Procesadores de Lenguajes, Grado de Ingeniería Informática

Enunciado de la práctica obligatoria

Jaime Urquiza Fuentes y Sergio Hernández garcía

En este documento se especifica el enunciado de la práctica obligatoria de la asignatura de Procesadores de Lenguajes (Grado de Ingeniería Informática y Dobles Grados). También se proporcionan fechas de entrega de las diferentes fases de la práctica.

Práctica obligatoria
Procesadores de Lenguajes

Tabla de contenidos

Introducción

La práctica se podrá realizar en grupos de, como máximo, 3 personas. La puntuación obtenida no depende del número de integrantes del grupo, tampoco tiene por qué ser igual para todos los integrantes.

No se permite la integración de personas en un grupo después de la primera entrega.

Sí se permite la salida de personas de un grupo, dejando claro en una entrevista con el profesor, quién continúa con la práctica y quién no hace la práctica o decide empezar una nueva.

Material de entrega

Una **memoria escrita** en formato electrónico que incluya:

- Un informe del trabajo realizado, así como cualquier anotación o característica que se desee hacer notar, **sin incluir listados fuente**. Específicamente debe incluir:
    - Una breve descripción de reglas léxicas que se crean de especial relevancia
    - Demostración del procedimiento realizado para transformar la gramática a LL(1)
        - Se deben aportar los conjuntos de cabecera, siguientes y directores
    - Explicación de los errores contenidos en los 4 casos de prueba aportados (ver siguientes puntos)
    - Muy breve descripción de la implementación de la notificación de errores (si se opta al notable)
    - Muy breve descripción de la resolución de la recuperación de errores (si se opta al sobresaliente)
- 8 casos de prueba de los cuales, 4 han de ser correctos y 4 erróneos, de forma que permitan observar el comportamiento del procesador.
- Recordad: ***LO BUENO, SI BREVE, DOS VECES BUENO***

Aplicación informática que implemente la funcionalidad requerida para la entrega correspondiente (léxico, sintáctico o completa):

- Ejecutable de la aplicación, que debe funcionar sobre **plataforma Windows 10 disponible en la URJC**.
- Proyecto de desarrollo completo incluyendo listados fuente de las especificaciones e implementación.
- Los ficheros asociados a los casos de prueba que aparecen en la memoria.

La calidad del material entregado es responsabilidad de los estudiantes. En caso de encontrar una entrega defectuosa será considerada suspensa.

Calificación

La calificación de la práctica se divide en tres niveles:

- **aprobado** (hasta 5), completando la parte obligatoria.
- **notable** (hasta 7), alcanzando el grado de aprobado, completando las sentencias de control de flujo if, while y do-until de la parte opcional así como notificación de errores detallada (línea, columna y posible causa)
- **sobresaliente** (hasta 9,5), alcanzando el grado de notable y proporcionando recuperación de errores léxica y sintáctica así como completando toda la parte opcional.

Además se otorgará **medio punto extra** en función de la **calidad de la memoria final**.

Plazos de entrega

A continuación, se detallan los periodos de entrega. El día específico de límite de entrega se encuentra disponible en el apartado evaluación de Aula Virtual.

- Abril de 2024 Analizadores léxico y sintáctico. Evaluación ordinaria.
- Mayo de 2024 Práctica completa. Evaluación ordinaria.
- **Junio de 2024 Práctica completa**. Evaluación extraordinaria.

Especificación de la práctica

La práctica consiste en el **diseño e implementación de un traductor de programas** escritos en un lenguaje de programación **similar a C** (de ahora en adelante **lenguaje fuente**), a otro **similar a PASCAL** (de ahora en adelante **lenguaje final**) utilizando la herramienta de generación automática ANTLR.

Parte obligatoria

Especificaciones léxicas del lenguaje fuente

Los elementos del lenguaje que aparecen entrecomillados en la gramática (que se muestra en las especificación sintáctica), deben aparecer **tal cual** (sin las dobles comillas) en cualquier programa correctamente escrito en este lenguaje, el resto de elementos se especifican a continuación.

Los **identificadores,** representados por el símbolo `IDENT`**,** son ristras de símbolos compuestas por letras, dígitos (de base decimal), símbolos "$" y guiones bajos "_" (underscore). Empiezan obligatoriamente por una letra o el símbolo "$". Ejemplos correctos: `contador`, `contador1`, `$acumulador_total_2`.

