УДК 004.423

РАЗРАБОТКА ЛЕГКОВЕСНЫХ ПАРСЕРОВ С РАЗНОЙ ДЕТАЛИЗАЦИЕЙ ЯЗЫКА GO

Д. С. Дроздов[1] [0000-0003-0381-1012], С. С. Михалкович[2] [0000-0003-0373-3886]

[1, 2]*Южный федеральный университет*

[1]ds-drozdov@yandex.ru, [2]miks@sfedu.ru

Аннотация

Рассмотрен подход к созданию семейства легковесных грамматик для языка Go со специальным символом Any, обозначающим пропускаемую часть программы [1]. Дано формальное определение более детализированной грамматики, приведены примеры увеличения детализации правил грамматики. Проведен анализ эффективности семейства построенных легковесных парсеров по памяти и времени работы на семи промышленных репозиториях. Показано, что увеличение детализации грамматики не ведет к существенному росту потребления ресурсов парсером и незначительно колеблется в зависимости от типа репозитория и стиля написания на Go. Кроме того, указаны преимущества использования легковесных грамматик с символом Any по сравнению с полными грамматиками. Представлен пример использования легковесной грамматики для определения сложности кода. Полученные результаты могут быть также применены для оценки доли парсера в общем потреблении ресурсов, например, в задаче привязки к коду и разметки проекта.

Ключевые слова*: легковесная грамматика, легковесный парсер, язык Go, грамматика Go, грамматика с символом Any.*

ВВЕДЕНИЕ

Первым этапом компиляции является парсинг, основанный на грамматике. Однако парсер строит AST-дерево только для правильных программ заданной версии языка. Кроме того, иногда требуются пре- и постобработка текста программы для корректной работы парсера. Например, в языке Go, который рассмотрен в данном исследовании, компилятор сначала расставляет точки с запятой по определенному правилу [2], а затем удаляет их после окончания разбора кода.

В ряде задач требуется выделить из программы только некоторые сущности, и потому использование полного парсера нецелесообразно. К таким задачам относится создание разметки кода для быстрой навигации по проекту [3, 4].

Проблема вычленения нужных фрагментов кода решается, в частности, с помощью островных грамматик [5, 6]. Согласно данному подходу, необходимые части кода являются «островами», а остальные — «водой», где допустим синтаксически неправильный код. В работе [1] приведен алгоритм разбора с символом Any, который является аналогом «воды» островных грамматик, и даны примеры легковесных грамматик для C#, PascalABC.NET и Roslyn. В настоящем исследовании названные алгоритмы применены для разбора программ на Go по трем разработанным грамматикам с Any.

Стоит отметить, что для Go официально опубликована[1] только спецификация, содержащая описания конструкций языка с помощью расширенной формы Бэкуса–Наура. В то же время отдельно от компилятора существует предназначенный для анализа кода парсер, созданный разработчиками Go. Он более простой, но, согласно документации[2], допускает некоторые неправильные программы. В открытом доступе[3] можно найти неофициальную грамматику Go для ANTLR, которая отличается от двух предыдущих. Таким образом, не существует единой правильной грамматики Go и универсального парсера, однако задача разбора программ на Go тем не менее открыта. Именно в этих ситуациях семейство легковесных парсеров наиболее востребовано.

При создании легковесных грамматик зачастую возникает проблема корректности описания подмножества языка, которая характерна и для обычных грамматик. В случае Go требуется обрабатывать несколько вариантов перечисления аргументов функций. Чтобы не усложнять грамматику с Any, данная проблема решается на уровне постобработки полученного синтаксического дерева [7].

Нетрудно предположить, что степень детализации правил влияет на размер синтаксического дерева и скорость его построения. В настоящем исследовании

[1] https://go.dev/ref/spec
[2] https://pkg.go.dev/go/parser
[3] https://github.com/antlr/grammars-v4

проведен анализ эффективности трех парсеров Go, соответствующих различным грамматикам: от менее до более детализированной.

В работе получены следующие результаты:

1. формализовано определение детализации грамматик;
2. созданы легковесные грамматики для Go разной степени детализации, экспериментально доказана их корректность;
3. проведен анализ эффективности разбора по новым грамматикам.

В разделе 1 описаны преимущества и основные принципы работы легковесных парсеров. Раздел 2 содержит формальное определение более детализированной грамматики и описание трех созданных грамматик Go с разной степенью детализации. Дано определение трудночитаемых функций и приведена легковесная грамматика для поиска таковых. В разделе 3 проведена проверка корректности созданных грамматик. Раздел 4 содержит оценку их эффективности по памяти и времени работы. В разделе 5 дано описание работ, близких к настоящей.

1. Легковесные парсеры

В работе [1] рассмотрены легковесные грамматики со специальным символом Any, который отвечает за пропуск определенных конструкций языка. С помощью инструмента LanD можно генерировать легковесные парсеры, использующие Any [8].

При построении синтаксического дерева символу Any соответствует единый узел, который хранит все пропускаемые токены языка в виде строки. В случае необходимости в дальнейшем можно обратиться к данному узлу и разобрать его текст, например, более детальным парсером. Поскольку уменьшается количество узлов и связей между объектами, то ожидается снижение потребления памяти, что и подтверждено результатами экспериментов.

