Восстановление после ошибок

Вот на мой взгляд самые популярные стратегии, используемые синтаксическим анализом для восстановления после синтаксической ошибки:

- Parse Level Recovery
- Error Production
- Global Production
- Panic Mode Recovery

Parse Level Recovery (Восстановление уровня фразы)

Стратегия восстановления на уровне фраз используется большинством компиляторов исправления ошибок для восстановления после синтаксической ошибки. Основная идея этого метода состоит в том, чтобы выполнить локальную коррекцию тех входных строк, которые остались непроверенными при обнаружении ошибки. Мы можем сделать это исправление, подставив строку вместо входного префикса.

- Когда синтаксический анализатор находит ошибку, он пытается принять корректирующие меры, чтобы остальные входные данные оператора позволили синтаксическому анализатору продолжить синтаксический анализ.
- Одно неверное исправление приведет к бесконечному циклу.

Локальная коррекция:

- Замена префикса на некоторую строку.
- Замена запятой точкой с запятой.
- Удаление лишней точки с запятой.
- Вставьте отсутствующую точку с запятой.

Преимущество:

- Он может исправить любую входную строку

Недостаток:

- Трудно справиться с фактической ошибкой, если она произошла до момента обнаружения.

Пример:

int a,b

// *после восстановления*:

int a,b; // *точка с запятой добавляется компилятором*

Для LL(1) грамматики:

Восстановление на уровне фразы реализуется путем заполнения пустых записей в таблице предиктивного синтаксического анализа указателями на процедуры обработки ошибок. Каждая незаполненная запись в таблице синтаксического анализа заполняется указателем на специальную процедуру обработки ошибок, которая будет специально обрабатывать этот случай ошибки. Эти процедуры ошибок могут быть разных типов, например:

- изменять, вставлять или удалять символы ввода.
- выдавать соответствующие сообщения об ошибках.
- извлекать элементы из стека.

Error production (Производство ошибок)

Данный метод требует хорошего знания распространенных ошибок, с которыми можно столкнуться во время синтаксического анализа. Мы можем дополнить грамматику соответствующего языка продуктами ошибок, которые генерируют ошибочные конструкции. Если во время синтаксического анализа используется производство ошибок, мы можем генерировать соответствующее сообщение об ошибке, чтобы указать ошибку, которая была распознана во входных данных. Например разработчикам компиляторов известны некоторые распространенные ошибки, которые могут возникать в коде. Дизайнеры же могут создавать расширенную грамматику для использования в качестве продуктов, которые генерируют ошибочные конструкции при обнаружении этих ошибок.

Этот метод чрезвычайно сложен в обслуживании и одним из основных недостатков этого метода является то, что он сталкивается с трудностями в соблюдении грамматики. Это связано с тем, что при изменении грамматики необходимо изменить соответствующее производство ошибок при синтаксическом анализе. Но есть и преимущество у данного метода - синтаксические фазовые ошибки обычно восстанавливаются путем производства ошибок.

Пример:

Пусть входная строка - *abcd$*

Grammar: S-> A

A-> aA | bA | a | b

B-> cd

Входная строка недоступна с помощью приведенной выше грамматики, поэтому нам нужно добавить дополнить нашу грамматику.

Grammar: E->SB // ДОПОЛНЕНИЕ ГРАММАТИКИ

S-> A

A-> aA| bA | a | b

B-> cd

Теперь можно получить строку *abcd*.

Global Production (Глобальная коррекция)

Нам часто нужен такой компилятор, который вносит очень мало изменений при обработке неправильной входной строки в правильную входную строку. Учитывая неправильную входную строку x и грамматику G **,** сам алгоритм может найти дерево синтаксического анализа для связанной строки y (ожидаемая выходная строка); таким образом, чтобы количество вставок, удалений и изменений токена, необходимых для преобразования x в y, было как можно меньше. Методы глобальной коррекции увеличивают требования к времени и пространству во время синтаксического анализа, но из-за сложности (времени и пространства) эта стратегия еще не реализована на практике.

Преимущества:

- он вносит очень мало изменений при обработке неправильной входной строки.

Недостатки:

- это просто теоретическая концепция, которую невозможно реализовать, потому что стратегия требует слишком больших затрат времени и места для реализации.

Panic Mode Recovery (восстановление в режиме паники)

Это простая стратегия, используемая большинством методов синтаксического анализа для восстановления после синтаксической ошибки. Идея состоит в том, чтобы после обнаружения единицы ошибки, отбрасывать входные символы один за другим, пока не будет найден набор синхронизирующих токенов, таких как точка с запятой или конец. Следовательно, эта стратегия в основном фокусируется на значительном количестве пропусков ввода, проверяя отсутствие дополнительных ошибок ввода.