Las **constantes numéricas** pueden ser de **dos tipos**, representados en la gramática por los símbolos terminales `CONSTINT` y `CONSTFLOAT`, y se pueden especificar en **tres bases distintas**: decimal, octal y hexadecimal.

- Las constantes numéricas en **base decimal** (10), estarán compuestas por los dígitos decimales, del 0 al 9, además de los símbolos necesarios en función del tipo al que pertenezcan (enteras o reales).
- Las constantes numéricas en **base octal** (8), **siempre comienzan con el dígito "0"** y estarán compuestas por los dígitos octales, del 0 al 7, además de los símbolos necesarios en función del tipo al que pertenezcan (enteras o reales).
- Las constantes numéricas en **base hexadecimal** (16), **siempre comienzan con "0x"** y estarán compuestas por los dígitos decimales, del 0 al 9 y las letras de la A a la F, además de los símbolos necesarios en función del tipo al que pertenezcan (enteras o reales).

Como se ha mencionado, existen dos tipos posibles de estas constantes: enteras y reales.

- Las constantes numéricas **enteras** son una ristra de dígitos, opcionalmente precedidas de un signo + o -.
- Las constantes numéricas **reales** son dos ristras de dígitos, opcionalmente precedidas de un signo + o - y separadas por el punto decimal.

Ejemplos de constantes correctamente escritas:

- Enteras:
  - Base decimal: `+123 , -69 , 45`
  - Base octal: `0+123 , 0-64 , 045`
  - Base hexadecimal: `0x+123 , 0x-A6F9 , 0xFFFF`
- Reales:
  - Base decimal: `+123.456 , -0.69 , 45.0`
  - Base octal: `0+123.456 , 0-64.77 , 045.16, 00.35`
  - Base hexadecimal: `0x+123.0 , 0x-E.A6F9 , 0x0.FFFF`

Las **constantes literales**, representadas en la gramática por el símbolo terminal `CONSTLIT`, son ristras de símbolos entre comillas simples: `'contenido de la constante literal'`. El contenido de las constantes puede ser cualquier carácter que pueda aparecer en el programa fuente. Si se desea que aparezca la comilla simple como contenido, ésta debe ir precedida del símbolo ' \', por ejemplo, el contenido de la constante: `'constante literal con \'contenido\' entrecomillado'` sería: `constante literal con 'contenido' entrecomillado`

El formato de los **comentarios de propósito general** es: cualquier carácter que pueda aparecer en el código fuente, entre los símbolos `//` hasta el final de la línea, o entre las parejas de símbolos `/*` y `*/`. Lógicamente el contenido del comentario no puede tener los caracteres de finalización del mismo. Este tipo de comentarios pueden aparecer antes o después de cualquier elemento del lenguaje.

Especificación sintáctica del lenguaje fuente

Un programa está compuesto por dos partes, la declaración de constantes (representada en la gramática por el símbolo `defines`) y un conjunto de sucesivas declaraciones de funciones o procedimientos (representada en la gramática por el símbolo `partes`). La diferencia entre procedimientos y funciones es que los primeros devuelven el tipo "void", mientras que las segundas devuelven el tipo "int" o "float".

```
program ::= defines partes
defines ::= Λ | "#define" IDENT ctes defines
ctes ::= CONSTINT | CONSTFLOAT | CONSTLIT
partes ::= part partes | part
part ::= type restpart
restpart ::= IDENT "(" listparam ")" blq
      | IDENT "(" "void" ")" blq
blq ::= "{" sentlist "}"
listparam ::= listparam "," type IDENT | type IDENT
type ::= "void" | "int" | "float"
```

Dentro de las funciones y procedimientos se pueden encontrar sentencias de declaración de variables, asignación y llamadas a funciones y procedimientos.

```
sentlist ::= sentlist sent | sent
sent ::= type lid ";" | IDENT "=" exp ";" | IDENT "(" lexp ")" ";"
      | IDENT "(" ")" ";" | "return" exp ";"
lid ::= IDENT | lid "," IDENT
lexp ::= exp | lexp "," exp
exp ::= exp op exp | factor
op ::= "+" | "-" | "*" | "/" | "%"
factor ::= IDENT "(" lexp ")" | IDENT "(" ")"
      | "(" exp ")" | IDENT | ctes
```

Tanto en esta parte como en la parte opcional hay que asegurarse de que la gramática usada con **ANTLR, es LL(1) y se especifica en notación BNF.**

Especificación de la traducción dirigida por la sintaxis

El objetivo es traducir el código en lenguaje fuente a un código en lenguaje final, cuya gramática proporcionaremos más adelante. El código en lenguaje final estará compuesto por un solo fichero de extensión .pas y cuyo nombre será el mismo que el del fichero de entrada. Así, el fichero de prueba "ejemplo.c" producirá como salida el fichero "ejemplo.pas".

