Как автоматизированно обходиться с неоднозначностями грамматик

Артём Смородский, 2 группа

20.10.2022

1) Введение:

   КС-грамматика G неоднозначна, если существует цепочка $a \in L(G)$, имеющая два или более различных деревьев вывода. Если грамматика используется для определения языка программирования, желательно, чтобы она была однозначной. В противном случае программист и компилятор могут по разному понять смысл некоторых программ. Неоднозначность — нежелательное свойство КС-грамматик и языков.

2) Расссмотрим методы решения неоднозначностей:

а) Рассмотрим следующую грамматику:

   $S \to if\ b\ then\ S\ else\ S$

   $S \to if\ b\ then\ S$

   $S \to a$

   Эта грамматика неоднозначна так как если рассмотреть цепочку $if\ b\ then\ if\ b\ then\ a\ else\ a$, то ее можно прочитать двумя способами:

   $if\ b\ then\ (if\ b\ then\ a)\ else\ a$

   $if\ b\ then\ (if\ b\ then\ a\ else\ a)$.

   Здесь неоднозначность можно устранить, договорившись, что else должно ассоциироваться с последним из предшествующих ему then (как это принято в языках программирования).

   Также данную неоднозначность можно устранить, написав эквивалентную грамматику:

   $S_1 \to if\ b\ then\ S_1$

   $S_1 \to if\ b\ then\ S_2\ else\ S_1$

   $S_1 \to a$

   $S_2 \to if\ b\ then\ S_2\ else\ S_2$

   $S_2 \to a$

b) Рассмотрим грамматику:

$A \to AA$

$A \to a$

Она является неоднозначной так как подцепочка AAA содержит 2 разбора:

$A \Rightarrow AA \Rightarrow (AA)A$

$A \Rightarrow AA \Rightarrow A(AA)$

Здесь можно устранить неоднозначность, если вместо предложенных правил с двухсторонней рекурсией использовать одностороннюю, то есть использовать грамматику

$A \to AB$

$A \to B$

$B \to a$

c) Рассмотрим грамматику:

$A \to aA$

$A \to Ab$

Эта грамматика создает неоднозначность так как цепочка aAb создает 2 разных левых вывода:

$A \Rightarrow aA \Rightarrow aAb$

$A \Rightarrow Ab \Rightarrow aAb$

Все примеры выше были связаны с двухсторонней рекурсией и введение нового нетерминального символа, устранение рекурсии с одной из сторон позволяло убрать неоднозначность.

d) Рассмотрим грамматику:

$A \to aA$

$A \to aAbA$

Эта грамматика также создает неоднозначность так как цепочка aaAbA имеет 2 вывода:

$A \Rightarrow aAbA \Rightarrow aaAbA$

$A \Rightarrow aA \Rightarrow aaAbA$

Если при двухсторонней рекурсии средством борьбы с неоднозначностью является устранение рекурсии с одной из сторон, то в последнем случае поможет левая факторизация

e) Рассмотрим грамматику порождающую арифметическое выражение:

$E \to E + E$

$E \to E * E$

$E \to (E)$

$E \to a$

Эта грамматика создает неодназначность так как цепочка $E + E * E$ имеет 2 вывода:

$E \Rightarrow E + E \Rightarrow E + E * E$

$E \Rightarrow E * E \Rightarrow E + E * E$

Эту неоднозначность можно устранить используя дополнительный нетерминал:

$E \to E + T$

$E \to E * T$

$E \to T$

$T \to (E)$

$T \to a$

3) Причины неоднозначностей

Неоднозначность вызвана тем фактом, что грамматика может вывести одно и то же предложение по крайней мере двумя способами. Это не особенно интересная причина, так как она характеризует всю двусмысленность вообще. Мы заинтересованы в том, чтобы объяснить инженеру по грамматике, что не так с конкретной грамматикой и предложением и как, возможно, решить эту конкретную проблему. Мы заинтересованы в коренных причинах конкретных вхождений узлов выбора в синтаксических лесах. Например, давайте рассмотрим конкретную грамматику языка программирования C, для которой подпредложение $\{S \cdot b; \}$ неоднозначно. В одном из производных это блок одного оператора, который умножает переменные. C a также б , в другом это блок одиночного объявления указателя переменной b на что-то типа C. C точки зрения языкового инженера, причины этого двусмысленного предложения таковы:

a) * используется как в правиле, определяющем умножение, так и в правиле, определяющем типы указателей, а также

b) Имена типов и имена переменных имеют одинаковый лексический синтаксис, а также

c) Блоки кода начинаются с возможно пустого списка объявлений и заканчиваются, возможно, пустым списком операторов, а также

d) Как утверждения, так и декларации заканчиваются на ;

Соединение всех этих причин объясняет нам, почему существует двусмысленность. Удаление только одного из них исправляет это. На самом деле мы знаем, что для C неоднозначность была устранена путем введения средства устранения неоднозначности, которое резервирует любое объявленное имя типа из имен переменных с использованием таблицы символов во время синтаксического анализа, эффективно устраняя вторую причину.

