Textos Docentes

Enseñanzas de Informática

Desarrollo de compiladores

CD-ROM incluído

Juan Marcos Sacristán Donoso

ISBN: 84-689-4299-5

Registro: 05/73562

Índice

Nota para el docente ... 3
1. Introducción ... 4
¿Qué es un compilador? ... 4
Diseño de un compilador ... 5
Analizador léxico ... 6
Analizador Sintáctico ... 7
Traductor sintáctico ... 9
Diseño de un compilador complejo ... 10
Guía de Desarrollo ... 11
2. Contenidos ... 12
3. Traducción de Java a Class ... 13
Introducción ... 13
Especificación léxica y analizador léxico ... 14
Especificación sintáctica y analizador sintáctico ... 17
Traducción al formato class ... 19
Constantes ... 20
Tipos básicos ... 20
Constant Pool (buffer de constantes) ... 21
Información de los atributos ... 21
Información de los campos ... 21
Información de los métodos ... 21
Fichero Class ... 22

4. C a Linux ASM 22
Introducción 22
C a ARBR 23
Especificación léxica y analizador sintáctico 23
Especificación sintáctica y analizador sintáctico 25
Traduciendo a ARBR 26
El lenguaje ARBR 27
ARBR a Linux ASM 28
Especificación léxica y analizador léxico 28
Especificación sintáctica y analizador sintáctico 29
Traduciendo a Linux ASM 30
El ensamblador de Linux 31
5. Entorno IDE 38
Introducción 38
Instalación 38
Uso 39
6. Resultados y conclusiones 40
7. Bibliografía 41
Anexo I: Gramática bison de Java 42
Anexo II: Gramática bison de C 77
Anexo III: Gramática bison de ARBR 83
Anexo IV: ETDS de ARBR a Linux ASM 85
Reglas de traducción 85
Funciones auxiliares 93

Nota para el docente

La presente obra ha sido realizada por Juan Marcos Sacristán Donoso, funcionario de carrera del cuerpo de profesores de enseñanza secundaria, especialidad de Informática. El autor tiene una amplia experiencia en el mundo de la informática, tanto de docente como de profesional.

Esta obra describe el Proyecto Megava, realizado por el autor y que consiste en la implementación de una serie de compiladores de diversos lenguajes de programación. También se incluye un entorno IDE para comodidad del usuario. Finalmente se ha optado por los lenguajes Java y C, y se ha implementado un analizador sintáctico de Java y un compilador de C dividido en front-end (C a ARBR) y back-end (ARBR a ensamblador de Linux). La programación ha sido en GNU C (Linux) ayudándose de flex (análisis léxico), bison (análisis sintáctico) y QT (librería gráfica).

Aún siendo bastante ambicioso, el contenido de la obra permite profundizar y dominar los contenidos de programación que se imparten en los ciclos formativos de grado superior de informática, **Administración de Sistemas Informáticos** (A.S.I.) y **Desarrollo de Aplicaciones Informáticas** (D.A.I.). ¿Qué mejor forma de entender un compilador que haciendo uno? Las técnicas descritas permiten comprender exhaustivamente el funcionamiento interno de un compilador de lenguaje de tercera generación, por ejemplo, Pascal y C.

El mayor impacto de la obra se producirá en los módulos de primer curso **Fundamentos de programación** (ciclo A.S.I.) y **Programación en lenguajes estructurados** (ciclo D.A.I.).

1. Introducción

El objetivo es, por un lado desarrolar una serie de front-ends y back-ends para diversos lenguajes. Entendemos como front-end la traducción de un lenguaje de alto nivel a uno intermedio (por ejemplo C a ARBR). Entendemos como back-end la traducción de un lenguaje intermedio a un lenguaje de bajo nivel (por ejemplo ARBR a ensamblador de Linux).

Por otro lado, también se desarrollará un entorno IDE, es decir, un editor gráfico que permite manejar ficheros y compilarlos con las herramientas anteriores.

¿Qué es un compilador?

Un compilador no es más que un traductor, es decir, un programa que nos permite pasar información de un lenguaje a otro. Por ejemplo, un compilador de C nos permite traducir ficheros escritos en lenguaje C a un lenguaje legible para la máquina (ensamblador).

El proceso de compilación no siempre es directo. Esto quiere decir que si partimos de un lenguaje A y queremos traducir a un lenguaje B, no siempre será recomendable traducir directamente de A a B, si no que puede ser conveniente usar un lenguaje intermedio (llamémosle X) y entonces diseñaríamos un traductor de A a X, y otro de X a B. En este caso, el primer traductor recibe el nombre de *front-end* mientras que el segundo se llama *back-end*. El lenguaje A es el *lenguaje fuente* y el lenguaje B es el *lenguaje objeto* o *destino*.

Diseño de un compilador

A continuación mostramos un diagrama de bloques que representa las fases a seguir en el proceso de diseño de un compilador además de los elementos auxiliares que necesitamos en cada fase.

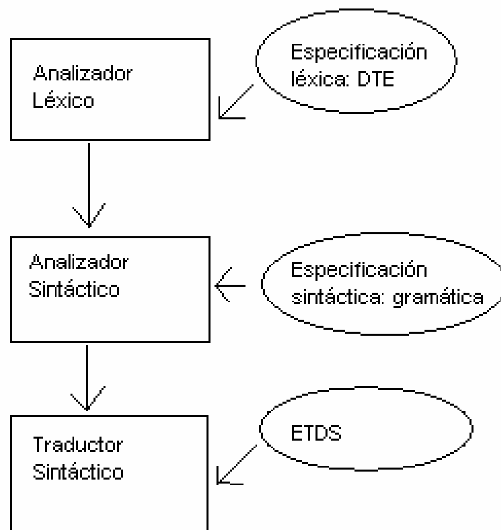


Figura 1: pasos a seguir para diseñar un compilador

Al final tendremos un único programa que sería el traductor. Este traductor tendría un analizador sintáctico, y a su vez el analizador sintáctico tendría un analizador léxico.

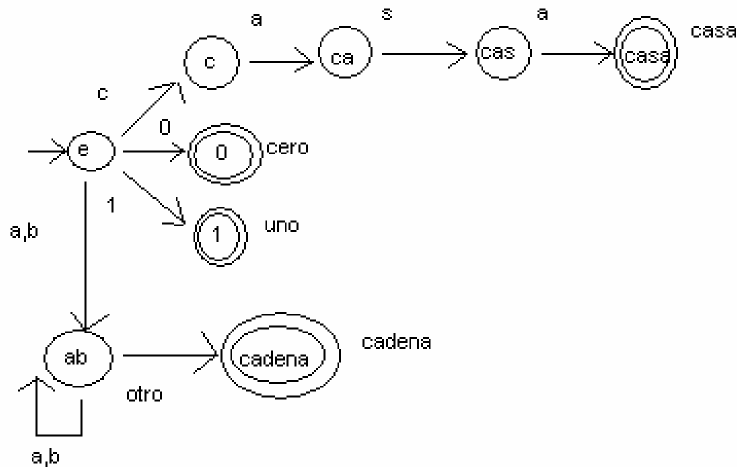
Veamos cada paso con mayor detenimiento:

Analizador léxico

Este analizador se encarga de formar los componentes léxicos del lenguaje fuente. Estos componentes reciben el nombre de *token*. Un token es un elemento que puede formar parte del lenguaje fuente.

Para implementar un analizador léxico necesitamos un DTE (diagrama de transición de estados) que nos proporcione la especificación léxica del lenguaje fuente.

Ejemplo: queremos reconocer como tokens el número 0 (**cero**), el número 1 (**uno**), la palabra reservada casa (**casa**) y cadenas formadas por las letras a y b (**cadena**). El DTE equivalente podría ser:



Partimos de un estado inicial (e) a partir de la cual vemos que con un 0 o con un 1 directamente obtenemos los tokens cero y uno respectivamente. Si tenemos una c, luego una a, luengo una s y luego una a obtenemos el token casa. Y si tenemos uno o más caracteres que sean a o b, obtenemos el token cadena. Cualquier otra posibilidad no contemplada en el DTE nos llevaría a un error léxico: por ejemplo la palabra cab, ya que a partir del estado ca no hay ningún arco que se active con la letra b.

Analizador Sintáctico

Este analizador se encarga de formar componentes sintácticos del lenguaje fuente. En definitiva, nos permitirá saber si un texto concreto pertenece al conjunto de posibles textos de un lenguaje.

Para implementar un analizador sintáctico necesitamos una gramática que nos proporcione la especificación sintáctica del lenguaje fuente.

Una gramática está formado por una serie de reglas. Cada regla tiene una parte izquierda y una parte derecha. Si se reconoce la parte derecha de una regla, se puede sustituir todo por la parte izquierda de la misma regla. Las reglas están formadas por dos tipos de símbolos:

- no terminales: también llamados variables y que se expanden en partes derechas de otras reglas.
- terminales: son tokens de la especificación léxica.

Ejemplo: queremos especificar la sintaxis de operaciones aritméticas de números enteros. Las operaciones en cuestión son la suma y la resta.

Una primera aproximación podría ser:

E → E **+** E

E → E **–** E

E → **nint**

Los tokens son:

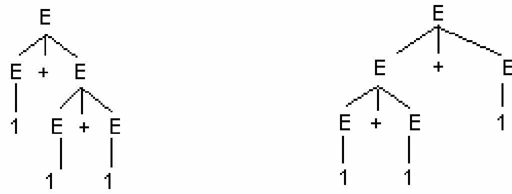
+: el signo más (suma)

-: el signo menos (resta)

nint: un número entero, cuya expresión regular sería [0-9]*

Sin embargo, esta gramática no es válida ya que es ambigua. La ambiguedad es un problema bastante importante ya que implica que puede haber más de un árbol de derivación para encontrar una misma cadena. No se pueden implementar analizadores mediante gramáticas ambiguas, ya que aparecerían conflictos.

Veamos dos árboles de derivación con la primera gramática para formar la expresión **1+1**:



La ambiguedad es bastante difícil de detectar. Para ello se definen una serie de subconjuntos de gramáticas, y a la hora de diseñar el analizador podemos detectar dicha ambiguedad. En nuestro caso interesa usar las gramáticas SLR a partir de las cuales se pueden implementar analizadores sintácticos ascendentes.

