



Instituto Politécnico Nacional
Escuela Superior de Cómputo



Compiladores

Practica 5: Bison

Nombre: Mauro Sampayo Hernández

Grupo: 3CV17

Profesor: Hernández Olvera Luis Enrique

Fecha de entrega: 15 de diciembre del 2021

1. Introducción:

1.1 Bison

Bison es un generador de analizadores sintácticos de propósito general que convierte una descripción gramatical para una gramática independiente del contexto LALR(1) en un programa en C que analice esa gramática.

Para que Bison analice un lenguaje, este debe ser descrito por una **gramática independiente del contexto**. Esto quiere decir que debe especificar uno o más grupos sintácticos y dar reglas para construirlos desde sus partes.

En las reglas gramaticales formales para un lenguaje, cada tipo de unidad sintáctica o agrupación se identifica por un **símbolo**. Aquellos que son construidos agrupando construcciones más pequeñas de acuerdo con reglas gramaticales se denominan **símbolos no terminales**; aquellos que no pueden subdividirse se denominan **símbolos terminales** o **tipos de tokens**. Denominamos **token** a un fragmento de la entrada que corresponde a un solo símbolo terminal, y **grupo** a un fragmento que corresponde a un solo símbolo no terminal.

Cada símbolo no terminal debe poseer reglas gramaticales mostrando como está compuesto de construcciones más simples.

Se debe distinguir un símbolo no terminal como el símbolo especial que define una declaración completa en el lenguaje. Este se denomina **símbolo de arranque**.

El analizador de Bison lee una secuencia de tokens como entrada, y agrupa los tokens utilizando las reglas gramaticales. Si la entrada es válida, el resultado final es que la secuencia de tokens entera se reduce a una sola agrupación cuyo símbolo es el símbolo de arranque de la gramática.

1.2 Estructura de los programas en Lex

La forma general de una gramática de Bison es la siguiente:

%{

declaraciones en C

%}

Declaraciones de Bison

%%

Reglas gramaticales

%%

Código C adicional

Los “%%”, “%{“ y “%}” son signos de puntuación que aparecen en todo archivo de gramática de Bison para separar las secciones.

1.2.1 Declaraciones en C

La sección de declaraciones en C contiene definiciones de macros y declaraciones de funciones y variables que se utilizan en las acciones en las reglas de la gramática. Estas se copian al principio del archivo del analizador de manera que precedan la definición de yyparse. Se puede utilizar “#include” para obtener las declaraciones de un archivo de cabecera. Si no se necesita ninguna declaración en C, se pueden omitir los delimitadores “%{“ y “%}” que delimitan esta sección.

1.2.2 Declaraciones de Bison

Las declaraciones de Bison declaran los nombres de los símbolos terminales y no terminales, y también pueden describir la precedencia de operadores y los tipos de datos de los valores semánticos de varios símbolos.

1.2.3 Reglas Gramaticales

Las reglas gramaticales definen cómo construir cada símbolo no terminal a partir de sus partes. Debe haber siempre al menos una regla gramatical, y el primer “%%” (que precede a las reglas gramaticales).

1.2.4 Código C adicional

La sección de *código C adicional* se copia al pie de la letra a la salida del fichero del analizador, al igual que la sección de *declaraciones en C* que se copia al principio. Este es el lugar más conveniente para poner cualquier cosa que quiera tener en el archivo del analizador pero que no deba venir antes que la definición de yyparse. Las definiciones de yylex e yyerror a menudo van en esta parte. Si esta sección está vacía, puede omitir el “%%” que los separa de las reglas gramaticales.

2. Problemas enfrentados al programar la práctica:

Uno de los primeros problemas de importancia que surgieron durante el desarrollo de la práctica fue en relación con el análisis léxico de un Salto de línea “\n”, pues al no haberse incluido su evaluación en un principio, el analizador sintáctico recibía Tokens erróneos al momento de evaluarla, provocando errores sintácticos. Dicho problema fue resuelto incluyendo la evaluación de el Salto e línea “\n” en el analizador léxico.

Posteriormente, y además el mayor problema que surgió fue el de implementar números enteros y decimales con signo para que pudiesen ser reconocidos tanto por el analizador léxico como por el sintáctico. Este problema se pudo resolver gracias a la directiva “%prec”, la cual declara la precedencia de alguna regla en particular especificando un símbolo terminal cuya precedencia deba utilizarse para dicha regla. De esta manera se le pudo especificar a Bison que en caso de que un número tenga un signo positivo (+) o negativo (-) precediéndolo, este será un número positivo o negativo respectivamente; y dicha precedencia se pudo implementar por medio de la creación de dos Tokens o símbolos terminales, los cuales representan las dos reglas mencionadas anteriormente, y a las cuales se les asignó la segunda precedencia más alta entre los operadores aceptados por el analizador sintáctico de la práctica.

