Procesamiento de Lenguajes (PL)
Curso 2024/2025

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 6 de junio de 2025**. Los ficheros fuente ("`plp5.l`", "`plp5.y`", "`comun.h`", "`TablaSimbolos.h`", "`TablaSimbolos.cc`", "`TablaTipos.h`", "`TablaTipos.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 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 proporcionarán constantes y funciones para 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. En el caso de los errores semánticos, en el código de las funciones se indica con comentarios a qué token debe referirse el error.

Especificación sintáctica

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

S	$\longrightarrow$	*Fun*
Fun	$\longrightarrow$	**fn id pari pard** *Cod* **endfn**
SType	$\longrightarrow$	**int**
SType	$\longrightarrow$	**real**
Type	$\longrightarrow$	*SType*
Type	$\longrightarrow$	**array** *SType Dim*
Dim	$\longrightarrow$	**numint coma** *Dim*
Dim	$\longrightarrow$	**numint**
Cod	$\longrightarrow$	*Cod* **pyc** *I*
Cod	$\longrightarrow$	*I*
I	$\longrightarrow$	*Blq*
I	$\longrightarrow$	**let** *Ref* **asig** *E*
I	$\longrightarrow$	**var id** *IT*
I	$\longrightarrow$	**print** *E*
I	$\longrightarrow$	**read** *Ref*
I	$\longrightarrow$	**while** *E I*
I	$\longrightarrow$	**loop id range** *Range I* **endloop**
I	$\longrightarrow$	**if** *E I Ip*
Range	$\longrightarrow$	**numint dosp numint**
Range	$\longrightarrow$	**numint**
Blq	$\longrightarrow$	**blq** *Cod* **fblq**
Ip	$\longrightarrow$	**else** *I* **fi**
Ip	$\longrightarrow$	**elif** *E I Ip*
Ip	$\longrightarrow$	**fi**
IT	$\longrightarrow$	**dosp** *Type*
IT	$\longrightarrow$	ϵ
E	$\longrightarrow$	*E* **opas** *T*
E	$\longrightarrow$	**opas** *T*
E	$\longrightarrow$	*T*
T	$\longrightarrow$	*T* **opmd** *F*
T	$\longrightarrow$	*F*
F	$\longrightarrow$	**numint**
F	$\longrightarrow$	**numreal**
F	$\longrightarrow$	**pari** *E* **pard**
F	$\longrightarrow$	*Ref*
Ref	$\longrightarrow$	**id**
Ref	$\longrightarrow$	**id cori** *LExpr* **cord**
LExpr	$\longrightarrow$	*LExpr* **coma** *E*
LExpr	$\longrightarrow$	*E*

Especificación léxica

Expresión regular	Componente léxico	Valor léxico entregado
`[ \n\t]+`	(ninguno)	
`fn`	**fn**	(palabra reservada)
`endfn`	**endfn**	(palabra reservada)
`int`	**int**	(palabra reservada)
`real`	**real**	(palabra reservada)
`array`	**array**	(palabra reservada)
`blq`	**blq**	(palabra reservada)
`fblq`	**fblq**	(palabra reservada)
`let`	**let**	(palabra reservada)
`var`	**var**	(palabra reservada)
`print`	**print**	(palabra reservada)
`read`	**read**	(palabra reservada)
`if`	**if**	(palabra reservada)
`else`	**else**	(palabra reservada)
`elif`	**else**	(palabra reservada)
`fi`	**fi**	(palabra reservada)
`while`	**while**	(palabra reservada)
`loop`	**loop**	(palabra reservada)
`range`	**range**	(palabra reservada)
`endloop`	**endloop**	(palabra reservada)
`[A-Za-z][0-9A-Za-z]*`	**id**	(nombre del ident.)
`[0-9]+`	**numint**	(valor numérico)
`([0-9]+)"."([0-9]+)`	**numreal**	(valor numérico)
`,`	**coma**	
`;`	**pyc**	
`:`	**dosp**	
`(`	**pari**	
`)`	**pard**	
`+`	**opas**	`+`
`-`	**opas**	`-`
`*`	**opmd**	`*`
`/`	**opmd**	`/`
`=`	**asig**	
`[`	**cori**	
`]`	**cord**	

Nota:

1. 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.
2. En el caso del token "**numint** ", además de devolver el lexema, la fila y la columna, es aconsejable que devuelva también un atributo con su valor (de tipo `int`, lógicamente).

