Эффективность по памяти и времени легковесных парсеров с разной детализацией языка Go

Д.С. Дроздов, С.С. Михалкович

Южный федеральный университет

Аннотация. Рассматривается подход к созданию семейства легковесных грамматик для языка Go со специальным символом Any, обозначающим пропускаемую часть программы [1]. Дается формальное определение более детализированной грамматики, приводятся примеры увеличения детализации правил грамматики. Проводится 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, приведены преимущества использования легковесных грамматик с символом Any по сравнению с полными грамматиками. Представлен пример использования легковесной грамматики для определения сложности кода. Помимо этого, полученные результаты могут быть 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 с символом Any

Memory and Time Efficiency of Lightweight Parsers with Different Go Language Granularity

D.S. Drozdov, S.S. Mikhalkovich

Southern Federal University

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, grammar with Any symbol

1. Введение

Первым этапом компиляции является парсинг, основанный на грамматике. Однако парсер строит AST-дерево только для правильных программ заданной версии языка. Кроме того, иногда требуется пре- и постобработка текста программы для корректной работы парсера. Например, в языке Go, который рассматривается 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 алгоритмическая привязка, которая позволяет разметить код и перемещаться по нему даже после изменения текста [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] приводятся алгоритмы LR(1) и LL(1) разборов с символом Any, который является аналогом «воды» островных грамматик, и даются примеры легковесных грамматик для C#, PascalABC.NET и Java. В текущем исследовании данные алгоритмы применяются 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.

---

[1] https://go.dev/ref/spec

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

[3] https://github.com/antlr/grammars-v4

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

В данной работе получены следующие результаты:

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

В разделе 2 описываются преимущества и основные принципы работы легковесных парсеров. Раздел 3 содержит формальное определение более детализированной грамматики и описание трех созданных грамматик Go с разной степенью детализации. Дается определение трудночитаемых функций и приводится легковесная грамматика для поиска таковых. В разделе 4 проведена валидация созданных грамматик и оценка их эффективности по памяти и времени работы. В разделе 5 дано описание работ, близких к данной.

2. Легковесные парсеры

В работе [1] рассматриваются 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 подтверждается результатами экспериментов.

В данном исследовании обрабатываются только сигнатуры функций, поэтому их тела и некоторые глобальные конструкции описываются через Any. Заметим, что если функция 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 (см. 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 ']'. В результате код будет разбираться корректно, а дженерики сохранятся в тексте узла Any.

3. Грамматики Go с разной детализацией

Определение. Обозначим через a_i — произвольный символ грамматики (терминал или нетерминал). Рассмотрим правило $S \to a_1 \dots a_{k-1}$ **Any** $a_{k+p+1} \dots a_n$, для которого выполнены следующие условия:

1. $a_1 \neq$ Any и не существует вывода $a_1 \Rightarrow^*$ Any α, где α — цепочка символов;
2. $a_n \neq$ Any и не существует вывода $a_n \Rightarrow^* \alpha$ Any, где α — цепочка символов.

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 символ Any должен быть обрамлен хотя бы одним символом слева и справа, в противном случае правило может захватывать символы без ограничения [1].

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

Определение. Грамматика языка G_1 называется *менее детализированной* версией грамматики G_2, если хотя бы одно правило G_1 является менее детализированной версией правила G_2, а остальные правила совпадают.

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

В данном 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*. 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 рисунке 2.

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

Рис. 2. Функция и метод в Go

Первая созданная грамматика является наиболее общей. На рисунке 3 приведен ключевой фрагмент данной грамматики, которая позволяет находить методы (f_reciever непуст) и функции (f_reciever пуст). Как видно, в каждом правиле используется 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. 3. 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.

Вторая грамматика имеет более детальное определение аргументов и возвращаемого значения. На рисунке 4 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. 4. 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].

Детализация аргументов и типа возвращаемого значения потребовала описания типов Go. Во второй грамматике определены конструкции, представленные на рисунке 5.

```
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. 5. 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 рисунке 6 показано, что в правилах map и chan вместо Any теперь используется go_type.

