RELIPMOC: Construcción de un Compilador Básico haciendo uso de las herramientas JLex y CUP
Semestre 2005-1

Mary Carmen Trejo Ávila
mary@ciencias.unam.mx
LDNL, UNAM

2 de septiembre de 2004

Índice

1. Introducción y Motivación. **3**

1.1. Compilador. 3
1.2. Análisis léxico. 3
1.3. Gramática independiente del contexto. 4
1.4. Árbol de análisis sintáctico. 4
1.5. Análisis sintáctico. 4
1.6. Definición dirigida por la sintaxis. 5
1.7. Esquema de traducción. 5
1.8. Análisis semántico. 6
1.9. Código de tres direcciones. 6
1.10. Generación de código intermedio. 6
1.11. Optimización de código. 6
1.12. Generación de código objeto. 7
1.13. Tabla de símbolos. 7
1.14. Gestión de errores. 7

2. Herramienta JLex: generador automático de analizadores léxicos. **8**

2.1. Código de usuario. 8
2.2. Directivas JLex. 9
2.3. Expresiones regulares. 10

3. Herramienta CUP: generador automático de analizadores sintácticos LALR. 11

3.1. Definición de paquetes e importación de paquetes necesarios. 11

3.2. Código de usuario. 11

3.3. Declaración de símbolos terminales y no terminales. 12

3.4. Declaraciones de precedencia. 12

3.5. Definición del símbolo inicial de la gramática y las reglas de producción. 12

4. RELIPMOC: Un compilador propio. 14

4.1. Introducción 14

4.2. Descripción del lenguaje de entrada. 15

4.2.1. Componentes léxicos de RELIPMOC. 15

4.2.2. Sintaxis de RELIPMOC. 16

4.3. Descripción de salida. 16

4.4. Implementación. 16

5. Proyecto. 18

1. Introducción y Motivación.

1.1. Compilador.

Es un programa que lee un programa escrito en un lenguaje fuente, y lo traduce a un lenguaje objeto de bajo nivel. Además generará una lista de los posibles errores que tenga el programa fuente. [ASU86] [App98]

La estructura de un compilador puede listarse de la siguiente manera:

1. Análisis léxico.
2. Análisis sintáctico.
3. Análisis semántico.
4. Generación de código intermedio.
5. Optimización de código intermedio.
6. Generación de código objeto.

Con cada una de estas fases interactua la Tabla de símbolos y la Gestión de errores.

1.2. Análisis léxico.

El analizador léxico (scanner), lee un texto fuente y lo transforma en una secuencia ordenada de elementos léxicamente válidos. Un carácter o conjunto de estos que constituya un componente léxico se llama lexema (token). Como componentes léxicos consideramos: palabras reservadas, separadores, operadores, identificadores, constantes y signos de puntuación.

Principales funciones de un analizador léxico:

- Manejar el archivo fuente (e.g. abrirlo, leerlo, cerrarlo).
- Generar y entregar tokens bajo la petición del analizador sintáctico.
- Rechazar un carácter o conjunto de estos que no concuerden con patrones especificados. Entendamos como patrón una expresión regular que se define en el lenguaje.
- Ignorar comentarios, espacios en blanco y tabuladores.
- Reconocer las palabras reservadas del lenguaje.
- Gestionar errores, contando los saltos de línea y asociando los mensajes de error con el número de la línea del archivo fuente donde se producen.
- Guardar tokens junto con su atributo en una tabla de símbolos. Este atributo es información adicional relevante, habitualmente con relación a los identificadores.

1.3. Gramática independiente del contexto.

Una gramática independiente del contexto tiene 4 componentes:

1. Un conjunto de símbolos terminales.
2. Un conjunto de símbolos no terminales.
3. La denominación de uno de los no terminales como símbolo inicial.
4. Un conjunto de producciones, en el que cada producción consta de un no terminal, llamado lado izquierdo de la producción, una flecha y una secuencia de terminales y/o no terminales, llamado lado derecho de la producción.

