**COMPILADORES**

**ANALIZADOR SINTACTICO DEL LENGUAJE “MATH UQ”**

**INFORME FASE 2**

**UNIVERSIDAD DEL QUINDIO**

**FACULTAD DE INGENIERIAS**

**INGENIERIA DE SISTEMAS NOCTURNO**

**ARMENIA QUINDIO**

**2018**

**COMPILADORES**

**ANALIZADOR SINTACTICO DEL LENGUAJE “MATH UQ”**

**INFORME FASE 2**

**Presentado a**

**RUTH STELLA CRUZ CETINA**

**Por**

**GUSTAVO ADOLFO SALGADO OCAMPO**

**CRISTIAN DAVID TORO GARCIA**

**JORGE MARIO BURBANO MARTINEZ**

**UNIVERSIDAD DEL QUINDIO**

**FACULTAD DE INGENIERIAS**

**INGENIERIA DE SISTEMAS NOCTURNO**

**ARMENIA QUINDIO**

**2018**

**INTRODUCCIÓN**

El análisis sintáctico es un análisis a nivel de sentencias, y es mucho más complejo que el análisis léxico. Su función es tomar el programa fuente en forma de tokens, que recibe del analizador léxico, y determinar la estructura de las sentencias del programa. Este proceso es similar a determinar la estructura de una frase en castellano, determinando quien es el sujeto, predicado, el verbo y los complementos. El análisis sintáctico agrupa a los tokens en clases sintácticas (denominadas no terminales en la definición de la gramática), tales como expresiones, procedimientos, etc.

El analizador sintáctico o parser obtiene un árbol sintáctico (u otra estructura equivalente) en la cual las hojas son los tokens, y cualquier nodo que no sea una hoja, representa un tipo de clase sintáctica (operaciones). Se evaluaran en primer lugar las constantes, variables y expresiones entre paréntesis.

Los árboles sintácticos se construyen con un conjunto de reglas conocidas como gramática, y que definen con total precisión el lenguaje fuente.

Al proceso de reconocer la estructura del lenguaje fuente se conoce con el nombre de análisis sintáctico (parsing). Hay distintas clases de analizadores o reconocedores sintácticos, pero en general se clasifican en 2 grandes grupos: A.S. Ascendentes y A.S. Descendentes.

La principal tarea del analizador sintáctico no es comprobar que la sintaxis del programa fuente sea correcta, sino construir una representación interna de ese programa y en el caso en que sea un programa incorrecto, dar un mensaje de

Error. Para ello, el analizador sintáctico (A.S.) comprueba que el orden en que el analizador léxico le va entregando los tokens es válido. Si esto es así significará que la sucesión de símbolos que representan dichos tokens puede ser generada por la gramática correspondiente al lenguaje del código fuente.

**TOKENS Y EXPRESIONES REGULARES**

***Convenciones***

**L** = Letras mayúsculas y minúsculas.

**D** = Digito (1,2,3,4,5,6,7,8,9,0).

|  |  |  |
| --- | --- | --- |
| **TOKEN** | **LEXEMA** | **EXPRESIÓN REGULAR** |
| Palabra Reservada **Leer** | Leer | L.e.e.r |
| Palabra Reservada **Escribir** | Escribir | E.s.c.r.i.b.i.r |
| Palabra Reservada **Si** | Si | S.i |
| Palabra Reservada **Sino** | Sino | S.i.n.o |
| Palabra Reservada **Finsi** | Finsi | F.i.n.s.i |
| Palabra Reservada **Entonces** | Entonces | E.n.t.o.n.c.e.s |
| Palabra Reservada **Numero** | Numero | N.u.m.e.r.o |
| Palabra Reservada **Cadena** | Cadena | C.a.d.e.n.a |
| Asignación | = | "=" |
| Entero | 1 | "1" |
| Concatenación | @ | "@" |
| Separador | , | "," |
| Identificador | a | L.(DuLu\_)\* |
| Fin de línea | . | “.” |
| Abrir corchete | [ | “[“ |
| Cerrar corchete | ] | “]” |
| Abrir llave | { | “{“ |
| Cerrar llave | } | “}” |
| Suma | + | “+” |
| Resta | - | “-“ |
| Multiplicación | \* | “\*” |
| División | / | “/” |
| Modulo | % | “%” |
| Lógicos | &, | | "&"|"|" |
| Relacionales | <, >, <=, >=, <>, ==, != | "<"|">"|" >="|"<="|"<>"|"=="|"!=" |
| Comentario | ##este programa permite## | "##".(COS)\*."##" |
| Cadena | “ por favor ingrese texto“ | “\””(~[COS])\*”\”” |

**\_** = Guion de piso, urdescore, raya baja.

**COS** = Cualquier otro Símbolo**.**

**BNF**

**(Backus normal form)**

<mathuq>::=<instrucción> [<instrucción>]

<instruccion>::=<definición\_var>|<imprimir>|<asignación\_var>|<condición>

< definicion\_var>::=<tipo\_dato><identificador> [“,”<listado\_identificadores>] “.”

