico, hemos utilizado *Bison*, que es una herramienta pensada para este propósito. *Bison* es un reconocedor *bottom-up* que lee el fichero fuente de izquierda a derecha y utiliza la técnica *shift-reduce* para la construcción del árbol sintáctico. Con esta herramienta podemos definir el orden de operadores e, indicando nuestra gramática, se encargará de reconocer los diferentes tokens e irlos reduciendo a símbolos no terminales hasta culminar en el nodo raíz del árbol sintáctico.

Además, definimos los *tokens* a utilizar en nuestra gramática desde *Bison*, por lo que primero tenemos que generar un *header* de *Bison* que será llamado desde el fichero de *Flex*, para poder utilizar ambos de una forma fusionada.

ANALIZADOR SEMÁNTICO

Nuestro analizador semántico se compone del analizador sintáctico creado anteriormente con *Bison* modificado con otras estructuras propias que permiten el chequeo de tipos y nombres.

Desde *Bison* utilizamos nuestra implementación de la tabla de símbolos en C++, que no es más que una clase que utiliza unas estructuras internas y funciones para facilitar el uso de la misma, implementando el concepto de ámbito. La estructura principal donde se almacenan los diferentes

nombres de variables y funciones es un mapa, de clave *std::string* y valor *Node*, donde Node es otra clase propia de la que derivan *FunctionNode*, *ParameterNode* y *VariableNode*. Esta construcción nos permite crear los diferentes ámbitos y añadirles sus elementos de diferentes tipos. Además, un ámbito contiene un puntero al ámbito padre, lo que permitirá acceder a sus elementos.

GENERACIÓN DE CÓDIGO

El último paso en la construcción de nuestro compilador sería la generación de código. Este se genera desde el propio analizador semántico y sin optimizaciones. Por tanto, la generación de código y el análisis léxico-semántico se realizan de forma simultánea, pues según se van detectando estructuras de tokens se analiza la validez léxico-semántica de las mismas y se genera el código correspondiente.

La generación de código está separada en 2 partes principalmente, la clase MemManager que controla el uso de los registros y el stack y además generar código relacionado con la memoria y las funciones de CodeGeneration en las que se genera código más especifico. También existe generación de código en el archivo de bison miint.y.

Para escribir el código en el archivo se usa el objeto gc de la clase std::ofstream que usa el operador << para añadir texto al archivo. El código ensamblador se escribirá en un fichero llamado program.q que se creará en la misma carpeta donde se sitúe el compilador.

Al inicio de nuestro código compilado, aparece la inclusión del header de Q y los defines que se corresponden con los registros enteros 6 y 7. En la sección del programa, que se encuentra entre BEGIN y END, empezamos inicializando R6 con el valor de R7 y guardamos el valor de las cadenas de caracteres en memoria. Una vez hemos guardado estos datos en memoria, empezamos con el código ejecutable correspondiente a las instrucciones de nuestro lenguaje, terminándolas con la salida de ejecución, saltando a la etiqueta -2.

Atomic Kitty permite escribir código en el ámbito global que se ejecutará en el comienzo del programa. Este código se interpretará hasta que el compilador lee el comienzo de una función y añadirá un salto hacia el final de la función para que el programa siga ejecutando las instrucciones que se encuentran detrás de ella y evitar que ejecute las instrucciones de la función.

Las variables declaradas en el ámbito global pueden ser accedidas desde el interior de una función, por lo que debemos recordar su posición en el stack. Es por ello por lo que MemManager tiene 2 stacks, uno para el ámbito global y otro para el local. También es necesario almacenar el estado de los registros.

Los datos guardados en la memoria están codificados en la estructura StackElement que principalmente guarda el tipo de dato, un nombre que será usado para depurar y un identificador. Este identificador permitirá determinar una variable o expresión y calcular su posición en memoria, por lo que es retornada en cada expresión en bison usando $$ y guardada en la tabla de símbolos para cada variable. El acceso a un subelemento de una tupla se realizará obteniendo su dirección en el stack y sumándole el desplazamiento del subelemento.

En las llamadas a funciones el compilador producirá código para salvar los registros usados en el stack, luego guardará el registro R6, seguido de la etiqueta a la cual retornará el programa al finalizar la función. Después guardará los parámetros, ajustará R6 y saltará a la etiqueta de comienzo de la función. En la función llamada se asumirá que los datos están en el stack local. Luego al retornar el valor, se recogerá la etiqueta de retorno y guardará el valor a retornar sobrescribiendo la etiqueta. Finalmente saltará a la etiqueta cargada. Al retornar se cargará el valor antiguo guardado de R6, los

registros y el valor retornado. Como la llamada a función es una expresión, se retornará el nuevo id del valor retornado.

PROBLEMAS DE IMPLEMENTACIÓN

Debido a que nos llevaría demasiado trabajo para el poco tiempo disponible, hemos decidido no implementar la aritmética en punto flotante, las cadenas de caracteres ni la conversión de tipos. Esto quiere decir que se analizará la entrada y se generará código, pero que este no será correcto y dará un error de ejecución.

En cuanto a la aritmética en punto flotante, finalmente no se podrán hacer operaciones con tipos float o double, aunque se aceptarán en la gramática.

En cuanto a las cadenas de caracteres, se aceptan en la gramática, pero en el código no se implementan, por lo que no se podrán hacer operaciones de string como la impresión o la concatenación (se permite la impresión de caracteres y enteros).

En cuanto a la conversión de tipos, no se permite desde el procesamiento léxico, pues teníamos dudas de cómo diseñarlo, pues queríamos buscar soluciones más implícitas, pero nos faltó tiempo.

CONCLUSIONES

Con este trabajo de curso hemos visto el proceso completo de compilación de un lenguaje, desde los análisis previos necesarios hasta la generación de código. Además, hemos utilizado algunas de las herramientas más comunes para este proceso, viendo las diferentes dificultades a las que hay que enfrentarse para compilar un lenguaje.

Destacamos las dificultades que hemos tenido, pues algunos procesos relativamente sencillos nos han llevado mucho más tiempo del esperado. Está claro que la generación de un compilador no es una tarea sencilla y hay muchos factores a tener en cuenta, lo que nos ha hecho enfrentarnos a varios problemas y a tener que buscar soluciones a los mismos.