Procesamiento de Lenguajes (PL)
Curso 2022/2023

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 miércoles 31 de mayo de 2023**. 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`.
- 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 quizá necesite alguna pequeña modificación en caso de conflictos):

S	$\longrightarrow$	**algoritmo dospto id** *SDec SInstr* **falgoritmo**
SDec	$\longrightarrow$	*Dec*
SDec	$\longrightarrow$	ϵ
Dec	$\longrightarrow$	**var** *DVar MDVar* **fvar**
DVar	$\longrightarrow$	*Tipo* **dospto id** *LId* **pyc**
MDVar	$\longrightarrow$	*DVar MDVar*
MDVar	$\longrightarrow$	ϵ
LId	$\longrightarrow$	**coma id** *LId*
LId	$\longrightarrow$	ϵ
Tipo	$\longrightarrow$	**entero**
Tipo	$\longrightarrow$	**real**
Tipo	$\longrightarrow$	**logico**
Tipo	$\longrightarrow$	**tabla nentero de** *Tipo*
SInstr	$\longrightarrow$	*SInstr* **pyc** *Instr*
SInstr	$\longrightarrow$	*Instr*
Instr	$\longrightarrow$	**escribe** *Expr*
Instr	$\longrightarrow$	**lee** *Ref*
Instr	$\longrightarrow$	**si** *Expr* **entonces** *Instr*
Instr	$\longrightarrow$	**si** *Expr* **entonces** *Instr* **sino** *Instr*
Instr	$\longrightarrow$	**mientras** *Expr* **hacer** *Instr*
Instr	$\longrightarrow$	**repetir** *Instr* **hasta** *Expr*
Instr	$\longrightarrow$	*Ref* **opasig** *Expr*
Instr	$\longrightarrow$	**blq** *SDec SInstr* **fblq**
Expr	$\longrightarrow$	*Expr* **obool** *Econj*
Expr	$\longrightarrow$	*Econj*
Econj	$\longrightarrow$	*Econj* **ybool** *Ecomp*
Econj	$\longrightarrow$	*Ecomp*
Ecomp	$\longrightarrow$	*Esimple* **oprel** *Esimple*
Ecomp	$\longrightarrow$	*Esimple*
Esimple	$\longrightarrow$	*Esimple* **opas** *Term*
Esimple	$\longrightarrow$	*Term*
Esimple	$\longrightarrow$	**opas** *Term*
Term	$\longrightarrow$	*Term* **opmd** *Factor*
Term	$\longrightarrow$	*Factor*
Factor	$\longrightarrow$	*Ref*
Factor	$\longrightarrow$	**nentero**
Factor	$\longrightarrow$	**nreal**
Factor	$\longrightarrow$	**pari** *Expr* **pard**
Factor	$\longrightarrow$	**nobool** *Factor*
Factor	$\longrightarrow$	**cierto**
Factor	$\longrightarrow$	**falso**
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)	
`algoritmo`	**algoritmo**	(palabra reservada)
`falgoritmo`	**falgoritmo**	(palabra reservada)
`var`	**var**	(palabra reservada)
`fvar`	**fvar**	(palabra reservada)
`entero`	**entero**	(palabra reservada)
`real`	**real**	(palabra reservada)
`logico`	**logico**	(palabra reservada)
`tabla`	**tabla**	(palabra reservada)
`de`	**de**	(palabra reservada)
`escribe`	**escribe**	(palabra reservada)
`lee`	**lee**	(palabra reservada)
`si`	**si**	(palabra reservada)
`entonces`	**entonces**	(palabra reservada)
`sino`	**sino**	(palabra reservada)
`mientras`	**mientras**	(palabra reservada)
`hacer`	**hacer**	(palabra reservada)
`repetir`	**repetir**	(palabra reservada)
`hasta`	**hasta**	(palabra reservada)
`blq`	**blq**	(palabra reservada)
`fblq`	**fblq**	(palabra reservada)
`cierto`	**cierto**	(palabra reservada)
`falso`	**falso**	(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**	
`:`	**dospto**	
`(`	**pari**	
`)`	**pard**	
`=`	**oprel**	`=`
`<>`	**oprel**	`<>`
`<`	**oprel**	`<`
`<=`	**oprel**	`<=`
`>`	**oprel**	`>`
`>=`	**oprel**	`>=`
`+`	**opas**	`+`
`-`	**opas**	`-`
`*`	**opmd**	`*`
`/`	**opmd**	`/`
`:=`	**opasig**	
`[`	**cori**	
`]`	**cord**	
`&&`	**ybool**	
`||`	**obool**	
`!`	**nobool**	

