Министерство науки и высшего образования Российской Федерации
Федеральное государственное бюджетное образовательное учреждение высшего образования
«Московский государственный технический университет имени Н.Э. Баумана
(национальный исследовательский университет)» (МГТУ им. Н.Э. Баумана)

ФАКУЛЬТЕТ «Информатика и системы управления» (ИУ)

КАФЕДРА «Теоретическая информатика и компьютерные технологии» (ИУ9)

ЛАБОРАТОРНАЯ РАБОТА №2

«Синтаксические деревья»

Вариант 4

Выполнила:

студентка группы ИУ9-61Б

Бойко Маргарита Сергеевна

Москва, 2021

СОДЕРЖАНИЕ

Цель работы

Целью данной работы является изучение представления синтаксических деревьев в памяти компьютера и приобретение навыков преобразования синтаксических деревьев.

Исходные данные

В качестве исходного языка и языка реализации программы преобразования синтаксических деревьев выберем язык Go. Пакеты `"go/token"`, `"go/ast"` и `"go/parser"` из стандартной библиотеки этого языка содержат готовый <front-end> компилятора языка Go, а пакет `"go/format"` восстанавливает исходный текст программы по её синтаксическому дереву. Документацию по этим пакетам можно посмотреть по адресу https://golang.org/pkg/go/.

Построение синтаксического дерева по исходному тексту программы выполняется функцией `parser.ParseFile`, возвращающей указатель типа `*ast.File` на корень дерева.

Синтаксические деревья в памяти представляются значениями структур из пакета `"go/ast"`. Изучать синтаксические деревья удобно по их листингам, попрождаемым функцией `ast.Fprint`. Небольшая программа `astprint`, которая, ко всему прочему, демонстрирует вызов парсера для построения синтаксического дерева программы, представлена на листинге 1.

Напомним, что для компиляции программы `astprint` нужно выполнить команду

```
go build astprint.go
```

Обход синтаксического дерева в глубину реализован в функции `ast.Inspect`, которая вызывает переданную ей в качестве параметра функцию для каждого посещённого узла дерева. С помощью этой функции удобно

осуществлять поиск узлов определённого типа в дереве. Например, представленная на листинге 2 функция `insertHello` выполняет поиск всех операторов `if` в дереве и вставляет в начало положительной ветки каждого найденного оператора печать строки `"hello"`.

Восстановление исходного текста программы из синтаксического дерева осуществляется функцией `format.Node`. Эта функция не обращает внимания на координаты узлов дерева, выполняя полное переформатирование текста программы, поэтому при преобразовании дерева координаты новых узлов прописывать не нужно.

```
package main

import (
    "fmt"
    "go/ast"
    "go/parser"
    "go/token"
    "os"
)

func main() {
    if len(os.Args) != 2 {
        fmt.Printf("usage: astprint <filename.go>\n")
        return
    }

    // Создаём хранилище данных об исходных файлах
    fset := token.NewFileSet()

    // Вызываем парсер
    if file, err := parser.ParseFile(
        fset,                 // данные об исходниках
        os.Args[1],           // имя файла с исходником программы
        nil,                  // пусть парсер сам загрузит исходник
```

```
        parser.ParseComments, // приказываем сохранять комментарии
    ); err == nil {
        // Если парсер отработал без ошибок, печатаем дерево
        ast.Fprint(os.Stdout, fset, file, nil)
    } else {
        // в противном случае, выводим сообщение об ошибке
        fmt.Printf("Error: %v", err)
    }
}
```

