Práctica de Compiladores

Compiladores
(Curso 2023 - 2024

Por: Samuel Avilés Conesa - Gloria Sánchez Alonso

Profesor: **Eduardo Martínez Gracia**

Grado en Ingeniería Informática
Universidad de Murcia

1. Introducción

El compilador de MiniC constituye el proyecto de prácticas para la asignatura de Compiladores. Se basa en la traducción de código escrito en el lenguaje `Mini C` a código escrito en ensamblador de `MIPS`.

En los siguientes apartados se desarrolla la especificación acerca del lenguaje `MiniC` y las etapas de su traducción a código ensamblador de `MIPS`, así como el proceso de diseño del traductor.

Aunque lo especificaremos en el correspondiente apartado, el lanzamiento de todas las herramientas se realiza desde el fichero `main.c` y se ejecuta desde el fichero `makefile` utilizando la orden `make run`. También es posible ejecutar el traductor utilizando el shell de linux mediante `./minic <ficheroEntrada>`.

---

2. Lenguaje MiniC

`MiniC` es un lenguaje parecido a `C` aunque más reducido en diversos aspectos. Solo maneja constantes y variables enteras. De esta manera, los tipos *booleanos* se representan con enteros, siendo 0 el valor *falso* y el resto *verdadero*, al igual que en `C`. El lenguaje tiene las siguientes sentencias de control de flujo de ejecución:

- `if`
- `if-else`
- `while`
  Además, se han implementado las diferentes mejoras por lo que existen pequeñas variaciones respecto al lenguaje original propuesto por los profesores de la asignatura. Se han introducido operadores relacionales y las sentencias de control del flujo de ejecución `do-while` y `for` las cuales especificaremos más adelante.

2.1. Símbolos terminales

El analizador sintáctico hace uso de una gramática que veremos más adelante. Esta gramática está compuesta por 29 símbolos terminales que permiten, en primer lugar, el análisis léxico del programa fuente. Son los siguientes.

Los enteros, `nume` (token `NUME` en el código), pueden tener un valor desde -2^{31} hasta 2^{31}. También existen cadenas de texto, `string` (token `string` en el código), delimitadas por comillas dobles.

Los identificadores, `id` (token `ID` en el código), están formados por secuencias de letras, dígitos y símbolos de subrayado, no comenzando por dígito y no excediendo los 16 caracteres.

Las palabras reservadas son `var` (`var`), `const` (`CONST`), `if` (`IF`), `else` (`ELSE`), `while` (`WHILE`), `for` (`FOR`), `do` (`DO`), `print` (`PRINT`) y `read` (`READ`).

Los operadores relacionales son `<` (`MENOR`), `≤` (`MENORI`), `>` (`MAYOR`), `≥` (`MAYORI`), `==` (`IGUALD`) y `≠` (`NOTIG`).

Por último, disponemos de los caracteres especiales de separación: `;` (`PYCO`) y `,` (`COMA`), `:` (`DOSP`); de operaciones aritméticas: `+` (`SUMA`), `-` (`REST`), `*` (`PROD`) y `/` (`DIVI`); de asignación: `=` (`IGUA`); y de control precedencia y bloques: `(` (`PARI`), `)` (`PARD`), `{` (`LLAVI`) y `}` (`LLAVD`).

2.2. Gramática para el análisis sintáctico

Nuestro lenguaje está basado en una gramática libre de contexto que permite al analizador sintáctico comprobar la corrección del programa fuente. A continuación se muestra dicha gramática en notación BNF:

```
program → id ( ) { declarations statement_list }
declarations → declarations var identifier_list ;
                        |   declarations const identifier_list ;
                        |   λ

identifier_list → identifier
                                |  identifier_list , identifier

identifier → id
                        | id = expresion

statement_list → statement_list statement
                                | λ

statement → id = expresion ;
                        | { statement_list }
                        | if ( expr_rel ) statement else_part
                        | while ( expr_rel ) statement
                        | for ( id = expression ; expr_rel : expression ) statement
                        | do statement while ( expr_rel ) ;
                        | print ( print_list ) ;
                        | read ( read_list ) ;

else_part: → else statement
                        | λ

print_list → print_item
                        | print_list , print_item
```

```
print_item → expression
                        | string

read_list → id
                        | read_list , id
expr_rel → expression < expression
                   | expression > expression
                   | expression ≤ expression
                   | expression ≥ expression
                   | expression == expression
                   | expression ≠ expression

expression → expression + expression
                        | expression - expression
                        | expression * expression
                        | expression / expression
                        | - expression
                        | ( expression )
                        | NUME
                        | id
```

---

3. Análisis léxico (herramienta *Flex*)

En primer lugar, diseñamos nuestro analizador léxico. Utilizamos, para ello, la herramienta Flex, que precisamente genera analizadores léxicos. Todo el desarrollo de esta parte se encuentra en el fichero `lexico.l`.

