Compiladores
Laboratorio 7
Patrón de diseño Visitors

Objetivo

Desarrollar un intérprete para una calculadora empleando el patrón de diseño Visitor, lo que permitirá aplicar operaciones sobre el AST sin necesidad de modificar las clases que lo componen.

Programa

El script permite parsear programas que siguen la siguiente regla gramaticales:

- Exp = Term ((+|-) Term)*
- Term = Factor ((*|/) Factor)*
- Factor = Num | “(“ Exp “)”

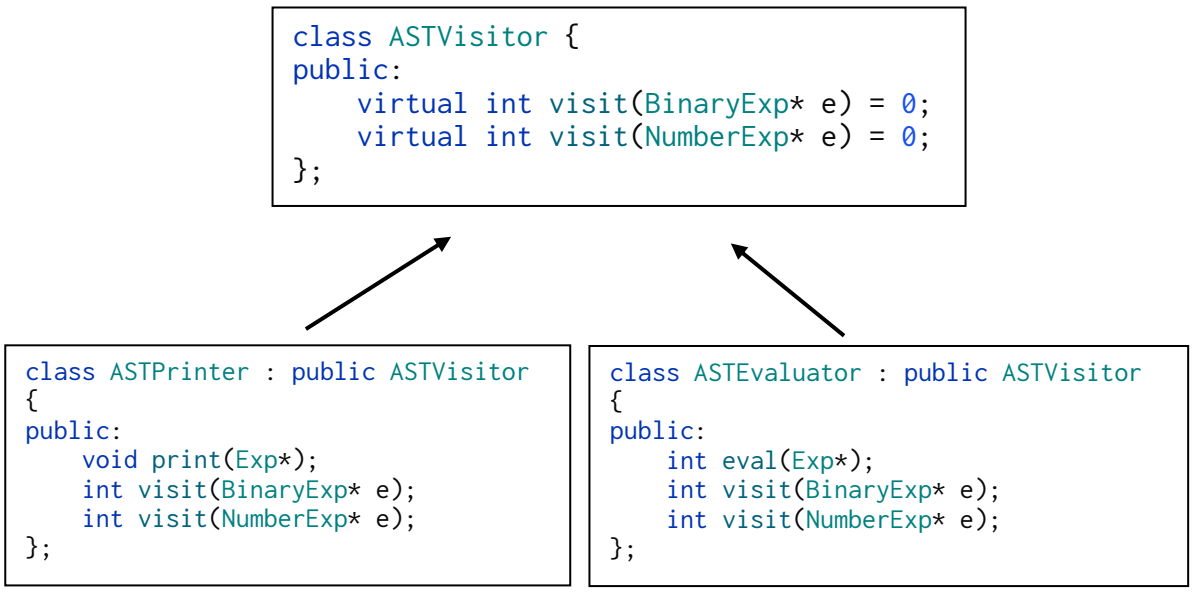
Problema 1

Implementar visitantes que permitan:

- Imprimir expresiones aritméticas.
- Evaluar expresiones aritméticas.

Solución

Implementar las clases:



Además, de la función virtual **accept** a las clases de expresiones:

```
class Exp {
public:
    virtual ~Exp() = 0;
    static char binopToChar(BinaryOp op);
    virtual int accept(ASTVisitor* v) = 0;
};
```

```
int BinaryExp::accept(ASTVisitor* v) {
    return v->visit(this);
}

int NumberExp::accept(ASTVisitor* v) {
    return v->visit(this);
}
```

Métodos ASTPrinter

```
void ASTPrinter::print(Exp* e) {
    cout << "expression: ";
    e->accept(this);
    cout << endl;
    return;
}

int ASTPrinter::visit(BinaryExp* e) {
    e->left->accept(this);
    cout << ' ' << Exp::binopToChar(e->op) << ' ';
    e->right->accept(this);
    return 0;
}

int ASTPrinter::visit(NumberExp* e) {
    cout << e->value;
    return 0;
}
```