Definición 1 *Una gramática independiente de contexto es un cuarteto:*

$$G = (T, N, S, P)$$

donde T es un conjunto finito de símbolos terminales, N es un conjunto de símbolos no terminales, $S \in N$ es un símbolo de comienzo y P es un conjunto finito de reglas de producción. Cada regla de producción en P es escrita convenientemente en la forma $A ::= \alpha$ donde $A \in N$ es un símbolo no terminal y $\alpha \in (T \cup N)^*$ es una cadena de símbolos terminales y no terminales.

1.4. Árbol de análisis sintáctico.

Indica gráficamente cómo del símbolo incial de una gramática deriva una cadena del lenguaje.

1.5. Análisis sintáctico.

Una forma de especificar la sintaxis de un lenguaje de programación (aspecto de sus programas) es mediante gramáticas independientes del contexto o BNF (Forma de Backus-Naur).

El analizador sintáctico (parser) recibe los tokens y determina si dicha cadena puede ser generada por una gramática. Para ello debe encontrarse un árbol de análisis sintáctico o, en su defecto, generar un error. A su vez este deberá recuperarse de los errores que ocurren frecuentemente para poder continuar procesando el resto de la entrada.

La mayoría de los métodos de análisis sintáctico están comprendidos en dos clases:

1. Métodos descendente (LL).
   - Análisis sintáctico descendente recursivo.
   - Análisis sintáctico predictivo.
   - Análisis sintáctico predictivo no recursivo.
2. Métodos ascendente (LR).

- Análisis sintáctico por desplazamiento y reducción.
- Análisis sintáctico por precedencia de operadores.
- Análisis sintáctico LR.
- Análisis sintáctico SLR.
- Análisis sintáctico LR canónico.
- Análisis sintáctico LALR.

Estos términos, descendente y ascendente, hacen referencia al orden en que se construyen los nodos del árbol de análisis sintáctico. En el primero, la construcción se inicia en la raíz y avanza hacia las hojas, mientras que en el segundo, la construcción se inicia en las hojas y avanza hacia la raíz. Las hojas de un árbol de análisis sintáctico, leídas de izquierda a derecha, forman la producción del árbol, que es la cadena generada o derivada del no terminal de la raíz del árbol de análisis sintáctico.

Los métodos más eficientes de análisis, tanto descendente como ascendente, no funcionan para todas las gramáticas independientes del contexto, sino sólo para las gramáticas que cumplen unas determinadas condiciones. En la mayoría de los casos, pueden encontrarse para los lenguajes de programación gramáticas de tipo LL o LR que los generen.

Los distintos analizadores sintácticos serán vistos con detalle en el salón de clase, así como la manera de atacar los errores en cada uno de ellos.

1.6. Definición dirigida por la sintaxis.

Es una generalización de una gramática independiente del contexto en la que cada símbolo gramatical tiene un conjunto de atributos asociado, dividido en dos subconjuntos llamados atributos sintetizados y los heredados.

Típicos ejemplos de atributos son: una cadena o un número (valor de una expresión), un tipo, una posición de memoria o el código objeto (destino) de una función. Si a es un atributo de un símbolo gramatical X, escribiremos $X.a$ para referirnos al valor del atributo a asociado al símbolo gramatical X. Si en una producción de la gramática independiente del contexto aparece más de una vez un símbolo gramatical X, entonces se debe añadir un subíndice a cada una de las apariciones para podernos referir a los valores de los atributos de un modo no ambiguo: $X_1.a, X_2.a, ..., X_n.a$.

1.7. Esquema de traducción.

Es una gramática independiente del contexto en la que se asocian atributos con los símbolos gramaticales y se insertan acciones semánticas encerradas entre llaves $\{\}$. Estos esquemas pueden tener tanto atributos sintetizados como heredados.

