



COMPILADORES
GRADO INGENIERÍA INFORMÁTICA, CURSO 23/24

COMPILADOR EN MINI C

CONVOCATORIA JULIO 2024

Hecho por

Irene Moreno Cegarra – i.morenocegarra@um.es

Daniel Aliaga Sánchez – daniel.a.s @um.es

Grupo 1.1 y 1.2, respectivamente
Profesor: Manuel Gil Pérez

12/05/2024

Índice

1. Análisis léxico.	3
2. Análisis sintáctico.	4
3. Análisis semántico y generación de código.	6
4. Cambios realizados.	8



1. Análisis léxico.

El analizador léxico debe identificar los diferentes elementos léxicos(tokens) del código a analizar, para ello hemos definido las expresiones regulares pertinentes para abarcar todos estos tokens, y cuando el analizador encuentre un token lo devolverá mediante la función `yylex()` en forma de número. En general todas son expresiones regulares sencillas sin mayor interés, las más interesantes son: los comentarios multilínea, para poder implementarlo necesitamos hacer uso de una herramienta que no ofrece flex, que nos da la capacidad de "cambiar de contexto" respecto a las demás expresiones regulares, como si utilizásemos otro autómata diferente. Esto es necesario para que el analizador sepa donde acaba el comentario, por ejemplo:

```
"/**"(.*)"**" //Si esta fuese la implementación de los comentarios:
```

Entrada:

```
/*Comentario
Multilinea
1*/
```

```
var x;
```

```
/*Comentario
Multilinea 2*/
```

Salida correcta:

```
Token VAR
Token IDEN
```

Salida incorrecta:

```
//salida vacia
```

En este caso el analizador tomaría como comentario todo lo que hay entre el primer `/*` (de la primera línea) hasta el último `*/` (de la última línea) puesto que la expresión regular indica que es todo lo que haya entre dos `/* */`, este debería dejar de reconocer un comentario en el primer `*/` que encontrase, pero al funcionar el analizador de manera voraz llega hasta el último `*/`, siendo esta implementación incorrecta.

Véase el correcto funcionamiento de los comentarios multilínea en prueba1(entrada y salida) en el directorio Pruebas_lexico/ficherosDeEntradaYSalida; En el modo pánico, para poder recuperarse de errores el analizador necesita ignorar la entrada cuando ocurra un error hasta que encuentre con el siguiente token, por tanto, si el siguiente carácter en la entrada no pertenece a ningún token este debe ser ignorado para poder continuar con el análisis, aunque el fichero ya se

consideraría erróneo y no se generará código. Un ejemplo del correcto funcionamiento del modo pánico se puede observar en prueba2(entrada y salida) del directorio Pruebas_lexico/ficherosDeEntradaYSalida.

Para la implementación de los operadores relacionales necesitaremos incluir los tokens "GT", "GE", "LT", "LE", "AND", "OR" para las operaciones mayor estricto, mayor o igual, menor estricto, menor o igual, and y or respectivamente. Para que los tokens se identifiquen de manera correcta primero se declaran las expresiones regulares de los tokens "LE" y "GE", puesto que al tener ambos como prefijo común el símbolo "<" queremos que se priorice identificar los símbolos "<=" y ">=" para no dar lugar a una situación donde esos símbolos se identifiquen como dos tokens "LT ASIG" y "GT ASIG" respectivamente, siendo esto un comportamiento erróneo.

Para la implementación del bucle for incluiremos un token "TO" que precederá al valor máximo que puede tomar la variable que hace la función de iterador y el token "FOR" para iniciar un bucle for.

2. Análisis sintáctico.

El analizador sintáctico se apoya en el analizador léxico, pues este le va mandando los tokens que se va a encontrando, con los que va reduciendo las expresiones.

Las reglas de producción son las indicadas en las practicas.

