Лабораторная работа №2
«Синтаксические деревья»

Скоробогатов С.Ю.

20 августа 2013

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, 2и 3.

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 `fmt.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).
15	Заменить константы, задаваемые через `iota` на явно заданные значения.
16	Заменить в программе все десятичные числа на шестнадцатиричные.
17	Подсчитать число присваиваний в процессе выполнения программы.

Таблица 2: Дополнительные возможности, приобретаемые преобразованной программой (продолжение)

18	Любая неанонимная функция должна в начале выполнения и перед возвратом (оператором `return` или при достижении конца блока) выводить слово `"Starts "` и `"Ends "`, соответственно, своё имя и время, прошедшее с начала выполнения программы (см. функции `time.Now()` и `time.Since()`).
19	Заменить вхождения глобальной переменной `LINE` на номер текущей строки, `FILE` — на имя текущего файла. В исходной программе эти переменные должны быть объявлены в глобальной области видимости с типами `int` и `string`, соответственно (в противном случае следует выдать ошибку). В новой программе они должны отсутствовать.
20	Заменить вхождения глобальной переменной `FUNCNAME` на имя неанонимной функции, в которой переменная упоминается, либо на `"(global)"`, если переменная упоминается в глобальном контексте. В исходной программе переменная `FUNCNAME` должна быть объявлена в глобальной области видимости типа `string`, в новой программе она должна отсутствовать.
21	Заменить запись `x := expr` на `var x = expr`.
22	Заменить запись `var x = expr` на `x := expr`. Внимание! Синтаксис с `:=` допустим только внутри функций, поэтому вне функций замену осуществлять не нужно.
23	Заменить вызов функции `assert(expr)` на конструкцию `if ! (expr) { fmt.Println("filename.go:NN:assertion failed"); }`, где вместо `filename.go` и `NN` должны находиться имя файла и номер строки с `assert`.
24	Заменить операторы инкремента `x++` и декремента `x--` на полную запись, соответственно, `x = x + 1` и `x = x - 1`.
25	Заменить запись сокращённых операторов присваивания `x OP= y` на соответствующие полные записи `x = x OP y`. (`OP` может быть `+`, `-`, `|`, `^`, `*`, `/`, `%`, `<<`, `>>`, `&`, `&^`).
26	Каждому оператору `if`, у которого нет ветки `else`, добавить ветку `else`, содержащую печать какого-нибудь сообщения.
27	Каждый оператор `for` вида `for expr {...}` преобразовать в оператор `for fmt.Println("init"); expr; fmt.Println("next") {...}`.
28	Вместо каждой именованной константы, имя которой заканчивается на `_`, подставить её значение.
29	Каждое целое положительное чётное число заменить на сумму двух нечётных чисел в скобках. Например, `30` можно заменить на `(9+21)`.
30	Для каждого присваивания распечатать имя переменной и значение присваиваемого выражения.

Таблица 3: Дополнительные возможности, приобретаемые преобразованной программой (продолжение)

31	Подсчет общего количества операций сложения, вычитания, умножения и деления (подсчет каждой операции вести отдельно).
32	Разбить блоки объявления нескольких переменных на несколько блоков объявления одной переменной.
33	Подсчет времени выполнения каждой глобальной функции относительно времени выполнения всей программы.
34	Для каждого оператора `if` подсчитать количество переходов по положительной ветке относительно общего количества вызова конкретного оператора `if`.
35	Заменить все операторы `switch` стандартного (C-образного) вида на аналогичные по логике работы операторы `switch` без выражения.
36	Поставить символ `;` в конце каждой строки, которая оканчивается на целый чётный числовой литерал.
37	Во всех вызовах функции `fmt.Println` добавить в форматную строку префикс вида `LINE(FUNC_NAME): `, где `LINE` - текущий номер строки, `FUNC_NAME` - имя функции, из которой происходит вызов `fmt.Println`.
38	Заменить все умножения и деления на число, являющееся степенью двойки, на соответствующий побитовый сдвиг.
39	Заменить все операторы `if` без отрицательной ветки на аналогичный по логике работы оператор `switch` без выражения.
40	Если у функции используется сокращенное указание типа для нескольких аргументов (например `func f(a, b int)`), то явно указать тип для каждого аргумента отдельно (например `func f(a int, b int)`).