La gramática anterior en SLR sería:

E → E **+** T

E → E **-** T

E → T

T → **nint**

Traductor sintáctico

Un traductor sintáctico es un programa que realiza traducciones parciales del lenguaje fuente al lenguaje destino. Cada regla efectúa una traducción parcial y la regla inicial del programa se encarga de proporcionar la traducción total.

Para implementar un traductor sintáctico necesitamos un ETDS (esquema de traducción dirigida por la sintaxis) que nos indique las traducciones parciales de cada regla. Un ETDS es una extensión sobre una gramática que contiene:

- atributos sintetizados: son valores devueltos por la una regla una vez que se ha reconocido toda la parte derecha de una regla
- atributos heredados: son valores pasados de una regla a otra, y que aparecen a la izquierda de la regla receptora.

Para indicar los valores de los atributos se emplean instrucciones de asignación que aparecen entre llaves ({…}). También se pueden poner instrucciones tipo C.

Ejemplo: queremos traducir las expresiones de la gramática anterior a un lenguaje tipo árbol en el que disponemos de las siguientes instrucciones:

Suma(a,b)

Resta(a,b)

La traducción de 1+2+3 sería **Suma(Suma(1,2),3)**

El ETDS resultante es:

E → E’ **+** T {E.t = “Suma(” || E’.t || “,” || T.t || “)”}

E → E’ **–** T {E.t = “Resta(“ || E’.t || “,” || T.t || “)”}

E → T {E.t = T.t}

T → **nint** {T.t = nint.lexema}

Aquí conviene explicar que:

- el operador || nos permite realizar concatenaciones
- Usamos la comilla simple (‘) para distinguir la E de la parte izquierda de la E de la parte derecha en las dos primeras reglas
- Se supone que el analizador léxico guarda el lexema de todos los tokens.

Diseño de un compilador complejo

Como hemos indicado antes, no siempre traducimos directamente de un lenguaje A a un lenguaje B si no que empleamos front-ends y back-ends. Además, hay que tener en cuenta que los lenguajes objeto de un compilador suelen ser de bajo nivel, mientras que los lenguajes fuente suelen ser de alto nivel. Esto implica que en la traducción se deben tratar conceptos de alto nivel como clases, objetos, etc…

A la hora de tratar estos conceptos podemos optar o bien por simular todo en el lenguaje destino mediante rutinas de librería (y de esta manera obtendríamos un ejecutable) o bien delegar en un intérprete. El caso de C es el primero, es decir, obtenemos un ejecutable y lo que hacemos es servirnos de rutinas de librería (entrada / salida, cadenas , etc…). Sin embargo, Java sigue el segundo camino, es decir, el lenguaje objeto de Java es el lenguaje Class, que es un lenguaje interpretado. Los intérpretes de Class, llamados intérpretes de la máquina virtual, son los encargados de manejar conceptos de alto nivel como la creación de clases, recolección de basura, etc…

Otro concepto es el de la optimización, ya que muchas el código generado es redudante y se puede mejorar. Una traducción ya optimizada sería bastante costoso y no compensa, es por ello que normalmente en la fase de traducción lo que se busca es rapidez y eficiencia. Luego se suele filtrar la salida con un optimizador. A continuación se muestra un diagrama resumen de todo el proceso:

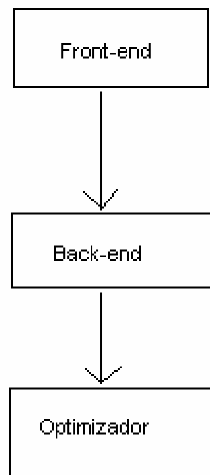


Figura 2: diagrama de bloques del proceso completo

Guía de Desarrollo

Para la implementación de las distintas partes del proyecto se ha usado como lenguaje de programación GNU C (Linux). Además, se han usado las siguientes herramientas auxiliares:

1. flex, que permite implementar analizadores léxicos.
2. bison, que permite implementar traductores sintácticos ascendentes y se complementa con flex.
3. makefile, herramienta para facilitar la compilación de programas complejos.
4. QT, librería gráfica que simplifica la creación de entornos gráficos. Requiere Xwindows.
5. Qtarch, programa para simplificar el diseño del aspecto del entorno gráfico.

2. Contenidos

Finalmente, se ha desarrollado lo siguiente:

- ✓ Analizador léxico y sintáctico de Java.
- ✓ Compilador de C a ensamblador de Linux dividido en un front-ent de C a ARBR y un back-end de ARBR a ensamblador de Linux.
- ✓ Entorno IDE.

Todo ello viene distribuido de la siguiente manera:

1. Carpeta C, que incluye el compilador de C:
   - 1.1. arbr.zip, el intérprete de lenguaje ARBR (lenguaje intermedio usado)
   - 1.2. mcc.tgz, código fuente del compilador
   - 1.3. mcc.txt, manual del compilador
2. Carpeta mjava, que incluye el analizador de Java:
   - 2.1. mjava.l, analizador léxico (en flex)
   - 2.2. mjava.y, analizador sintáctico (en bison)
   - 2.3. class_set.h, constantes y estructuras del formato class
   - 2.4. makemjava, makefile
   - 2.5. mjava.txt, manual del analizador
3. Carpeta graf, que incluye el entorno IDE:
   - 3.1. mide.tgz, código fuente del entorno IDE
   - 3.2. mide.txt, manual del entorno IDE

3. Traducción de Java a Class

Introducción

Para realizar el traductor de Java a Class se han seguido los siguientes pasos:

1) Implementación del analizador léxico de Java.
2) Implementación del analizador sintáctico de Java.
3) Comprobaciones semánticas (en ambos analizadores).
4) Traducción al formato class.

Especificación léxica y analizador léxico

El desarrollo de este analizador ha sido realizado con la herramienta flex de Linux, que permite implementar un analizador léxico especificando los tokens mediante expresiones regulares.

Conviene mencionar que se reconocen como palabras reservadas **const** y **goto** a pesar de que no aparecen en ninguna regla de la gramática. Esto se debe a que no se debe aceptar ninguna de las dos palabras como identificador.

La especificación exacta de los tokens ha sido extraída de la especificación de Java:

1) Elementos reconocibles pero que no son tokens:

/* … */	comentarios en varias líneas
/** ... */	comentarios en varias líneas de documentación (para usar con javadoc por ejemplo
// ...	comentario en una línea
[\t\n]	expresión regular para detectar separadores: hay que controlar la posición de cada carácter para emitir errores precisos.

2) Palabras reservadas:

const, goto, boolean, byte, short, int, long, char, float, double, package, import, public, protected, private, static, abstract, final, native, synchronized, transient, volatitle, class, extends, implements, void, throws, this, super, interface, if, else, switch, case, default, while, do, for, break, continue, return, throw, try, catch, finally, new, instanceof

3) Otros tokens

TOKEN	**DESCRIPCIÓN**
LBRA	[
RBRA	]
POINT	.
PYCOMA	;
STAR	*
COMA	,
LPAR2	{
RPAR2	}

ASSOP	=, *=, /=, +=, -=, &=, !=, <<=, >>=, >>>=
LPAR	(
RPAR	)
DOSP	:
PPLUS	++
MMIN	
PLUS	+
MINUS	-
WAVE	~
NO	!
SLASH	/
PERC	%
LSH	<<
RSH	>>
DRSH	>>>
RELOP	<, >, >=, <=
EQ	==
NEQ	!=
AND	&
HAT	^
BAR	|
DAND	&&
DBAR	||
ASK	?
INTEGER	número entero: 0xffl
FP	número en coma flotante: 3.2e -2d
BOOL	True ó false
CHARAC	caracter: 'x'
STRING	cadena de caracteres: "xx"
NULL2	null
IDENT	identificador: a14

3.1 Identificadores: están formados por letras (A -Z, a-z, _ y $) y dígitos (0 -9). El primer carácter es una letra y el resto pueden ser letras o dígitos.

3.2 Caracteres especiales: hay que contemplar las secuencias de escape como posibles caracteres:

\b	borrar
\t	tabulador
\n	Salto de línea
\f	Feed-forward
\r	carriage-return
\"	Comillas dobles: "
\'	Comillas simples: '
\\	barra: \

3.3 Concatenación de cadenas:

La concatenación de cadenas se resuelve en tiempo de compilación, de manera que si el compilador procesa una línea del estilo **String s="ab"+"c"** el compilador almacena la cadena "abc".

Tratamiento de números

Notación	prefijo	ejemplo
decimal	-	14
octal	0	07
hexadecimal	0x	0xff

Números enteros: por defecto int

precisión	sufijo	máximo
int	-	2147483647
long	l ó L	9223372036854775808L

Números flotantes: por defecto double

precisión	sufijo	máximo	mínimo
float	f ó F	3.40282347E+38F	1.40239846E -45F
Double	D ó D	1.79769313486231 570E+308	4.964065645841246544E -324

También están definidas las constantes Float.NaN y Double.NaN (NaN = not a number: no es un número).

Especificación sintáctica y analizador sintáctico

El desarrollo de este analizador ha sido realizado con la herramienta Bison de Linux. Esta herramienta nos permite implementar un traductor ascendente mediante una notación similar a la de un ETDS, como se ha visto en la asignatura Compiladores II. Evidentemente, un traductor que traduce siempre cadena vacía sería un analizador sintáctico.

La gramática usada es la de la especificación de Java. Cabe destacar el hecho de que no hay conflicto en las instrucciones de control (IF - THEN - ELSE) como suele ocurrir en la mayoría de lenguajes de alto nivel.

Java es un lenguaje de programación de alto nivel orientado a objetos. Cada fichero java puede contener como máximo una clase pública (accesible desde cualquier parte). La estructura básica de un programa Java es:

```
Class MyClass {
  Public int a;
   Public final static void main( String Args[]) {
    System.out.println("This is my class");
  }
}
```

Destacar que tenemos dentro de la clase:

- Una variable entera pública cuyo identificador es a
- Un método (función) público que además es final y estático cuyo identificador es main. Este método representa la función principal que se ejecuta al arrancar el programa. Debe ser siempre estático y final. Además no devuelve nada (void) y recibe como parámetro una array de cadenas que representa los parámetros especificados en línea de comandos.

Java dispone de un repertorio limitado de instrucciones, pero las distribuciones de Java disponen de una serie de librerías que aumentan su potencia. Para referenciar a una librería usamos import. Por ejemplo: **import miclase;**

La dificultad de compilar Java radica en la necesidad de tener que ir analizando otros ficheros class con tal de comprobar la sintaxis. No hacerlo significaría disponer de un conjunto final muy reducido.

