Autómatas, Teoría del Lenguaje y Compiladores: Implementación de un compilador

TPE - Raptoken

Alumnos

53214 - Fraga, Matias

52066 - Soncini, Lucas

52051 - De Lucca, Tomás

Profesores

Santos, Juan Miguel

Arias Roig, Ana Maria

Ramele, Rodrigo Ezequiel

Índice

1. Consideraciones realizadas

A la hora de realizar el compilador, consideramos distintas posibilidades para la creación del lenguaje propio, intentando distanciarlo lo más posible del lenguaje Java dentro de la consigna ya que al utilizar mismos bloques (if, while, main) y mismos tipos era bastante complicado salirse mucho del mismo. Más que nada pudimos separarnos más a la hora de la entrada y salida de datos como se explica en la sección propia de la explicación de la gramática.

Además de los requisitos obligatorios del trabajo práctico, se cuenta con un script para la generación de pruebas.

2. Descripción del desarrollo del trabajo

Para el desarrollo del trabajo se generó un Scanner utilizando la libreria jFlex, configurado para generar un scanner de Tokens definidos para nuestra gramática (la cual se encuentra explicada en la próxima sección). Dichos tokens serán procesados posteriormente utilizando el motor CUP, configurado igualmente, con las reglas que compondrán nuestra gramática, generando de esa forma el Parser.

Por otra parte, el trabajo cuenta con 7 programas, 5 de los cuales eran requeridos por la consigna:

- Factorial: Calcula el factorial de un número natural.
- Test de primalidad: Determina si un número natural es o no un numero primo.
- Fibonacci: Calcula los primeros n valores de la serie de Fibonacci.
- Ta-Te-Ti: Juego Ta-Te-Ti multijugador con un tablero interactivo.
- Rock Paper And Scissors: Juego de piedra papel y tijera multijugador.

Además, se diseñaron 2 programas adicionales:

- Pow: Calcula cualquier potencia de un número dado.
- Drawer: Interactivo programa que permite dibujar en pantalla cualquier combinación de caracteres, de forma dinámica y manejando tiempos de ejecución.

3. Descripción de la gramática

A la hora de definir la gramática buscamos generar un lenguaje que facilite la fluidez a la hora de programar, evitando así escribir caracteres innecesariamente, como por ejemplo el ';' como delimitador de una línea de código. Ademá buscamos que sea un código legible y fácil de seguir y entender para quienes no estén familiarizados con el.

Definimos nuestra gramática G en BFN de la siguiente forma:

$$G : \langle NT, T, s, P \rangle$$

La misma parte de un símbolo inicial s para generar todo el lenguaje definido. Por la sintaxis de nuestro lenguaje, dicho primer símbolo solo puede expandirse en una única producción que genere el bloque que enmarca nuestro programa. Dicho bloque comienza con la palabra reservada "BEGIN" y finaliza con otra palabra reservada "FINISH". Luego, utilizando siempre saltos de linea "
n" como delimitadores de operaciones entre dichas sentencias se encontrará nuestro código. Para el código utilizamos el símbolo no terminal "codee" (ya que "code" era una palabra reservada de cup y por ende no era permitida). El mismo esta compuesto simplemente de una sucesión de líneas de código (code_line). Para mayor facilidad a la hora de la compilación dejamos en dichas producciones el análisis de los saltos de línea.

Luego, cada code_line de nuestro lenguaje puede ser una expresión numérica, alguna operación relacionada a asignación o definición de variables o constantes, una entrada de input del usuario, un print, un bloque IF ELSE, un bloque DO WHILE o bien simplemente una línea vacía. Para cada caso se tiene la producción necesaria para analizarlo, salvo en los casos de entrada y salida que se cuenta con dos producciones para tratar de distinta manera las expresiones de texto y numéricas.

Con respecto al manejo de variables, decidimos simplificar la lectura de nuestra gramática separando dichos casos por medio de otro símbolo no terminal "var". El mismo luego puede expandirse en varios casos que abarcan todas las posibilidades de nuestro lenguaje a la hora de definir, asignar, definir y asignar en simultáneo, o bien modificar variables reasignándolas. Cabe destacar que nuestro lenguaje como cambio importante con java permite una asignación default tanto para Strings como para Integers a la hora de crear los objetos sin setear. También en las producciones propias de "var" definimos el uso de operadores de asignación $<+, <-, <*, </$ para facilitar la escritura a la hora de querer hacer una operación simple con una variable asignando el

valor.

