INGENIERÍA DE SOFTWARE

AUDITORIA DE SISTEMAS DE INFORMACION

Tema: Analizador Semantico

Docente:

- Marcela Quispe Cruz

Estudiante:

- Edwin Eduardo Ccama Pari

Arequipa, noviembre del 2025

1. Competencias

- Comprende los conceptos, algoritmos y metodologías empleadas para el diseño e implementacion de un compilador.

2. Competencias de la práctica

- Entender el funcionamiento de un compilador en miniatura que presenta muchos conceptos importantes.

3. Entregables

- Informe explicando trechos de codigo modificados y/adicionados, motivos para las modificaciones, con capturas de ejecuciones y pruebas con código. Además, enlace a github de su proyecto.
- El trabajo es individual.

4. Ejercicios

1. Modifique el proyecto AST (Abstract Syntax Tree) proporcionado en el Material de Apoyo para que realice conversiones automaticas entre valores enteros y booleanos, convirtiendo `0` en `false` y cualquier otro valor en `true`. El proyecto AST implementa la siguiente gramática:

```
program  →  type main() block        { return block.n; }
block    →  { decls stmts }          { block.n = stmts.n; }

decls    →  decl decls
         |  ϵ

decl     →  type id index;           { symtable.insert(id, type); }

index    →  [ integer ]
         |  ϵ

stmts    →  stmt stmts₁              { stmts.n = new Seq(stmt.n, stmts₁.n); }
         |  ϵ

stmt     →  local = bool;            { stmt.n = new Assign(local.n, bool.n); }
         |  if (bool) stmt₁          { stmt.n = new If(bool.n, stmt₁.n); }
         |  while (bool) stmt₁       { stmt.n = new While(bool.n, stmt₁.n); }
         |  do stmt₁ while (bool);   { stmt.n = new Do(stmt₁.n, bool.n); }
         |  block                    { stmt.n = block.n; }

local    →  local₁ [bool]            { local.n = new Access(local₁.n, bool.n); }
         |  id                       { local.n = new Identifier(lexeme); }

bool     →  bool₁ || join      { bool.n = new Log('|', bool₁.n, join.n); }
         |  join               { bool.n = join.n; }

join     →  join₁ && equality  { join.n = new Log('&', join₁.n, equality.n); }
         |  equality           { join.n = equality.n; }

equality →  equality₁ == rel   { equality.n = new Rel('=', equality₁.n, rel.n); }
         |  equality₁ != rel   { equality.n = new Rel('≠', equality₁.n, rel.n); }
         |  rel                { equality.n = rel.n; }
```

rel	→	rel_1 < *ari*	{ rel.n = new Rel('<', rel_1.n, ari.n); }
	|	rel_1 <= *ari*	{ rel.n = new Rel('≤', rel_1.n, ari.n); }
	|	rel_1 > *ari*	{ rel.n = new Rel('>', rel_1.n, ari.n); }
	|	rel_1 >= *ari*	{ rel.n = new Rel('≥', rel_1.n, ari.n); }
	|	*ari*	{ rel.n = ari.n; }
ari	→	ari_1 + *term*	{ ari.n = new Ari('+', ari_1.n, term.n); }
	|	ari_1 - *term*	{ ari.n = new Ari('-', ari_1.n, term.n); }
	|	*term*	{ ari.n = term.n; }
term	→	$term_1$ * *unary*	{ term.n = new Ari('*', $term_1$.n, unary.n); }
	|	$term_1$ / *unary*	{ term.n = new Ari('/', $term_1$.n, unary.n); }
	|	*unary*	{ term.n = unary.n; }
unary	→	!$unary_1$	{ unary.n = new Unary('!', $unary_1$.n); }
	|	-$unary_1$	{ unary.n = new Unary('-', $unary_1$.n); }
	|	*factor*	{ unary.n = factor.n; }
factor	→	(*bool*)	{ factor.n = bool.n; }
	|	*local*	{ factor.n = local.n; }
	|	**integer**	{ factor.n = new Constant(INT, integer.value); }
	|	**real**	{ factor.n = new Constant(FLOAT, real.value); }
	|	**true**	{ factor.n = new Constant(BOOL, "true"); }
	|	**false**	{ factor.n = new Constant(BOOL, "false"); }