Traducción al formato class

Realizaremos la traducción de java a class mediante un traductor sintáctico ascendente (SLR), de manera que se siguen los siguientes pasos:

- Comprobaciones léxicas, realizadas por el analizador léxico.
- Comprobaciones sintácticas, realizadas por el analizador sintáctico.
- Comprobaciones semánticas, implementa das como acciones a realizar por el traductor.
- Traducción parcial de cada parte del programa reconocida como una regla de la gramática, a realizar por el traductor.

Puesto que el formato class se puede representar como una estructura, la traducción parcial del programa se efectúa añadiendo información a la estructura a medida que se van reconociendo partes del programa. Dadas sus características, el uso de la estructura hace innecesario emplear tablas de símbolos y tablas de tipos.

El formato class.

Constantes

```
#define ACC_PUBLIC 0x0001

#define ACC_PRIVATE 0x0002

#define ACC_PROTECTED 0x0004

#define ACC_STATIC 0x0008

#define ACC_FINAL 0x0010

#define ACC_SUPER 0x0020

#define ACC_VOLATITLE 0x0040

#define ACC_TRANSIENT 0x0080

#define ACC_NATIVE 0x010 0

#define ACC_INTERFACE 0x0200

#define ACC_ABSTRACT 0x0400

#define ACC_STRICT 0x0800

#define MAGIC 0xCAFEBABE
```

Tipos básicos

Un byte, dos bytes o cuatro bytes.

```
typedef unsigned char u1;

typedef unsigned char* u2;

typedef unsigned char* u4;
```

Constant Pool (buffer de constantes)

En este buffer se guarda la información de variables y métodos de cada clase.

```
typedef struct
{
  u1 tag;
  u1 *info;
} cp_info
```

Información de los atributos

```
typedef struct
{
  u2 attribute_name_index; al constant_pool
  u4 attribute_length; tamaño
  u1 *info;
} attribute_info;
```

Información de los campos

```
typedef struct
{
  u2 access_flags; permisos
  u2 name_index;
  u2 descriptor_index;
  u2 attributes_count; tamaño
  attribute_info *attributes;
} field_info;
```

Información de los métodos

```
typedef struct
{
  u2 access_flags; permisos
  u2 name_index;
  u2 descriptor_index;
  u2 attributes_count; tamaño
  attribute_info *attributes;
} method_info;
```

Fichero Class

typedef struct {
u4 magic; 0xCAFEBABE (identificador)
u2 minor_version ; M
u2 major_version ; m

la versión es M.m

u2 constant_pool_count; tamaño
cp_info *constant_pool;
u2 access_flags; permisos
u2 this_class; puntero a la clase
u2 super_class; puntero al padre
u2 interfaces_count ; tamaño
u2 *interfaces; apunta al constant_pool
u2 fields_count; tamaño
field_info *fields;
u2 methods_count; tamaño
method_info *methods;
u2 attributes_count; tamaño
attribute_info *attributes;
} ClassFile;

4. C a Linux ASM

Introducción

Para realizar el traductor de C a Linux ASM se han seguido los siguientes pasos:

- Front end de C a ARBR
  1) Analizador Léxico
  2) Analizador Sintáctico
  3) Lenguaje ARBR
- Back end de ARBR a Linux ASM
  1) Analizador Léxico
  2) Analizador Sintáctico
  3) Lenguaje ensamblador (de Linux)

C a ARBR

Especificación léxica y analizador sintáctico

El desarrollo de este analizador ha sido realizado con la herramienta flex de Linux, que permite implementar un analizador léxico especificando los tokens mediante expresiones regulares.

1) Elementos reconocibles pero que no son tokens

/* ... */	comentarios en varias líneas
/** ... */	comentarios en varias líneas de documentación
// ...	comentario en una línea
[\t\n]	expresión regular para detectar separador es: hay que controlar la posición de cada carácter para emitir errores precisos

2) Palabras reservadas:

if, int, for, main, char, else, case, float, scanf, while, break, printf, switch, default

3) Otros tokens

<table>
<tr><th>TOKEN</th><th>DESCRIPCIÓN</th></tr>
<tr><td>PUNTOYCOMA</td><td>;</td></tr>
<tr><td>COMA</td><td>,</td></tr>
<tr><td>LBRA</td><td>{</td></tr>
<tr><td>RBRA</td><td>}</td></tr>
<tr><td>ASSOP</td><td>=</td></tr>
<tr><td>LPAR</td><td>(</td></tr>
<tr><td>RPAR</td><td>)</td></tr>
<tr><td>DOSPUNTOS</td><td>:</td></tr>
<tr><td>ADDOP</td><td>+,</td></tr>
<tr><td>MULOP</td><td>*, /</td></tr>
<tr><td>RELOP</td><td><, >, >=, <=, ==, !=</td></tr>
<tr><td>REFERENCIA</td><td>&</td></tr>
<tr><td>YBOOL</td><td>&&</td></tr>
<tr><td>OBOOL</td><td>||</td></tr>
<tr><td>NOBOOL</td><td>!</td></tr>
<tr><td>NINT</td><td>número entero: 47</td></tr>
<tr><td>NFIX</td><td>número en coma flotante: 3.2</td></tr>
<tr><td>CTECHAR</td><td>caracter: 'x'</td></tr>
<tr><td>CADCAR</td><td>cadena de caracteres: "xx"</td></tr>
<tr><td>ID</td><td>identificador: a14</td></tr>
</table>

a) Identificadores

Los identificadores están formados por letras (A -Z, a-z) y dígitos (0-9). El primer carácter es una letra y el resto pueden ser letras o dígitos.

b) Formatos

Para las instrucciones printf y scanf se soportan los siguientes formatos:

"%c\n", expr, siendo expr una expresión de tipo carácter.

"%d\n", expr, siendo expr una expresión de tipo entero.

"%8.3f\n", expr, siendo expr una expresión de tipo flotante.

"%s\n", cad, siendo cad una cadena d e caracteres

Especificación sintáctica y analizador sintáctico

El desarrollo de este analizador ha sido realizado con la herramienta Bison de Linux. Esta herramienta nos permite implementar un traductor ascendente mediante una notación similar a la de u n ETDS, como se ha visto en la asignatura Compiladores II. Evidentemente, un traductor que traduce siempre cadena vacía sería un analizador sintáctico.

El subconjunto que se ha tomado de C engloba programas con una única función (main) que tienen variables (enteras, reales y de tipo carácter), instrucciones de flujo y de control, lectura y escritura estándar, y expresiones aritméticas, lógicas y relacionales.

```
main()
{
  int a;
  int b;
  for (a = 1; a < 10; a = a + 1)
  {
        b = a * 2;
        printf("%d \n",b);
  }
```

Traduciendo a ARBR

Realizaremos la traducción de C a ARBR mediante un traductor sintáctico ascendente (SLR), de manera que se siguen los siguientes pasos:

- Comprobaciones léxicas, realizadas por el analizador léxico.
- Comprobaciones sintácticas, realizadas por el analizador sintáctico.
- Comprobaciones semánticas, implementadas como acciones a realizar por el traductor.
- Traducción parcial de cada parte del programa reconocida como una regla de la gramática, a realizar por el traductor.

En la fase de análisis, se gestiona la tabla de símbolos, lo que permite realizar comprobaciones semánticas como:

1) variable no declarada.
2) variable ya declarada.

El lenguaje ARBR

El lenguaje ARBR es un lenguaje interpretado de bajo nivel usado en las prácticas de Compiladores II en el curso 1997 - 1998 (entre otros). Es lo suficientemente potente como para permitir una traducción amigable de un lenguaje de alto nivel con C.

Es un lenguaje de árboles, lo que permite anidar operaciones, facilitando en gran medida la tarea de traducción de expresiones.

Si por ejemplo queremos traducir 1+2+3 haríamos **addi(#1,addi(#2,#3))**.

Además de un extenso repertorio de instrucciones, dispone de 16384 posiciones de memoria. Si queremos acceder al dato almacenado en la posición 0 haríamos **loadi(#0)**.

El direccionamiento relativo se consigue mediante anidamiento de instrucciones. Si queremos almacenar un 1 en la posición de memoria indicada por la posición 47, haríamos **stori(#1,loadi(#47))**.

Existen instrucciones de salto. El destino se puede especificar bien mediante un número de instrucción, bien mediante una etiqueta:

```
L1 jmp #3
   loadi(#0)
   jmp L1
```

Puesto que solo hay tres instrucciones, el primer salto es a la tercera instrucción, que a su vez salta a la instrucción que va después de la etiqueta. La segunda instrucción no se ejecuta nunca.

Finalmente, comentar que todo programa debe acabar con la instrucción halt.

ARBR a Linux ASM

Especificación léxica y analizador léxico

El desarrollo de este analizador ha sido realizado con la herramienta flex de Linux, que permite implementar un analizador léxico especificando los tokens mediante expresiones regulares.

1) Palabras reservadas

addi, subi, muli, divi, loadi, stori, andi, ori, halt, eqli, wri, wrc, wrl, jz, jnz, jmp

2) Otros tokens

TOKEN	DESCRIPCIÓN
COMA	,
LPAR	(
RPAR	)
ENTERO	número entero: #47
ETIQ	etiqueta: Lx, siendo x un número entero

Especificación sintáctica y analizador sintáctico

El desarrollo de este analizador ha sido realizado con la herramienta Bison de Linux. Esta herramienta nos permite implementar un traductor ascendente mediante una notación similar a la de un ETDS, como se ha visto en la asignatura Compiladores II. Evidentemente, un traductor que traduce siempre cadena vacía sería un analizador sintáctico.

El lenguaje ARBR es un lenguaje interpretado de bajo nivel, y de árboles. Se ejecuta instrucción tras instrucción, aunque dispone de instrucciones de salto. Ejemplo:

Stori(addi(#1,#2),#0) *suma 1 y 2, y guarda el resultado en la dirección 0*

Halt *fin*

Traduciendo a Linux ASM

Realizaremos la traducción de ARBR a Linux ASM mediante un traductor sintáctico ascendente (SLR), de manera que se siguen los siguientes pasos:

- Comprobaciones léxicas, realizadas por el analizador léxico.
- Comprobaciones sintácticas, realizadas por el analizador sintáctico.
- Comprobaciones semánticas, implementadas como acciones a realizar por el traductor.
- Traducción parcial de cada parte del programa reconocida como una regla de la gramática, a realizar por el traductor.

