V. A. Kharakhinov, Graduate Student, e-mail: tes4obse@mail.ru,
National Research Irkutsk State Technical University

The Genetic Algorithm as the Alternative Method for Training Kohonen Layer

In this article genetic algorithm suggested as the alternative method for training Kohonen layer. This paper considers evaluations of cluster analysis which were produced by follow methods: K-means; Kohonen layer trained with standard kohonen learning algorithm; Kohonen layer trained with genetic algorithm.

The evaluations of clustering were determined using two indices: the Rand index and Adjusted Rand index.

In addition to this, the article contains the graphical display of the training time for standard kohonen learning algorithm and realized genetic algorithm.

The cluster analysis was accomplished using banknote authentication dataset that described by a set of numerical feature and considers two output flag: authentic and counterfeit.

Keywords: *Cluster analysis, K-means, Kohonen layer, genetic algorithm, Rand index*

DOI: 10.17587/it.24.642-648

References

1. **Exploratory** *Data Analysis with MATLAB*, 2nd Edition. URL: http://pi-sigma.info/EDA.htm.
2. **Mandel' I. D.** *Klasternyj analiz*, Moscow, Finansy i statistika, 1988, 176 p. (in Russian).
3. **Gladkov L. A., Kurejchik V. V., Kurejchik V. M.** *Geneticheskie algoritmy,* Moscow, Fizmatlit, 2006, 320 p. (in Russian).
4. **Goldberg D. E.** *Genetic Algorithms in Search, Optimization and Machine Learning*, Addison-Wesley Publishing Company, Inc., 1989, 412 p.
5. **Rutkovskaja D., Pilin'skij M., Rutkovskij L.** *Nejronnye seti, geneticheskie algoritmy i nechetkie sistemy*, Moscow, Gorjachaja linija-Telekom, 2006, 452 p. (in Russian).
6. **Wierzchon S. T., Klopotek M. A.** *Modern Algorithms of Cluster Analysis*, Springer, 2018. 421 p.
7. **Mirkin B. G.** *Metody klaster-analiza dlja podderzhki prinjatija reshenij: obzor*, Moscow, Vysshaja shkola jekonomiki, 2011. 88 p. (in Russian).
8. **Maulik U., Bandyopadhyay S., Mukhopadhyay A.** *Multiobjective Genetic Algorithms for Clustering,* Berlin—Heidelberg, Springer-Verlag, 2011, 281 p.
9. **Rand W. M.** Objective criteria for the evaluation of clustering methods, *Journal of the American Statistical Association*, 1971, vol. 66, pp. 846—850.
10. **Santos J. M., Embrechts M.** *On the Use of the Adjusted Rand Index as a Metric for Evaluating Supervised Classification.* Berlin—Heidelberg. Springer-Verlag, 2009. Vol. 2, pp. 175—184.
11. **Repozitorij** real'nyh i model'nyh zadach mashinnogo obuchenija. URL: https://archive.ics.uci.edu/ml/datasets/banknote + authentication (in Russian).

УДК 004.434 DOI: 10.17587/it.24.648-656

Л. Н. Лядова, канд. физ.-мат. наук, доц., доц. кафедры информационных технологий в бизнесе, e-mail: LLyadova@hse.ru,
А. О. Сухов, канд. физ.-мат. наук, доц. кафедры информационных технологий в бизнесе, e-mail: ASuhov@hse.ru,
Е. Ю. Медведева, магистрант, e-mail: medvedevaeyu@mail.ru,
Национальный исследовательский университет "Высшая школа экономики", г. Пермь

Алгоритмы синтаксического разбора для текстовых динамически настраиваемых предметно-ориентированных языков

Предложены алгоритмы разбора для текстовых динамически настраиваемых предметно-ориентированных языков и проверки синтаксической корректности написанных с их помощью программ. В процессе своей работы на основе описания расширенной грамматики языка анализатор строит псевдодерево разбора, которое в дальнейшем используется при проверке синтаксической корректности программ пользователя. В основе алгоритма проверки синтаксиса лежит метод леворекурсивного спуска с возвратом.

