GO-07 02: Кодогенерация. Работа с абстрактным синтаксическим деревом go

Описание:

В этом задании мы познакомимся еще с одним механизмом анализа кода. Но в этот раз код, который мы будем анализировать, будет немножко другим. То есть, до этого при помощи reflect и стандартных средств языка мы могли лишь анализировать типы данных в нашей программе, теперь же мы попробуем проанализировать исходный код.

AST

Первым механизмом, с которым мы познакомимся, будет механизм работы с AST (абстрактным синтаксическим деревом). Для более полного понимания ситуации давайте заглянем немного вглубь построения языков программирования и вспомним, как работают компиляторы.

Структура компилятора

Первое, на что можно разделить схему работы компилятора, - это Frontend и Backend, где второй отвечает за генерацию машинного кода и применение различных оптимизаций на разных уровнях, а мы остановимся чуть подробнее на первом.

Frontend компилятора, в свою очередь, отвечает за преобразование вашего текстового файла с набором символов в удобное для бэкенда представление, чтобы дальше из этого представления компилятор легче мог генерировать другой, нужный ему код. Но бэкенду компилятора очень сложно из просто напечатанных символов генерировать код, ему намного проще работать с чем-то программным (переменными, циклами, структурами и т.д). И вот здесь вступает в игру AST.

AST (абстрактное синтаксическое дерево) представляет из себя некую древовидную структуру данных, которая полностью описывает работу программы, написанную на любом языке программирования. Каждая нода этого дерева является частью написанной программы. Чтобы на это взглянуть, можно перейти вот сюда и попробовать попарсить разные Go программы, например, hello, world.

Но как же компилятор это делает? Как он из обычного текста генерирует такое дерево? В этом ему помогают два компонента - это Lexer и Parser.

Lexer

Для начала поговорим о самой первой стадии превращения ваших написанных символов в AST. Каждая программа на любом языке программирования состоит из маленьких компонентов var, if, for, func, etc. Например, функция ниже суммирует два числа и на выход отдает результат, вот так мы ее обычно видим:

```
func sum(a, b int) int {
    return a + b
```

```
}
```

Но компилятор видит эту программу немного иначе - это не конкретные токены языка Go, а просто абстрактное видение типичного лексера:

```
FUNC_DEF IDENT("sum") '(' IDENT("a") ',' IDENT('b') TYPE_INT ')'
TYPE_INT '{'
  RETURN IDENT('a') PLUS IDENT('b')
'}'
```

На первом этапе компилятор разбивает написанную программу на маленькие части Лексемы и каждой лексеме дает свой заранее подготовленный для нее идентификатор Токен. Если же для какой-то лексемы токен не будет найден, тогда мы обычно видим ошибку компиляции. После генерации всех токенов они отправляются на следующую стадию парсинга.

Parser

Парсер же служит в компиляторах для того, чтобы генерировать из токенов AST и, возможно, проводить какие-то оптимизации на этом этапе. Но кроме токенов, парсеру для составления AST не хватает еще одной важной детали - грамматики. Как и любые другие человеческие языки, языки программирования имеют свои правила написания (синтаксис). Например, мы не можем написать вот такую программу:

```
sum() func a, b int } {
```

Вроде, если присмотреться, мы не нарушаем лексики языка, и лексер генерирует все нужные для нашей программы токены, но на этапе парсинга нашей программы в AST мы будем наблюдать ошибку компиляции из-за того, что программа не отвечает правилам написания (грамматике). Это как начать соседу говорить много бессвязных слов, вроде все слова на родном языке, а смысла в них нет.

Так вот при помощи токенов и грамматики языка компилятор способен сгенерировать AST, а также выкинуть ненужные для генерации машинного кода части (Смысл в том, что все эти запятые, скобочки и стрелки нужны только нам, программистам, для понимания того, что мы пишем. Для компилятора же это все мусор, который только усложнил бы генерацию правильного кода, поэтому парсер генерирует дерево так, чтобы исключить все ненужные части написанной программы и оставить только то, что точно пригодится. От этого и происходит название AST).