Para comenzar, identificamos los tokens que hemos definido en la gramática del lenguaje. Este proceso se basa en las expresiones regulares correspondientes a cada token. Reconocemos y eliminamos, además, comentarios de una o varias líneas y separadores (espacios en blanco, tabuladores y retornos de carro.

También comprobamos en este apartado la corrección de identificadores y literales enteros, según lo especificado, e implementamos la detección de errores en modo pánico. Si existe algún problema en el análisis, se informará de un error léxico. Además, pasamos a la siguiente fase del análisis los lexemas de los símbolos terminales `id` (`ID`), `string` (`string`) y `num` (`NUME`).

---

4. Análisis sintáctico y semántico y generación de código (herramienta *Bison*)

La siguiente fase sería el analizador sintáctico. Sin embargo, combinamos esta y las dos últimas en una misma sección ya que las especificamos dentro de un mismo fichero `sintactico.y`. Este fichero pertenece a la herramienta Bison, que nos permite generar el analizador sintáctico y, además, nos incluye lo necesario para guiar las tareas de análisis semántico y generación de código.

4.1. Análisis sintáctico

Por parte del analizador sintáctico, la función llevada a cabo es reconocer sintácticamente ficheros generados por la gramática que hemos mostrado anteriormente. Se establecen las precedencias necesarias en los operadores y, además, se inserta una opción para que el analizador acepte un único conflicto de desplaza/reduce, el de las sentencias `if-else`.

También se realizan algunas recuperaciones de errores mediante puntos de sincronización para facilitar la corrección de errores en un programa escrito en MiniC. Si existe algún problema en el análisis, se informará de un error sintáctico y no se detendrá la ejecución del proceso de traducción gracias a las reglas de error añadida a la gramática haciendo uso de la función `parse.error` de Bison, usando la directiva `%define parse.error verbose`.

Cabe destacar el uso de las directivas `%precedence`, `%left` y `%nonassoc` de Bison, usadas en el fichero `sintactico.y` para establecer la jerarquía de operadores y así poder determinar de manera correcta el orden en el que Bison tiene que hacer las reducciones de las expresiones aritméticas y de los operadores evitando así conflictos.

4.2. Análisis semántico (contenedor `listaSimbolos`)

Para realizar el análisis semántico, definimos una tabla de símbolos mediante la estructura contenedora `listaSimbolos`. Esta se trata de una lista simplemente enlazada que contiene instancias del tipo `Simbolo`. Este tipo almacena el nombre, el tipo y el valor de un símbolo, pudiendo almacenar variables (`var`), constantes (`const`) y, como veremos ahora, cadenas de texto (`string`).

En cuanto a manipulación de la tabla de símbolos, hemos reducido los métodos a algunos básicos, de manera que podemos crearla (`creaLS()`), insertar elementos (`insertaLS()`) y liberarla (`liberaLS()`). También podemos comprobar si un símbolo existe en ella (`buscaLS()`) y recuperar distintas posiciones en la lista (`inicioLS()`, `finalLS()`, `siguienteLS()`).

Como hemos explicado en el apartado anterior, desde el analizador léxico (referencia al campo `cadena` de la `union`) llegan, además de los tokens, algunos atributos de símbolos terminales. Estos atributos constituyen el nombre del símbolo a insertar.

Reutilizamos el tipo `Simbolo` para almacenar cadenas de texto (en el campo nombre) que luego imprimiremos, junto a los símbolos de tipos variable y constante, en el segmento de datos de nuestro resultado en ensamblador de MIPS. Para la inserción de cadenas, disponemos de un método especial `insertaLS()` que nos permite insertar un nuevo símbolo en la lista en una posición dada.

