

Procesamiento de Lenguajes (PL)

Curso 2018/2019

Práctica 5: traductor a código m2r

Fecha y método de entrega

La práctica debe realizarse de forma individual o por parejas¹, y debe entregarse a través del servidor de prácticas del DLSI **antes de las 23:59 del viernes 31 de mayo de 2019**.

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”.

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 (en el formulario de entrega se pide el DNI de la otra persona, si la hay), y se debe poner en un comentario los nombres y DNIs de los dos componentes de la pareja. Si no aparece dicho comentario se entenderá que la práctica se entrega de forma individual.

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 **plp5.1** y **plp5.y**. Para compilar la práctica se utilizarán las siguientes órdenes:

```
$ flex plp5.1
$ 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.

¹Una 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):

<i>S</i>	→	class <i>id</i> llavei <i>attributes</i> dosp <i>BDecl</i> methods dosp <i>Metodos</i> llaved
<i>Metodos</i>	→	int main pari pard <i>Bloque</i>
<i>Tipo</i>	→	int
<i>Tipo</i>	→	float
<i>Bloque</i>	→	llavei <i>BDecl</i> <i>SeqInstr</i> llaved
<i>BDecl</i>	→	<i>BDecl</i> <i>DVar</i>
<i>BDecl</i>	→	ϵ
<i>DVar</i>	→	<i>Tipo</i> <i>LIdent</i> pyc
<i>LIdent</i>	→	<i>LIdent</i> coma <i>Variable</i>
<i>LIdent</i>	→	<i>Variable</i>
<i>Variable</i>	→	id <i>V</i>
<i>V</i>	→	ϵ
<i>V</i>	→	cori nentero cord <i>V</i>
<i>SeqInstr</i>	→	<i>SeqInstr</i> <i>Instr</i>
<i>SeqInstr</i>	→	ϵ
<i>Instr</i>	→	pyc
<i>Instr</i>	→	<i>Bloque</i>
<i>Instr</i>	→	<i>Ref</i> asig <i>Expr</i> pyc
<i>Instr</i>	→	print pari <i>Expr</i> pard pyc
<i>Instr</i>	→	scan pari <i>Ref</i> pard pyc
<i>Instr</i>	→	if pari <i>Expr</i> pard <i>Instr</i>
<i>Instr</i>	→	if pari <i>Expr</i> pard <i>Instr</i> else <i>Instr</i>
<i>Instr</i>	→	while pari <i>Expr</i> pard <i>Instr</i>
<i>Expr</i>	→	<i>Expr</i> relop <i>Esimple</i>
<i>Expr</i>	→	<i>Esimple</i>
<i>Esimple</i>	→	<i>Esimple</i> addop <i>Term</i>
<i>Esimple</i>	→	<i>Term</i>
<i>Term</i>	→	<i>Term</i> mulop <i>Factor</i>
<i>Term</i>	→	<i>Factor</i>
<i>Factor</i>	→	<i>Ref</i>
<i>Factor</i>	→	nentero
<i>Factor</i>	→	nreal
<i>Factor</i>	→	pari <i>Expr</i> pard
<i>Ref</i>	→	this punto id
<i>Ref</i>	→	id
<i>Ref</i>	→	<i>Ref</i> cori <i>Esimple</i> cord

Especificación léxica del lenguaje fuente

EXPRESIÓN REGULAR	COMPONENTE LÉXICO	VALOR LÉXICO ENTREGADO
[\n\t]+	(ninguno)	
class	class	(palabra reservada)
attributes	attributes	(palabra reservada)
methods	methods	(palabra reservada)
int	int	(palabra reservada)
float	float	(palabra reservada)
main	main	(palabra reservada)
print	print	(palabra reservada)
scan	scan	(palabra reservada)
if	if	(palabra reservada)
else	else	(palabra reservada)
while	while	(palabra reservada)
this	this	(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)
:	dosp	
,	coma	
;	pyc	
.	punto	
(pari	
)	pard	
==	relop	==
!=	relop	!=
<	relop	<
<=	relop	<=
>	relop	>
>=	relop	>=
+	addop	+
-	addop	-
*	mulop	*
/	mulop	/
=	assig	
[cori	
]	cord	
{	llavei	
}	llaved	

Notas:

1. El analizador léxico también debe ignorar los blancos² y los comentarios, que comenzarán por ‘//’ y terminarán al final de la línea.

Especificación semántica del lenguaje fuente

Las reglas semánticas de este lenguaje son similares a las de C++/Java, 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 en el mismo ámbito. 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.

²Los tabuladores se contarán como un espacio en blanco.

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
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. Se puede acceder a los atributos del objeto en cualquier momento, como a cualquier variable local, pero como es posible declarar una variable con el mismo nombre que un atributo, se permite utilizar la referencia a “`this`” para acceder a los atributos cuando exista otro símbolo que *oculte* la visibilidad del atributo. Obviamente, no se permite usar “`this`” con símbolos que no sean atributos.