Преимущества :

- Это самый простой метод, и его очень легко реализовать.
- Он никогда не входит в бесконечный цикл.

Недостатки :

- Огромное количество входных данных отбрасывается без их обработки, и он больше не обнаруживает ошибок, поскольку не обрабатывает их дальше.

Режим паники для LL(1) грамматики:

В этом методе восстановления мы используем символы FOLLOW в качестве маркеров синхронизации, а «синхронизация» в таблице предиктивного синтаксического анализа указывает на маркеры синхронизации, полученные из наборов FOLLOW нетерминала.

Что такое синхронизирующий токен?

Терминальные символы, лежащие в последующем наборе нетерминалов, могут использоваться в качестве синхронизирующего набора токенов для этого нетерминала. В простом аварийном режиме восстановления после синтаксического анализа LL(1).

Все пустые записи помечаются как синхронизированные, чтобы указать, что синтаксический анализатор будет пропускать все входные символы до символа в следующем наборе из трех нетерминалов A, которые находятся на вершине стека. Затем нетерминал A будет извлечен из стека синтаксическим анализатором. Анализ продолжается из этого состояния. Для обработки не сопоставленных терминальных символов синтаксический анализатор вставляет несовпадающий терминальный символ из стека, после чего генерирует сообщение об ошибке, изображающее несовпадающий терминал.

Пример:

Первая входная строка - ***aab$*** (первая картинка)

Вторая входная строка - ***ceadb$*** (вторая картинка)

$S \to AbS \mid E \mid \varepsilon$
$A \to a \mid cAd$
Follow(S) = {d, $}
Follow(A) = {b, d}

	a	b	c	d	e	$
S	Abs	sync	Abs	sync	e	ε
A	a	sync	cAd	sync	sync	sync

stack	input	output
$S	aab$	S → AbS
$SbA	aab$	A → a
$Sba	aab$	
$Sb	ab$	Error: missing b, inserted d
$S	ab$	S → AbS
$SbA	ab$	A → a
$Sba	ab$	
$Sb	b$	
$S	$	S → ε
$	$	accept

stack	input	output
$S	ceadb$	S → AbS
$SbA	ceadb$	A → cAd
$SbdAc	ceadb$	
$SbdA	eadb$	Error: unexpected (illegal A)
e		

(Удалим все токены до первого b или d, затем удалим A)

$Sbd	db$	
$Sb	b$	
$S	$	S → ε
$	$	accept.

Realization Panic mode recovery

Восстановление в режиме паники реализовано в функции p_error(). Например, эта функция начинает отбрасывать токены до тех пор, пока не достигнет закрывающего '}'. Затем он перезапускает анализатор в его исходном состоянии.

```
def p_error(p):
    if not p:
        print("End of file!")
        return

    #Читайте дальше в поисках заключения '}'
    while True:
        tok = parser.token()            # Получите следующий токен
        if not tok or tok.type == 'RBRACE':
            break
    parser.restart()
```

Эта функция просто отбрасывает неверный токен и сообщает анализатору, что ошибка была исправлена.

```
def p_error(p):
    if p:
        print("Syntax error at token", p.type)
        # Просто отбросьте токен и скажите анализатору, что все в порядке.
        parser.errok()
    else:
        print("Syntax error at EOF")
```

Более подробная информация об этих методах:

- **parser.errok()** - *сбрасывает состояние анализатора, чтобы он не думал, что находится в режиме восстановления после ошибок. Это предотвратит генерирование маркера ошибки и сбросит внутренние счетчики ошибок, так что следующая синтаксическая ошибка снова вызовет p_error().*
- **parser.token()** - *возвращает следующий токен во входном потоке.*
- **parser.restart()** - *отбрасывает весь стек синтаксического анализа и возвращает анализатор в исходное состояние.*

Чтобы предоставить синтаксическому анализатору следующий токен, p_error() может вернуть токен. Это может быть полезно при попытке синхронизации по специальным символам. Например:

```
def p_error(p):
    # Читайте дальше в поисках завершающего ";"
    while True:
       tok = parser.token()          # Получите следующий токен
       if not tok or tok.type == 'SEMI': break
    parser.errok()

    # Верните SEMI синтаксическому анализатору в качестве следующего контрольного токена
    return tok
```

Заключение:

Суммируя все вышесказанное, можно сделать вывод, что каждый метод идеален для своей конкретной цели, если мы хотим максимально точную обработку ошибок, при это ограничения на время обработки у нас нету, то тогда следует использовать **Error Production** или **Parse Level Recovery,** если нужно справиться с восстановление после ошибок, при этом не сильно углубляясь в мелочи синтаксического анализа конкретной грамматики, то мы используем метод **Panic Mode Recovery.** Для себя я выбрал второй вариант, потому есть много источников, дающих всю необходимую информацию для изучения данной стратегии.