Procesamiento de Lenguajes (PL)
Curso 2019/2020

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 2020**. Los ficheros fuente ("`plp5.l`", "`plp5.y`", "`comun.h`", "`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 proporcionará un `Makefile`.
- 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, y al final, en la regla correspondiente al símbolo inicial, se imprimirá por la salida estándar; 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

La sintaxis del lenguaje fuente puede ser representada por la siguiente gramática:

S	$\longrightarrow$	**prg id dosp** *BlVar Bloque*
Bloque	$\longrightarrow$	**lbra** *SeqInstr* **rbra**
BlVar	$\longrightarrow$	**var** *Decl* **pyc**
Decl	$\longrightarrow$	*Decl* **pyc** *DVar*
Decl	$\longrightarrow$	*DVar*
DVar	$\longrightarrow$	*Tipo* **dosp** *LIdent*
TipoSimple	$\longrightarrow$	**int**
TipoSimple	$\longrightarrow$	**real**
TipoSimple	$\longrightarrow$	**char**
Tipo	$\longrightarrow$	*TipoSimple*
Tipo	$\longrightarrow$	**cori** *Rango Dims*
Dims	$\longrightarrow$	**coma** *Rango Dims*
Dims	$\longrightarrow$	**cord** *TipoSimple*
Rango	$\longrightarrow$	**numentero ptopto numentero**
LIdent	$\longrightarrow$	*LIdent* **coma id**
LIdent	$\longrightarrow$	**id**
SeqInstr	$\longrightarrow$	*SeqInstr* **pyc** *Instr*
SeqInstr	$\longrightarrow$	*Instr*
Instr	$\longrightarrow$	*Bloque*
Instr	$\longrightarrow$	*Ref* **asig** *Expr*
Instr	$\longrightarrow$	**prn** *Expr*
Instr	$\longrightarrow$	**read** *Ref*
Instr	$\longrightarrow$	**if** *Expr* **dosp** *Instr*
Instr	$\longrightarrow$	**if** *Expr* **dosp** *Instr* **else** *Instr*
Instr	$\longrightarrow$	**while** *Expr* **dosp** *Instr*
Expr	$\longrightarrow$	*Esimple* **oprel** *Esimple*
Expr	$\longrightarrow$	*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	$\longrightarrow$	**numentero**
Factor	$\longrightarrow$	**numreal**
Factor	$\longrightarrow$	**ctechar**
Factor	$\longrightarrow$	**pari** *Expr* **pard**
Factor	$\longrightarrow$	**toChr pari** *Esimple* **pard**
Factor	$\longrightarrow$	**toInt pari** *Esimple* **pard**
Ref	$\longrightarrow$	**id**
Ref	$\longrightarrow$	**id cori** *LisExpr* **cord**
LisExpr	$\longrightarrow$	*LisExpr* **coma** *Expr*
LisExpr	$\longrightarrow$	*Expr*

Nota: esta gramática es ambigua (por las reglas del **if-else**), produce un conflicto desplazamiento/reducción en las tablas que genera `yacc`/`bison` que se resuelve desplazando, con lo que se asocia el **else** con el **if** anterior más cercano. Esta forma de resolver los conflictos refleja la semántica de la instrucción correctamente, por lo que dicho conflicto debe ignorarse (aunque no deben ignorarse otros conflictos, si aparecen seguramente hay algún error).

Especificación léxica

Expresión regular	Componente léxico	Valor léxico entregado
`[ \n\t]+`	(ninguno)	
`prg`	**prg**	(palabra reservada)
`var`	**var**	(palabra reservada)
`int`	**int**	(palabra reservada)
`real`	**real**	(palabra reservada)
`char`	**char**	(palabra reservada)
`println`	**prn**	(palabra reservada)
`print`	**prn**	(palabra reservada)
`read`	**read**	(palabra reservada)
`if`	**if**	(palabra reservada)
`else`	**else**	(palabra reservada)
`while`	**while**	(palabra reservada)
`toChr`	**toChr**	(palabra reservada)
`toInt`	**toInt**	(palabra reservada)
`[A-Za-z][0-9A-Za-z]*`	**id**	(nombre del ident.)
`[0-9]+`	**numentero**	(valor numérico)
`([0-9]+)"."([0-9]+)`	**numreal**	(valor numérico)
Cualquier carácter imprimible entre dos “’”	**ctechar**	(código ASCII)
`,`	**coma**	
`;`	**pyc**	
`:`	**dosp**	
`(`	**pari**	
`)`	**pard**	
`{`	**lbra**	
`}`	**rbra**	
`==`	**oprel**	`==`
`!=`	**oprel**	`!=`
`<`	**oprel**	`<`
`<=`	**oprel**	`<=`
`>`	**oprel**	`>`
`>=`	**oprel**	`>=`
`+`	**opas**	`+`
`-`	**opas**	`-`
`*`	**opmd**	`*`
`/`	**opmd**	`/`
`=`	**asig**	
`[`	**cori**	
`]`	**cord**	
`..`	**ptopto**	