El ensamblador de Linux

El lenguaje ensamblador es el lenguaje de bajo nivel por excelencia: es el más cercano a la máquina y suele ofrecer resultados óptimos.

El ensamblador de Linux tiene un formato especial y que difiere ligeramente del formato de los ensambladores típicos de plataforma DOS/WINDOWS, como puedan ser MASM, TASM, etc.

Otro detalle a tener en cuenta es que Linux es un sistema operativo protegido, por lo que la mayoría de las acciones se realizan a través de llamadas al sistema (interrupción 80h).

Traducir de ARBR a ensamblador de Linux es una tarea "posible" en la medida de que ambos son lenguajes liberados de carga de alto nivel. El repertorio de instrucciones del lenguaje ensamblador es bastante limitado, y por ello siempre hay que procurar traducir desde una representación de bastante bajo nivel. De lo contrario, habría que implementar bastantes librerías auxiliares.

Conceptos previos

Un programa escrito en lenguaje ensamblador está dividido en varias zonas llamadas segmentos. Como mínimo debe haber un segmento de código (con las instrucciones a ejecutar), un segmento de datos (con la declaración de las variables) y un segmento de pila (con la declaración de la pila).

Sin embargo, en ensamblador de Linux, basta con declarar la etiqueta inicio del segmento de código. El resto de segmentos se definen por defecto, dejando su definición en manos de usuarios expertos que quieran especificar algún detalle importante.

Un elemento clave para trabajar en este lenguaje son los registros. Disponemos de un banco de registros en el que podemos almacenar valores enteros. El tratamiento con valores reales exige el uso del coprocesador matemático. A la hora de acceder a un registro, debemos especificar la longitud del dato (8, 16 o 32 bits) ya que la longitud determina que parte del registro ocupamos. De hecho, podemos llamar al registro de distintas maneras según la parte que queremos usar.

Registros (32 bits): eax, ebx, ecx, edx, esi, edi, esp, ebp

Registros (16 bits): ax, bx, cx, dx, si, di, sp, bp

Registros (8 bits, parte alta): ah, bh, ch, dh

Registros (8 bits, parte baja): al, bl, cl, dl

Otro concepto importante es el de las interrupciones. Tanto la BIOS como el sistema operativo proporcionan una serie de rutinas muy interesantes que facilitan la tarea del programador. Para acceder a una interrupción debemos indicar en ax la función y el servicio. También puede ser necesario pasar parámetros (bien por pila, bien por registros). Una vez se ejecuta la llamada a la interrupción, se devuelven los datos de salida (bien por pila, bien por registros).

Además de los registros ya nombrados, existe un registro especial denominado palabra de estado. Este registro permite identificar casos especiales como acarreo en suma operación, devuelve cero, desborde, etc.

Existen instrucciones cuyo funcionamiento depende del valor de la palabra de estado. Esto permite mayor flexibilidad a la hora de programar.

Un primer programa en ensamblador sería algo así como:

```
.text ; empieza el segmento de código
.globl _start ; _start es una etiqueta global
_start: ; aquí va el código
; rutina de finalización
movl $1, %eax ; meter un 1 en eax
xor %ebx, %ebx ; meter un 0 en ebx
int $0x80 ; ejecutar interrupción 80h
```

El lenguaje ARBR (especificado en otros apartados) es un lenguaje intermedio interpretado (usar intérprete de fichero arbr.zip) utilizado en las prácticas de Compiladores II (curso 1997 - 1998, entre otros).

Repertorio

El repertorio de instrucciones está formado por tres tipos de instrucciones:

- op fuente, destino: opera fuente con destino, dejando el resultado en destino.
- op destino: opera el acumulador (ax) con destino, dejando el resultado en destino; o bien opera con un único operando.
- op: realiza la operación sin más

Para especificar los parámetros se puede emplear:

- Direccionamiento inmediato: $1
- Direccionamiento a memoria: variables+4
- Direccionamiento indirecto con registro: variables(%eax)

Paso de parámetros

El paso de parámetros se puede realizar de tres maneras:

- mediante registros. para ello se debe procurar no machacar valores anteriores. Esto es muy limitado.
- mediante variables: existe una etiqueta llamada variables que apunta a la primera dirección de la zona de variables. Mediante direccionamiento relativo podemos acceder a cualquier variable que queramos. Por ejemplo si queremos tener dos variables a y b de tamaño 32 bits haremos
    - movl $1,variables+0 para meter un 1 en a, y
    - movl $2,variables+4 para meter un 2 en b.
- mediante pila: mediante las instrucciones pushl y popl, o bien empleando el registro sp, podemos apilar y desapilar a nuestro antojo. Sin embargo, conviene tener en cuenta que la instrucción call apila la ip (puntero a instrucción a ejecutar) mientras que ret lo desapila. Si queremos dejar tres parámetros (a, b y c de 32 bits) en la pila haremos:

```
pushl variables+0
pushl variables+4
pushl variables+8
```

Llamada a función

Supongamos que queremos realizar una llamada a una función que espera un parámetro en la pila:

Código para llamar

```
...
pushl %eax ; apilar el parámetro
call funcion_1 ; llamar a la función
addl $4, %esp ; desapilar el parámetro
```

Código de la función

```
...
movl 8(%esp),%ebx; coger el parámetro (descartar ip)
...
movl %ebx, %eax ; dejar resultado en eax
ret ; salir de la función
```

Histórico de traducción

Aquí se explica paso a paso la traducción de las distintas operaciones y las dificultades encontradas:

1) Salto incondicional

Para ello cambiamos jmp(L1) por jmp L1. Igualmente las etiquetas cambian de ser L1 instrucción, por L1: instrucción

2) Salto condicional.

Debido a cambios en la interpretación de los flags, jz se convierte en jnz, y vice -versa.

3) Comparación.

La instrucción eqli(exp1,exp2) se realiza mediante la instrucción cmpl:

```
cmpl %eax, %ebx
```

Y el valor del flag Z (Z = 1 si eax == ebx) se guarda en ax usando la instrucción sete.

4) Suma y resta

addl %eax, %ebx ; ebx = eax + ebx

subl %eax, %ebx ; ebx = eax - ebx

5) Multiplicación y división

imull %ebx ; eax = eax * ebx

idivl %ebx ; eax = eax / ebx

6) AND y OR (operaciones lógicas)

andl %eax, %ebx ; ebx = eax AND ebx

orl %eax, %ebx ; ebx = eax OR ebx

7) Escritura en pantalla

Se ha desarrollado una librería (fichero mcclib.c) con tres funciones:

- **void write_ent(int):** escribir un número entero
- **void write_ent(char):** escribir un carácter
- **void write_lin():** escribir un salto de línea.

Las tres funciones emplean la interrupción 80h para ejecutar la llamada al sistema write: write(fd, buffer, longitud)

- **fd:** descriptor de fichero. En nuestro caso 1 que es la salida estándar
- **buffer:** bloque de bytes a volcar
- **longitud:** longitud del buffer

Problemas encontrados

1) El manejo de números reales exige el uso del coprocesador, lo cual complica en gran medida la tarea de traducción.
2) La posibilidad de traducir funciones se ha descartado debido a que la traducción parcial a ARBR complica la posterior traducción a ensamblador (Cada instrucción ARBR no tiene por qué equivaler a una instrucción en ensamblador). La solución sería o bien volver a contar las instrucciones o bien usar call y ret (aumentar la funcionalidad del arbr). No se ha creído conveniente modificar arbr, ya que de esta manera se puede seguir empleando el intérprete.
3) Los operadores de comparación son muy difíciles de llevar al ensamblador ya que no están definidas en el repertorio. Se podría simular mediante saltos condicionales. Por ejemplo, si queremos implementar gtri(exp1, exp2) podemos hacer algo como:

```
cmp %ebx, %ecx ; comparar
jg L1 ; si ebx > ecx ir a L1
xor %eax, %eax ; eax = 0, ebx no es mayor que ecx
jmp L2
L1: mov %eax, 1 ; eax = 1, ebx es mayor que ecx
L2:
... ; y sigue el programa
```

4) Como se puede haber visto en el apartado anterior, es bastante factible agotar el banco de registros. La solución es o bien usar direcciones de memoria, o bien usar la pila para poder reutilizar registros y luego dejarlos como estaban. Expresiones muy largas desbordarían la pila.

Como corolario final, comentar que todo lo mencionado anteriormente perjudica seriamente la optimización. Desde luego, el código generado no es óptimo, pero las técnicas de optimización son muy sensibles a la plataforma utilizada. Es mejor diseñar un compilador sencillo y rápido, y después hacer un optimizador por plataforma.

5. Entorno IDE

Introducción

MIDE es un editor gráfico para XWindows desarrollado en C++ ayudándose de la librería gráfica QT.

Instalación

- Descomprimir el fichero mide.tgz
  - tar zxfv mide.tgz
- Compilar
  - make -f makemide
- Limpiar (borrar ficheros auxiliares)
  - make -f makemide clean

Uso

La aplicación se arranca tecleando 'mide', pero si se desea se pueden especificar archivos para abrir en línea de comandos.

Una vez arrancado, el uso es bastante intuitivo (extracto del archivo de ayuda):

- ✓ Pulse el botón FICHERO para seleccionar el fichero: para crear un nuevo fichero escribimos el nombre de un archivo no existente.
- ✓ Pulse el botón GUARDAR para guardar el fichero abierto
- ✓ Pulse el botón CERRAR para cerrar el fichero
- ✓ Pulse el botón BUSCAR para hacer una búsqueda
- ✓ Pulse el botón COMPILAR para compilar el fichero editado
- ✓ Pulse el botón AYUDA para ver esta ventana
- ✓ Pulse el botón ACERCA DE para ver información adicional
- ✓ Pulse el botón SALIR para salir de la aplicación
- ✓ Los botones ARRIBA y ABAJO sirven para desplazarse en el texto un número de líneas hacia arriba o hacia abajo
- ✓ Seleccione el numero de líneas a desplazarse con el selector de rango
- ✓ Los botones PRINCIPIO y FIN son para colocarse al principio o al final del fichero
- ✓ El botón VER LINEAS permite activar o desactivar la visualización del numero de linea
- ✓ El botón SALTAR sirve para saltar a una línea determinada

6. Resultados y conclusiones

Diferencias destacables entre un front – end y un back-end:

- El front-end tiene mayor coste de análisis.
- El back-end tiene mayor coste de traducción.

