Introducción

El siguiente informe detalla el desarrollo de un analizador sintáctico y léxico de un lenguaje diseñado con fines solo educativos. Se informará sobre aspectos generales de cada componente, decisiones importantes tomadas e implementaciones adoptadas.

Desarrollo

Temas asignados

- **Punteros a función**
- **Uinteger**
- **Do until**
- **Comentarios 1 Línea**
- **Cadenas Multilínea**

Analizador Léxico

Aspectos generales

Debido a que los lenguajes a escanear son gramáticas regulares, la solución consistirá en el modelado de una máquina finita de estados determinística. La cantidad de estados que posea la misma va a depender de la granularidad con que se agrupen los distintos tipos de caracteres de entrada y de las acciones semánticas necesarias. Éste aspecto se detallará posteriormente.

A partir de las condiciones léxicas brindadas en lenguaje natural por el enunciado del Trabajo Práctico 1, se ha podido desarrollar el autómata y la matriz de transiciones de estados presentes en el anexo. Para un mejor entendimiento se detallarán las acciones semánticas desarrolladas para la solución propuesta.

- **InputBuffer**: lee próximo carácter y lo almacena en buffer
- **ConsumeInput**: descarta próximo carácter.
- **ValidateX**: tipo de acciones semánticas que contienen el chequeo del tipo X.
- **Validate_symbol_unget**: aparte de validación, retrocede en una posición en el análisis de la entrada.
- **Invalid_action**: ocurre ante un error en el reconocimiento del token. Por ejemplo, el símbolo *:* seguido de cualquier carácter excepto el =.

Como el sistema estará basado en la matriz de transición de estados, es necesario identificar componentes bien definidos y con funcionalidades específicas. La matriz deberá contener estáticamente toda la información de las transiciones y las acciones semánticas a seguir, y cada acción semántica deberá tener una funcionalidad no dependiente de los estados anteriores ni posteriores. Estos dos aspectos vitales ya se tuvieron en cuenta al desarrollar el autómata.

Implementación

El próximo paso a realizar una vez diseñada la matriz de transiciones es la definición de un lenguaje de programación. La elección del mismo ya definirá los atributos de calidad que se priorizarán en el sistema. El sistema se ha implementado en Java, logrando así un fácil desarrollo y seguimiento del código fuente, aunque comprometiendo a la eficiencia del programa.

Como se puede observar en el código fuente, se han modelado los siguientes paquetes de clases:

- **SemanticAction**: superclase de las que heredan dos tipos de acciones semánticas: las de buffer y las de validación. Las primeras realizan el tratamiento del buffer; mientras que las de validación se encargan también de chequear el correcto estado del buffer e informar de errores, también se encargan de la generación de la entrada correspondiente a la tabla de símbolos.

- **SymbolTable**: paquete que contiene no solo la tabla, sino también los distintos tipos de entrada que tienen éstos. La tabla de símbolos cargará al inicializarse todas las palabras reservadas y comparadores existentes, y luego será la encargada de la generación de tokens de referencia hacia estas entradas. Cabe destacar que la tabla de símbolos es la encargada de chequear si un identificador es o no una palabra reservada.

- **Token**: como se ha optado por ingresar todos los elementos a la tabla de símbolos, solo existe un tipo de token, el cual es el de referencia a la tabla. Un token contiene la dupla <typo,referencia> y son generados por la tabla de símbolos.

- **Lexer**: es la clase que posee la matriz de estados y la controla. Se le deben pasar por parámetros iniciales datos como la tabla de símbolos, el controlador de eventos y la entrada, y mediante el método de acceso getToken(), éste analiza la entrada y devuelve un token.

- **Event**: se ha generado una interfaz de eventos, el cual puede recibir eventos de tipos definidos (error, warning, nueva regla, nuevo token, cambio de tabla de símbolos), para ser implementado por la interfaz gráfica.

Aclaraciones

Las acciones semánticas de validación tienen la capacidad de generar errores de tipos críticos o no. Los errores críticos eliminarán el buffer de entrada y reiniciará la máquina de estados, mientras que los no críticos sólo sirven como alertas.

Las validaciones de los strings también realiza el borrado de los saltos de línea utilizando expresiones regulares. Se ha optado por esta opción para evitar el innecesario crecimiento en tamaño y complejidad funcional de la matriz de estados.

En el caso de las constantes *uinteger*, se ha optado por utilizar como palabra reservada *UI* y delegar la correcta enunciación de las constantes al analizador sintáctico.

