Departamento de Cs. e Ingeniería de la Computación
Universidad Nacional del Sur

Compiladores E Intérpretes

Proyecto N°4

Analizador Semántico - Chequeo de Sentencias

García Amado Andrés
LU: 89522

Índice

Índice **1**

Introducción: **2**

Ejecución **3**

Esquema de Traducción: **4**

Chequeos del Compilador **9**

Introducción:

En este documento se describe como es el funcionamiento del Analizador Semántico parte 2. Se registran los detalles del analizador semántico de dos pasadas realizado para la tercera entrega del proyecto.
En primer lugar se muestran el esquema de traducción que realiza el compilador.
Luego se explica, los chequeos que realiza el compilador.

Ejecución

Para la ejecución se debe respetar la siguiente forma:

java –jar AnalizadorSemantico.jar <in_File> [<out_File>]

Donde el AnalizadorLexico.jar es el nombre del archivo ejecutable. in_file es el archivo que se analiza léxicamente y el out_file es un parámetro opcional, que de ser usado, se crea un archivo con el análisis léxico correspondiente, de ser ignorado el análisis se mostrará en consola.

Esquema de Traducción:

```
1       <Inicial> ::= <Clase> <C> EOF
1.1     <C> ::= <Inicial> | ε

2       <Clase>::= class idClase <H>
                                {if(!ts.estaClase(idClass.getLexema())){
                                    ts.agregarClase(idClase)
                                    clase.hereda <- h.nombre
                                  else
                                    Mostrar Error: Ya existe una clase con ese nombre
                                }
                                  { <Miembros> }
2.1     <H>::= <Herencia> {h1.nombre <- herencia.nombre}
        <H>::= ε {h.nombre <- "Object"}
2.2 <Miembros> ::= <Miembro> <Miembros> | ε

3       <Herencia>::= extends idClase {herencia.nombre <- idClase.nombre}

4       <Miembro>::= <Atributo> | <Ctor> | <Metodo>

5       <Atributo>::= <Visibilidad> <Tipo> {
                                            atributo.visibilidad <- visibilidad.nombre
                                            atributo.tipo <- tipo.salida
                                            listaDecVars <- new Lista()
                                           }
                                    <ListaDecVars>{
                                            para cada elemento en listaDecVars.lista
                                        ts.getClaseActual().agregarVariable(elto, atributo.visibiliadad,
atributo.tipo)
                                            }   ;

6       <Metodo>::= <FormaMetodo> <TipoMetodo> idMetVal {
                        Metodo m = new Metodo(idMetVal.lexema, FormaMetodo.fm,              .
TipoMetodo.tipo);
                        ts.setUnidadActual(m);
                        ts.getClaseActual().agregarMetodo(m);

                        }<ArgsFormales> <Bloque>{
                                        m.setCuerpo(bloque.salida)
                                    }

7       <Ctor>::= idClase {
                                if(!ts.getClaseActual().getNombre().equals(aux.getLexema()))
                            Mostrar Error: El constructor debe tener el mismo nombre que
.                                       la clase.
     if(ts.getClaseActual.getTieneConst()){
```

```
                        Mostrar Error: Ya tiene un constructor.
                            }
      else{
           ts.eliminarConstructorPredefinido();
           Unidad m = ts.getClaseActual().agregarConstructor(idClase);
           ts.setUnidadActual(m);
          }
                        }<ArgsFormales> <Bloque>{
                                        m.setCuerpo(bloque.salida);
                                          }

8       <ArgsFormales>::= ( <LAF> )
8.1     <LAF>::= <ListaArgsFormales> |  Ɛ

9       <ListaArgsFormales>::= <ArgFormal> <LAFS>
9.1     <LAFS>::= , <ListaArgsFormales> |  Ɛ

11      <ArgFormal>::= <Tipo> idMetVal
                                                                {
                    ts.getUnidadActual().agregarParametro(idMetVal.lexema,tipo.Salida)

}

12      <FormaMetodo>::= static {FormaMetodo.fm <- “static”}
        <FormaMetodo>::= dynamic {FormaMetodo.fm <- “dynamic”}

13      <Visibilidad>::= public {Visibilidad.visibilidad <- “public”}
        <Visibilidad>::= private {Visibilidad.visibilidad <- “private”}

14      <TipoMetodo>::= void {TipoMetodo <- new Void() }
        <TipoMetodo>::= <Tipo> {TipoMetodo <- tipo.salida}

15      <Tipo>::= boolean <Bool> | char <Ch> | int <I> | idClase | String
