Содержание

|  |  |  |  |
| --- | --- | --- | --- |
| Введение........................................................................................................ | |  | 2 |
| 1 | Теоретическая часть ........................................................................ |  | 3 |
| 1.1 | Принципы языка Golang.................................................................. |  | 3 |
| 2 | Ход работы ....................................................................................... |  | 4 |
| 2.1 | Лексический анализатор.................................................................. |  | 4 |
| 2.2 | Проблемы при разработке лексического анализатора.................. |  | 5 |
| 2.3 | Синтаксический анализатор............................................................ |  | 6 |
| 2.4 | Анализ конфликтов.......................................................................... |  | 9 |
| 2.5 | Проблемы при разработке синтаксического анализатора............ |  | 14 |
| 2.6 | Примеры работы программы.......................................................... |  | 15 |
| Вывод............................................................................................................. | |  | 16 |
| Приложение 1 | |  |  |
| Приложение 2 | |  |  |

введение

**Цель работы**: проанализировать составляющие языка Golang, исследовать принципы работы синтаксических анализаторов, с помощью которых необходимо создать распознаватель входного языка на основе построенной грамматики.

**Задачи**:

1. Составить лексический анализатор, распознающий токены входного языка.
2. Написать грамматику, определяющую, составляю ли входные символы конструкции языка Golang.
3. **Теоретическая часть**
   1. **Принципы языка Golang**

Go (Golang) — это компилируемый многопоточный язык программирования от Google с открытым исходным кодом. Считается языком общего назначения, но основное применение — разработка веб-сервисов и клиент-серверных приложений. Цель языка — создать современную альтернативу C и C++ и сделать разработку программ в Google более быстрой.

1. Строгая статическая типизация.

При создании переменной тип данных объявляется сразу — например, строка или число. За все время существования переменная может принимать значение только указанного типа. Благодаря явному указанию зависимостей код легко собирать из составных частей, что облегчает разработку крупных проектов.

1. Простой синтаксис.

Простые и понятные конструкции языка позволяют пользователям быстро овладеть написанием кода.

1. Намеренный уход от объектно-ориентированного стиля

Это сделано также для обеспечения простоты кода и быстрой скорости выполнения программ.

1. **ход работы**
   1. **Лексический анализатор**

Для того, чтобы решить вышесказанные задачи, прежде всего необходимо создать лексический анализатор. В нем нужно объявить множество символов и соответствующих им токенов, которые буду передаваться грамматике.

Для создания лексера нужно было проанализировать существующие элементы языка программирования:

Рисунок 1 – лексемы анализатора.

На рисунке представлена часть лексем, необходимых при распознавании языка, такие как операторы, импорт пакетов и интерфейс.

Кроме этого, были созданы лексемы, отвечающие за булевые операции, скобки, символы и типы данных.

Также для обработки языка необходимо было воспользоваться регулярными выражениями:

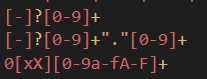


Рисунок 2 – объявление лексемы.

Данные выражения обрабатывают целые числа, числа с плавающей запятой (также они могут быть отрицательными) и шестнадцатеричные числа.

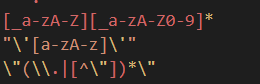


Рисунок 3 – объявление лексемы.

Первое выражение будет отвечать за названия переменных, структур и методов, второе и третье за руны (одиночные символы) и строки соответственно.

Еще одной важной деталью стал токен, отвечающий за распознавание конца файла. <<EOF>> является паттерном Flex и с помощью него можно вернуть синтаксическому анализатору токен, указывающий на конец файла.

* 1. **Проблемы при разработке лексического анализатора**

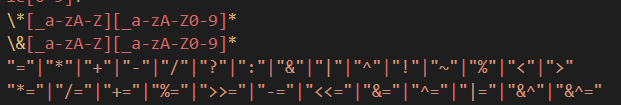
Основной и единственной проблемой, связанной с лексическим анализатором, стало использование символов, повторяющихся в лексемах.

Рисунок 4 – объявление лексем.

