TALLER DE DISEÑO DE SOFTWARE

PROYECTO:
“Compilador para C-TDS”

Estudiantes: *Calcagno Camila*
Dosantos Claudio
Passalacqua Santiago.

Análisis sintáctico y léxico:

En esta etapa utilizamos las herramientas .jflex y .cup para poder desarrollar el scaner y el parser presentados para nuestro lenguaje ctds. En esta primer etapa no nos encontramos con mayores complicaciones, solo en algunas decisiones respecto a convenciones en la manera que escribimos la gramática.

Analisis semántico:
En esta etapa, para poder realizar los análisis correspondientes y chequeos, nos servimos del diagrama AST provisto por la cátedra, con ello comenzamos construyendo la tabla de símbolos, y luego construimos los Visitor necesarios. En nuestro proyecto tenemos las clases ASTVisitor, BuildVisitor, CheckTypeVisitor, MainVisitor, PrettyPrintVisitor. A medida que avanzábamos en estos chequeos debímos regresar a la tabla de simbolos porque notábamos que nos faltaban métodos, o atributos que iban surgiendo a raíz de nuestras necesidades.

Las restricciones semánticas de nuestro lenguaje, se encuentran plasmadas aquí:

* Los atributos de las clases solo pueden ser definidos de tipos básicos.

```
public String visit(Class_decl expr) {
    TableLevel x = new TableLevel();
    this.createLevel(x);
    if (expr.getField_decl() != null)
        for (Field_decl c :expr.getField_decl()){
            if (c.getType().isObject() || c.getType().toString()=="void")
                this.addError(c, "Invalid type");
            c.accept(this);
        }
    if (expr.getMethod_decl() != null)
        for (Method_decl c :expr.getMethod_decl()){
            c.accept(this);
        }
    this.closeLevel();
    return "";
}
```

* La estructura de los programas tienen distintos niveles los cuales son creados en BuildVisitor y garantizan la visibilidad y alcance de declaraciones.
    “1”: donde se encuentran todas las clases hasta el momento definidas.
    “2”: donde se encuentran los atributos y metodos de la clase corriente.
    “3”: donde se encuentran las declaraciones locales a un metodo.
    “mas niveles”: que surgen de la utilizacion de bloques dentro de un metodo.

* El programa debe contener la declaración de una clase llamada main, con un método llamado del mismo nombre.

```
public Integer visit(Program expr) {
    int cantMain = 0;
    if (expr.getClasses() != null)
        for (Class_decl c : expr.getClasses())
            if (c.getId().equals("main"))
                cantMain = cantMain + c.accept(this);
    if (cantMain!=1)
        this.addError(expr, "No cumple el requisito de tener un unico main con un metodo main");
    return cantMain;
}

public Integer visit(Class_decl expr) {
    int cantMain = 0;
    if (expr.getMethod_decl() != null)
        for (Method_decl c : expr.getMethod_decl())
            cantMain = cantMain + c.accept(this);
    return cantMain;
}

public Integer visit(Method_decl expr) {
    if (expr.getParam_decl() == null)
        if (expr.getId().equals("main"))
            return 1;
    return 0;
}
```

* Los nombres de los identificadores son unicos en cada scope. Es decir, no se puede utilizar el mismo identificador mas de una vez en cada ambito:
- En BuildVisitor se chequea que las clases, los atributos, los metodos y las declaraciones locales tengan identificadores distintos.

