Санкт-Петербургский государственный университет

Кафедра системного программирования
Математическое обеспечение и администрирование
информационных систем

Андреев Илья Алексеевич

Восстановление синтаксического анализа RuC после обнаружения ошибок

Курсовая работа

Научный руководитель:
д. ф.-м. н. проф. А. Н. Терехов

Санкт-Петербург
2020

Оглавление

Введение

В наше время активно развиваются многие языки программирования. Кроме высокоуровневых языков, таких как Python или Swift, продолжают свое развитие и низкоуровневые языки. К таким относится язык C, который стал стандартом для программирования систем управления, контроля и мониторинга благодаря своей простоте и эффективности [2].

На кафедре системного программирования профессором А. Н. Тереховым ведется разработка языка RuC, который продолжает идеи языка C. RuC преследует цель облегчить изучение языков программирования и наряду с этим создать среду для разработки высоконадёжнего программного обеспечения [10].

Синтаксический анализ – это первый этап компиляции, результатом которого является синтаксическое дерево разбора. В процессе сопоставления токенов языка с его формальной грамматикой операторы исходного кода сопоставляются внутренним узлам дерева, а операнды - листьям. В предыдущей реализации синтаксического анализатора RuC, если встречалась ошибка в исходном коде, то анализатор заканчивал работу. Данный подход был не всегда удобен пользователям, так как ошибок в тексте могло быть много, а пользователь узнавал о каждой ошибке лишь после того, как была исправлена предыдущая, что сильно замедляло фазу исправления ошибок.

Задачей данной курсовой работы является написание такого синтаксического анализатора, который может продолжать анализ после нахождения ошибки, обнаружив как можно больше действительных ошибок и как можно меньше наведенных.

1. Цели и задачи

1.1. Цель работы

Реализация синтаксического анализатора RuC, который сможет продолжать анализ после нахождения ошибки, обнаружив при этом как можно больше действительных ошибок и как можно меньше наведенных.

1.2. Поставленные задачи

Для выполнения обозначенной цели необходимо выполнить следующие задачи:

- изучить аналогичные решения;
- для каждой ошибки, распознаваемой компилятором RuC, написать алгоритм восстановления;
- подготовить тесты и протестировать анализатор;
- сравнить с аналогами.

2. Обзор существующих решений

Существует несколько стратегий восстановления анализа после ошибки, но ни одна из них не является универсальной. По этой теме было проведено значительное количество исследований, однако в этом разделе будут описаны только те варианты, которые чаще остальных используются на практике.

Недостаточно проигнорировать токен, который не соответствует формальной грамматике или приводит к семантической ошибке. Внутреннее состояние анализатора должно быть модифицировано таким образом, чтобы он смог проанализировать оставшуюся часть ввода. Эта адаптация и называется восстановлением после ошибок.

Кроме того, изменение состояния анализатора может привести к семантическим ошибкам. Простейшим решением этой проблемы является игнорирование любых семантических ошибок после обнаружения синтаксической. Но такой метод не подходит в данной работе, так как перед данным анализатором стоит задача обнаружить как можно больше ошибок.

2.1. Продукции ошибок

Продукции ошибок – правила грамматики, добавленные для того, чтобы ожидаемые синтаксические ошибки стали частью языка. Эти продукции обычно имеют связанное с ними действие, сообщающее об ошибке, которое запускается, когда используется продукция ошибки.

Примером использования этого метода может служить if-

оператор в языке Pascal. Его синтаксис выглядит следующим образом:

```
if-statement -> IF boolean-expression
                THEN statement else-part
   else-part -> ELSE statement | ε
```

то есть в случае, если if-оператор имеет часть else, то ';' будет ошибкой. Эту ситуацию можно обнаружить, если приведенное правило грамматики для части else будет заменено на следующее:

```
else-part -> ELSE statement | ε | ; ELSE statement
```

Наиболее важные недостатки этого метода:
• Анализатор обрабатывает только те ошибки, которые явно были добавлены в формальную грамматику языка;
• Правила, введенные для обработки ошибок, могут вносить конфликты в формальную грамматику языка, что может привести к ошибкам самого анализатора.
Подробнее о роли продукций ошибок написано в статье [3].