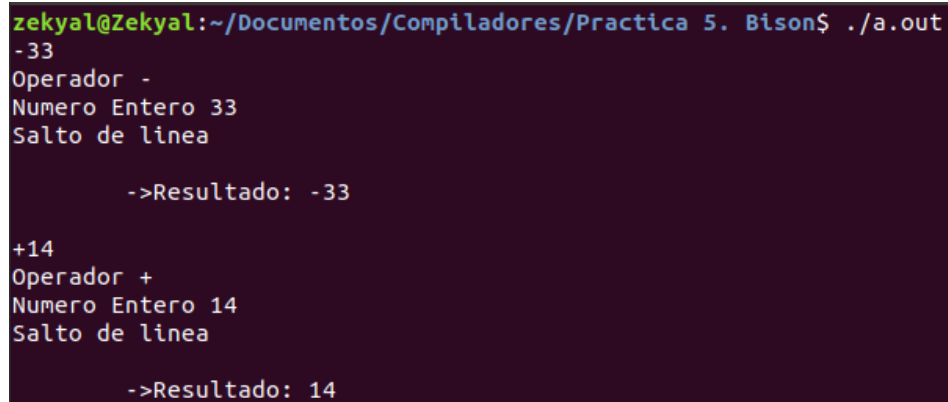
Finalmente, el último problema de importancia a mencionar tuvo relación con la implementación de la operación de Módulo como función. Para solucionar dicho problema se partió asignándole un Token a cualquier combinación de letras en mayúscula y minúscula que generen en su conjunto la palabra “MOD”; a dicho Token se le asignó la prioridad más alta entre los operadores aceptados por el analizador sintáctico de la práctica. Finalmente se añadieron las respectivas reglas gramaticales para lograr la correcta ejecución de dicha función.

3. Pruebas y Capturas de Pantalla:

Se realizó el desarrollo de un analizador léxico en Flex y un analizador sintáctico en Bison, los cuales reconocen ciertos lexemas escritos en ANSI C los cuales se enlistarán a continuación junto con algunas pruebas realizadas con el objetivo de mostrar su correcto funcionamiento:

3.1 Números Enteros con y sin signo

- Número entero con signo:



```
zekyal@Zekyal:~/Documentos/Compiladores/Practica 5. Bison$ ./a.out
-33
Operador -
Numero Entero 33
Salto de línea

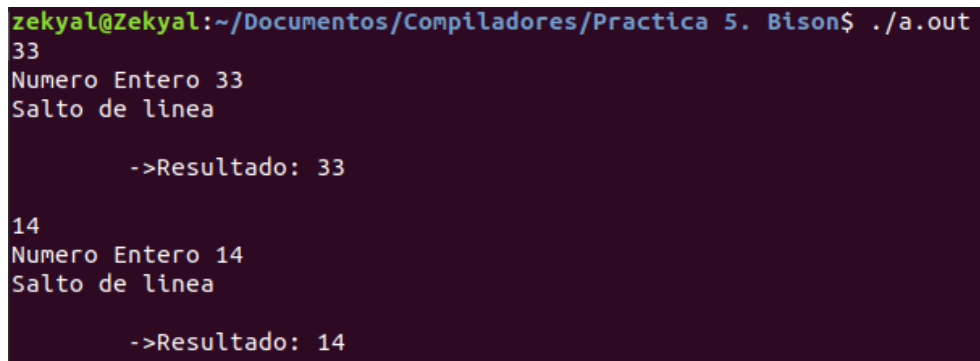
->Resultado: -33

+14
Operador +
Numero Entero 14
Salto de línea

->Resultado: 14
```

Figura 1. Números enteros CON signo

- Número entero sin signo:



```
zekyal@Zekyal:~/Documentos/Compiladores/Practica 5. Bison$ ./a.out
33
Numero Entero 33
Salto de línea

->Resultado: 33

14
Numero Entero 14
Salto de línea

->Resultado: 14
```