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 Pascal, 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 unas funciones para emitir mensajes de error. La siguiente tabla

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

indica qué error se corresponde con cada constante:

`ERR_YADECL`	simbolo '...' ya declarado
`ERR_NODECL`	identificador '...' no declarado
`ERR_DIM`	la dimension debe ser mayor que cero
`ERR_FALTAN`	faltan indices
`ERR_SOBRAN`	sobran indices
`ERR_INDICE_ENTERO`	la expresion entre corchetes debe ser de tipo entero
`ERR_EXP_LOG`	la expresion debe ser de tipo logico
`ERR_EXDER_LOG`	la expresion a la derecha de '...' debe ser de tipo logico
`ERR_EXDER_ENT`	la expresion a la derecha de '...' debe ser de tipo entero
`ERR_EXDER_RE`	la expresion a la derecha de '...' debe ser de tipo real o entero
`ERR_EXIZQ_LOG`	la expresion a la izquierda de '...' debe ser de tipo logico
`ERR_EXIZQ_RE`	la expresion a la izquierda de '...' debe ser de tipo real o entero
`ERR_NOCABE`	la variable '...' ya no cabe en memoria
`ERR_MAXTEMP`	no hay espacio para variables temporales

En cada error semántico debe indicarse la fila y columna (y a veces el lexema) de un token, según se explica más adelante. Las reglas semánticas se pueden resumir como:

1. No es posible declarar dos veces un símbolo en el mismo ámbito, aunque sea con el mismo tipo (error `ERR_YADECL`, indicando el identificador declarado por segunda vez). Por simplificar, el nombre del algoritmo debe ignorarse, no debe guardarse en ninguna tabla de símbolos.

2. No es posible utilizar un símbolo sin haberlo declarado previamente (error `ERR_NODECL`, indicando el identificador no declarado).

3. Si al declarar una variable el espacio ocupado por ésta sobrepasa el tamaño máximo de memoria para variables, el compilador debe producir un mensaje de error indicando el lexema exacto de la variable que ya no cabe en memoria (error `ERR_NOCABE`). El espacio máximo para variables debe ser de 16000 posiciones,[2] de la 0 a la 15999, dejando las últimas 384 (de la 16000 a la 16383) para temporales. 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.

4. Cuando en una expresión aritmética aparece un valor real (una variable o un número real), el tipo de la expresión es real. Si solamente aparecen valores enteros, entonces el tipo será entero. Si se combina una subexpresión real con una subexpresión entera utilizando cualquier operador válido (op. relacionales o aritméticos), debe generarse código adicional para convertir el valor que calcula la subexpresión entera en un valor real antes de operarlo con la subexpresión real.

5. Los operadores relacionales admiten expresiones reales o enteras, y el resultado es de tipo lógico. Los operadores aritméticos (suma, resta, producto y división) solamente pueden utilizarse con valores reales o enteros, nunca con valores lógicos, y el resultado será entero si los dos operandos son enteros, y será real en otro caso. Los operadores lógicos ('o' lógico, 'y' lógico y 'no' lógico) solamente pueden utilizarse con valores lógicos, nunca con valores reales o enteros, y el resultado es de tipo lógico. El incumplimiento de estas condiciones debe generar los errores `ERR_EXDER_LOG`, `ERR_EXDER_RE`, `ERR_EXIZQ_LOG` y `ERR_EXIZQ_RE`, indicando en todos los casos el operador implicado.

6. La expresión que aparece en las instrucciones **si**, **mientras** y **repetir-hasta** debe ser de tipo lógico (error `ERR_EXP_LOG`, indicando el token **si**, **mientras** o **hasta**)

7. No está permitido asignar un valor real a una variable entera (error `ERR_EXDER_ENT`), pero al revés (un valor entero a una variable real) sí está permitido y se debe generar código para convertir el valor entero a real. Por otro lado, no se permite asignar valores lógicos a variables enteras o reales (errores `ERR_EXDER_ENT` o `ERR_EXDER_RE`, según el tipo de la variable), ni valores reales o enteros a variables lógicas (`ERR_EXDER_LOG`). En todos los errores se indicará el operador de asignación.

