真值表生成器实现

一个基于Web的真值表生成器，示例demo运行于[真值表生成器]

一、实现理论分析

1.用户输入待解析表达式，前端页面获取输入并请求后端

本项目使用**Go**语言开发，旨在构建一个web页面可用且可分享的真值表生成器。前端**JavaScript**代码会将用户输入的表达式动态更新向后端接口发包。

2.后端获取表达式并将表达式进一步解析为AST表达式树

什么是AST?

AST（**Abstract Syntax Tree**，抽象语法树）是计算机科学中用于表示编程语言语法结构的一种树形数据结构。在编译器、解释器等语言处理工具中，**AST** 被广泛应用于语法分析、语义分析和代码生成等阶段。

AST 是源代码的抽象表示，它将源代码转换为树形结构，每个节点代表源代码中的一个语法结构，如表达式、语句、函数、变量等。树的根节点表示整个源代码，而子节点表示源代码的各个部分。

在编程语言中，**AST** 的构建过程通常包括词法分析和语法分析。词法分析器负责将源代码字符串转换为一系列的 **Token**（标记），而语法分析器则根据这些 **Token** 构建 **AST**。

通过遍历 **AST**，可以进行语法分析、语义分析和代码生成等操作。编译器或解释器可以根据 **AST** 进行优化、错误检查和代码生成等工作，从而实现对源代码的分析和处理。

3.提取表达式中的变元个数，并生成不同的独立初始变元值，共2^n行真值取值情况

4.将AST树割裂为多个节点，用于后续遍历生成真值表

5.遍历AST树，生成真值表取值

6.将真值表处理为Json并返回数据到前端

7.前端根据返回的Json数据渲染真值表

二、实例代码实现

1.用户输入待解析表达式，前端页面获取输入并请求后端

```
<label for="expression">输入你的逻辑表达式😘</label>
```

```
<script>
    // 获取Json功能
    function fetchTruthTableData() {
        const exprInput =
encodeURIComponent(document.getElementById('expression').value);
        if (!exprInput) {
```

```
            clearTables();
            return;
        }
        fetch('/api/data?exp=' + exprInput) //使用用户输入表达式
            .then(function (response) {
                if (!response.ok) {
                    throw new Error("There was a problem with the fetch
operation: " + response.statusText);
                }
                return response.json(); // 解析Json数据流
            })
            .then(function (data) {
                var tableId = data.length > 10 ? "truthTableRight" :
"truthTableLeft"; // 判断表单区域
                displayTruthTable(data, tableId);
            })
            .catch(function (error) {
                console.error(error);
            });
    }

    // 清除表单
    function clearTables() {
        document.getElementById("truthTableLeft").innerHTML = "";
        document.getElementById("truthTableRight").innerHTML = "";
    }
    function replaceLogicSymbols(expression) {
        return expression.replace(/!/g, '¬')
            .replace(/&/g, '∧')
            .replace(/\|/g, '∨');
    }
    // 监听器，监测到数据变化时调用fetchTruthTableData函数
    document.getElementById('expression').addEventListener('input',
fetchTruthTableData);

    // 初始化表单
    fetchTruthTableData();
</script>
```

后端分词解析器

```
func Lexer(input string) []Token {
    var tokens []Token
    var currentVar string
    //判断字符是否为标准变元
    isVarChar := func(ch rune) bool {
       return (ch >= 'a' && ch <= 'z') || (ch >= 'A' && ch <= 'Z') || (ch >= '0'
&& ch <= '9')
    }
    //定义匿名函数用于添加变元
    commitVar := func() {
       if currentVar != "" {
          tokens = append(tokens, Token{Type: TokenTypeVar, Value: currentVar})
          currentVar = ""
       }
```

```
    }

    for _, ch := range input {
        switch {
        case isVarChar(ch):
            currentVar += string(ch)
        case ch == '&':
            commitVar()
            tokens = append(tokens, Token{Type: TokenTypeAnd})
        case ch == '|':
            commitVar()
            tokens = append(tokens, Token{Type: TokenTypeOr})
        case ch == '!':
            commitVar()
            tokens = append(tokens, Token{Type: TokenTypeNot})
        case ch == '→':
            commitVar()
            tokens = append(tokens, Token{Type: TokenTypeImplies})
        case ch == '⇔':
            commitVar()
            tokens = append(tokens, Token{Type: TokenTypeEquivalent})
        case ch == '(':
            commitVar()
            tokens = append(tokens, Token{Type: TokenTypeLparen})
        case ch == ')':
            commitVar()
            tokens = append(tokens, Token{Type: TokenTypeRparen})
        default:
            // 忽略其他符号
        }
    }

    commitVar() // 提交最后一个变元
    return tokens
}
```