Los comentarios que se encuentren en el lenguaje fuente deben ser ignorados (no aparecerán en el lenguaje final). Por otro lado, tanto los identificadores como las constantes numéricas y literales se expresarán en el lenguaje final de forma idéntica a como aparecen en el lenguaje fuente. Esto no ocurre de forma completa con otros elementos que se detallan a continuación:

- Palabras reservadas sobre tipos de datos básicos como "`int`" y "`float`", que aparecerán en el lenguaje final como "`INTEGER`" y "`REAL`" respectivamente. El tipo "`void`" del lenguaje fuente no tiene representación en el lenguaje final. De hecho, en el lenguaje fuente no es necesario contemplar la posibilidad de que se declaren variables de tipo "`void`".
- Operadores aritméticos. Mientras "+", "–" y "`*`" se escriben de forma idéntica en el lenguaje fuente y el lenguaje final, "`/`" y "`%`" del lenguaje fuente se escriben en el lenguaje final como "`div`" y "`mod`" respectivamente.
- Sentencias "`return`" del lenguaje fuente. Estas sentencias se sustituyen en el lenguaje final por una sentencia de asignación cuya parte izquierda es el nombre de la función en que se ha especificado la sentencia return.
- La declaración de constantes, funciones y procedimientos aparecerán en el lenguaje final según se especifica en la gramática correspondiente en los símbolos *dercte*, *defproc* y *deffun*. Del mismo modo, el uso de los elementos anteriores se especifica en la gramática correspondiente en los símbolos *proc_call* y *factor*.

Se supondrá que el lenguaje fuente tendrá al menos una función cuyo nombre sea "main" que se corresponderá con el programa principal del lenguaje final, cuyo nombre también será "main". Las constantes definidas en el código en lenguaje fuente (representadas por el símbolo `defines` de la gramática del lenguaje fuente), las variables definidas en el procedimiento "main" junto con el resto de funciones y procedimientos declarados, aparecerán en el código en lenguaje final como constantes, variables y subrutinas del programa principal.

La gramática del leguaje final se describe a continuación. Un programa está compuesto por dos partes: la zona de declaraciones (*dcllist*) y la zona de sentencias del programa (*sentlis*).

```
prg ::= "program" IDENTIFIER ";" blq "."
blq ::= dcllist "begin" sentlist "end"
dcllist ::= Λ | dcllist dcl
sentlist ::= sent | sentlist sent
```

La zona de declaraciones es una lista de declaraciones de constantes, tipos, variables, procedimientos y/o funciones.

```
dcl ::= defcte | defvar | defproc | deffun
defcte ::= "const" ctelist
ctelist ::= IDENTIFIER "=" simpvalue ";"
      | ctelist IDENTIFIER "=" simpvalue ";"
simpvalue ::= NUMERIC_INTEGER_CONST | NUMERIC_REAL_CONST
      | STRING_CONST

defvar ::= "var" defvarlist ";"
defvarlist ::= varlist ":" tbas
      | defvarlist ";" varlist ":" tbas
vasrlist ::= IDENTIFIER | IDENTIFIER "," varlist
defproc ::=  "procedure" IDENTIFIER formal_paramlist ";" blq ";"
```

```
deffun ::=  "function" IDENTIFIER formal_paramlist ":" tbas ";" blq
";"
formal_paramlist ::= Λ | "(" formal_param ")"
formal_param ::= varlist ":" tbas
      | varlist ":" tbas ";" formal_param
tbas ::= "INTEGER" | "REAL"
```

La zona de sentencias es una lista de sentencias como asignaciones, sentencias de flujo, llamadas a procedimientos y bloques de ejecución anónimos:

```
sent ::= asig ";" | proc_call ";"
asig ::= id ":=" exp
id := IDENTIFIER
exp ::= exp op exp | factor
op ::= oparit
oparit ::= "+" | "-" | "*" | "div" | "mod"
factor ::= simpvalue | "(" exp ")" | IDENTIFIER subpparamlist
subpparamlist ::= Λ | "(" explist ")"
explist ::= exp | exp "," explist
proc_call ::= IDENTIFIER subpparamlist
```

Además, el código final generado tendrá las siguientes propiedades en cuanto a tabulación se refiere:

- Las sentencias de declaración de constantes y variables se colocarán en un nivel de tabulación mayor (indentación a la derecha) que las palabras reservadas *"const"* y *"var"* respectivamente.
- Todas las sentencias entre las palabras reservadas "begin" y "end", se colocarán en un nivel de tabulación mayor (indentación a la derecha) que estas.