Diferencias entre C y Java:

- C permite mayor flexibilidad al usuario.
- Java es orientado a objetos

Diferencias entre ensamblador de Linux y el formato Class:

- El formato Class es independiente de la plataforma.
- El lenguaje ensamblador es sencillo de depurar.
- Los ficheros class están codificados y son difíciles de depurar (requiere un desensamblador).

Sobre el entorno IDE:

- La programación de aplicaciones gráficas es un importante, ya que enriquece cualquier aplicación.
- La programación de aplicaciones gráficas es eficiente cuando se trabaja por capas (una aplicación rudimentaria en modo texto por debajo de la capa gráfica, que tan solo debe llamar a la capa anterior).

7. Bibliografía

(Java)

- Programming for the Java Virtual Machine, Joshua Engel.
- The Java Virtual Machine Specification Second Edition, Tim Lindholm and Frank Yellin.
- The Java Language Specification, James Gosling and Bill Joy and Guy Steele.
- JDK v1.1.8, Java2 y documentación variada

(C)

- Documentación sobre Linux ASM (GNU, DJGPP).
- Compilador C de GNU.

(Compiladores)

- Libro de Aho, Sethi, Ullman.
- Apuntes de las asignaturas Compiladores I y Compiladores II (impartido por Francisco Moreno y Mikel Forcada) del Departamento de Lenguajes y Sistemas Informáticos de la Universidad de Alicante, curso 1997-1998.

Anexo I: Gramática bison de Java

```
goal : compilationunit
{
}
;
integerliteral : INTEGER
{
}
;
floatingpointliteral : FP
{
}
;
booleanliteral : BOOL
{
}
;
characterliteral : CHARAC
{
}
;
stringliteral : STRING
{
}
;
nullliteral : NULL2
{
}
;
identifier : IDENT
{
}
;
identifiere : identifier
{
```

```
}
| /* epsilon */

{
}
;
literal : integerliteral
{
}
| floatingpointliteral
{
}
| booleanliteral
{
}
| characterliteral
{
}
| stringliteral
{
}
| nullliteral
{
}
;
type : primitivetype
{
}
| referencetype
{
}
;
primitivetype : numerictype
{
}
```

```
| BOOLEAN
{
}
;
numerictype : integraltype
{
}
| floatingpointtype
{
}
;
integraltype : BYTE
{

}
| SHORT
{
}
| INT
{
}
| LONG
{
}
| CHAR
{
}
;
floatingpointtype : FLOAT
{
}
| DOUBLE
{
}
;
```

```
referencetype : classorinterfacetype
{
}
| arraytype
{
}
;
classorinterfacetype : name
{
}
;
classtype : classorinterfacetype
{
}
;
interfacetype : classorinterfacetype
{
}
;
arraytype : primitivetype LBRA RBRA
{
}
| name LBRA RBRA

{
}
| arraytype LBRA RBRA
{
}
;
name : simplename
{
}
| qualifiedname
{
```

```
}
;
simplename : identifier
{
}
;
qualifiedname : name POINT identifier
{
}
;
compilationunit : packagedeclaration
importdeclarations typedeclarations
{
}
;
importdeclarations : importdeclaration
{
}
| importdeclarations2 importdeclaration
{
}
| /* epsilon */
{
}
;
importdeclarations2 : importdeclaration
{
}
| importdeclarations2 impo rtdeclaration
{
}
;

typedeclarations : typedeclaration
{
```

```
}
| typedeclarations2 typedeclaration
{
}
| /* epsilon */
{
}
;
typedeclarations2 : typedeclaration
{
}
| typedeclarations2 typedeclaration
{
}
;
packagedeclaration : PACKAGE name PYCOMA
{
}
| /* epsilon */
{
}
;
importdeclaration : singletypeimportdeclaration
{
}
| typeimportondemanddeclaration
{
}
;
singletypeimportdeclaration : IMP ORT name PYCOMA
{
}
;
typeimportondemanddeclaration : IMPORT name
POINT STAR PYCOMA
```

```
{
}
;
typedeclaration : classdeclaration
{
}

| interfacedeclaration
{
}
;
modifiers : modifier
{
}
| modifiers2 modifier
{
}
| /* epsilon */
{
}
;
modifiers2 : modifier
{
}
| modifiers2 modifier
{
}
;
modifier : PUBLIC
{
}
| PROTECTED
{
}
| PRIVATE
```

```
{
}
| STATIC
{
}
| ABSTRACT
{
}
| FINAL
{
}
| NATIVE
{
}
| SYNCHRONIZED
{
}
| TRANSIENT
{

}
| VOLATILE
{
}
;
classdeclaration : modifiers CLASS identifier super
interfaces classbody
{
}
;
super : EXTENDS classtype
{
}
| /* epsilon */
{
```

```
}
;
interfaces : IMPLEMENT S interfacetypelist
{
}
| /* epsilon */
{
}
;
interfacetypelist : interfacetype
{
}
| interfacetypelist COMA interfacetype
{
}
;
classbody : LPAR2 classbodydeclarations RPAR2
{
}
;
classbodydeclarations : classbodydeclaration
{
}
| classbodydeclarations2
classbodydeclaration
{
}
| /* epsilon */
{

}
;
classbodydeclarations2 : classbodydeclaration
{
}
```

```
| classbodydeclarations2
classbodydeclaration
{
}
;
classbodydeclaration : classmemberdeclaration
{
}
| staticinitializer
{
}
| constructordeclaration
{
}
;
classmemberdeclara tion : fielddeclaration
{
}
| methoddeclaration
{
}
;
fielddeclaration : modifiers type variabledeclarators
PYCOMA
{
}
;
variabledeclarators : variabledeclarator
{
}
| variabledeclarators COMA
variabledeclarator
{
}
;
```

```
variabledeclarator : variabledeclaratorid
{
}

| variabledeclaratorid ASSOP
variableinitializer
{
}
;
variabledeclaratorid : identifier
{
}
| variabledeclaratorid LBRA RBRA
{
}
;
variableinitializer : expression
{
}
| arrayinitializer
{
}
;
methoddeclaration : methodheader methodbody
{
}
;
methodheader : modifiers type methoddecla rator
throws
{
}
| modifiers VOID methoddeclarator throws
{
}
;
```

```
methoddeclarator : identifier LPAR formalparameterlist
RPAR
{
}
| methoddeclarator LBRA RBR A
{
}
;
formalparameterlist : formalparameter
{
}
| formalparameterlist2 COMA
formalparameter

{
}
| /* epsilon */
{
}
;
formalparameterlist2 : formalparameter
{
}
| formalparameterlist2 COMA
formalparameter
{
}
;
formalparameter : type variabledeclaratorid
{
}
;
throws : THROWS classtypelist
{
}
```

```
| /* epsilon */
{
}
;
classtypelist : classtype
{
}
| classtypelist COMA classtype
{
}
;
methodbody : block PYCOMA
{
}
;
staticinitializer : STATIC block
{
}
;
constructordeclaration : modifiers constructordeclarator
throws constructorbody

{
}
;
constructordeclarator : simplename LPAR
formalparameterlist RPAR
{
}
;
constructorbody : LPAR2 explicitconstructorinvocation
blockstatements RPAR2
{
}
;
```

```
explicitconstructorinvocation : THIS LPAR argumentlist
RPAR PYCOMA
{
}
| SUPER LPAR argumentlist RPAR
PYCOMA
{
}
;
interfacedeclaration : modifiers INTERFACE identifier
extendsinterfaces interfacebody
{
}
;
extendsinterfaces : EXTENDS interfacetype
{
}
| extendsinterfaces2 COMA interfacetype
{
}
| /* epsilon */
{
}
;
extendsinterfaces2 : EXTENDS interfacetype
{
}
| extendsinterfaces2 COMA interfacetype
{
}

;
interfacebody : LPAR2 interfacememb erdeclarations
RPAR2
{
```

```
}
;
interfacememberdeclarations :
interfacememberdeclaration
{
}
| interfacememberdeclarations2
interfacememberdeclaration
{
}
| /* epsilon */
{
}
;
interfacememberdeclarations2 :
interfacememberdeclaration
{
}
| interfacememberdeclarations2
interfacememberdeclarati on
{
}
;
interfacememberdeclaration : constantdeclaration
{
}
| abstractmethoddeclaration
{
}
;
constantdeclarat ion : fielddeclaration
{
}
;
abstractmethoddeclaration : methodheader PYCOMA
```

```
{
}
;

arrayinitializer : LPAR2 variableinitializers RPAR2
{
}
;
variableinitializers : variableinitializer
{
}
| variableinitializers COMA
variableinitializer
{
}
| /* epsilon */
{
}
;
block : LPAR2 blockstatements RPAR2
{
}
;
blockstatements : blockstatement
{
}
| blockstatements2 blockstatement
{
}
| /* epsilon */
{
}
;
blockstatements2 : blockstatement
{
```

```
}
| blockstatements2 blockstatement
{
}
;
blockstatement : localvariabledeclarationstatement
{
}
| statement
{
}
;
localvariabledeclarationstatement :

localvariabledeclaration PYCOMA
{
}
;
localvariabledeclaration : type variabledeclarators
{
}
;
statement : statementwithouttrailingsubstatement
{
}
| labeledstatement
{
}
| ifthenstatement
{
}
| ifthenelsestatement
{
}
| whilestatement
```

```
{
}
| forstatement
{
}
;
statementnoshortif :
statementwithouttrailingsubstatement
{
}
| labeledstatementnoshortif
{
}
| ifthenelsestatementnoshortif
{
}
| whilestatementnoshortif
{
}
| forstatementnoshortif
{
}
;
statementwithouttrailingsub statement : block
{

}
| emptystatement
{
}
| expressionstatement
{
}
| switchstatement
{
```

```
}
| dostatement
{
}
| breakstatement
{
}
| continuestatement
{
}
| returnstatement
{
}
| synchronizedstatement
{
}
| throwstatement
{
}
| trystatement
{
}
;
emptystatement : PYCOMA
{
}
;
labeledstatement : identifier DOSP statement
{
}
;
labeledstatementnoshortif : identifier DOSP
statementnoshortif
{
}
```

```
;
expressionstatement : statementexpression PYCOMA

{
}
;
statementexpression : assignment
{
}
| preincrementexpression
{
}
| predecrementexpression
{
}
| postincrementexpression
{
}
| postdecrementexpression
{
}
| methodinvocation
{
}
| classinstancecreationexpression
{
}
;
ifthenstatement : IF LPAR expression RPAR statement
{
}
;
ifthenelsestatement : IF LPAR expression RPAR
statementnoshortif ELSE statement
{
```

```
}
;
ifthenelsestatementnoshortif : IF LPAR expression
RPAR statementnoshortif ELSE statementnoshortif
{
}
;
switchstatement : SWITCH LPAR expression RPAR
switchblock
{
}
;

switchblock : LPAR2 switchblockstatementgroups
switchlabels RPAR2
{
}
;
switchblockstatementgroups :
switchblockstatementgroup
{
}
| switchblockstatementgroups
switchblockstatementgroup
{
}
;
switchblockstatementgroup : switchlabels2
blockstatements2
{
}
;
switchlabels : switchlabel
{
}
```

```
| switchlabels2 switchlabel
{
}
| /* epsilon */
{
}
;
switchlabels2 : switchlabel
{
}
| switchlabels2 switchlabel
{
}
;
switchlabel : CASE constantexpression DOSP
{
}
| DEFAULT DOSP
{
}
;

whilestatement : WHILE LPAR expression RPAR
statement
{
}
;
whilestatementnoshortif : WHILE LPAR expression
RPAR statementnoshortif
{
}
;
dostatement : DO statement WHILE LPAR expressio n
RPAR PYCOMA
{
```

```
}
;
forstatement : FOR LPAR forinit PYCOMA expressione
PYCOMA forupdate RPAR statement
{
}
;
forstatementnoshortif : FOR LPAR forinit PYCOMA
expressione PYCOMA forupdate RPAR
statementnoshortif
{
}
;
forinit : statementexpressionlist
{
}
| localvariabledeclaration
{
}
| /* epsilon */
{
}
;
forupdate : stat ementexpressionlist
{
}
| /* epsilon */
{
}
;

statementexpressionlist : statementexpression
{
}
| statementexpressionlist COMA
```

```
statementexpression
{
}
;
breakstatement : BREAK identifiere PYCOMA
{
}
;
continuestatement : CONTINUE identifiere PYCOMA
{
}
;
returnstatement : RETURN expressione P YCOMA
{
}
;
throwstatement : THROW expression PYCOMA
{
}
;
synchronizedstatement : SYNCHRONIZED LPAR
expression RPAR block
{
}
;
trystatement : TRY block catches2
{
}
| TRY block catches finally
{
}
;
catches : catchclause
{
}
```

```
| catches2 catchclause
{
}
| /* epsilon */

{
}
;
catches2 : catchclause
{
}
| catches2 catchclause
{
}
;
catchclause : CATCH LPAR formalparameter RPAR
block
{
}
;
finally : FINALLY block
{
}
;
primary : primarynonewarray
{
}
| arraycreationexpression
{
}
;
primarynonewarray : literal
{
}
| THIS
```

```
{
}
| LPAR expression RPAR
{
}
| classinstancecreationexpression
{
}
| fieldaccess
{
}
| methodinvocation
{
}
| arrayaccess

{
}
;
classinstancecreationexpression : NEW classtype
LPAR argumentlist RPAR
{
}
;
argumentlist : expression
{
}
| argumentlist2 COMA expression
{
}
| /* epsilon */
{
}
;
argumentlist2 : expression
```

```
{
}
| argumentlist2 COMA expression
{
}
;
arraycreationexpression : NEW primitivetype dimexprs
dims
{
}
| NEW classorinterfacetype dimexprs
dims
{
}
;
dimexprs : dimexpr
{
}
| dimexprs dimexpr
{
}
;
dimexpr : LBRA expression RBRA
{
}

;
dims : LBRA RBRA
{
}
| dims2 LBRA RBRA
{
}
| /* epsilon */
{
```

```
}
;
dims2 : LBRA RBRA
{
}
| dims2 LBRA RBRA
{
}
;
fieldaccess : primary POINT identifier
{
}
| SUPER POINT identifier
{
}
;
methodinvocation : name LPAR argu mentlist RPAR
{
}
| primary POINT identifier LPAR
argumentlist RPAR
{
}
| SUPER POINT identifier LPAR
argumentlist RPAR
{
}
;
arrayaccess : name LB RA expression RBRA
{
}
| primarynonewarray LBRA expression RBRA
{
}
;
```

```
postfixexpression : primary
{
}
| name
{
}
| postincrementexpression
{
}
| postdecrementexpression
{
}
;
postincrementexpression : postfixexpression PPLUS
{
}
;
postdecrementexpression : postfixexpression MMIN
{
}
;
unaryexpression : preincrementexpression
{
}
| predecrementexpression
{
}
| PLUS unaryexpression
{
}
| MINUS unaryexpression
{
}
| unaryexpressionnotplusminus
```

```
{
}
;
preincrementexpression : PPLUS unaryexpression
{
}
;
predecrementexpression : MMIN unaryexpression
{
}


;
unaryexpressionnotplusminus : postfixexpression
{
}
| WAVE unaryexpression
{
}
| NO unaryexpression
{
}
| castexpression
{
}
;
castexpression : LPAR primitivetype dims RPAR
unaryexpression
{
}
| LPAR expression RPAR
unaryexpressionnotplusminus
{
}
| LPAR name dims2 RPAR
unaryexpressionnotplusminus
```

```
{
}
;
multiplicativeexpression : unaryexpression
{
}
| multiplicativeexpression STAR
unaryexpression
{
}
| multiplicativeexpression SLASH
unaryexpression
{
}
| multiplicativeexpression PERC
unaryexpression
{
}
;
additiveexpression : multiplicativeexpression
{
}

| additiveexpression PLUS
multiplicativeexpression
{
}
| additiveexpression MINU S
multiplicativeexpression
{
}
;
shiftexpression : additiveexpression
{
}
```

```
| shiftexpression LSH additiveexpression
{
}
| shiftexpression RSH additi veexpression
{
}
| shiftexpression DRSH additiveexpression
{
}
;
relationalexpression : shiftexpression
{
}
| relationalexpression RELOP
shiftexpression
{
}
| relationalexpression INSTANCEOF
referencetype
{
}
;
equalityexpression : relationalexpression
{
}
| equalityexpression EQ
relationalexpression
{
}
| equalityexpression NEQ
relationalexpression
{
}
;
```

```
andexpression : equalityexpression
{
}
| andexpression AND equalityexp ression
{
}
;
exclusiveorexpression : andexpression
{
}
| exclusiveorexpression HAT
andexpression
{
}
;
inclusiveorexpression : exclusiveor expression
{
}
| inclusiveorexpression BAR
exclusiveorexpression
{
}
;
conditionalandexpression : inclusiveorexpression
{
}
| conditionalandexpression DAND
inclusiveorexpression
{
}
;
conditionalorexpression : conditionalandexpression
{
}
| conditionalorexpression DBAR
```

```
conditionalandexpression
{
}
;
conditionalexpression : conditionalorexpression
{
}
| conditionalorexpression ASK
expression DOSP conditionalexpression
{

}
;
assignmentexpression : conditionalexpression
{
}
| assignment
{
}
;
assignment : lefthandside ASSOP
assignmentexpression
{
}
;
lefthandside : name
{
}
| fieldaccess
{
}
| arrayaccess
{
}
;
```

```
expression : assignmentexpression
{
}
;
expressione : expression
{
}
| /* epsilon */
{
}
;
constantexpression : expression
{
}
;
```