1.8. Análisis semántico.

La semántica de un lenguaje de programación (significado de sus programas) es la parte de su especificación que no puede ser descrita por la sintaxis.

El analizador semántico se encarga de detectar la validez semántica (e.g. declaración de identificadores, comprobación: de tipos, del flujo de control, de unicidad) del árbol generado por el analizador sintáctico.

Para ello se asocia información a una construcción del lenguaje de programación proporcionando atributos a los símbolos de la gramática. Los valores de los atributos se calculan mediante "reglas semánticas" asociadas a las producciones gramaticales.

Hay dos notaciones para asociar reglas semánticas con producciones, las definiciones dirigidas por la sintaxis y los esquemas de traducción.

Este tema será visto con mayor cuidado en el salón de clase.

1.9. Código de tres direcciones.

Es una secuencia de proposiciones de la forma general:

$$x := y \; op \; z$$

donde x, y y z son nombres, constantes o variables temporales generados por el compilador; *op* representa cualquier operador, como un operador aritmético de punto fijo o flotante, o un operador lógico sobre datos con valores booleanos.

1.10. Generación de código intermedio.

El generador de código intermedio transforma un árbol semántico en una representación en un lenguaje intermedio cercano al código objeto. Este puede ser representado mediante:

- Código de tres direcciones (cuádruplos, triples y triples indirectos).
- Árboles abstractos
- Grafos dirigidos acíclicos
- Código de máquina virtual

1.11. Optimización de código.

El optimizador de código realiza modificaciones sobre el código intermedio para mejorar la eficiencia en velocidad y tamaño.

1.12. Generación de código objeto.

El generador de código objeto transforma el código intermedio optimizado en código objeto de bajo nivel. Este puede ser ensamblador o código máquina.

1.13. Tabla de símbolos.

Son estructuras de datos que almacenan toda la información de los identificadores del archivo fuente.

Las funciones principales de una tabla de símbolos es colaborar con las comprobaciones semánticas (uso de variables no declaradas, variables declaradas varias veces, incompatibilidad en los tipos de una expresión, ámbitos, etc.), así como facilitar ayuda a la generación de código.

Las operaciones básicas para estas estructuras son: insertar, consultar y borrar. Por la complejidad de estas operaciones es común el uso de tablas hash para su implementación.

1.14. Gestión de errores.

El manejo de errores es una de las misiones más importantes de un compilador, aunque, al mismo tiempo, es lo que más dificulta su realización. Donde más se utiliza es en las fases de análisis sintáctico y semántico, aunque los errores se pueden descubrir en cualquier fase de un compilador. Es una tarea difícil, por dos motivos:

- A veces unos errores ocultan otros.
- A veces un error provoca una avalancha de muchos errores que se solucionan con el primero.

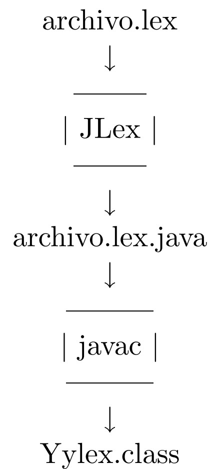
Es conveniente un buen manejo de errores, y que el compilador detecte todos los errores que tiene el programa y no se pare en el primero que encuentre. Hay dos criterios a seguir a la hora de manejar errores:

- Detenerse al detectar el primer error.
- Detectar el mayor número posible de errores de una pasada.

2. Herramienta JLex: generador automático de analizadores léxicos.

Para construir un compilador el primer paso que debemos dar es el de desarrollar el analizador léxico de nuestro lenguaje. Para esta tarea existen diversas herramientas tales como: Flex, Lex, Alex y JLex [Berex]. Existen diversas ventajas sobre el uso de estas herramientas, por ejemplo: la construcción de autómatas finitos determinísticos a partir de una especificación formal de los tokens es una tares automática, son fácilmente acoplables a otras herramientas de análisis sintáctico, entre otras.

