1. Министерство образования и науки Российской Федерации
2. Санкт-Петербургский Политехнический Университет Петра Великого
3. —
4. Институт кибербезопасности и защиты информации

**КУРСОВАЯ ЛАБОРАТОРНАЯ РАБОТА**

1. «Разработка валидатора языка *GoLang*»
2. по дисциплине «Формальные грамматики и теория компиляторов»
3. Выполнили Веселов Б. В.
4. студенты гр. 4851003/90801 Львов А. В.

1. Руководитель
2. старший преподаватель Семьянов П. В.
4. Санкт-Петербург
5. 2022

Оглавление

[1 ВВЕДЕНИЕ 3](#_Toc117629990)

[1.1 Применение валидаторов 3](#_Toc117629991)

[1.2 Выбор языка программирования 3](#_Toc117629992)

[1.3 Спецификация языка *GoLang* 4](#_Toc117629993)

[2 Практическая часть 4](#_Toc117629994)

[*2.1* Ключевые конструкции языка *GoLang* 4](#_Toc117629995)

[2.1.1 Используемый алфавит 4](#_Toc117629996)

[2.1.2 Пакеты 4](#_Toc117629997)

[2.1.3 Объектно-ориентированное программирование 5](#_Toc117629998)

[2.1.4 Ошибки 5](#_Toc117629999)

[2.1.5 Программные модули 6](#_Toc117630000)

[2.1.6 Встроенные типы данных 6](#_Toc117630001)

[2.1.7 Строение комментариев 6](#_Toc117630002)

[2.1.8 Объявление переменных 7](#_Toc117630003)

[2.1.9 Видимость идентификатора 7](#_Toc117630004)

[2.1.10 Присваивание значений 8](#_Toc117630005)

[2.1.11 Объявление функций 8](#_Toc117630006)

[2.1.12 Конструкции ветвления 8](#_Toc117630007)

[*2.2* *Разработка валидатора* 9](#_Toc117630008)

[2.2.1 Интерпретация расширенных форм Бэкуса-Наура 9](#_Toc117630009)

[2.2.2 Автоматическое выделение конструкций языка 10](#_Toc117630010)

[2.2.3 Разработка основных конструкций языка 12](#_Toc117630011)

[2.2.4 Конфликты грамматического разбора и их решение 22](#_Toc117630012)

[2.2.5 Ограничения в работе валидатора. Нерешенные конфликты 26](#_Toc117630013)

[3 ЗАКЛЮЧЕНИЕ 28](#_Toc117630014)

[4 ИСПОЛЬЗОВАННЫЕ ИСТОЧНИКИ 29](#_Toc117630015)

# ВВЕДЕНИЕ

## Применение валидаторов

Валидаторы в языках программирования применяются для анализа кода на наличие в нем ошибок, которые могут привести к неправильному выводу информации или к сбою в программе. Вне зависимости от профессионализма программиста с увеличением количества строк кода увеличивается возможность допущения ошибок по невнимательности, которые могут существенно повлиять на работоспособность программы.

Валидаторы присутствуют во встроенном функционале большинства современных *IDE* и помогают разработчикам писать чистый и рабочий код. Основная задача таких программных средств – обнаруживать ошибки в программном коде и своевременно уведомлять об этом программиста.

Программа может пропускать некритические ошибки, однако наиболее правильным подходом при разработке современного программного обеспечения, будет стремление свести количество ошибок и недочетов к минимуму. Любой известный недочет или уязвимость в долгосрочной перспективе может привести к полноценному сбою в работе программы.

## Выбор языка программирования

Для разработки собственного валидатора был выбран востребованный в современном мире язык программирования *GoLang*, разработанный компанией *Google* в 2009 году. Преимуществами данного языка является знакомый *C-*подобный синтаксис, а также простота и компактность в парсинге, позволяющая проводить быстрый и доступный анализ с применением средств автоматизации. Семантика языка *Go* обычно *C*-подобна. Это компилируемый, статически типизированный, процедурный язык с указателями.

## Спецификация языка *GoLang*

Несмотря на схожие черты с *C*, *Go* имеет значительные отличия:

* Отсутствие арифметики указателей
* Границы массива всегда проверяются
* Отсутствие неявных числовых преобразований

Также для языка *GoLang* доступна подробная спецификация, описывающая все основные аспекты языка, такие, как модули, переменные, объекты, функции, и. т. д. Разработка собственного валидатора исходных кодов велась в соответствии с документацией для достижения максимального соответствия между валидатором и актуальной структурой языка.

# Практическая часть

## Ключевые конструкции языка *GoLang*

Разработка валидатора началась с изучения ключевых конструкций выбранного языка:

### Используемый алфавит

Исходный код языка *GoLang* представляет собой текст *Unicode*, закодированный в *UTF-8*. Конструкции языка пишутся символами латиницы. Строковые константы (руны) могут содержать произвольные символы таблицы *Unicode*.

