Procesamiento de Lenguajes (PL)
Curso 2014/2015

Práctica 5: traductor a código `m2r`

Fecha y método de entrega

La práctica debe realizarse de forma individual o por parejas[1], y debe entregarse a través del servidor de prácticas del DLSI **antes de las 23:59 del 22 de mayo de 2015**.

Al servidor de prácticas del DLSI se puede acceder de dos maneras:

- Desde la web del DLSI (http://www.dlsi.ua.es), en el apartado "Entrega de prácticas"
- Desde la URL http://pracdlsi.dlsi.ua.es

Una vez en el servidor, se debe elegir la asignatura PL y seguir las instrucciones. En el caso de que la práctica la haya realizado una pareja, debe entregarla solamente una de las dos personas de la pareja, y poner en un comentario los nombres y DNIs de los dos componentes de la pareja.

Descripción de la práctica

- La práctica 5 consiste en realizar un compilador para el lenguaje fuente que se describe más adelante, que genere código para el lenguaje objeto `m2r`, utilizando como en la práctica anterior `bison` y `flex`. Los fuentes deben llamarse `plp5.l` y `plp5.y`. Para compilar la práctica se utilizarán las siguientes órdenes:

```
$ flex plp5.l
$ bison -d plp5.y
$ g++ -o plp5 plp5.tab.c lex.yy.c
```

- El compilador debe diseñarse teniendo en cuenta las siguientes restricciones:

  1. En ningún caso se permite que el código objeto se vaya imprimiendo por la salida estándar conforme se realiza el análisis, debe imprimirse al final.
  2. Tampoco se permite utilizar ningún *buffer* o *array* global para almacenar el código objeto generado, el código objeto que se vaya generando debe circular por los atributos de las variables de la gramática (en memoria dinámica), y al final, en la regla correspondiente al símbolo inicial, se imprimirá por la salida estándar el código objeto; éste debe ser el único momento en que se imprima algo por la salida estándar. Se recomienda utilizar el tipo `string` para almacenar el código generado.
  3. Se aconseja utilizar muy pocas variables globales, especialmente en las acciones semánticas de las reglas; en ese caso siempre es aconsejable utilizar atributos. El uso de variables globales está, en general, reñido con la existencia de estructuras recursivas tales como las expresiones o los bloques de instrucciones. Se pueden utilizar variables globales temporales mientras se usen sea únicamente dentro de una misma acción semántica, sin efectos fuera de ella.

- Si se produce algún tipo de error, léxico, sintáctico o semántico, el programa enviará un mensaje de error de una sola línea a la salida de error (`stderr`). Se proporcionará una función "`msgError`" que se encargará de generar los mensajes de error correctos, a la que se debe pasar la fila, la columna y el lexema del token más relacionado con el error; en el caso de los errores léxicos y sintácticos, el lexema será la cadena "`yytext`" que proporciona el analizador léxico.

[1] Una pareja está compuesta por exactamente dos personas, no tres, ni cuatro, ...

Especificación sintáctica del lenguaje fuente

La sintaxis del lenguaje fuente puede ser representada por la gramática siguiente (que puede ser modificada para facilitar el diseño del compilador):