Métodos ASTEvaluator

```
int ASTEvaluator::eval(Exp* e) {
    return e->accept(this);
}

int ASTEvaluator::visit(BinaryExp* e) {
    int v1 = e->left->accept(this);
    int v2 = e->right->accept(this);
    int result = 0;
    switch(e->op) {
        case PLUS: result = v1+v2; break;
        case MINUS: result = v1-v2; break;
        case MUL: result = v1 * v2; break;
        case DIV: result = v1 / v2; break;
    }
    return result;
}

int ASTEvaluator::visit(NumberExp* e) {
    return e->value;
}
```

Main

```
Exp *exp = parser.parse();
ASTPrinter printer;
printer.print(exp);
ASTEvaluator evaluator;
cout << evaluator.eval(exp);
```

Problema 2

Implementar **visitantes** para imprimir y evaluar para la gramática:

- Program ::= StmtList
- StmtList := Stmt ('**;**' Stmt)*
- Stmt ::= **id** '**=**' Exp | '**print**' '**(**' Exp '**)**'
- Exp ::= Term ((´**+**´ | ´**-**´) Term)*
- Term ::= Factor ((´*****´|´**/**´) Factor)*
- Factor ::= **id** | **Num** | ´**(**´ Exp ´**)**´

Con las siguientes clases

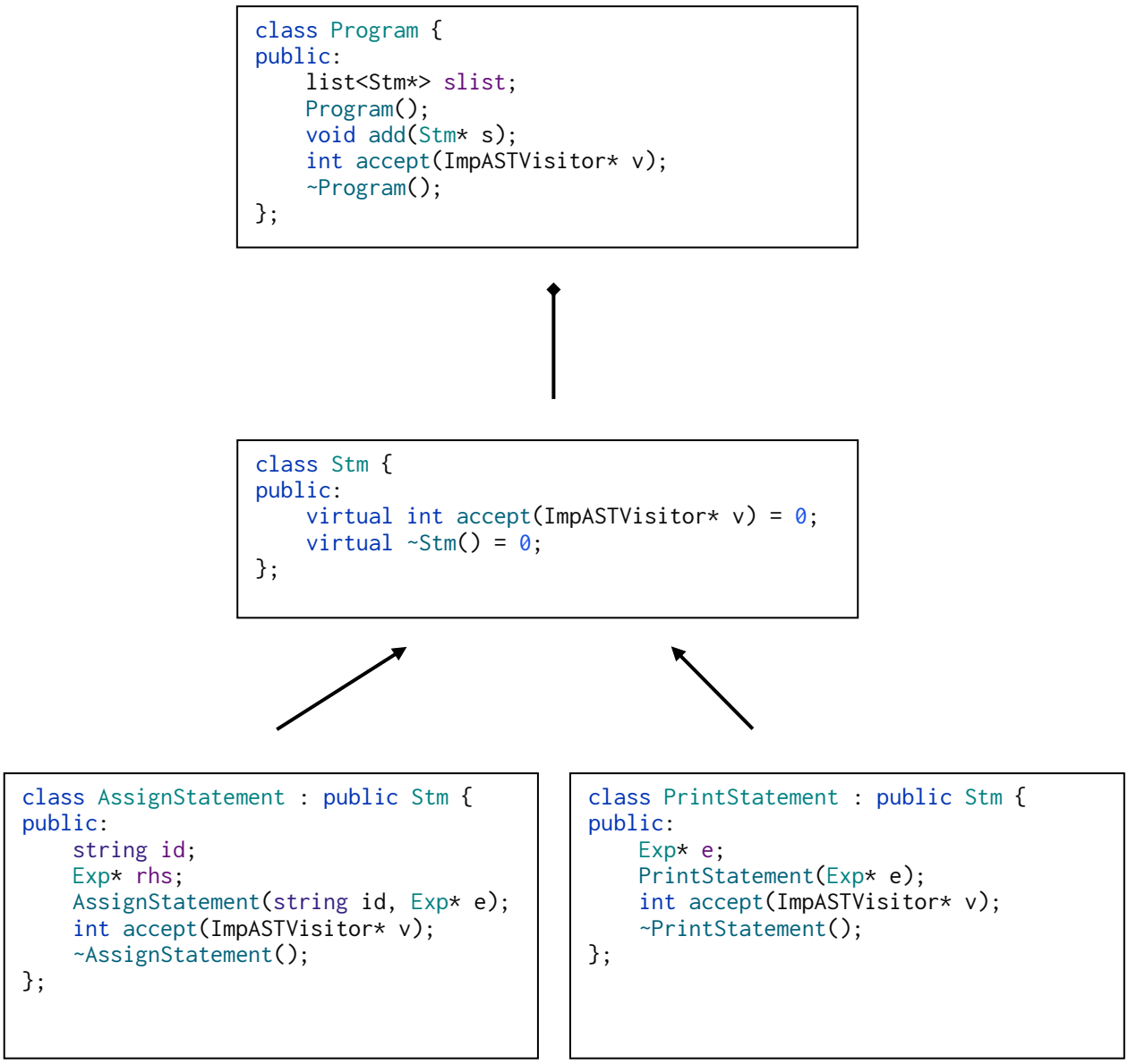
```
class Exp {
public:
    virtual int accept(ImpASTVisitor* v) = 0;
    static string binopToChar(BinaryOp op);
    virtual ~Exp() = 0;
};
```

```
class BinaryExp : public Exp {
public:
    Exp *left, *right;
    BinaryOp op;
    BinaryExp(Exp* l, Exp* r, BinaryOp op);
    int accept(ImpASTVisitor* v);
    ~BinaryExp();
};
```

```
class NumberExp : public Exp {
public:
    int value;
    NumberExp(int v);
    int accept(ImpASTVisitor* v);
    ~NumberExp();
};
```

```
class IdExp : public Exp {
public:
    string id;
    IdExp(string id);
    int accept(ImpASTVisitor* v);
    ~IdExp();
};
```

```
class ParenthExp : public Exp {
public:
    Exp *e;
    ParenthExp(Exp *e);
    int accept(ImpASTVisitor* v);
    ~ParenthExp();
};
```