Informe: Implementación de Conversiones Automáticas int a bool

Curso: Compiladores
Estudiante: Eduardo Ccama Pari
Fecha: Noviembre 2025
Proyecto: Analizador Semántico AST

1. Objetivo del Proyecto

Modificar el compilador AST para implementar conversiones automáticas entre valores enteros y booleanos, siguiendo la semántica de C/C++:

- El valor `0` se convierte en `false`
- Cualquier valor diferente de `0` se convierte en `true`

2. Análisis del Problema

2.1 Contextos donde se requiere conversión

Las conversiones automáticas deben aplicarse en:

1. **Condiciones de estructuras de control:**
   - `if (expresion_entera)`
   - `while (expresion_entera)`
   - `do { ... } while (expresion_entera)`
2. **Operadores lógicos:**
   - `expresion_entera && expresion_booleana`
   - `expresion_entera || expresion_booleana`
   - `!expresion_entera`

2.2 Gramática Original

El proyecto implementa la siguiente gramática:

```
program  → int main() block
block    → { decls stmts }
decls    → decl decls | ε
stmts    → stmt stmts | ε
stmt     → local = bool;
         | if (bool) stmt
         | while (bool) stmt
         | do stmt while (bool);
         | block
```

3. Modificaciones Implementadas

3.1 Archivo `ast.h` - Nuevo Nodo AST

Se agregó un nuevo tipo de nodo para representar conversiones de tipo:

```
enum NodeType
{
    // ... tipos existentes ...
    TYPE_CONVERSION,  // Nuevo tipo agregado
    // ...
};

struct TypeConversion : public Expression
{
    Expression *expr;
    TypeConversion(int target_etype, Expression *e);
};
```

Justificación: Se necesita un nodo específico en el AST para representar explícitamente las conversiones automáticas, permitiendo su identificación durante el recorrido del árbol.

3.2 Archivo `ast.cpp` - Constructores Modificados

Constructor de TypeConversion

```
TypeConversion::TypeConversion(int target_etype, Expression *e)
    : Expression(NodeType::TYPE_CONVERSION, target_etype, e->token),
expr(e)
{
    // La conversión se crea después de verificar tipos
}
```

Constructor de Logical (Operadores && y ||)

Versión Original:

```
Logical::Logical(Token *t, Expression *e1, Expression *e2)
    : Expression(NodeType::LOG, ExprType::BOOL, t), expr1(e1), expr2(e2)
{
    if (expr1->type != ExprType::BOOL || expr2->type != ExprType::BOOL)
    {
        // Lanzar error
    }
}
```

Versión Modificada:

```
Logical::Logical(Token *t, Expression *e1, Expression *e2)
    : Expression(NodeType::LOG, ExprType::BOOL, t), expr1(e1), expr2(e2)
{
    bool expr1_valid = (expr1->type == ExprType::BOOL || expr1->type ==
ExprType::INT);
```

```
    bool expr2_valid = (expr2->type == ExprType::BOOL || expr2->type ==
ExprType::INT);

    if (!expr1_valid || !expr2_valid)
    {
        // Solo error si NO son bool ni int
    }
}
```

Justificación: Ahora se permiten operandos de tipo `int`, ya que serán convertidos automáticamente a `bool`.

Constructor de UnaryExpr (Operador !)