2.后端获取表达式并将表达式进一步解析为AST表达式树

```
func ExpressHandler(c *gin.Context) {
    expression := c.Query("exp") //获取表达式参数
    tokens := Lexer(expression)  //解析表达式为Token块
    var pos int
    ast, err := parseEquivalent(tokens, &pos)
    printAST(ast, 0)
    if err != nil {
        fmt.Println("Error parsing expression:", err)
        c.JSON(http.StatusBadRequest, gin.H{"error": err.Error()})
        return
    }
    variables := ExtractVariables(expression)           //提取所有的变量
    truthTable, err := truthTableToJson(ast, variables) // 修改了函数名
    if err != nil {
        fmt.Println("Error generating truth table:", err)
        c.JSON(http.StatusInternalServerError, gin.H{"error": err.Error()}) // 修改
了错误消息
```

```
        return
    }
    c.Header("Content-Type", "application/json")
    c.JSON(http.StatusOK, truthTable)
}
```

AST解析器函数

```
func parsePrimary(tokens []Token, pos *int) (Expr, error) {
    if *pos >= len(tokens) {
        return nil, fmt.Errorf("unexpected end of expression")
    }

    token := tokens[*pos]
    switch token.Type {
    case TokenTypeVar:
        *pos++
        return Var{Name: token.Value}, nil
    case TokenTypeLparen:
        *pos++
        expr, err := parseEquivalent(tokens, pos) // 从最低优先级的操作符‘等值’开始
        if err != nil {
            return nil, err
        }
        if *pos >= len(tokens) || tokens[*pos].Type != TokenTypeRparen {
            return nil, fmt.Errorf("expected ')' at position %d", *pos)
        }
        *pos++
        return expr, nil
    default:
        return nil, fmt.Errorf("unexpected token '%s' at position %d",
token.Value, *pos)
    }
}

func parseNot(tokens []Token, pos *int) (Expr, error) {
    if *pos < len(tokens) && tokens[*pos].Type == TokenTypeNot {
        *pos++
        expr, err := parsePrimary(tokens, pos)
        if err != nil {
            return nil, err
        }
        return Not{Operand: expr}, nil
    }
    return parsePrimary(tokens, pos)
}

func parseAnd(tokens []Token, pos *int) (Expr, error) {
    expr, err := parseNot(tokens, pos)
    if err != nil {
        return nil, err
    }

    for *pos < len(tokens) && tokens[*pos].Type == TokenTypeAnd {
        *pos++
```

```
        rightExpr, err := parseNot(tokens, pos)
        if err != nil {
            return nil, err
        }
        expr = And{Left: expr, Right: rightExpr}
    }

    return expr, nil
}

func parseOr(tokens []Token, pos *int) (Expr, error) {
    expr, err := parseAnd(tokens, pos)
    if err != nil {
        return nil, err
    }

    for *pos < len(tokens) && tokens[*pos].Type == TokenTypeOr {
        *pos++
        rightExpr, err := parseAnd(tokens, pos)
        if err != nil {
            return nil, err
        }
        expr = Or{Left: expr, Right: rightExpr}
    }

    return expr, nil
}

func parseImplies(tokens []Token, pos *int) (Expr, error) {
    expr, err := parseOr(tokens, pos)
    if err != nil {
        return nil, err
    }

    for *pos < len(tokens) && tokens[*pos].Type == TokenTypeImplies {
        *pos++
        rightExpr, err := parseImplies(tokens, pos) // 递归调用parseImplies来处理蕴涵
的右结合性
        if err != nil {
            return nil, err
        }
        expr = Implies{Left: expr, Right: rightExpr}
    }

    return expr, nil
}

// 步骤6:处理等价(⇔)
func parseEquivalent(tokens []Token, pos *int) (Expr, error) {
    expr, err := parseImplies(tokens, pos) //从蕴涵开始解析，因为它具有更高的优先级
    if err != nil {
        return nil, err
    }
    //此循环将处理等价表达式链(例如，a⇔b⇔c)
    for *pos < len(tokens) && tokens[*pos].Type == TokenTypeEquivalent {
        *pos++ //移过'⇔'符号
```

```
        //解析等价的右边。注意这一点很重要
        //我们再次调用parseImplies以确保正确的结合性和优先级。
        //这意味着'a⇔b⇔c'被视为'a⇔(b⇔c)'。
        rightExpr, err := parseImplies(tokens, pos)
        if err != nil {
            return nil, err
        }

        //将当前表达式与新的右侧组合成一个新的等效表达式。
        expr = Equivalent{Left: expr, Right: rightExpr}
    }

    return expr, nil
}
```