В данном исследовании обработаны только сигнатуры функций и методов Go, поэтому их тела и некоторые глобальные конструкции описаны через Any. Заметим, что если тело содержит ошибки с точки зрения полного парсера, то сигнатура все равно будет распознана легковесным парсером.

Легковесные грамматики хорошо расширяются, оставаясь такими же простыми. Например, начиная с Go 1.18, в язык добавили обобщенные типы — дженерики. Описание такой конструкции может быть довольно громоздким (рис. 1):

```
func Do[S ~[]E, K comparable, E constraints.Integer,
    V int64 | float64, T map[E]interface{}]() {
```

Рис. 1. Дженерики в Go

Однако для расширения легковесной грамматики и поддержки новых версий языка достаточно добавить в правило заголовка функции конструкцию '[' Any ']'. На рис. 2 приведены пример упрощенного правила для сигнатуры функции (func) и его версия с учетом дженериков (func2).

```
func  = 'func' ID             '(' Any ')' Any '{' Any '}'
func2 = 'func' ID '[' Any ']' '(' Any ')' Any '{' Any '}'
```

Рис. 2. Изменение правила грамматики при учете дженериков Go

В результате такого изменения код будет разбираться корректно, а дженерики сохранятся в тексте узла Any.

С помощью опции skip инструмента LanD выполняется пропуск при разборе некоторых конструкций. Мы игнорируем все комментарии, потому что закомментированные функции и методы не должны распознаваться, а также строки, которые тоже могут содержать сигнатуры функций. Пропускаемые конструкции зададим при помощи регулярных выражений (рис. 3).

```
COMMENT     : COMMENT_L|COMMENT_ML
COMMENT_L   : '//' ~[\n\r]*
COMMENT_ML  : '/*' .*? '*/'
STRING      : STRING_ENC|STRING_STD
STRING_ENC  : '`' ('\\"'|'\\\\'|.)*? '`'
STRING_STD  : '"' ('\\"'|'\\\\'|.)*? '"'
%%
%parsing {
    skip COMMENT STRING
}
```

Рис. 3. Пропускаемые конструкции

2. Грамматики Go с разной детализацией

Определение 1. *Обозначим через a_i произвольный символ грамматики (терминал или нетерминал), через α, β — произвольные цепочки символов. Рассмотрим правило $S \to a_1 \dots a_{k-1}\ \boldsymbol{Any}\ a_{k+p+1} \dots a_n$, для которого выполнено хотя бы одно из следующих условий:*

1. $a_1 \neq Any$, и не существует вывода $a_1 \to^* Any\ \alpha$;

$a_n \neq Any$, и не существует вывода $a_n \to^* \alpha\ Any$.

2. Существует хотя бы одно правило вида $R \to a_1 \dots R' \dots a_n$, где $R' \to^* \alpha S \beta$, и для любого такого правила R выполнены ограничения из пункта 1 на a_1 и a_n.

Тогда правило S называется менее детализированным по сравнению с правилом $S' \to a_1 \dots a_{k-1} \boldsymbol{a_k} \dots \boldsymbol{a_{k+p}} a_{k+p+1} \dots a_n$ для любых натуральных k, n, p: $1 < k < n - p$.

Из определения следует, что Any заменяет символы от a_k до a_{k+p} включительно. Их содержимое войдет в узел Any в виде текста. Важно отметить, что в менее детализированном правиле S символ Any должен быть обрамлен хотя бы одним символом слева и справа, в противном случае правило может захватывать символы без ограничения [1]. Допускается также вариант, когда все правила R, из которых можно получить последовательность символов, содержащую S, будут обрамлять Any другими символами.

Менее детализированное правило допускает те же цепочки, что и исходное, плюс, быть может, и другие цепочки.

Определение 2. *Грамматика языка G_1 называется менее детализированной версией грамматики G_2, если выполнены следующие условия:*

1. *Хотя бы одно правило G_1 является менее детализированной версией правила G_2.*
2. *Символы G_2, ставшие недостижимыми после уменьшения детализации, и правила вывода из таких символов отсутствуют в G_1.*
3. *Остальные правила совпадают.*

Определение 3. *Грамматика языка G_1 называется более детализированной версией грамматики G_2, если грамматика G_2 является менее детализированной версией грамматики G_1.*

В настоящем исследовании рассмотрены три версии грамматик языка Go, каждая из которых является более детализированной, чем предыдущая. Все грамматики описывают синтаксические части функций и методов.