S	$\longrightarrow$	*FVM*
FVM	$\longrightarrow$	*DVar FVM*
FVM	$\longrightarrow$	**int main pari pard** *Bloque*
Tipo	$\longrightarrow$	**int**
Tipo	$\longrightarrow$	**float**
Bloque	$\longrightarrow$	**llavei** *BDecl SeqInstr* **llaved**
BDecl	$\longrightarrow$	*BDecl DVar*
BDecl	$\longrightarrow$	ϵ
DVar	$\longrightarrow$	*Tipo LIdent* **pyc**
LIdent	$\longrightarrow$	*LIdent* **coma** *Variable*
LIdent	$\longrightarrow$	*Variable*
Variable	$\longrightarrow$	**id** *V*
V	$\longrightarrow$	ϵ
V	$\longrightarrow$	**cori nentero cord** *V*
SeqInstr	$\longrightarrow$	*SeqInstr Instr*
SeqInstr	$\longrightarrow$	ϵ
Instr	$\longrightarrow$	**pyc**
Instr	$\longrightarrow$	*Bloque*
Instr	$\longrightarrow$	*Ref* **asig** *Expr* **pyc**
Instr	$\longrightarrow$	**printf pari formato coma** *Expr* **pard pyc**
Instr	$\longrightarrow$	**scanf pari formato coma referencia** *Ref* **pard pyc**
Instr	$\longrightarrow$	**if pari** *Expr* **pard** *Instr*
Instr	$\longrightarrow$	**if pari** *Expr* **pard** *Instr* **else** *Instr*
Instr	$\longrightarrow$	**while pari** *Expr* **pard** *Instr*
Expr	$\longrightarrow$	*Expr* **relop** *Esimple*
Expr	$\longrightarrow$	*Esimple*
Esimple	$\longrightarrow$	*Esimple* **addop** *Term*
Esimple	$\longrightarrow$	*Term*
Term	$\longrightarrow$	*Term* **mulop** *Factor*
Term	$\longrightarrow$	*Factor*
Factor	$\longrightarrow$	*Ref*
Factor	$\longrightarrow$	**nentero**
Factor	$\longrightarrow$	**nreal**
Factor	$\longrightarrow$	**pari** *Expr* **pard**
Ref	$\longrightarrow$	**id**
Ref	$\longrightarrow$	*Ref* **cori** *Esimple* **cord**

Especificación léxica del lenguaje fuente

Los componentes léxicos de este lenguaje fuente son los siguientes:

nentero: un número entero sin signo

nreal: un número en coma fija sin signo

id: un identificador

addop: los operadores de suma '`+`' o resta '`-`'

mulop: los operadores de producto '`*`' o división '`/`'

relop: los operadores de comparación '`==`' (igual), '`!=`' (distinto), '`<`' (menor), '`>`' (mayor), '`>=`' (mayor o igual), y '`<=`' (menor o igual)

coma: la coma, '`,`'

asig: el operador de asignación, '`=`'

pari: el paréntesis izquierdo, '`(`'

pard: el paréntesis derecho, '`)`'

cori: el corchete izquierdo, '`[`'

cord: el corchete derecho, '`]`'

llavei: la llave izquierda, '{'

llaved: la llave derecha, '}'

pyc: el carácter punto y coma, '`;`'

formato: el formato puede ser '`"%d"`' (entero) o bien '`"%g"`' (real)

referencia: el operador de referencia, '`&`'

Palabras reservadas: '**main**', '**int**', '**float**', '**printf**', '**scanf**', '**if**', '**else**', '**while**'

El analizador léxico también debe ignorar los blancos[2] y los comentarios, que comenzarán por '`//`' y terminarán al final de la línea, como en C++.

Especificación semántica del lenguaje fuente

Las reglas semánticas de este lenguaje son similares a las de C/C++, en muchos casos es necesario hacer conversiones de tipos, y en algunos casos se debe producir un mensaje de error semántico cuando aparezca alguna construcción no permitida. Las reglas se pueden resumir como:

1. No es posible declarar dos veces un símbolo. Tampoco es posible utilizar un identificador sin haberlo declarado previamente.
2. Si al declarar una variable el espacio ocupado por ésta sobrepasa el tamaño máximo de memoria para variables (siempre inferior a 16384), el compilador debe producir un mensaje de error indicando el lexema de la variable que ya no cabe en memoria. El tamaño de todos los tipos básicos es 1, es decir, una variable simple ocupa una única posición, ya sea de tipo entero o real.
3. En las expresiones aritméticas y relacionales es posible combinar cualquier expresión de cualquiera de los tipos simples (entero o real) con cualquier otra expresión de otro tipo simple. El compilador debe generar las conversiones de tipo necesarias
4. Aunque los argumentos de los operadores relacionales pueden ser enteros o reales (o mezclas), el resultado será siempre un entero que será 0 (falso) o 1 (cierto).[3]

[2] Los tabuladores se contarán como un espacio en blanco.