### Пакеты

Программы *Go* создаются путем связывания пакетов. Пакет, в свою очередь, создается из одного или нескольких исходных файлов, которые вместе объявляют константы, типы, переменные и функции, принадлежащие пакету и доступные во всех файлах того же пакета. Эти элементы разрешается экспортировать и использовать в другом пакете.

### Объектно-ориентированное программирование

*Go* использует необычный подход к объектно-ориентированному программированию, позволяя использовать методы не только для классов, но без какой-либо формы наследования на основе типов, такой как создание подклассов. Вместо этого у *Go* есть интерфейсы – набор методов. Например, на рисунке ниже представлено определение интерфейса *Hash* из стандартной библиотеки.

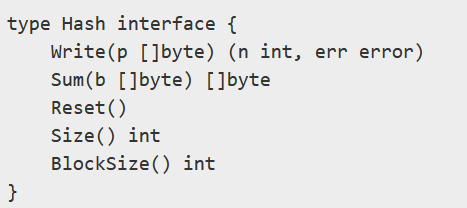


Рисунок 1 – Определение интерфейса.

### Ошибки

*Go* не имеет средств исключений в общепринятом смысле, то есть нет структуры управления, связанной с обработкой ошибок. Но в то же время *Go* предоставляет механизмы для обработки исключительных ситуаций, например, деление на ноль. Особенностью языка для обработки ошибок является предопределенный тип интерфейса, называемый *error*, который представляет значение, которое имеет метод *Error*, возвращающий строку.

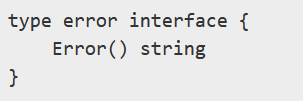


Рисунок 2 – Интерфейс *error* из стандартной библиотеки *GoLang*.

Таким образом, библиотеки используют тип *error*, чтобы вернуть описание ошибки в виде строки.

### Программные модули

Модуль — это набор пакетов *Go*, хранящихся в файловом дереве с файлом *go.mod* в корне. Файл *go.mod* определяет путь модуля, который также является путем импорта, используемым для корневого каталога, а также его требования к зависимостям, которые являются другими модулями, необходимыми для успешной сборки. Каждое требование зависимости записывается в виде пути к модулю и определенной семантической версии.

### Встроенные типы данных

Встроенные типы языка *GoLang* разделяются на четыре категории:

1. Базовые типы. К ним относят числа (целые и с плавающей точкой), строки и логические (булевые) значения.
2. Агрегированные типы. К ним относят массивы и структуры (записи).
3. Ссылочные типы. К ним относят указатели, срезы, ассоциативные массивы, функции и каналы.
4. Интерфейсные типы. Данную категорию составляют интерфейсы, которые были отдельно рассмотрены раннее.

### Строение комментариев

Комментарии служат программной документацией.

Поддерживаются комментарии, аналогичные комментариям языка *C:*

* Комментарии к строке. Начинаются с последовательности символов // и заканчиваются в конце строки.
* Общие комментарии. Начинаются с последовательности символов /\* и заканчиваются первой последовательностью символов \*/.

Комментарий не может начинаться внутри рунического или строкового литерала или внутри комментария. Общий комментарий, не содержащий символов новой строки, воспринимается парсером как пробел. Любой другой комментарий воспринимается, как новая строка.

### Объявление переменных

Объявление переменной начинается с ключевого слова *var* с последующим именем переменной и ее типом. Вместе с определением переменной может происходить ее инициализация, например *var a int = 100*.

Объявление собственных типов данных начинается с ключевого слова *type*, за которым следует имя типа, например, *type Person*.

Text

Description automatically generated with low confidence

Рисунок 1 – Объявление переменных без инициализации.

Chart

Description automatically generated with medium confidence

Рисунок 2 – Объявление переменных с последующей инициализацией.

### Видимость идентификатора

В *Go* используется необычный подход к определению видимости идентификатора. Например, в отличие от ключевых слов *private* и *public*, в *Go* само имя отражает информацию о поле видимости, а именно – регистр первой буквы идентификатора говорит разработчику, в каком поле находится идентификатор. Если начальный символ является заглавной буквой, то идентификатор экспортируется (общедоступный), в противном случае это не так. Данное правило распространяется на все переменные, типы, функции, методы.

### Присваивание значений

Присваивание в *Go* происходит при помощи символа . Определение переменной может совмещаться с ее инициализацией. Компилятор *Go* строго разграничивает определения и присваивания и запрещает переопределение переменной с тем же именем. В пределах одной единицы трансляции переменная может встретиться слева от знака только один раз.

Graphical user interface, text

Description automatically generated

Рисунок 3 – Присваивание значений переменной.

### Объявление функций

Для объявления функции в языке *Go* необходимо написать ключевое слово *func,* имя объявляемой функции, список входных параметров, список возвращаемых параметров и тело функции. Функции в *GoLang* могут возвращать неограниченное количество параметров, завернутых в кортеж с использованием круглых скобок.

A picture containing chart

Description automatically generated

Рисунок 4 – Объявление функции.