En la gramática del lenguaje se produce ambigüedad en dos puntos: en las sentencias if-else, al haber varios if's seguidos el analizador no sabría a que if se corresponde el else. Esto no es un problema puesto que el else siempre va correspondido del if más próximo a él y como el analizador generado por bison cuando se encuentra un conflicto desplaza/reduce por defecto se queda con la opción de desplazar, el analizador "desplaza los if" hasta llegar a un else, reduciendo con el if anterior; En las expresiones aritméticas, causada por la precedencia y la asociatividad de los operadores. En este caso la opción por defecto de bison no es suficiente ya que no siempre deberemos reducir ante un conflicto desplaza/reduce, esto dependerá de los operadores implicados. Por ejemplo:

Entrada:

```
main() {  
    x=x*x+x;  
}
```

Salida correcta:

```
declarations -> lambda  
statement_list -> lambda  
expression -> IDEN  
expression -> IDEN  
expression -> expression MULT expression  
expression -> IDEN  
expression -> expression SUMA expression  
statement -> IDEN ASIG expression PYCO  
statement_list -> statement_list statement  
program -> IDEN PARI PARD LLAVI declarations statement_list  
LLAVD
```

Salida incorrecta (debida a tomar como acción por defecto reducir) :

```
declarations -> lambda
statement_list -> lambda
expression -> IDEN
expression -> IDEN
expression -> IDEN /* en vez de reducir la multiplicación se
desplaza SUMA e IDEN llegando a la sentencia IDEN * IDEN +
IDEN reduciendo primero la suma y después la multiplicación
*/
expression -> expression SUMA expression
expression -> expression MULT expression
statement -> IDEN ASIG expression PYCO
statement_list -> statement_list statement
program -> IDEN PARI PARD LLAVI declarations statement_list
LLAVD
```

Para solucionarlo hay que indicarle a bison cuál es la precedencia de los operadores poniendo en la sección de declaraciones de tokens el token correspondiente a cada operación precedido de su asociatividad (%right o %left) y, escribiendo los operadores en diferentes líneas, los de menor precedencia encima de los de mayor precedencia, de esta manera bison sabrá interpretar las operaciones aritméticas sin problema, aunque aún queda un caso particular, el del operador -. El operador - puede simbolizar tanto como una resta (operador binario) tanto como un número negativo (operador unario), en este último caso debería tener la máxima

precedencia, a diferencia del - como operador de resta que debe tener una precedencia más baja empatado con la suma. Para ello, tendremos que crear un nuevo token para simbolizar el - como operador unario y en la regla en la que aparezca habrá que indicarle a bison que la precedencia de ese token debe cambiara la del token – como operador unario utilizando %prec
TOKEN_MENOS_UNARIO.

Como prueba del correcto funcionamiento de la precedencia de los operadores véase la prueba Sintactico2(ficheros de entrada y salida) de Pruebas_sintáctico/ficherosDeEntradaYSalida, en la que puede apreciarse como la primera regla que se reduce es aquella cuya operación tiene más precedencia (la multiplicación primero y después las sumas). Para verificar el correcto funcionamiento del analizador léxico véase pruebaSintactico1(ficheros de entrada y salida) de Pruebas_sintactico/ficherosDeEntradaYSalida.

Para la implementación de los operadores relacionales deberemos introducir una nueva regla de producción "relational" que deriva en cualquier expresión relacional, pero antes deberemos encargarnos de la precedencia de los operadores relacionales. La asociatividad será por la izquierda para todos los operadores, teniendo más precedencia los símbolos "<", ">", "<=" y ">=" que los símbolos "AND" y "OR" puesto que para hacer una operación OR o AND debemos

conocer el resultado de los dos operandos que serán expresiones relacionales. Las reglas de producción de son similares a las de las expresiones aritméticas.

Para la implementación del bucle do-while solo necesitaremos una regla de producción de la forma statement -> DO statement PARI relational PARD PYCO