JLex es una herramienta desarrollada en Java, la cual genera un programa en Java a partir de una especificación en el que se indican los tokens permitidos por nuestro lenguaje.



Un archivo de especificación para JLex está organizado en tres secciones, separadas por: "%%", y tiene el siguiente formato:

Código de usuario
%%
Directivas JLex
%%
Expresiones regulares

"%%" debe estar al principio de la línea. El resto de la línea conteniendo "%%" debe descartarse y no debe usarse para declaraciones o código.

2.1. Código de usuario.

En esta sección se crean las clases necesarias para nuestro analizador léxico, las cuales serán copiadas directamente al inicio del archivo de salida. Se importan los paquetes necesarios, por ejemplo: `import java_cup.runtime.Symbol;`.

2.2. Directivas JLex.

En esta sección se incluyen algunas reglas propias de JLex, por ejemplo: *%cup* para especificar el uso de CUP. También se pueden definir macros, que resuman ciertas expresiones regulares que nos serán útiles al momento de identificar tokens en la siguiente sección, y estados.

La sintaxis de una macro es:

`<nombre_macro>= <expresión_regular_válida>`

Por ejemplo, podría definirse una macro para reconocer números enteros:

```
NUMERO = [1-9][0-9]*
```

Los estados son declarados usando la directiva *%state*, y son usados para controlar cuándo reconocer ciertas expresiones regulares, por ejemplo para descartar comentarios y espacios en blanco. Sólo un estado se declara de forma implícita por JLex llamado YYINITIAL. Por default el analizador léxico obtenido iniciará en YYINITIAL.

Cada regla debe ser colocada en una nueva línea. Las reglas comienzan con una lista opcional de estados. La regla se aplica sólo si el analizador léxico está en el estado especificado. Si no se especifica un estado se usa indondicionalmente.

A continuación listo las directivas de mayor uso (a mi consideración):

- %{*código*%}, permite declarar código interno al analizador léxico, el cual será copiado al inicio de la clase de salida. Generalmente se usa para declarar variables y métodos.
- *%type*, permite cambiar el tipo de regreso del analizador léxico. Por default este tipo es Yytoken, declarado por JLex. Por ello es posible redefinir la clase Yytoken en la primera sección.
- *%eofval*{*código*%*eofval*}, permite cambiar el tipo de regreso para indicar el fin de archivo. Este valor debe ser compatible con el tipo de regreso del método yylex de la clase Yylex. El valor por default es NULL.
- *%eof*{*código*%*eof*}, permite añadir código Java el cual será ejecutado después de encontrarse un fin de archivo.
- *%init*{*código*%*init*} Permite añadir código Java el cual será copiado en el constructor de la clase Yylex.
- *%char*, *%line*, permite activar el conteo de carácteres y de líneas.
- *%eofthrow*{, *%initthrow*, *%yylexthrow*{ *código*%*yylexthrow*}, *%initthrow*}, *%eofthrow*}, permite declarar las posibles excepciones arrojadas por los respectivos métodos.
- *%class* < *nombre* >, *%function* < *nombre* >, permite cambiar el nombre de la clase del analizador léxico, por default Yylex, y el nombre de la función yylex respectivamente.

- *%public*, permite que la clase del analizador léxico generada por JLex sea una clase pública.
- *%cup*, permite activar el uso de CUP.

2.3. Expresiones regulares.

En esta sección se definen las expresiones regulares que representan los tokens de nuestro lenguaje, y qué debemos hacer una vez detectados dichos tokens. Las reglas tienen 3 partes:

```
[<estado(s)>] <expresión>{<acción>}
```

donde $< estado(s) >$ es opcional y es un listado de los posibles estados a partir de los cuales la regla debe ser aceptada. Si no se indica un estado en específico la regla se relaciona con todos. $<$ *expresión* $>$ es una expresión regular (regla). Si más de una regla coincide con el patrón se elige la regla que genera la cadena más larga. En caso de que la longitud sea la misma tiene prioridad la regla que está especificada primero en el archivo. Si ninguna regla coincide se dá un error. $<$ *acción* $>$ es la acción asociada con la regla léxica, se puede declarar cualquier código Java que se necesite. También se pueden generar transiciones entre los estados declarados. Esto es mediante una llamada a la función: `yybegin(estado)`.

