PRÁCTICA DE PROCESADORES DE LENGUAJES

Curso 2013 – 2014

Entrega de Septiembre

APELLIDOS Y NOMBRE: GARCÍA PAREDERO, FRANCISCO JAVIER

IDENTIFICADOR: FGARCIA1173

DNI: 51081513P

CENTRO ASOCIADO MATRICULADO: MADRID

CENTRO ASOCIADO DE LA SESIÓN DE CONTROL: MADRID

MAIL DE CONTACTO: GARCIA.PAREDERO@GMAIL.COM

TELÉFONO DE CONTACTO: 617252500

GRUPO (A ó B): B

1. CAMBIOS REALIZADOS EN LOS ANALIZADORES LÉXICO Y SINTÁCTICO

1.1. CAMBIOS EN EL ANALIZADOR LÉXICO

A nivel léxico no se han realizado cambios en general. Los tokens definidos son exactamente los mismos. Sobre el fichero Scanner.flex, sin embargo se han realizado tres modificaciones menores, se han eliminado ciertas trazas que se habían introducido en la entrega de febrero. En concreto, se trata de las siguientes modificaciones:

- En el reconocimiento de los tokens **if**, **end** e **IDENTIFICADOR**, se había añadido la palabra IF, END e IDENTIFICADOR respectivamente al lexema del token generado, para poder verificar que se estaba realizando un reconocimiento correcto. Esto ha sido eliminado en la presente entrega

1.2. CAMBIOS EN EL ANALIZADOR SINTÁCTICO

A lo largo del desarrollo de la práctica, la gramática ha ido cambiando para adaptarse a las directrices presentadas en las transparencias del curso virtual. Estas modificaciones han sido radicales, de tal modo que puede considerarse que la gramática ha sido prácticamente reescrita de nuevo.

2. EL ANALIZADOR SEMÁNTICO Y LA COMPARACIÓN DE TIPOS

Para realizar el análisis semántico se han creado subclases de la clase NonTerminal únicamente allí donde ha sido necesario, de este modo, en aquellas producciones donde no era necesario propagar información hacia arriba en el árbol, no se ha creado una subclase de NonTerminal.

Entre las acciones semánticas que se han introducido destacan:

- Verificación que el identificador de apertura de un programa o subprograma coincide con el identificador de cierre.

- Introducción de los tipos simples en la tabla de tipos al inicio del análisis. Para ello se ha empleado la clase proporcionada **TypeSimple**, en la que se han añadido constantes para referenciar a los tipos simples del lenguaje, a saber: NUMERO, CADENA y LOGICO.

- Comprobación de que no se declaran símbolos duplicados. Para ello, se emplea una tabla de símbolos en cada ámbito en las que se registran los símbolos según se van declarando. Gracias a estas tablas, puede verificarse si un elemento con el mismo identificador ya ha sido declarado previamente. Esto se hace en las declaraciones de constantes, de tipos,

- En las declaraciones de tipos, los tipos se almacenan en la tabla de tipos, previamente se comprueba que la tabla de tipos no contenga un tipo con el mismo nombre.

- Comprobaciones de tipos: se comprueba que el tipo definido en la declaración es un tipo existente

    - En las declaraciones de constantes.

    - En las declaraciones de variables.

    - En los parámetros de los subprogramas

    - En los tipos de retorno de las funciones.

- Apertura de nuevos ámbitos en la declaración de subprogramas

- Almacenamiento de los símbolos correspondientes a los parámetros en los subprogramas y el tipo del resultado en el caso de las funciones.

- Comprobación de la existencia de alguna sentencia de tipo return en las funciones.

- La compatibilidad de tipos en las expresiones.

2.1. DESCRIPCIÓN DE LA TABLA DE SÍMBOLOS IMPLEMENTADA

Para la tabla de símbolos se ha empleado la clase SymbolTableIF proporcionada por el framework de desarrollo. En esta tabla se han almacenado elementos de tipo constante, variable, funciones,

La clase SymbolConstant ha sido modificada, añadiéndole una propiedad value para almacenar el valor de la constante.

Para almacenar los campos de los registros, se ha empleado una tabla de símbolos interna dentro de la clase TypeRecord. No se ha empleado la tabla de símbolos global

Para evitar que se distinga entre letras mayúsculas y minúsculas, se ha optado por almacenar los símbolos con su nombre siempre en letras mayúsculas.

3. GENERACIÓN DE CÓDIGO INTERMEDIO

La generación de código intermedio se ha implementado siempre que se ha podido en el interior de las clases no terminales. Esta decisión se ha tomado para aprovechar las ventajas que proporciona un IDE como Eclipse para programar clases java. Para ello se ha creado un método llamado generarCodigoIntermedio en aquellos no terminales que requieran generar o propagar el código.

El juego de instrucciones de código intermedio que se han empleado se almacena en la clase InstructionSet.

4. GENERACIÓN DE CÓDIGO FINAL

Para la generación de código intermedio, se ha creado una estructura en la que existe una clase abstracta Translator de la que heredan los traductores concretos. La clase base tiene un método abstracto translate que devuelve el código correspondiente a la traducción de cada cuádrupla. Este método ha sido implementado por las implementaciones concretas de Translator.