Ключевые слова: *разбор грамматики, проверка синтаксиса, предметно-ориентированные языки, текстовые языки, языковой инструментарий, метод леворекурсивного спуска, формальные грамматики, дерево разбора*

Введение

В настоящее время все большее число инструментальных программных систем предоставляют в распоряжение пользователей встроенные текстовые языки программирования, позволяющие выполнять создание моделей предметной

области решаемой задачи, настраивать систему на предметную область, меняющиеся условия эксплуатации, потребности пользователей. Примерами таких языков являются:

- встроенный язык *1С: Предприятие*, предназначенный для настройки системы на предметную область деятельности предприятия [1];
- языки, интегрированные в системы имитационного моделирования (*Plant Simulation, AnyLogic* и др.), предназначенные для построения имитационных моделей [2—3];
- языки, встроенные в средства автоматизации проектирования и графические пакеты (*Autodesk 3ds Max, Autodesk AutoCAD, CorelDraw* и др.), используемые для автоматизации рутинных операций и программного создания мелких фрагментов моделей [4].

Большинство таких систем предоставляет в распоряжение пользователей предметно-ориентированные языки (Domain-Specific Language, DSL) — языки, разработанные для решения определенного круга задач в конкретной предметной области. Алфавит и конструкции таких языков близки понятиям предметной области, ее терминологии, отражают операции предметной области, поэтому работать с ними могут не только профессиональные ИТ-специалисты, но и будущие пользователи системы, обладающие навыками алгоритмизации, построения моделей предметной области. При этом в процессе использования таких языков часто возникает потребность не просто построить или модифицировать описание решения задачи, но и настроить язык на потребности пользователей или предметную область. Языки, описание которых может быть модифицировано в процессе эксплуатации системы, в которую они интегрированы, без перегенерации ее исходного кода, называются *динамически настраиваемыми языками*.

В работе [5] описан подход к разработке инструментальных средств создания текстовых динамически настраиваемых DSL. Данная статья является продолжением работы [5].

Целью данного исследования является разработка алгоритмов синтаксического разбора грамматик текстовых динамически настраиваемых DSL, проверки синтаксической корректности написанных с их помощью программ.

1. Языковые инструментарии

Для разработки новых DSL используется специальный класс программного обеспечения — языковые инструментарии (language workbench).

На сегодняшний день существует несколько языковых инструментариев, которые позволяют создавать текстовые DSL: OpenArchitectureWare [6], Meta Programming System [7], IDE Meta-Tooling Platform [8], MontiCore [9], Spoofax Language Workbench [10] и др. Каждый из инструментов имеет как достоинства, так и ограничения в использовании. Подробный анализ языковых инструментариев приведен в работе [5].

Устранить ограничения существующих инструментариев решено при разработке собственного языкового инструментария, подход к созданию которого описан в работе [5].

Основным преимуществом разрабатываемого языкового инструментария является то, что он позволяет выполнять динамическое изменение описания языка. Данная возможность позволяет модифицировать описания языка и разработанных с его помощью моделей во время работы системы без необходимости перегенерации кода транслятора и редактора языка. Например, в процессе функционирования системы имитационного моделирования ИТ-инфраструктуры появилась потребность создания узлов нового типа. Для этого требуется добавить в описание DSL новую языковую конструкцию. Традиционный подход к созданию DSL предполагает, что разработчику необходимо с помощью языкового инструментария изменить описание DSL, модифицировать редактор языка, далее сгенерировать исходный код транслятора и редактора, и только после этого пользователь может вернуться к построению новых моделей. Данный процесс является длительным. Было бы намного удобнее внести все изменения во время работы системы и приступить к построению новых моделей без необходимости перегенерации исходного кода транслятора и редактора языка.

Кроме того, разрабатываемый языковой инструментарий предоставляет возможность выполнять многоуровневое моделирование. Наличие такой возможности упростит процесс создания новых специализированных DSL, поскольку, определив один DSL общего назначения, например, для описания систем массового обслуживания, разработчик может создать на его основе несколько специализированных DSL имитационного моделирования для конкретных предметных областей, например, языки моделирования ИТ-инфраструктуры, цепочек поставок, движения автотранспорта и др.

2. Описание подхода к построению предметно-ориентированных языков

В процессе создания программы на DSL с помощью разрабатываемого языкового инструментария строится целая иерархия моделей:

- *метаязык* — язык, с помощью которого выполняется описание грамматики создаваемых DSL. Перед началом работы языковой инструментарий уже включает встроенный метаязык, который в случае необходимости может быть изменен пользователем;
- *метамодель* — модель создаваемого предметно-ориентированного языка, т. е. описание его грамматики;
- *модель предметной области* — программа, написанная с помощью DSL. В дальнейшем на ее основе может быть сгенерирован код на одном из целевых языков, например, языке программирования высокого уровня.