### Конструкции ветвления

Для объявления блока условного перехода *if* или *if-else,* условный блок не следует оборачивать круглыми скобками, в отличие от языков *C/C++*. Фигурные скобки для блока выполнения, напротив, являются обязательными для языка *GoLang.*

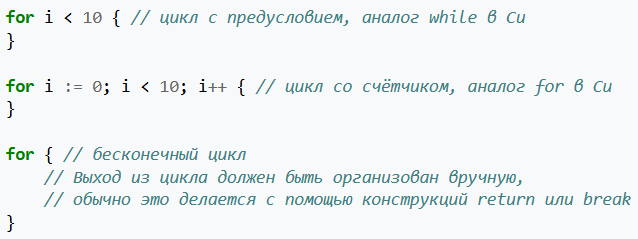


Рисунок 5 – Объявление условных конструкций.

## *Разработка валидатора*

Разработка валидатора велась с использованием генератора синтаксических анализаторов *Lex/Yacc.*

### Интерпретация расширенных форм Бэкуса-Наура

Для разработки валидатора было необходимо разработать грамматику понятную *Yacc/Bison*, так как данные генераторы не поддерживают написание грамматик с использованием *РФБН*.

Расширенная форма Бэкуса-Наура представляет собой доработку нормальной нотации Бэкуса – Наура (*БНФ*) путем улучшения синтаксиса, позволившего упростить и сократить в объеме используемые конструкции, сохраняя при этом лаконичность и простоту грамматики. Применяются следующие правила:

* Операнд () определяет «группу захвата», и используется, как обычные скобки. Для обработки данного операнда в *Yacc/Bison* необходимо разделить ключевые конструкции на отдельные пути в грамматике:
* Операнд [] означает опцию – наличие, либо отсутствие данного токена в грамматике. Выражения, использующие данный операнд, раскрываются путем разделения грамматики на два возможных пути:
* Операнд {} означает наличие выбранного токена в грамматике от нуля до бесконечности раз. Данный операнд преимущественно применяется в определении списков выражений, типов, операторов. Такие конструкции раскрываются путем добавления дополнительной абстракции – списка, которая определяется рекурсивно через себя:
* Операнд | - “или” присутствует в *Yacc/Bison* по умолчанию и изменений не требует.

### Автоматическое выделение конструкций языка

В *Go* применяется механизм автоматического выделения конструкций языка путем добавления граничных символов точек с запятой во время лексического разбора, который подвергается активной критике. Данный механизм приводит к тому, что некоторые формы записи операторов, вызовов функций и списков могут быть неправильно восприняты компилятором.

При разработке собственного валидатора данный механизм был успешно реализован путем оптимизации алгоритма разбора на фазе лексера.

Для автоматического добавления разделяющих символов в необходимых местах программы, в фазу лексического разбора была добавлена дополнительная функция проверки токенов.

Алгоритм разработанной функции, следующий: в функции имеется статическая переменная - флаг, показывающая, является ли последний токен одним из токенов, после которого в конце строки необходимо добавлять точку с запятой. При нахождении одного из таких токенов, контрольная переменная выставляется со значением *True*. При нахождении любого другого токена, выставляется значение *False.* При нахождении перевода строки, парсеру на вход передается *‘;’*.

Символ-разделитель (точка с запятой) автоматически добавляется, если последний токен в строке – *identifier* (имя переменной), строковая константа, число, *imaginary* (встроенная в язык *Go* поддержка комплексных чисел), или *руна* (строковая константа, поддерживающая символы *Unicode*), токены *break*, *continue*, *fallthrough*, *return*, операторы инкремента и декремента, и все возможные закрывающие скобки (‘)’, ‘]’, ‘}’).

int check(int token) {  
 static unsigned insert = 0;  
  
 */\* Если нашли перевод строки и флаг выставлен, то возвращаем ; и опускаем флаг. \*/* if(token == '\n') {  
 if(insert == 1) {  
 fprintf(lexOutput, ";\n");  
 insert = 0;  
 return ';';  
 } else {  
 */\* Иначе возвращаем перевод строки. \*/* fprintf(lexOutput, "\n");  
 return '\n';  
 }  
 }  
  
 */\* Если находим один из токенов, для которых нужно   
 добавлять ; в конце линии, поднимаем флаг. \*/* if(token == IDENTIFIER || token == INTEGER || token == FLOAT ||  
 token == IMAGINARY || token == RUNE || token == STRING ||  
 token == BREAK || token == CONTINUE || token == FALLTHROUGH ||  
 token == RETURN || token == INCREMENT || token == DECREMENT ||  
 token == ')' || token == ']' || token == '}') {  
 insert = 1;  
 } else {  
 */\* В случае обычного токена, опускаем флаг. \*/* insert = 0;  
 }  
 return token;  
}

### Разработка основных конструкций языка

#### Строение исходного файла

Исходный файл на языке *GoLang* строится из объявления пакета, списка импортируемых пакетов и списка определений верхнего уровня, который может быть пустым:

SourceFile:  
 PackageClause ';' ImportDecls TopLevelDecls ;

Объявление пакета производится с использованием ключевого слова *Package* и имени пакета:

PackageClause : "package" Identifier ;

Список импортируемых пакетов объявляется с использованием ключевого слова *Import* и может состоять как из одного пакета, так и из нескольких пакетов, заключенных в скобки:

ImportDecl : "import" ImportSpec | "import" '(' ImportSpecs ')' ;   
  
 Можно импортировать пакеты из глобального пространства имен, из текущего пакета и из конкретного пакета по его имени:

ImportSpec : String | '.' String | Identifier String ;

Определение верхнего уровня представляет из себя определение константы, пользовательского типа, глобальной переменной, функции или метода.