Anexo II: Gramática bison de C

```
s : MAIN LPAR RPAR bloque
{
}
;
bloque : LBRA bdecl seqinstr RBRA
{
}
;
bdecl : /* epsilon */
{
}
| decl
{
}
;
decl : decl decvar
{
}
| decvar
{
}
;
decvar : tipo lident PUNTOYCOMA
{
}
;
tipo : INT
{
}
| FLOAT
{
}
| CHAR
{
```

```
}
;
lident : lident COMA variable
{
}
| variable
{
}
;
variable : ID
{
}
;
seqinstr : seqinstr instr
{
}
| instr
{
}
;
instr : PUNTOYCOMA
{
}
| bloque
{
}
| expr PUNTOYCOMA
{
}
| PRINTF LPAR FORMATO COMA expr RPAR
PUNTOYCOMA
{
}
| PRINTF LPAR CADENA COMA CAD RPAR
PUNTOYCOMA
```

```
{
}
| SCANF LPAR FORMATO COMA REFERENCIA
ref RPAR PUNTOYCOMA
{
}
| IF LPAR expr RPAR ifmas instr
{
}
| IF LPAR expr RPAR ifmas instr ELSE instr
{
}
| WHILE LPAR expr RPAR instr
{
}
| FOR LPAR ref ASSOP expr PUNTOYCOMA
ebool PUNTOYCOMA expr RPAR instr
{
}
| SWITCH LPAR esimple RPAR LBRA
{
}
seqcasos RBRA
{
}
| BREAK PUNTOYCOMA
{
}
;
ifmas : /* epsilon */
{
}
;
seqcasos : caso seqcasos
{
```

```
}
| casodefault
{
}
| caso
{
}
;
caso : CASE NINT DOSPUNTOS seqinstrswitch
{
}
| CASE CTECHAR DOSPUNTOS seqinstrswitch
{
}
;
seqinstrswitch : seqinstr
{
}
| /* epsilon */
{
}
;
casodefault : DEFAULT DOSPUNTOS seqinstrswitch
{
}
;
expr : ref ASSOP expr
{
}
| ebool
{
}
;
ebool : ebool OBOOL econj
{
```

```
}
| econj
{
}
;
econj : econj YBOOL ecomp
{
}
| ecomp
{
}
;
ecomp : ecomp RELOP esimple
{
}
| esimple
{
}
;
esimple : esimple ADDOP term
{
}
| ADDOP term
}
| term
{
}
;
term : term MULOP factor
{
}
| factor
{
}
;
```

```
factor : ref
{
}
| NINT
{
}
| NFIX
{
}
| NOBOOL factor
{
}
| LPAR tipo RPAR factor
{
}
| CTECHAR
{
}
;
ref : ID
{
}
| LPAR expr RPAR
{
}
;
```