Además, la clase Translator implementa un método traducirOperando que realiza la traducción de los operandos según sean variables globales, parámetros, variables locales, o valores literales.

5. INDICACIÓNES ESPECIALES

Por falta de tiempo, la generación de código final no ha podido finalizarse. Aunque se han probado los test proporcionados del 1 al 7, a partir del 8 no he logrado obtener un resultado satisfactorio. Este test numero 8 prueba la invocación de funciones, y no he logrado definir un registro de activación válido, por lo que he logrado implementar el salto a la función, no he logrado implementar el paso de parámetros ni un retorno correcto.

Aun así realizo la entrega por tratarse de la última posibilidad de hacerla y, dado el considerable trabajo realizado en la asignatura, con la esperanza de que sea suficiente para aprobar. Pues teniendo la teoría aprobada, solo es esta práctica lo que me queda para terminar la carrera.

6. GRAMÁTICA

start with program;

program ::= axiom

axiom ::= cabecera declaraciones cuerpo

| error

vacio ::=

cabecera ::= PROCEDURE IDENTIFICADOR ABRE_PARENTESIS CIERRA_PARENTESIS IS

| error

tipo ::= IDENTIFICADOR

| INTEGER

| BOOLEAN

tipo_primitivo ::= INTEGER

| BOOLEAN

valor ::= NUMERO

| TRUE

| FALSE

| CADENA

declaraciones ::= declaracion_constantes:

declaracion_constantes ::= constante declaracion_constantes

| declaracion_registros

constante ::= lista_nombres DOS_PUNTOS CONSTANT ASIGNACION valor PUNTO_COMA

lista_nombres ::= IDENTIFICADOR SEPARADOR_COMA lista_nombres

| IDENTIFICADOR

declaracion_registros ::= registro declaracion_registros

| declaracion_variables

registro ::= TYPE IDENTIFICADOR IS RECORD lista_campos END RECORD PUNTO_COMA

lista_campos ::= IDENTIFICADOR DOS_PUNTOS tipo PUNTO_COMA lista_campos

| vacio

declaracion_variables ::= variable declaracion_variables

| declaracion_subprogramas

variable ::= lista_nombres DOS_PUNTOS tipo PUNTO_COMA

declaracion_subprogramas ::= subprograma declaracion_subprogramas

| vacio

subprograma ::= funcion

| procedimiento

funcion ::= FUNCTION IDENTIFICADOR ABRE_PARENTESIS parametros_subprograma CIERRA_PARENTESIS RETURN tipo_primitivo

IS declaraciones cuerpo_funcion

procedimiento ::= PROCEDURE IDENTIFICADOR ABRE_PARENTESIS parametros_subprograma CIERRA_PARENTESIS

IS declaraciones:declaraciones cuerpo

parametros_subprograma ::= lista_nombres DOS_PUNTOS tipo PUNTO_COMA parametros_subprograma

| lista_nombres DOS_PUNTOS tipo

| vacio;

cuerpo_funcion::= BEGIN lista_sentencias_funcion END IDENTIFICADOR PUNTO_COMA

lista_sentencias_funcion ::= sentencia_funcion PUNTO_COMA lista_sentencias_funcion

| vacio

sentencia_funcion ::= sentencia

| sentencia_return

sentencia_return ::= RETURN expresion

cuerpo ::= BEGIN lista_sentencias END IDENTIFICADOR PUNTO_COMA

| error

expresion ::= expresion PLUS expresion

| expresion MAYOR expresion

| expresion IGUAL expresion

| expresion OR expresion

| ABRE_PARENTESIS expresion CIERRA_PARENTESIS

| llamada_subprograma

| referencia

| valor

referencia ::= IDENTIFICADOR

| acceso_registro

acceso_registro ::= referencia ACCESO IDENTIFICADOR

lista_sentencias ::= sentencia PUNTO_COMA lista_sentencias

| vacio

sentencia ::= sentencia_entrada_salida

| sentencia_if

| sentencia_for

| sentencia_asignacion

| llamada_subprograma

sentencia_if ::= IF expresion THEN lista_sentencias END IF

| IF expresion THEN lista_sentencias ELSE lista_sentencias END IF

| error

sentencia_for ::= FOR IDENTIFICADOR IN expresion

DELIMITADOR_RANGO expresion LOOP lista_sentencias

END LOOP

sentencia_asignacion ::= referencia ASIGNACION expresion

sentencia_entrada_salida ::= PUT_LINE ABRE_PARENTESIS expresion CIERRA_PARENTESIS

| PUT_LINE ABRE_PARENTESIS CADENA CIERRA_PARENTESIS

parametro ::= valor

| referencia

llamada_subprograma ::= IDENTIFICADOR ABRE_PARENTESIS lista_parametros CIERRA_PARENTESIS

lista_parametros ::= expresion SEPARADOR_COMA lista_parametros

| expresion

| vacio