TopLevelDecl :  
 ConstDecl   
 | TypeDecl   
 | VarDecl   
 | FunctionDecl   
 | MethodDecl ;

Определение константы сопровождается ключевым словом *const* по аналогии с языками *C/C++*.

ConstDecl :   
 "const" IdentifierList "=" ExpressionList   
 | "const" IdentifierList Type "=" ExpressionList ;

const World = "世界"

Для определения переменной, как уже было сказано выше, используется ключевое слово *var*.

VarDecl :   
 "var" IdentifierList Type "=" ExpressionList  
 | "var" IdentifierList "=" ExpressionList ;

var i, j int = 1, 2

Для определения типа используется слово *type*. Это может быть как добавление псевдонима к другим существующим типам (аналогично *typedef/using* для языков *C/C++),* так и создание собственного типа путем объединения одного или нескольких существующих:

TypeDecl :  
 "type" AliasDecl  
 | "type" TypeDef ;

type PublicKey []byte

type foo struct{}  
type bar = foo

#### Определение типа

Определение типа возможно через вложенный тип, или через литерал типа.

Type : TypeOperandName | TypeLit ;  
  
TypeOperandName : Identifier | Identifier '.' Identifier;

Литерал типа может являться массивом, структурой данных, указателем, функцией, интерфейсом, срезом, хеш-таблицей или каналом.

TypeLit :  
 ArrayType   
 | StructType   
 | PointerType   
 | FunctionType   
 | InterfaceType   
 | SliceType   
 | MapType   
 | ChannelType ;

Литерал типа «массив» состоит из квадратных скобок с указанием размера массива, а также типа элементов:

ArrayType : '[' ArrayLength ']' ElementType ;  
  
ArrayLength : Expression ;  
   
ElementType : Type ;

Литерал типа «структура данных» состоит из ключевого слова *Struct*, и перечисления полей структуры в фигурных скобках:

StructType : "struct" '{' FieldDecls '}' ;

Грамматика поля структуры данных порождала конфликты грамматического разбора и представлено в главе «Конфликты грамматического разбора и их решение» далее.

Литерал типа «указатель» состоит из символа звезды и типа указателя:

PointerType : '\*' Type ;

Литерал типа «функция» состоит из ключевого слова *Func* и сигнатуры функции:

FunctionType : "func" Signature ;

Литерал типа «интерфейс» состоит из ключевого слова *Interface* и перечисления полей интерфейса в фигурных скобках. Поля интерфейса состоят из методов объекта или типов данных:

InterfaceType : "interface" '{' InterfaceElems '}' ;   
   
InterfaceElem : MethodElem | TypeElem ;  
  
MethodElem : Identifier Signature ;

Литерал типа «срез» образуется из квадратных скобок и типа элементов среза:

SliceType : '[' ']' ElementType ;

Литерал типа «хеш-таблица» состоит из ключевого слова *Map*, типа ключей таблицы в квадратных скобках и типа значений таблицы:

MapType : "map" '[' Type ']' Type ;

#### Описание литерала типа «канал» порождало конфликты грамматического разбора, и описано в главе «Конфликты грамматического разбора и их решение» далее.

#### Определение функций и методов

Определение функции состоит из ключевого слова *func,* имени функции, входных и возвращаемых параметров и тела функции. Для generic функций после имени добавляется также описание параметров используемых типов в квадратных скобках. Поддерживается объявление функции без тела, по аналогии с *C/C++.*

FunctionDecl :  
 "func" Identifier Signature *// Объявление*   
 | "func" Identifier TypeParameters Signature *// Объявление*   
 | "func" Identifier Signature FunctionBody *// Определение*  
 | "func" Identifier TypeParameters Signature FunctionBody ; *// Определение*

TypeParameters: '[' TypeParamList ']' | '[' TypeParamList ',' ']' ;

Signature : Parameters | Parameters Result ;

Result : Parameters | Type ;

В определении метода, между именем функции и входными параметрами добавляется также список параметров объекта, для которого вызывается метод.

MethodDecl :  
 "func" **Receiver** Identifier Parameters Result *// Объявление*  
 | "func" **Receiver** Identifier Parameters Result FunctionBody ; *// Определение*

Receiver : Parameters ;

Грамматика определения сигнатуры функции через цепочки входных параметров порождала конфликты грамматического разбора, и описано в главе «Конфликты грамматического разбора и их решение» далее.