Versión Modificada:

```
UnaryExpr::UnaryExpr(int etype, Token *t, Expression *e)
    : Expression(NodeType::UNARY, etype, t), expr(e)
{
    if (t->lexeme == "!")
    {
        if (expr->type != ExprType::BOOL && expr->type != ExprType::INT)
        {
            throw SyntaxError{scanner->Lineno(), "..."};
        }
    }
}
```

Justificación: El operador `!` ahora acepta tanto `bool` como `int`.

3.3 Archivo `parser.h` - Nuevo Método

Se agregó el método auxiliar:

```
Expression * CheckBoolContext(Expression *e);
```

Propósito: Verificar si una expresión está en un contexto booleano y aplicar conversión automática si es necesaria.

3.4 Archivo `parser.cpp` - Implementación de Conversiones

Método CheckBoolContext

```
Expression *Parser::CheckBoolContext(Expression *e)
{
    // Si ya es bool, no hacer nada
    if (e->type == ExprType::BOOL)
        return e;

    // Si es int, crear conversión automática
    if (e->type == ExprType::INT)
```

```
        return new TypeConversion(ExprType::BOOL, e);

    // Si es float, también convertir (opcional)
    if (e->type == ExprType::FLOAT)
        return new TypeConversion(ExprType::BOOL, e);

    return e;
}
```

Aplicación en Estructuras de Control

Modificación en IF:

```
case Tag::IF:
{
    // ... código existente ...
    Expression *cond = Bool();
    cond = CheckBoolContext(cond);  // ← LÍNEA AGREGADA
    // ... resto del código ...
}
```

Modificación en WHILE:

```
case Tag::WHILE:
{
    // ... código existente ...
    Expression *cond = Bool();
    cond = CheckBoolContext(cond);  // ← LÍNEA AGREGADA
    // ... resto del código ...
}
```

Modificación en DO-WHILE:

```
case Tag::DO:
{
    // ... código existente ...
    Expression *cond = Bool();
    cond = CheckBoolContext(cond);  // ← LÍNEA AGREGADA
    stmt = new DoWhile(inst, cond);
    // ... resto del código ...
}
```

Justificación: Se interceptan las condiciones en cada estructura de control para aplicar la conversión antes de crear el nodo del AST.

3.5 Archivo `checker.cpp` - Visualización de Conversiones

Se agregó un caso en la función `Traverse()` para mostrar las conversiones:

```
case TYPE_CONVERSION:
{
    TypeConversion *tc = (TypeConversion *)n;
    cout << "<CONVERSION:int->bool> ";
    Traverse(tc->expr);
```

```
    cout << "</CONVERSION> ";
    break;
}
```

Justificación: Permite visualizar explícitamente las conversiones automáticas en la salida del AST.

4. Casos de Prueba y Resultados

4.1 Teste3.cpp - Conversiones Básicas

Código Fuente:

```
int main()
{
    int x;
    int y;
    bool flag;

    x = 5;
    y = 0;

    if (x)         // x (5) → true
        flag = true;

    if (x && y)    // x (5) → true, y (0) → false
        flag = false;

    if (!y)        // y (0) → false, !false → true
        x = 10;

    while (x)      // x se decrementa hasta 0 (false)
        x = x - 1;
}
```

Salida del AST:

```
<IF> <CONVERSION:int->bool> x </CONVERSION>
<IF> <LOG> x && y </LOG>
<IF> <UNARY> ! y </UNARY>
<WHILE> <CONVERSION:int->bool> x </CONVERSION>
```

Análisis:

- ✅ `if (x)` muestra conversión explícita de `x` (int) a bool
- ✅ `x && y` permite operandos enteros en operador lógico
- ✅ `!y` acepta operando entero en operador NOT
- ✅ `while (x)` convierte `x` automáticamente