<imprimir>::=escribir “[“ <imprimible>”]” “.”

<imprimible>::=<identificador>|<entero>|<concatenación>

<asignación\_var>::=<identificador>”=”<tipo\_asignacion> “.”

<tipo\_asignacion>::=<leer>|<concatenación>|<operación\_mat>

<condición>::=Si<grupo\_condicions>Entonces<listado\_sentencias>[Sino<listado\_sentencias>] Finsi

<grupo\_condiciones>::=<condicionador><operador\_relacional><condicionador>

<condicionador>::=<identificador>|<entero>

<leer>::= leer “[“ “]”

<operación\_mat>::=<termino [<operación\_adicion><operación\_mat]

<termino>::=<factor>[<operación\_multpl><termino>]

<factor>::=<entero>|<identificador>| “[“operación\_mat>”]”

<operadores\_adicion>::= “+”| “-“

<operadores\_multipl>::=”\*”| “/” | “%”

<operador\_relacional>::= “<” | “>” | “>=” | “<=” | “==” | “!=”

<tipo\_dato>::= numero | cadena

<listado\_identificadores>::=<identificador>[ “,” <listado\_identificadores>]

<identificador>::=

<cadena>::= “ “ “[<listado\_ascii>]” “ “

<listado\_ascii>::= <ascii>[<listado\_ascii>]

<entero>::=<digito>[<entero>]

<digito>::= “0” | “1” | “2” | “3” | …

**FUNCIONALIDAD DEL ANALIZADOR SINTACTICO**

|  |
| --- |
| En el modelo del compilador, el analizador sintáctico obtiene una cadena de componentes léxicos del analizador léxico, y comprueba si la cadena puede ser generada por la gramática del programa fuente.  En nuestro  modelo de compilador, el analizador sintáctico obtiene una cadena de tokens del analizador léxico, como se muestra en la figura, y verifica que la cadena de nombres de los tokens pueda generarse mediante la gramática para el lenguaje fuente.  Esperamos que el analizador sintáctico reporte cualquier error sintáctico en forma inteligible y que se recupere de los errores que ocurren con frecuencia para seguir procesando el resto del programa. De manera conceptual, para los programas bien formados, el analizador sintáctico construye un árbol de análisis sintáctico y lo pasa al resto del compilador para que lo siga procesando.  De hecho, el árbol de análisis sintáctico no necesita construirse en forma explícita, y a que las acciones de comprobación y traducción pueden intercalarse con el análisis sintáctico, como veremos más adelante. Por ende, el analizador sintáctico y el resto de la interfaz de usuario podrían implementarse sin problemas mediante un solo módulo.  *Grafica posicionamiento del analizador sintáctico dentro del compilador* |
| Posición del analizador sintáctico en el modelo del compilador |

El analizador sintáctico obtiene una cadena de componentes léxicos del analizador léxico, y comprueba si la cadena puede ser generada por la gramática del programa fuente.

Tipos generales de analizadores sintácticos para gramáticas:

a) Análisis sintáctico descendente. Construye árboles de análisis sintáctico desde arriba (raíz) hacia abajo (hojas). El análisis se realiza de lo general a lo particular.

b) Análisis sintáctico ascendente. Construyen árboles de análisis sintáctico comenzando en las hojas y suben hacia la raíz. El análisis se realiza de lo particular a lo general.

En ambos casos, se examina la entrada al analizador léxico de izquierda a derecha, un símbolo a la vez.

GRAMATICAS INDEPENDIENTES DE CONTEXTO

Una gramática describe de forma natural la estructura jerárquica de muchas construcciones de los lenguajes de programación. Las gramáticas libres de contexto permiten describir la mayoría de los lenguajes de programación, de hecho, la síntaxis de la mayoría de lenguajes de programación está definida mediante gramáticas libres de contexto. Por otro lado, estas gramáticas son suficientemente simples como para permitir el diseño de eficientes algoritmos de análisis sintáctico que, para una cadena de caracteres dada determinen como puede ser generada desde la gramática,Consta de:

* TERMINALES. Símbolos básicos con que se forman las cadenas. Para un lenguaje de programación, cada palabra clave/reservada es un terminal.
* NO TERMINALES. Son variables sintácticas que denotan conjuntos de cadenas (identificadotes o variables). Los no terminales definen conjuntos de cadenas que ayudan a definir el lenguaje generado por la gramática. Imponen una estructura jerárquica sobre el lenguaje que es útil tanto para el análisis sintáctico como para la traducción.
* UN SÍMBOLO INICIAL. En una gramática, es un no terminal que representa un conjunto de cadenas.
* PRODUCCIONES. Especifican cómo se pueden combinar los terminales y no terminales para formar cadenas. Cada producción consta de un No terminal (símbolo inicial), seguido por una flecha o símbolo de asignación, seguida por una cadena de no terminales y terminales.