Каждая нижестоящая модель является экземпляром вышестоящей и строится путем создания экземпляров конструкций вышестоящей модели.

Таким образом, процесс создания программы на DSL состоит из следующих этапов:

1. Настройка метаязыка на особенности предметной области за счет добавления в его описание новых конструкций или изменения существующих. Данный этап является необязательным и выполняется только в том случае, если требуется настройка метаязыка на специфику предметной области, например, при создании специализированных DSL для конкретных предметных областей на основе языка общего назначения.

2. Построение метамодели, т. е. описание грамматики разрабатываемого DSL с помощью конструкций метаязыка.

3. Описание правил генерации кода. На данном этапе на уровне метамодели определяются правила преобразования конструкций DSL в конструкции одного из целевых языков программирования.

4. Построение модели предметной области (написание программы на созданном DSL).

5. Валидация построенной модели предметной области. На данном этапе выполняется проверка синтаксической корректности написанной программы.

6. Генерация кода. На данном этапе выполняется преобразование программы, написанной на DSL, в программу на целевом языке на основе определенных пользователем правил генерации кода.

В рамках данного исследования будут разработаны алгоритмы, используемые на этапах 1, 2, 4, 5.

В качестве встроенного метаязыка языкового инструментария используются формы Бэкуса—Наура, имеющие следующие расширения:

- символы "{", "}" (открывающая и закрывающая фигурные скобки) указывают на возможность присутствия конструкций языка, заключенных в фигурные скобки, ноль или более раз;
- символы "[", "]" (открывающая и закрывающая квадратные скобки) указывают на необязательность присутствия конструкций языка, заключенных в квадратные скобки;
- символ "|" (вертикальная черта) обозначает альтернативность использования конструкций языка, находящихся по разные стороны от данного символа;
- символ "\" (обратный слеш) экранирует символы алфавита описываемого языка, которые совпадают с метасимволами, позволяя применять их в качестве символов алфавита.

Описание грамматики метаязыка с помощью расширенных форм Бэкуса—Наура имеет следующий вид:

```
синтаксис_языка ::= правило {правило}
правило ::= нетерминальный_символ "::=" правая_часть
правая_часть ::= элемент {элемент}
элемент ::= терминальный_символ [элемент] | нетерминальный_символ [элемент] | элемент "|" элемент | "[" элемент "]" | "{" элемент "}"
нетерминальный_символ ::= буква {буква}
терминальный_символ ::= \" буква {буква} \"
```

Начальным символом данной грамматики является нетерминальный символ "синтаксис_языка".

Символ "\" (обратный слеш) используется для экранирования символов алфавита описываемого языка, которые совпадают с метасимволами, позволяя применять их в качестве символов алфавита.

Для упрощения создания новых DSL в описание их грамматик автоматически встраиваются правила для таких конструкций, как "Цифра", "Буква", "Идентификатор".

В случае необходимости создания многоуровневой модели и настройки встроенного метаязыка на предметную область пользователь имеет возможность изменить описание встроенного метаязыка.

Как видно из описания грамматики метаязыка, создаваемые с его помощью DSL будут иметь контекстно-свободную грамматику.

3. Алгоритмы разбора описания грамматики DSL

После того как пользователь с помощью встроенного метаязыка описал синтаксис DSL, необходимо выполнить разбор описания грамматики языка. Промежуточное представление, полученное в результате разбора описания грамматики, будет использовано при проверке синтаксической корректности модели. Кроме того, поскольку языковой инструментарий предоставляет возможность динамического изменения описания языка, то необходимо выполнять отслеживание всех модификаций грамматики DSL и оперативно вносить изменения в промежуточное представление, являющееся результатом работы алгоритма разбора описания грамматики. Таким образом, в качестве промежуточного представления следует выбрать такую структуру данных, которая позволяла бы эффективно выполнять поиск требуемых символов языка в процессе синтаксического разбора модели.