Изначально знак умножения и логического И был объявлен под токеном t\_sign. При разработке грамматики понадобилось использование этих же знаков для операции взятия указателя и его разыменовывания, однако парсер не воспринимал знак и соответственно код распознавался не так, как нужно. Для решения такой проблемы было решено написать регулярное выражение, распознающее имена элементов языка с символом «&» перед ними, а также лексему, отвечающую за знак «\*».

* 1. **Синтаксический анализатор**

Язык Golang является очень гибким и удобным для пользователя, ведь ему предоставляется большое количество возможностей для написания кода. Однако в ходе выполнения курсовой работы именно эта гибкость функционала стала ключевым моментом в разработке анализатора, так как для успешного парсинга нужно рассмотреть многочисленные вариации инициализации, расположения и написания элементов языка, избегая большого количества конфликтов.

В первую очередь было обозначено правило Start, которое будет рекурсивно собирать все нетерминалы. Нетерминал Global разворачивается во множество других. Задумка была такова, что в нем будут содержаться все элементы языка, не входящие в тело функции.

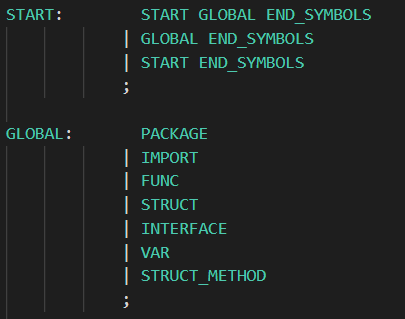


Рисунок 5 – первоначальное описание грамматики.

Следующим этапом было создание обработки функции. В нее могут входить практически все созданные операторы, поэтому было решено создать правило, объединяющее их:

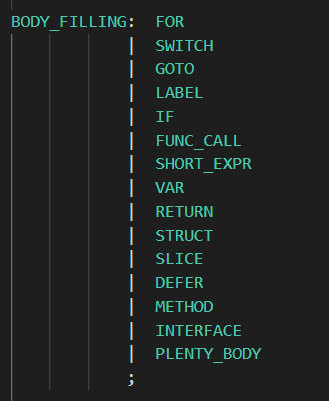


Рисунок 6 – наполнение функций в грамматике.

Далее одной из самых важных задач было создание присваиваний значений другим значениям. На первый взгляд задача кажется очевидной, но, как упоминалось ранее, язык Go обладает вариативностью написания, и это напрямую затрагивает следующий этап создания грамматики:

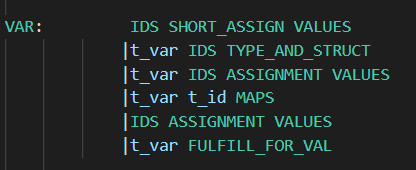


Рисунок 7 – описание присваивания значений.

Присваивание в Golang может быть реализовано классическим способом с помощью знака равенства (ASSIGNMENT), но в этом случае перед переменной должно указываться ключевое слово «var». Тип может указываться явно или же опускаться, так как Go способен определить тип по литералу, которым переменная инициализируется. В другом случае существует вариант сокращенного присваивания с помощью «:=» (SHORT\_ASSIGN), как указано в первом правиле.

И, наконец, практически самой основной единицей разработанной грамматики является VALUE, где перечислены различные элементы языка, которые могут быть присвоены, переданы в качестве параметров или использованы в операторах:

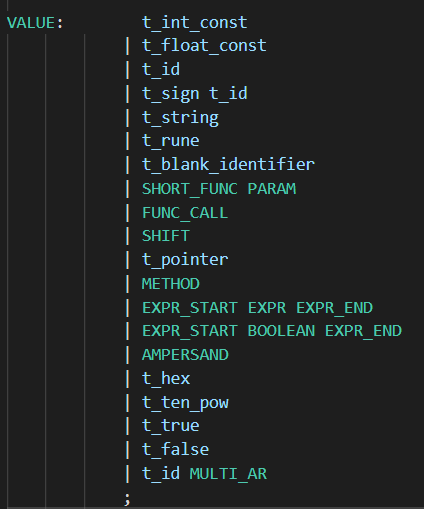


Рисунок 8 – объявление значений в грамматике.