Пакет go/ast

В стандартной библиотеке go есть пакеты, которые нужны специально для анализа golang программ и представления их в AST виде. Например, мы можем написать при помощи этих пакетов функцию, которая будет выводить все импортированные пакеты в любом go файле:

```
package main

import (
  "fmt"
```

```
  "go/ast"
  "go/parser"
  "go/token"
)

func main() {
      fileSet := token.NewFileSet()
      node, err := parser.ParseFile(fileSet, "another_file.go", nil,
parser.ParseComments)
      if err != nil {
            panic(err)
      }

      for _, c := range node.Imports {
            fmt.Println(c.Path)
      }
}
```

Или, чутка изменив предыдущую функцию, можно найти все экспортируемые функции в пакете:

```
for _, decl := range node.Decls {
  f, ok := decl.(*ast.FuncDecl)
  if !ok && !f.Name.IsExported() {
    continue
  }

  fmt.Println(f.Name.Name)
}
```

Или чуть более сложный пример, попробуем найти что-то внутри ноды, например, все return выражения, а также вывести, на какой строке кода мы нашли return выражение:

```
ast.Inspect(node, func(n ast.Node) bool {
  ret, ok := n.(*ast.ReturnStmt)
  if ok {
    fmt.Printf("return statement found on line %d:\n\t",
fileSet.Position(ret.Pos()).Line)
  }
  return true
})
```

В этих программах при помощи пакетов go/ast, go/token и go/parser мы сумели проанализировать исходный код нашей программы, что позволило получить много дополнительной информации.

- Пакет go/ast содержит в себе все, что нужно для работы с нодами дерева. В нем есть методы для просмотра нод, структуры, которые представляют из себя какие-либо выражения в go, например, выше мы видели ast.ReturnStmt - это структура, в которой хранится информация о выражениях вида return ..., а из структуры мы можем вытащить такую информацию, как:
    - позиция, то есть, на какой строчке в исходном коде находится данное выражение, - через returnStmt.Pos();
    - какие аргументы у этого выражения (ноды, которые идут за return) - через returnStmt.Results;
    - токен - через returnStmt.Return;
    - токен конца выражения - через returnStmt.End().
- Также у go/ast есть методы для создания новых нод, такие как ast.NewPackage, ast.NewObj, ast.NewScope, etc. При помощи этих методов и структур для выражений можно вполне создать новое AST дерево, затем из него сгенерировать исходный код через другой пакет go/format.
- Пакет go/format нужен для того, чтобы либо отформатировать исходный код в стандартный gofmt style, либо отформатировать наше AST дерево и записать результат, например, в новый файл.
- Пакет go/token хранит в себе все токены языка go и методы для работы с ними.
- Пакет go/parser нужен для парсинга исходных файлов в AST дерево.

Полезные ссылки:

- Golang AST Visualizer
- Understand go programm with go/parser
- Basic AST Traversal in Go

Задание:

В этом задании вам нужно проанализировать код, который вы написали в прошлом задании, и собрать следующую информацию:

- Сколько объявлений было в файле. Объявлениями или Declaration считаются конструкции типа var a, const a, etc.
- Сколько присваиваний (Assigns) было в файле. Присваиванием считается такое выражение, где любой переменной было назначено (присвоено) значение.
- Какие импорты есть в файле.
- Количество объявленых вызовов функций в файле.

Функции на вход передает строка, обозначающая путь к файлу, который нужно проанализировать. На выходе нужно предоставить структуру с результатами анализа. Структура имеет вид:

```
type AnalysResult struct {
  DeclCount    int
  CallCount    int
  AssignCount  int
  ImportsCount int
}
```

Файлы для анализа находятся в папке module07/assets/astanalys. В тестах будут проверятся именно эти файлы.

Порядок действий:

1. В вашем проекте module07 сделайте новую ветку module07_02.
2. В пакете module07/internal/analys заполните логикой функцию Analys.
3. Проверьте, что все тесты проходят успешно (тесты можно запустить при помощи команды make test_02).
4. В качестве ответа пришлите ссылку на merge request в ветку master вашего проекта ветки module07_02.