Notas:

1. Las constantes de tipo char estarán formadas por una comilla simple “’”, un carácter cuyo código ASCII sea superior al 32 (el espacio en blanco) y otra comilla simple. Además, si el carácter es una barra invertida “\”, se deben permitir las siguientes secuencias especiales: “`'\n'`”, “`'\0'`” y “`'\\'`”. En estos casos, los códigos ASCCI serán 10, 0 y 92, respectivamente. Si aparece cualquier otro carácter después de la barra invertida se producirá un error léxico.

   Las expresiones regulares en `lex` serían:

   ```
   "'"[\040-\377]"'"  {...}
   "'\\n'"            {...}
   "'\\0'"            {...}
   "'\\\\'"           {...}
   ```

2. El analizador léxico debe ignorar los comentarios, que en esta práctica empiezan con la secuencia de caracteres “`(*`” y terminan con la secuencia “`*)`”. Un comentario puede ocupar varias líneas y no se permiten los comentarios anidados.
3. El terminal **ptopto** esta formado por dos puntos que deben ir seguidos, sin ningún carácter entre ellos.

Especificación semántica

Las reglas semánticas que debe cumplir este lenguaje son las siguientes:

Declaración de variables

- No es posible declarar dos veces un símbolo, aunque sea con el mismo tipo. El identificador que aparece después de la palabra reservada "`prg`" no se debe almacenar en la tabla de símbolos y por tanto se puede usar ese identificador para una variable. Como se puede observar, no hay ámbitos anidados, solamente hay un ámbito en todo el programa fuente.
- No es posible utilizar un símbolo sin haberlo declarado previamente.
- 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. El espacio máximo para variables debe ser de 16000 posiciones, de la 0 a la 15999, dejando las últimas 384 (de la 16000 a la 16383) para temporales.

Instrucciones

Asignación: en esta instrucción tanto la referencia que aparece a la izquierda del operador "`=`" como la expresión de la derecha deben ser del mismo tipo. Solamente hay una excepción a esta regla, y consiste en que está permitida también la asignación cuando la referencia es real y la expresión es entera. En caso contrario debe emitirse un error semántico.

Lectura y escritura: en la sentencia de lectura, "`read`", se generará la instrucción en `m2r` apropiada para el tipo de la referencia.[1] Las dos sentencias de escritura, "`print`" y "`println`", se diferencian únicamente en que la segunda escribe un carácter de nueva línea (un "`\n`" de C) después de haber escrito el valor de la expresión.

Control de flujo: las instrucciones "`if`" y "`while`" tienen una semántica similar a la de C, con la única excepción de que se exige que el tipo de la expresión sea entero; en caso contrario, se debe producir un error semántico.

Expresiones

Operadores relacionales: son equivalentes a los de C/C++; estos operadores permiten comparar valores numéricos (enteros o reales) entre sí, y también valores de tipo caracter entre sí (en cuyo caso se utiliza el código ASCII correspondiente). No se permite comparar valores de tipo caracter con valores enteros o reales. El resultado de la operación siempre será un valor entero que valdrá 0 o 1.

Operadores aritméticos: solamente pueden utilizarse con valores enteros o reales y son similares a los de C/C++.

Cambio de signo: La regla

$$\textit{Esimple} \longrightarrow \textbf{opas}\ \textit{Term}$$

permite cambiar el signo del término (si **opas** es "`-`"), como por ejemplo en `-3*4+2` .[2]

Funciones predefinidas

En esta práctica se deben implementar las siguientes funciones predefinidas:

toChr: Esta función admite un argumento de tipo entero y devuelve el carácter cuyo código ASCII está representado por los 8 bits más bajos del argumento.[3] Si el argumento no es un entero se emitirá un error.

toInt: Esta función admite enteros, caracteres o reales. Si el argumento es entero, simplemente devuelve el mismo valor; si el argumento es carácter, devuelve el código ASCII del carácter; finalmente, si el argumento es real, devuelve un entero que es la parte entera del argumento.

[1] Si se intenta leer un valor entero y el usuario introduce un valor real, el comportamiento del programa depende del intérprete de `m2r`, no es problema del diseñador del compilador.