Представление метамодели в виде дерева разбора является наиболее приемлемым вариантом, так как данная структура позволяет быстро проводить поиск нужной вершины, содержащей символ языка, реконфигурацию дерева, а также эффективно применяется при разработке трансляторов для контекстно-свободных грамматик [11, 12].

Поскольку один и тот же символ языка может встретиться в описании правил грамматики более одного раза, необходимо либо дублировать его описание в дереве разбора, либо позволить вершине дерева иметь более одной родительской вершины. Дублирование информации усложнит процесс отслеживания динамических изменений описания языка, поскольку при модификации описания некоторой конструкции необходимо выполнить обход дерева для поиска всех ее вхождений. Именно поэтому в предлагаемом подходе решено использовать второй вариант и отступить от классического определения дерева разбора. Введем определение дерева разбора описания контекстно-свободной грамматики DSL.

Псевдодеревом назовем дерево, имеющее кратные ребра и петли.

Упорядоченное ориентированное псевдодерево $G = (V, E)$ назовем *псевдодеревом разбора* описания для расширенной контекстно-свободной грамматики $Gr = (T, N, P, S, M)$, где T — множество терминальных символов, N — множество нетерминальных символов, P — множество правил вывода, S — начальный символ грамматики, M — множество метасимволов метаязыка, которые использовались для описания грамматики, при выполнении следующих условий:

- каждая вершина псевдодерева принадлежит одному из множеств T, N, M, т. е. $\forall v \in V: v \in$ $\in T \cup N \cup M$; корнем псевдодерева является символ S, листья псевдодерева принадлежат одному из множеств T, M;
- если множество правил вывода расширенной грамматики Gr содержит правило $A ::= A_1A_2...A_n$, где $\forall A \in N$, $A_i \in T \cup N \cup M$, $i = \overline{1, n}$, то вершина A псевдодерева имеет в качестве потомков вершины A_1, A_2, ..., A_n.

На основе построенного псевдодерева система выполняет анализ корректности написанной пользователем программы.

Алгоритм проверки программы в соответствии с правилами описания расширенной контекстно-свободной грамматики имеет следующий вид:

```
global Gramma Gr;
global Tree Tr;
function CheckProgram()
{
       Gr ← InitializeGrammar();
       Tr.V ← ∅;
       Tr.E ← ∅;
       RulesParse();
       l ← NextLexeme();
       NonTerminalNodeParse(Tr.Root);
}
```

Функция *InitializeGrammar* выполняет инициализацию расширенной грамматики на основе заданных пользователем правил: определяет набор терминальных, нетерминальных символов, метасимволов, начальный символ грамматики. Функция *NextLexeme* возвращает очередную лексему исходной программы.

В процессе разбора описания расширенной грамматики DSL языковой инструментарий посредством функции *RulesParse* выполняет последовательный анализ всех правил грамматики в соответствии со следующим алгоритмом:

```
function RulesParse()
{
   foreach (p ∈ Gr.P)
      A ← Tr.FindNode(p.LeftHandSide);
      if (A ≠ null)
         if (A. Children = null)
            p.ParseRightHandSide(A);
         else
            Error(p, "Символ" + A + "описан повторно");
   else
   {
      A← Tr.CreateNonterminalNode (p.LeftHandSide);
      p.ParseRightHandSide(A);
   }
   foreach (A ∈ Tr.V)
      if (A ∈ Gr.N AND A. Children = null)
         Error(p, "Символ" + A + "не определен");
   PrintErrors();
}
```

Функция *Error* заносит информацию об ошибках, допущенных при описании правил расширенной грамматики, в таблицу ошибок. Функция *PrintErrors* выводит список всех ошибок по окончании анализа правил.

Метод анализа правой части *ParseRightHandSide* правила имеет следующий вид:

```
function ParseRightHandSide(Node A)
{
   foreach (s ∈ p.RightHandSide)
   {
      if (s ∈ Gr.T ∪ Gr.M)
         B ← Tr.CreateTerminalNode(s);
      else
      {
         B ← Tr.FindNode(s);
         if (B = null)
            B ← Tr.CreateNonterminalNode(s);
      }
      Tr.CreateArc(A, B);
   }
}
```

Приведенный алгоритм разбора описания расширенной грамматики DSL имеет полиномиальную сложность.

В процессе разбора описания расширенной грамматики система проверяет корректность описания правил:

- соответствие описания правил грамматики синтаксису метаязыка;
- наличие определения для всех нетерминальных символов;
- отсутствие повторного определение нетерминального символа.