В языке Go функция, относящаяся к структуре, называется *методом* [2]. В отличие от классов C# и C++ в структурах Go нет упоминания о методах, они определяются снаружи и в сигнатуре имеют название данной структуры. Пример функции и метода Go приведен на рис. 4.

```
func action(r *SomeType, s string) int  { return 0 } // функция
func (r *SomeType) action(s string) int { return 0 } // метод
```

Рис. 4. Функция и метод в Go

Первая созданная грамматика является наиболее общей. На рис. 5 приведен ключевой фрагмент данной грамматики, которая позволяет находить методы (f_reciever непуст) и функции (f_reciever пуст). Как видно, в каждом правиле использован символ Any, что позволяет выделить компоненты функции или метода, но не детализировать их. Кроме того, по значению f_reciever можно определить, какой структуре принадлежит метод.

```
1 func = 'func' f_reciever? f_name generic2? f_args f_returns ('{' Any '}')
2 f_reciever = LB Any RB
3 f_args = LB Any RB
4 f_returns = Any | LB Any RB
5 generic = '[' Any ']'
```

Рис. 5. Фрагмент первой грамматики

Здесь и далее символы грамматики LB и RB означают соответственно открывающую и закрывающую скобки. Символ Any предполагает сбалансированность по скобкам и другим парным символам, которые указываются в отдельной секции грамматики.

На рис. 6 приведен метод на языке Go и выделены штриховкой области символа Any, полученные при разборе по первой грамматике.

```go
func (u *User) Action(opt map[string]string) (int, error) {
    return 0, nil
}
```

Рис. 6. Области символа Any, полученные по первой грамматике

Вторая грамматика имеет более детальное определение аргументов и возвращаемых значений. На рис. 7 приведен фрагмент грамматики и выделены позиции, в которых выполнено уточнение Any.

```
func = 'func' f_reciever? f_name generic? f_args f_returns ('{' Any '}')
f_reciever = LB (param '*')? ID generic? RB
f_args = LB (f_arg ',')* f_arg? ','? RB
f_arg = go_type? SPREAD? go_type
f_returns = Any
        | LB (f_return ',')* f_return ','? RB
f_return = go_type? go_type
go_type = (ID | arr_ptr* (ID | anon | map | chan)) generic?
arr_ptr = '*'* ('[' Any ']' '*'*)*
generic = '[' Any ']'
anon = anon_func_title | anon_struct | anon_interface
map = 'map' '[' Any ']' Any
chan = 'chan' Any | 'chan<-' Any | '<-chan' Any
```

Рис. 7. Фрагмент второй грамматики

Из правил грамматики следует, что компоненты аргументов и возвращаемых значений определены через два подряд идущих типа (go_type). Однако первый из них должен отвечать за идентификатор переменной, а второй – за тип. Данное разделение выполняется на этапе постобработки и занимает 1.5–2% от общего времени разбора кода [7].

Необходимость такой постобработки обусловлена тем, что на уровне правил грамматики затруднительно описать одновременно три варианта перечисления аргументов в Go [9]:

1. Указаны переменные и их типы: func f(a int, b int, c byte).
2. Переменные одного типа объединены: func f(a, b int, c byte).
3. Указаны только типы: func f(int, int, byte).

На рис. 8 приведен метод и выделены штриховкой области символа Any, полученные при разборе по второй грамматике. В сравнении с первой грамматикой (рис. 6) видно, что в сигнатуре метода символ Any остался только для типа map.

```
func (u *User) Action(opt map[string]string) (int, error) {
    return 0, nil
}
```

Рис. 8. Области символа Any, полученные по второй грамматике

Детализация аргументов и возвращаемых значений потребовала описания типов Go. Во второй грамматике определены конструкции, представленные на рис. 9.

```
map[K]V          // 1. отображение из значения типа K в значение типа V
chan T           // 2. двунаправленный канал для обмена данными типа T
<-chan T         // 3. канал только на чтение
chan<- T         // 4. канал только на запись
func(K) V        // 5. anon_func_title – анонимная функция
struct{ K }      // 6. anon_struct – анонимная структура
interface{ f() } // 7. anon_interface – анонимный интерфейс
```

Рис. 9. Фрагмент третьей грамматики

Детализация типов 1–4 описана ниже в третьей грамматике. Типы 5–7 детализированы в работе [7].

Итак, вторая грамматика позволяет определять не только функцию и ее границы, но и содержимое аргументов. Ее можно использовать для задач разметки кода, подробно описанных в работах [3, 4]. Действительно, при поиске функциональностей в первую очередь анализируются функции и методы языка, при этом типы аргументов помогают различать функции с одинаковым названием.

Третья грамматика является наиболее детализированной. Отличие от второй состоит в определении отображений и каналов. На рис. 10 показано, что в правилах map и chan вместо Any теперь используется go_type.

```
1  func = 'func' f_reciever? f_name generic? f_args f_returns ('{' Any '}')
2  f_reciever = LB (param '*')? ID generic? RB
3  f_args = LB (f_arg ',')* f_arg? ','? RB
4  f_arg = go_type? SPREAD? go_type
5  f_returns = Any
6          | LB (f_return ',')* f_return ','? RB
7  f_return = go_type? go_type
8  go_type = (ID | arr_ptr* (ID | anon | map | chan)) generic?
9  arr_ptr = '*'* ('[' Any ']' '*'*)*
10 generic = '[' Any ']'
11 anon = anon_func_title | anon_struct | anon_interface
12 map = 'map' '['go_type']' go_type
13 chan = 'chan' go_type | 'chan<-' go_type | '<-chan' go_type
```

Рис. 10. Фрагмент третьей грамматики

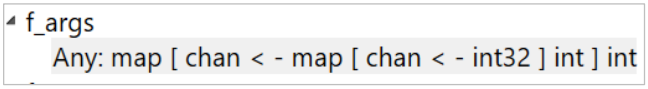
Заметим, что символы chan и map входят в правило узла go_type, который отвечает за любой тип языка Go. Поэтому третья грамматика допускает самовложения в данных конструкциях.

