ISSN 1814-1196
Научный вестник НГТУ
том 75, № 2, 2019, с. 37–48

http://journals.nstu.ru/vestnik
Science Bulletin of the NSTU
Vol. 75, No. 2, 2019, pp. 37–48

ИНФОРМАТИКА,
ВЫЧИСЛИТЕЛЬНАЯ ТЕХНИКА
И УПРАВЛЕНИЕ

INFORMATICS,
COMPPUTER ENGINEERING
AND CONTROL

УДК 004.43

DOI: 10.17212/1814-1196-2019-2-37-48

Обработка ошибок в синтаксическом анализаторе компилятора языка El*

А.А. МАЛЯВКО

630073, РФ, г. Новосибирск, пр. Карла Маркса, 20, Новосибирский государственный технический университет

a.malyavko@corp.nstu.ru

Рассматривается задача обработки синтаксических ошибок в компиляторе функционально-императивного языка программирования El, выполняемой с целью организации продолжения синтаксического разбора после останова для нахождения максимально возможного количества действительно допущенных ошибок. Предлагается комбинация нескольких известных способов нейтрализации синтаксических ошибок, ориентированная на особенности используемого в трансляторе алгоритма детерминированного синтаксического разбора. Парсер компилятора реализован в виде стекового автомата, управляемого входным токеном и символом, снимаемым с верхушки стека. Предлагаемый метод нейтрализации ошибок основан на полном переборе всех возможных вариантов модификации одного ошибочного токена: поочередная вставка, замена и удаление для выбора такого варианта перезапуска автомата, при котором следующая синтаксическая ошибка обнаруживается на максимальном расстоянии от нейтрализуемой. Если никакая модификация одиночного ошибочного токена не приводит к успешной нейтрализации, то в предлагаемом методе после каждого удаления расширяется множество допустимых токенов, вычисляемое как объединение множеств выбора нескольких символов, находящихся на верхушке стека автомата. Этот переход от обработки одиночной ошибки к обработке множественной позволяет уменьшить количество входных токенов, считываемых парсером без анализа, по сравнению с известным режимом паники. Описываемый алгоритм нейтрализации сочетается в El-компиляторе с «правилами для типичных ошибок», включаемыми в грамматику для таких возможных ситуаций, когда для успешной нейтрализации ошибки требуется вставить не один, а два или три токена. Подобные ошибки возможны в программах на языке El, их эффективная нейтрализация другими способами представляется весьма затруднительной.

Ключевые слова: формальная грамматика, порождающее правило, автомат, множество выбора, нисходящий разбор, одиночная ошибка, нейтрализация ошибок, компилятор

* *Статья получена 10 января 2019 г.*

ВВЕДЕНИЕ

Разработка транслятора для нового языка программирования [1], отличающегося от уже существующих языков рядом существенных свойств и характеристик, может быть связана с необходимостью поиска путей решения ранее неизвестных задач [2, 3] или, по меньшей мере, сочетания способов решения задач, ранее рассматривавшихся отдельно друг от друга. Некоторые из таких задач и пути их решения применительно к новому функционально-императивному языку программирования El были рассмотрены в [4].

С точки зрения программиста-пользователя существенной характеристикой компилятора является его способность обнаруживать и адекватно описывать все реально имеющиеся в его программе лексические, синтаксические и семантические ошибки. Другими словами, компилятор, останавливающийся после первой обнаруженной ошибки, имеет очень плохие потребительские качества. В хорошем компиляторе должна быть реализована эффективная стратегия нейтрализации ошибок. Лексические ошибки могут рассматриваться как синтаксические, поскольку никакое лексически неправильное слово не может являться правильной частью какой-либо синтаксической конструкции. Поэтому обработку лексических и синтаксических ошибок можно рассматривать совместно.

Применительно к семантическим ошибкам следует отметить, что далеко не все они могут быть обнаружены на этапе компиляции, некоторые проявляются только при выполнении скомпилированной программы (достаточно вспомнить известную ситуацию возможного деления на ноль).

В этой работе не рассматриваются семантические ошибки и проблемы их обработки. Предлагаются некоторые модификации известных стратегий нейтрализации синтаксических ошибок и описывается их реализация в трансляторе языка El.