Volviendo a "code_line", para el caso de entrada y salida de datos se definieron funciones out() y in(). Las mismas no son como las de java, sino que las pensamos de una forma mucho más cómoda y simple a la hora de usarlas. En el caso de impresión de datos en pantalla, out funciona directamente como un "System.out.print()" de java, es decir, se puede pasar distintos tipos de datos como parámetro y los mismos se convierten a String para imprimirse. Para la entrada de datos, decidimos usar una sintaxis ya directamente distinta a la usada en java. El parámetro pasado a dicha función es una variable ya definida, por lo tanto según el tipo de la misma, el tipo con el que interpreta la entrada de datos. Con respecto a los bloques IF y WHILE, decidimos no hacer algo muy raro ya que estos son similares en todos los lenguajes de programación. Simplemente usamos una sintaxis más parecida a pseudocódigo con END para marcar fines de bloques.

Al bloque while preferimos llamarlo LOOP pero sacando eso sigue la misma sintaxis a excepción de no necesitar paréntesis obligatoriamente para la condición de ciclo. Agregamos además para los bloques IF otra producción para admitir ELSE en la gramática.

Por último, con respecto a las expresiones, tenemos expresiones booleanas, numéricas y de texto. Para cada una utilizamos un símbolo no terminal diferente para así poder validar en nuestra gramática el correcto uso de los tipos de datos. De esta forma podemos verificar que una variable String solo pueda ser asignada un valor String y lo mismo para expresiones y variables numéricas. Cada una de estas cuenta con distintas producciones para especificar las operaciones permitidas, uso de paréntesis y demás. Cabe destacar que en el caso particular de las expresiones booleanas permitimos el uso de palabras reservadas true y false para poder implementar ciclos infinitos con los mismos y a futuro poder tan sólo incorporando un caracter de identificación de tipo de variable nuevo tener tipo booleano de datos.

Al decorar el arbol sintáctico optamos en la mayoría de los casos por tipos de datos String para los atributos de los símbolos no terminales salvando algunos casos particulares. Dicho es el caso del símbolo inicial s, el cual no requería de decorado alguno, y del símbolo propio de las expresiones numéricas el cual definimos como tipo Object para poder o bien asignar un valor o asignarle una cadena. Esto lo realizamos para poder a la hora de compilar realizar operaciones con valores numéricos y simplificar las expresiones a ejecutarse al correr el programa. De esta forma, al realizar por ejemplo "out(var_name + 3*4)" el binario generado será exactamente el mismo de "out(var_name + 12)" simplificando las operaciones a realizar en runtime.

$T : \{\ PLUS,\ TIMES,\ MINUS,\ DIVIDE,\ ASSIGN,\ MOD,\ LPAREN,\ RPAREN,\ JUMP,$
$BEGIN,\ FINISH,\ AS\_PLUS,\ AS\_MINUS,\ AS\_TIMES,\ AS\_DIV,\ IO\_OUT,\ IO\_IN,\ IF,\ ELSE,$
$LOOP,\ END,\ CMP\_LT,\ CMP\_GT,\ CMP\_EQ,\ CMP\_NE, BOOL\_OR,\ BOOL\_AND,$
$BOOL\_NOT,\ TRUE,\ FALSE,\ NUM\_DEF,\ TEXT, NUM\_VAR,\ TEXT\_VAR,\ TEXT\_DEF,$
$NUM\_CONST\_DEF,\ TEXT\_CONST\_DEF,\ NUM\_CONST,\ TEXT\_CONST,\ NUMBER\}$

$NT : \{s, codee, code\_line, var, text\_expr, if\_block, boolean\_exp, boolean\_term, num\_expr\}$

$P : \{$

$$s \rightarrow BEGIN\ JUMP\ codee\ FINISH$$

$$codee \rightarrow codee\ code\_line\ JUMP \mid code\_line\ JUMP$$

$$code\_line \rightarrow num\_expr \mid var \mid IO\_OUT\ LPAREN\ text\_expr\ RPAREN$$

$$code\_line \rightarrow IO\_OUT\ LPAREN\ num\_expr\ RPAREN \mid IO\_IN\ LPAREN\ NUM\_VAR\ RPAREN$$

$$code\_line \rightarrow IO\_IN\ LPAREN\ TEXT\_VAR\ RPAREN\ if\_block$$

$$code\_line \rightarrow LOOP\ boolean\_exp\ JUMP\ codee\ END \mid LAMBDA$$

$$if\_block \rightarrow IF\ boolean\_exp\ JUMP\ codee\ ELSE\ JUMP\ codee\ END \mid IF\ boolean\_exp\ JUMP\ codee\ END$$

$$boolean\_exp \rightarrow LPAREN\ boolean\_exp\ RPAREN \mid boolean\_term\ BOOL\_OR\ boolean\_exp$$

$$boolean\_exp \rightarrow boolean\ \_term\ BOOL\_AND\ boolean\_exp \mid boolean\_term \mid BOOL\_NOT\ boolean\_exp$$

$$boolean\_term \rightarrow TRUE \mid FALSE \mid LPAREN\ num\_expr\ CMP\_LT\ num\_expr\ RPAREN$$