*Ningún identificador es declarado dos veces en un mismo bloque.
Mostramos un mensaje “REDEFINED” en el caso que ocurra esto.

```
public String visit(Field_decl expr) {
    SymbolTable aux;
    if (expr.getName() != null)
        for (Name c :expr.getName()){
            if (c.isArray()){
                if (c.getIntLiteral()<=0)
                    this.addError(c,"[index] must be greater than zero");
                aux = new SymbolTable(c.getId(), expr.getType(), true, c);
                this.incOffset(c.getIntLiteral());
                c.setOffset(offset+4);  // El fin del arreglo.
            }else{
                aux = new SymbolTable(c.getId(), expr.getType(), c);
                c.setOffset(offset);
                this.incOffset();
            }
            if (stack.getLast().search(aux)){   // Repeated checking
                this.addError(c, "Redefined");
            }else{
                stack.getLast().setSymbol(aux);
            }
        }
    return "";
}
```

*El <literal_integer> en la declaración de un arreglo debe ser mayor a cero (es la longitud del arreglo).

```
public String visit(Field_decl expr) {
    SymbolTable aux;
    if (expr.getName() != null)
        for (Name c :expr.getName()){
            if (c.isArray()){
                if (c.getIntLiteral()<=0)
                    this.addError(c,"[index] must be greater than zero");
                aux = new SymbolTable(c.getId(), expr.getType(), true, c);
                this.incOffset(c.getIntLiteral());
                c.setOffset(offset+4);  // El fin del arreglo.
            }else{
                aux = new SymbolTable(c.getId(), expr.getType(), c);
                c.setOffset(offset);
                this.incOffset();
            }
            if (stack.getLast().search(aux)){   // Repeated checking
                this.addError(c, "Redefined");
            }else{
                stack.getLast().setSymbol(aux);
            }
        }
    return "";
}
```

*Ningún identificador es usado antes de ser declarado.
Esto es controlado por el parser.

*El número y tipos de los argumentos en la invocación a un método debe ser iguales al número y tipos declarados en la definición del método.

```
public String visit(Method_call expr) {
    SymbolTable symbol;
    if (expr.isObjectCall())
        symbol = this.searchSymbol(expr.getId(), false);
    else
        symbol = this.searchSymbol(expr.getId(), true);
    if (symbol==null)
        this.addError(expr, "Undefined");
    else{
        if (expr.isObjectCall()){   // Si es una llamada a un objeto
            SymbolTable sym = this.searchClass(symbol.getType().toString());
            if (sym==null) // Si no existe la clase
                this.addError(expr, " Undefined type");
            else{
                Class_decl class_decl = (Class_decl) sym.getAst();
                Type methodType = class_decl.getMethodType(expr.getId_param());
                if (methodType==null) // Si el metodo no existe
                    this.addError(expr, "."+expr.getId_param()+" Undefined");
                else
                    expr.setClase(methodType.toString());
            }
```

*Le expresión en una sentencia return debe ser igual al tipo de retorno declarado para el método.

```
public Type visit(Block expr) {
    if (expr.getField_decl() != null)
        for (Field_decl f :  expr.getField_decl())
            if (f.accept(this).toString().compareTo("void")==0)
                this.addError(f, "Incorrect type");
    Type ret = null;
    if (expr.getStatement() != null)
        for (Statement c :expr.getStatement())
            if (c.getId()=="expr" || c.getId()=="void")// caso return expr; ó return;
                ret = c.accept(this); // asumo que las expr son distintas de void
            else{
                if (c.getId()=="break" || c.getId()=="continue")
                    if (whileOrFor==0)
                        this.addError(c, "Incorrect definition");   // no se deben definir aca
                c.accept(this);
            }
    return ret;
}
```

*Si la invocación a un método es usada como una expresión, el método debe retornar un resultado.

```
public Type visit(Method_decl m) {
    method_list.add(m);
    methodType = m.getType();
    if (!(m.getBody().isExtern())){
        Type aux = m.getBody().accept(this);
        if (aux==null){
            this.addError(m, "not found return"); // exigimos un return en bloque principal del metod
            return null;
        }
    }
    return null;
}
```

*Un <id> usado como un <location> debe estar declarado como un parámetro o como una variable local o global.

