Лабораторная работа №2
«Синтаксические деревья»

Скоробогатов С.Ю.

15 февраля 2014 г.

1 Цель работы

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

2 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 `http://golang.org/pkg/go/`.

Построение синтаксического дерева по исходному тексту программы выполянется 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, порождаемым 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.

Алгоритм 1 Исходный текст программы astprint.go

```
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)
    }
}
```

Алгоритм 2 Исходный текст функции `insertHello`

```
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
    })
}
```

3 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 должно вносить в преобразуемую программу дополнительные возможности, перечисленные в таблице 1.

Таблица 1: Дополнительные возможности, приобретаемые преобразованной программой

1	Подсчёт количества вызовов каждой глобальной функции в ходе работы программы
2	Подсчёт общего количества итераций всех циклов в процессе выполнения программы
3	Вычисление максимальной глубины стека вызовов функций, достигнутой в ходе работы программы (считать, что программа не порождает сопрограмм)
4	Подсчёт, сколько раз в ходе работы программы вызывались сопрограммы
5	Объявления переменных в конструкции `var` в исходном тексте программы должны быть отсортированы по алфавиту
6	Объявления глобальных функций должны располагаться в конце программы в алфавитном порядке
7	Любая неанонимная функция, имеющая ровно один параметр типа `int`, должна выводить своё имя и значение параметра
8	Любая неанонимная функция, возвращающая ровно одно значение типа `int`, должна выводить своё имя и возвращаемое значение
9	После выхода из любого цикла должно выводиться количество выполненных итераций
10	В каждый оператор `switch`, не имеющий `default`-секции, добавляется `default`-секция, содержащая печать какого-нибудь сообщения
11	Динамические срезы, создаваемые функцией `make`, должны иметь в два раза большую вместимость, чем в исходной программе (при этом их длина меняться не должна)
12	Если в исходной программе при создании отображения с помощью функции `make` не указан размер отображения, следует этот размер сделать равным 16
13	Во всех вызовах функции `println`, имеющих несколько фактических параметров, вставить между параметрами строку `", "`
14	Каждое вхождение строкового литерала в текст программы должно быть заменено идентификатором константы, добавленной в начало программы и имеющей соответствующее значение (при этом значения добавляемых констант не должны дублироваться)