1. ПОСТАНОВКА ЗАДАЧИ

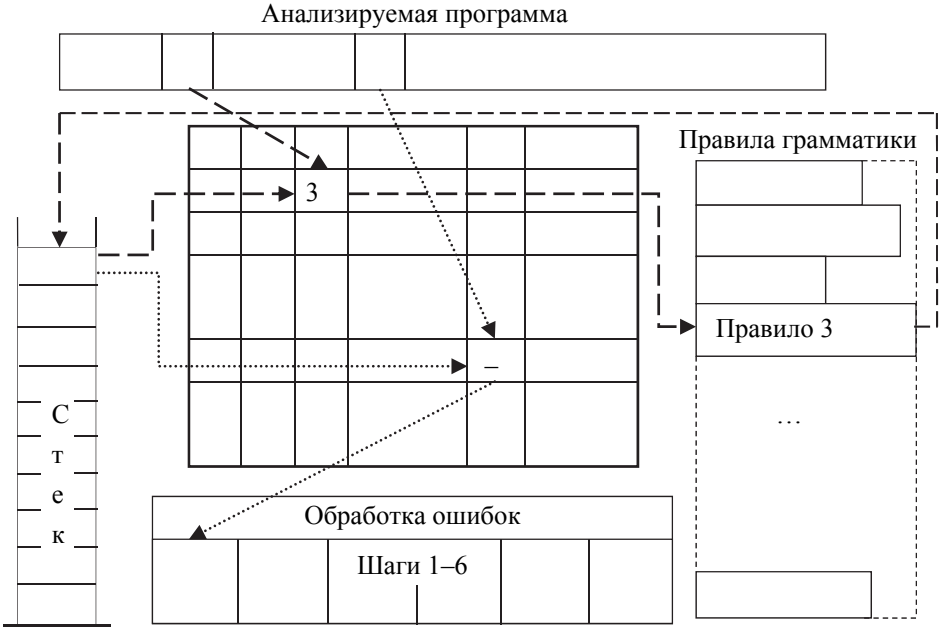
Синтаксический анализатор (парсер) транслятора функционально-императивного языка El представляет собой детерминированный стековый автомат с одним состоянием [5, 6], осуществляющий нисходящий разбор входного текста. Автомат парсера был построен пакетом автоматизации проектирования трансляторов «Вебтранслаб» [7] путем преобразования LL(1)-грамматики языка El. Небольшой фрагмент этой грамматики (без встроенных в нее действий, расширяющих парсер и осуществляющих функции семантического анализа, оптимизации и генерации кода) выглядит так:

```
module : moduleName [ options ]? [ imports ]* [ initialyzers ]*
    [ functionDef ]+
moduleName : "module" ident [ "." ident ]* ";"
options : "options" "(" options ")" ";"
imports : "import" ident [ ("." | ",") ident ]* ";"
initialyzers : [ tuple [ guard ]? ]? block
initialyzers : "record" ident "(" options ")" ";"
functionDef : [access ]? ident argTuple bodyFunction
access : "public" | "private"
```

argTuple : "(" [argTupleElem ["," argTupleElem]]? ")"*
argTupleElem : varDef [varType]? ident
argTupleElem : varType ident
argTupleElem : expression
bodyFunction : guard block restBody
bodyFunction : block
restBody : bodyFunction
...

Полная грамматика языка El [8] содержит 342 порождающих правила (без специальных правил для типичных ошибок), в которых используются 112 терминалов (токенов) и 157 нетерминалов. Принадлежность грамматики к классу LL(1) позволяет реализовать на ее основе детерминированный нисходящий синтаксический разбор [3]. Для построения El-компилятора эта грамматика была преобразована в стековый автомат, управляемый входным токеном и символом, снимаемым с верхушки стека [3, 5, 9].

Управляющая таблица автомата, полученная в результате этого преобразования, имеет размер 157 строк на 112 колонок и является сильно разреженной. В программе парсера эта таблица хранится в виде связанного списка строк, содержащих только непустые клетки. Структура автомата и его логические связи с обрабатываемой программой (сверху), стеком (слева), грамматикой (справа) и функционалом нейтрализации ошибок (снизу) показаны на рисунке.



