Trabajo Práctico Especial

Gramatica & Compilador

Integrantes del Grupo X:

Calatayud, Agustin
Vazques, Agustin
Monti, Maria Florencia
Scomazzon, Martina

Profesores a cargo:

Santos, Juan Miguel

TEORIA DE LENGUAJES & COMPILADORES | INSTITUTO TECNOLOGICO DE BUENOS AIRES | JUN 2018

ÍNDICE

INTRODUCCIÓN

Con el fin de implementar los conocimientos adquiridos a lo largo del cuatrimestre se nos pidió a los alumnos el desarrollo completo de un lenguaje y su compilador, construyendo las dos componentes principales; analizador léxico y analizador sintáctico.

Para la definición o construcción del mismo podemos destacar dos grandes partes:

1. **La gramática,** quien se encargará de producir el lenguaje que luego el compilador aceptará y transformará.

2. **El compilador**, responsable de la 'transformación.' Toma un lenguaje fuente que pertenece a una gramática bien definida y lo transforma a un lenguaje de salida, de más bajo nivel.

IDEA SUBYACENTE & OBJETIVO DEL LENGUAJE

A lo largo de la carrera universitaria y profesional se nos presentan una gran variedad de lenguajes de programación: C, Java, Assembler, Ruby, SQL, Octave, entre tantas.

Con el fin de crear un lenguaje que describa al grupo se buscó obtener, de los lenguajes conocidos entre pares, aquellas características o elementos que sean de preferencia para cada uno. Por ejemplo, la simplicidad de la existencia del tipo de dato booleano, muy común en Java.

El lenguaje se diseñó con el objetivo de optimizar el uso del lenguaje C, que bajo nuestro punto de vista, posee algunas limitaciones, entre ellas, ya nombrada. La falta de un tipo de dato booleano.

A través de las pequeñas mejoras realizadas consideramos que logramos brindarle al usuario shortcuts que le permitirán completar su tarea más rápido. Combinamos la facilidad de escritura de C, sin tener que crear objetos, con estructuras provistas por Java, tales como Queue.

CONSIDERACIONES REALIZADAS

Comúnmente los parsers mientras analizan sintácticamente los datos de entrada de los tokens crea un árbol sintáctico el cual puede ser usado posteriormente en el proceso de compilación. Dicho árbol funciona como una representación media del programa.

Muchas veces, nuestro caso, el output del programa no es realizado directamente por acciones, si no que, el árbol es construido en memoria y las transformaciones correspondientes son aplicadas previo a la salida generada.

Por lo tanto, propusimos, generar un nodo distinto para cada tipo de elemento no terminal de nuestra gramática.

IMPLEMENTACIÓN

Anteriormente, en la introducción, hablamos de dos partes fundamentales para este trabajo, la gramática y el compilador. A continuación detallaremos los métodos y/o archivos implementados con el fin de llevar a cabo la tarea deseada.

Descripción de la Gramática

La gramática; ya explicamos el porqué y en qué nos basamos a la hora de diseñarla, ahora llegó el momento de exponerla.

Se creó una gramática de acuerdo a los requisitos planteados y solicitados por la cátedra que consiste en:

- **Tipos de Datos**
  - Simples
    - String
    - Integer
    - Boolean
  - Compuestos
    - Queue [ ]
    - Stack < >

- **Constantes**
  - Define
- **Operadores aritméticos**
  - +
  - -
  - *
  - /
  - %
- **Operadores Relacionales**
  - <
  - >
  - <=
  - >=
  - ==
  - !=
- **Operadores Lógicos**
  - &&
  - ||
- **Operadores de Asignación**
  - =
- **Bloque Condicional**
  - If
  - Else
- **Bloque de Repetición hasta condición**
  - While
  - For

Los operadores aritméticos fueron redefinidos según el tipo de dato que se trataba. La novedad llega para los datos compuestos. Los operadores '+' y '-' funcionan como operadores para agregar o quitar elementos.