Рассмотрим пример построения псевдодерева разбора для языка "Записная книга", который имеет следующее описание:

```
Книга ::= {Запись ";"}
Запись ::= Фамилия Имя | Фамилия Имя Город
Фамилия ::= Идентификатор
Имя ::= Идентификатор
Город ::= Идентификатор
```

Начальным символом данной расширенной грамматики является нетерминальный символ "Книга".

Перед началом анализа псевдодерево разбора содержит описание предопределенных конструкций языка "Цифра", "Буква", "Идентификатор", которые задаются следующими правилами:

```
Идентификатор ::= Буква {Буква | Цифра}
Буква ::= "A" | "B" | ... | "Z" | "a" | "b" |
... | "z" | "А" | "Б" | ... | "Я" | "а" | "б" |
... | "я"
Цифра ::= "0" | "1" | ... | "9".
```

Если пользователь изменяет правило описания одной из предопределенных конструкций языка, то поддерево с ее описанием перестраивается в соответствии с новым определением.

Псевдодерево разбора для предопределенных конструкций приведено на рис. 1—3. В псевдодереве разбора прямоугольниками обозначены нетерминальные символы, окружностями — терминальные символы, ромбами — метасимволы, прямоугольником с двойной границей — начальный символ расширенной грамматики.

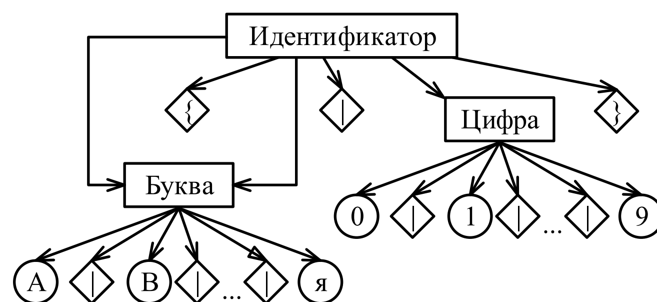

Рис. 1. Построение псевдодерева разбора. Этап I

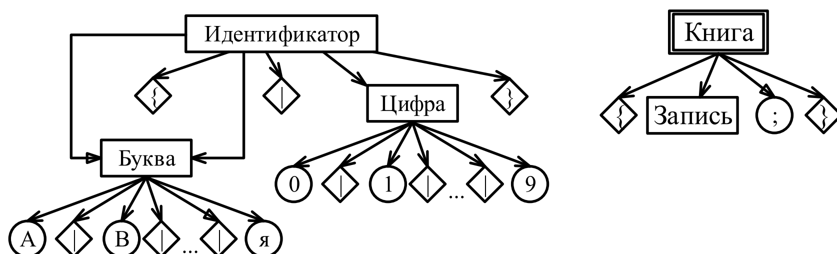

Рис. 2. Построение псевдодерева разбора. Этап II

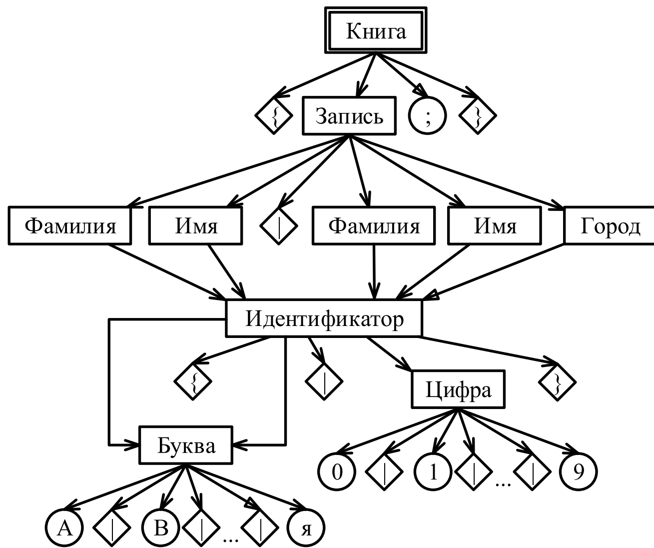

Рис. 3. Построение псевдодерева разбора. Этап III

После анализа первого правила в псевдодерево разбора будет добавлено описание нетерминального символа "Книга" (рис. 2).