Взаимные связи управляющей таблицы парсера, анализируемой программы, стека, правил грамматики и функционала нейтрализации ошибок

Interconnections of the parser control table, the program being analyzed, the, grammar rules stack, and error neutralization functional

Управляющая таблица в центре показана в обычном неразреженном представлении. Каждая строка соответствует одному нетерминалу, который может находиться на верхушке стека автомата. Непустые клетки управляющей таблицы содержат порядковые номера порождающих правил, правые части которых должны быть занесены в стек автомата в порядке от последнего символа к первому. С каждой непустой клеткой сопоставлен терминал, принадлежащий множеству выбора [10, 11] того правила, порядковый номер которого находится в клетке.

Перед запуском парсера в его стек заносятся вначале терминал (токен) *EndOfFile*, а затем начальный нетерминал грамматики (в данном случае это нетерминал *module*). Из входного файла считывается первый токен (далее считанный, но еще не обработанный токен называется текущим). Автомат запускается, далее он работает циклически [12, 13].

В каждом повторении цикла работы автомат синтаксического анализатора снимает один верхний символ со стека и проверяет его тип.

Если это терминал, то он сравнивается с текущим токеном. При совпадении текущий токен заменяется следующим из входного файла и автомат переходит к следующему циклу. Несовпадение означает первичное обнаружение синтаксической ошибки и вызывает запуск процедуры нейтрализации ошибок. Перед входом в режим нейтрализации выполняется сохранение полного состояния автомата, в том числе содержимого его стека и позиции ошибочного токена во входном потоке.

Если символ, снятый с верхушки стека, является нетерминалом, то из управляющей таблицы автомата выбирается клетка, находящаяся на пересечении строки, соответствующей этому нетерминалу, и колонки, озаглавленной текущим токеном. Если клетка содержит число (номер правила), то правая часть этого правила посимвольно в обратном порядке (начиная с последнего символа) заносится в стек и автомат переходит к следующему циклу работы. На рисунке эта ситуация отображена стрелками крупного пунктира, связанными с клеткой таблицы, содержащей номер правила 3. Если же клетка пуста (нет правила для замены снятого со стека нетерминала при текущем входном токене), то это означает первичное обнаружение синтаксической ошибки. На рисунке эта ситуация отображена стрелками мелкого пунктира, связанными с пустой клеткой управляющей таблицы. Осуществляется запуск процедуры нейтрализации ошибок после полного сохранения состояния автомата, в том числе содержимого его стека и позиции ошибочного токена во входном потоке.

Обнаружение любой ошибки приводит к формированию подробного диагностического сообщения для автора программы и останову автомата до завершения выполнения процедуры нейтрализации.

Идея нейтрализации ошибок [14–16] основывается на понятии одиночной ошибки. Это означает, что программа, содержащая синтаксическую ошибку, может быть представлена как правильное начало – цепочка токенов φ, один ошибочный токен ε и, вероятно, правильное продолжение – цепочка ψ:

program : $\varphi\ \varepsilon\ \psi$

Таким образом, суть задачи нейтрализации состоит в том, чтобы определить такой способ модификации одного ошибочного токена, который позволяет после его нахождения перезапустить автомат для нормальной обработки правильного продолжения ψ – остатка обрабатываемой программы.

Нейтрализация предпринимается для того, чтобы предоставить программисту максимум информации о действительно допущенных в программе ошибках, но должна выполняться так, чтобы не формировать диагностических сообщений о несуществующих (называемых наведенными) ошибках.

2. ОБРАБОТКА ОШИБОК В EL-ТРАНСЛЯТОРЕ

Обычно под нейтрализацией [14–16] понимается попытка продолжения анализа программы после обнаружения любой синтаксической ошибки путем последовательного перебора следующих вариантов поиска «исправления» ошибки:

- удаление ошибочного токена;
- вставка перед ошибочным токеном некоторого другого токена;
- замена ошибочного токена некоторым другим токеном.

После каждой модификации ошибочного токена полностью восстанавливается состояние парсера, зафиксированное на момент обнаружения ошибки, в том числе содержимого стека автомата, и позиция текущего токена во входном потоке, и выполняется его перезапуск. Новый останов по ошибке на некотором заданном расстоянии от ошибочного токена воспринимается как неудача данной попытки нейтрализации ошибки. В этом случае выбирается следующий вариант модификации ошибочного токена и процедура перезапуска парсера повторяется.

