Лабораторная работа № 2.1. Синтаксические деревья

6 марта 2024 г.

Илья Афанасьев, ИУ9-61Б

Цель работы

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

Индивидуальный вариант

Подсчёт общего количества итераций всех циклов в процессе выполнения программы.

Реализация

Демонстрационная программа:

```
package main

import "fmt"

func foo() {
    strs := []string{"foo", "bar", "baz"}
    for k, str := range strs {
        fmt.Printf("Range-loop in another function: %d, %s\n", k, str)
    }
}

func main() {
    var n int

    if _, err := fmt.Scan(&n); err != nil {
        panic(err)
```

```
    }

    func() {
        for i := 0; i < 10; i++ {
            fmt.Printf("Anonymous function loop with pre-known iterations: %d\n", i)
        }
    }()

    for i := 0; i < n; i++ {
        for j := 0; j < n; j++ {
            fmt.Printf("Nested loop with user defined iterations: %d, %d\n", i, j)
        }
    }

    foo()

  counter := 5
  for {
    fmt.Printf("A loop with no init, cond and post statements: %d\n", counter)

    counter--
    if counter < 0 {
      break
    }
  }
}
```

Программа, осуществляющая преобразование синтаксического дерева:

```
package main

import (
    "fmt"
    "os"

    "go/ast"
    "go/format"
    "go/parser"
    "go/token"

  "github.com/rs/xid"
)

func getUniqueVarName() string {
    return "counter_" + xid.New().String()
}
```

```
func insertIntVar(file *ast.File, name string, value int) {
    var before, after []ast.Decl

    if len(file.Decls) > 0 {
        hasImport := false
        if genDecl, ok := file.Decls[0].(*ast.GenDecl); ok {
            hasImport = genDecl.Tok == token.IMPORT
        }

        if hasImport {
            before, after = []ast.Decl{file.Decls[0]}, file.Decls[1:]
        } else {
            after = file.Decls
        }
    }

    file.Decls = append(before,
        &ast.GenDecl{
            Tok: token.VAR,
            Specs: []ast.Spec{
                &ast.ValueSpec{
                    Names: []*ast.Ident{ast.NewIdent(name)},
                    Type:  ast.NewIdent("int"),
                    Values: []ast.Expr{
                        &ast.BasicLit{
                            Kind:  token.INT,
                            Value: fmt.Sprintf("%d", value),
                        },
                    },
                },
            },
        },
    )
    file.Decls = append(file.Decls, after...)
}

func findFuncDeclByName(file *ast.File, name string) *ast.FuncDecl {
    for _, decl := range file.Decls {
        if funcDecl, ok := decl.(*ast.FuncDecl); ok {
            if funcDecl.Name.Name == name {
                return funcDecl
            }
        }
    }

    return nil
```

```
}

func insertCounterPrinting(file *ast.File, name string) {
    mainFuncDecl := findFuncDeclByName(file, "main")
    if mainFuncDecl == nil {
        panic("Expected main function in the transformed file")
    }

    mainFuncDecl.Body.List = append(
        mainFuncDecl.Body.List,
        &ast.ExprStmt{
            X: &ast.CallExpr{
                Fun: &ast.SelectorExpr{
                    X:   ast.NewIdent("fmt"),
                    Sel: ast.NewIdent("Printf"),
                },
                Args: []ast.Expr{
                    &ast.BasicLit{
                        Kind:  token.STRING,
                        Value: "\"Total loop iterations count: %d\\n\"",
                    },
                    &ast.Ident{
                        Name: name,
                    },
                },
            },
        },
    )
}

func getVarIncStmt(name string) *ast.IncDecStmt {
    return &ast.IncDecStmt{
        X: &ast.Ident{
            Name: name,
        },
        Tok: token.INC,
    }
}

func addLoopIterationCount(file *ast.File) {
    counterName := getUniqueVarName()
    insertIntVar(file, counterName, 0)
    insertCounterPrinting(file, counterName)

    ast.Inspect(file, func(node ast.Node) bool {
        switch x := node.(type) {
```

```
        case *ast.ForStmt:
            x.Body.List = append(x.Body.List, getVarIncStmt(counterName))
        case *ast.RangeStmt:
            x.Body.List = append(x.Body.List, getVarIncStmt(counterName))
        }

        return true
    })
}

func main() {
    if len(os.Args) != 2 {
        fmt.Println("usage: transform <filename.go>")
        return
    }

    fset := token.NewFileSet()
    if file, err := parser.ParseFile(fset, os.Args[1], nil, parser.ParseComments);
            err == nil {
        addLoopIterationCount(file)

        if format.Node(os.Stdout, fset, file) != nil {
            fmt.Printf("Formatter error: %v\n", err)
        }
    } else {
        fmt.Printf("Errors in %s\n", os.Args[1])
    }
}
```