Parte opcional

Especificación sintáctica del lenguaje fuente

Sentencias de control de flujo

Las sentencias de control de flujo se basan en la comprobación de expresiones condicionales, cuya gramática es:

```
lcond ::= lcond opl lcond | cond | "!" cond
opl ::= "||" | "&&"
cond ::= exp opr exp
opr "==" | "<" | ">" | ">=" | "<="
```

Nos encontramos con 4 sentencias de control de flujo cuya gramática se presenta a continuación.

```
sent ::= …
       | "if" "(" lcond ")" blq "else" blq
       | "while" "(" lcond ")" blq
       | "do" blq "until" "(" lcond ")"
       | "for" "(" IDENT "=" exp ";" lcond ";" IDENT "=" exp ")" blq
```

Especificación de la traducción dirigida por la sintaxis

Esta parte contempla sentencias de control de flujo así como otras características a tener en cuenta. Se detallará el formato final de la traducción respecto a estas cuestiones.

Traducción de las sentencias de control de flujo

Como se explicaba anteriormente, las sentencias de control de flujo se basan en la comprobación de expresiones condicionales. La gramática del lenguaje final de estas expresiones es:

```
expcond ::= expcond oplog expcond | factorcond
oplog ::= "or" | "and"
factorcond ::= exp opcomp exp | "(" exp ")" | "not" factorcond
opcomp ::= "<" | ">" | "<=" | ">=" | "="
```

La gramática del lenguaje generado para las sentencias de control de flujo es:

```
sent ::= ...
       | "if" expcond "then" blq "else" blq
       | "while" expcond "do" blq
       | "repeat" blq "until" expcond ";"
       | "for" IDENTIFIER ":=" exp inc exp "do" blq
inc ::= "to" | "downto"
```

Mientras que las tres primeras sentencias tienen una traducción directa, la sentencia `for` tiene una peculiaridad explicada a continuación. Como se puede ver en la gramática del lenguaje final, esta sentencia sólo permite decrementos o incrementos unitarios. Por ello la traducción a realizar desde el lenguaje fuente tendrá dos posibilidades:

- Que la sentencia de actualización del bucle for del código fuente sea un incremento o decremento unitario, p.ej. "`a = a + 1`" o "`a = a - 1`". En cuyo caso la traducción se hará utilizando la regla gramatical del lenguaje final correspondiente al `for`.
- Que la sentencia de actualización del bucle for del código fuente **NO** sea un incremento o decremento unitario. En ese caso habrá que traducir el bucle for del lenguaje fuente como un bucle `while-do` del lenguaje final.

Distinguir entre librerías programas

Si en el programa fuente **no hay ninguna función con identificador "main"**, esto significa que es una librería. Además, las librerías sólo contendrán declaraciones de constantes y subrutinas. Las librerías en el lenguaje final se especifican sustituyendo la palabra reservada "program" por la palabra "unit" y sin cuerpo de programa principal. La gramática del lenguaje final quedaría como sigue:

```
prg ::= … | "unit" IDENTIFIER ";" dcllist "."
```

Mejora de la generación de declaraciones

Según la descripción de la parte obligatoria, el código final tendrá tantas sentencias de declaración de variables como existen en el código fuente. Esta parte opcional consiste en detectar declaraciones consecutivas de identificadores del mismo tipo, generando una sola sentencia de declaración para ellos, siempre respetando el orden de declaración de los identificadores.

Entrada fuente	Salida en lenguaje final

`int vi1, vi2;`
`int vi3;`
`int vi4;`
`float vr1;`
`float vr2, vr3;`
	`vi1, vi2, vi3, vi4 : INTEGER ;`
	`vr1, vr2, vr3 : REAL;`

Lo mismo pasaría en el caso de los parámetros formales de funciones y procedimientos

Entrada fuente
`… void procVariosParam( int a, int b, float c) …`
Salida en lenguaje final
`… procedure procVariosParam(a, b : INTEGER; c : REAL) …`