#### Операторы в GoLang

Тело функции или метода состоит из списка операторов. Оператором может являться определение константы, типа или переменной, операции ветвления или цикла, метка перехода оператора *goto*, оператор выражения, отправка данных по встроенному каналу, использование операторов инкремента или декремента, операции с ключевыми конструкциями *return*, *break*, *continue*, *goto*, *fallthrough*, *switch*, *select*, *defer* или ничего.

Statement :  
 Declaration | LabeledStmt | SimpleStmt | GoStmt | ReturnStmt | BeakStmt | ContinueStmt | GotoStmt | FallthroughStmt | Block | IfStmt | SwitchStmt | SelectStmt | ForStmt | DeferStmt ;

Цепочка «Оператор объявления» образуется из объявления константы, типа или переменной:

Declaration : ConstDecl | TypeDecl | VarDecl ;

Цепочка «Оператор метки перехода», используемая в связке с оператором *Goto*, образуется из маркера и выражения через точку с запятой:

LabeledStmt : Identifier ':' Statement;

Цепочка «Простое оператор» образуется из пустого оператора, оператора выражения, оператора отправки по типу «канал», оператора инкрементирования или декрементирования, оператора присваивания или укороченного объявления переменной:

SimpleStmt : EmptyStmt | ExpressionStmt | SendStmt | IncDecStmt | Assignment | ShortVarDecl ;

ExpressionStmt : Expression ;  
  
SendStmt : Channel "<-" Expression ;   
   
IncDecStmt : Expression "++" | Expression "--";  
  
Assignment :  
 ExpressionList ":=" ExpressionList  
 | ExpressionList "=" ExpressionList ;

В отличие от языков *C/C++*, в языке *Go* не реализована поддержка префиксного инкремента и декремента.

Цепочка «Оператор *Go*», описывающая вычисления в отдельном потоке (горутине), состоит из ключевого слова *Go*, и выражения:

GoStmt : "go" Expression ;

Цепочка «Оператор *Return*», описывающая возврат из функции, состоит из ключевого слова *Return*, и опциональных значений возврата:

ReturnStmt : "return" | "return" ExpressionList ;

Цепочка «Оператор *Break*», описывающая оператор прерывания цикла, состоит из ключевого слова *Break*, и опциональной метки перехода. В отличие от классической реализации данного оператора в языках *C/C+*, *Break* в *Go* позволяет реализовывать не только выход из ближайшего цикла, но и осуществление перехода по метке, по аналогии с оператором *Goto*:

BreakStmt : "break" | "break" Label ;

#### Цепочка «Оператор Continue», описывающая оператор продолжения цикла, состоит из ключевого слова Continue и опциональной метки перехода, по аналогии с Break:

ContinueStmt : "continue" | "continue" Label ;

Цепочка «Оператор *Goto*», описывающая оператор безусловного перехода, состоит из ключевого слова *Goto* и метки перехода:

GotoStmt : "goto" Label ;

Оператор ветвления состоит из ключевого слова *If*, выражения и блока операторов. Как уже было сказано выше, оператор ветвления в *Go* не требует добавления круглых скобок вокруг условного выражения. Также оператор ветвления позволяет создавать конструкции с применением простого выражения внутри условия – например, выносить значение условия в отдельную переменную, и сразу же проводить сравнение:

IfStmt :  
 "if" Expression Block   
 | "if" SimpleStmt ';' Expression Block  
 | "if" Expression Block ELSE IfStmt  
 | "if" Expression Block ELSE Block  
 | "if" SimpleStmt ';' Expression Block ELSE IfStmt  
 | "if" SimpleStmt ';' Expression Block ELSE Block ;

if x := f(); x < y { ... }

Оператор множественного выбора *Switch* состоит из ключевого слова *Switch,* а также перечисления условий исполнения. Оператор *Switch* в *Go* позволяет сравнивать не только значения выражений, но и значения типов.

SwitchStmt : ExprSwitchStmt | TypeSwitchStmt ;  
  
ExprSwitchStmt :  
 "switch" '{' ExprCaseClauses '}'   
 | "switch" SimpleStmt ';' '{' ExprCaseClauses '}'   
 | "switch" Expression '{' ExprCaseClauses '}'   
 | "switch" SimpleStmt ';' Expression '{' ExprCaseClauses '}' ;

TypeSwitchStmt :  
 "switch" TypeSwitchGuard '{' TypeCaseClauses '}'  
 | "switch" SimpleStmt ';' TypeSwitchGuard '{' TypeCaseClauses '}' ;

Условия исполнения перечисляются с использованием ключевого слова Case. Применяется ключевое слово Default:

ExprSwitchCase : "case" ExpressionList | DEFAULT ;

Оператор *Select*, описывающий выбор операции для отправки по типу «канал», образуется из ключевого слова *Select* и множества операторов:

SelectStmt : "select" '{' CommClauses '}' ;   
   
CommClause : CommCase ':' StatementList ;  
   
CommCase : "case" SendStmt | "case" RecvStmt |"default";

Оператор *Defer*, вызывающий выражение, выполнение которого отложено до момента возврата из окружающей функции, образуется из ключевого слова *Defer* и выражения:

DeferStmt : "defer" Expression ;

Цепочка «Оператор *Fallthrough*», применяемый в операторах множественного выбора *Case*, состоит из ключевого слова *Fallthrough*.