$$boolean\_term \rightarrow LPAREN\ num\_expr\ CMP\_GT\ num\_expr\ RPAREN$$

$$boolean\_term \rightarrow LPAREN\ num\_expr\ CMP\_EQ\ num\_expr\ RPAREN$$

$$boolean\_term \rightarrow LPAREN\ num\_expr\ CMP\_NE\ num\_exprRPAREN$$

$$var \rightarrow NUM\_DEF \mid NUM\_DEF\ ASSIGN\ num\_expr \mid TEXT\_DEF \mid TEXT\_DEF\ ASSIGNtext\_expr$$

$$var \rightarrow TEXT\_VAR\ AS\_PLUS\ text\_expr \mid NUM\_VAR\ AS\_PLUS\ num\_expr$$

$$var \rightarrow NUM\_VAR\ AS\_MINUS\ num\_expr \mid NUM\_VAR\ AS\_TIMES\ num\_expr$$

$$var \rightarrow NUM\_VAR\ AS\_DIV\ num\_expr \mid NUM\_CONST\_DEF\ ASSIGNnum\_expr$$

$$var \rightarrow TEXT\_CONST\_DEF\ ASSIGN\ text\_expr \mid TEXT\_VAR\ ASSIGN\ text\_expr$$

$$var \rightarrow NUM\_VAR\ ASSIGN\ num\_expr$$

$$text\_expr \rightarrow TEXT \mid text\_expr\ PLUStext\_expr \mid TEXT\_VAR \mid TEXT\_CONST$$

$$num\_expr \rightarrow NUMBER \mid num\_expr\ PLUS\ num\_expr \mid num\_expr\ MINUS\ num\_expr$$

$$num\_expr \rightarrow num\_expr\ TIMES\ num\_expr \mid num\_expr\ DIVIDE\ num\_expr \mid num\_expr\ MOD\ num\_expr$$

$$num\_expr \rightarrow LPAREN\ num\_expr\ RPAREN \mid NUM\_VAR \mid NUM\_CONST$$

}

4. Dificultades encontradas en el desarrollo

A lo largo del desarrollo del trabajo fueron surgiendo inconvenientes que el grupo ha ido resolviendo con solvencia. El primer problema con el que nos encontramos, fue la imposibilidad de definir el lenguaje ya que requería conocimientos de gramáticas BFN los cuales no poseíamos cuando la consigna fue presentada, debido a ello el tiempo disponible para la realización del trabajo fue considerablemente menor a la estimada por el grupo. A pesar de ello, hicimos uso del tiempo previo a la definición de la gramática para involucrarnos con la sintaxis tanto de jFlex como de CUP, con las cuales no hubo mayores inconvenientes.

Otro de los problemas que surgieron fue la falta de conocimiento acerca del bytecode de Java, lo cual generó grandes inconvenientes en el desarrollo del trabajo, debido a ello, se optó por realizar un script que compile el código generado por nuestro parser.

Se encontraron problemas al momento de comprobar que no se asignen variables con tipos distintos. Es decir, por ejemplo, que una variable que ya había sido declarada con el tipo de dato numérico, no le pueda ser asignada una del tipo de texto. Al mismo tiempo, otro inconveniente encontrado surgió al momento de prevenir que se redefinan variables ya existentes. Otro problema de esta índole, fue el limitar que las constantes no puedan ser asignadas a nada una vez definidas, es decir, que sean utilizadas solamente como constantes. La solución a todos estos problemas fue la misma: se implementó un Map en el archivo generado por *jlex* que se encarga de llevar un registro de todas las variables y constantes generadas. De esta forma, se puede no solo prevenir todo lo descripto en este párrafo, sino que también poder mostrar errores de compilación descriptivos.

5. Futuras extensiones

Dado el scope del trabajo práctico muchas funcionalidades básicas de los lenguajes de programación tradicionales no fueron implementadas.

Algunas futuras extensiones podrían ser las siguientes:

- Manejo de vectores
- Llamados a función (subprocesos)
- Uso de librerías

Requieren identificadores de Tokens en jFlex y un posterior procesamiento de dichos tokens en CUP para ser implementadas (al encontrarse en Java, es posible traducirlas de nuestro lenguaje sin introducir complejidad en el código). En el caso de los llamados a función y el uso de librerías será necesario introducir un mecanismo de detección de errores cuando la función que se desea llamar no esté declarada o la librería no exista.

Cabe destacar que los lenguajes de programación que hoy en día triunfan en el mercado, poseen funcionalidades mucho mas complejas que, sin embargo, han sido añadiendose a lo largo de los años (por ejemplo, el uso de Generics en Java, o más recientemente, la incorporación de cualidades del paradigma funcional).

6. Referencias

- http://jflex.de/manual.pdf
- http://www.cs.princeton.edu/ appel/modern/java/CUP/manual.html
- http://www.cs.princeton.edu/ appel/modern/java/JLex/current/manual.html