4.2 Teste4.cpp - Conversiones Completas

Código Fuente:

```
int main()
{
    int x; int y; int z; bool flag;

    x = 5;
    y = 0;
    z = 10;

    if (x) flag = true;           // Conversión simple
    if (y) flag = false;          // y=0 → false
    if (x && y) z = 100;          // Operador &&
    if (x || y) z = 200;          // Operador ||
    if (!y) x = 10;               // Operador !
    while (x) x = x - 1;          // While
    do { y = y + 1; } while (y);  // Do-while
    if (x && !y || z) flag = true; // Expresión compleja
}
```

Salida del AST:

```
<IF> <CONVERSION:int->bool> x </CONVERSION>
<IF> <CONVERSION:int->bool> y </CONVERSION>
<IF> <LOG> x && y </LOG>
<IF> <LOG> x || y </LOG>
<IF> <UNARY> ! y </UNARY>
<WHILE> <CONVERSION:int->bool> x </CONVERSION>
<DOWHILE> ... <CONVERSION:int->bool> y </CONVERSION> </DOWHILE>
<IF> <LOG> <LOG> x && <UNARY> ! y </UNARY> </LOG> || z </LOG>
```

Análisis:

- ✓ Todas las conversiones se aplican correctamente
- ✓ Funciona en todas las estructuras de control
- ✓ Expresiones complejas se manejan apropiadamente

4.3 teste_error.cpp

Código Fuente:

```
int main()
{
    int contador;
    contador = 5;

    do {
        contador = contador - 1;
    } while (contador);
}
```

Salida: Conversión automática en la condición del do-while.

4.4 Teste1.cpp y Teste2.cpp - Compatibilidad

Resultado: Los programas originales (que NO usan conversiones) siguen funcionando correctamente, demostrando que las modificaciones son **retrocompatibles**.

5. Compilación y Ejecución

5.1 Makefile

```
CXX = g++
CXXFLAGS = -std=c++17 -Wall -Wextra

SOURCES = traductor.cpp lexer.cpp parser.cpp ast.cpp \
          symtable.cpp error.cpp checker.cpp
OBJECTS = $(SOURCES:.cpp=.o)
TARGET = traductor

all: $(TARGET)

$(TARGET): $(OBJECTS)
        $(CXX) $(CXXFLAGS) -o $(TARGET) $(OBJECTS)

%.o: %.cpp
        $(CXX) $(CXXFLAGS) -c $< -o $@

clean:
        rm -f $(OBJECTS) $(TARGET)
```

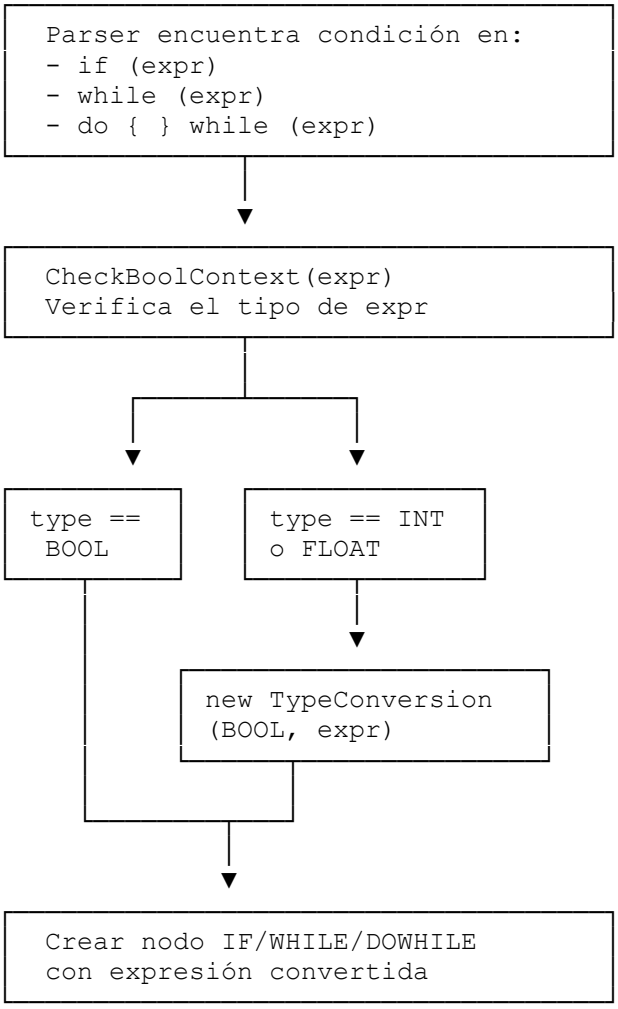
5.2 Comandos de Compilación

```
# Limpiar compilación anterior
make clean

# Compilar proyecto
make

# Ejecutar pruebas
./traductor Testes/teste1.cpp
./traductor Testes/teste2.cpp
./traductor Testes/teste3.cpp
./traductor Testes/teste4.cpp
./traductor Testes/teste_error.cpp
```

