DISEÑO DE COMPILADORES : TP1 Y TP2

INFORME DE IMPLEMENTACIÓN

Grupo 20

Nistal, Nicolás

Petruccelli, Augusto

Piolanti, Inti

2013

INTRODUCCIÓN

El presente informe servirá de reseña que abarcará todos los detalles del desarrollo de los trabajo práctico 1 y 2 para Diseño de Compiladores.

Esta primera parte del diseño del compilador comprende el diseño e implementación de un Analizador Léxico y una gramática que se correspondan con el lenguaje asignado por la cátedra.

Adicionalmente a la implementación de la gramática y el analizador léxico, se debe generar el código a través de una herramienta que provee YACC, que, contando con los dos primeros componentes mencionados, generará una salida interpretable.

OBJETIVOS

Para esta instancia del trabajo práctico, se debía implementar una gramática mediante una sintaxis similar a la de BNF, particularmente, la especificada para YACC.

La estructura del programa a reconocer como correcto por el analizador sintáctico debía contener los siguientes elementos:

FUNCIONES

Las funciones deben corresponderse con la siguiente especificación:

```
function <nombre_de_funcion>(<parámetros>)
begin
   Sentencias declarativas (*)
   Sentencias ejecutables de la función
End
```

- Los parámetros deberán incluir el tipo y los nombres separados por coma (‘**,**’)
  <tipo> <lista_de_parámetros>
- Las sentencias ejecutables pueden incluir una, varias o ninguna sentencia **return**(<expresión>)
- Una función se invoca utilizando su nombre seguido de los parámetros entre paréntesis:
  <nombre_de_función>(<lista_de_parámetros>)

VARIABLES

Hay un solo tipo para las variables:

- **unsigned long**: Constantes enteras con valores entre 0 y 2^{32}-1

La palabra reservada **ulong** servirá para identificar el tipo de variable *unsigned long*.

COMENTARIOS

Los comentarios comenzarán con " **(*** " y terminarán con "***)**". Los comentarios podrán tener más de una línea.

CADENAS DE CARACTERES

Las cadenas de caracteres comenzarán y terminarán con " **'** ". Estas cadenas no podrán tener más de una línea.

Utilizando YACC, se deberá generar el código a partir de la gramática, que hará las veces de analizador sintáctico. Este analizador sintáctico deberá ser proveído de tokens para ir formando las sentencias y reconocerlas como correctas. Estos tokens mencionados, deberán ser extraídos uno a uno de un analizador léxico.

Es responsabilidad del analizador léxico reconocer los diferentes tokens de una cadena de caracteres, y ofrecer una interfaz para que éstos puedan ser consumidos uno a uno por una clase cliente, que en nuestro caso será el analizador sintáctico.

El código fuente a analizar deberá ser levantado de un archivo, y trabajar sobre él en memoria primaria.

ASUNCIONES

Los requisitos expresados en el trabajo práctico pueden dejar a veces detalles sin mencionar. En este contexto, se mencionarán aquellas asunciones que fueron hechas y consideradas a la hora de la implementación.

ASIGNACIÓN

En la especificación de funcionalidades para la instancia del trabajo a tratar, no había mención alguna que alentara la consideración de sentencias de asignación. Sin embargo, como se trata de una de las funciones más comunes en un lenguaje de programación, fue tomada en cuenta y se la agregó al conjunto de reglas mediante la siguiente especificación:

```
<Asignación> :== ID '=' CTE

<Asignación> :== ID '=' ID

<Asignación> :== ID '=' <Expresión>
```

ORDEN DE LAS SENTENCIAS EN SENTENCIA *FOR*

Cuando se detalla la manera en la que se deben manejar las sentencias en el programa, se dice que:

Programa:

- Programa constituido por sentencias declarativas **seguidas de** sentencias ejecutables.

Esto mismo se considera en el cuerpo de las funciones. Sin embargo, en el momento de definir la estructura del cuerpo de la iteración *for,* sólo se menciona que contiene un bloque de sentencias.

Con el objetivo de mantener la cohesión entre los distintos bloques de sentencias, se consideró que en el cuerpo de un *for,* siempre irán **primero** las sentencias declarativas, y **segundo** las sentencias de ejecución.

DETALLES DE IMPLEMENTACIÓN

Se escribió una gramática siguiendo la sintaxis especificada para YACC, según las indicaciones para la instancia asignada del trabajo práctico.

El analizador léxico fue implementado en Java, siguiendo el paradigma de la programación orientada a objetos.