[3] Las instrucciones de `m2r` para los operadores relacionales generan un número entero que vale 0 o 1.

5. La semántica de las instrucciones **if** y **while** es similar a la que tienen en C: si la expresión (sea del tipo simple que sea) tiene un valor distinto de cero, se considera como cierta; si el valor es cero, se considera como falsa.

6. En los bloques, las declaraciones sólo son válidas en el cuerpo del bloque, es decir, una vez que se cierra el bloque, el compilador debe *olvidar* todas las variables declaradas en él. Además, es posible declarar en un bloque variables con el mismo nombre que las variables de otros bloques de niveles superiores; en el caso en que aparezca una referencia a una variable que se ha declarado en más de un bloque, se entenderá que se refiere a la declaración (en un bloque no cerrado, por supuesto) más cercana al lugar en que se utiliza la variable. Esta construcción y su semántica son similares a las de los bloques entre llaves de C.

7. La división entre valores enteros siempre es una división entera; si alguno de los operandos es real, la división es real (y puede ser necesario convertir uno de los operandos a real).

8. En la instrucción **printf** se debe generar las conversiones de tipo necesarias, y se debe generar una instrucción `wrl` (como si hubiera un "`\n`" en el formato) después de escribir el valor de la expresión. De igual forma, en **scanf** se debe generar las conversiones de tipo necesarias.

9. No se permite utilizar una variable de tipo *array* con más ni con menos índices de los necesarios (según la declaración de la variable) para obtener un valor de tipo simple.

10. No se permite poner índices (corchetes) a variables que no sean de tipo *array*.

11. No está permitida la asignación entre valores de tipo *array*. Las asignaciones deben realizarse siempre con valores de tipo simple.

12. En las declaraciones de variables de tipo *array*, el número debe ser estrictamente mayor que cero. El rango de posiciones del *array* será, como en C, de 0 a $n - 1$.

13. La expresión entre corchetes (el índice) debe ser de tipo entero (aunque C permita otros tipos que se convierten automáticamente a entero).

14. En principio, el número de dimensiones que puede tener una variable de tipo *array* no está limitado más que por la memoria disponible. Por este motivo es aconsejable utilizar una tabla de tipos.

Ampliación opcional: funciones

Para obtener más nota (y optar a la máxima calificación en la asignatura) se puede realizar la siguiente ampliación, que consiste en añadir la posibilidad de declarar y usar funciones al lenguaje anterior. En caso de optar por esta ampliación, es conveniente empezar directamente considerando que puede haber funciones, ya que cambia bastante la generación de código para variables temporales y para *arrays*.

Para implementar esta ampliación habría que añadir las siguientes reglas a la gramática anterior, y la palabra reservada `return` a la especificación léxica:

$$
\begin{array}{lcl}
\mathit{FVM} & \longrightarrow & \mathit{Func}\ \mathit{FVM} \\
\mathit{Func} & \longrightarrow & \mathit{Tipo}\ \textbf{id}\ \textbf{pari}\ \mathit{Arg}\ \textbf{pard}\ \mathit{Bloque} \\
\mathit{Arg} & \longrightarrow & \epsilon \\
\mathit{Arg} & \longrightarrow & \mathit{CArg} \\
\mathit{CArg} & \longrightarrow & \mathit{Tipo}\ \textbf{id}\ \mathit{CArgp} \\
\mathit{CArgp} & \longrightarrow & \textbf{coma}\ \mathit{Tipo}\ \textbf{id}\ \mathit{CArgp} \\
\mathit{CArgp} & \longrightarrow & \epsilon \\
 & & \\
\mathit{Instr} & \longrightarrow & \textbf{return}\ \mathit{Expr}\ \textbf{pyc} \\
\mathit{Factor} & \longrightarrow & \textbf{id}\ \textbf{pari}\ \mathit{Par}\ \textbf{pard} \\
\mathit{Par} & \longrightarrow & \epsilon \\
\mathit{Par} & \longrightarrow & \mathit{Expr}\ \mathit{CPar} \\
\mathit{CPar} & \longrightarrow & \epsilon \\
\mathit{CPar} & \longrightarrow & \textbf{coma}\ \mathit{Expr}\ \mathit{CPar}
\end{array}
$$