3. Análisis semántico y generación de código.

El análisis semántico corresponde a la etapa del análisis sensible al contexto, al contrario que en el análisis sintáctico. Esto implica que: cuando se declara una variable/constante esta no debe haber sido declarada con anterioridad; cuando se utiliza una variable/constante esta debe haber sido declarada previamente; una constante es inmutable, por lo que si ya ha sido declarada no se le debe poder volver a asignar un valor. Para poder implementar una solución a estos puntos, tendremos que empezar con parte de la generación de código, concretamente con el segmento de datos, ya que deberemos llevar un registro de las variables ya declaradas. Para ello utilizaremos una estructura enlazada que hará la función de tabla de símbolos. Para saber el tipo (variable, constante) de cada identificador en el momento en el que se encuentre en la regla de producción utilizaremos una variable "tipo", añadiendo código C a mitad de cada regla de declaración para asignar a la variable tipo el tipo reconocido en esa regla (VAR o CONST). Deberemos utilizar un contador del número de cadenas reconocidas hasta el momento, para crear un identificador único para cada una de ellas. También necesitamos indicarle a bison el tipo que tendrán asociados los lexemas, con %union incluyendo entre llave la variable lexema, en nuestro caso de tipo char*, y también indicando que token tienen ese tipo con %type<lexema>. Para que el sintáctico pueda tener los lexemas antes mencionados, debemos actualizar su valor desde el analizador léxico cuando se encuentre con los tokens correspondientes (cuando encuentre un token de tipo identificador, número o cadena pondrá el valor del lexema en yylval.lexema). En aquellas reglas de producción donde se declare una variable o constante deberemos añadirla en la tabla de símbolos (definiendo una función anadeEntrada que rellena un nodo de la estructura y finalmente lo añade a la tabla), marcando si es variable o constante, su identificador (lexema), y en caso de ser una cadena, el contador de cadenas (en el campo valor de su nodo). Las reglas de producción que utilicen variables o constantes (expresiones aritméticas etc.) deberán verificar antes si las variables y constantes implicadas han sido declaradas previamente, buscando su identificador en la tabla de símbolos (función perteneceTabla). Por último, las reglas de producción que asignen un nuevo valor a una variable deben verificar que el identificador se corresponde con el de una variable, y no una constante (función esConst), y por supuesto que esta variable haya sido declarada previamente (utilizando de nuevo la función perteneceTabla).

También habrá que definir una función que imprimirá el segmento de datos, imprimiendo en el formato utilizado en mips las diferentes variables y constantes almacenadas en nuestra tabla de símbolos. Con esta sección tendremos cubierto el segmento de datos y los errores semánticos.

Para la implementación del bucle for hay que considerar que se hace una asignación de una variable ya declarada que hace de iterador a un valor inicial, por tanto, habría que comprobar que dicha variable ya estuviese declarada.

Puede verificarse el correcto funcionamiento del análisis semántico en los ficheros de prueba: prueba_semántico y prueba_con_error_semántico(ficheros de entrada y salida) en Pruebas_semántico_generacion_codigo.


Lo último que nos queda por abordar es la generación del segmento de código, donde nuestra labor es traducir cada regla de producción al código mips correspondiente, añadiendo este código a una estructura enlazada que utilizaremos para listar todas las instrucciones que se van generando en orden. Necesitaremos gestionar los registros, para lo cual utilizaremos un array para almacenar si cada registro se está usando o no, manejaremos este array mediante las funciones selectRegistro (para seleccionar un registro libre) y liberarRegistro (para seleccionar el registro que se pase como parámetro). Para las instrucciones de salto tenemos que crear etiquetas, al igual que con las cadenas, necesitamos que el nombre de cada etiqueta sea único, por lo que necesitaremos llevar la cuenta del número de etiquetas, este será utilizado por una función nuevaEtiqueta para crear una etiqueta utilizando el contador de etiquetas.

La generación de código de los operadores relacionales se lleva a cabo de manera muy similar a las expresiones aritméticas puesto que no dejan de ser operaciones realizadas por la ALU, lo único que cambia es la instrucción (glt, slt, and, or). La excepción viene en las operaciones "<=" y ">=" puesto que no hay instrucciones en mips que realicen dicha comparación. Para resolverlo hemos optado por utilizar un registro que guarde 1 si una de las dos condiciones ("<" o "="; ">" o "=") inicializado a 0. Posteriormente se realizará un salto en caso de que se cumpla la condición opuesta (si la instrucción es "<=" saltará la condición de ese salto será ">") hacia una etiqueta omitiendo la instrucción donde se cambia el valor del registro a 1:

```
expression GE expression

codigo minic:
expression >= expression
    $1

codigo mips:
li $t2, 0 guardar en un registro auxiliar un 0(false)
blt $1, $2, etiqueta si el primer valor no es >= que el segundo se salta a la etiqueta, evitando poner el resultado a 1
li $t2, 1 si el primer valor es >= que el segundo se pone el registro auxiliar a 1(true)
etiqueta:
move $1, $t2 se guarda el resultado en $1
```