8. Cuando se tenga que escribir por pantalla un valor lógico, se escribirá una '`c`' minúscula si el valor es cierto, o una '`f`' minúscula si es falso. Al leer del teclado una variable lógica, si se lee el carácter '`c`' se entenderá que el valor es cierto, y si se lee cualquier otro carácter se entenderá que es falso.

[2] En las primeras posiciones se guardarán las variables globales, y a continuación las definidas en la función `main`

9. La división entre valores enteros siempre es una división entera; si alguno de los operandos es real, la división es real.
10. No se permite utilizar una variable de tipo **tabla** con más (error `ERR_SOBRAN`) ni con menos (error `ERR_FALTAN`) índices de los necesarios (según la declaración de la variable) para obtener un valor de tipo simple. Cuando en una referencia faltan índices, el token que debe indicarse en el error es el último "`]`", o bien el "`id`" si no hay corchetes; cuando sobran índices, el token debe ser el primer "`[`" que sobra.
11. No se permite poner índices (corchetes) a variables que no sean de tipo **tabla** (error `ERR_SOBRAN`, indicando el identificador que no es una **tabla**)
12. Como consecuencia de las restricciones anteriores, no está permitida la asignación entre valores de tipo **tabla**, las asignaciones deben realizarse siempre con valores de tipo simple (no hay que hacer ninguna comprobación adicional).
13. En las declaraciones de variables de tipo **tabla**, el número debe ser estríctamente mayor que cero (error `ERR_DIM` indicando el número). El rango de posiciones del *array* será, como en C, de 0 a $n-1$.
14. La expresión entre corchetes (el índice) debe ser de tipo entero, en otro caso se emitirá el error `ERR_INDICE_ENTERO`, indicando como token el "`[`" inmediatamente anterior al índice.
15. En principio, el número de dimensiones que puede tener una variable de tipo **tabla** no está limitado más que por la memoria disponible. Por este motivo es aconsejable utilizar una tabla de tipos.

Implementación

- Como en prácticas anteriores, en el Moodle de la asignatura se publicará un autocorrector. Además, se publicará una descripción completa del lenguaje `m2r` y el código fuente de un intérprete para ese lenguaje (que también irá incluido en el autocorrector).
- En el Moodle de la asignatura se publicará el código fuente de una clase para manejar la tabla de símbolos (similar a la de la práctica 4, pero no igual porque la información de los símbolos es diferente), y también se publicará el código de una clase para manejar la tabla de tipos, un `Makefile` y algunas constantes y funciones para emitir mensajes de error.
- De las 16384 posiciones de memoria de la máquina virtual se deben reservar 16000 para variables (0-15999) y las últimas 384 (16000-16383) para variables temporales. Para crear variables temporales es suficiente con una variable global y una función:

  ```
  int ctemp = 16000;   // contador de direcciones temporales

  int nuevaTemp(void)
  {
    // devolver el valor de ctemp, incrementando antes su valor

    // pero antes de retornar, comprobar que no se ha sobrepasado el valor
    // máximo, 16383
  }
  ```

  Además, para evitar agotar todo el espacio de temporales es necesario reutilizarlas una vez se ha generado el código. Para ello hay que guardar en un marcador el valor de `ctemp` antes de la instrucción, y dejarlo como estaba después:

  ```
  ... :   { $$.nlin = ctemp; }  Instr    { .... ; ctemp=$1.nlin; }
  ```

  Aunque hay varias reglas con *Instr*, es suficiente con hacerlo en las reglas de *SInstr*.[3]
- Es recomendable que el atributo para guardar el código generado sea de tipo `string`, aunque para construir nuevo código puede ser más conveniente usar `stringstream` de forma temporal.

[3] En condiciones normales no se van a agotar las temporales, pero es posible agotarlas en una única expresión, p.ej. `1+2+3+...+384`, en cuyo caso hay que emitir el error `ERR_MAXTEMP`.