2.2. Глобальная коррекция

Существуют методы, которые не только изменяют состояние анализатора, но и исправляют исходный код. При этом перед таким анализатором стоит задача внести минимально возможное количество исправлений. По входной строке алгоритм должен найти такое дерево разбора, чтобы число вставок, удалений и замен токенов входного файла было минимальным. В

общем случае такие методы очень требовательны к памяти и времени работы анализатора, а потому не так часто применяются в компиляторах. К таким методам, например, относится метод исправления ошибок только вставками, описанный в статье [1], и метод исправления наименьшего количества ошибок, описанный в статье [5].

2.3. Восстановление на уровне фразы

При обнаружении ошибки анализатор может выполнить коррекцию на уровне фразы, то есть заменить начало оставшегося исходного кода на такую строку, которая позволит продолжить анализ. Например, такой метод может заменить `','` на `';'` или подставить недостающую `';'`. Недостатком этого метода являются сложности, которые могут возникнуть, если ошибка в исходном коде расположена до ее места обнаружения. Примером такого метода может служить метод восстановления ошибок при движении назад/вперед, описанный в статье [4].

2.4. Панический режим

Панический режим, вероятно, является самым простым, но все же в некоторой степени эффективным методом исправления ошибок. В этом методе приемлемый набор определяется составителем синтаксического анализатора и фиксируется для всего процесса анализа. Символы в этом наборе обычно указывают конец синтаксической конструкции, например, оператора языка программирования. Для языка C этот набор может содержать

символы `';'`, `')'`, `']'` и `'}'`. При обнаружении ошибки символы пропускаются до тех пор, пока не будет найден символ, который является членом этого набора. Затем синтаксический анализатор должен быть приведен в состояние, при котором этот символ становится приемлемым. Возможности восстановления в режиме паники часто довольно хороши, но многие ошибки могут остаться незамеченными, потому что иногда большие части ввода пропускаются.

Именно этот метод был выбран для реализации в рамках этой работы, так как он не обладает перечисленными выше недостатками, хотя это и может привести к пропущенным ошибкам.

3. Реализация

Синтаксический анализатор в RuC устроен так, что в каждый момент времени читаются два подряд идущих токена: currtoken и nexttoken. Так сделано, чтобы можно было видеть дальше текущего токена, но при этом не нужно было делать возвраты для того, чтобы узнать, началом какой конструкции это является. Например, если анализатор ожидает оператор, а в currtoken находится идентификатор, то анализатор смотрит на значение nexttoken: если в нем `':'`, то это начало labeled statement, и нужно разбирать объявление метки и следующий оператор, иначе это начало expression statement. Подробнее механизм работы анализатора был описан в статье [9].

Этот механизм помогает и в случае анализатора, умеющего восстанавливаться после ошибок. Если ожидается один токен, а встречен другой, то и в этом случае анализатору не нужно делать возврат по исходному тексту. Вместо этого анализатор просто сравнивает содержимое nexttoken с ожидаемым токеном, и в случае несовпадения анализатор выдает соответствующую ошибку и пользуется алгоритмом восстановления для этой ошибки.

В этом разделе будут описаны инструменты, используемые для восстановления после ошибок, и случаи, когда они применяются. В остальных случаях особых действий производить не нужно. Например, в случае обнаружения ошибок видозависимого анализа (проверки совместимости типов) анализатор сигнализирует о них, но продолжает работу так, как будто их и не было.

3.1. Функция mustbe(Token, errorType)

Функция `mustbe(Token, errorType)` сравнивает следующий токен в читаемом коде и первый переданный аргумент. Если они равны, то считывает токен из кода, иначе выдает ошибку, код которой передан вторым аргументом.

Данная функция используется в тех случаях, когда из-за несовпадения ожидаемого и читаемого токенов не нужно предпринимать особых действий. Например, после большинства операторов по грамматике языка C должен быть токен `';'`. Если его нет, то с помощью `mustbe` анализатор это обнаружит и выдаст ошибку. Затем без дополнительных действий анализатор может начинать разбор следующего оператора.