```
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' '['go_type']' go_type
chan = 'chan' go_type | 'chan<-' go_type | '<-chan' go_type
```

Рис. 6. 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 рисунке 7 слева показано синтаксическое дерево для функции

func F(map[chan<- map[chan<- int32]int]int)

по третьей грамматике, а справа — по первой. Видно, насколько меняется максимальная глубина дерева при изменении детализации грамматики.

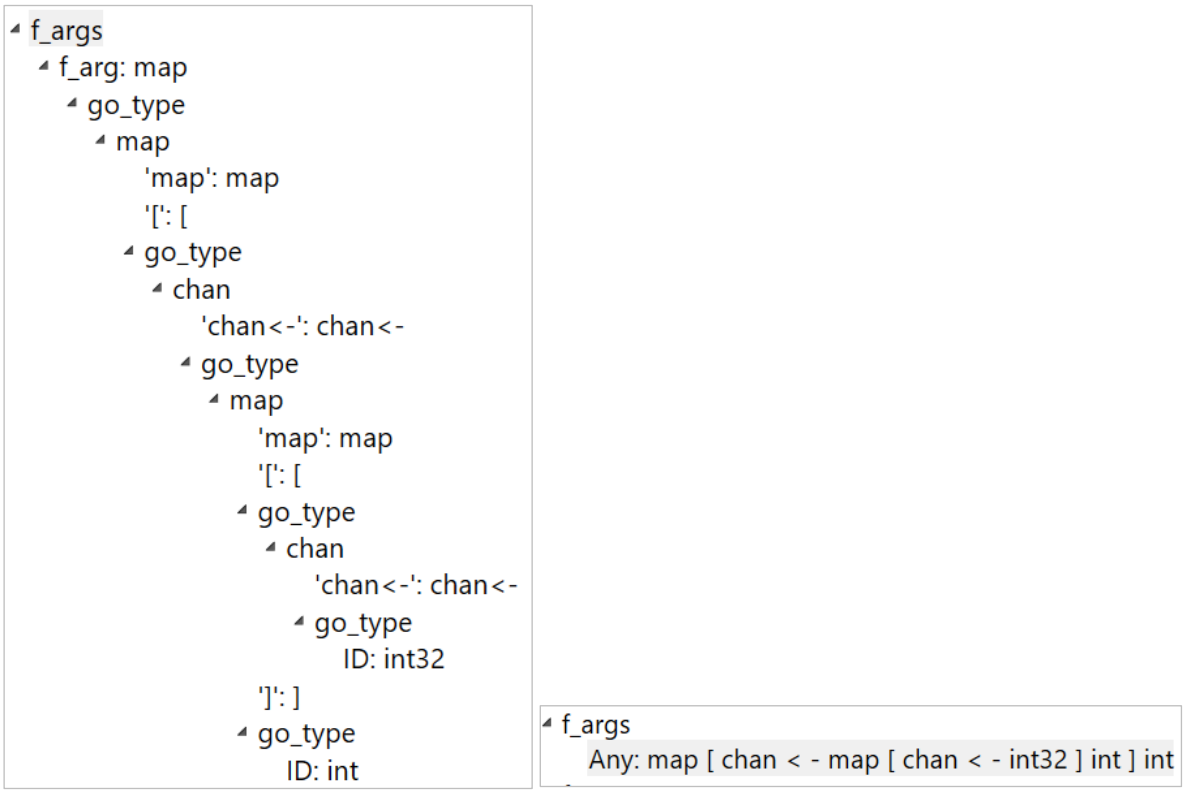

Рис. 7. Деревья, построенные по третьей и первой грамматикам

Первое 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 [9]. 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. 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:

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

Тогда сложность аргумента или типа возвращаемого значения вычисляется как глубина его синтаксического дерева, построенного по третьей грамматики. 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. 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 рисунке 8.

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

Рис. 8. 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 определяется функция с аргументом, имеющем самую высокую сложность (см. рис. 9). Она рассчитывается аналогично предыдущему примеру и равна пяти.

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

Рис. 9. Аргумент с высокой сложностью

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

4. Эффективность грамматик

Для сравнения парсеров в данной работе используется семь репозиториев с общей кодовой базой из более чем 43 000 файлов и 180 000 функций.

На первом этапе все грамматики прошли автоматическую проверку полным парсером Go. Результаты приведены в таблице 2. В работе [7] показано, что вторая грамматика правильно разбирает 100% аргументов из тестового набора. В данном исследовании аналогичный результат получен для менее и более детализированных грамматик (первая и третья).

Таблица. 2. Результаты валидации 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	
mts-lmi	695	695		695		695	

На втором этапе выполнены замеры времени работы и потребляемой памяти. Установлено, что при увеличении детализации грамматики от первой ко второй время работы увеличивается в среднем на 27%. Наибольший абсолютный прирост составил 10 секунд для репозитория

Kubernetes. Добавление дополнительной детализации аргументов (от второй к третьей грамматике) увеличивает время работы в среднем на 2%.

Показано, что при переходе от первой грамматики к более детальной второй расход памяти увеличивается в среднем на 34%. При этом сильнее всего потребление возросло для проекта Chainlink и Docker CE — на 340 и 490 Мегабайт соответственно. Наиболее детализированная грамматика (третья) требует еще на 6% больше памяти.