Especificación semántica

Las reglas semánticas que debe cumplir este lenguaje se describen a continuación. Cuando el programa fuente no cumple una de estas reglas se debe emitir un error semántico; 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 indica qué error se corresponde con cada constante:

`ERR_YADECL`	variable '...' ya declarada
`ERR_NODECL`	variable '...' no declarada
`ERR_NOCABE`	la variable '...' ya no cabe en memoria
`ERR_IFWHILE`	la expresion del '...' debe ser de tipo entero
`ERR_LOOP`	la variable del '...' debe ser de tipo entero
`ERR_DIM`	la dimension debe ser mayor que 0
`ERR_FALTAN`	faltan indices
`ERR_SOBRAN`	sobran indices
`ERR_INDICE_ENTERO`	el indice de un array debe ser de tipo entero
`ERR_ASIG`	tipos incompatibles en la asignacion
`ERR_MAXTEMP`	no hay espacio para variables temporales

Declaración de variables

- No es posible declarar dos veces un símbolo, aunque sea con el mismo tipo (error `ERR_YADECL`). Por simplificar, el nombre de la función debe ignorarse, no debe guardarse en ninguna tabla de símbolos.
- No es posible utilizar un símbolo sin haberlo declarado previamente (error `ERR_NODECL`).
- 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, de la 0 a la 15999, dejando las últimas 384 (de la 16000 a la 16383) para temporales.
- Además del ámbito de la función, solamente los bloques tienen su propio ámbito, en el que se pueden declarar variables con el mismo nombre que en ámbitos anteriores. Por tanto, las instrucciones "`if`", "`else`", "`while`" y "`loop`" no tienen su propio ámbito, si necesitan declarar variables deben usar un bloque.

  Cuando termina el ámbito del bloque, el espacio ocupado por las variables declaradas en dicho ámbito debe reutilizarse en las siguientes instrucciones.
- Las variables se declaran con la instrucción "`var`", de dos formas:[1]
  - indicando el tipo después de "`:`".
  - sin indicar nada, en cuyo caso la variable se almacenará con el tipo entero

Instrucciones

Asignación: en la instrucción "`let`" tanto la referencia que aparece a la izquierda del operador "`=`" como la expresión de la derecha deben ser del mismo tipo (entero o real). 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 el error semántico `ERR_ASIG`.

Lectura y escritura: en la sentencia de lectura, "`read`", se generará la instrucción en `m2r` apropiada para el tipo de la referencia.[2] La sentencia de escritura "`print`" debe escribir 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 (si el valor de la expresión es distinto de 0 se considera cierto, y falso si es 0), con la única excepción de que se exige que el tipo de la expresión sea entero; en caso contrario, se debe producir el error semántico `ERR_IFWHILE`.

Bucle `loop`: en esta instrucción se debe haber declarado la variable como una variable entera simple (en caso contrario se debe producir el error `ERR_LOOP`) y se debe generar un bucle en el que, en cada paso, la variable debe ir tomando los valores entre el primer número y el segundo, y a continuación se ejecutará la instrucción del bucle. Si después de `range` solamente aparece un número se entenderá que es el segundo número del bucle y que el primero es 0, es decir, si aparece "`range 7`" es equivalente a "`range 0:7`".

Algunos ejemplos del bucle `loop`:

```
loop i range 3
  print i
endloop
```

```
0
1
2
3
```

```
loop i range 10:7
  print i
endloop
```

```
10
9
8
7
```

```
loop i range 15:15
  print i
endloop
```

```
15
```

[1] La instrucción "`let`" no permite declarar variables, solamente permite asignar valores a variables ya declaradas, o a posiciones de *arrays*.

[2] 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.

Expresiones

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

Cambio de signo: La regla

$$E \longrightarrow \textbf{opas } T$$

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

Declaración de *arrays*

- En la declaración de un *array*, los números que indican el tamaño de cada dimensión deben ser mayores que 0 (error `ERR_DIM`). El rango de valores posibles es el mismo que en C, desde 0 al tamaño menos 1 (p.ej. si el tamaño es 7, el rango es de 0 a 6).
- 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 (error `ERR_NOCABE`).

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, lo antes posible (especialmente cuando sobran índices), un error semántico (`ERR_FALTAN` o `ERR_SOBRAN`). De esta manera, el no terminal *Ref* debe devolver un tipo básico (entero o real) al resto del compilador.
- No está permitido poner corchetes (índices) a una variable que no sea de tipo *array* (error `ERR_SOBRAN`).
- El índice de un *array* debe ser de tipo entero; en caso contrario, se debe producir el error `ERR_INDICE_ENTERO`.
- 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 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
}
```

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

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 I, es suficiente con hacerlo en las reglas de Cod.[4]

- 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.

[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`, en cuyo caso hay que emitir el error `ERR_MAXTEMP`.