Результирующее псевдодерево разбора, полученное после анализа всех правил, приведено на рис. 3.

4. Алгоритм проверки синтаксической корректности модели

После описания расширенной контекстно-свободной грамматики пользователь получает в распоряжение DSL, с помощью которого может разрабатывать программы. Программа считается корректной, если она удовлетворяет правилам описания расширенной грамматики языка.

Поскольку языковой инструментарий позволяет выполнять динамическое изменение описания языка, то проверка корректности программы будет выполняться в реальном времени, во время ее построения. Это обстоятельство заставляет отказаться от использования генераторов анализаторов, таких как ANTLR, Yacc, GNU Bison, Coco/R [13] и др.

В качестве основы для разработки алгоритма синтаксического разбора используется метод леворекурсивного спуска как наиболее простой в реализации. Поскольку пользователь сам определяет расширенную контекстно-свободную грамматику DSL, то нельзя гарантировать выполнение достаточного условия применимости метода рекурсивного спуска, поэтому для реализации используется метод рекурсивного спуска с возвратом.

Одним из ключевых понятий при реализации метода рекурсивного спуска является понятие множества стартовых символов нетерминального символа грамматики.

Множество терминальных символов, с которых может начинаться символ $A \in N$, назовем *множеством его стартовых символов*, обозначим его *starters*(A) и будем вычислять по следующим правилам:

1. Если правило грамматики имеет вид $A ::= t\omega$, где $t \in T$, $\omega \in (T \cup N)^*$, то $starters(A) = \{t\}$.

2. Если правило грамматики имеет вид $A ::= B\omega$, где $B \in N$, $\omega \in (T \cup N)^*$, то $starters(A) = starters(B)$.

3. Если правило грамматики имеет вид $A ::= \omega_1 \mid \omega_2 \mid ... \mid \omega_n$, где $\omega_i \in (T \cup N)^*$, $i = \overline{1,n}$, то

$$starters(A) = \bigcup_{i=1}^{n} starters(\omega_i).$$

4. Если правило грамматики имеет вид $A ::= \{\omega_1 \mid \omega_2 \mid ... \mid \omega_n\}\ \omega$ или $A ::= [\omega_1 \mid \omega_2 \mid ... \mid \omega_n]\ \omega$, где $\omega_i \in (T \cup N)^*, i = \overline{1,n}$, то $starters(A) =$

$$= \bigcup_{i=1}^{n} starters(\omega_i) \bigcup starters(\omega).$$

Описание алгоритма разбора последовательности лексем по псевдодереву разбора для расширенной грамматики *Gr* на псевдокоде имеет вид:

```
function NonTerminalNodeParse(Node A)
{
   i = 0;
   while (i < A. Children.Count)
   {
      a ← A. Children[i];
      while (l ∉ starters(a))
         a ← NextAlternative();
      if (l ≠ null)
      {
         NodeParse(A, i);
         ++i;
      }
      else
         Error(l, "Ошибка при описании
         конструкции" + a);
   }
}
```

Если правая часть правила расширенной грамматики содержит обязательную группу альтернатив, то в процессе анализа необходимо определить, в соответствии с какой альтернативой следует выполнять разбор. Для этого анализатор последовательно проверяет, принадлежит ли текущая лексема стартовым символам каждой альтернативы, если нет, то пропускает соответствующую данной альтернативе последовательность символов в описании конструкции с помощью функции *NextAlternative*.

Функция разбора терминальных, нетерминальных символов и последовательности альтернатив имеет следующий вид:

```
function NodeParse(Node A, ref int i)
{
   a ← A.Children[i];
   if (a ∈ Gr.T)
      if (a = l)
         l ← NextLexeme();
      else
      {
         Error(l, "Ошибка при описании
         конструкции" + a);
         return;
      }
   else
      if (a ∈ Gr.N)
         NonTerminalNodeParse(a);
      else
         if (a = '[')
            do
               a ← A.Children[+ + i];
               if (l ∈ starters(a))
               {
                  while (a ≠ '|' OR a ≠ ']') do
                  {
                     NodeParse(A, i);
                     a ← A.Children[+ + i];
                  }
                  while (a ≠ ']') do
                     a ← A.Children[+ + i];
               }
               else
                  while (a ≠ '|' OR a ≠ ']') do
                     a ← A.Children[+ + i];
            while (a ≠ ']');
         else
            if (a = '{')
            {
               StartAlternative ← a;
               do
                  continue ← false;
                  a ← A.Children[+ + i];
                  if (l ∈ starters(a))
                  {
                     continue ← true;
                     while (a ≠ '|' OR a ≠ '}') do
                     {
                        NodeParse(A, i);
                        a ← A.Children[+ + i];
                     }
                  }
                  else
                     while (a ≠ '|' OR a ≠ '}') do
                        a ← A.Children[+ + i];
                  if (a = '}' AND continue)
                     a ← StartAlternative;
               while (a ≠ '}');
            }
}
```