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 рисунке 10 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.

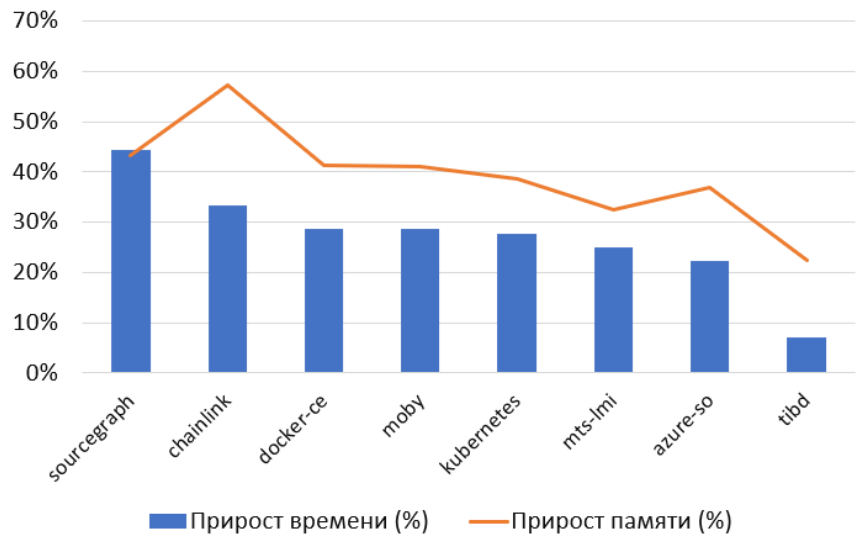

Рис. 10. Увеличение потребления ресурсов при использовании третьей грамматики по сравнению с первой

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 [10]. Во-первых, данные о затратах на разбор позволяют вычислить чистое время переприязки. 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. Данная проблема рассматривалась в работе [11], 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 применяется машинное обучение на основе набора данных о том, как разработчик просматривает проект. В статье [12] приводится 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 [13] 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».

6. 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 валидации 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.

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

Литература

1. Goloveshkin A.V., Mikhalkovich S.S. Tolerant parsing using modified LR(1) and LL(1) algorithms with embedded “Any” symbol // Труды ИСП РАН, 2019. — Vol. 31. P. 7–28. — https://doi.org/10.15514/ISPRAS-2019-31(3)-1
2. Bodner J. Learning Go. An Idiomatic Approach to Real-World Go Programming. — O’Reilly Media Inc., 2024. — p. 353.
3. Головешкин А.В., Михалкович С.С. Разметка сквозных функциональностей в коде программы // Научный сервис в сети Интернет: труды XXI Всероссийской научной конференции, 2019. — С. 245–256. — https://doi.org/10.20948/abrau-2019-83
4. Malevannyy M., Mikhalkovich S. Context-based model for concern markup of a source code // Труды ИСП РАН, 2016. — Vol. 28. P. 63–78. — https://doi.org/10.15514/ISPRAS-2016-28(2)-4
5. Moonen L. Generating Robust Parsers Using Island Grammars // Proceedings of the 8th Working Conference on Reverse Engineering, 2001. — P. 13–22. — https://doi.org/10.1109/WCRE.2001.957806
6. Moonen L. Lightweight Impact Analysis using Island Grammars // Proceedings of the 10th International Workshop on Program Comprehension, 2002. — P. 219–228. — https://doi.org/10.1109/WPC.2002.1021343
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», 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», 2018. — С. 53–56.
9. Мельцов В.Ю. Лекции по теории автоматов. — ВятГУ, 2010. — Часть 2. С. 24.
10. 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. — https://doi.org/10.1016/j.procs.2021.10.024