*La <expr> en una sentencia if o while debe ser bool.

```
public Type visit(Statement_if expr) {
    Type cond = expr.getExpr().accept(this);
    if (cond.toString().compareTo("bool")!=0)
        this.addError(expr, "Expected bool, not "+cond.toString());
    Statement aux = expr.getStatement();
    if (aux.getId()=="expr" || aux.getId()=="void")
        return aux.accept(this);
    if (aux.getId()=="break" || aux.getId()=="continue")
        if (whileOrFor==0)
            this.addError(aux, "Incorrect definition");     // no se
    aux.accept(this);
    return null;
}
```

```
public Type visit(Statement_while expr) {
    Type cond = expr.getExpr().accept(this);
    if (cond.toString().compareTo("bool")!=0)
        this.addError(expr, "Expected bool, not "+cond.toString());
    whileOrFor++;
    expr.getStatement().accept(this);
    whileOrFor--;
    return null;
}
```

*En toda locación de la forma <id> [<expr>]. El tipo de <expr> debe ser integer.

```
public Type visit(Location expr) {
    if (expr.isArray()){
        expr.getExpr().accept(this);
        Type type = new Type("integer");
        if (!(expr.getExpr().getType().equals(type)))
            this.addError(expr, "["+expr.getExpr().getType()+"]"+" expected integer");
    }
    return expr.getType();
}
```

*Los operandos de <arith_op>’s y <rel_op>’s deben ser de tipo integer o float. Cabe aclarar que nosotros dividimos estos operadores en binarios y unarios. Con esto chequeamos que tanto los operadores binarios como unarios tengan los tipos correctos.

```
public Type visit(Bin_op expr) {
    Type expr1 = expr.getExpr1().accept(this);
    Type expr2 = expr.getExpr2().accept(this);
    if (!(expr1.equals(expr2) && expr.typeOk(expr1)))
        this.addError(expr, "Error: "+expr1.toString()+expr.getOperacion()+expr2.toString());
    expr.setType(expr.inferType(expr1));
    return expr.getType();
}
```

```
public Type visit(Unary_op expr) {
    Type expr1 = expr.getExpr().accept(this);
    if (!expr.typeOk(expr1))
        this.addError(expr, "Error: "+expr.getOperacion()+expr1.toString());
    expr.setType(expr.inferType(expr1));
    return expr.getType();
}
```

*Los operandos de <eq_op> deben tener el mismo tipo (int, float, bool). Los operandos <arith_op> y <rel_op> deben ser float o int. Los operandos de <cond_op> y negación, deben ser bool.

```
public boolean typeOk(Type x){

    if (operacion == "+" || operacion == "-" || operacion == "*" || operacion == "/" || operacion == "%" ||
        operacion == "<" || operacion == ">" || operacion == "<=" || operacion == ">=")
        return (x.toString() == "integer" || x.toString() == "float");

    if (operacion == "==" || operacion == "!=")
        return (x.toString() == "integer" || x.toString() == "float" || x.toString() == "bool");

    if (operacion == "!" || operacion == "||" || operacion == "&&")
        return (x.toString() == "bool");

    if (operacion == "%")
        return (x.toString() == "integer");

    return false;
}
```

*La <location> y la <expr> en una asignación, <location> = <expr>, deben tener el mismo tipo.

```
public Type visit(Statement_asig expr) {
    Type loc = expr.getLocation().accept(this);
    Type exp = expr.getExpr().accept(this);
    if (!loc.equals(exp))
        this.addError(expr.getLocation(), loc.toString()+expr.getAsign_op().toString()+exp.toString());
    return loc;
}
```

* Las expresiones (<expr>) iniciales y finales de un for, deben ser de tipo integer.

```
public Type visit(Statement_for expr) {
    if (!((expr.getExpr1().accept(this).equals(expr.getExpr2().accept(this)) &&
            expr.getExpr1().accept(this).toString().compareTo("integer")==0) &&
            (expr.getType().toString().compareTo("integer")==0)))
        this.addError(expr, "Expected integer");
    whileOrFor++;
    expr.getStatement().accept(this);
    whileOrFor--;
    return null;
}
```