На рис. 11 приведен метод и выделена штриховкой область символа Any, полученная при разборе по третьей грамматике. В сравнении со второй грамматикой (рис. 8) видно, что в сигнатуре метода символ Any больше не используется.

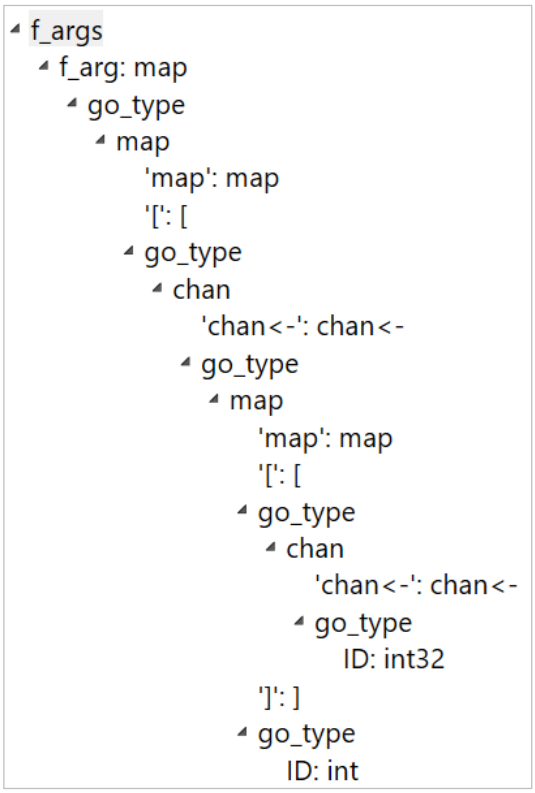
```
func (u *User) Action(opt map[string]string) (int, error) {
    return 0, nil
}
```

Рис. 11. Области символа Any, полученные по третьей грамматике

Сравним деревья, построенные по первой (рис. 12a) и третьей (рис. 12b) грамматикам для функции func F(map[chan<- map[chan<- int32]int]int). Видно, насколько изменяется максимальная глубина дерева при изменении детализации грамматики.



(a)



(b)

Рис. 12. Деревья, построенные по первой и третьей грамматикам

Второе дерево показывает, как выглядит самовложение map при разборе. Наличие таких конструкций делает задачу поиска функций и методов не реализуемой регулярными выражениями, поскольку они не допускают самовложения [10]. В то же время, легковесные грамматики справляются с этой задачей, так как являются контекстно-свободными.

Благодаря детализации построенного дерева, можно рассчитать вложенность всех аргументов и возвращаемых значений.

Определение 4. *Пусть d – узел с именем $name$ синтаксического дерева с дочерними узлами $c_1, c_2, \dots c_n$ либо без них. Сложностью узла d будем называть функцию*

$$f(d) = \begin{cases} 0, & \text{если } d - \text{лист} \\ 1 + f(c_n), & \text{если } name = go\_type \\ 2 + \max\limits_{1 \le i \le n}(f(c_i)), & \text{если } name = anon\_func\_title, struct\_content \end{cases}$$

Тогда сложность аргумента или типа возвращаемого значения вычисляется как глубина его синтаксического дерева, построенного по третьей грамматике. При этом узлы с анонимными структурами и функциями увеличивают глубину на 2 вместо 1, поскольку их использование само по себе усложняет прочтение заголовка.

Данным способом в открытых репозиториях можно найти трудночитаемые функции, которые следует декомпозировать.

Определение 5. *Будем называть функцию трудночитаемой, если она содержит хотя бы один аргумент или тип возвращаемого значения, узел d которого имеет сложность $f(d) > 2$.*

Величина порога, равная двум, выбрана на основе анализа распределения значений функции f на семи промышленных репозиториях. Из таблицы 1 следует, что 99.2% узлов имеют сложность не более двух.

Таблица. 1. Распределение функции сложности

Сложность f	**1**	**2**	**3**	**4**	**5**	**6**
Количество	413805	12680	3304	90	17	2
Частотность	96.25%	2.95%	0.77%	0.02%	<0.01%	<0.01%

С использованием данной грамматики и функции f в репозитории Kubernetes найден самый трудночитаемый код с точки зрения возвращаемого значения, представленный на рис. 13.

```
func resolveDualStackLocalDetectors(t *testing.T)
    func(localDetector proxyutiliptables.LocalTrafficDetector, err1 error)
        func(proxyutiliptables.LocalTrafficDetector, error)
            [2]proxyutiliptables.LocalTrafficDetector {
```

Рис. 13. Трудночитаемая функция

Сложность данного узла равна пяти, так как глубина дерева равна трем, а наиболее длинная ветвь дерева содержит две анонимные функции.

