Entrega Final
Procesadores del Lenguaje:

Introducción:

Para esta práctica se nos plantea la modificación del compilador Tinto dado para añadir nuevas instrucciones al compilador, añadiendo las instrucciones do-while,switch-case-default, break y continue.

1º Descripción de las modificaciones léxicas realizadas

Para el desarrollo de esta práctica he modificado dentro de la parte léxica del Tito los archivos TintoParser.jj y TintoHeaderParser.jj.

Nota: realizando antes la instalación del compilador java cc dentro de eclipse.

Las modificaciones realizadas son la adicción de nuevas palabras reservadas para el léxico nuevo.

Añadiendo los DDOT (:) y las palabras reservadas de: continue, break, default, for, do, switch, case.

2º Descripción de las clases añadidas al Árbol de Sintaxis Abstracta del compilador:

Antes de explicar las clases añadidas defino que es un Árbol de Sintaxis Abstracta:

“Es una representación de árbol de la estructura sintáctica abstracta (simplificada) del código fuente escrito en cierto lenguaje de programación. Cada nodo del árbol denota una construcción que ocurre en el código fuente. Estos difieren de los árboles de análisis sintáctico en que las distinciones superficiales de forma, sin importancia en la traducción, no aparecen en los árboles sintácticos La sintaxis es abstracta en el sentido que no representa cada detalle que aparezca en la sintaxis verdadera. Los AST (Abstract Syntax Trees, o Árboles de Sintaxis Abstracta) sirven para manejar la información semántica de un código. La forma más eficiente de manejar la información proveniente de un lenguaje de programación es la forma arbórea; por eso la estructura de datos elegida es un árbol. Además, construyendo AST a partir de un texto podemos obviar mucha información irrelevante; si un AST se construye bien, no habrá que tratar con símbolos de puntuación. Al contrario que los flujos, una estructura en árbol puede especificar la relación jerárquica entre los símbolos de una gramática.”

Las clases nuevas añadidas seria:

SwitchStatement: que almacena como atributos privados el conjunto de cases que tiene, un Statement Default, un Statement body que es el conjunto de instrucciones que tiene dentro y una Expression condition que almacena la variable que toma como referencia para realizar sus posteriores comparaciones.

BreakStatement: clase que no posee atributos, pero es necesario crearla para controlar su existencia dentro de nuestro compilador.

ContinueStatement: igual que la clase BreakStatement, mismo uso.

DefaultStatement: clase que posee el único atributo de Statement body, el conjunto de instrucciones que tiene dentro de su bloque.

DoWhileStatement: clase que posee una Expression condition para alamacenar la condición de repetición del do-while, un Statement instruction que contiene todo el bloque de instrucciones del mismo.

ForStatement: clase que posee como atributos privados una Expression condition que corresponde a la condición de ejecución del For, un BlockStatement update que corresponde a la parte de la instrucción ForUpdate, BlockStatement Inicial que corresponde a la parte del ForInit(Inicializacion del for) y un Statement body que contiene las instrucciones del bloque de código del for(parte que se repite).

CaseStatement: que posee como atributos privados un Statement body que corresponde al cuerpo del case con todas sus instrucciones y dos Expression una del Switch al que pertenece y otra la del propio case para llevar un control de ambos en la generación de código.

Nota: todas las clases tienen creado sus métodos get de sus atributos necesario más adelante.

3º Descripción de las modificaciones sintácticas/semáticas realizadas a los parsers:

Para esta parte de la modificación he tocado el archivo TintoParser.jj que al compilarlo genera automáticamente el código a java. Realizando a cada una de las nuevas instrucciones sus métodos Try_Stm y su parse_.

Como ejemplo explicare uno en la memoria porque todos he seguido la misma mecánica de creación.

Por ejemplo para la instruccions Do-While he creado el método Statement tryDoWhileStm(SymbolTable symtab), donde almaceno en rsync el conjunto de siguientes de la instrucción.