Bison nos permite realizar acciones dentro de la gramática que se ejecutarán a medida que se produzcan las correspondientes reducciones. Así, para implementar las acciones de manipulación de la tabla de símbolos, entre llaves en cada regla de producción, insertamos la funcionalidad necesaria.

En las secciones de declaración, el analizador semántico se encargará de comprobar si un elemento ya estaba insertado en la tabla y lanzar un error, o no e insertarlo correctamente. Para saber el tipo que hay que insertar, tenemos una variable global `Tipo` a la cual se le asigna el valor **(constante o variable)** en una acción en medio de la regla de producción `declarations` de manera que cuando se lee un símbolo se ejecuta la función `insertaID()`, que luego utiliza la función de inserción.

En las secciones de asignación, el analizador semántico comprueba si el símbolo al que se asigna existe y es variable, en caso de no serlo imprimirá que existe un error y sumará uno al contador global

de errores.

4.3. Generación de código (contenedor `ListaCodigo`)

Para implementar la generación de código de MiniC, utilizamos un subconjunto del código ensamblador de MIPS. En el fichero ensamblador final, se vuelca, en primer lugar, una representación de la tabla de símbolos en el segmento de datos (`.data`). Aquí se declaran las cadenas de texto (`.asciiz)` y las variables enteras globales (`.word` de 32 bits inicializada a 0 y con el identificador precedido del carácter `_` para diferenciarlo de las instrucciones de MIPS).

A continuación, comienza el segmento de texto que contiene las instrucciones del código ensamblador. Se define el punto de entrada al programa (`.globl main`) y se imprime la lista de código generada.

Para almacenar las instrucciones de MIPS generadas en la última etapa del compilador, utilizamos un tipo `Operacion` y un contenedor de tipo lista `ListaCodigo`.

El tipo `Operacion` está simplemente compuesto por cuatro cadenas de texto, pudiendo representar todas las instrucciones de MIPS. Estos campos son `op` (código de la operación), `res` (resultado de la operación), `arg1` (primer argumento de la operación) y `arg2` (segundo argumento de la operación). Si alguno de ellos no se usa, se marca como `NULL`.

Aprovechamos el tipo `Operacion` para almacenar las etiquetas generadas en las listas de código, asignando `etiq` el campo `op` para que y estableciendo los demás campos como `NULL` para que a la hora de imprimir la lista de código general, podamos tratarlas adecuadamente.

En cuanto a la lista de código, cada símbolo no terminal de la gramática dispone de su propia `ListaCodigo`, almacenada en su correspondiente atributo `$$` (referencia al campo `codigo` de la `union`). Mediante las reglas de producción, en las reducciones, nos encargamos de crear las listas de código (`creaLC()`), liberarlas (`liberaLC()`), añadirles operaciones en alguna `PosicionListaC` (`insertaLC()`) y concatenarlas con las listas hijas (`concatenaLC()`), según las necesidades de cada caso.

Para el control de los registros temporales en el codigo MIPS se han creado una tabla de 10 posiciones la cual sirve para llevar el control de los registros libres, además se han creado diferentes funciones para trabajar con los registros, la función `obtenerReg()`, utilizada para obtener un registro libre para utilizarlo en la generación de código durante una reducción. La función `liberarReg()` utilizada para indicar que se ha terminado de usar un registro y que este pueda ser asignado a otra operación que lo solicite.

Por otra parte, para la generación de las etiquetas se ha creado una función la cual devuelve la etiqueta ya construida `nuevaEtiqueta()`, dicha función se apoya en un contador global que lleva el recuento de las etiquetas generadas, al llamar a esta función se incrementa el contador y la función devuelve una cadena del tipo `etiqX` siendo *X* el valor actual del contador.

Además, se han creado algunas funciones auxiliares para evitar la repetición de código y facilitar la comprensión del código. La función `concatena(str1,str2)` sirve para concatenar dos cadenas de caracteres introducidas como parámetro. La función `analisis_ok()` sirve para realizar la comprobación de que el número de errores siga a 0 durante la generación de código, esta función es utilizada en todas las acciones de las reglas de producción para continuar generando código solamente si no se ha producido ningún error.

---

5. Manual de uso

5.1. Compilación inicial de MiniC

Para generar el compilador desde Ubuntu, utilizamos una shell. Primeramente, nos colocamos en el directorio del proyecto. Ahora, ejecutamos la orden `make`. Este programa se encargará ahora de seguir las directivas establecidas en el fichero `makefile` a través de un autómata para generar el programa objeto de MiniC Compiler, `minic`.