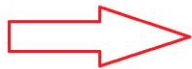


La generación de código del bucle do-while es similar a la de un bucle while pero más simple, puesto que la condición se comprueba al final del bucle, teniendo que utilizar solo una etiqueta en vez de 2. Primero se crea la etiqueta del inicio del bucle, posteriormente se concatenan las instrucciones del bloque de sentencias del bucle y finalmente se añade una instrucción de salto que saltará al inicio del bucle si la condición se cumple:

statement: DO statement WHILE PARI relational PARD PYCO

codigo mini_c:

```
do
statement
while(relational);
```



codigo mips:

```
etiqueta1:      etiqueta de inicio del bucle
statement      bloque de sentencias del bucle
relational      condicion del bucle
bnez recuperaResLC($5) etiqueta1 comprobar si se cumple la condicion del bucle
```

Para la generación de código del bucle for necesitamos, como en el bucle while, dos etiquetas, una para saltar al inicio del bucle y otra para saltar al final del bucle. La sentencia for especificada utiliza una variable que se utiliza como iterador, por tanto, en cada iteración habrá que guardar el valor de iterador en memoria. Este valor va incrementando en 1 en cada iteración, para ello se utilizará una instrucción addi al final de cada iteración. Una iteración se realiza si el iterador no ha excedido el valor máximo y se cumple la condición de terminación, para ello necesitaremos dos instrucciones de salto, una que salte al final del bucle si el iterador es mayor que el valor máximo(bgt) y otra que salte al final del bucle si el valor de la condición de terminación es 0(beqz). Al llegar al final de una iteración se realizará un salto incondicional al inicio del bucle:

FOR PARI IDEN ASIG expression TO expression PYCO relational PARD statement
 valor mínimo valor maximo condicion de terminacion

Codigo minic:

```
for(x=1 to 10;x!=7)
statement
```



Codigo mips:

```
etiqueta1: Etiqueta de inicio de bucle
sw recuperaResLC($5),_x Guardar en la variable el valor del iterador
bgt recuperaResLC($5), recuperaResLC($7), etiqueta2 Si el iterador es mayor que el valor maximo se sale fuera del bucle
beqz recuperaResLC($9), etiqueta2 Si no se cumple la condicion de terminación de sale del bucle
//statement Bloque de sentencias del bucle for
addi recuperaResLC($5), recuperaResLC($5), 1 Incrementar en 1 el iterador
b etiqueta1 Saltar al inicio del bucle
etiqueta2: Etiqueta fin de bucle
```

Se puede comprobar el correcto funcionamiento del compilador ejecutado con spim la salida del compilador al compilar cualquier fichero.mc en el directorio Pruebas_semántico_generación_código.

4. Cambios realizados.

Con respecto a la primera entrega los cambios realizados han sido los siguientes: se ha cambiado la longitud máxima de un identificador de 16 caracteres a 32 de acuerdo al enunciado de la práctica; se ha eliminado el token VOID puesto que no pertenece al enunciado de la práctica; se ha solucionado un pequeño error que causaba que los print de cadenas no funcionasen, este error era que, al añadir la cadena a la lista de símbolos con la función anadeCadena, en vez de poner

dentro de la función mencionada CADENA para indicar que el dato que se va a introducir en la lista de símbolos es una cadena pusimos CAD por equivocación, siendo CADENA el valor del enumerado de el fichero listaSimbolos y CAD el nombre del token asignado a las cadenas, este error impedía que se añadiera la cadena a las lista de símbolos provocando un error sintáctico al intentar hacer print de una cadena, ahora funciona adecuadamente; hemos añadido los ficheros de salida de las pruebas que olvidamos incluir en la versión anterior; se ha ampliado la gramática del compilador, añadiendo operadores relacionales, bucle do-while y bucle for, con entradas y salidas de prueba de generación de código para todas ellas.