В том случае, если после очередного перезапуска автомат заново не останавливается по ошибке на заданном расстоянии от точки первичного останова, ошибка считается нейтрализованной и парсер возвращается в обычный режим работы. Если же на очередном шаге оказались исчерпаны все возможные варианты модификации ошибочного токена, то ошибка квалифицируется как множественная (не одиночная). Парсер либо останавливает обработку транслируемой программы вообще, либо переходит в режим паники [17–19], ожидая появления на входе одного из так называемых реперных токенов. Такими токенами обычно считаются слова, завершающие целые синтаксические конструкции [20, 21].

При первом же входе в режим нейтрализации ошибок блокируются все процессы синтеза, вызываемые парсером в нормальном режиме работы: преобразование последовательности токенов в абстрактное синтаксическое дерево, далее – в псевдокод, его оптимизация и формирование объектного кода.

В трансляторе языка El с целью уменьшения количества неанализируемых токенов при переходе к режиму паники описанный механизм модифицирован следующим образом. Процедура нейтрализации синтаксических ошибок организована циклически. Изменена по сравнению с обычно описываемой в литературе очередность применения модификаций ошибочного токена. Вначале обрабатываются все допустимые вставки, затем все допустимые замены, после чего – удаление ошибочного токена. Цикл, включающий все вставки, все замены и одно удаление, повторяется до тех пор, пока не бу-

дет найден хотя бы один вариант успешной нейтрализации. В каждом повторении цикла реализуются следующие шаги.

Шаг 1, инициализация. Для выполнения всех попыток вставки и замены предварительно формируется множество всех допустимых токенов. Его составляет множество выбора символа, находящегося на верхушке стека автомата. Если этот символ – терминал, то только он и будет входить в множество допустимых токенов. Если же это нетерминал, то вычисляется объединение множеств выбора всех правил грамматики с этим нетерминалом в левой части. Только допустимые токены будут использоваться в попытках вставки/замены при выполнении шагов 2, 3 и 4 процедуры нейтрализации. Символ, для которого вычислено множество всех допустимых токенов, считается обработанной верхушкой стека (см. шаг 5).

Шаг 2, вставка. Перебираются все допустимые токены. Для каждого из них полностью восстанавливается стек автомата, очередной допустимый токен вставляется перед позицией ошибочного символа во входной поток (вставка), и парсер перезапускается. Если он останавливается вновь на первичном ошибочном токене или на токене, следующем за ним, то попытка считается неудачной. В противном случае попытка нейтрализации воспринимается как успешная и определяется расстояние в токенах до следующего останова по ошибке либо до исчерпания входного потока, т. е. считывания токена *EndOfFile*. В последнем случае (при достижении конца входного потока) процесс компиляции завершается, поскольку больше ошибок в обрабатываемой программе найдено быть не может. Поэтому нет смысла искать другие варианты нейтрализации первичной ошибки. Если же автомат остановился по обнаружению новой синтаксической ошибки, то запоминается тот вариант допустимого токена, для которого расстояние от первичной до следующей ошибки является наибольшим.

Шаг 3, замена. Аналогичным образом обрабатывается множество допустимых символов для реализации всех попыток замены. При каждой из них полностью восстанавливается стек автомата, допустимый токен вставляется вместо ошибочного во входной поток, и парсер перезапускается. Попытка считается неудачной, если автомат останавливается по ошибке на токене, следующем во входном потоке за первичным ошибочным. Иначе попытка считается успешной и, точно так же, как на шаге 2, фиксируется наиболее удачный вариант нейтрализации в случае последующего останова по ошибке либо работа компилятора завершается, если достигнут конец входного потока.

Шаг 4, удаление. Те же действия с теми же условиями оценки успешности попытки нейтрализации выполняются при удалении первичного ошибочного токена. После этого выполняется либо шаг 5, либо шаг 6 в зависимости от того, была ли успешной хотя бы одна из попыток вставки/замены/удаления.