Тестирование

Результат трансформации демонстрационной программы:

```
package main

import "fmt"

var counter_cnk56nse8dehqpc44r5g int = 0

func foo() {
    strs := []string{"foo", "bar", "baz"}
    for k, str := range strs {
        fmt.Printf("Range-loop in another function: %d, %s\n", k, str)
        counter_cnk56nse8dehqpc44r5g++
    }
```

```
}

func main() {
    var n int

    if _, err := fmt.Scan(&n); err != nil {
        panic(err)
    }

    func() {
        for i := 0; i < 10; i++ {
            fmt.Printf("Anonymous function loop with pre-known iterations: %d\n", i)
            counter_cnk56nse8dehqpc44r5g++
        }
    }()

    for i := 0; i < n; i++ {
        for j := 0; j < n; j++ {
            fmt.Printf("Nested loop with user defined iterations: %d, %d\n", i, j)
            counter_cnk56nse8dehqpc44r5g++
        }
        counter_cnk56nse8dehqpc44r5g++
    }

    foo()

    counter := 5
    for {
        fmt.Printf("A loop with no init, cond and post statements: %d\n", counter)

        counter--
        if counter < 0 {
            break
        }
        counter_cnk56nse8dehqpc44r5g++
    }
    fmt.Printf("Total loop iterations count: %d\n", counter_cnk56nse8dehqpc44r5g)
}
```

Вывод тестового примера на `stdout` при входном параметре 4:

```
A loop in an anonymous function with pre-known iterations: 0
A loop in an anonymous function with pre-known iterations: 1
A loop in an anonymous function with pre-known iterations: 2
A loop in an anonymous function with pre-known iterations: 3
A loop in an anonymous function with pre-known iterations: 4
A loop in an anonymous function with pre-known iterations: 5
```

```
A loop in an anonymous function with pre-known iterations: 6
A loop in an anonymous function with pre-known iterations: 7
A loop in an anonymous function with pre-known iterations: 8
A loop in an anonymous function with pre-known iterations: 9
Nested loop with user defined iterations: 0, 0
Nested loop with user defined iterations: 0, 1
Nested loop with user defined iterations: 0, 2
Nested loop with user defined iterations: 0, 3
Nested loop with user defined iterations: 1, 0
Nested loop with user defined iterations: 1, 1
Nested loop with user defined iterations: 1, 2
Nested loop with user defined iterations: 1, 3
Nested loop with user defined iterations: 2, 0
Nested loop with user defined iterations: 2, 1
Nested loop with user defined iterations: 2, 2
Nested loop with user defined iterations: 2, 3
Nested loop with user defined iterations: 3, 0
Nested loop with user defined iterations: 3, 1
Nested loop with user defined iterations: 3, 2
Nested loop with user defined iterations: 3, 3
Range-loop in another function: 0, foo
Range-loop in another function: 1, bar
Range-loop in another function: 2, baz
A loop with no init, cond and post statements: 5
A loop with no init, cond and post statements: 4
A loop with no init, cond and post statements: 3
A loop with no init, cond and post statements: 2
A loop with no init, cond and post statements: 1
A loop with no init, cond and post statements: 0
Total loop iterations count: 38
```

Вывод

В результате выполнения лабораторной работы я изучил представление синтаксического дерева программы на языке Go и получил навыки преобразования этого дерева с использованием пакетов `go/ast`, `go/parser`, выполнив подсчёт общего числа итераций всех `for`- и `range`-циклов во время выполнения программы.