

Práctica de compilación: Documentación

UPV/EHU
Facultad de Ingeniería Informática

Asignatura de Compilación

eman ta zabal zazu



Universidad del País Vasco Euskal Herriko Unibertsitatea

27/02/2022

Grupo 4

dbernal003@ikasle.ehu.eus
smartin126@ikasle.ehu.eus
xbarberena001@ikasle.ehu.eus

David Bernal Gómez
Sara Martín Aramburu
Xuban Barberena Apezetxea

Índice

Introducción	3
Autoevaluación	3
Descripción por fases y miembros del grupo	3
Análisis léxico	4
Especificación de los tokens	4
Autómata del analizador	7
Casos de Prueba	8

Introducción

La práctica consiste en construir el front-end de un compilador utilizando la técnica de construcción de traductores ascendentes, a partir de un esquema de traducción dirigida por la sintaxis. El lenguaje de entrada (fuente) al compilador es un lenguaje de alto nivel, y el de salida un código de tres direcciones. Utilizamos Flex y Bison.

Para esta primera entrega, ha sido necesario crear un analizador léxico y sintáctico. Se ha hecho uso de Flex en el fichero `traza.l` para reconocer los distintos tipos de tokens de la práctica, además de la herramienta JFlap para el diseño del autómata único.

Además, se han añadido 4 ficheros de prueba, siendo 2 de ellos aceptados correctamente y los otros restantes siendo erróneos y rechazados.

Autoevaluación

Para la autoevaluación valoramos el correcto funcionamiento de la gramática y la gestión del trabajo en el grupo.

Para empezar, el analizador léxico acepta la gramática propuesta, como se puede ver en los [casos de prueba](#). El único problema que se nos presenta es la expresión regular para los comentarios multilínea, podría mejorarse.

Por otro lado, trabajar en la práctica durante algunas de las sesiones de laboratorio como se nos indicó, ha favorecido un correcto desarrollo, sin ningún problema grave destacable. Además, el tiempo invertido por cada miembro del grupo es más o menos similar.

Consideramos que una nota correcta para esta primera entrega sería un notable.

Descripción por fases y miembros del grupo

El trabajo se ha dividido en cinco fases:

- Creación del analizador léxico mediante Flex: La persona encargada de realizar esta tarea ha sido David, dedicando aproximadamente 2h y 30 mins. Además, los demás miembros han colaborado a la hora de obtener algunas expresiones regulares, como el comentario multilínea o los números reales (float)
- Creación de ficheros de prueba: David se ha encargado de realizar los ficheros `pruebaBuena1.in`, `pruebaBuena1.in`, `pruebaBuena1.in` y `pruebaBuena1.in`, y ha comprobado que se aceptaban las gramáticas de los dos primeros ficheros, y se rechazaban las de los dos últimos. Tiempo aproximado 30 mins.
- Diseño del autómata: Xuban ha sido el encargado de acabar esta tarea, dedicando 2h a su diseño y 30 minutos a la elaboración.
- Descripción léxica de los tokens mediante una tabla: Sara ha sido quien ha trabajado en esta tarea, habiéndola terminado en 1h y 30 mins aproximadamente.

- Documentación: Por último, Sara ha trabajado para finalizar el proyecto mediante la realización de un documento en el que se explica todo lo trabajado hasta ahora. La duración aproximada del tiempo dedicado a esta tarea es de 1h30mins.

Análisis léxico

Especificación de los tokens

La tabla con especificación léxica de los tokens es la siguiente:

Nombre token	Descripción informal	Expresión regular	Lexemas
<i>TIDENTIFIER</i>	Comienzan con un carácter alfabético, pueden contener caracteres alfanuméricos y guiones bajos, pero no admiten dos guiones bajos seguidos ni puede acabar con guión bajo.	<code>[a-zA-Z]([_]?[a-zA-Z0-9])*</code>	Sara_123 S_1_2_3 S_Martin
<i>TINTEGER</i>	Concatenación de dígitos, al menos uno.	<code>[0-9]+</code>	4 67
<i>TDOUBLE</i>	Secuencia de dígitos (al menos uno) concatenado con punto, concatenado con dígitos (al menos uno) y concatenado opcionalmente con una parte exponencial que puede tener signo.	<code>[0-9]+[.][0-9]+([eE][+-]?[0-9]+)?</code>	2.037e-4 3.5 5.0e45
<i>TCOMENMULTI</i>	Empiezan por <code>'''</code> y acaban en <code>'''</code> . En medio puede haber cualquier secuencia de caracteres excepto <code>'''</code> y no acepta el carácter <code>#</code> .	<code>'''[^#]*[\n]*[^#]*'''</code>	<code>'''Hola'''</code> <code>''' Hola,</code> <code>soy Sara'''</code>
<i>TCOMEN</i>	Empieza por <code>#</code> y acaba en fin de línea (incluido). En medio no puede aparecer el carácter <code>#</code> .	<code>#[^#]*\n</code>	<code>#Hola</code> <code>#Buenas tardes</code>