La tabla de símbolos se ha implementado utilizando un Vector. Éste es el cuello de botella del analizador léxico, ya que las búsquedas tendrán complejidades del orden de la cantidad de entradas de la tabla. Una mejora a realizar es el cambio de estructura por una estructura de acceso veloz, como un árbol. La implementación de un Hash no se ha considerado debido a la dispersión que se obtienen en las referencias.

Las referencias de los tokens son indirectas, es decir contienen claves de enteros para referenciar entradas en la tabla de símbolos. Por otro lado, los strings se almacenan con las comillas, simplificando así la diferenciación entre palabras reservadas y strings.

En un principio se decidió optar por utilizar ANSI C, buscando así un sistema eficiente pero con un costo alto de programación. La elección de un lenguaje no orientado a objetos para el desarrollo del analizador léxico no mostraba grandes desventajas, pero lamentablemente esta implementación se tuvo que abandonar debido al alto costo que traía la creación de estructuras eficientes para el manejo de la tabla de símbolos.

Analizador sintáctico

El desarrollo de la gramática correspondiente para el lenguaje analizado no ha constado de grandes dificultades una vez que se tuvo en claro los componentes de la misma. En un principio se desarrolló la misma sin considerar posibles errores sintácticos, la cual no produjo dificultades exceptuando un problema shift-reduce en cuanto a los IF anidados con ELSE. Éste es un problema conocido como "dangling else" o "if-else ambiguity", y la solución más sencilla consta en forzar al Yacc a realizar el shift. De todas formas, como es requisito una gramática sin ambigüedades se ha optado por la solución que consta en generar dos tipos de sentencias llamadas sentencias abiertas o cerradas. Ésta solución logra unificar los else con los if externos logrando una gramática determinística.

La gramática básica realizada es de recursividad a la izquierda y se podrá observar en el anexo. A continuación se detallarán los símbolos no terminales utilizados que pueden llegar a generar problemas de interpretación.

- **Input**: Entrada de la gramática.
- **Open sentence**: unifica inicialmente con IF en caso de sentencias que pueden generar ambigüedad if-else, y con IF sin ELSE.

- **Closed sentence**: unifica con sentencias de ejecución, declaración y bloques. También unifica con sentencias IF que poseen un ELSE. Tanto sentencias abiertas o cerradas permiten loops con sentencias de su mismo tipo dentro de ellas.

- **Execute**: sentencias de ejecución (print, asignación, función).

- **Asignment**: sentencias de asignación.

Implementación

En el archivo fuente *grammar.y* se podrá observar también los errores sintácticos considerados y las acciones que realiza el parser al llegar a estos. Los anuncios de errores o nuevas reglas se hacen mediante una interfaz Java, siguiendo la idea utilizada en el analizador léxico.

El programa utilizado para la generación del código presente en el paquete parser fue la de BYacc/Java, el cual utiliza parsing ascendente predictivo.

Una vez compilado la gramática presente en *grammar.y*, se ha añadido las clases resultantes al paquete Parser. La clase parser es el que a través de la función interna yylex le pide un token nuevo al analizador léxico. Esto significa que la estructura que tiene el compilador es con control del parser.

Cabe aclarar que aunque el sistema se encuentra preparado para fácilmente implementar una interfaz gráfica, se ha optado por no hacerlo debido a que el objetivo del trabajo es de desarrollo de un compilador, no de un IDE. Aclaraciones sobre uso del ejecutable se encuentran en la sección de entregables.

Aclaraciones

Los errores reconocidos por la gramática son principalmente de omisión de símbolos. También se han añadido algunos de control de bloques y paréntesis, siempre evitando generar una gramática ambigua. Cada error descubierto por la gramática generará un evento de error.

La ejecución del parser devuelve como eventos información sobre nuevas reglas o errores. Debido a que la generación de eventos se genera al unificar una sentencia por completo, estos se anuncian al finalizar la regla. Es por esto que el anuncio de una nueva regla viene asociado con el número de línea de finalización de ésta.

Entregables

Junto a este informe se adjunta un CD con los siguientes elementos:

- **Código fuente:** Realizado en java, se encuentra dentro de la carpeta src.

- **Gramática**: archivo *grammar.y*

- **Copia de informe**

- **Ejecutable**: con el nombre \emph{compiler.jar}. El modo de uso es mediante consola, utilizando como único parámetro la dirección del archivo que contiene el código a compilar. El mismo devolverá por consola los tokens y reglas reconocidas. También informará errores y warnings.