Además, habría que añadir las siguientes reglas semánticas:

1. No es posible usar el nombre de una función sin poner los paréntesis y tantos argumentos como necesite (ni más ni menos[4]), y tampoco se le pueden poner paréntesis (con o sin argumentos) a un identificador que no sea el nombre de una función. De esta manera, se consigue que el valor que devuelve el *Factor* siempre sea un valor de tipo simple.

2. Los parámetros se pasan por valor, y aunque dentro de una función se modifique el valor de un parámetro (operación que, por lo tanto, está permitida), dicha modificación no tendrá efecto fuera de esa función. Los argumentos son todos por tanto de entrada y no pueden ser de salida.

3. Aunque en C está permitido (y es recomendable) utilizar la instrucción `return` en la función `main` (que es una función como otra cualquiera), en este lenguaje la función `main` no se utiliza como función y por tanto el compilador debe producir un error si se utiliza el `return` en ficha función.

4. Si en una función (que no sea `main`) se alcanza el final del código de una función sin haberse ejecutado ninguna instrucción `return`, el compilador debe generar código para salir de la función sin poner ningún valor en la posición reservada al valor devuelto por la función; en este caso, el compilador no debe producir ningún aviso ni error al respecto (aunque un buen compilador lo haría).

5. Por supuesto, el compilador debe permitir las llamadas recursivas sin ningún tipo de limitación, aunque la especificación sintáctica solamente permite la recursividad directa (que una función se llame a sí misma).

6. Las variables declaradas al principio del "`main`" (las que aparezcan inmediatamente después de la primera llave abierta "`{`") deben ser consideradas como pertenecientes al ámbito de las variables globales (las declaradas fuera del "`main`"), aunque esta norma no es válida en C.[5]

7. Es perfectamente posible declarar una función como:

   ```
   float unaFun(int unaFun) { ... }
   ```

   La práctica no debe producir ningún tipo de error ni aviso, ya que ambos símbolos pertenecen a ámbitos diferentes. En cambio, no sería correcto un programa como:

   ```
   int unaVariable;

   float unaVariable() { ... }  /* ERROR !!! */
   ```

   Aunque el primer símbolo sea una variable y el segundo una función, se almacenan en el mismo ámbito, y por tanto no pueden tener el mismo nombre.

8. Tanto el tipo devuelto por la función como los tipos de los argumentos pueden ser enteros o reales. Esta característica implica almacenar el tipo de la función en la tabla de tipos (como se ha estudiado en las clases de teoría), pero además tiene otra implicación muy importante: puede ser necesario realizar conversiones de tipos en las llamadas a las funciones, como en el siguiente ejemplo:

   ```
   int a;
   float f(int ent) { ... }

   ...

   a = f(1003.4);
   ```

   En este caso, en la llamada habría que convertir el "`1003.4`" a tipo "`int`", con lo que quedaría "`1003`". Posteriormente, el valor devuelto por la función habría que convertirlo a entero (esto último se haría en la asignación, sin tener en cuenta que la expresión a la derecha del "`=`" es una llamada a una función).

9. Como ocurre con las conversiones en las llamadas a funciones, también puede ser necesario realizar conversiones en la instrucción "`return`", como en el siguiente ejemplo:

[4] El incumplimiento de esta norma en C no produciría un error (como debe producir la práctica), sino que simplemente daría un *warning* (depende del compilador).

[5] Realmente, en C, el `main` es una función más con su propio registro de activación, por eso las variables del `main` no son globales.

```
int a;
float f() {
   return 8;          /* el "8" debe convertirse a "8.0" */
}
int fe()
{
  return 7.999;       /* el "7.999" debe convertirse a "7" */
}
```