Considérense los siguientes puntos para la construcción de expresiones regulares:

- El alfabeto para JLex es el conjunto de carácteres ASCII.
- Como metacarácteres tenemos: ?, ∗, +, |, (,), ∧, \$, ., [,], {, }, “, \\.
- Para concatenar usamos: |.
- Existe un conjunto de secuencias de escape (e.g. \n, \t y \r) que son reconocidas.
- El punto “.” coincide con cualquier carácter menos el de \n.
- Los metacarácteres pierden su significado cuando están entre comillas excepto \n.
- *nombre_macro* denota la expansión de una macro, donde $<$ *nombre_macro* $>$ es el nombre con que fue declarada.
- El signo ∗ representa la cerradura de Kleene.
- El signo + significa una o más repeticiones.
- El signo ? coincide con cero o una repetición de la expresión que lo precede.
- Los ( ) se usan para agrupar expresiones.
- Los [ ] denotan una clase de carácteres y coinciden con cualquiera de los carácteres encerrados.
- Si el primer carácter siguiendo a [es ∧, el conjunto está negado y la expresión coincide con cualquier carácter excepto los especificados.
- [\ − \\] coincide con un guión o una barra.

3. Herramienta CUP: generador automático de analizadores sintácticos LALR.

Para construir un compilador el segundo paso que debemos dar es el de desarrollar el analizador sintáctico de nuestro lenguaje. Para esta tarea existen diversas herramientas tales como: Bison, Yacc, Happy, JavaCC, ANTLR y CUP [HudUP].

CUP es una herramienta desarrollada en Java para crear analizadores sintácticos LALR. Genera dos clases en Java, por default sym y parser, a partir de una especificación en la que se indica una gramática formal así como también se asocian una serie de acciones a los símbolos aparecidos en cada regla gramatical. Por tanto esta herramienta cubre la fase de análisis sintáctico por el proceso de traducción dirigida por la sintaxis. También define un interfaz para acoplarse a los analizadores léxicos construidos con JLex.

La clase sym está constituida por los símbolos terminales declarados en la gramática, la cual es utilizada para hacer referencia a los mismos. La clase parser contiene al analizador sintáctico.

Un archivo de entrada para CUP consta de las siguientes 5 secciones:

1. Definición de paquete e importación de paquetes necesarios.
2. Código de usuario.
3. Declaración de símbolos terminales y no terminales.
4. Declaraciones de precedencia.
5. Definición del símbolo inicial de la gramática y las reglas de producción.

3.1. Definición de paquetes e importación de paquetes necesarios.

En este sección se incluyen las construcciones para indicar que las clases Java generadas a partir de este archivo pertenecen a un determinado paquete y/o también importar las clases Java necesarias. Por ejemplo: `import java_cup.runtime.*;`.

3.2. Código de usuario.

En esta sección se puede incluir código Java que el usuario desee incluir en el analizador sintáctico que se va a obtener con CUP. Existen varias partes del analizador sintáctico generado con CUP en donde se puede incluir código de usuario. A su vez este esta dividido en 4 partes:

- *actioncode*{: *código* :}, permite incluir código Java el cual es utilizado por las acciones especificadas en la gramática.
- *parsercode*{: *código* :}, permite incluir código Java en la clase parser. Aquí se pueden redefinir los métodos que se invocan como consecuencia de errores de sintaxis.

- *initwith*{: *código* :}, permite incluir código Java el cual será ejecutado por al traductor antes de que este pregunte por el primer token.
- *scanwith*{: *código* :}, permite indicar cómo el traductor preguntará por el siguiente token al analizador léxico.