         <Tipo>::= idClase return new TipoClase();
         <Tipo>::= String return new TipoString();


15.1    <Bool>::= [] return new TipoArregloBool()
        <Bool>::= Ɛ  return new Bool()
15.2    <Ch>::= [] return new TipoArregloChar()
        <Ch>::=  Ɛ return new Char()
15.3    <I>::= [] return new TipoArregloInt();
        <I>::= Ɛ return new Int();

16      <TipoPrimitivo>::= boolean | char | int

17      <TipoReferencia>::= idClase | String | <TipoPrimitivo>

18      <ListaDecVars>::= idMetVal <LDVFact>
18.1    <LDVFact>::= , <ListaDecVars> | Ɛ
```

```
20      <Bloque>::= {
                    {
                    bloque.b <- new Bloque();
                    bloque.aux <- ts.getBloqueActual();
                    ts.setBloqueActual(bloque.b)
                    }
                    <S> }
                     {
                      ts.setBloqueActual(bloque.aux);
                      bloque.salida <- bloque.b;
                      }
20.1 <S>::= <Sentencia>{ S.sentencia <- Sentencia.salida;
                          ts.getBloqueActual().agregarSentencia(S.sentencia)
                        } <S> |  Ɛ


21      <Sentencia>::= ; {return SentenciaVacia()} | <Asignacion> ; {return Asignacion()}|
<SentenciaLlamada> ; {return sentenciaLlamada()}|
<Tipo>{
        listaDecVar.entrada <- new Lista()
       } <ListaDecVars> ; {
                           sentencia.listaVars<- new Lista();
                            for(idMetVar var: listaDecVars.salida)
                                  Variable v = new Variable();
                                   sentecia.listavars.add(v);
                                   bloqueActual.agregarVariable(v);
                                   sentencia.salida <- new DecVars(tipo.nombre, sentencia.listaVars)

}| if ( <Expresion> ) <Sentencia> <SentenciaFact>{ return If() }| while ( <Expresion> ) <Sentencia>{ return
While()}| <Bloque>  { return Bloque() }| return <Exp> ; { return Return()}
21.1    <SentenciaFact>::= else <Sentencia> |  Ɛ
21.2    <Exp>::= <Expresion> |  Ɛ

30      <Asignacion>::= <AccesoVar> = <Expresion> {Asignacion.salida <- New Asignacion()}   |

                        <AccesoThis> = <Expresion> {Asignacion.salida <- New Asignacion()}

32      <SentenciaLlamada>::= ( <Primario> )

33      <Expresion>::= <ExpOr> {Expresion.salida <- ExpOr.salida}

34      <ExpOr>::= <ExpAnd> {RestoExpOr.entrada <- ExpAnd.salida } <RestoExpOr> {expOr.salida <-
                   RestoExpOr.salida}

34.1    <RestoExpOr>::= || {  }<ExpAnd> {RestoExpOr.aux <- new ExpBinaria();
                                          RestoExpOr.entrada <- ExpAnd.salida
                                        }<RestoExpOr> {
                                                          ExpAnd.salida <- RestoExpOr.salida;
                                                       }|  Ɛ {RestoExpOr.salida <- RestoExpOr.entrada}
```

```
35      <ExpAnd>::= <ExpIg> <RestoExpAnd>
35.1    <RestoExpAnd>::= && <ExpIg> <RestoExpAnd> | ε

36      <ExpIg>::= <ExpComp> <RestoExpIg>
36.1    <RestoExpIg>::= <OpIg> <ExpComp> <RestoExpIg> | ε

37      <ExpComp>::= <ExpAd> <ExpCompFactorizada>
37.1 <ExpCompFactorizada>::= <OpComp> <ExpAd> | ε

38      <ExpAd>::= <ExpMul> <RestoExpAd>
38.1    <RestoExpAd>::= <OpAd> <ExpMul> <RestoExpAd> | ε

39      <ExpMul>::= <ExpUn> <RestoExpMul>
39.1    <RestoExpMul>::= <OpMul> <ExpUn> <RestoExpMul> | ε

40      <ExpUn>::= <OpUn> <ExpUn> {ExpUn1.salida <- new ExpUnaria()}
                                  | <Operando> {expUn.salida <- operando.salida;}

41      <OpIg>::= == | !=

42      <OpComp>::= < | > | <= | >=

43      <OpAd>::= + | -

44      <OpUn>::= + | - | !

45      <OpMul>::= * | /

46      <Operando>::= <Literal> {Operando.salida <- Literal.salida} |
                      <Primario> {Operando.salida <- Literal.salida}

48      <Literal>::= null | true | false | intLiteral | charLiteral | stringLiteral

49      <Primario>::= <ExpresionParentizada> | <AccesoThis> | <LlamadaMetodoEstatico> |
<LlamadaCtor> | idMetVal <PrimarioFactorizado>
49.1    <PrimarioFactorizado>::= <ArgsActuales> <Enca> | <Enca>

55      <ExpresionParentizada>::= ( <Expresion> ) <Enca>
55.1 <Enca>::= <Encadenado> | ε

56      <AccesoThis>::= this <Enca>

57      <AccesoVar>::= idMetVal <Enca>

58      <LlamadaMetodo>::= idMetVal <ArgsActuales> <Enca>

59      <LlamadaMetodoEstatico>::= idClase . <LlamadaMetodo> <Enca>

60      <LlamadaCtor>::= new <LlamadaCtorFactorizado>
```