*Las sentencias break and continue solo pueden encontrarse en el cuerpo de un ciclo.

```
public Type visit(Block expr) {
    if (expr.getField_decl() != null)
        for (Field_decl f :  expr.getField_decl())
            if (f.accept(this).toString().compareTo("void")==0)
                this.addError(f, "Incorrect type");
    Type ret = null;
    if (expr.getStatement() != null)
        for (Statement c :expr.getStatement())
            if (c.getId()=="expr" || c.getId()=="void")// caso return expr; ó return;
                ret = c.accept(this); // asumo que las expr son distintas de void
            else{
                if (c.getId()=="break" || c.getId()=="continue")
                    if (whileOrFor==0)
                        this.addError(c, "Incorrect definition");   // no se deben definir aca
                c.accept(this);
            }
    return ret;
}
```

```
public Type visit(Statement_if expr) {
    Type cond = expr.getExpr().accept(this);
    if (cond.toString().compareTo("bool")!=0)
        this.addError(expr, "Expected bool, not "+cond.toString());
    Statement aux = expr.getStatement();
    if (aux.getId()=="expr" || aux.getId()=="void")
        return aux.accept(this);
    if (aux.getId()=="break" || aux.getId()=="continue")
        if (whileOrFor==0)
            this.addError(aux, "Incorrect definition");     // no se deben definir aca
    aux.accept(this);
    return null;
}

public Type visit(Statement_ifelse expr) {
    Type cond = expr.getExpr().accept(this);
    if (cond.toString().compareTo("bool")!=0)
        this.addError(expr, "Expected bool, not "+cond.toString());
    Statement aux1 = expr.getStatement1();
    Statement aux2 = expr.getStatement2();
    if (aux1.getId()=="break" || aux1.getId()=="continue")
        if (whileOrFor==0)
            this.addError(aux1, "Incorrect definition");        // no se deben definir aca
    if (aux2.getId()=="break" || aux2.getId()=="continue")
        if (whileOrFor==0)
            this.addError(aux2, "Incorrect definition");        // no se deben definir aca
    Type s1 = aux1.accept(this);
    Type s2 = aux2.accept(this);
    if (aux1.getId()=="expr" || aux1.getId()=="void")
        return s1;
    if (aux2.getId()=="expr" || aux2.getId()=="void")
        return s2;
    return null;
}
```

Generador de Código Intermedio:

Al comenzar esta etapa creamos la clase IntermediateCodeVisitor, y tuvimos que realizar dos tareas simultáneamente, nos encargamos de concluir y mejorar la etapa anterior de análisis semántico, en la que habían quedado un par de chequeos inconclusos, y al mismo tiempo hicimos la generación del código intermedio (código de tres direcciones), para la cual no tuvimos mayores inconvenientes. Mientras mas eficiente realizabamos esta etapa, menor era el esfuerzo en la etapa siguiente de Assembler, asi que tratamos de hacerla lo mejor posible, así el pasaje a código Assembler era casi directo.

Generador de Código Assembler:

En esta etapa creamos la clase AsemblerGenerator y con ello generamos código Assembler de 32 bits. El primer problema con el que nos encontramos fue que no teníamos el offset seteado de las etapas anteriores, entonces agregamos el offset en las correspondientes variables, y con ello pudimos comenzar con la generación de código objeto. Primero generamos el código que nos parecía mas sencillo como las operaciones matemáticas (sum,res,div,mul..), luego realizamos las sentencias condicionales (if,if-else), los ciclos (while,for) que fueron traducciones casi directas de nuestro código intermedio. Con los tests provistos por la cátedra logramos encontrar algunos errores, uno de ellos fue que nos faltaba la definición de mod (%) en nuestra gramática, otro error fue que no habiamos definido la negación en assembler, entre otros. Logramos que corran todos los tests de funciones, y con los resultados esperados, tomando como convención que al hacer llamadas a funciones,

debemos invocarlas en funciones auxiliares ya que nuestro compilador no soporta llamadas a funciones anidadas.