3.3. Declaración de símbolos terminales y no terminales.

En esta sección se declaran los símbolos terminales y no terminales de la gramática que define el analizador sintáctico que deseamos producir. Tanto los símbolos no terminales como los símbolos terminales pueden, opcionalmente, tener asociado un objeto Java de una cierta clase. Por ejemplo, en el caso de un símbolo no terminal, esta clase Java puede representar los subárboles de sintaxis abstracta asociados a ese símbolo no terminal. En el caso de los símbolos terminales, el objeto Java representa el dato asociado al token. Por ejemplo, un objeto de la clase Integer que represente el valor de una constante, o un String que represente el nombre de un identificador.

La sintaxis es:

terminal [<nombre_clase>] nombre01, nombre02, ..., nombreN y
non terminal [<nombre_clase>] nombre01, nombre02, ..., nombreN

3.4. Declaraciones de precedencia.

En esta sección es posible definir niveles de precedencia y la asociatividad de símbolos terminales. Las declaraciones de precedencia de un archivo CUP consisten en una secuencia de construcciones que comienzan con la palabra clave *precedence*. A continuación, viene la declaración de asociatividad, que puede tomar los valores $left$ (el terminal se asocia por la izquierda), y *right* (el terminal se asocia por la derecha). Finalmente, la construcción termina con una lista de símbolos terminales separados por comas, seguido del símbolo ;.

La precedencia de los símbolos terminales viene definida por el orden en que aparecen las construcciones *precedence*. Los terminales que aparecen en la primera construcción *precedence* son los de menor precedencia, a continuación vienen los de la segunda construcción *precedence* y así sucesivamente, hasta llegar a la última, que define a los termianles con mayor precedencia.

3.5. Definición del símbolo inicial de la gramática y las reglas de producción.

Para definir el símbolo inicial de la gramática se utiliza: *startwith* $< no_terminal >$.

Las reglas de producción tienen esta sintaxis:

expresión ::= expresión <símbolo_terminal>expresión {: *código* :};

donde expresión son no terminales, y a la derecha está el código Java que se ejecutará al aplicarse esta producción.

Se pueden definir todas las reglas de producción que tengan a un mismo símbolo no terminal como antecedente separándolas por el símbolo |.

4. RELIPMOC: Un compilador propio.

4.1. Introducción

Hasta esta sección tenemos un panorama básico de los conceptos necesarios para poder construir un compilador; por lo que ahora mi tarea es realizar uno.

Este compilador recibirá como entrada un archivo donde se encontrará un programa escrito en mí propio lenguaje, y mostrará como salida el proceso en las fases de análisis léxico, sintáctico y semántico, así como la tabla de símbolos, código intermedio y el resultado de las operaciones especificadas.

Para ello utilizaré las herramientas abarcadas en las dos secciones anteriores. Este compilador será sencillo (con la única finalidad de mostrar el uso de las herramientas de forma conjunta) y de una sola pasada.

Para poder hacer uso de las herramientas es necesario instalar el SDK para J2SE (Java 2 Standard Edition), el cual puede descargarse de la página: `http://java.sun.com`.

Para ejecutar JLex es necesario crear un directorio que se llame "JLex" y ahí copiar el archivo fuente de JLex (Main.java) el cual puede descargarse de la página: `http://www.cs.princeton.edu/ appel/modern/java/JLex/`. Finalmente, este archivo debe ser compilado para obtener las clases Java que componen la distribución de JLex. La sintaxis para usar JLex es:

```
java JLex.Main [archivo.lex]
```

donde archivo.jlex es el nombre del archivo con una especificación JLex.

Para ejecutar CUP es necesario descargar el código fuente el cual puede obtenerse de la página: `http://www.cs.princeton.edu/ appel/modern/java/CUP/`. Finalmente, debes descomprimirlo. En caso de ser necesario compila todos los archivos fuente que se encuentren en los directorios `java_cup/` y `java_cup/runtime/`. La sintaxis para usar CUP es:

```
java java_cup.Main [opciones] <[archivo.cup]
```

donde archivo.cup es el nombre del archivo con una especificación CUP.

