

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 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, 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

```
1 package main
2
3 import (
4     "fmt"
5     "go/ast"
6     "go/parser"
7     "go/token"
8     "os"
9 )
10
11 func main() {
12     if len(os.Args) != 2 {
13         fmt.Printf("usage: astprint <filename.go>\n")
14         return
15     }
16
17     // Создаём хранилище данных об исходных файлах
18     fset := token.NewFileSet()
19
20     // Вызываем парсер
21     if file, err := parser.ParseFile(
22         fset,                                // данные об исходниках
23         os.Args[1],                          // имя файла с исходником программы
24         nil,                                 // пусть парсер сам загрузит исходник
25         parser.ParseComments, // приказываем сохранять комментарии
26     ); err == nil {
27         // Если парсер отработал без ошибок, печатаем дерево
28         ast.Fprint(os.Stdout, fset, file, nil)
29     } else {
30         // в противном случае, выводим сообщение об ошибке
31         fmt.Printf("Error: %v", err)
32     }
33 }
```

Алгоритм 2 Исходный текст функции insertHello

```
1 func insertHello(file *ast.File) {
2     // Вызываем обход дерева, начиная от корня
3     ast.Inspect(file, func(node ast.Node) bool {
4         // Для каждого узла дерева
5         if ifStmt, ok := node.(*ast.IfStmt); ok {
6             // Если этот узел имеет тип *ast.IfStmt,
7             // добавляем в начало массива операторов
8             // положительной ветки if'а новый оператор
9             ifStmt.Body.List = append(
10                []ast.Stmt{
11                    // Новый оператор -- выражение
12                    &ast.ExprStmt{
13                        // Выражение -- вызов функции
14                        X: &ast.CallExpr{
15                            // Функция -- "fmt.Printf"
16                            Fun: &ast.SelectorExpr{
17                                X: ast.NewIdent("fmt"),
18                                Sel: ast.NewIdent("Printf"),
19                            },
20                            // Её параметр -- строка "hello"
21                            Args: []ast.Expr{
22                                &ast.BasicLit{
23                                    Kind: token.STRING,
24                                    Value: "\"hello\"",
25                                },
26                            },
27                        },
28                    },
29                },
30                ifStmt.Body.List...,
31            )
32        }
33        // Возвращая true, мы разрешаем выполнять обход
34        // дочерних узлов
35        return true
36    })
37 }
```

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 и 2.

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 <code>var</code> 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 <code>int</code> , должна выводить своё имя и значение параметра.
8	Любая неанонимная функция, возвращающая ровно одно значение типа <code>int</code> , должна выводить своё имя и возвращаемое значение.
9	После выхода из любого цикла должно выводиться количество выполненных итераций.
10	В каждый оператор <code>switch</code> , не имеющий <code>default</code> -секции, добавляется <code>default</code> -секция, содержащая печать какого-нибудь сообщения.
11	Динамические срезы, создаваемые функцией <code>make</code> , 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 <code>make</code> не указан размер отображения, следует этот размер сделать равным 16.
13	Во всех вызовах функции <code>fmt.Println</code> , имеющих несколько фактических параметров, вставить между параметрами строку <code>" , "</code> .
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 <code>iota</code> 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 <code>return</code> или при достижении конца блока) выводить слово <code>"Starts "</code> и <code>"Ends "</code> , соответственно, своё имя и время, прошедшее с начала выполнения программы (см. функции <code>time.Now()</code> и <code>time.Since()</code>).
19	Заменить вхождения глобальной переменной <code>LINE</code> на номер текущей строки, <code>FILE</code> — на имя текущего файла. В исходной программе эти переменные должны быть объявлены в глобальной области видимости с типами <code>int</code> и <code>string</code> , соответственно (в противном случае следует выдать ошибку). В новой программе они должны отсутствовать.
20	Заменить вхождения глобальной переменной <code>FUNCNAME</code> на имя неанонимной функции, в которой переменная упоминается, либо на <code>"(global)"</code> , если переменная упоминается в глобальном контексте. В исходной программе переменная <code>FUNCNAME</code> должна быть объявлена в глобальной области видимости типа <code>string</code> , в новой программе она должна отсутствовать.
21	Заменить запись <code>x := expr</code> на <code>var x = expr</code> .
22	Заменить запись <code>var x = expr</code> на <code>x := expr</code> . Внимание! Синтаксис <code>s :=</code> допустим только внутри функций, поэтому вне функций замену осуществлять не нужно.
23	Заменить вызов функции <code>assert(expr)</code> на конструкцию <code>if ! (expr) { fmt.Println("filename.go:NN:assertion failed"); }</code> , где вместо <code>filename.go</code> и <code>NN</code> должны находиться имя файла и номер строки с <code>assert</code> .
24	Заменить операторы инкремента <code>x++</code> и декремента <code>x--</code> на полную запись, соответственно, <code>x = x + 1</code> и <code>x = x - 1</code> .
25	Заменить запись сокращённых операторов присваивания <code>x OP= y</code> на соответствующие полные записи <code>x = x OP y</code> . (<code>OP</code> может быть <code>+</code> , <code>-</code> , <code> </code> , <code>^</code> , <code>*</code> , <code>/</code> , <code>%</code> , <code><<</code> , <code>>></code> , <code>&</code> , <code>&^</code>).
26	Каждому оператору <code>if</code> , у которого нет ветки <code>else</code> , добавить ветку <code>else</code> , содержащую печать какого-нибудь сообщения.
27	Каждый оператор <code>for</code> вида <code>for expr {...}</code> преобразовать в оператор <code>for fmt.Println("init"); expr; fmt.Println("next") {...}</code> .
28	Вместо каждой именованной константы, имя которой заканчивается на <code>_</code> , подставить её значение.
29	Каждое целое положительное чётное число заменить на сумму двух нечётных чисел в скобках. Например, <code>30</code> можно заменить на <code>(9+21)</code> .
30	Для каждого присваивания распечатать имя переменной и значение присваиваемого выражения.