В том же репозитории с помощью данного подхода определена функция с аргументом, имеющем самую высокую сложность (рис. 14). Она рассчитана аналогично предыдущему примеру и равна пяти.

```
func MakeMatcher[T interface{}](
    match func(actual T) (failure func() string, err error),
) types.GomegaMatcher {
```

Рис. 14. Аргумент с высокой сложностью

Таким образом, третью грамматику можно использовать и в задаче разметки кода, и в задаче определения метрик.

3. Проверка корректности грамматик

Для проверки парсеров в данной работе использованы семь репозиториев с общей кодовой базой из более чем 43 000 файлов и 180 000 функций. Все грамматики прошли автоматическую проверку полным парсером Go по следующему алгоритму:

1. Организован проход по файлам с расширением «.go» в выбранном репозитории.
2. Для файла запускается полный парсер Go[4], опубликованный авторами языка.

[4] https://pkg.go.dev/go/parser

3. Выполняется обход в глубину синтаксического дерево, полученного на шаге 2. Если найден узел с объявлением функции или метода, то сохраняются следующие данные:
    a. имя функции;
    b. типы аргументов и возвращаемых значений;
    c. название структуры, для которой определен метод (для функций данный параметр отсутствует).
4. Файлы с кодом Go из предыдущих шагов разбираются легковесным парсером.
5. Обходится синтаксическое дерево, полученное на шаге 4. Если найден узел с объявлением функции или метода, то сохраняются те же данные, что и на шаге 3.
6. Информация, полученная от полного парсера (шаг 3) и от легковесного парсера (шаг 5), проверяется на совпадение.

Легковесные парсеры работают на C# [1], а полный парсер – на Go, поэтому проверка осуществляется посредством передачи данных в формате JSON. Информация, полученная на шаге 5, сохраняется в JSON, которую затем на шаге 6 считывает программа на Go и сравнивает со своим результатом полного разбора.

Поскольку нас не интересуют вложенные объявления, то при нахождении заголовка функции или метода обход данной ветки дерева полным парсером прекращается (шаг 3).

Обход дерева, построенного полным парсером, является менее эффективным, чем обход дерева, полученного по легковесной грамматике. В полном дереве придется обойти конструкции, которые расположены на одном уровне с объявлением функций и методов: объявления переменных, констант, пользовательских типов, структур и интерфейсов.

В зависимости от степени детализации грамматики перечень сравниваемых данных будет изменяться (подпункты алгоритма 3.a–3.c). Для первой, наименее детализированной грамматики сохраняется только имя функции или метода. Для второй добавляются типы аргументов и возвращаемых значений без детализации (например, для карты map[string]int учитывается только то, что это map). Наконец, для третьей, наиболее детализированной грамматики указываются полные типы.

По результатам проверки (таблица 2) экспериментально доказано, что парсеры, построенные по трем легковесным грамматикам, верно разбирают 100% заголовков функций и методов. Ложноположительные и ложноотрицательные срабатывания отсутствуют.

Таблица 2. Результаты проверки трех легковесных грамматик

	Всего аргументов	**Первая грамматика**		**Вторая грамматика**		**Третья грамматика**	
		Распознано аргументов					
kubernetes	115550	115550	100%	115550	100%	115550	100%
docker-ce	9632	9632		9632		9632	
azure-so	53659	53659		53659		53659	
sourcegraph	50629	50629		50629		50629	
moby	18072	18072		18072		18072	
chainlink	37527	37527		37527		37527	
tidb	64377	64377		64377		64377	

4. Эффективность грамматик

Для трех легковесных парсеров выполнены замеры времени работы и потребляемой памяти. Установлено, что при увеличении детализации грамматики от первой ко второй время работы увеличивается в среднем на 28% (таблица 3). Наибольший абсолютный прирост составил 10 секунд для репозитория Kubernetes. Добавление дополнительной детализации аргументов (от второй к третьей грамматике) увеличивает время работы в среднем на 4%.

Таблица. 3. Время работы трех легковесных парсеров

	Первая грамматика (сек.)	**Прирост времени (%)**	**Вторая грамматика (сек.)**	**Прирост времени (%)**	**Третья грамматика (сек.)**
kubernetes	26	38%	36	0%	36
docker-ce	10	10%	11	9%	12
azure-so	14	7%	15	7%	16
sourcegraph	5	40%	7	14%	8
moby	5	40%	7	0%	7
chainlink	4	50%	6	0%	6
tibd	13	8%	14	0%	14

Показано, что при переходе от первой грамматики к более детальной второй расход памяти увеличивается в среднем на 38% (таблица 4). При этом сильнее всего потребление возросло для проекта Chainlink и Docker CE – на 300 и 490 Мегабайт соответственно. Наиболее детализированная грамматика (третья) требует памяти еще на 8% больше.

Таблица. 4. Расход памяти трех легковесных парсеров

	Первая грамматика (Мбайт)	**Прирост памяти (%)**	**Вторая грамматика (Мбайт)**	**Прирост памяти (%)**	**Третья грамматика (Мбайт)**
kubernetes	1744	13%	2008	7%	2144
docker-ce	773	63%	1263	5%	1320
azure-so	653	11%	725	12%	814
sourcegraph	360	17%	420	8%	454
moby	328	32%	433	10%	476
chainlink	254	118%	554	7%	595
tibd	626	12%	703	8%	757

Отметим, что время работы парсера на самой общей грамматике составляет 4–6 секунд для средних промышленных репозиториев из 3000–4500 файлов. Соответственно, прирост времени работы при увеличении степени подробности грамматики составляет не более одной секунды.