Adicionalmente, podemos ejecutar `make` con los parámetros `clean` para limpiar los restos de la compilación o `run` para, una vez generado el programa objeto, ejecutarlo con un fichero de entrada `test.mc` y generar un fichero de salida `test.s` en el mismo directorio.

5.2. Uso básico de MiniC Compiler

Una vez disponemos del programa ejecutable `minic`, podemos ejecutarlo para compilar algún programa escrito en MiniC y almacenar el resultado en un fichero de código ensamblador MIPS de la siguiente manera.

```
./minic source.mc > target.s
```

5.3. Prueba del código final

Para probar nuestro fichero resultado en esamblador de MIPS, podemos utilizar los simuladores SPIM o MARS, tanto en sus versiones gráficas como de consola.

```
./spim -file target.s
```

5.4. Ejemplo

A continuación se plantea un ejemplo de código origen en MiniC (fichero adjunto `test.mc`).

```
prueba () {
    const a = 1;
    const b = 2 * 3;
    var c;
    var d = 5 + 2, e = 9 / 3;

    print "Inicio del programa\n";

    print "Introduce el valor de \"c\":\n";
    read c;

    if (c) print "\"c\" no era nulo.", "\n";
    else print "\"c\" si era nulo.", "\n";

    /* Imprimir d */
    while (d) {
        print "\"d\" vale", d, "\n";
        d = d - 1;
    }

    /* Imprimir e */
```

```
    do {
        print "\"e\" vale", e, "\n";
        e = e - 1;
    } while(e);

        print "Final","\n";
}
```

Y su correspondiente fichero resultado de la compilación en ensamblador de MIPS (fichero adjunto `test.s`).

```
.data
_a: .word 0
_b: .word 0
_c: .word 0
_d: .word 0
_e: .word 0
$str1: .asciiz "Inicio del programa\n"
$str2: .asciiz "Introduce el valor de \"c\":\n"
$str3: .asciiz "\"c\" no era nulo."
$str4: .asciiz "\n"
$str5: .asciiz "\"c\" si era nulo."
$str6: .asciiz "\n"
$str7: .asciiz "Valores de d hasta que valga 0\n"
$str8: .asciiz "\"d\" vale"
$str9: .asciiz "\n"
$str10: .asciiz "Valores de e hasta que valga 0\n"
$str11: .asciiz "\"e\" vale"
$str12: .asciiz "\n"
$str13: .asciiz "Valores de c hasta que valga 10\n"
$str14: .asciiz "\"c\" vale"
$str15: .asciiz "\n"
$str16: .asciiz "Final"
$str17: .asciiz "\n"
.text
.globl main
main:
li $t0,1
sw $t0,_a
li $t0,2
li $t1,3
mul $t0,$t0,$t1
sw $t0,_b
li $t0,5
li $t1,2
add $t0,$t0,$t1
sw $t0,_d
li $t0,9
li $t1,3
```

```
div $t0,$t0,$t1
sw $t0,_e
la $a0,$str1
li $v0,4
syscall
la $a0,$str2
li $v0,4
syscall
li $v0,5
syscall
sw $v0,_c
lw $t0,_c
li $t1,0
sne $t0,$t0,$t1
beqz $t0,etiq1
la $a0,$str3
li $v0,4
syscall
la $a0,$str4
li $v0,4
syscall
b etiq2
etiq1:
la $a0,$str5
li $v0,4
syscall
la $a0,$str6
li $v0,4
syscall
etiq2:
la $a0,$str7
li $v0,4
syscall
etiq3:
lw $t0,_d
li $t1,0
sne $t0,$t0,$t1
beqz $t0,etiq4
la $a0,$str8
li $v0,4
syscall
lw $t1,_d
li $v0,1
move $a0,$t1
syscall
la $a0,$str9
li $v0,4
syscall
```

```
lw $t1,_d
li $t2,1
sub $t1,$t1,$t2
sw $t1,_d
b etiq3
etiq4:
la $a0,$str10
li $v0,4
syscall
etiq5:
la $a0,$str11
li $v0,4
syscall
lw $t0,_e
li $v0,1
move $a0,$t0
syscall
la $a0,$str12
li $v0,4
syscall
lw $t0,_e
li $t1,1
sub $t0,$t0,$t1
sw $t0,_e
lw $t0,_e
li $t1,0
sgt $t0,$t0,$t1
bnez $t0,etiq5
la $a0,$str13
li $v0,4
syscall
li $t0,1
sw $t0,_c
etiq6:
lw $t1,_c
li $t2,10
slt $t1,$t1,$t2
beqz $t1,etiq7
la $a0,$str14
li $v0,4
syscall
lw $t3,_c
li $v0,1
move $a0,$t3
syscall
la $a0,$str15
li $v0,4
syscall
```

```
li $t2,2
add $t0,$t0,$t2
sw $t0,_c
b etiq6
etiq7:
la $a0,$str16
li $v0,4
syscall
la $a0,$str17
li $v0,4
syscall
li $v0, 10
syscall
```