11. Robillard M., Murphy G. Concern graphs: finding and describing concerns using structural program dependencies // Proceedings of the 24th international conference on Software engineering, 2002. — P. 406–416. — https://doi.org/10.1109/ICSE.2002.1007986
12. Paltenghi M., Pandita R. et al. Extracting Meaningful Attention on Source Code: An Empirical Study of Developer and Neural Model Code Exploration // arXiv:2210.05506 [cs.SE], 2022. — https://doi.org/10.48550/arXiv.2210.05506

13. Okuda K., Chiba S. Lake symbols for island parsing // arXiv:2010.16306 [cs.PL], 2020. — https://doi.org/10.48550/arXiv.2010.16306

References

1. Goloveshkin A.V., Mikhalkovich S.S. Tolerant parsing using modified LR(1) and LL(1) algorithms with embedded “Any” symbol // Trudy ISP RAN, 2019. — Vol. 31. P. 7–28. — https://doi.org/10.15514/ISPRAS-2019-31(3)-1
2. Bodner J. Learning Go. An Idiomatic Approach to Real-World Go Programming. — O’Reilly Media Inc., 2024. — p. 353.
3. Goloveshkin A.V., Mikhalkovich S.S. Razmetka skvoznykh funktsionalnostei v kode programmy // Nauchnyi servis v seti Internet: trudy XXI Vserossiiskoi nauchnoi konferentsii, 2019. — S. 245–256. — https://doi.org/10.20948/abrau-2019-83
4. Malevannyy M., Mikhalkovich S. Context-based model for concern markup of a source code // Trudy ISP RAN, 2016. — Vol. 28. P. 63–78. — https://doi.org/10.15514/ISPRAS-2016-28(2)-4
5. Moonen L. Generating Robust Parsers Using Island Grammars // Proceedings of the 8th Working Conference on Reverse Engineering, 2001. — P. 13–22. — https://doi.org/10.1109/WCRE.2001.957806
6. Moonen L. Lightweight Impact Analysis using Island Grammars // Proceedings of the 10th International Workshop on Program Comprehension, 2002. — P. 219–228. — https://doi.org/10.1109/WPC.2002.1021343
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», 2024. — S. 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», 2018. — S. 53–56.
9. Meltsov V.Iu. Lektsii po teorii avtomatov. — ViatGU, 2010. — Chast 2. S. 24.
10. 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. — https://doi.org/10.1016/j.procs.2021.10.024
11. Robillard M., Murphy G. Concern graphs: finding and describing concerns using structural program dependencies // Proceedings of the 24th international conference on Software engineering, 2002. — P. 406–416. — https://doi.org/10.1109/ICSE.2002.1007986
12. Paltenghi M., Pandita R. et al. Extracting Meaningful Attention on Source Code: An Empirical Study of Developer and Neural Model Code Exploration

// arXiv:2210.05506 [cs.SE], 2022. — https://doi.org/10.48550/arXiv.2210.05506
13. Okuda K., Chiba S. Lake symbols for island parsing // arXiv:2010.16306 [cs.PL], 2020. — https://doi.org/10.48550/arXiv.2010.16306