Листинг 1. Исходный текст программы astprint.go

```
func insertHello(file *ast.File) {
    // Вызываем обход дерева, начиная от корня
    ast.Inspect(file, func(node ast.Node) bool {
        // Для каждого узла дерева
        if ifStmt, ok := node.(*ast.IfStmt); ok {
            // Если этот узел имеет тип *ast.IfStmt,
            // добавляем в начало массива операторов
            // положительной ветки if'а новый оператор
            ifStmt.Body.List = append(
                []ast.Stmt{
                    // Новый оператор — выражение
                    &ast.ExprStmt {
                        // Выражение — вызов функции
                        X: &ast.CallExpr {
                            // Функция — "fmt.Printf"
                            Fun: &ast.SelectorExpr {
                                X:   ast.NewIdent("fmt"),
                                Sel: ast.NewIdent("Printf"),
                            },
                            // Её параметр — строка "hello"
                            Args: []ast.Expr {
                                &ast.BasicLit {
                                    Kind:  token.STRING,
                                    Value: "\"hello\"",
                                },
```

```
                            },
                        },
                    },
                },
                ifStmt.Body.List...,
            )
        }
        // Возвращая true, мы разрешаем выполнять обход
        // дочерних узлов
        return true
    })
}
```

Листинг 2. Исходный текст функции insertHello

Задание

Выполнение лабораторной работы состоит из нескольких этапов:

1. Подготовка исходного текста демонстрационной программы, которая в дальнейшем будет выступать в роли объекта преобразования (демонстрационная программа должна размещаться в одном файле и содержать функцию `main`).
2. Компиляция и запуск программы `astprint` для изучения структуры синтаксического дерева демонстрационной программы.
3. Разработка программы, осуществляющей преобразование синтаксического дерева и порождение по нему новой программы.
4. Тестирование работоспособности разработанной программы на исходном тексте демонстрационной программы.

Преобразование синтаксического дерева в программный код должно вносить новую возможность: подсчет, сколько раз в ходе работы программы были вызваны сопрограммы.

Реализация

На листинге 3 представлена функция `countIncOnGo`, производящая инкремент счетчика вызванных сопрограмм на каждый вызов сопрограммы.

```
func countIncOnGo(file *ast.File, counterName string) {
  ast.Inspect(file, func(node ast.Node) bool {

     if blockStmt, ok := node.(*ast.BlockStmt); ok {
        for i := range blockStmt.List {
           if _, ok := blockStmt.List[i].(*ast.GoStmt); ok {
              newList := append(blockStmt.List[:i+1], &ast.IncDecStmt{
                 X: &ast.Ident{
                    Name: counterName,
                    Obj:  nil,
                 },
                 Tok: token.INC,
              })
              blockStmt.List = append(newList, blockStmt.List[i:]...)
           }
        }
     }

     return true
  })
}
```

Листинг 3. Исходный текст функции countIncOnGo

На листинге 4 представлена функция `insertCounterPrint`, которая выводит значение счетчика.

```
func insertCounterPrint(file *ast.File, counterName string) {
  isCurrentNodeMain := false
  ast.Inspect(file, func(node ast.Node) bool {
     if isCurrentNodeMain {
        if block, ok := node.(*ast.BlockStmt); ok {
           block.List = append(
```

```
                block.List, []ast.Stmt{
                    &ast.ExprStmt{
                        X: &ast.CallExpr{
                            Fun: &ast.SelectorExpr{
                                X:   ast.NewIdent("fmt"),
                                Sel: ast.NewIdent("Printf"),
                            },
                            Args: []ast.Expr{
                                &ast.BasicLit{
                                    Kind:  token.INT,
                                    Value: counterName,
                                },
                            },
                        },
                    },
                }...,
            )
            isCurrentNodeMain = false
        }
    }
    if ident, ok := node.(*ast.Ident); ok {
        if ident.Name == "main" {
            isCurrentNodeMain = true
        }
    }
    return true
  })
}
```

Листинг 4. Исходный текст функции insertCounterPrint

Выводы

В результате выполнения лабораторной работы был получен навык построения и преобразования синтаксических деревьев исходного текста программы. Была реализована программа, которая осуществляет преобразование синтаксического дерева в программный код с подсчетом количества вызванных сопрограмм.