Anexo III: Gramática bison de ARBR

```
s : seqins
{
};
seqins: inst seqins
{
};
seqins:
{
};
inst: ETIQ
{
}
inst: SALTO_CERO LPAR func COMA ETIQ RPAR
{
}
inst: SALTO_NOCERO LPAR func COMA ETIQ
RPAR
{
}
inst: SALTO LPAR ETIQ RPAR
{
}
inst: func
{
};
inst: HALT
{
};
inst: WRITE_ENT LPAR func RPAR
{
};
inst : WRITE_CAR LPAR func RPAR
{
};
```

```
inst : WRITE_LIN
{
}
func: ENTERO
{
};
func: IGUAL_ENT LPAR func COMA func RPAR
{
}
func: STORE_ENT LPAR func COMA func RPAR
{
}
func: LOAD_ENT LPAR func RPAR
{
}
func: SUMA_ENT LPAR func COMA func RPAR
{
};
func: RESTA_ENT LPAR func COMA func RPAR
{
};
func: MUL_ENT LPAR func COMA func RPAR
{
};
func: DIV_ENT LPAR func COMA func RPAR
{
};
func: AND_ENT LPAR func COMA func RPAR
{
};
func: OR_ENT LPAR func COMA func RPAR
{
};
```

Anexo IV: ETDS de ARBR a Linux ASM

Reglas de traducción

```
s : seqins
  {
    $$.trad = concat2(".text\n.globl _start\n _start:\n", $1.trad);
    trad = $$.trad;
  };

seqins: inst seqins
  {
    $$.trad = concat2($1.trad, $2.trad);


  };

seqins:
  {
    $$.trad = malloc(1);
    $$.trad[0] = 0;
  };

inst: ETIQ
  {
    $$.trad = concat2($1.trad,":\n");
  }

inst: SALTO_CERO LPAR func COMA ETIQ RPAR
  {
    $$.trad = concat4($3.trad,"\tjnz\t",$5.trad,"\n");
  }

inst: SALTO_NOCERO LPAR func COMA ETIQ RPAR
  {
    $$.trad = concat4($3.trad,"\tjz\t",$5.trad,"\n");
  }
```

```
inst: SALTO LPAR ETIQ RPAR
  {
   $$.trad = concat3("\tjmp\t",$3.trad,"\n");
  }

inst: func
  {
   $$ = $1;
  };

inst: HALT
  {
    $$.trad = strdup("\tmovl\t$1, %eax\n\txorl\t%ebx,%ebx\n\tint\t$0x80\n");
  };

inst: WRITE_ENT LPAR func RPAR
  {
    $$.trad = concat4($3.trad, "\tpushl\t", $3.fvalue,
                  "\n\tcall\twrite_ent\n\taddl\t$4, %esp\n");
  };

inst : WRITE_CAR LPAR func RPAR
  {
    $$.trad = concat4($3.trad, "\tpushl\t", $3.fvalue,
                  "\n\tcall\twrite_car\n\taddl\t$4, %esp\n");
  };

inst : WRITE_LIN
  {
   $$.trad = strdup("\tcall\twrite_lin\n");
  }

func: ENTERO
  {
   $$.trad = malloc(1);
```

```
   $$.trad[0] = 0;
   $$.fvalue = strdup(yytext);
   $$.fvalue[0] = '$';
   $$.type = INMEDIATO;
   $$.registro = -1;
  };

func: IGUAL_ENT LPAR func COMA func RPAR
  {
   int registro, reg2, guardar = 0, count;
   char * temp1, * temp2, * temp3, * old1, * old2;
   temp1 = concat2($3.trad, $5.trad);

   if ($5.type == REGISTRO) {

    reg2 = registro_libre();
    if (reg2 == -1) {
     fprintf(stderr, "No puedo encontrar en registro libre para operar.\n");
     exit(-1);
    }

    temp2 = concat2( "\tcmpl\t",concat6($3.fvalue,", ",$5.fvalue,
                "\n\tsete\t",intel[registro].reg8,
                "\n"));
    $$.fvalue = strdup(intel[registro].reg32);
    $$.registro = reg2;
    $$.trad = concat2(temp1, temp2);
    if ($3.registro != -1)
     intel[$3.registro].count = 0;
    free(temp1);
    free(temp2);
   } else {
    registro = registro_libre($5.registro);
    if (registro == -1) {
     for (count = 0; count < 6; count++)
```

```
      if ($5.registro != count)
        break;
    registro = count;
    guardar = 1;
  }
  if (!guardar)
    temp2 = concat5("\tmovl\t", $5.fvalue, ", ", intel[registro].reg32,"\n");
  else {
    temp2 = concat3("\tpush\t", intel[registro].reg32,"\n");
    old1 = temp2;
    temp2 = concat2(old1, temp2);
    free(old1);
  }
  temp3 = concat2( "\tcmpl\t",concat6($3.fvalue,", ",
                   intel[registro].reg32,
                   "\n\tsete\t",intel[registro].reg8,
                   "\n"));

  if (guardar) {
    old1 = temp3;
    temp3 = concat4(temp3, "\tpopl\t", intel[registro].reg32,"\n");
    free(old1);
  }
  $$.trad = concat3(temp1, temp2, temp3);
  $$.type = REGISTRO;
  $$.fvalue = strdup(intel[registro].reg32);
  $$.registro = registro;
  if ($3.registro != -1)
    intel[$3.registro].count = 0;
  if ($5.registro != -1)
    intel[$5.registro].count = 0;
  free(temp1);
  free(temp2);
 }
}
```

```
func: STORE_ENT LPAR func COMA func RPAR
 {
  int count, guardar = 0, registro, aux;
  char c, * temp,aux2[16];

  $$.type = MEMORIA;

  if ($5.type == INMEDIATO) {
     aux = atoi( ( & ( ($5.fvalue)[1] ) ) ) << 2;
     sprintf(aux2,"%d",aux);
     $$.fvalue = concat2("variables+",aux2);

     if ($3.type == REGISTRO)
       $$.trad = concat6($3.trad,"\n\tmovl\t",
                 $3.fvalue,", ",$$.fvalue,"\n");
     else
     {
       registro = registro_libre();
       if (registro == -1) {
          fprintf(stderr, "No puedo encontrar en registro libre para operar.\n");
          exit(-1);
       }
       $$.trad = concat8($3.trad,"\n\tmovl\t",
                 $3.fvalue,", ",intel[registro].reg32,
                 "\n\tmovl\t",intel[registro].reg32,
                 concat3(", ",$$.fvalue,"\n"));
     }

     $$.registro = -1;
   } else
  if ($5.type == REGISTRO) {
     $$.fvalue = concat3("variables(", $5.fvalue,",4)");

     if ($3.type == REGISTRO)
```

```
      $$.trad = concat6($3.trad,"\n\tmovl\t",
                  $3.fvalue,", ",$$.fvalue,"\n");
    else
    {
      registro = registro_libre();
      if (registro == -1) {
          fprintf(stderr, "No puedo encontrar en registro libre para operar.\n");
          exit(-1);
      }
      $$.trad = concat8($3.trad,"\n\tmovl\t",
                  $3.fvalue,", ",intel[registro].reg32,
                  "\n\tmovl\t",intel[registro].reg32,
                  concat3(", ",$$.fvalue,"\n"));
    }
    $$.registro = $5.registro;
   } else
  if ($5.type == MEMORIA) {
    if ($5.registro != -1)
        registro = $5.registro;
    else {
        registro = registro_libre();
        if (registro == -1) {
            fprintf(stderr, "No puedo encontrar en registro libre para
operar.\n");
            exit(-1);
          }
      }
    temp = concat5("\tmovl\t", $3.fvalue, ", ", intel[registro].reg32,"\n");
    $$.fvalue = concat3("(", intel[registro].reg32,")");
    $$.trad = concat3($3.trad, temp,
                concat5("\n\tmovl\t",
                $3.fvalue,", ",$$.fvalue,"\n"));
    $$.registro = registro;
    free(temp);
   }
```

```
}

func: LOAD_ENT LPAR func RPAR
 {
   int count, guardar = 0, registro, aux;
   char c, * temp,aux2[16];

   if ($3.type == INMEDIATO) {
        aux = atoi( ( & ( ($3.fvalue)[1] ) ) ) << 2;
        sprintf(aux2,"%d",aux);
        $$.fvalue = concat2("variables+",aux2);
        $$.trad = strdup($3.trad);
        $$.registro = -1;
     } else
   if ($3.type == REGISTRO) {
        $$.fvalue = concat3("variables(", $3.fvalue,",4)");
        $$.trad = strdup($3.trad);
        $$.registro = $3.registro;
      } else
   if ($3.type == MEMORIA) {
       if ($3.registro != -1)
           registro = $3.registro;
       else {
           registro = registro_libre();
           if (registro == -1) {
               fprintf(stderr, "No puedo encontrar en registro libre para
operar.\n");
               exit(-1);
             }
         }
       temp = concat5("\tmovl\t", $3.fvalue, ", ", intel[registro].reg32,"\n");
       $$.fvalue = concat3("(", intel[registro].reg32,")");
       $$.trad = concat2($3.trad, temp);
       $$.registro = registro;
       free(temp);
```

```
    }
  $$.type = MEMORIA;
 }

func: SUMA_ENT LPAR func COMA func RPAR
 {
  $$ = traduce_op_ent("\taddl\t", $3, $5, 1);
 };

func: RESTA_ENT LPAR func COMA func RPAR
 {
  $$ = traduce_op_ent("\tsubl\t", $3, $5, 0);
 };

func: MUL_ENT LPAR func COMA func RPAR
 {
  $$ = traduce_divmul_ent("\timull\t", $3, $5);
 };

func: DIV_ENT LPAR func COMA func RPAR
 {
  $$ = traduce_divmul_ent("\tidivl\t", $3, $5);
 };

func: AND_ENT LPAR func COMA func RPAR
 {
  $$ = traduce_log_ent("\tandl\t", $3, $5);
 };

func: OR_ENT LPAR func COMA func RPAR
 {
  $$ = traduce_log_ent("\torl\t", $3, $5);
 };
```