Y dentro de un try-catch llamamos a la función parseDoWhileStm(symtab). Creando esta función también. En la que se realizan un parseo de todos los tokens y palabras reservadas para observar que sigue la estructura especificada.

Para el caso del Do-While la secuencia del parse seria:

DO

Body -> alamacenando el cuerpo Statement del Do-While(dentro de este ya poseen el control de los corchetes si los tiene).

WHILE

LPAREN

Cond -> almacenamos la condición del Do-While (de tipo Expression)

RPAREN

SEMICOLON

Y una vez realizado el parseo de todos los pasos anteriores realizamos una llamada al método action correspondiente que generara y comprobara si el objeto creado para este caso objeto DoWhileStatement se ha creado correctamente.

Lo pasos de creación para esta instrucción son los seguidos para todos las demás.

4º Descripción de la estructura del código intermedio asociado a cada instrucción y las modificaciones realizadas al código intermedio:

Para la modificación y creación de los nuevos métodos que generan el código intermedio para las nuevas instrucciones se realizan modificaciones en el archivo CodeGenerator.java del proyecto.

La creación de estos métodos sigue el esquema propuesto por usted en las prácticas de ejemplo de la asignatura.

Como ejemplo de explicación de lo desarrollado en esta parte del programa explicare la instrucción Do-While. Y como genero el código intermedio.

El objetivo de la función generateCodeOfDoWhileStatement seria devolver un CodeInstructionList con todas las etiquetas, saltos e instrucciones que tenga para este caso el Do While generado.

Por ello creamos un atributo de este tipo llamado codelist donde incluiremos de manera ordenada todas estas cosas generadas anteriormente.

En primer lugar creamos labels para saber el inicio de la instrucción, otro para el caso true de la condición del DoWhile, otro para el caso false del DoWhile, uno para controlar los Breaks y otro para los Continue.

Utilizando para estos dos últimos una estructura auxiliar, un arraylist de CodeLabel para controlar los saltos a los que tendría que dar los breaks y continues en caso de que se encuentre uno saber hacia donde debería de saltar.

Para los labels begin, true, continue, break false generamos su CodeInstruction, pasándole como parámetro su label correspondiente.

Y un CodeInstruction de tipo JUMP para almacenar el salto hacia el begin de la instrucción DoWhile.

Ahora realizamos la creación del CodeInstructionlist para la condición y el bloque.

Para la condición realizamos una llamada a la función CodeForCondition en la que le pasamos la expresión, el label de true y el label de false.

Y para el bloque llamamos a la función CodeOfStatement pasándole como parámetro el bloque.

Tanto la expresión como el bloque son obtenido con los métodos get generados en la clase DoWhile por ello realice la creación de estos métodos para utilizarlos en la generación de código. Todas las demás presentan también sus métodos get de sus atributos privados.

Y una vez tenemos todos los CodeInstruction y CodeInstructionList los almacenamos de manera ordenada en el codelist generado anteriormente para que lo devuelva la función.

Para este caso, el orden seria:

labelBeginInstruction

BloqueInstruction

labelContinueInstructon

CondicionInstruction

labelTrueInstruction

SaltoBeginInstruction

labelFalseInstruction

labelBreakInstruction

Nota: siendo el labelbreak al final para que en caso de que durante el bloque de código se encuentre un break salte al fin del DoWhile.

5º Las pruebas de funcionamiento del nuevo compilador, incluyendo ejemplos con las instrucciones añadidas y explicando el código ensamblador generado para estas instrucciones.

Las pruebas de cada una de las nuevas instrucciones generadas están almacenadas en una carpeta junto al proyecto con el nombre de “PruebasEjecucion”, donde se encuentra para cada fragmento de código un recorte para saber el código main ejecutado y un guardado de la ejecución con el programa QtSpim.

Nota: para la ejecución de las modificaciones realizadas he realizado un export del proyecto de tipo jar para ponerlo en la misma dirección que el proyecto. Para después ejecutarlo con un ejecutable .bat que lo compile y genere el código intermedio.