Al ser herramientas desarrolladas en Java, pueden modificar cualquier archivo con extensión .java con total responsabilidad (e.g. permitir que la clase llamada Yylex por default se nombre distinto, etc.), y tener distintos beneficios.

4.2. Descripción del lenguaje de entrada.

Este lenguaje tiene una sintaxis similar a la del lenguaje Java. Fue definido, por mí, para ilustrar el uso de JLex y CUP conjuntamente, basándome en el ejemplo original (el cual puedes obtener de las páginas oficiales de las herramientas), que consiste en una calculadora.

Al igual que en la mayoría de los lenguajes de programación modernos, los espacios en blanco, tabulaciones, y saltos de línea no afectan el significado del programa. Sin embargo, se suelen respetar convenciones en la escritura para facilitar la lectura del código.

4.2.1. Componentes léxicos de RELIPMOC.

InputElement	⟶	*WhiteSpace* | *Comment* | *Token*
WhiteSpace	⟶	*space* | `\t` | `\r` | `\n` | `\f`
Comment	⟶	`/** any string */`
Token	⟶	*Identifier* | *Keyword* | *Literal* | *Separator* | *Operator*
Identifier	⟶	*Letter* | *Identifier Letter* | *Identifier Digit*
Letter	⟶	`a	b	...	z	A	B	...	Z`
Digit	⟶	`0	1	...	9`
Keyword	⟶	`class	int	void`
Literal	⟶	*Integer*
Integer	⟶	*Digit* | *Integer Digit*
Separator	⟶	`(	)	{	}	;`
Operator	⟶	`=	+	-	*`

Los tokens que pueden aparecer se pueden dividir en cinco clases: *Identifier, Keyword, Literal, Separator y Operator*. El resto *WhiteSpace y Comment* es ignorado durante el análisis léxico.

4.2.2. Sintaxis de RELIPMOC.

$$
\begin{array}{rcl}
\textit{program} & \longrightarrow & \textit{class_declaration} \\
\textit{class_declaration} & \longrightarrow & \texttt{class}\ \textit{Identifier class_body} \\
\textit{class_body} & \longrightarrow & \{\ \textit{method_declaration}\ \} \\
\textit{method_declaration} & \longrightarrow & \texttt{void}\ \textit{Identifier}\ (\ )\ \textit{method_body} \\
\textit{method_body} & \longrightarrow & \{\ \textit{variable_declarations block_statements}\ \} \\
\textit{variable_declarations} & \longrightarrow & \textit{variable_declaration} \\
 & | & \textit{variable_declarations variable_declaration} \\
\textit{variable_declaration} & \longrightarrow & \textit{type variable_declarator}\ ; \\
\textit{type} & \longrightarrow & \textit{numeric_type} \\
\textit{numeric_type} & \longrightarrow & \texttt{int} \\
\textit{variable_declarator} & \longrightarrow & \textit{Identifier} = \textit{Literal} \\
\textit{block_statements} & \longrightarrow & \textit{statements} \\
 & | & \textit{block_statements statements} \\
\textit{statements} & \longrightarrow & \textit{statement}\ ; \\
\textit{statement} & \longrightarrow & \textit{expression} \\
\textit{expression} & \longrightarrow & \textit{Literal} \\
 & | & \textit{Identifier} \\
 & | & \textit{expression} * \textit{expression} \\
 & | & \textit{expression} + \textit{expression} \\
 & | & \textit{expression} - \textit{expression} \\
 & | & (\ \textit{expression}\ )
\end{array}
$$

4.3. Descripción de salida.

La salida mostrada será el proceso en las fases de análisis léxico, sintáctico y semántico, así como la tabla de símbolos, código intermedio (código de tres direcciones sin optimización -estilo Mary Carmen-) y el resultado de las operaciones especificadas (recordemos que está basado en una calculadora).