3.提取表达式中的变元个数，并生成不同的独立初始变元值，共2^n行真值取值情况

```
// 提取表达式中的变量（递归地处理复合表达式）
func extractVariablesFromExpr(expr Expr, parentVars []string) []string {
    var variables []string
    // 根据表达式类型递归提取变量
    switch e := expr.(type) {
    case And:
        variables = append(variables, extractVariablesFromExpr(e.Left,
parentVars)...)
        variables = append(variables, extractVariablesFromExpr(e.Right,
parentVars)...)
    case Or:
        variables = append(variables, extractVariablesFromExpr(e.Left,
parentVars)...)
        variables = append(variables, extractVariablesFromExpr(e.Right,
parentVars)...)
    case Not:
        variables = append(variables, extractVariablesFromExpr(e.Operand,
parentVars)...)
    case Var:
        variables = append(variables, e.Name)
    }
    // 确保变量列表中的变量是唯一的
    variables = append(variables, parentVars...)
    sort.Strings(variables)
    return removeDuplicates(variables)
}
```

```
func evaluateExpressionAndSubexpressions(expr Expr, combination map[string]bool)
map[string]int {
    result := make(map[string]int)

    // 计算当前表达式的值
    if expr.Evaluate(combination) {
        result[expr.String()] = 1
```

```
    } else {
        result[expr.String()] = 0
    }

    // 根据表达式类型，递归处理子表达式
    switch e := expr.(type) {
    case And:
        mergeResults(result, evaluateExpressionAndSubexpressions(e.Left,
combination))
        mergeResults(result, evaluateExpressionAndSubexpressions(e.Right,
combination))
    case Or:
        mergeResults(result, evaluateExpressionAndSubexpressions(e.Left,
combination))
        mergeResults(result, evaluateExpressionAndSubexpressions(e.Right,
combination))
    case Not:
        mergeResults(result, evaluateExpressionAndSubexpressions(e.Operand,
combination))
    case Implies:
        mergeResults(result, evaluateExpressionAndSubexpressions(e.Left,
combination))
        mergeResults(result, evaluateExpressionAndSubexpressions(e.Right,
combination))
    case Equivalent:
        mergeResults(result, evaluateExpressionAndSubexpressions(e.Left,
combination))
        mergeResults(result, evaluateExpressionAndSubexpressions(e.Right,
combination))
    }

    return result
}
```