3.2. Функция skipUntil(Token, unsigned flags)

Функция `skipUntil` - основной инструмент в восстановлении после ошибок. Эта функция пропускает токены из исходного файла, пока не встретит токен, переданный первым аргументом. Кроме того, эта функция принимает вторым аргументом набор флагов, который позволяет управлять её работой. Один из этих флагов говорит, нужно ли останавливаться перед искомым токеном, или необходимо сделать еще один шаг вперед (то есть остановиться, когда искомый токен будет совпадать со значением в currtoken или в nexttoken), другие флаги говорят, нужно ли останавливаться перед `';'`, `')'` и подобными символами. Кроме того, функция умеет пропускать токены, равные искомому, если они были вложены в другие конструкции, например, в скобки или в тернарный оператор. Та-

ким образом, если во время работы `skipUntil` встретилась `'('`, а первым аргументом был передан другой токен, то происходит вызов `skipUntil(')', NoFlags);` и при этом игнорируются все остальные токены, даже если встретится искомый. Если был встречен токен `'?'`, то будут пропущены все токены до `':'`. При этом вызов функции рекурсивный, то есть вложенные конструкции в скобках или вложенные тернарные операторы тоже будут пропущены.

Рассмотрим конструкции, в которых используется эта функция.

3.2.1. Ошибки в выражениях

При разборе выражений функция `skipUntil` используется в двух случаях: при разборе первичного и постфиксного выражений.

В случае первичного выражения анализатор ожидает увидеть один из пяти токенов: числовую константу, символьную константу, строковый литерал, открывающую круглую скобку или идентификатор. В остальных случаях токен не является началом первичного выражения, и анализатор пропускает токены исходного файла до `';'`, если не указано иначе.

В случае постфиксного выражения по грамматике RuC ожидается, что оно применяется к идентификатору. Если это не так, то анализатор выполняет действия, аналогичные описанным выше.

3.2.2. Ошибки в списках объявлений

Списки объявлений в RuC записываются следующим образом: сначала идет описание типа (стандартного или описанного пользователем) с возможностью указать хранение переменной на регистре, затем через запятую перечисляются либо идентификаторы, либо конструкция `'*'` + идентификатор, и заканчивается это все токеном `';'`. В случае если будет обнаружена ошибка в каком-нибудь объявлении (например, будет отсутствовать идентификатор, или сразу после него не будет разделяющего токена `','`), анализатор сделает вызов `skipUntil(',', StopAtSemi);` который пропустит все токены до `','` (или до токена `';'`, если он встретился раньше). Это сделано для того, чтобы анализатор смог разобрать как можно больше объявлений идентификаторов, и тогда при их дальнейшем использовании не будет ошибок "Идентификатор не объявлен".

3.2.3. Ошибки в операторах

Если в операторах встречается ошибка, то анализатор пропускает токены до `';'`, так как в языке С он является признаком окончания оператора. Например, если в операторе `do-while` после тела цикла не был встречен токен `while`, то для того чтобы не появилось наведенных ошибок, лучше пропустить все токены до конца этого оператора. Другой пример - оператор `for`. По синтаксису С в его условии ожидаются три конструкции: объявление или оператор-выражение, оператор-выражение и выражение. Если встретилась ошибка в первой части условия, то вызывается функция `skipUntil`, которая пропустит токены до

`';'`, который обозначает окончание первой части условия. При этом на случай, если и `';'` там не будет, в вызов передается флаг `StopAtRParen`, который не даст преждевременно выйти анализатору из условия цикла `for`. Аналогично происходит и при разборе второй части условия. В случае ошибки при разборе третьей части условия необходимо пропустить токены до `')'`, но на самом деле происходит такой же вызов, так как завершающего условие токена `')'` может и не быть.

3.2.4. Ошибки в списках инициализации

Списки инициализации в C выглядит как заключенный в фигурные скобки список выражений, разделенных `','`. В RuC запрещено присваивать такое значение переменным простых типов, этот список участвует при присваивании значений структурам и массивам.

В общем случае, если встречается ошибка в списке инициализации (например, не хватает `','` после него, анализатор пропускает токены до `','`, при этом функция `skipUntil` принимает флаги `StopAtRBrace` и `StopAtSemi`, которые диктуют ей остановиться перед токенами `'}'` и `';'` соответственно. Первый служит признаком окончания списка инициализации, второй - окончания конструкции, в которой этот список использовался. Это мог быть список описаний или оператор с выражением в своем составе. Флаг `StopAtSemi` необходим на тот случай, если в конце списка инициализации не хватает `'}'`.

В случае если список инициализации используется при инициализации структуры, то добавляется еще и проверка его дли-

ны. В RuC, в отличие от C, длина такого списка строго должна соответствовать количеству полей в инициализируемой структуре. В случае если анализатор считал необходимое количество выражений, а закрывающая список инициализации `'}'` не была встречена, то во избежания появления наведенных ошибок происходит пропуск токенов до `'}'` или до `';'`, завершающим список описаний.