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 `print` se debe generar siempre una instrucción `wrl` después de escribir el valor de la expresión.
10. No se permite utilizar una variable/atributo 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.
11. No se permite poner índices (corchetes) a variables/atributos que no sean de tipo *array*.
12. No está permitida la asignación entre valores de tipo *array*. Las asignaciones deben realizarse siempre con valores de tipo simple. De igual forma, la referencia en la instrucción `scan` también debe ser de tipo simple.
13. 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 $n - 1$.
14. La expresión entre corchetes (el índice) debe ser de tipo entero siempre (aunque C permita otros tipos que se convierten automáticamente a entero).
15. 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 otros métodos al lenguaje anterior. En caso de optar por esta ampliación, es conveniente empezar directamente considerando que puede haber métodos, 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:

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

<i>Metodos</i>	\longrightarrow	<i>Met Metodos</i>
<i>Met</i>	\longrightarrow	<i>Tipo id pari Arg pard Bloque</i>
<i>Arg</i>	\longrightarrow	ϵ
<i>Arg</i>	\longrightarrow	<i>CArg</i>
<i>CArg</i>	\longrightarrow	<i>Tipo id CArgp</i>
<i>CArgp</i>	\longrightarrow	coma <i>Tipo id CArgp</i>
<i>CArgp</i>	\longrightarrow	ϵ
<i>Instr</i>	\longrightarrow	return <i>Expr pyc</i>
<i>Factor</i>	\longrightarrow	id pari Par pard
<i>Par</i>	\longrightarrow	ϵ
<i>Par</i>	\longrightarrow	<i>Expr CPar</i>
<i>CPar</i>	\longrightarrow	ϵ
<i>CPar</i>	\longrightarrow	coma <i>Expr CPar</i>

Además, habría que añadir las siguientes reglas semánticas:

1. No es posible usar el nombre de un método sin poner los paréntesis y tantos argumentos como necesite (ni más ni menos, y tampoco se le pueden poner paréntesis (con o sin argumentos) a un identificador que no sea el nombre de un método. 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 un método se modifique el valor de un parámetro (operación que, por lo tanto, está permitida), dicha modificación no tendrá efecto fuera de ese método. 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 un método (que no sea **main**) se alcanza el final del código sin haberse ejecutado ninguna instrucción **return**, el compilador debe generar código para salir del método sin poner ningún valor en la posición reservada al valor devuelto; 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 un método se llame a sí mismo).
6. El método **main** es un método más y debería tener su propio registro de activación, pero está permitido considerar que ese método no va a ser llamado de forma recursiva, por lo que se puede optar por asignar direcciones absolutas a sus variables, o bien direcciones relativas (las variables temporales deberían ser siempre relativas). Los atributos pueden considerarse como variables globales a todos los métodos y deben tener direcciones absolutas.
7. Es perfectamente posible declarar un método como:

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

La práctica no debe producir ningún tipo de error ni aviso, ya que ambos símbolos pertenecen a ámbitos diferentes.

8. Tanto el tipo devuelto por el método como los tipos de los argumentos pueden ser enteros o reales. Esta característica implica almacenar el tipo del método 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 los métodos, como en el siguiente ejemplo:

```
class ej1 {
  attributes:
    int a;
```

```

methods:
    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 el método 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 un método).

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

```

class ej1 {
    attributes:
        int a;

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

```