4.将AST树割裂为多个节点，用于后续遍历生成真值表

5.遍历AST树，生成真值表取值

```
func evaluateExpressionAndSubexpressions(expr Expr, combination map[string]bool)
map[string]int {
    result := make(map[string]int)

    // 计算当前表达式的值
    if expr.Evaluate(combination) {
        result[expr.String()] = 1
    } else {
        result[expr.String()] = 0
    }

    // 根据表达式类型，递归处理子表达式
    switch e := expr.(type) {
    case And:
        mergeResults(result, evaluateExpressionAndSubexpressions(e.Left,
combination))
```

```
        mergeResults(result, evaluateExpressionAndSubexpressions(e.Right,
combination))
    case Or:
        mergeResults(result, evaluateExpressionAndSubexpressions(e.Left,
combination))
        mergeResults(result, evaluateExpressionAndSubexpressions(e.Right,
combination))
    case Not:
        mergeResults(result, evaluateExpressionAndSubexpressions(e.Operand,
combination))
    case Implies:
        mergeResults(result, evaluateExpressionAndSubexpressions(e.Left,
combination))
        mergeResults(result, evaluateExpressionAndSubexpressions(e.Right,
combination))
    case Equivalent:
        mergeResults(result, evaluateExpressionAndSubexpressions(e.Left,
combination))
        mergeResults(result, evaluateExpressionAndSubexpressions(e.Right,
combination))
    }

    return result
}
```

6.将真值表处理为Json并返回数据到前端

```
func truthTableToJson(expr Expr, variables []string) (string, error) {
    var totalCombinations int = int(math.Pow(2, float64(len(variables))))
    results := make([]map[string]int, 0)

    for i := 0; i < totalCombinations; i++ {
        var combination = make(map[string]bool)
        for j, variable := range variables {
            // 根据当前索引i和变量的位置j计算每个变量的布尔值
            combination[variable] = (i>>j)&1 == 1
        }
        // 对当前的变量组合求值，并获取表达式及其子表达式的结果
        result := evaluateExpressionAndSubexpressions(expr, combination)
        results = append(results, result)
    }

    // 将结果序列化为JSON字符串
    jsonResults, err := json.Marshal(results)
    if err != nil {
        return "", fmt.Errorf("error marshalling results to JSON: %v", err)
    }
    return string(jsonResults), nil
}
```

7.前端根据返回的Json数据渲染真值表

```
function displayTruthTable(data, tableId) {
        var truthTable = JSON.parse(data); // 解析Json
        var table = document.getElementById(tableId);

        // 清除真值表
        table.innerHTML = "";

        // 创建表头
        var headerRow = document.createElement("tr");
        var headerKeys = Object.keys(truthTable[0]);
        headerKeys.forEach(function (key) {
            var th = document.createElement("th");
            th.textContent = replaceLogicSymbols(key);
            headerRow.appendChild(th);
        });
        table.appendChild(headerRow);

        // 创建表身
        var tableBody = document.createElement("tbody");
        truthTable.forEach(function (row) {
            var tr = document.createElement("tr");
            headerKeys.forEach(function (key) {
                var td = document.createElement("td");
                td.textContent = row[key];
                tr.appendChild(td);
            });
            tableBody.appendChild(tr);
        });
        table.appendChild(tableBody);

        // 添加渐变动画
        table.classList.add('loaded');
    }
```

8.启动项目

```
func main() {
    r.GET("/api/data", ExpressHandler)
    r.LoadHTMLFiles("index.html")
    r.GET("/", func(c *gin.Context) {
       c.HTML(200, "index.html", 0)
    })
    r.Run(":8000")
}
```

三、项目快速部署

1.本地IDE启动项目

进入项目文件夹，使用

```
go run main.go
```

非常简单的启动项目

2.编译可执行文件

```
$env:GOOS = "linux"//操作系统根据实际选择
$env:GOARCH = "amd64"//架构根据实际选择
go build -o app//编译出名为app的可执行文件
```

可以编译出可执行文件（供服务器使用），双击即可启动项目，适合小白食用。