UNIVERSIDAD NACIONAL AUTÓNOMA DE MÉXICO

FACULTAD DE INGENIERÍA

COMPILADORES

SEMESTRE 2020 - 1

Proyecto Final: Compilador

Profesor:

Adrián Ulises Mercado Martínez

Alumnos:

Cárdenas Cárdenas Jorge

Murrieta Villegas, Alfonso

Reza Chavarria Sergio Gabriel

Valdespino Mendieta Joaquín

Contacto: alfonsomvmx@comunidad.unam.mx

Índice

Introducción

En el presente proyecto a través de los conocimientos de la materia de Compiladores y mediante herramientas como Flex y Bison es como se realizará un compilador desde su análisis léxico, sintáctico y semántico hasta la generación del código de 3 direcciones obtenido como resultado del análisis de un archivo a través de estos 3 análisis.

Cabe destacar que el lenguaje de programación escogido para la programación del compilador fue Java debido a la gran ventaja que ofrecía el lenguaje al momento de utilizar distintas estructuras de datos como pilas, colas e incluso tablas hash.

Sin embargo, al igual que para la realización de compiladores para otros lenguajes como C, se utilizó *Flex* para el análisis léxico, *Bison* para el análisis sintáctico y semántico.

Marco Teórico

Compilador

Un compilador es un programa (traductor) que se encarga de hacer la traducción de un programa fuente escrito en lenguaje de alto nivel a un programa escrito en lenguaje objeto que por lo general es un lenguaje de bajo nivel (lenguaje ensamblador).
Para realizar esta traducción el compilador se auxilia de otros programas como lo son el preprocesador, que se encarga de recolectar el programa escrito en módulos en archivos separados, expandir fragmentos de código abreviados de uso frecuentes, llamados macros y la inclusión de las bibliotecas.

El programa modificado por el preprocesador ingresa al compilador, este producirá el programa destino escrito en lenguaje ensamblador, que a continuación es procesado por el ensamblador que genera el código de máquina para la arquitectura destino.

Una vez que el programa ha sido ensamblado, es necesario vincular los archivos de código máquina con otros archivos objeto y de biblioteca para que se produzca el código ejecutable. El código máquina que es generado por el ensamblador no es un código que pueda ser ejecutado ya que contiene direcciones de memoria relativas por ello recibe el nombre de código re-localizable.
Finalmente, el cargador lleva el archivo objeto a la memoria para su ejecución, asignando direcciones de memoria absolutas.

Instrucciones para montar el proyecto

Requisitos de software

A continuación se muestra todo el software necesario para poder montar el proyecto:

- NetBeans en su versión 8.0.2 como mínimo
- El JDK de JAVA en su versión 1.8.0.202 al menos
- JFlex en su versión 1.7.0
- Yacc 1.8

Instrucciones

1) Tener instalado Java y Netbeans, para esto se pueden checar realmente varios tutoriales en internet.
   a) La opción más rápida es instalar directamente desde la página oficial de Netbeans (La de Oracle) el IDE que incluye dentro al JDK, de esta forma garantizamos las últimas versiones tanto de la maquina como del IDE.

2) Posteriormente para instalar Jflex:
   2.1) Crear una carpeta en el disco local y descomprimir el archivo descargado

   2.2) Agregar a las variables de entorno la variable Jflex

   2.3) Para mayor detalle de la instalación de la parte Léxica se puede checar en el siguiente link https://jflex.de/manual.html

   NOTA: Una forma de verificar si está bien instalado, podemos utilizar en terminal el comando jflex.

3) Posteriormente para instalar Yacc:
   3.1) Primero debemos cambiar la dirección tanto en el disco local como en la dirección dentro del proyecto del archivo **parse.bat**

   3.2) Cabe destacar que para mayor información de la instalación y del uso se puede checar el siguiente link: http://byaccj.sourceforge.net/

   NOTA: Cabe destacar que a pesar de que está versión es muy similar a la de C, realmente todo debe manejarse con el paradigma orientado a objetos, por lo que todo los recursos se menejarán mediante clases.

Diseño y estructura del proyecto

En este apartado se describirá cada una de las clases empleados en la creación del compilador.
Cabe destacar que muchas de estas clases realmente se programaron convenientemente para poder utilizar algunas colecciones propias de Java sin realizar alguna instancia de estos. Por otro lado también se mencionarán cuales son las clases creadas por el mismo parser para poder realizar toda la lógica propia.