3.2.5. Ошибки в списках аргументов

Списки аргументов в языке C встречаются в качестве оператора вызова или при объявлении функции. При этом единственная разница между этими конструкциями - при объявлении функций для каждого аргумента нужно явно указать типы аргументов.

В случае если это вызов функции, анализатор производит действия, аналогичные предыдущему пункту, только за тем исключением, что используется флаг `StopAtRParen`, так как в данном случае роль завершающего конструкцию токена будет играть `')'`.

Для объявлений функции в RuC есть отличные от C условия - если это предописание функции, то для аргументов запрещено использовать имена, нужно просто перечислить через запятые их типы, а если это описание, то имена аргументов необходимы. Поэтому в случае если для некоторых аргументов были встречены имена, а для некоторых - нет, анализатор пропускает токены до завершающего объявление токена `')'`, а затем проверяет, есть ли тело у этой функции. Если после токена `')'`

идет токен `'{'`, то тело функции - составной оператор, и необходимо пропустить токены до `'}'`, иначе - пропустить до токена `';'` (в независимости от того, обозначает ли он окончание предописания функции или окончание тела функции, состоящего из одного оператора)

3.2.6. Ошибки в объявлении массива

Объявление массива в языке C отличается от остальных объявлений тем, что после идентификатора, который служит именем массива, объявляются размеры массива в квадратных скобках. Соответственно, в данном случае, если разборе выражения, отвечающего за размер, встречена ошибка, пропускать нужно до `']'`. При этом, опять же, в функцию `skipUntil` передаётся флаг `StopAtSemi`, задачей которого является не дать пропустить токены анализатору за пределами списка объявлений, в котором и встретилось объявление массива.

3.3. Ошибки типов

В языке C присутствуют такие конструкции, для которых явно указано, какого типа должны быть результаты выражений. Например, индекс массива должен быть целочисленным, нельзя складывать числа и структуры. Кроме того, в RuC, в отличие от языка C, всегда запрещено неявное преобразование из типа с плавающей точкой в целочисленный или символьный тип. Например, запрещено округление чисел с плавающей точкой при передаче параметром вместо целого числа, при присваивании значения переменной целочисленного типа, тип усло-

вия в операторе `switch` также должно быть целочисленным или символьным и так далее. В таких случаях анализатор RuC выдает ошибку.

Однако из-за того что теперь анализатор продолжает анализ после ошибок, может произойти такое, что в объявлении некоторого идентификатора была ошибка, и это может повлечь новые ошибки видозависимого анализа. Для сокращения числа наведенных ошибок был введен тип `Undefined`. Запись об идентификаторе все равно будет занесена в таблицу, но ее типом будет `Undefined`. Если далее встретится использование этого идентификатора в таком контексте, где нужно будет следить за типами, анализатор не будет выдавать ошибку о несоответствии типов. Например, поле `b` у структуры `point` было объявлено следующим образом:

```
struct point { int a; typespec b; } pt;
```

но при это идентификатор `typespec` не был объявлен ранее. В таком случае запись об идентификаторе `point` будет занесена в таблицу типов таким образом, что у второго типа будет тип `Undefined`. Тогда в коде

```
int t[5];
int t2 = t[pt.b];
```

не появится ошибок ни о том, что идентификатор `b` не был описан, ни о том, что его тип выборки `pt.b` не подходит в эту конструкцию.

4. Апробация

Для подтверждения работоспособности анализатора, разработанного в ходе данной работы, была применена уже существующая в проекте RuC автоматическая система тестирования Travis-CI [7]. Для большинства ошибок, распознаваемых компилятором, уже были созданы тесты, использовавшиеся для того, чтобы проверять, обнаруживает ли анализатор ошибки. В ходе работы тестовая база была дополнена, так чтобы для каждой ошибки был свой тест. С её помощью система тестирования проверяет, что анализатор правильно продолжает разбор и не сигнализирует о наведённых ошибках.

Кроме того, были написаны содержащие несколько ошибок тесты, такие что в каждой инструкции языка находится не более одной ошибки. Они применяются для того, чтобы проверить, что анализатор может находить последующие ошибки. Благодаря ограничению на распределение ошибок по тесту можно заранее указать, обнаружение каких ошибок ожидается, а это в свою очередь позволяет встроить тесты в автоматическую систему тестирования.