На рис. 15 представлено изменение потребления ресурсов при переходе от первой грамматики к третьей. Между увеличением времени и ростом памяти прослеживается ожидаемая зависимость: чем меньше прирост памяти, тем меньше и прирост времени. Однако некоторые репозитории нарушают эту закономерность.

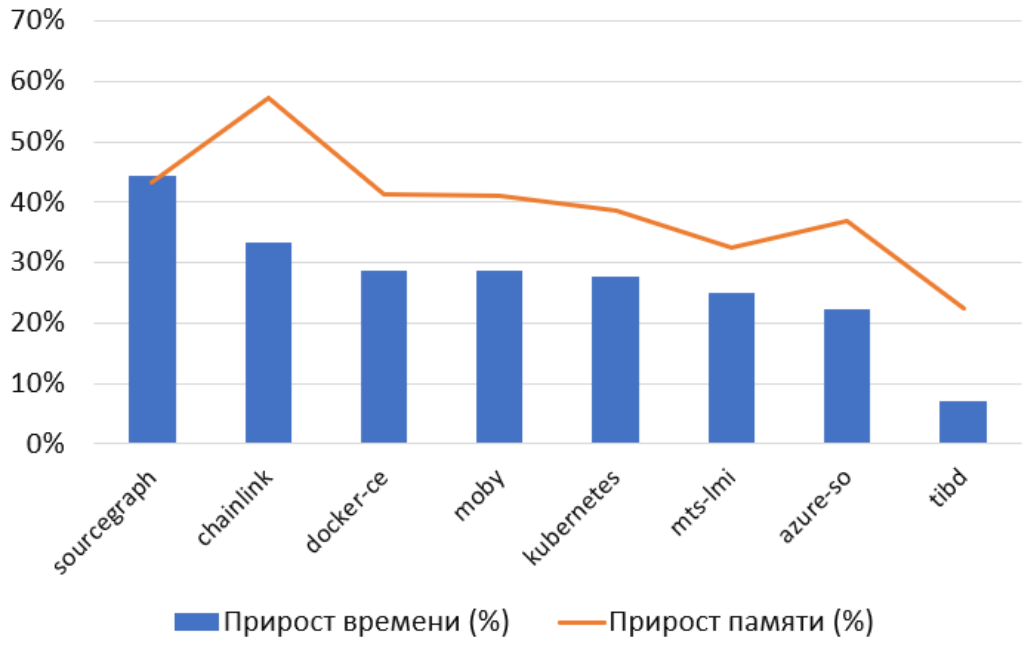


Рис. 15. Увеличение потребления ресурсов при использовании третьей грамматики по сравнению с первой

Из полученных результатов видно, что увеличение детализации грамматики Go не влечет за собой существенного замедления парсера или увеличения потребления памяти.

Эффективность внутреннего представления парсеров важна для задач привязки к коду [11]. Во-первых, данные о затратах на разбор позволяют вычислить чистое время перепривязки. Во-вторых, эти алгоритмы достаточно ресурсоемкие, поэтому недопустимо вносить замедление парсером, который является базовым элементом логики.

5. Близкие работы

Разметка кода, осуществляемая с помощью парсеров, является первым этапом в решении задач алгоритмической привязки и быстрой навигации по коду. Эта проблема рассматривалась в работе [12], где введено понятие графа функциональностей. Такой граф отображает зависимости между фрагментами кода и позволяет разработчику перемещаться по функциональности, а также документировать связи между логическими частями программы.

Кроме того, для быстрой навигации по коду применяют машинное обучение на основе набора данных о том, как разработчик просматривает проект. В статье [13] приведен алгоритм предсказания позиций в коде, наиболее связанных по смыслу с текущей. Для этого требуются размеченные данные по конкретному языку программирования, которые представлены авторами только для Python, C# и C++. В то же время, разметка с использованием легковесных грамматик требует знания только нужного фрагмента полной грамматики языка [3].

Отметим, что легковесные грамматики можно создавать не только для языков программирования, но и для других формализованных форматов записи. Например, в работе [1] приводятся грамматики с Any для языков спецификации Yacc и Lex, а также для языков разметки XML и Markdown. Несмотря на разнообразие рассмотренных языков, до настоящего исследования легковесные грамматики для Go не создавались.

Наконец, в статье [14] исследователи развивают теорию островных грамматик и вводят термин «озеро» («вода» посреди «острова»). В то же время символ Any, обрамленный разобранным кодом, фактически также является «озером».

ЗАКЛЮЧЕНИЕ

В данной работе рассмотрен подход к построению семейства легковесных грамматик языка Go. Дано формальное определение более и менее детализированных грамматик.

На примере трех созданных грамматик Go показано, что использование символа Any позволяет задавать детализацию нужного уровня для разбираемого

кода. Приведены результаты проверки корректности данных грамматик. Установлено, что эффективность легковесных парсеров Go незначительно колеблется в зависимости от типа проекта и вида грамматики.