DISEÑO DE CLASES

Se denotan las siguientes familias de clases con sus respectivas responsabilidades

LEXICALANALIZER

La clase *LexicalAnalizer,* es, como su nombre lo indica, el analizador léxico. Una instancia de esta clase se construye proveyéndola de una ruta al archivo de texto contenedor del código fuente a compilar.

Esta clase será utilizada para obtener uno a uno los tokens reconocidos en el texto. El método *getToken* de esta clase es el que retorna un token al ser invocado.

TOKEN

Cada uno de los objetos de retorno del método *getToken* del *LexicalAnalyzer*, será un Token. Los objetos de este tipo serán utilizados por el analizador sintáctico para su posterior análisis. Es por esto que esta clase agrupa los valores del literal que lo compone, y la línea en que fue hallado en el archivo.

ISEMANTICACTION

ISemanticAction supone una familia de clases que compondrán, en conjunto, a todas las acciones semánticas. Cada transición entre estados puede tener asociadas acciones semánticas a ser ejecutadas, por lo que, todas estas, tendrán como comportamiento similar, la ejecución de una acción particular. Es el método abstracto *performAction*, el que implementará cada clase concreta de ISemanticAction.

ERRORHANDLER

Utilizando el patrón Singleton, *errorHandler,* constituye un log centralizado, que puede ser accedido en distintos momentos de la ejecución, y permite agregar errores, y obtenerlos. El tipo de objetos que contiene *errorHandler*, son de tipo *Error,* estos, a su vez, describen, obviamente, errores, con sus respectivos tipos y mensajes.

SYMBOLTABLE

SymbolTable es la clase diseñada para representar la tabla de símbolos. Dada el valor literal de un token, una instancia de *SymbolTable*, retornará un *SymbolElement.* Para poder llevar a cabo esto, internamente está constituida por una estructura de hash que usa valores de String como llave. El método *identify* es el que se ocupa de retornar el elemento correspondiente a un valor literal.

Tiene sentido que en todo momento durante la ejecución exista sólo una instancia de la tabla de símbolos, por esta razón, se usó el patrón *Singleton.*

TRANSITIONMATRIX

Esta clase es utilizada para representar la matriz de transición de estados. Su función es: para un estado, y un carácter, debe retornar el siguiente estado al que debe irse, y, opcionalmente, un conjunto de acciones semánticas a ser ejecutadas. Estos objetos de retorno están agrupados en la clase *TransitionCell*.

Para permitir el almacenamiento y guardado, se utilizó una estructura compuesta por un arreglo primitivo, cuyos campos representan cada fila de la matriz. Para poder hacer esto, cada campo del arreglo es una estructura de Hash, que utiliza cada carácter recibido como llave para acceder a una *TransitionCell* determinada.

CHARACTERINTERPRETER

Dado un carácter de entrada, se lo filtra y retorna el carácter apropiado para poder buscarlo en la matriz de transición de estados.

DIAGRAMA DE TRANSICIÓN DE ESTADOS

Por la extensión

automata_transicion.pdf

MATRIZ DE TRANSICIÓN DE ESTADOS

transitionMatrix.pdf

ACCIONES SEMÁNTICAS

INICIALIZAR TOKEN (AS1)

Inicializa el token, es decir, genera uno nuevo para avanzar desde el estado inicial con el proceso de reconocimiento de caracteres. Esta acción semántica fue implementada en la clase *TokenInitializer.*

AÑADIR CARÁCTER (AS2)

Toma el último carácter reconocido, y lo añade al valor literal del token en reconocimiento. Esta acción semántica fue implementada en la clase *CharacterAdder.*

ACTUALIZAR TABLA DE SÍMBOLOS (AS3)

Revisa la tabla de símbolos en busca el último token reconocido, en caso de que no encontrarlo, lo agrega. Esta acción semántica fue implementada en la clase *SymbolTableHandler.*

CHEQUEAR RANGO (AS4)

Chequea el valor en caso de reconocer un valor literal de un *unsigned long*, y lo compara frente a un rango definido. En caso de no encajar en este, se lo trunca al extremo más próximo. Esta acción semántica fue implementada en la clase *RangeChecker.*

RETROCEDER ITERADOR (AS5)

Vuelve el iterador sobre el texto una posición hacia atrás, para los casos en que se haya consumido un carácter que no debería. Esta acción semántica fue implementada en la clase *CharacterReturner.*

TRUNCAR IDENTIFICADOR (AS6)

En caso de haber consumido un identificador, chequea si su largo no es mayor a determinado límite, en caso contrario, lo trunca al largo permitido. Esta acción semántica fue implementada en la clase *CharacterTruncator.*

CERRAR CADENA DE CARACTERES (AS9)