В общем случае задача, поставленная перед анализатором – обнаружить как можно больше действительных и как можно меньше наведённых ошибок – не формализуема. Из-за этого тесты, на которые не наложены описанные ранее ограничения, необходимо прогонять вручную и оценивать, насколько приемлемый результат выдаёт анализатор.

Далее будут приведены несколько примеров кодов с ошибками и ошибки, о которых сигнализирует анализатор RuC, ко-

торый был реализован в рамках данной работы. Для сравнения сигнализации об ошибках был выбран компилятор Clang [8], являющийся одним из самых популярных компиляторов языка C. В своей работе при обнаружении ошибки Clang также использует метод восстановления в режиме паники [6]. При проведении тестирования использовался Clang версии 12.0.0.

4.1. Пример 1

В данном примере используются неописанные ранее идентификаторы `typespec1` и `typespec2`, а в теле функции используются аргументы `a` и `b` в контекстах, где важен тип выражения. Кроме того, после одного из списков описаний не хватает `';'`, и функция не возвращает значения, хотя была объявлена с типом значения `int`.

```
int func(typespec1 a, typespec2 b) {
    int t1[5];
    int t2 = t1[b] + a
    int t3[b];
}
```

Вывод анализатора RuC:

```
Использование неописанного идентификатора 'typespec1'
Использование неописанного идентификатора 'typespec2'
Ожидалась ';' после списка описаний
Функция, имеющая непустой тип, не возвращает значения
```

Вывод анализатора Clang:

```
test.c:1:10: error: unknown type name 'typespec1'
test.c:1:23: error: unknown type name 'typespec2'
```

4.2. Пример 2

В данном примере используются неописанные ранее идентификаторы `a` и `b`, после `if` не хватает `'('` и постфиксное выражение применяется не к идентификатору.

```
int main() {
    if a == 0)
        b += 5(a);
    return 0;
}
```

Вывод анализатора RuC:

```
Ожидалась левая скобка перед условием
Использование неописанного идентификатора 'a'
Использование неописанного идентификатора 'b'
Постфиксное выражение должно применяться к идентификатору
```

Вывод анализатора Clang:

```
test.c:3:5: error: expected '(' after 'if'
```

4.3. Вывод

Тестирование, в частности приведённые выше примеры, показали, что разработанный в ходе данной работы анализатор в ряде тестов обнаруживает больше ошибок, чем компилятор Clang, а в прочих случаях справляется с задачей так же.

Заключение

В рамках данной работы были выполнены следующие задачи:

• изучены аналогичные решения;

• для каждой ошибки, распознаваемой компилятором RuC, написан алгоритм восстановления;

• подготовлены тесты и анализатор протестирован;

• произведено сравнение с аналогами.

Список литературы

[1] Anderson S.O., Backhouse R.C. An Alternative Implementation of an Insertion-Only Recovery Technique. — Department of Computer Science, Heriot-Watt University, Edinburgh EHI 2HJ, Scotland, 1982.

[2] Barr Michael, Massa Anthony. Programming Embedded Systems, 2nd Edition. — O'Reilly Media, Inc., 2006.

[3] C.N.Fischer, J.Mauney. On the role of error productions in syntactic error correction. — University of Wisconsin-Madison, Madison, WI 53706, U.S.A., 1979.

[4] Graham Susan L., Rhodes Steven P. Practical syntactic error recovery. — Communications of the ACM, 1975.

[5] Lyon Gordon. Syntax-directed least-errors analysis for context-free languages: a practical approach. — Communications of the ACM, 1974.

[6] Исходный код компилятора Clang // `https://clang.llvm.org/doxygen/Parse_2Parser_8h_source.html#l01178`. — (дата обращения: 10.12.2020).

[7] Официальный сайт Travis-CI // `https://travis-ci.com`. — (дата обращения: 10.12.2020).

[8] Официальный сайт компилятора Clang // `https://clang.llvm.org`. — (дата обращения: 10.12.2020).

[9] Терехов А.Н. Инструментальное средство обучения программированию и технике трансляции. — Компьютерные инструменты в образовании, 2016.

[10] Терехов А.Н., Терехов М.А. Проект РуСи для обучения и создания высоконадежных программных систем. — Известия высших учебных заведений. Северо-Кавказский регион. Технические науки, 2017.