Более подробное описание некоторых правил будет представлено в следующем пункте о решениях конфликтов, связанных с неоднозначностью правил, так как создание грамматики сопровождалось их решением.

* 1. **Анализ конфликтов**

В процессе написания грамматики многие правила были изменены частично или полностью несколько раз, чтобы обработка более сложных конструкций была корректной, а также чтобы избежать неоднозначности грамматики. Анализ конфликтов был произведен, используя вспомогательный файл y.output (флаг -y при запуске bison.exe).

Разберем подробнее появлявшиеся конфликты, их решения или попытки их решить.

* Return

Первые возникшие конфликты были вызваны после написания оператора return (1 shift/reduce и 6 reduce/reduce). Нетерминалы CALL и RET\_PARAM разворачивались в терминал t\_return. После этого анализатор не понимал, по какому именно правилу ему необходимо свернуться.

Остальные 6 конфликтов связаны с нетерминалами EXPR и VALUE. VALUE изначально состоял из объявления различных типов и правила, вызывающего EXPR:

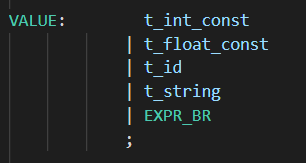


Рисунок 9 – описание нетерминала VALUE.

Конфликты возникали из-за того, что, если анализатор перейдет в состояние VALUE, то у него будет выбор сдвига на вышеперечисленные терминалы, но из состояния VALUE он также может попасть в EXPR, который тоже переходит на VALUE, попадая в те же варианты сдвига:

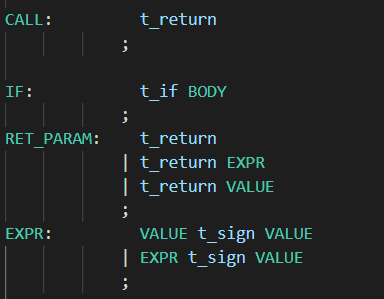


Рисунок 9 – некорректная реализация.

Для решения данных конфликтов в последствии был удален нетерминал CALL и более подходящим решением стало создание отдельного правила для оператора return, которому могут передаваться значения из VALUE.

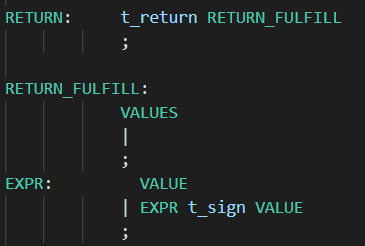


Рисунок 10 – измененная реализация грамматики.

* Массивы

На следующих этапах разработки грамматики конфликты появились при создании массивов.

Изначально их реализация была такова, что элементы (VALUE) массива, заключенные в квадратные скобки, рекурсивно могут вызываться в правиле MULTI\_AR, тем самым образуя одномерные или многомерные массивы. В теле массива также были объявлены оба случая:

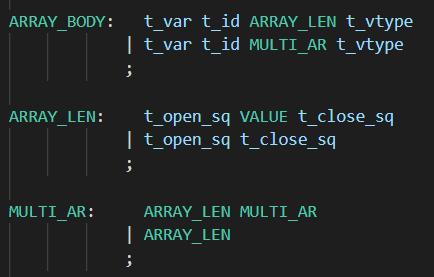


Рисунок 11 – некорректная грамматика.

Такая последовательность переходов вызывала 2 shift/reduce. Первый конфликт связан с правилом MULTI\_AR, так как возникает неоднозначность свертки: сделать сдвиг и войти в состояние t\_open\_sq в ARRAY\_LEN (открывающаяся скобка), или же перейти в нетерминал MULTI\_AR, который также начинается с терминала t\_open\_sq.

Второй shift/reduce возникает из-за схожей ситуации, когда в грамматике, описывающий нетерминал ARRAY\_BODY анализатор может свернуться по второму правилу, вызывающему в свою очередь ARRAY\_LEN.