Шаг 5, расширение. Если ни одна из выполненных на шагах 2–4 попыток нейтрализации первичной ошибки не удалась, то реализуется существенно модифицированный режим паники. Как известно, при переходе в этот режим из входного потока считываются и без анализа отбрасываются все токены, не совпадающие с одним из так называемых реперных символов. Такие символы выбираются разработчиком транслятора и в правильных программах обычно завершают такие синтаксические конструкции, как оператор

(символ «;») или блок операторов (символ «}»). После обнаружения одного из этих токенов во входном потоке анализируется стек автомата и с его верхушки последовательно сбрасываются те символы, которые не являются эквивалентами конструкции, соответствующей считанному реперному символу. В отличие от такого способа в El-компиляторе после каждого повторения шага 5 множество всех допустимых токенов, вычисленное на шаге 1, дополняется токенами, принадлежащими множеству выбора символа, находящегося в стеке под обработанной верхушкой. Обработанная верхушка расширяется на одну позицию вниз. После этого повторяется шаг 4 и, в зависимости от его успешности, либо шаг 5, либо шаг 6.

Шаг 6, завершение. Если в очередном цикле удалась хотя бы одна попытка вставки/замены/удаления, то парсер выводится из режима нейтрализации. Восстанавливается состояние стека автомата и позиция во входном потоке, сохраненная для наиболее успешной попытки, и автомат перезапускается. Если в процессе нейтрализации шаг 4 выполнялся несколько раз, то перед перезапуском автомата просматривается обработанная верхушка стека и с него сбрасываются все те символы, для которых первый из неудаленных входной токен не принадлежит множеству выбора. Для каждой следующей вновь обнаруженной синтаксической ошибки весь процесс нейтрализации повторяется (следует помнить, что достижение конца входного потока в любой попытке нейтрализации приводит к немедленному завершению работы транслятора).

Приведем пример ситуации, в которой предложенный и реализованный метод нейтрализации позволяет пробросить без анализа меньше входных токенов, чем известный режим паники. Пусть в С-подобном языке (в языке El синтаксис присваивания такой же) сильно искаженный оператор присваивания выглядит так:

*a = b +)) c – d / e + * f ;*

Ни вставки, ни замены, ни удаление одного символа не приведут к успешной нейтрализации ошибки, обнаруженной при чтении первой закрывающей круглой скобки. Вход в режим паники приведет к игнорированию всех входных токенов вплоть до «;». При этом не будет обнаружена вторая ошибка, допущенная в этом операторе (последовательность токенов « *) f + g)*»), которая могла бы быть нейтрализована путем замены закрывающей скобки на открывающую.

Парсер El-компилятора, удалив сначала первую, а затем вторую открывающую скобки, восстановит процесс распознавания арифметического выражения при чтении токена «c» и вновь остановится по ошибке, прочитав токен «*».

Описанная процедура нейтрализации синтаксических ошибок в компиляторе языка El сочетается с включением в грамматику «правил для типичных ошибок». Эти правила добавляются для выявления и обработки таких возможных ситуаций, когда для успешной нейтрализации ошибки требуется вставить в анализируемую программу (или заменить в ней) не один, а два или три токена. Подобные ошибки возможны в программах на языке El, их эффективная обработка какими-либо другими способами представляется весьма затруднительной.

Примером подобной ошибки может являться появление одного или нескольких выражений *when* (или *of*) вне контекста управляющего оператора *by*. Любые одиночные модификации ошибочного токена *when* (или *of*) согласно вышеприведенному алгоритму нейтрализации в лучшем случае приведут к возникновению наведенных ошибок при обработке токена «:», следующего после выражения за ним, т. е. на расстоянии, большем единицы. Другими словами, описанная выше процедура нейтрализации будет завершаться успехом, но приводить к формированию диагностических сообщений о несуществующих ошибках.

Выходом из этой ситуации [22], используемым и в El-компиляторе, стало включение в грамматику дополнительных правил, согласно которым появление выражений *when* и *of* вне контекста оператора *by* не вызывают запуска процедуры нейтрализации одиночных ошибок. Однако в эти правила в форме действий включены вызовы специально написанных методов класса «парсер», которые формируют сообщения об ошибке и блокируют процессы синтеза.