Оператор задания цикла *For* в языке *GoLang* отвечает сразу за все типы циклов – бесконечный цикл, цикл *while*, стандартный цикл for, а также за встроенную в язык возможность итерации по последовательности с ключевым словом *Range*:

ForStmt :  
 "for" Block   
 | "for" Condition Block  
 | "for" ForClause Block   
 | "for" RangeClause Block ;  
   
Condition : Expression ;   
   
RangeClause :  
 "range" Expression   
 | ExpressionList "=" "range" Expression   
 | IdentifierList ":=" "range" Expression   
 | ExpressionList ":=" "range" Expression ;

Описание оператора задания стандартного цикла *For* порождало конфликты грамматики. Финальная грамматика описания данного оператора описана в главе «Конфликты грамматического разбора и их решение» далее.

#### Выражения в GoLang

Выражением в языке Go делятся на унарные и бинарные. Для бинарных выражений предусмотрены стандартные бинарные операторы, аналогичные языкам *C/C++*.

Унарное выражение может быть представлено через первичное выражение или рекурсивно через себя с использованием унарного оператора:

UnaryExpr : PrimaryExpr | unary\_op UnaryExpr ;

Поддерживаются все стандартные унарные операторы: плюс, минус, инверсия, операция и, операция или, операция исключающее или, разыменование и операция отправки по каналу.

Первичное выражение представляется через операнд (элементарные значения в выражении – числа, строки, значения переменной, полученные по имени), а также через другое унарное выражение и селектор (доступ к полям объекта), индекс (обращение к объектам массива), слайс (получение подмасива из масива), приведение к типу, или вызов функции/метода с параметрами.

PrimaryExpr:  
 Operand   
 | UnaryExpr Selector  
 | UnaryExpr Index  
 | UnaryExpr Slice  
 | UnaryExpr TypeAssertion  
 | UnaryExpr Arguments ;

#### Цепочка «Операнд» образуется из выражения в скобках, обращения к вложенному типу или одного из возможных литералов:

Operand : Literal | TypeOperandName | '(' Expression ')' ;

#### Цепочка «Литерал» объединяет простые литералы (целых и десятичных чисел, сроковых литералов и рун, литералов комплексных чисел), сложные литералы – встроенных объявлений структур данных, массивов, срезов и хеш-таблиц и встроенные объявления функций функций.

Literal : BasicLit | CompositeLit | FunctionLit ;  
   
BasicLit : INTEGER | FLOAT | IMAGINARY | RUNE | STRING ;  
   
CompositeLit : LiteralType LiteralValue ;  
   
LiteralType :  
 StructType   
 | ArrayType   
 | '[' MULTIDOT ']' ElementType   
 | SliceType   
 | MapType   
 | TypeOperandName ;  
   
LiteralValue: '{''}' | '{' ElementList '}' | '{' ElementList ',' '}' ;

FunctionLit : "func" Signature FunctionBody ;

#### Обращение к полям структуры:

Selector : '.' Identifier ;

#### Обращение к объекту в массиве:

Index: '[' Expression ']' ;

#### Получение подмассива-слайса:

Slice : '[' ':' ']'  
 | '[' Expression ':' ']'  
 | '[' ':' Expression ']'  
 | '[' Expression ':' Expression ']'   
 | '[' ':' Expression ':' Expression ']'   
 | '[' Expression ':' Expression ':' Expression ']' ;

#### Как было сказано выше, вызов функции возможен не только с традиционными параметрами в качестве выражений, а также с типами в роли входных параметров:

Arguments :  
 '(' ')'   
 | '(' ExpressionList ')'   
 | '(' ExpressionList "..." ')'   
 | '(' ExpressionList ',' ')'   
 | '(' ExpressionList "..." ',' ')'  
  
 | '(' Type ')'   
 | '(' Type "..." ')'   
 | '(' Type ',' ')'   
 | '(' Type "..." ',' ')'   
  
 | '(' Type ',' ExpressionList ')'   
 | '(' Type ',' ExpressionList "..." ')'   
 | '(' Type ',' ExpressionList ',' ')'  
 | '(' Type ',' ExpressionList "..." ',' ')' ;

Бинарное выражение определяется через два других выражения, разделенных бинарным оператором:

BinaryExpression: Expression binary\_op Expression ;

Поддерживаются все стандартные бинарные операторы:

#### Определение элементарных типов на этапе лексического разбора

Все элементарные типы - числа, строки, руны, имена переменных, ключевые слова, а также комментарии, определяются с использованием регулярных выражений в лексере.