ESCRITURA DE UNA GRAMATICA

Las gramáticas describen la mayoría de las sintaxis de los lenguajes de programación.Toda construcción que se pueda describir mediante una expresión regular también se puede describir por medio de una gramática.

Por ejemplo, para la expresión regular (a|b)\* abb

Y la gramática: A0 → a A0 | b A0| b A1

A1 → b A2

A2 → bA3

A3 → є

ANALISIS SINTACTICO DESCENDENTE

Se considera un intento de encontrar una derivación por la izquierda para una cadena de entrada.También se puede considerar como un intento de construir un árbol de análisis sintáctico para la entrada comenzando desde la raíz y creando nodos del árbol en orden previo.

ANALISIS SINTACTICO ASCENDENTE

El análisis sintáctico ascendente intenta construir un árbol para la cadena de entrada que comienza por las hojas (el fondo) y avanza hacia la raíz (la cima).

ANÁLISIS SINTÁCTICO POR PRECEDENCIA DE OPERADORES

Para una pequeña clase de gramáticas se puede construir con facilidad a mano eficientes analizadores sintácticos ascendentes.Estas gramáticas, por precedencia de operadores, tienen la propiedad de que ningún lado derecho de la producción es є ni tiene 2 terminales adyacentes.

Una gramática con esta última propiedad de denomina gramática de operadores.

ANALIZADORES SINTÁCTICOS IZQUIERDA-DERECHA

Es una técnica eficiente de análisis sintáctico ascendente que se puede utilizar para analizar una amplia clase de gramáticas independientes de contexto, denominada Análisis sintáctico LR(k)

L es por el examen de la entrada de izquierda a derecha (left to right)

R por construir una derivación por la derecha (right most derivation) en orden inverso.

K por el número de símbolos de entrada de examen por anticipado utilizados para tomar decisiones del análisis sintáctico. Cuando se omite, se asume que k es 1.

Este análisis es atractivo por varias razones:

• Reconocen prácticamente todas las construcciones de los lenguajes de programación para los que se pueden escribir gramáticas independientes del contexto.

• Puede detectar un error sintáctico tan pronto como sea posible hacerlo en un examen de izquierda a derecha de la entrada.

USO DE GRAMÁTICAS AMBIGUAS

Gramática ambigua es aquella que produce más de un árbol de análisis sintáctico para alguna frase, es decir, una gramática ambigua es la que produce más de una derivación por la izquierda o por la derecha para la misma frase.Algunos tipos de gramáticas ambiguas son útiles en la especificación e implementación de lenguajes. Se utiliza para el aislamiento de construcciones sintácticas habituales para optimación en casos especiales.

Una Sentencia es ambigua si hay más de una derivación distinta. Si una sentencia es ambigua, el árbol de análisis sintáctico no es único; podemos crear más de un árbol de análisis sintáctico para la misma sentencia.Con una gramática ambigua se pueden especificar las construcciones de casos especiales añadiendo cuidadosamente nuevas producciones a la gramática. Las construcciones ambiguas se deben usar raramente y de una manera estrictamente controlada, pues de lo contrario no se puede reconocer con seguridad el lenguaje que reconoce el analizador.

**IMPLEMENTACION DEL PROCESO DE RECUPERACION DE ERRORES**

En teoría, se supone que la salida del analizador sintáctico es alguna representación del árbol sintáctico que reconoce la secuencia de tokens suministrada por el analizador léxico.

Si un compilador tuviera que procesar sólo programas correctos, su diseño e implantación se simplificarían mucho.

A menudo se pueden escribir programas incorrectos, y un buen compilador debería ayudar al programador a identificar y localizar errores.

Es más, considerar desde el principio el manejo de errores puede simplificar la estructura de un compilador y mejorar su respuesta a los errores.

Los errores en la programación pueden ser de los siguientes tipos:

* Léxicos, producidos al escribir mal un identificador, una palabra clave o un operador.
* Sintácticos, por una expresión aritmética o paréntesis no equilibrados. • Semánticos, como un operador aplicado a un operando incompatible.
* Lógicos, puede ser una llamada infinitamente recursiva.