Пример правила для типичной ошибки в грамматике языка El выглядит так:

operator : "when" Expression ":" { typicalError(tError.whenOutsideBy);}

В этом правиле в фигурных скобках записан вызов функции фиксации типичной ошибки, код которой задается значением элемента перечисления *tError.whenOutsideBy*. Функция *typicalError* вызывается в момент применения этого правила и выполняет все те действия, которые необходимо выполнить при обнаружении любой синтаксической ошибки: формирование диагностического сообщения, блокировка процессов синтеза результата компиляции и т. д. Однако процедура нейтрализации ошибок не запускается. Как следствие, не будут формироваться ненужные диагностические сообщения о наведенных ошибках.

Кроме приведенного примера с выражениями *when* и *of* вне контекста охватывающего их оператора *by* в грамматику языка El включены и другие «правила для типичных ошибок», уменьшающие возможность выдачи парсером сообщений о наведенных ошибках.

ЗАКЛЮЧЕНИЕ

Предложен и реализован в трансляторе языка El способ нейтрализации синтаксических ошибок, эффективно использующий особенности алгоритма детерминированного синтаксического разбора, реализованного в виде стекового автомата, который управляется входным токеном и символом с верхушки стека.

Для реализации этого способа в момент первичного обнаружения ошибки формируется множество допустимых токенов, использование которых для нейтрализации не приведет к немедленному останову парсера. Выполняется полный перебор всех возможных вариантов модификации одного ошибочного токена – поочередная вставка допустимых токенов перед ним, замена его на допустимый токен и удаление его – для выбора того варианта перезапуска

автомата, при котором следующая синтаксическая ошибка будет обнаружена на максимальном расстоянии от первичной. В том случае, если никакая модификация одиночного ошибочного токена не приводит к успешной нейтрализации, после каждого удаления расширяется множество допустимых токенов, вычисляемое как объединение множеств выбора нескольких символов, находящихся на верхушке стека автомата. После этого повторяются попытки модификации текущего токена. Реализация такого алгоритма вместо переключения в режим паники позволяет уменьшить количество неанализируемых парсером токенов.

Описываемый алгоритм нейтрализации сочетается в El-компиляторе с «правилами для типичных ошибок», включаемыми в грамматику для обработки таких возможных ситуаций, когда для успешной нейтрализации ошибки требуется вставить в анализируемую программу (или заменить в ней) не один, а два или три токена. Появление подобных ошибок в принципе возможно в программах на языке El, их эффективная обработка другими способами представляется весьма затруднительной.

СПИСОК ЛИТЕРАТУРЫ

1. *Maliavko A.* Novel functional-imperative programming language El: a brief introduction // Proceedings of the 2018 International Multi-Conference on Industrial Engineering and Modern Technologies (FarEastCon). – Vladivostok, October 2018. – P. 1–7.

2. *Crespi-Reghizzi S.* Formal languages and compilation. – London: Springer, 2009. – 364 p.

3. Compilers: principles, techniques, and tools / A. Aho, M. Lam, R. Sethi, J. Ullman. – Reading: Addison-Wesley, 2006. – 795 p.

4. *Maliavko A., Zhurkin P., Nagornov N.* The functionally-imperative programming language El and its translator // 14th International Scientific-Technical Conference on Actual problems of Electronic Instrument Engineering (APEIE-2018). – Novosibirsk, 2018. – Vol. 1, pt. 4. – P. 469–476.

5. *Watson D.* A practical approach to compiler construction. – Springer, 2017. – 254 p.

6. *Carroll J., Long D.* Theory of finite automata with an introduction to formal languages. – New Jersey: Prentice Hall, 1989. – 447 p.

7. *Малявко А.А.* Использование веб-приложений и веб-технологий при разработке учебного программного обеспечения для изучения методов трансляции // Современное образование: технические университеты в модернизации экономики России: материалы Международной научно-методической конференции, 27–28 января 2011 года. – Томск: Изд-во ТУСУР, 2011. – С. 45–47.

8. *Meduna A.* Formal languages and computation: models and their applications. – Boca Raton: CRC Press, 2014. – 315 p.

9. *Rosenkrantz D., Stearns R.* Properties of deterministic top down grammars // Information and Control. – 1970. – Vol. 17 (3). – P. 226–256.

10. *Waite W., Goos G.* Compiler construction. – Heidelberg: Springer, 2012.

11. *Dos Reis A.* Compiler construction using Java, JavaCC, and Yacc. – Hoboken: Wiley & Sons, 2012. – 664 p.