В новом варианте грамматики MULTI\_AR также разворачивается в рекурсивный вызов элементов массива, но сами составляющие многомерного массива теперь выглядят проще, что исключает многочисленные переходы с конфликтами:

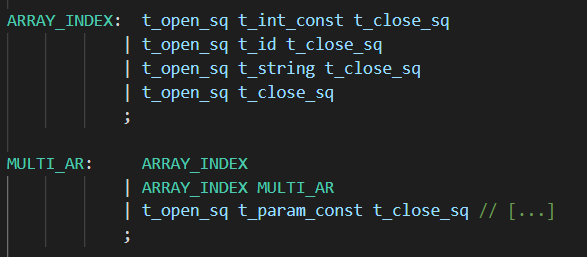


Рисунок 12 – решение конфликтов.

* Цикл For.

В языке Golang цикл for в классическом варианте имеет следующее определение, не отличающееся от языка С:

for [инициализация счетчика]; [условие]; [изменение счетчика]

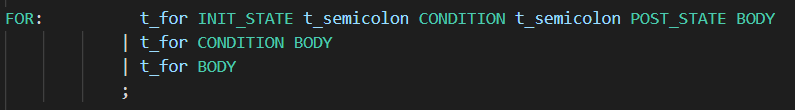
Однако пользователю необязательно указывать все условия при объявлении цикла. Можно вынести объявление переменной, убрать изменение счетчика, или оставить после for только условие. Так и было прописано в следующих правилах:

Рисунок 13 – изначальная реализация обработки цикла.

При такой реализации возникал 1 shift/reduce по причине того, что одно из правил CONDITION являлось пустым, поэтому очевидным решением было убрать третье правило, избежав неоднозначности перехода.

* Конфликты, оставшиеся нерешенными.

На момент завершения работы bison выделил 30 конфликтов shift/reduce. Решение большинства из них привело бы к некорректному анализу кода. Одним из ярких примеров, доказывающий это, является конфликт операции умножения и знака указателя.

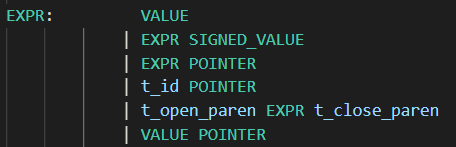


Рисунок 14 – правило, описывающее выражения.

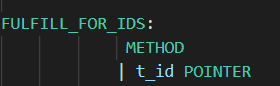


Рисунок 15 – описание вызова указателя.

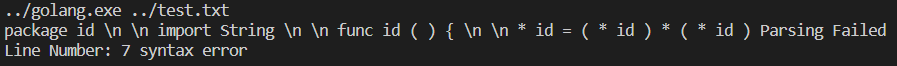
В данном случае анализатор может не понять, какой знак стоит во входном потоке и воспринимает их не так, как нужно. Тогда вывод программы будет таковой:

Рисунок 16 – некорректный анализ кода.

* Конфликты в сложных структурах

Так же при реализации сложных структур и больших структур, в которых последовательно записываются переменные и типы возникало огромное количество конфликтов. В правиле для аргументов функций (рисунок 17) в одной из первой версий правила, оно просто рекурсивно собирало само себя, это порождало 10 конфликтов shift/reduce и 1 конфликт reduce/reduce, так у правила не было какого-либо конечного терминала. Решением этой проблемы была разбиение правила на более мелкие компоненты такие как: Граничные правила (скобки) возможные значение (values), поглощение лишних переносов строк или символов ‘;’ (рисунок 18)

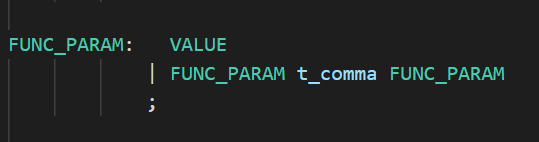


Рисунок 16 – рекурсивное поглощение аргументов

После переработки правила оно имело следующий вид:

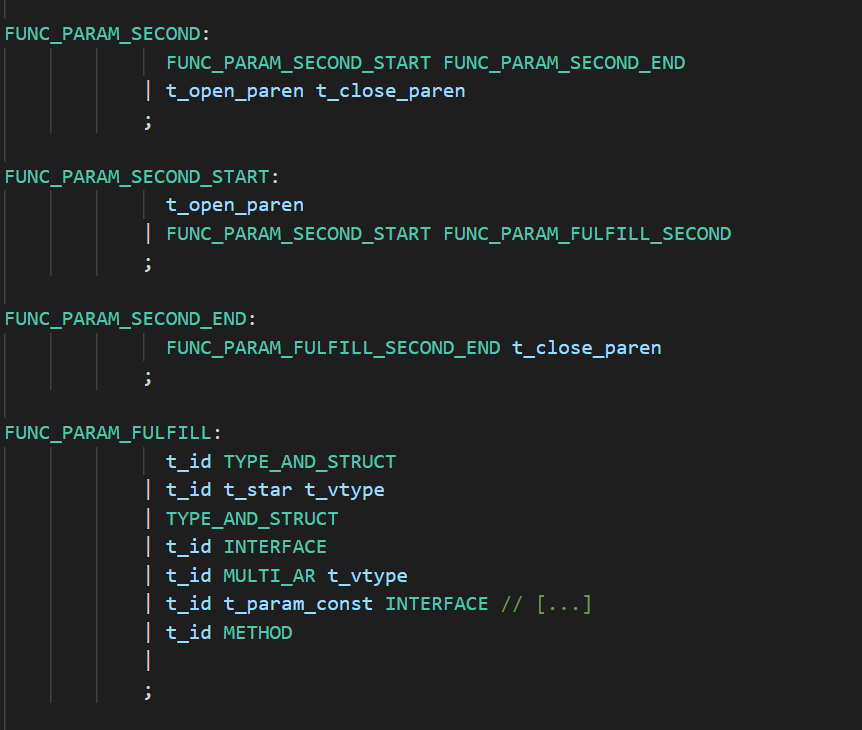


Рисунок 17 – переработанное поглощение аргументов

В новом правиле гораздо проще отслеживать какие значение обрабатываются тк они все вынесены в отдельное правило. Так же данное решение частично решает проблему переноса строк и не генерирует конфликты.

* 1. **Проблемы при разработке синтаксического анализатора**
* **Перенос на новую строку**

Одной из самых больших проблем, с которой пришлось столкнуться при выполнении курсовой работы стало разнообразие написание одного и того же участка кода. Допустим, мы хотим передать структуре начальные значения:

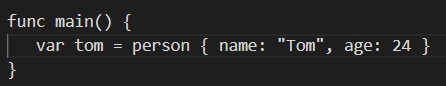


Рисунок 18 – пример кода.

Тоже самое мы можем сделать и в таком виде:

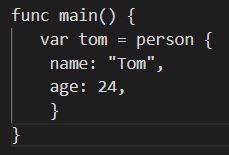


Рисунок 19 – пример кода.

В первоначальной реализации перенос строк после лексем вообще не рассматривался, и из-за этого был возможен только первый вариант написания. Для решения такой проблемы было решено ввести новый нетерминал, определяющий перенос строки, точку с запятой и конец файла:

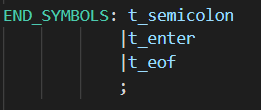


Рисунок 20 – реализация правил.

Теперь там, где это необходимо, нужно прописать такое правило, благодаря которому возможна вариативность написания.

* 1. **Примеры работы программы**



Рисунок 21 – успешный анализ вызовов методов.