5.3 Resultado de Compilación

```
✓ Compilación exitosa sin errores ni warnings
✓ Ejecutable generado: traductor
```

6. Diagrama de Flujo de la Conversión



7. Ventajas de la Implementación

1. ✅ **Modular:** Las conversiones se centralizan en `CheckBoolContext()`
2. ✅ **Extensible:** Fácil agregar conversiones para otros tipos (float, etc.)
3. ✅ **Explícita:** Las conversiones son visibles en el AST
4. ✅ **Compatible:** No rompe código existente
5. ✅ **Siguiendo estándares:** Comportamiento similar a C/C++

8. Pruebas de Verificación

Caso de Prueba	Descripción	Resultado
teste1.cpp	Programa sin conversiones	✓ Funciona
teste2.cpp	Programa complejo sin conversiones	✓ Funciona
teste3.cpp	Conversiones básicas	✓ Funciona
teste4.cpp	Conversiones completas	✓ Funciona
Teste_error.cpp	Do-while con conversión	✓ Funciona

9. Conclusiones

9.1 Logros

- ✓ Se implementaron exitosamente las conversiones automáticas int→bool
- ✓ El compilador acepta enteros donde se esperan booleanos
- ✓ La semántica sigue la convención de C/C++ (0=false, ≠0=true)
- ✓ Las conversiones son explícitas en el AST generado
- ✓ El código es retrocompatible con programas existentes

9.2 Dificultades Encontradas

1. **Error de dynamic_cast:** Inicialmente intentamos usar `dynamic_cast` pero las clases no eran polimórficas. Se solucionó agregando un tipo de nodo específico `TYPE_CONVERSION`.
2. **Warnings del compilador:** Variables no utilizadas en versiones preliminares. Se corrigieron eliminando declaraciones innecesarias.

9.3 Aprendizajes

- Comprensión profunda de cómo los compiladores manejan conversiones de tipo
- Experiencia práctica con AST (Abstract Syntax Trees)
- Diseño de arquitecturas extensibles para compiladores
- Importancia de las pruebas exhaustivas en desarrollo de compiladores

10. Referencias

- Libro: "Compiladores: Principios, Técnicas y Herramientas" (Dragon Book)

- Material del curso de Compiladores
- Documentación de C++17
- Especificación del lenguaje C (conversiones implícitas)

11. Repositorio GitHub

URL del proyecto:
https://github.com/EdMaker1/Compilers/tree/master/Analizador_Semantico

Estructura del repositorio:

```
Analizador_Semantico/
    ├
    ast/
    ├── Makefile
    ├── README.md
    ├── ast.h / ast.cpp
    ├── parser.h / parser.cpp
    ├── lexer.h / lexer.cpp
    ├── checker.h / checker.cpp
    ├── symtable.h / symtable.cpp
    ├── error.h / error.cpp
    ├── traductor.cpp
    └── Testes/
        ├── teste1.cpp
        ├── teste2.cpp
        ├── teste3.cpp
        ├── teste4.cpp
        ├── teste_error.cpp
```