12. *Малявко А.А.* Формальные языки и компиляторы: учебное пособие для вузов. – М.: Юрайт, 2017. – 429 с.

13. *Kumar R.* Theory of automata, languages and computation. – Tata: McGraw-Hill, 2010.

14. *Medeiros S. Mascarenhas F.* Syntax error recovery in parsing expression grammars // Applied computing 2018: the 33rd Annual ACM Symposium on Applied Computing. – New York, 2018.

15. *Cooper K., Torczon L.* Engineering a compiler. – San Francisco: Morgan Kaufmann, 2003.

16. *Wilhelm R., Seidl H., Hack S.* Compiler design: syntactic and semantic analysis. – New York: Springer, 2013. – 216 p.

17. *Parr T., Harwell S., Fisher K.* Adaptive LL(*) parsing: the power of dynamic analysis // Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications. – New York: ACM, 2014. – P. 579–598.

18. *Chandak M., Khurana K.* Compiler design. – Hyderabad, India: Universitet press, 2018. – 480 p.

19. *Grune R., Jacobs C.* Parsing techniques: a practical guide. – New York: Springer, 2007. – 585 p.

20. *Plaisted D.* Source-to-source translation and software engineering // JSEA Special Issue on Software Dependability. – 2013. – Vol. 6, N 4A.

21. *Afroozeh A., Izmaylova A.* Faster, practical GLL parsing // Compiler Construction: 24th International Conference. – Heidelberg: Springer, 2015. – P. 89–108.

22. *Safonov V.* Trustworthy compilers. – Hoboken: Wiley & Sons, 2010. – 317 p.

Малявко Александр Антонович, кандидат технических наук, доцент кафедры вычислительной техники факультета автоматики и вычислительной техники Новосибирского государственного технического университета. Основные направления научных исследований: языки программирования и теория трансляции, нейронные сети с элементами самообучения, параллельные вычисления. Имеет около 50 публикаций, в их числе одна монография и один учебник для вузов. E-mail: a. malyavko@corp.nstu.ru

Malyavko Alexander Antonovich, PhD (Eng.), associate professor, Department of Computer Engineering, Faculty of Automation and Computer Engineering, Novosibirsk State Technical University. His research interests are focused on programming languages and translation theory, neural networks with elements of self-learning and parallel computations. He is the author of about 50 publications including 1 monograph and 1 textbook for universities. E-mail: a. malyavko@corp.nstu.ru

DOI: 10.17212/1814-1196-2019-2-37-48

Errors handling in the parser for the El-language compiler [*]

A.A. MALYAVKO

Novosibirsk State Technical University, 20, K. Marx Prospekt, Novosibirsk, 630073, Russian Federation
a.malyavko@corp.nstu.ru

Abstract

We consider the problem of processing syntax errors in the compiler of the functional-imperative programming language El aimed at continuing syntactic parsing after a stop to find the maximum possible number of actually made errors. A combination of several well-known methods of neutralizing syntax errors is proposed. It is focused on the features of the deterministic parsing algorithm used in the translator. The compiler parser is implemented in the form of a stack automaton controlled by an input token and a symbol taken from the top of the stack. The proposed error neutralization method is based on a complete enumeration of all possible options for modifying one erroneous token – its alternate insertion, replacement and deletion to select such an option for restarting the automat in which the following syntax error is detected

[*] *Received 10 January 2019.*

at the maximum distance from the initially detected error. If no modification of a single erroneous token leads to a successful neutralization, then in the proposed method, after each deletion of the input token, a set of valid tokens expands, calculated as an integration of multiple selection sets for symbols at the top of the automaton stack. This transition from processing a single error to processing a multiple error allows you to reduce the number of input tokens read by the parser without analysis as compared to the known panic mode. The proposed neutralization algorithm combines in the El-compiler with “rules for typical errors”, which are included in the grammar for such possible situations when, in order to successfully neutralize an error, you need to insert (or replace) not one, but two or three tokens.

Keywords: grammar, production rule, automat, selection sets, top-down parsing, single error, errors neutralization, compiler

REFERENCES

1. Maliavko A. Novel functional-imperative programming language El: a brief introduction. *Proceedings of the 2018 International Multi-Conference on Industrial Engineering and Modern Technologies (FarEastCon)*, Vladivostok, October 2018, pp. 1–7.