Figura 2. Números enteros SIN signo

3.2 Números Decimales con y sin signo

- Número decimal con signo:

```
zekyal@Zekyal:~/Documentos/Compiladores/Practica 5. Bison$ ./a.out
-234.55
Operador -
Numero Decimal 234.55
Salto de línea

->Resultado: -234.550003

+23.44
Operador +
Numero Decimal 23.44
Salto de línea

->Resultado: 23.440001
```

Figura 3. Números decimales CON signo

- Número decimal sin signo:

```
zekyal@Zekyal:~/Documentos/Compiladores/Practica 5. Bison$ ./a.out
234.55
Numero Decimal 234.55
Salto de línea

->Resultado: 234.550003

23.44
Numero Decimal 23.44
Salto de línea

->Resultado: 23.440001
```

Figura 4. Números decimales SIN signo

3.3 Operaciones Matemáticas

A continuación, se enlistan todas las operaciones matemáticas que reconoce el analizador léxico y sintáctico, junto con algunas pruebas realizadas utilizando números con y sin signo:

3.3.1 Suma “+”

- Suma con números con signo:

```
zekyal@Zekyal:~/Documentos/Compiladores/Practica 5. Bison$ ./a.out
-24+-45
Operador -
Numero Entero 24
Operador +
Operador -
Numero Entero 45
Salto de linea

->Resultado: -69

-234.2++23.4
Operador -
Numero Decimal 234.2
Operador +
Operador +
Numero Decimal 23.4
Salto de linea

->Resultado: -210.800003
```

Figura 5. Sumas con valores numéricos CON signo

- Suma son números sin signo:

```
zekyal@Zekyal:~/Documentos/Compiladores/Practica 5. Bison$ ./a.out
234+14
Numero Entero 234
Operador +
Numero Entero 14
Salto de linea

->Resultado: 248

23.5+6
Numero Decimal 23.5
Operador +
Numero Entero 6
Salto de linea

->Resultado: 29.500000
```

Figura 6. Sumas con valores numéricos SIN signo

3.3.2 Resta “-”

- Resta con números con signo:

```
zekyal@Zekyal:~/Documentos/Compiladores/Practica 5. Bison$ ./a.out
-234--234
Operador -
Numero Entero 234
Operador -
Operador -
Numero Entero 234
Salto de línea

->Resultado: 0

63.5--12
Numero Decimal 63.5
Operador -
Operador +
Numero Entero 12
Salto de línea

->Resultado: 51.500000
```

Figura 7. Restas con valores numéricos CON signo

- Resta con números sin signo:

```
zekyal@Zekyal:~/Documentos/Compiladores/Practica 5. Bison$ ./a.out
23-45
Numero Entero 23
Operador -
Numero Entero 45
Salto de línea

->Resultado: -22

63.5-.5
Numero Decimal 63.5
Operador -
Numero Decimal .5
Salto de línea

->Resultado: 63.000000
```

Figura 8. Restas con valores numéricos SIN signo

3.3.3 Multiplicación “*”

- Multiplicación con números con signo:

```
zekyal@Zekyal:~/Documentos/Compiladores/Practica 5. Bison$ ./a.out
-23*-2
Operador -
Numero Entero 23
Operador *
Operador -
Numero Entero 2
Salto de línea

->Resultado: 46

-63.56*+2
Operador -
Numero Decimal 63.56
Operador *
Operador +
Numero Entero 2
Salto de línea

->Resultado: -127.120003
```

Figura 9. Multiplicaciones con valores numéricos CON signo

- Multiplicación con números sin signo:

```
zekyal@Zekyal:~/Documentos/Compiladores/Practica 5. Bison$ ./a.out
6*4
Numero Entero 6
Operador *
Numero Entero 4
Salto de línea

->Resultado: 24

12.67*3.2
Numero Decimal 12.67
Operador *
Numero Decimal 3.2
Salto de línea

->Resultado: 40.544003
```

Figura 10. Multiplicaciones con valores numéricos SIN signo

3.3.4 División “/”

- División con números con signo:

```
zekyal@Zekyal:~/Documentos/Compiladores/Practica 5. Bison$ ./a.out
-10/5
Operador -
Numero Entero 10
Operador /
Numero Entero 5
Salto de línea

->Resultado: -2

-64.4/-2.2
Operador -
Numero Decimal 64.4
Operador /
Operador -
Numero Decimal 2.2
Salto de línea

->Resultado: 29.272728
```

Figura 11. Divisiones con valores numéricos CON signo

- División con números sin signo:

```
zekyal@Zekyal:~/Documentos/Compiladores/Practica 5. Bison$ ./a.out
3/2
Numero Entero 3
Operador /
Numero Entero 2
Salto de línea

->Resultado: 1

23.4/2
Numero Decimal 23.4
Operador /
Numero Entero 2
Salto de línea

->Resultado: 11.700000
```

Figura 12. Divisiones con valores numéricos SIN signo

3.3.5 Función Módulo

Para esta operación, se mostrará la ejecución de la función utilizando combinaciones de letras mayúsculas y minúsculas para cada prueba.