En caso de reconocer un salto de línea durante el consumo de una cadena de caracteres, se la cierra automáticamente añadiendo la comilla faltante al final, y notificando del conflicto. Esta acción semántica fue implementada en la clase *SingleQuoteAdder.*

INCREMENTAR CONTADOR DE LÍNEAS (AS10)

Incrementa el contador de líneas de código. Esta acción semántica fue implementada en la clase *LineCounter.*

DEFINIR TIPO DE TOKEN (AS11, AS12, AS13, AS14)

Cada token es enviado al analizador sintáctico con su respectivo tipo reconocido por el analizador léxico. Esta es en realidad un conjunto de acciones semánticas, que se ocupan, dependiendo del estado y de la transición, de definir el tipo correcto a cada token a enviar. *CharchainTokenSetter, IDTokenSetter, CteTokenSetter,* y *LiteralTokenSetter*, son las clases que representan a este conjunto de acciones semánticas.

GRAMÁTICA

Se definió una gramática libre de contexto mediante la sintaxis de YACC, para servir de estructura a seguir al analizador sintáctico.

Durante la construcción de la gramática, se siguieron las buenas prácticas en las definiciones de reglas. Una de ellas es respetar siempre la recursividad al mismo lado, en este caso, se respetaron siempre las recursividades a izquierda. Esta manera de definir la gramática lleva a reducir en gran medida los problemas que pueden surgir.

LISTA DE NO TERMINALES

Fueron definidos los siguientes no terminales en la construcción de la gramática:

- Program: Raíz del programa, un programa es aceptado si se llega a esta regla
- DeclarationList: Lista de declaraciones de variables y funciones
- Declaration: Declaración singular de variable o función
- VarDeclarationList: Lista de declaraciones de variables
- VarDeclaration: Declaración singular de una variable
- VarList: Lista de identificadores
- FunctionDeclaration: Declaración singular de una función
- Executions: Lista de sentencias de ejecución sin sentencias de return
- ExecutionWReturnList: Lista de sentencias de ejecución incluídas sentencias de return
- ExecutionWReturn: Sentencia de ejecución incluyendo cláusula return
- Execution: Sentencia única de ejecución sin incluir cláusulas return
- Ret: Sentencia de return
- Print: Sentencia de print, impresión por pantalla
- FunctionExecution: Sentencia de llamada a función
- Iteration: Sentencia de iteración For
- Selection: Sentencia condicional If
- Condition: Sentencia de comparación de expresiones
- Block: Bloque de declaraciones y ejecuciones
- Assign: Sentencia de asignación
- Expression: Sentencias de expresión de suma y resta
- Term: Sentencia de expresión de mayor precedencia, para división y producto
- Factor: Constante o Id

MANEJO DE ERRORES

Se pueden diferenciar dos tipos de errores principales, estos son, los errores léxicos, y los sintácticos.

ERRORES LÉXICOS

- Warning: Informe de que un literal fue truncado por exceder el límite permitido
- Warning: Informe de que un *unsigned long*, no correspondía al rango permitido y fue redondeado.
- Warning: Informe de la falta de una comilla de cierre en una cadena de caracteres

ERRORES SINTÁCTICOS

- Warning: Informe de falta de paréntesis izquierdo en sentencias Print, Return, Function, For, If.
- Warning: Informe de falta de paréntesis derecho en sentencias Print, Return, Function, For, If.
- Fatal: Informe de la falta de begin o end en un bloque de sentencias.
- Fatal: Informe de la falta de un token then luego de la condición de un if
- Notification: Notificación de la creación de: variables, funciones, sentencias return, sentencias print, sentencias de llamadas a función, if, for, y asignaciones.

CONCLUSIÓN

La implementación de un compilador desde las mismas bases es esencial para la correcta compresión de los diferentes procesos involucrados. En esta experiencia, quedaron detalles de implementación a mejorar, como los comentarios en Javadoc en los métodos, y algunas responsabilidades que hubieran sido mejor extraer, y que de hecho lo serán para la próxima entrega. Por ejemplo, no debería ser responsabilidad del analizador léxico el parsing del archivo.

Como un comentario al margen, hubiera sido interesante tener la posibilidad de usar ANTLR para la construcción de la gramática, ya que está más mantenido y es más actual que YACC, y posee herramientas como ANTLRWorks 2, que facilita mucho la construcción de gramáticas, permitiendo analizar cadenas de texto *on the fly*, y poder ver el árbol de parsing generado. Incluso, en caso de producirse un error, puede ser revisado sobre el árbol de parsing, lo que mejora en gran medida el sentido de la compresión de los errores en las reglas.