Определение комментариев:

**"//".\*** | **[/][\*][^\*]\*[\*]+([^\*/][^\*]\*[\*]+)\*[/]**

Определение строк и рун:

**\"[^"]\*\"** | **\'[^']\*\'** | **`[^`]\*`**

Определение целых чисел:

**[0-9]+([eE][+-]?[0-9]+)?** | **0x[0-9a-fA-F]+** | **0o[0-7]+** | **0b[0-1]+**

Определение чисел с плавающей точкой:

**([0-9]+)?[\.]([0-9]+)?([eE][+-]?[0-9]+)?**

Определение идентификаторов (имен переменных и названий типов):

**[a-zA-Z\_][a-zA-Z0-9\_]\***

### Конфликты грамматического разбора и их решение

При разработке валидаторов неизбежно возникают конфликты грамматического разбора. Грамматика является неоднозначной (конфликтной), если существует входная строка, которая в соответствии с этой грамматикой может быть разобрана двумя или более различными способами.

Конфликт *shift/reduce* возникает в случае, если грамматика для одного и того же правила допускает одновременно сдвиг и свертку. Конфликты такого типа автоматически разрешаются во время парсинга в *Yacc*.

Конфликт *reduce/reduce* возникает, если существует два или более правил свертки, которые применяются к одной и той же последовательности ввода. Обычно это указывает на серьезную ошибку в грамматике. Генератор *Yacc* не может самостоятельно решать конфликты данного типа. При разрешении конфликта свертка/свертка *Yacc* выбирает для свертки правило, определенное в грамматике раньше, что приводит к полному удалению второго правила. Конфликты данного типа крайне не рекомендуется оставлять в грамматике.

При разработке валидатора для языка *GoLang* возникает множество различных конфликтов вследствие неоднозначности грамматики данного языка. Большинство из них удалось успешно решить способами, описанными ниже.

#### Конфликт при определении грамматики полей в структуре данных

Грамматика языка *Go* позволяет определять поля в структуре двумя основными способами – традиционное определение поля через имя и тип, а также определение встроенного поля через имя типа. Конфликт заключается в том, что цепочка «тип» в определении поля структуры присутствует в двух разных вариантах:

FieldDecl :  
 IdentifierList **Type**  
 | EmbeddedField ;  
   
EmbeddedField :  
 '\*' **Type** TypeArgs ;

Конфликт удалось решить путем объединения различных путей грамматики в один общий путь с созданием дополнительной грамматики *IdentifierListOrNothing* обозначающий список идентификаторов или ничего.

FieldDecl :  
 IdentifierListOrNothing Type   
 | '\*' IdentifierListOrNothing Type ;

#### Конфликт при определении входных параметров функции или метода

Язык *Go* поддерживает передачу в функцию именованных и неименованных параметров встроенных типов, а также типов, выводимых в процессе работы программы. В отличие от многих других языков программирования, например *C/C++*, в *Go* можно сокращать список параметров одинакового типа, перечисляя параметры через запятую. Отдельные наборы параметров также перечисляются через запятую. Помимо передачи конкретных параметров, поддерживается также передача неограниченного числа параметров через оператор троеточия, аналогично языку *C*. Так как и имена параметров и значения их типов в корне являются идентификаторами, данная грамматика порождала очень много конфликтов.

ParameterDecl :  
 Type   
 | IdentifierList Type   
 | "..." Type   
 | IdentifierList MULTIDOT Type ;

Данная проблема была решена путем добавления в грамматику параметра двух дополнительных абстракции и объединения:

ParameterDecl :   
 IdentifierListOrNothing TypeOrNothing   
 | IdentifierListOrNothing "..." TypeOrNothing  
 ;

#### Конфликт при определении канала

*GoLang* имеет встроенный функционал каналов, являющих усложненной заменой примитива многопоточной синхронизации мьютекс. Данный тип позволяет передавать сообщения между несколькими потоками путем записи и чтения канала связи.

Создание канала происходит с использованием ключевого слова *chan*, и типа посылаемых по каналу параметров. Можно также указать направление канала, добавив символ до или после ключевого слова *chan*.

ChannelType :   
 "chan" ElementType   
 | "chan" "<-" ElementType   
 | "<-" "chan" ElementType ;

Данная грамматика порождает конфликты. Грамматика описывает несколько способов как можно собрать *ChannelType* из ключевого слова *chan* и *ElementType*. Кроме того, символ может появиться слева от *ChannelType*, как знак унарной операции.