[2] En este ejemplo se cambiaría el signo de `12`, no el de `3`.

[3] Para obtener los 8 bits más bajos se puede usar "`modi #256`"

Declaración de *arrays*

- En la declaración de un *array*, el segundo número que aparece en el rango debe ser mayor o igual que el primero. Debe tenerse en cuenta que ambos valores son válidos como índices (al contrario que en C, que no incluye el valor que aparece entre corchetes en el rango de valores posibles de los índices del *array*).
- Al declarar una variable de tipo *array* es posible que se agote la memoria disponible en la máquina objeto, por lo que debe producirse un error semántico. En ningún caso está permitido que se reserve más memoria que la que sea imprescindible para un *array*.

Acceso a componentes de *arrays*

- En este lenguaje no es posible hacer referencia a una componente de un *array* sin poner tantos índices como sean necesarios según la declaración de la variable. Tanto si faltan como si sobran índices se debe producir un error semántico (lo antes posible, especialmente cuando sobran índices). De esta manera, el no terminal *Ref* debe devolver un tipo básico (entero, real o caracter) al resto del compilador.
- No está permitido poner corchetes (índices) a una variable que no sea de tipo *array*.
- El índice de un *array* debe ser de tipo entero; en caso contrario, se debe producir un error.
- Se debe recordar que es posible utilizar referencias a *arrays* en la parte izquierda de una asignación, en una sentencia de lectura y en una expresión. En los dos primeros casos el código que se debe generar es similar, y es ligeramente distinto del tercer caso.

Implementación

- Como en prácticas anteriores, en la web 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 la web 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.
- También se publicará en la web un `Makefile` y algunas 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 *SeqInstr*.[4]

[4] En condiciones normales no se van a agotar las temporales, pero es posible agotarlas en una única expresión, p.ej. `1+2+3+...+384`.

- Es recomendable que el atributo para guardar el código generado sea de tipo `string`, aunque para construir nuevo código puede ser más conveniente usar `sprintf` con vectores de caracteres, siempre que no se use código de otras instrucciones (el código generado en otras instrucciones puede ser muy largo y sobrepasar el vector de caracteres).[5]

Evaluación de la práctica

Para obtener la máxima calificación (10) es necesario realizar la práctica tal y como se especifica más arriba. Sin embargo, será posible realizar estas versiones simplificadas de la práctica, aunque la calificación máxima será menor:

1. Versión sin el tipo `char` (calificación máxima: 8): esta versión consiste en realizar la práctica sin el tipo `char`, sólo con enteros y reales. Para ello habría que:
   - Eliminar de la gramática las siguientes reglas:

     TipoSimple ⟶ **char**
     Factor ⟶ **ctechar**
     Factor ⟶ **toChr pari** *Esimple* **pard**

   - Eliminar los *tokens* **char**, **ctechar** y **toChr** del analizador léxico.
   - Adaptar las restricciones semánticas teniendo en cuenta que no va a haber valores de tipo `char` en el programa fuente. No es necesario modificar el código ni las constantes de las clases para manejar la tabla de tipos y la tabla de símbolos, aunque permitan manejar el tipo `char` no se va a utilizar y no supone ningún problema.
2. Versión sin `char` ni *arrays* (calificación máxima: 5): esta versión consiste en eliminar el tipo `char` y además eliminar los *arrays*, para lo que habría que hacer todo lo comentado en la versión anterior y además lo siguiente:
   - Eliminar las siguientes reglas:

     Tipo ⟶ **cori** *Rango Dims*
     Dims ⟶ **coma** *Rango Dims*
     Dims ⟶ **cord** *TipoSimple*
     Rango ⟶ **numentero ptopto numentero**
     Ref ⟶ **id cori** *LisExpr* **cord**
     LisExpr ⟶ *LisExpr* **coma** *Expr*
     LisExpr ⟶ *Expr*

   - Eliminar los *tokens* **cori**, **cord** y **ptopto** del analizador léxico.
   - Eliminar la tabla de tipos y las restricciones semánticas relativas a los *arrays*.

Por supuesto, es posible empezar la práctica haciendo esta última versión, y luego añadir los *arrays*, y posteriormente el tipo `char`, aunque lo recomendable es hacer directamente la práctica completa.[6]

[5] Otra opción puede ser convertir todas las direcciones de memoria, que son enteros, a `string` justo antes de construir el nuevo código, con lo que se podría usar el operador "`+`" y no usar `sprintf`.

[6] No, no va a haber una versión intermedia sin *arrays* pero con el tipo `char`.