1. Clases del compilador

Como se mencionó previamente, las presentes clases son código que Yacc genera por automatica al momento de utilizarlo, esto debido a que hay cierto código que por un lado necesita pero además hay código donde nosotros debemos incluir cierta información al momento de llamarlo.
Las siguientes clases son aquellas que entran en esta parte del compilador:

- Parser.java
  - Al igual que ParserTokens es una clase genera al utilizar yacc, realmente en esta clase se encuentra gran parte de la lógica que utiliza yacc para el análisis sintáctico y semántico
- ParserTokens.java
  - Es la clase que utiliza yacc para el análisis semántico, en esa clase solamente se declaran de forma estática para que no hayan instancias de objetos, todos los tokens.

2. Clases auxiliares y de lógica

Debido a que se escogió un lenguaje Orientado a Objetos es por ello que nos vimos en la necesidad de poder manejar ciertas “clases auxiliares” para poder manejar de una manera más limpia y cómoda a todas las variables, listas y incluso “tablas” donde se guardaran las etiquetas, las variables y todos los elementos que se tuvieran que obtener al momento de pasar por los distintos análisis.

Las siguientes clases solamente tienen código para crear objetos de almacenamiento además de indices y otros recursos necesarios para los análisis:

- TablaTipos.java y TablaSimbolos.java
    - Ambas clases se basan en una tabla hash denominada hash map la cual podemos encontrarla como una colección de java.util.
    - La diferencia entre cada una de las clases es que al definir el atributo de la clase (La tabla hash) en el caso de la tabla de tipos se utilizó a una cadena y un tipo mientras que en la clase de las tablas de símbolos una cadena y a un símbolo.
    - Para ambas tablas se crearon los siguientes métodos:
        - El método de inserción en la tabla
        - Los métodos getters y setter de cada uno de los atributos correspondientes de las clases.
- Etiqueta.java
    - Es la clase encargada de hacer los objetos de tipo etiqueta para ello solo se utilizó un cadena como atributo además de los 3 tipicos métodos de cualquier clase, constructor, getter y setter.
- Tipo.java
    - En esta clase se manejan todos los tipos que podrá validar nuestro compilador, como es debido, la clase contiene su respectivo método constructor donde asigna los valores que que se le pasa como argumentos.

- Pilas.java
    - Clase destinada para el uso de las distitnas pilas ocupadas en el compilador, algunos casos son la de simbolos, tipos y de enteros.

- Index.java
    - La presente clase se utiliza para poder tener un contador en todas las clases que sea necesario sin la necesidad de tener un atributo independiente. como se puede apreciar en el método constructor solamente es un incremento al valor que se le pasa como argumento al constructor.

- Temp.java
    - Clase destinada al manejo de temporales en el análisis semántico, 3 atributos necesarios para el manejo de temporales, el tipo, el símbolo además de la dirección asociada al temporal. Se puede observar en el método constructor que solamente se hace instancias de la clase tipo y simbolo en donde se asignaran los respectivos valores para el manejo del temporal.

- Atributos.java
    - Clase donde se declararon todas los tipos de datos (Variables, listas ligadas, clases creadas

por nosotros) necesarias para el guardado de atributos al momento del análisis semántico

Por otro lado, las siguientes clases a diferencia de las anteriores, contienen la lógica en su gran mayoria del análisis semántico, es necesario mencionar que para este analisis se requierieron clases para el guardado de etiquetas, clases para la creación de las distintas tablas y también clases que nos permitieran generar tanto las cuadruplas como los archivos de salida.

- Cuadruplas.java
    - Esta clase es la encargada de guardar todos los datos necesarios para la generación del código de 3 direcciones. Sus atributos son los 4 posibles casos que debe contener una cuadrupla, operador, 2 argumentos y el resultado como se observan son cadenas. Cabe destacar que además de los atributos se crearon 3 métodos, el método construtor al que se le pasan todos los datos mediante argumentos, la eliminación de la cuadrupla lo cual realmente solo vacía la información contenida y un método de impresión el cual sirve para regresar el contenido a la cuadrupla.