60.1 <LlamadaCtorFactorizado>::= idClase <ArgsActuales> <Enca> | <TipoPrimitivo> [<Expresion>] <Enca>

62 <ArgsActuales>::= **(** <LE> **)**
62.1 <LE>::= <ListaExps> | ℰ

63 <ListaExps>::= <Expresion> <ListaExpsFact>
63.1 <ListaExpsFact>::= **,** <Expresion> | ℰ

65 <Encadenado>::= . idMetVal <EncadenadoFactorizado> | [<Expresion>] <Enca>
65.1 <EncadenadoFactorizado>::= <ArgsActuales> <Enca> | <Enca>

68 <LlamadaMetodoEncadenado>::= idMetVal <ArgsActuales> <Enca>

69 <AccesoVarEnca>::= idMetVal <Enca>

70 <AccesoArregloEncadenado>::= [<Expresion>] <Enca>

Chequeos del Compilador

Chequeo en sentencias:

- If: Consiste en chequear que el tipo de la expresión sea boolean y chequear la sentencia.
- Else: Consiste en chequear que el tipo de la expresión sea boolean y chequear ambas sentencias (la del if y la del else).
- Sentencia simple: Consiste en chequear la expresión
- Return: Consiste en chequear:
  - Que no se esté intentando retornar un valor dentro de un constructor.
  - Que no se esté intentando retornar un valor dentro un método void.
  - Si la expresión no es nula, se chequea la expresión.
- DecVars: Se chequea cada una de las variables.
- Asignación: Se chequea que el tipo de la expresión del lado izquierdo de la asignación se compatible con el tipo de la expresión del lado derecho.
- While: Se chequea que la expresión sea de tipo boolean y se chequea la sentencia.

Chequeo en expresiones:

- ExpBinaria: A partir del operador de la expresión, se chequea que este sea aplicable a partir de los tipos de las 2 expresiones adyacentes a este.
- ExpUnario: Se chequea que el tipo de la expresión sea boolean si el operador es '!' o que sea de tipo int si el operador es '+' o '-'.
- This: Se chequea que no se esté intentando utilizar dentro de un método estático. El chequeo de este nodo devuelve un objeto de tipo TipoClase con la clase actual.
- Literal: Devuelve un objeto del tipo que corresponda según el literal.
- Construir: Chequea que la tipo del objeto que se está queriendo crear exista en la tabla de símbolos y se verifican que los argumentos correspondan con los del constructor.
- LlamadaVar: Se controla que la variable esté entre las variables locales del método actual o que esté dentro de las variables de instancia de la clase.
- ExpConEncadenado: Se chequea la expresión.
- LlamadaConClase: Se chequea la clase a la que se referencia exista en la tabla de símbolos, y de existir, que posea el método estático que se está queriendo usar.