Funciones auxiliares

```
int yyerror(char * s)
  {
   fprintf(stderr, "Error (%d:%d): token '%s' incorrecto\n", nlin, ncol -
          strlen(yytext), yytext);
   exit(-1);
  }

YYSTYPE  traduce_op_ent(char * op, YYSTYPE s1, YYSTYPE s2, int com)

   {
     YYSTYPE  SS;
     int registro, guardar = 0, count;
     char * temp1, * temp2, * temp3, * old1, * old2;
     if (s1.type == REGISTRO) {
         temp1 = concat2(s1.trad, s2.trad);
         temp2 = concat5(op, s2.fvalue, ", ", s1.fvalue,"\n");
         if (s2.registro !=-1)
           intel[s2.registro].count = 0;
         SS.type = REGISTRO;
         SS.fvalue = strdup(s1.fvalue);
         SS.registro = s1.registro;
         SS.trad = concat2(temp1, temp2);
         free(temp1);
         free(temp2);
       } else
     if (s2.type == REGISTRO && com) {
         temp1 = concat2(s1.trad, s2.trad);
         temp2 = concat5(op, s1.fvalue, ", ", s2.fvalue,"\n");
         SS.type = REGISTRO;
         SS.fvalue = strdup(s2.fvalue);
         SS.registro = s2.registro;
         SS.trad = concat2(temp1, temp2);
         if (s1.registro != -1)
              intel[s1.registro].count = 0;
```

```
free(temp1);
free(temp2);
} else {
temp1 = concat2(s1.trad, s2.trad);
registro = registro_libre(s2.registro);
if (registro == -1) {
     for (count = 0; count < 6; count++)
        if (s2.registro != count)
               break;
     registro = count;
     guardar = 1;
   }
if (!guardar)
   temp2 = concat5("\tmovl\t", s1.fvalue, ", ",
intel[registro].reg32,"\n");
else {
   temp2 = concat3("\tpush\t", intel[registro].reg32,"\n");
   old1 = temp2;
   temp2 = concat2(old1, temp2);
   free(old1);
  }

temp3 = concat5(op, s2.fvalue,
            ", ", intel[registro].reg32, "\n");

if (guardar) {
    old1 = temp3;
    temp3 = concat4(temp3, "\tpopl\t", intel[registro].reg32,"\n");
    free(old1);
   }
SS.trad = concat3(temp1, temp2, temp3);
SS.type = REGISTRO;
SS.fvalue = strdup(intel[registro].reg32);
SS.registro = registro;
if (s2.registro != -1)
```

```
            intel[s2.registro].count = 0;
        if (s1.registro != -1)
            intel[s1.registro].count = 0;
        free(temp1);
        free(temp2);
     }
   return(SS);
  }

YYSTYPE  traduce_divmul_ent(char * op, YYSTYPE s1, YYSTYPE s2)

  {
    YYSTYPE  SS;
    int registro, guardar = 0, count;
    char * temp1, * temp2, * temp3, * old1, * old2;
    temp1 = concat2(s1.trad, s2.trad);
    intel[0].count++;
    if (s2.type == REGISTRO) {
        temp2 = concat8("\tpushl\t%eax\n\tmovl\t",
                  s1.fvalue,", %eax\n",op,s2.fvalue,
                  "\n\tmovl\t%eax, ",s2.fvalue,
                  "\n\tpopl\t%eax\n");
        SS.type = REGISTRO;
        SS.fvalue = strdup(s2.fvalue);
        SS.registro = s2.registro;
        SS.trad = concat2(temp1, temp2);
        if (s1.registro != -1)
            intel[s1.registro].count = 0;
        free(temp1);
        free(temp2);
      } else {
        registro = registro_libre(s2.registro);
        if (registro == -1) {
            for (count = 0; count < 6; count++)
               if (s2.registro != count)
```

```
                break;
        registro = count;
        guardar = 1;
       }
    if (!guardar)
      temp2 = concat5("\tmovl\t", s2.fvalue, ", ",
intel[registro].reg32,"\n");
    else {
      temp2 = concat3("\tpush\t", intel[registro].reg32,"\n");
      old1 = temp2;
      temp2 = concat2(old1, temp2);
      free(old1);
     }

    temp3 = concat8("\tpushl\t%eax\n\tmovl\t",
              s1.fvalue,", %eax\n",
              op, intel[registro].reg32,
              "\n\tmovl\t%eax, ",intel[registro].reg32,
              "\n\tpopl\t%eax\n");

    if (guardar) {
        old1 = temp3;
        temp3 = concat4(temp3, "\tpopl\t", intel[registro].reg32,"\n");
        free(old1);
       }
    SS.trad = concat3(temp1, temp2, temp3);
    SS.type = REGISTRO;
    SS.fvalue = strdup(intel[registro].reg32);
    SS.registro = registro;
    if (s2.registro != -1)
        intel[s2.registro].count = 0;
    if (s1.registro != -1)
        intel[s1.registro].count = 0;
    free(temp1);
    free(temp2);
```

```
   }
  intel[0].count--;
  return(SS);
 }

YYSTYPE  traduce_log_ent(char * op, YYSTYPE s1, YYSTYPE s2)

 {
   YYSTYPE  SS;
   int registro1, registro2, guardar = 0, count;
   char * temp1, * temp2, * temp3, * temp4, * old1, * old2;
   temp1 = concat2(s1.trad, s2.trad);
   if (s1.type == REGISTRO) {
       registro1 = s1.registro;
       temp2 = concat5("\torl\t", s1.fvalue, ", ", s1.fvalue,"\n");
       old1 = temp2;

       temp2 = strdup(old1);
       free(old1);
       SS.fvalue = strdup(s1.fvalue);
   } else {
       registro1 = registro_libre(-1);
       temp2 = concat5("\tmovl\t", s1.fvalue, ", ",
                 intel[registro1].reg32, "\n");
       old1  = temp2;
       temp2 = concat6(temp2, "\torl\t",intel[registro1].reg32, ", "
                 ,intel[registro1].reg32, "\n");
       old2  = temp2;

       temp2 = strdup(old2);
       free(old2);
       free(old1);
       SS.fvalue = strdup(intel[registro1].reg32);
     }
   if (s2.type == REGISTRO) {
```

```
        registro2 = s2.registro;
        temp3 = concat5("\torl\t", s2.fvalue, ", ", s2.fvalue,"\n");
        old1 = temp3;

        temp3 = strdup(old1);
        free(old1);
        SS.fvalue = strdup(s2.fvalue);
      } else {
        registro2 = registro_libre(-1);
        temp3 = concat5("\tmovl\t", s2.fvalue, ", ",
                    intel[registro2].reg32, "\n");
        old1  = temp3;
        temp3 = concat6(temp3, "\torl\t",intel[registro2].reg32, ", "
                    ,intel[registro2].reg32, "\n");
        old2  = temp3;

        temp3 = strdup(old2);
        free(old2);
        free(old1);
        SS.fvalue = strdup(intel[registro1].reg32);
      }
    temp4 = concat5(op, intel[registro2].reg32, ", ",intel[registro1].reg32,"\n");
    SS.trad = concat4(temp1, temp2, temp3, temp4);
    SS.type = REGISTRO;
    SS.registro = registro1;
    intel[registro2].count = 0;
    free(temp1);
    free(temp2);
    free(temp3);
    free(temp4);
    return(SS);
  }

int main(int argc, char ** argv)
  {
```

```
FILE *fent;
int  fd_s, count, size, status;
char * name_s;
char * name_o;
char * name_elf;
if (argc == 2)
{
  fent = fopen(argv[1],"r");
  if (fent) {
    yyin = fent;
    yyparse();
    fclose(fent);
    name_elf = concat2(argv[1],".out");
    name_s = concat2(argv[1], ".s");
    name_o = concat2(argv[1], ".o");
    fd_s = open(name_s, O_WRONLY | O_CREAT | O_TRUNC, 0666);
    write(fd_s, trad, strlen(trad));
    if (!fork())
      execlp("as", "as", name_s,"-o", name_o, NULL);
    wait(NULL);
    if (!fork())
      execlp("ld","ld", name_o, "mcclib.o", "-o", name_elf, NULL);
    wait(NULL);
  }
  else
  {
    fprintf(stderr,"Error: no se puede abrir el fichero %s\n",argv[1]);
    return -1;
  }
}
else
{
  fprintf(stderr,"USO: %s nom_fichero\n",argv[0]);
  return -1;
}
```

```
    return 0;
   }

int registro_libre(int no_reg)

{
 int reg;
 for (reg = 0; reg < 6; reg++)
     if (!intel[reg].count && reg != no_reg) {
          intel[reg].count = 1;
          return reg;
         }
 return  -1;
}

char * concat2(char *s1, char *s2)

{
 int l;
 char *res;
 l = strlen(s1) + strlen(s2) + 2;
 res = (char *) malloc(l);
 sprintf(res,"%s%s", s1, s2);
 return(res);
}


char * concat3(char *s1, char *s2, char * s3)

{
 int l;
 char *res;
 l = strlen(s1) + strlen(s2) + strlen(s3)+ 2;
 res = (char *) malloc(l);
 sprintf(res,"%s%s%s", s1, s2, s3);
```

```
  return(res);
}

char * concat4(char *s1, char *s2, char * s3, char *s4)

{
  int l;
  char *res;
  l = strlen(s1) + strlen(s2) + strlen(s3) + strlen(s4) + 2;
  res = (char *) malloc(l);
  sprintf(res,"%s%s%s%s", s1, s2, s3, s4);
  return(res);
}


char * concat5(char *s1, char *s2, char * s3, char *s4, char *s5)

{
  int l;
  char *res;
  l = strlen(s1) + strlen(s2) + strlen(s3) + strlen(s4) + strlen(s5) + 2;
  res = (char *) malloc(l);
  sprintf(res,"%s%s%s%s%s", s1, s2, s3, s4, s5);
  return(res);
}

char * concat6(char *s1, char *s2, char * s3, char *s4, char *s5, char *s6)
{
  int l;
  char *res;
  l = strlen(s1) + strlen(s2) + strlen(s3) + strlen(s4) + strlen(s5) + strlen(s6) + 2;
  res = (char *) malloc(l);
  sprintf(res,"%s%s%s%s%s%s", s1, s2, s3, s4, s5, s6);
  return(res);
}
```

```
char * concat8(char *s1, char *s2, char * s3, char *s4, char *s5, char *s6,
               char *s7, char *s8)
{
  int l;
  char *res;
  l = strlen(s1) + strlen(s2) + strlen(s3) + strlen(s4) + strlen(s5)
                + strlen(s6) + strlen(s7) + strlen(s8) + 2;
  res = (char *) malloc(l);
  sprintf(res,"%s%s%s%s%s%s%s%s", s1, s2, s3, s4, s5, s6, s7, s8);
  return(res);
}
```