Procesamiento de Lenguajes (PL)
Curso 2023/2024

Práctica 5: traductor a código `m2r`

Fecha y método de entrega

La práctica debe realizarse de forma individual, y debe entregarse a través del servidor de prácticas del DLSI **antes de las 23:59 del 31 de mayo de 2024**. Los ficheros fuente ("`plp5.l`", "`plp5.y`", "`comun.h`", "`TablaTipos.h`", "`TablaTipos.cc`", "`TablaSimbolos.h`", "`TablaSimbolos.cc`", "`Makefile`" y aquellos que se añadan) se comprimirán en un fichero llamado "`plp5.tgz`", sin directorios.

Al servidor de prácticas del DLSI se puede acceder desde la web del DLSI (http://www.dlsi.ua.es), en el apartado "Entrega de prácticas". Una vez en el servidor, se debe elegir la asignatura PL y seguir las instrucciones.

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.

Especificación sintáctica del lenguaje fuente

La sintaxis del lenguaje fuente puede ser representada por la gramática siguiente (que no es necesario modificar):

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*
Instr	$\longrightarrow$	**for pari id asig** *Esimple* **pyc** *Expr* **pyc id incrdecr pard** *Instr*
Expr	$\longrightarrow$	*Expr* **oprel** *Esimple*
Expr	$\longrightarrow$	*Esimple*
Esimple	$\longrightarrow$	*Esimple* **opas** *Term*
Esimple	$\longrightarrow$	*Term*
Term	$\longrightarrow$	*Term* **opmd** *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:

Expresión regular	Componente léxico	Valor léxico entregado
`[ \n\t]+`	(ninguno)	
`main`	**main**	(palabra reservada)
`int`	**int**	(palabra reservada)
`float`	**float**	(palabra reservada)
`printf`	**printf**	(palabra reservada)
`scanf`	**scanf**	(palabra reservada)
`if`	**if**	(palabra reservada)
`else`	**else**	(palabra reservada)
`while`	**while**	(palabra reservada)
`for`	**for**	(palabra reservada)
`[A-Za-z][0-9A-Za-z]*`	**id**	(nombre del ident.)
`[0-9]+`	**nentero**	(valor numérico)
`([0-9]+)"."([0-9]+)`	**nreal**	(valor numérico)
`,`	**coma**	
`;`	**pyc**	
`(`	**pari**	
`)`	**pard**	
`==`	**oprel**	`==`
`!=`	**oprel**	`!=`
`<`	**oprel**	`<`
`<=`	**oprel**	`<=`
`>`	**oprel**	`>`
`>=`	**oprel**	`>=`
`+`	**opas**	`+`
`-`	**opas**	`-`
`*`	**opmd**	`*`
`/`	**opmd**	`/`
`=`	**asig**	
`&`	**referencia**	
`[`	**cori**	
`]`	**cord**	
`{`	**llavei**	
`}`	**llaved**	
`"\"%d\""`	**formato**	`"%d"`
`"\"%g\""`	**formato**	`"%g"`
`++`	**incrdecr**	`++`
`--`	**incrdecr**	`--`

El analizador léxico también debe ignorar los blancos[1] 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. En el Moodle de la asignatura se publicará un fichero con la declaración de unas constantes y una función para emitir mensajes de error.

Las reglas se pueden resumir como:

1. No es posible declarar dos veces un símbolo en el mismo ámbito. Tampoco es posible utilizar un identificador sin haberlo declarado previamente.

2. 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. El tamaño máximo de memoria para variables es de 16000 posiciones, desde la 0 a la 15999; el resto, desde la 16000 a la 16383 debe usarse para temporales. Si al declarar una variable el espacio ocupado por ésta sobrepasa el tamaño máximo, el compilador debe producir un mensaje de error indicando el lexema de la variable que ya no cabe en memoria; de igual forma, si se agota el espacio para temporales también se debe producir un mensaje de error (en este caso, da igual el token al que se haga referencia en el mensaje).

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

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).[2]

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. La instrucción **for** es similar a la de C, pero hay que hacer las siguientes comprobaciones:

   - Que el primer identificador se haya declarado
   - Que sea de tipo entero; si la expresión después del "`=`" es real, debe convertirse a entero antes de asignársela a la variable
   - Que el segundo identificador se haya declarado
   - Que el segundo identificador sea de tipo entero

   Por tanto, el primer y el segundo identificador podrían no ser iguales (no hay que comprobarlo), aunque no es lo habitual.

7. 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 y reutilizar las posiciones que ocupaban en siguientes bloques. 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.

8. 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).

9. En la instrucción **printf** se debe generar las conversiones de tipo necesarias (si el tipo de la expresión no coincide con el formato), 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 si el tipo de la referencia no coincide con el formato.[3]

10. 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. No se permite poner índices (corchetes) a variables que no sean de tipo *array*. Cuando en una referencia faltan índices, el token que debe indicarse en el error es el último "`]`", o bien el identificador si no hay corchetes; cuando sobran índices, el token debe ser el primer "`[`" que sobra.

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). En el error debe indicarse como token el "`]`" inmediatamente posterior al índice.

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.

[2] Las instrucciones de `m2r` para los operadores relacionales generan un número entero que vale 0 o 1.
[3] En C no se hacen estas conversiones, pero en esta práctica debes hacerlas.