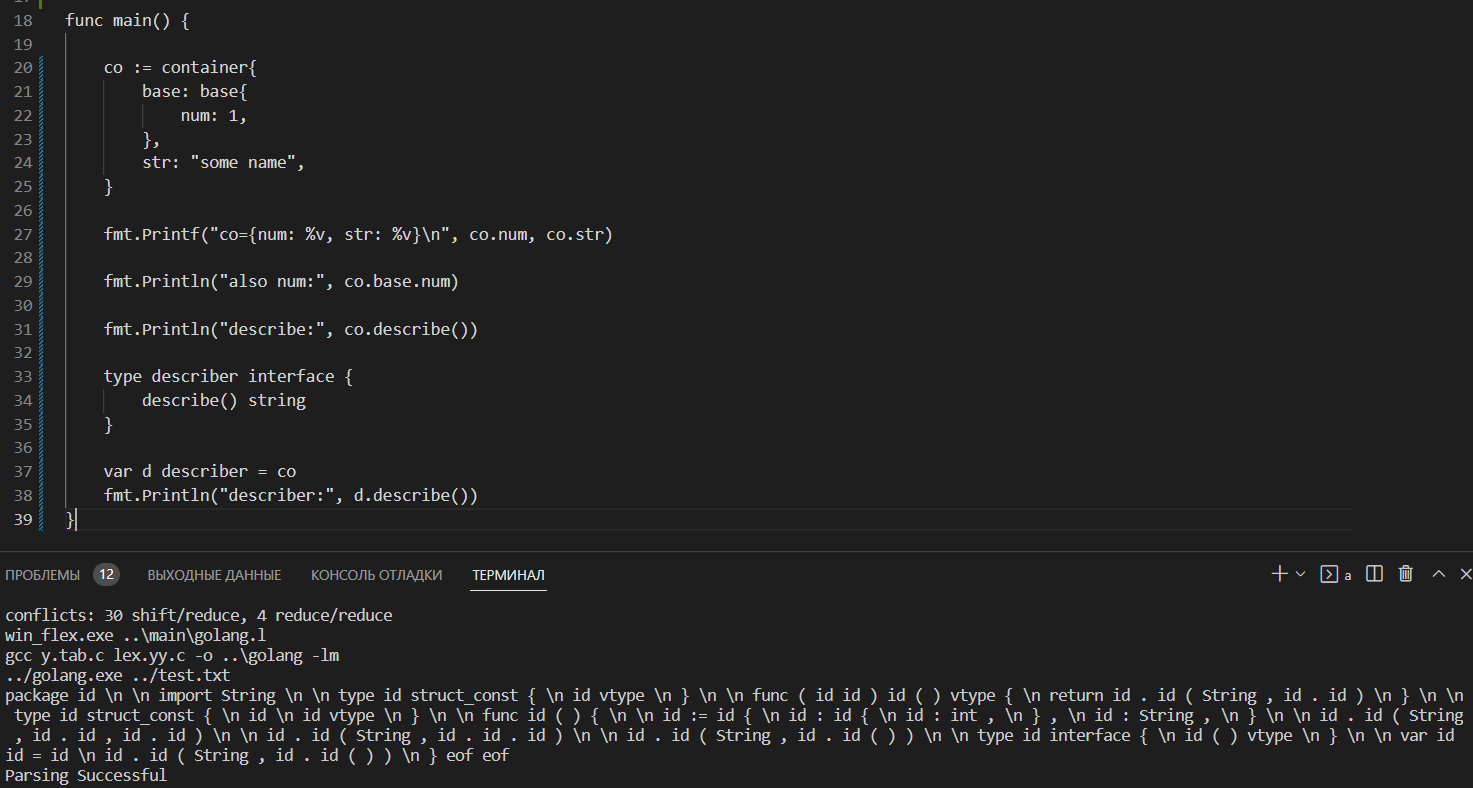


Рисунок 22 – анализ вложенных структур.

вывод

В результате выполнения курсовой работы были получены навыки разработки анализаторов с использованием bison/flex. Был разработан собственный анализатор языка Golang, состоящий из лексического и синтаксического анализаторов. Созданная грамматика языка рекурсивно собирает входные данные, благодаря чему анализатор может распознавать файлы с довольно большим объемом кода.

Хотя в процессе выполнения работы было описано много правил, созданный анализатор не является идеальным и способным распознать абсолютно все существующие элементы языка Golang. Также на момент завершения работы были решены не все конфликты, а это значит, что написанная грамматика не является однозначной.

При составлении грамматики большое значение придавалось работе с присваиваниями, так как на них строится большинство конструкций языка. Также ключевым моментом была обработка переносов на новую строку и поиска конца файла. Несмотря на то, что грамматика обрабатывает далеко не весь функционал Golang, при надобности можно добавить его и тем самым улучшить анализатор.

Таким образом, в конечной версии проекта существует 60 конфликтов shift/reduce и 6 конфликтов reduce/reduce, решение которых повлечет за собой некорректное распознавание кода. Это говорит о том, что создание полноценной однозначной грамматики данного языка практически невозможно или трудновыполнимо, и в целом такое возможно только для небольших и узконаправленных языков программирования.