class Program {
public:
list<Stm*> slist;
Program();
void add(Stm* s);
int accept(ImpASTVisitor* v);
~Program();
};
class Stm {
public:
virtual int accept(ImpASTVisitor* v) = 0;
virtual ~Stm() = 0;
};
class AssignStatement : public Stm {
public:
string id;
Exp* rhs;
AssignStatement(string id, Exp* e);
int accept(ImpASTVisitor* v);
~AssignStatement();
};
class PrintStatement : public Stm {
public:
Exp* e;
PrintStatement(Exp* e);
int accept(ImpASTVisitor* v);
~PrintStatement();
};

Parser

```
Program* Parser::parse() {
    current = scanner->nextToken();
    if (check(Token::ERR)) {
        exit(0);
    }
    Program* p = parseProgram();
    if (current->type != Token::END) {
        delete p;
        p = NULL;
    }

    if (current) delete current;
    return p;
}
```

```
Program* Parser::parseProgram() {
    Program* p = new Program();
    p->add(parseStatement());
    while(match(Token::SEMICOLON)) {
        p->add(parseStatement());
    }
    return p;
}
```

```
Stm* Parser::parseStatement() {
    Stm* s = NULL;
    Exp* e;
    if (match(Token::ID)) {
        string lex = previous->lexema;
        if (!match(Token::ASSIGN)) {
            exit(0);
        }
        s = new AssignStatement(lex, parseExpression());
    } else if (match(Token::PRINT)) {
        if (!match(Token::LPAREN)) {
            exit(0);
        }
        e = parseExpression();
        if (!match(Token::RPAREN)) {
            exit(0);
        }
        s = new PrintStatement(e);
    } else {
        exit(0);
    }
    return s;
}
```