- Modulo con números con signo:

```
zekyal@Zekyal:~/Documentos/Compiladores/Practica 5. Bison$ ./a.out
MOD(-24,-2)
Modulo
Simbolo (
Operador -
Numero Entero 24
Simbolo ,
Operador -
Numero Entero 2
Simbolo )
Salto de linea

->Resultado: 0

mod(+34.5,-3.333)
Modulo
Simbolo (
Operador +
Numero Decimal 34.5
Simbolo ,
Operador -
Numero Decimal 3.333
Simbolo )
Salto de linea

->Resultado: 1.170001
```

Figura 13. Función Módulo con valores numéricos CON signo

- Modulo con números sin signo:

```
zekyal@Zekyal:~/Documentos/Compiladores/Practica 5. Bison$ ./a.out
mOd(2,3)
Modulo
Simbolo (
Numero Entero 2
Simbolo ,
Numero Entero 3
Simbolo )
Salto de linea

->Resultado: 2

MoD(5.6,3.4)
Modulo
Simbolo (
Numero Decimal 5.6
Simbolo ,
Numero Decimal 3.4
Simbolo )
Salto de linea

->Resultado: 2.200000
```

Figura 14. Función Módulo con valores numéricos SIN signo

- Modulo combinado con otros operadores:

```

zekyal@Zekyal:~/Documentos/Compiladores/Practica 5. Bison$ ./a.out
-23.4+mOD(3,4)/-3+12.3
Operador -
Numero Decimal 23.4
Operador +
Modulo
Simbolo (
Numero Entero 3
Simbolo ,
Numero Entero 4
Simbolo )
Operador /
Operador -
Numero Entero 3
Operador +
Numero Decimal 12.3
Salto de linea

->Resultado: -12.099999

Mod(3.4+1,6-5)*2--4/mOd(5.6,-4+5)
Modulo
Simbolo (
Numero Decimal 3.4
Operador +
Numero Entero 1
Simbolo ,
Numero Entero 6
Operador -
Numero Entero 5
Simbolo )
Operador *
Numero Entero 2
Operador -
Operador -
Numero Entero 4
Operador /
Modulo
Simbolo (
Numero Decimal 5.6
Simbolo ,
Operador -
Numero Entero 4
Operador +
Numero Entero 5
Simbolo )
Salto de linea

->Resultado: 7.466668

```

Figura 15. Función Módulo combinada con otros operadores

4. Conclusión:

Los analizadores sintácticos son una herramienta esencial para llevar a cabo el análisis de gramáticas independientes del contexto, las cuáles son la base de muchos lenguajes de programación y de esta manera poder llevar a cabo la construcción de intérpretes que validen a partir de dichas gramáticas, patrones léxicos de entrada. De esta forma se puede llevar a cabo la comprobación de la sintaxis dichos patrones para asegurarnos de que estos pertenecen a la gramática en cuestión, y poder realizar ciertas acciones dependiendo de la sintaxis que conformen los patrones recibidos.

Además, a partir de un analizador sintáctico, complementado con analizadores léxicos y semánticos es que se puede crear la base de cualquier lenguaje de programación, razón por la que el uso de herramientas que generan dichos analizadores (como lo es para este caso Flex y Bison) son muy eficientes para construir interpretes para dichos lenguajes.

5. Referencias:

- Aho, A., 2011. Compiladores. 2da edición. Pearson Educación de México, SA de CV, pp.140-144.
- Levine, J., 2009. Flex & Bison. 1ra edición. O'Reilly Media Inc., pp.19-79.
- Donnelly, C. and Stallman, R., 1999. Bison 1.27. [online] es.tldp.org. Available at: <<http://es.tldp.org/Manuales-LuCAS/BISON/bison-es-1.27.html>> [Accedido el 12 de diciembre de 2021].
- Paxson, V., 1995. Flex - un generador de analizadores léxicos. [online] es.tldp.org. Disponible en: <<http://es.tldp.org/Manuales-LuCAS/FLEX/flex-es-2.5.html#SEC5>> [Accedido el 12 de diciembre de 2021].