Para esta primera fase se desarrollaron los archivos LEX y BNF encargados específicamente de describir y procesar la entrada de un programa en el lenguaje creado y posteriormente genere como salida una palabra del lenguaje.

Descripción del Desarrollo

Por último, el compilador. Aquí nos basamos en las sugerencias provistas por la cátedra en el enunciado inciso 3.1 - Yet Another Compiler Compiler - donde brevemente se detalla que los compiladores de compiladores son herramientas que permiten en base a la definición de un autómata finito y una gramática en BNF generar un parser que pueda aceptar la palabra del lenguaje, detectar posibles errores sintácticos y/o semánticos y traducir el programa para generar un binario ejecutable.

Se creó un archivo de tipo YACC donde se describieron cada una de las producciones asociadas a los no terminales correspondientes a nuestra gramática ya escrita en LEX.

Y ahora, llegó el turno de programar en C. Cada una de los no terminales fue asociado a un tipo de nodo distinto con el fin de poder crear correctamente un árbol de análisis sintáctico mientras realizamos el parsing de los correspondientes datos de entrada. Cada uno de estos nodos fue creado en 'nodes.h' y luego sus funciones de creación en un archivo aparte, 'nodes.c.'

Al momento de crear los nodos podrían surgir errores. Por ejemplo, un error en los parámetros, es decir, un error en la producción. Los errores son tratados internamente con el fin de que no sean detectados por otro compilador que no sea el nuestro.

Aquellas producciones que inicializan variables o funciones además poseen un segundo chequeo. ¿El nombre de la variable o función ya existe? En caso de existir, nuestro compilador indicará que hay un error, el código de entrada debe ser chequeado. Todas las funciones relacionadas con el chequeo y creación de variables pueden ser encontradas en 'typeChecks.c.'

Los errores más importantes que se detectan y tratan a lo largo de toda la implementación son: incompatibilidad de tipos, repetición de variables, variables inexistentes a las que se quieren referenciar, error en los argumentos de la función, entre otros.

Por último, translateNodes.c aquí es donde el compilador traduce cada una de las producciones escritas en YACC en código de lenguaje C.

DIFICULTADES ENCONTRADAS

Inicialmente, la idea fue otra, buscamos la posibilidad de generar un lenguaje en donde no haya que indicar de qué tipo de dato estamos hablando al crear una variable.

Para ello planteamos la posibilidad de almacenar cada uno de estos datos en una tabla, algo similar a lo que comúnmente llamamos hashtable.

El problema surge más adelante, cada vez que referenciamos a una variable o función. Por ejemplo llamando una función dentro de otra función y asignarlo a una variable definida anteriormente, debíamos recorrer dicha tabla incansablemente hasta encontrarla, o no. Procesarla, almacenarla, y luego, volver a chequear si el valor que la función retornó corresponde con el tipo de dato de la variable a la que se le está asignando, esto podría consumir mucho tiempo y memoria.

Dado que nuestro objetivo era la optimización del lenguaje, decidimos que sería mejor utilizar o crear un lenguaje tipado.

FUTURAS EXTENSIONES & CONCLUSIONES

¿Que mejoras podríamos realizar? En primer lugar, sin ninguna duda, completar nuestra idea inicial, realizar un lenguaje no tipado, o parcialmente no tipado dado que si fuese completamente no tipado no podríamos luego realizar la traducción a lenguaje C.

La complejidad depende únicamente de qué factor preferimos priorizar, optimidad y utilidad o simplemente que funcione sin importar cuanto tiempo pueda llegar a tardar.

En conclusión, a lo largo de este trabajo logramos aprender y entender el uso de herramientas como LEX & YACC.

Además, se aplicó de forma práctica lo aprendido en las clase; análisis ascendente de un LALR(1).

REFERENCIAS

1. https://efxa.org/2014/05/25/how-to-create-an-abstract-syntax-tree-while-parsing-an-input-stream/
2. https://avinashsuryawanshi.files.wordpress.com/2016/10/9.pdf
3. http://dinosaur.compilertools.net