Compiladores:
Proyecto Final

Construcción de un compilador

M. CONCHA VÁZQUEZ
A. FLORES MARTÍNEZ
P.J.S. SÁNCHEZ PÉREZ
G.G. ÁNGEL IVÁN
V.S. EDUARDO EDER

Facultad de Ciencias, UNAM

Facultad de Ciencias
Universidad Nacional Autónoma de México
http://www.fciencias.unam.mx/

Título del proyecto:
Proyecto Final

Tema del proyecto:
Análisis léxico; expresiones regulares; *lexemas*; *tokens* y clases léxicas; Analizar sintáctico *LALR(1)*; `yacc`; `lex`; `flex`; `bison`; código intermedio; cuadruplas; código objeto ensamblador; `MIPS`; reglas de producción; pseudovariables; definición dirigida por sintaxis; esquema de traducción.

Periodo:
Diciembre, 2018.

Número de Grupo:
7006

Participantes:
Concha Vázquez Miguel.
Flores Martínez Andrés.
Gladín García Ángel Iván.
Sánchez Pérez Pedro Juan Salvador.
Vázquez Salcedo Eduardo Eder.

Supervisores:
Profesor Adrian Ulises Mercado Martínez.
Karla Adriana Esquivel Guzmán.
Carlos Gerardo Acosta Hérnandez.

Fecha en que se completó el proyecto:
14 de diciembre del 2018.

Resumen:

A partir de una gramática provista se definieron las expresiones regulares para los símbolos terminales. Más aún, se utilizó la herramienta `Flex` para implementar el analizador léxico y devolver los *tokens* de las clases léxicas correspondientes. Posteriormente, se hizo la especificación de la gramática en `Bison` (versión nueva de `yacc`) para el análisis sintáctico y se dio semántica a las reglas de producción con ayuda de la definición dirigida por sintaxis. El objetivo fue la generación de código intermedio de tres direcciones y su almacenamiento en cuadruplas con la intención de la obtención posterior de código objeto en ensamblador `MIPS`.

Índice general

Capítulo 1

Documentos solicitados

Seguir los enlaces a los documentos que se quieran en cada caso:

1.1. Manual de usuario

Para consultar el documento del manual de usuario:
Enlace al manual de usuario.

1.2. Diseño y especificación del analizador léxico

Enlace al documento en donse se especifica el analizador léxico.

1.3. Definición dirigida por sintaxis

Enlace al documento que desglosa la definición dirigida por sintaxis.

1.4. Esquema de traducción

Enlace al documento que contiene el esquema de traducción.

Capítulo 2

Ejecución y Ejemplos

Deberá ubicarse en el directorio en donde fueron extraídos los archivos y luego de dirigirse al directorio `/src`.

2.1. Compilación

2.1.1. *Lexer*

Primero deberá generarse el analizar léxico con el siguiente comando:

```
flex lexer.l
```

Esto generará el archivo `lex.yy.c` que podría compilarse con el comando:

```
gcc lex.yy.c -o lexer
```

Para generar como tal el `lexer`.

2.1.2. *Parser* y Análisis semántico

Se provee la posibilidad de llevar a cabo únicamente un análisis sintáctico —y no el semántico al mismo tiempo— debido al archivo `parser.y` provisto. Sin embargo, recomendamos llevar a cabo ambos al mismo tiempo, pues en análisis sobre el código fuente se reportará también si la entrada es válida o no en un sentido sintáctico[1].

Primero deberá generarse en este caso el *parser* que lleva a cabo también el análisis semántico con el comando:

```
bison -d semantic.y
```

Lo cual generará los archivos `semantic.tab.h` y `semantic.tab.c`.

[1] En caso de generar el *parser* con `bison -d parser.y`, deberá de generarse luego de nueva cuenta el analizador léxico, pero fijándose que se incluya ahora el archivo de cabecera `parser.tab.h` en lugar de `semantic.tab.h`.

2.1.3. Compilación

Ahora se procede a compilar los archivos generados antes, además de los que son necesarios para la ejecución del análisis semántico como sigue:

```
gcc semantic.tab.c lex.yy.c mastertab.c intermediate_code.c backpatch.c attributes.c
                            pila.c main.c -o analizador
```

En donde el nombre de `analizador` es posible cambiarlo como se desee.

2.2. Ejecución

Ahora, para ejecutar el análisis sobre un archivo de entrada llamado `archivo`, bastará con hacer:

```
./analizador archivo
```

2.2.1. Ejemplos provistos

Proporcionamos algunos ejemplos para ver cómo se ejecuta todo sobre diferentes construcciones permitidas. Hay dos tipos de ejemplos: los que representan código bien hecho y los que tienen código que a propósito tiene errores para ver cómo se manejan estos casos y cómo se reportan al usuario los errores.

Ejemplos *correctamente formados*:

Se encuentran localizados en el directorio `/Ejemplos` y tienen nombres del tipo `test_n.txt` donde la `n` denota el número del ejemplo. Bastará con ejecutarlos como se indicó antes:

```
./analizador Ejemplos/test_n.txt
```

Ejemplos *incorrectamente formados*:

Se encuentran en el directorio `/Ejemplos/Errores`. Para analizarlos, usar el comando:

```
./analizador Ejemplos/test_ejemplo_n.txt
```

Donde aquí también la `n` denota al número del ejemplo.

Capítulo 3

Conclusiones

A lo largo del desarrollo de este proyecto nos fue posible ver que `Flex` y `Bison/yacc` en efecto son de gran ayuda para el diseño y creación de analizadores léxicos y sintácticos. Además, una vez tenida la definición dirigida por sintaxis y un panorama del esquema de traducción, pasar todo a `Bison` es sencillo. LA desventaja evidentemente es que no permite trabajar con atributos heredados dado que genera analizadores del tipo *LALR(1)*. Para solventar este inconveniente es posible trabajar con variables globales y pilas para los atributos heredados.

Llevar a cabo el diseño de un compilador para un lenguaje de programación a partir de su gramática es una gran tarea que no debe ser menospreciada. Requiere de mucha paciencia y una capacidad de abstracción considerable. Fue interesante realizar un trabajo de esta magnitud en equipo pues nos tuvimos que poner de acuerdo en varios aspectos del diseño desde un principio y coordinar nuestros esfuerzos para que hubiera el menor número posible de conflictos al momento de juntar el código.