Рассмотрены возможные варианты применения легковесных парсеров с разной степенью детализации. В частности, дано определение трудночитаемых функций и приведена грамматика для вычленения таких конструкций в коде на Go.

Кроме того, созданные грамматики могут быть использованы в задаче алгоритмической привязки к коду [10] как фундамент для разметки. Для этого лучше всего подходит вторая из рассмотренных грамматик, поскольку она имеет достаточную детализацию аргументов для работы алгоритма привязки.

Полученные результаты измерения эффективности могут послужить основой для оценки вклада парсера в общий расход ресурсов во время привязки.

СПИСОК ЛИТЕРАТУРЫ

1. *Goloveshkin A.V., Mikhalkovich S.S.* Tolerant parsing with a special kind of «Any» symbol: the algorithm and practical application // Proc. ISP RAS. 2018. Vol. 30. P. 7–28.

2. *Bodner J.* Learning Go. An Idiomatic Approach to Real-World Go Programming. Sebastopol: O'Reilly Media Inc., 2024. 353 p.

3. *Головешкин А.В., Михалкович С.С.* Разметка сквозных функциональностей в коде программы // Труды XXI Всероссийской научной конференции «Научный сервис в сети Интернет», Новороссийск, Россия, 23–28 сентября 2019 г. ИПМ им. М.В. Келдыша: 2019, с. 245–256.

4. *Malevannyy M., Mikhalkovich S.* Context-based model for concern markup of a source code // Proc. ISP RAS. 2016. Vol. 28. P. 63–78.

5. *Moonen L.* Generating Robust Parsers Using Island Grammars // Proceedings of the 8th Working Conference on Reverse Engineering, Stuttgart, Germany, Oct. 2 to Oct. 5 2001. IEEE: 2001, P. 13–22.

6. *Moonen L.* Lightweight Impact Analysis using Island Grammars // Proceedings of the 10th International Workshop on Program Comprehension, Paris, France, June 27 2002 to June 29 2002. IEEE: 2002, P. 219–228.

7. *Дроздов Д.С., Михалкович С.С.* Создание и постобработка легковесных грамматик Go и GraphQL для разметки функциональностей кода // Труды XXXI всероссийской научной конференции «Современные информационные технологии: тенденции и перспективы развития», Ростов-на-Дону, Россия, 18–20 апреля 2024. ЮФУ: 2024, С. 163–165.

8. *Головешкин А.В., Михалкович С.С.* LanD: инструментальный комплекс поддержки послойной разработки программ // Труды XXV всероссийской научной конференции «Современные информационные технологии: тенденции и перспективы развития», Ростов-на-Дону, Россия, 17–18 мая 2018. ЮФУ: 2018, С. 53–56.

9. *Freeman A*. Pro Go: The Complete Guide to Programming Reliable and Efficient Software Using Golang. New York: Apress, 2022. 1105 p.

10. *Мельцов В.Ю.* Лекции по теории автоматов. Часть 2. Киров: ВятГУ, 2010. 24 с.

11. *Goloveshkin A.V., Mikhalkovich S.S.* Using improved context-based code description for robust algorithmic binding to changing code // Procedia Computer Science, 2021. Vol. 139. P. 239–249.

12. *Robillard M., Murphy G.* Concern graphs: finding and describing concerns using structural program dependencies // Proceedings of the 24th international conference on Software engineering, New York, United States, May 19 to May 25 2002. ACM: 2002, P. 406–416.

13. *Paltenghi M., Pandita R. et al.* Extracting Meaningful Attention on Source Code: An Empirical Study of Developer and Neural Model Code Exploration // IEEE Transactions on Software Engineering. 2022. Vol. 50, No. 10, P. 256–2582.

14. *Okuda K., Chiba S.* Lake symbols for island parsing // The Art, Science, and Engineering of Programming. 2021. Vol. 5. Issue 2. P. 11:3–11:32.

DEVELOPMENT OF LIGHTWEIGHT PARSERS WITH DIFFERENT GO LANGUAGE GRANULARITY

D.S. Drozdov[1] [0000-0003-0381-1012], S.S. Mikhalkovich[2] [0000-0003-0373-3886]

[1, 2]*Southern Federal University*

[1]ds-drozdov@yandex.ru, [2]miks@sfedu.ru

Abstract

We consider an approach to creating a family of lightweight grammars with the Any symbol denoting skipping code parts [1]. Definition and examples of increasing the granularity of grammar rules are given. Memory and time efficiency of lightweight parsers is analyzed on seven industrial repositories. It is shown that increasing grammar granularity does not significantly increase parser resource consumption and varies slightly depending on repository type and Go writing style. Furthermore, the advantages of using lightweight grammars with Any over full grammars are summarized. An example of using a lightweight grammar to determine code complexity is presented. In addition, the results can be applied to estimate the parser's share of the total resource consumption, for example in the task of code binding and project markup.