<i>RIF</i>	Define la instrucción <i>if</i> , cuando es verdadera se ejecuta el bloque,	<code>if</code>	<code>if</code>
<i>RWHILE</i>	Define la instrucción <i>while</i> , se repite siempre que se cumpla la condición.	<code>while</code>	<code>while</code>
<i>RELSE</i>	Define la instrucción <i>else</i> , se ejecuta cuando la condición no se cumpla.	<code>else</code>	<code>else</code>
<i>RFOREVER</i>	El bucle se repite indefinidamente.	<code>forever</code>	<code>forever</code>
<i>RCONTINUE</i>	La ejecución de un bucle se detiene y se vuelve a evaluar.	<code>continue</code>	<code>continue</code>
<i>RBREAK</i>	Se evalúa la expresión. Si es verdadera, se sale del bucle.	<code>break</code>	<code>break</code>
<i>RDEF</i>	Define un nuevo programa.	<code>def</code>	<code>def</code>
<i>RMAIN</i>	Define el programa principal.	<code>main</code>	<code>main</code>
<i>RBEGIN</i>	Define el comienzo de un nuevo programa.	<code>comienzo</code>	<code>comienzo</code>
<i>RENDPROGRAM</i>	Define el final de un programa.	<code>fin</code>	<code>fin</code>
<i>RREAD</i>	Sirve para definir la instrucción de lectura <i>read</i> .	<code>read</code>	<code>read</code>
<i>PRINT</i>	Sirve para definir la instrucción <i>print</i> .	<code>println</code>	<code>println</code>
<i>RLET</i>	Define la instrucción <i>let</i> .	<code>let</code>	<code>let</code>
<i>RIN</i>	Define la instrucción <i>in</i> .	<code>in</code>	<code>in</code>
<i>RINTEGER</i>	Define la estructura <i>integer</i> .	<code>integer</code>	<code>integer</code>
<i>RFLOAT</i>	Define la estructura	<code>float</code>	<code>float</code>

	<i>float.</i>		
<i>TASSIG</i>	Sirve para asignar valores.	“=”	=
<i>TSEMIC</i>	Define el final de línea en la parte de código.	“;”	;
<i>TDOSPUNTOS</i>	Define el inicio de una nueva instrucción.	“:”	:
<i>TCOMA</i>	Sirve para diferenciar elementos en una enumeración.	“,”	;
<i>TLLAVEI</i>	Define la apertura de una llave.	“{”	{
<i>TLLAVED</i>	Define el cierre de una llave.	“}”	}
<i>TPARENI</i>	Define la apertura de un paréntesis.	“(”	(
<i>TPAREND</i>	Define el cierre de un paréntesis.	“)”)
<i>TAND</i>	Define el operador <i>and</i> .	“&”	&
<i>TEQUAL</i>	Sirve para comparar si dos elementos son equivalentes.	“==”	==
<i>TMAYOR</i>	Sirve para comparar si un elemento es mayor que otro.	“>”	>
<i>TMENOR</i>	Sirve para comparar si un elemento es menor que otro.	“<”	<
<i>TMAYOREQ</i>	Sirve para comparar si un elemento es mayor o igual a otro.	“>”	>=
<i>TMENOREQ</i>	Sirve para comparar si un elemento es menor o igual a otro.	“<”	<=
<i>TNOTEQUAL</i>	Sirve para comparar si dos elementos no son equivalentes.	“/”	/=
<i>TPLUS</i>	Define la operación	“+”	+

Casos de Prueba

Para comprobar el correcto funcionamiento del analizador léxico, se han utilizado un total de 5 ficheros de prueba:

El primer fichero es **prueba2.in**, facilitado por la profesora durante la sesión de laboratorio. Además de para completar el analizador léxico, este fichero nos sirvió también para entender de una manera más práctica, la estructura de la gramática. El fichero prueba2.in no se encuentra en la carpeta de entregables.

Ficheros **pruebaBuena1.in** y **pruebaBuena2.in**: son dos ficheros que contienen una gramática aceptada por el analizador léxico. PruebaBuena1.in es una variante del fichero facilitado prueba2.in, pero contiene algunos añadidos para comprobar los comentarios. Por otra parte, pruebaBuena2.in contiene dos funciones auxiliares, con el fin de comprobar que acepta correctamente la estructura de los subprogramas.

Finalmente, los ficheros **pruebaMala1.in** y **pruebaMala2.in**, dos ficheros que contienen código en una gramática que no es aceptada. En pruebaMala1.in se realizan asignaciones gramaticalmente incorrectas y no se cierra el programa principal. Por otro lado, pruebaMala2.in tiene comentarios y asignaciones incorrectas, además de una estructura while no aceptada por la gramática.

Para probar el analizador léxico, se ha añadido un fichero makeFile. Bastará con hacer make en el terminal para analizar los cuatro ficheros de prueba (prueba2.in no se incluye). Como se puede ver, acepta las gramáticas de las pruebas buenas y rechaza las pruebas malas.

```
david@david-virtual-machine:~/Escritorio/C/practica/practicalexsin$ make
./parser <../pruebaBuena1.in
ha comenzado...

ha finalizado BIEN...

./parser <../pruebaBuena2.in
ha comenzado...

ha finalizado BIEN...

./parser <../pruebaMala1.in
ha comenzado...

line 8: syntax error at ':'
ha finalizado MAL...

./parser <../pruebaMala2.in
ha comenzado...

Token desconocido: #
line 5: syntax error at '#'
ha finalizado MAL...

david@david-virtual-machine:~/Escritorio/C/practica/practicalexsin$
```