2. Crespi-Reghizzi S. *Formal languages and compilation*. London, Springer, 2009. 364 p.

3. Aho A., Lam M., Sethi R., Ullman J. *Compilers: principles, techniques, and tools*. Reading, Addison-Wesley, 2006. 795 p.

4. Maliavko A., Zhurkin P., Nagornov N. The functionally-imperative programming language El and its translator. *14th International Scientific-Technical Conference on Actual problems of Electronic Instrument Engineering (APEIE-2018)*, Novosibirsk, 2018, vol. 1, pt. 4, pp. 469–476.

5. Watson D. *A practical approach to compiler construction*. Springer, 2017. 254 p.

6. Carroll J., Long D. *Theory of finite automata with an introduction to formal languages*. New Jersey, Prentice Hall, 1989. 447 p.

7. Malyavko A.A. [Using web applications and web technologies in the development of educational software for studying methods of translation]. *Sovremennoe obrazovanie: tekhnicheskie universitety v modernizatsii ekonomiki Rossii: materialy Mezhdunarodnoi nauchno-metodicheskoi konferentsii* [Modern education: technical universities in the modernization of the Russian economy: materials of the International Scientific and Methodological Conference]. Tomsk, TUSUR Publ., 2011, pp. 45–47. (In Russian).

8. Meduna A. *Formal languages and computation: models and their applications.* Boca Raton, CRC Press, 2014. 315 p.

9. Rosenkrantz D., Stearns R. Properties of deterministic top down grammars. *Information and Control*, 1970, vol. 17 (3), pp. 226–256.

10. Waite W., Goos G. *Compiler construction.* Heidelberg, Springer, 2012.

11. Dos Reis A. *Compiler construction using Java, JavaCC, and Yacc*. Hoboken, Wiley & Sons, 2012. 664 p.

12. Maliavko A. *Formal'nye yazyki i kompilyatory* [Formal languages and compilers]. Moscow, Yurait Publ., 2017. 429 p.

13. Kumar R. *Theory of automata, languages and computation*. Tata, McGraw-Hill, 2010.

14. Medeiros S. Mascarenhas F. Syntax error recovery in parsing expression grammars. *Applied computing 2018: the 33rd Annual ACM Symposium on Applied Computing*. New York, 2018.

15. Cooper K., Torczon L. *Engineering a compiler.* San Francisco, Morgan Kaufmann, 2003.

16. Wilhelm R., Seidl H., Hack S. Compiler design: syntactic and semantic analysis. New York, Springer, 2013. 216 p.

17. Parr T., Harwell S., Fisher K. Adaptive LL(*) parsing: the power of dynamic analysis. *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications*. New York, ACM, 2014, pp. 579–598.

18. Chandak M., Khurana K. *Compiler design*. Hyderabad, India, Universitet press, 2018. 480 p.

19. Grune R., Jacobs *C. Parsing techniques:* a *practical guide.* New York, Springer, 2007. 585 p.

20. Plaisted D. Source-to-source translation and software engineering. *JSEA Special Issue on Software Dependability*, 2013, vol. 6,no. 4A.

21. Afroozeh A., Izmaylova A. Faster, practical GLL parsing. *Compiler Construction: 24th International Conference*. Heidelberg, Springer, 2015, pp. 89–108.

22. Safonov V. *Trustworthy compilers*. Hoboken, Wiley & Sons, 2010. 317 p.

Для цитирования:

Малявко А.А. Обработка ошибок в синтаксическом анализаторе компилятора языка El / А.А. Малявко // Научный вестник НГТУ. – 2019. – № 2 (75). – С. 37–48. – DOI: 10.17212/1814-1196-2019-2-37-48.

For citation:

Maliavko A.A. Obrabotka oshibok v sintaksicheskom analizatore kompilyatora yazyka El [Errors handling in the parser of the El-language compiler]. *Nauchnyi vestnik Novosibirskogo gosudarstvennogo tekhnicheskogo universiteta – Science bulletin of the Novosibirsk state technical university*, 2019, no. 2 (75), pp. 37–48. DOI: 10.17212/1814-1196-2019-2-37-48.