En este caso, el código de tres direcciones consiste en crear asignaciones y resultados de operaciones binarias tales como suma, resta y multiplicación, en variables temporales. Así como su reutilización para operaciones anidadas.

4.4. Implementación.

Esta sección se encuentra descrita en dos archivos de especificación, specification.lex y specificatio.cup. Básicamente, en estos archivos se reflejan los componentes léxicos y la gramática que compone al lenguaje, así como código Java para crear código de tres direcciones, manejo de errores, etc.

Además es desarrollaron 3 clases: *ElementSymbolTable*, *SymbolTable* y *Main*. La primera describe los atributos, que en lo personal, interesantes para la toma de decisiones en la fase de análisis sintáctico y semántico. En la segunda clase se describe la estructura en la cual serán almacenados temporalmente los tokens adecuados junto con sus atributos. Finalmente, en la

última clase (clase principal) se realizan las llamadas a los objetos, constructores, métodos y atributos adecuados, para la visualización de un completo proceso llevado a cabo por el compilador.

Para ver este compilador ejecutarse, es necesario obtener el archivo relipmoc.tgz, el cual puede descargarse del URL: `http://www.dynamics.unam.edu/ mtrejo/cursos/compiladores/relipmoc.tgz`, descomprimirlo, y ejecutar `make`, `make execute` (para un único ejemplo) y finalmente `make clean`. Este archivo contiene un directorio llamado Ejemplos, donde se encuentran 3 únicos ejemplos.

5. Proyecto.

Como uno de los objetivos principales de este curso, así como la intención de realizar este documento, es que los alumnos diseñen e implementen un compilador. Este pequeño proyecto estará dividido en tres entregas:

1. Análisis léxico, manejo de errores (nivel análisis léxico), componentes léxicos de su lenguaje y bosquejo de la gramática de su lenguaje.
2. Análisis sintáctico, manejo de errores (nivel análisis sintáctico), tabla de símbolos y gramática de su lenguaje.
3. Análisis semántico, manejo de errores (nivel análisis semántico), obtención de código intermedio.

Para cada una de estas entregas se debe incluir:

- El archivo(s) de especificación que se le ingresa a las herramientas generadores de analizadores.
- Un programa que reciba como entrada un archivo con un programa en su propio lenguaje y verifique si es léxica, sintácticamente, semánticamente correcto, así como la obtención de resultados necesarios en cada entrega.
- Ejemplos de programas, escritos en su propio lenguaje, correctos e incorrectos.

Ustedes elegirán el lenguaje fuente y el lenguaje objeto sobre el cual trabajarán, siempre y cuando estos tengan una estrecha relación para poder levar a cabo la tarea del compilador. Por lo que no, necesariamente, harán uso de las herramientas expuestas en este documento, sin embargo deberán hacer entrega de los puntos antes estipulados. El uso de herramientas, estrictamente para apoyo, es totalmente libre. También elegirán la gramática para su lenguaje, teniendo como mínima la presentada en el libro *Compilers* de Aho, Sethi y Ullman.

El compilador aquí descrito puede ser usado con total libertad como base del de ustedes, a excepción del código de tres direcciones. Esta fase deberá ser ampliamente extendida por ustedes.

Referencias

[App98] Andrew W. Appel. *Modern compiler implementation in Java.* Cambridge University Press, 1998.

[ASU86] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers; principles, techniques and tools.* Addison-Wesley Publishing Company, 1986.

[Berex] Elliot Berk. *JLex: A lexical analyzer generator for Java. JLex user manual.* 1997. http://www.cs.princeton.edu/∼appel/modern/java/JLex/.

[HudUP] Scott E. Hudson. *CUP user's manual*. 1999. http://www.cs.princeton.edu/∼appel/modern/java/CUP/.