- Operaciones.java
    - En esta clase se definen como se realizarán cada una de las operaciones soportadas por nuestro compilador, además de que en esta clase se definen 3 de las acciones más importantes del analisis semántico que son el ampliar, reducir y backpatch.
    - Por último, en esta clase también se define el método “GenerarArchivoCode” el cual es el encargado de escribir todo lo analizado por el compilador en un archivo externo (El archivo de salida)

3. NOTAS

A pesar de que los siguientes archivos no todos son clases también son parte del compilador:

- Compiler.java
    - Es la clase main del compilador, en esta clase solamente se instancia un objeto de tipo compiler, el cual genera los siguientes archivos y clases.
- ParserVal.java
    - Es la clase que define todos los objetos que serán ocupados por el parser para realizar su análisis respectivo.
- Ylex.java
    - Es la clase encargada de ligar la parte léxica con la sintáctica y semántica del compilador.
- Scanner.lex
    - Es el archivo de flex que contiene todo el apartado destinado para el analisis léxico
- Parser.y
    - Es el archivo de yac que contiene todo el apartado destinado al parser o análisis sintáctico
- Parser.bat:
    - Como se mencionó es el archivo que necesitamos para poder hacer ejecución del parse en el proyecto

Resultados y Pruebas de Escritorio

A continuación, se muestran los resultados obtenidos con el compilador al leer distintos archivo de entrada:

1) Archivo de entrada y compilación

```
ant -f E:\\uni\\compiladores\\compi\\Compiler -Dnb.internal.action.name=run run
init:
Deleting: E:\uni\compiladores\compi\Compiler\build\built-jar.properties
deps-jar:
Updating property file: E:\uni\compiladores\compi\Compiler\build\built-jar.properties
compile:
run:
COMPILED SUCCESSFUL
BUILD SUCCESSFUL (total time: 0 seconds)
```

```
ent x

func car main(sin) inicio
        car p
        escribir "c"
        si 2 < 1 yy 2 >= 2
                escribir 5
        sino
                escribir 10
        fin
        leer p
        leer x
        devolver "c"
        <**>
fin
```

2) Archivo de entrada y compilación

```
ant -f E:\\uni\\compiladores\\compi\\Compiler -Dnb.internal.action.name=run run
init:
Deleting: E:\uni\compiladores\compi\Compiler\build\built-jar.properties
deps-jar:
Updating property file: E:\uni\compiladores\compi\Compiler\build\built-jar.properties
compile:
run:
Error de reducción de tipo
COMPILED SUCCESSFUL
BUILD SUCCESSFUL (total time: 0 seconds)
```

Archivo Edición Formato Ver Ayuda

```
ent p

func ent main(sin) inicio
        car p
        ent Holas
        dreal dr
        Holas := 3
        p := "c"
        dr := 100 + 1
        escribir "c"
        si 2 < 1 oo falso yy 2 >= 2 entonces
                escribir 5
        fin
        leer p
        devolver 5
        <**>
fin
```

3) Archivo de entrada con errores de ampliación

```
ant -f E:\\uni\\compiladores\\compi\\Compiler -Dnb.internal.action.name=run run
init:
Deleting: E:\uni\compiladores\compi\Compiler\build\built-jar.properties
deps-jar:
Updating property file: E:\uni\compiladores\compi\Compiler\build\built-jar.properties
compile:
run:
Error de reducción de tipo
Error de reducción de tipo
COMPILED SUCCESSFUL
BUILD SUCCESSFUL (total time: 0 seconds)
```

```
ent x
--Hola comentario
func car main(sin) inicio
        car p
        escribir "c"
        si 2 < 1 yy 3 >= 2
                escribir 5
        sino
                escribir 10
        fin
        leer p
        leer x
        devolver "c"
        <**>
fin
func ent suma(sin) inicio
        ent y
        ent z, res
        res := 3 + 1
        devolver 1
fin
func dreal funcion(sin) inicio
        dreal z
        dreal za
        z := 1 + 1
        devolver 3
fin
```

Referencias

- Alfred V. Aho, Monica S. Lam. Compiladores principios, técnicas y herramientas. Pearson Addison Wesley.
- Sergio Gálvez Rojas, Miguel Ángel Mora Mata. Compiladores. Traductores y Compiladores. Dpto. de Lenguajes y Ciencias de la Computación. Universidad de Málaga.
- Recuperado del 19 de octubre de 2019, de https://www.gnu.org/software/bison/manual/html_node/index.html
- Recuperado el 20 de noviembre de 2019, de http://byaccj.sourceforge.net/
- Recuperado el 10 de noviembre de 2019, de https://jflex.de/manual.html