Ejemplo de salida esperada:

```
Inicio del programa
Introduce el valor de "c":
0
"c" si era nulo.
Valores de d hasta que valga 0
"d" vale7
"d" vale6
"d" vale5
"d" vale4
"d" vale3
"d" vale2
"d" vale1
Valores de e hasta que valga 0
"e" vale3
"e" vale2
"e" vale1
Valores de c hasta que valga 10
"c" vale1
"c" vale3
"c" vale5
"c" vale7
"c" vale9
Final

-- program is finished running --
```

---

6. Mejoras implementadas

Se han implementado todas las mejoras propuestas para el traductor.

En primer lugar, para llevar a cabo la mejora del tratamiento de errores sintácticos se han añadido las siguientes reglas de producción

```
declarations → declarations var error ";"
                              | declarations CONST error ";"
statement →  error ";"
```

Con estas reglas de producción logramos que Bison retroceda hasta el ultimo carácter de escape para poder continuar con el análisis como si de una sentencia bien formada se tratase pero devolviendo 1 en el `yyparse`.

En segundo lugar, para llevar a cabo la mejora de los operadores relacionales se ha declarado los tokens `<` (`MENOR`), `≤` (`MENORI`), `>` (`MAYOR`), `≥` (`MAYORI`), `==` (`IGUALD`) y `≠` (`NOTIG`) y las reglas siguientes reglas de producción:

```
expr_rel → expression < expression
                  | expression > expression
                  | expression ≤ expression
                  | expression ≥ expression
                  | expression == expression
                  | expression ≠ expression
```

Con estas reglas de producción es posible generar el código para realizar las diferentes comparaciones y poder evaluar así el cumplimiento de las condiciones establecidas en las sentencias de control de flujo.

En tercer lugar, para la implementación de la sentencia de control de flujo de ejecución `do-while` se ha añadido una regla de producción y su correspondiente generación de código.

```
statement →  do statement while ( expr_rel ) ;
```

Por último, para llevar a cabo la implementación de la sentencia de control de flujo de ejecución `for` se ha añadido una regla de producción y su correspondiente generación de código, además del token `:` (`DOSP`).

```
statement → for ( id = expression ; expr_rel : expression ) statement
```

Cabe destacar que ante la posibilidad de libre elección de sintaxis, la sentencia implementada toma el siguiente formato `for(c=X;expr-rel:num)` siendo `c` una variable previamente declara a la que se le asignara como valor inicial el valor `X`, `expr-rel` una expresión utilizando alguno de los operadores relacionales previamente especificados y `num` un número que indica el paso al que avanza el índice en cara iteración del bucle.

---

7. Conclusiones

La implementación de este compilador/traductor supone una puesta en práctica de los conocimientos obtenidos en la parte teórica de la asignatura. Hemos disfrutado durante el desarrollo del proyecto dado que los profesores de la asignatura nos han proporcionado todo el material necesario para la implementación así como las instrucciones y explicaciones necesarias para todo. Además, el proyecto hace que comprendamos como funciona un compilador y cual es su nivel de complejidad más allá de todo el estudio teórico.

Finalmente, a nuestro parecer este proyecto hace que cobre sentido todo lo estudiado en la asignatura de Autómatas y en la asignatura de Compiladores ya que es una puesta en práctica directa de todo lo estudiado hasta ahora para lograr construir algo parecido a un compilador como los que utilizamos a diario.