Конфликт удалось решить путем выделения дополнительной конструкции *ArrowLeftOrNothing* и объединения первых двух путей в грамматике:

ChannelType :  
 "chan" ArrowLeftNothing ElementType  
 | "<-" "chan" ElementType ;

ArrowLeftNothing :   
 | "<-" ;

#### Конфликт при определении цикла for

Определение цикла с конструкцией *for* позволяет опускать каждый из трех выражений внутри условия, в том числе, создание бесконечного цикла конструкцией *for ; ; { … }.* Оригинальная грамматика содержала все 8 возможных вариантов:

ForClause :  
 ';' ';'  
 | SimpleStmt ';' ';'  
 | ';' Expression ';'   
 | ';' ';' SimpleStmt   
 | SimpleStmt ';' Expression ';'   
 | ';' Expression ';' SimpleStmt  
 | SimpleStmt ';' ';' SimpleStmt   
 | SimpleStmt ';' Expression ';' SimpleStmt ;

При проведении анализа оказалось, что такая грамматика является избыточной и порождает конфликты. Цепочка *SimpleStmt* может являться пустым выражением, а значит оригинальную грамматику можно существенно сократить, оставив только два исходных варианта:

ForClause :  
 SimpleStmt ';' ';' SimpleStmt   
 | SimpleStmt ';' Expression ';' SimpleStmt  
 ;

#### Конфликт при обращении к полям структуры по ключу

Из грамматики, определяющей способ обращения к полям в структурах, было убрано избыточное правило *Key: Identifier*. Данное правило порождало конфликт «свертка/свертка», так как грамматика *Key* могла быть представлена через *Identifier* напрямую, а также через *Expression*, которая тоже может являться *Identifier*.

Key :  
 Identifier   
 | Expression *// Could be an Identifier* | LiteralValue   
 ;

### Ограничения в работе валидатора. Нерешенные конфликты

При тестировании разработанного валидатора был обнаружен конфликт, который не удалось решить в условиях использования *Yacc/Bison.*

Конфликт заключается в одновременном создании временного объекта структуры данных, и передачи его для вычисления выражения или при вызове функции:

type pendingDelete struct {  
 service string  
 version string  
 op \*appengine.Operation  
}

append(left, pendingDelete{service: service, version: v.Id, op: op})

В примере выше, функции *append* на вход передается временный объект структуры *pendingDelete* (аналогично *r-value* в языках *C/C++*).

Данная грамматика вводит валидатор в заблуждение: название структуры *pendingDelete* воспринимается валидатором, как имя традиционного аргумента функции, а не как название структуры, для которой создается объект, так как в обоих случаях названия представляются токеном *identifier*. Вследствие чего валидатор сворачивает текущее положение в аргумент вызова функции, и ожидает далее увидеть запятую либо закрывающую круглую скобку, означающую конец списка параметров, а получает открывающую фигурную скобку и список параметров по ключам.

Operand :  
 **Literal**   
 | **Identifier**  
 | '(' Expression ')'   
 ;

Иными словами, грамматика позволяет свернуть токен *Identifier* в цепочку *Operand* двумя способами – через *Literal* и через *Identifier* напрямую. Валидатор на основе *Yacc/Bison* выбирает второй, более высокоуровневый вариант.

CompositeLit :  
 LiteralType LiteralValue ;  
   
LiteralType :  
 StructType

| Identifier

... ;

LiteralValue :  
 '{' '}'   
 | '{' ElementList '}'   
 | '{' ElementList ',' '}'  
 ;

Данные грамматики являются взаимозаменяемыми, и обе крайне важными для корректной работы валидатора. В ходе разработке не удалось сконструировать возможное решение, которое бы позволяло разрешить конфликт.

Данное ограничение можно обойти, создав объект структуры pendingDelete в отдельную переменную и взаимодействовать с созданной переменной, а не со временным объектом:

var elem pendingDelete = pendingDelete{service: service, version: v.Id, op: op}

pending = append(left, elem)

# ЗАКЛЮЧЕНИЕ

В ходе выполнения лабораторной работы был разработан валидатор исходных кодов для языка программирования *GoLang* на основе генератора синтаксических анализаторов *Yacc/Bison,* позволяющий проводить проверку синтаксиса программ на языке *Go*.

Получены ценные навыки разработки грамматик формального разбора современных языков программирования, построения оптимальных грамматических цепочек, разбора лексических последовательностей с использованием языка регулярных выражений, разрешения грамматических конфликтов *shift/reduce* и *reduce/reduce* путем выявления и изменения конфликтующих цепочек разбора.

Разработанный валидатор хорошо показал себя при проведении тестирования на подлинных исходных кодах и почти полностью покрывает актуальную грамматику языка *Go,* из чего можно сделать вывод, что на основе генераторов *Yacc/Bison* возможно реализовать полноценный компилятор для данного языка программирования.

# ИСПОЛЬЗОВАННЫЕ ИСТОЧНИКИ

[1] Официальная спецификация языка *GoLang.*

<https://go.dev/ref/spec>

[2] Официальная спецификация генератора синтаксических анализаторов *GNU - Bison*

<https://www.gnu.org/software/bison/>

[3] Тестировщик регулярных выражений, используемых в лексере

<https://regex101.com>

[4] *Golang* в *Google*: дизайн языка на службе разработки программ https://golang-blog.blogspot.com/2019/11