Keywords*: lightweight grammar, lightweight parser, Go language, Go grammar, grammar with Any symbol.*

REFERENCES

1. *Goloveshkin A.V., Mikhalkovich S.S.* Tolerant parsing with a special kind of «Any» symbol: the algorithm and practical application // Proc. ISP RAS. 2018. Vol. 30. P. 7–28.

2. *Bodner J.* Learning Go. An Idiomatic Approach to Real-World Go Programming. Sebastopol: O'Reilly Media Inc., 2024. 353 p.

3. *Goloveshkin A.V., Mikhalkovich S.S.* Razmetka skvoznykh funktsionalnostei v kode programmy // Trudy XXI Vserossiiskoi nauchnoi konferentsii «Nauchnyi servis v seti Internet», Novorossiisk, Russia, 23–28 September 2019. IPM im. M.V. Keldysha: 2019, p. 245–256.

4. *Malevannyy M., Mikhalkovich S.* Context-based model for concern markup of a source code // Proc. ISP RAS. 2016. Vol. 28. P. 63–78.

5. *Moonen L.* Generating Robust Parsers Using Island Grammars // Proceedings of the 8th Working Conference on Reverse Engineering, Stuttgart, Germany, Oct. 2 to Oct. 5 2001. IEEE: 2001, P. 13–22.

6. *Moonen L.* Lightweight Impact Analysis using Island Grammars // Proceedings of the 10th International Workshop on Program Comprehension, Paris, France, June 27 2002 to June 29 2002. IEEE: 2002, P. 219–228.

7. *Drozdov D.S., Mikhalkovich S.S.* Sozdanie i postobrabotka legkovesnykh grammatik Go i GraphQL dlia razmetki funktsionalnostei koda // Trudy XXXI vserossiiskoi nauchnoi konferentsii «Sovremennye informatsionnye tekhnologii: tendentsii i perspektivy razvitiia», Rostov-na-Donu, Russia, 18–20 April 2024. SFeDU: 2024, P. 163–165.

8. *Goloveshkin A.V., Mikhalkovich S.S.* LanD: instrumentalnyi kompleks podderzhki posloinoi razrabotki programm // Trudy XXV vserossiiskoi nauchnoi konferentsii «Sovremennye informatsionnye tekhnologii: tendentsii i perspektivy razvitiia», Rostov-na-Donu, Russia, 17–18 May 2018. SFeDU: 2018, P. 53–56.

9. *Freeman A*. Pro Go: The Complete Guide to Programming Reliable and Efficient Software Using Golang. New York: Apress, 2022. 1105 p.

10. *Meltsov V.Iu.* Lektsii po teorii avtomatov. Part 2. Kirov: ViatGU, 2010. 24 p.

11. *Goloveshkin A.V., Mikhalkovich S.S.* Using improved context-based code description for robust algorithmic binding to changing code // Procedia Computer Science, 2021. Vol. 139. P. 239–249.

12. *Robillard M., Murphy G.* Concern graphs: finding and describing concerns using structural program dependencies // Proceedings of the 24th international conference on Software engineering, New York, United States, May 19 to May 25 2002. ACM: 2002, P. 406–416.

13. *Paltenghi M., Pandita R. et al.* Extracting Meaningful Attention on Source Code: An Empirical Study of Developer and Neural Model Code Exploration // IEEE Transactions on Software Engineering. 2022. Vol. 50, No. 10, P. 256–2582.

14. *Okuda K., Chiba S.* Lake symbols for island parsing // The Art, Science, and Engineering of Programming. 2021. Vol. 5. Issue 2. P. 11:3–11:32.

СВЕДЕНИЯ ОБ АВТОРАХ

ДРОЗДОВ Дмитрий Сергеевич – Аспирант Южного федерального университета по научной специальности 2.3.5 «Математическое и программное обеспечение вычислительных систем, комплексов и компьютерных сетей». В 2023 году получил степень магистра по направлению подготовки «Фундаментальная информатика и информационные технологии». Область интересов: языки программирования, компиляторы, грамматики.

Dmitry Sergeevich DROZDOV – Postgraduate student of the Southern Federal University in the scientific specialty 2.3.5 "Mathematical and software support for computing systems and computer networks". In 2023, he received a master's degree in the specialty "Fundamental informatics and information technology". Area of interest: programming languages, compilers, grammars.

email: ds-drozdov@yandex.ru

ORCID: 0000-0003-0381-1012

МИХАЛКОВИЧ Станислав Станиславович – Кандидат физ.-мат. наук, доцент, заведующий кафедрой информатики и вычислительного эксперимента Института математики, механики и компьютерных наук им. И.И. Воровича Южного федерального университета, руководитель проекта PascalABC.NET. Основные научные интересы: разработка компиляторов, управление прорезающей функциональностью в программах, теория типов.

Stanislav Stanislavovich MIKHALKOVICH – Candidate of Physical and Mathematical Sciences, Associate Professor, Head of the Department of Computer Science and Computational Experiments at the I.I. Vorovich Institute of Mathematics, Mechanics and Computer Science of the Southern Federal University, Head of the PascalABC.NET project. Main research interests: compiler development, management of crosscutting concerns in programs, type theory.

email: miks@sfedu.ru

ORCID: 0000-0003-0373-3886

Материал поступил в редакцию 9 ноября 2024 года