4) Неоднозначности в YACC

Множество грамматических правил неоднозначно, если существует входная цепочка, которую можно структурировать несколькими различными способами. Например, правило

$expr : expr - expr;$

естественный способ выразить тот факт, что арифметическое выражение можно сформировать, взяв два других выражения и разделив их знаком минус. К сожалению, это правило не полностью определяет способ структуризации всей совокупности исходных дан- ных. Так, если входная цепочка - это

$expr - expr - expr$

правило позволяет представить ее как в виде

$(expr - expr) - expr$

так и в виде

$expr - (expr - expr)$

YACC выявляет подобные неоднозначности при построении алгоритма разбора. На примере входной цепочки

$expr - expr - expr$

рассмотрим проблему, с которой сталкивается алгоритм. После того, как алгоритм считывает второе вхождение expr, исходная цепочка вида

$expr - expr$

сопоставляется с правой частью правила. Алгоритм может выполнить свертку с применением данного правила, после чего цепочка преобразуется в expr (левая часть правила). Затем алгоритм считывает окончание входной цепочки

$-expr$

и снова выполняет свертку. Такая последовательность действий соответствует левой ассоциативности.

Если алгоритм прочитал

$expr - expr$

он, напротив, может отложить немедленное применение правила и продолжить чтение до тех пор, пока не будет считана цепочка

$expr - expr - expr$

Алгоритм может теперь применить правило к трем правым символам и свернуть их в expr, что даст в результате

$expr - expr$

Теперь можно еще раз выполнить свертку. Такая последовательность действий соответствует правой ассоциативности. Итак, прочитав

$expr - expr$

алгоритм может сделать один из двух шагов, каждый из которых законен: перенос или свертку. Нет оснований предпочесть один из них. Такая ситуация называется конфликтом перенос-свертка. Может произойти и так, что алгоритм будет иметь выбор между двумя допустимыми свертками. Такой конфликт называется конфликтом свертка-свертка. Отметим, что конфликтов перенос-перенос возникнуть не может.

YACC по умолчанию использует два метода разрешения неоднозначностей:

При конфликте перенос-свертка по умолчанию выполняется перенос.

При конфликте свертка-свертка по умолчанию выбирается правило, встретившееся в уасс-спецификации первым.

Первый метод устанавливает, что свертки менее предпочтительны, чем переносы, если есть выбор. Второй метод дает пользователю довольно грубый механизм управления разбором, поэтому, когда возможно, конфликтов свертка-свертка следует избегать.

5) Неоднозначности в Antlr

В естественном языке существуют неоднозначно трактуемые фразы. В формальных языках подобные конструкции также могут встречаться. Например, следующий фрагмент:

stat: expr ';' // expression statement

| ID '(' ')' ';' // function call statement;

;

expr: ID '(' ')'

| INT

;

Однако, в отличие от естественных языков, они скорее всего являются следствием неправильно разработанных грамматик. ANTLR не в состоянии обнаруживать такие неоднозначности в процессе генерации парсера, но может обнаруживать их непосредственно в процессе парсинга, если устанавливать опцию $LL\_EXACT\_AMBIG\_DETECTION$. Неоднозначность может возникать как в лексере, так и в парсере. В лексере для двух одинаковых лексем, формируется токен, объявленный выше в файле (пример с идентификаторами). Однако в языках, где неоднозначность действительно допустима (например, C++), можно использовать семантические предикаты (вставки кода) для ее разрешения, например:

expr: isfunc(ID) ? ID '(' expr ')' // func call with 1 arg

| istype(ID) ? ID '(' expr ')' // ctor-style type cast of expr

| INT

| void

;

Также иногда неоднозначность можно исправить с помощью небольшого переделывания грамматики. Например, в C существует оператор побитового сдвига вправо RIGHT_SHIFT: >>; две угловые скобки могут также использоваться для описания классов-генериков: $List < List < int >>$. Если определить токен >>, то конструкция из двух списков никогда не сможет распарситься, потому что парсер будет считать, что вместо двух закрывающихся скобок написан оператор >>. Чтобы решить такую проблему, достаточно просто отказаться от токена RIGHT_SHIFT. При этом токен LEFT_SHIFT: << можно оставить, поскольку такая последовательность символов при парсинге угловых скобок не может встретиться в валидном коде.

6) Ссылки:

http://cable.pu.ru/info/unix_c/p13.html04

https://habr.com/ru/company/pt/blog/210772/?ysclid=l9j3luk65j691438830

https://homepages.cwi.nl/ jurgenv/papers/SLE2011-2.pdf