El manejo de errores de sintaxis es el más complicado desde el punto de vista de la creación de compiladores.

Nos interesa que cuando el compilador encuentre un error, se recupere y siga buscando errores.

Por lo tanto el manejador de errores de un analizador sintáctico debe tener como objetivos:

* Indicar los errores de forma clara y precisa. Aclarar el tipo de error y su localización.
* Recuperarse del error, para poder seguir examinando la entrada.
* No ralentizar significativamente la compilación. Un buen compilador debe hacerse siempre teniendo también en mente los errores que se pueden producir; con ello se consigue:

1. Simplificar la estructura del compilador.
2. Mejorar la respuesta ante los errores.

En nuestro caso la estrategia utilizada es la de capturar el error e ingresarlo a un arreglo de errores y que se vaya al fin de línea, con el propósito de continuar ejecutando el programa.

void asignarVar():

{Token t;}

{

try {

t=<IDENTIFICADOR> {jjtThis.value = t.image;} #IDENTIFICADOR(true) t=<ASIGNACION> {jjtThis.value = t.image;} #ASIGNACION(true) tipoAsignacion() t=<FIN\_LINEA> {jjtThis.value = t.image;} #FIN\_LINEA(true)

}catch(ParseException e) {

errorSaltar(FIN\_LINEA);

}

}

De igual manera, utilizamos esta estrategia pero también para un bloque de instrucciones, cuando se requiere cambio en el final de este.

void condicion():

{Token t;}

{

try {

t=<PALABRA\_RESERVADA\_SI> {jjtThis.value = t.image;} #SI(true) grupoCondiciones() t=<PALABRA\_RESERVADA\_ENTONCES> {jjtThis.value = t.image;} #ENTONCES(true) instruccion() [t=<PALABRA\_RESERVADA\_SINO> {jjtThis.value = t.image;} #SINO(true) instruccion()] t=<PALABRA\_RESERVADA\_FINSI> {jjtThis.value = t.image;} #FINSI(true)

}catch(ParseException e) {

errorSaltar(PALABRA\_RESERVADA\_FINSI);

}

}

**ANALISIS DE DIFICULTADES EN EL PROCESO DE GENERACION DEL ANALIZADO SINTACTICO**

Una de las principales dificultades en el proceso, radica en la correcta creación de la gramática, en virtud a que las expresiones deben contener la información precisa y así evitar la generación de ambigüedades.

La generación clásica de un BNF, en nuestro caso, el borrador inicial, puede resultar un poco ajena a como el complemento de *JavaCC* demuestra su funcionalidad, caso ejemplar la cerradura de estrella.

Es necesario adquirir un conocimiento mas amplio sobre el funcionamiento del Plugin *JavaCC*, ya que esta herramienta provee características de eficiencia en su desarrollo dentro de cualquier programa.

Hubo dudas respecto al funcionamiento del *Lookahead*, pues esta aplicabilidad usada de modo correcto debe proveer mayor eficiencia en la programación del analizador.

**CONCLUSIONES**

Un analizador sintáctico (o parser) es un programa informático que analiza una cadena de símbolos de acuerdo a las reglas de una gramática formal. El término proviene del latín pars, que significa parte (del discurso). Usualmente hace parte de un compilador, en cuyo caso, transforma una entrada en un árbol sintáctico de derivación.

El análisis sintáctico convierte el texto de entrada en otras estructuras (comúnmente árboles), que son más útiles para el posterior análisis y capturan la jerarquía implícita de la entrada. Un analizador léxico crea tokens de una secuencia de caracteres de entrada y son estos tokens los que son procesados por el analizador sintáctico para construir la estructura de datos, por ejemplo un árbol de análisis o árboles de sintaxis abstracta.

El análisis sintáctico también es un estado inicial del análisis de frases de lenguaje natural. Es usado para generar diagramas de lenguajes que usan flexión gramatical, como los idiomas romances o el latín. Los lenguajes habitualmente reconocidos por los analizadores sintácticos son los lenguajes libres de contexto. Cabe notar que existe una justificación formal que establece que los lenguajes libres de contexto son aquellos reconocibles por un autómata de pila, de modo que todo analizador sintáctico que reconozca un lenguaje libre de contexto es equivalente en capacidad computacional a un autómata de pila.

Todas las técnicas existentes se aplican sin mayores cambios. La posibilidad de dotar de comportamiento a los datos simplifica el desarrollo de las etapas.

Además, las librerías habitualmente provistas en los lenguajes orientados a objetos como Java facilitan la codificación del compilador, en particular el sistema de colecciones.

Los próximos pasos a seguir son la implementación de la etapa de generación de código.