Данный алгоритм имеет полиномиальную сложность.

Поскольку структура языка заранее неизвестна, в отличие от языков с неизменным синтаксисом, а также не всегда удается однозначно сопоставить анализируемую лексему с конструкцией метамодели, важно сохранять историю разбора программы для наличия возможности возврата к последнему успешно разобранному символу. Возврат на предыдущие этапы анализа выполняется при сопоставлении последовательности лексем с правилами, содержащими группы альтернатив.

Если на некотором этапе разбора возникает ситуация, при которой анализатор не может сопоставить текущую лексему ни с одним ожидаемым символом языка, то необходимо выполнить возврат к предыдущему этапу анализа в соответствии со следующим алгоритмом:

1. Если правило, в соответствии с которым выполняется разбор, имеет вид

$$A ::= \alpha_1[\omega_1|\omega_2|...|\omega_n]\alpha_2,$$

где $\alpha_1, \alpha_2 \in (T \cup N)^+$, $\omega_i \in (T \cup N)^*$, $i = \overline{1, n}$, и на очередном шаге был выполнен разбор в соответствии с альтернативой ω_k, то следует вернуться к лексеме l, следующей сразу за последней корректно разобранной конструкцией α_1, и попытаться выполнить разбор в соответствии с описанием ранее нерассмотренных альтернатив $\{\omega_i\}$, $i = \overline{k+1, n}$. При этом возможны два варианта:

1.1. Анализируемая последовательность лексем не может быть разобрана в соответствии с описанием ни одной из альтернатив, это значит, что группа альтернатив была опущена в силу своей необязательности, необходимо выполнить переход к следующему за группой символу α_2 и провести разбор в соответствии с его описанием.

1.2. Лексема l соответствует стартовому символу одной из альтернатив, т. е. $l \in starters(\omega_i)$, $i = \overline{k+1, n}$, тогда следует выполнить разбор последовательности лексем в соответствии с описанием i-й альтернативы. Если в процессе анализа окажется, что очередная лексема не может быть разобрана в соответствии с описанием данной альтернативы, то следует вновь вернуться к лексеме l и найти другую подходящую альтернативу в рассматриваемой группе $\{\omega_j\}$, $j = \overline{i+1, n}$. Это продолжается до тех пор, пока все альтернативы не будут рассмотрены, либо анализ альтернативы не завершится успехом.

После успешного завершения анализа одной из альтернатив следует продолжить разбор

в соответствии со следующим за группой символом α_2. Если в какой-то момент лексема не может быть сопоставлена с текущим символом правила, то анализатор вновь возвращается к лексеме *l*, предпринимает попытку найти следующую альтернативу и выполнить разбор в соответствии с ее описанием.

2. Если правило, в соответствии с которым выполняется разбор, имеет вид

$$A ::= \alpha_1\{\omega_1|\omega_2|...|\omega_n\}\alpha_2,$$

где $\alpha_1, \alpha_2 \in (T \cup N)^+$, $\omega_i \in (T \cup N)^*$, $i = \overline{1,n}$, необходимо выполнить разбор лексемы в соответствии с пунктом 1 с той лишь разницей, что после успешного завершения анализа одной из альтернатив, необходимо вернуться к началу группы и попытаться выполнить разбор очередной лексемы в соответствии с описанием одной из альтернатив.

Если на каком-либо этапе анализа конструкций, следующих за группой альтернатив, невозможно сопоставить текущую лексему ни с одним из символов правила, то необходимо вернуться к лексеме, следующей за $m - 1$ успешно разобранной альтернативой группы, где m — общее число успешно разобранных альтернатив, и попытаться выполнить анализ в соответствии с описанием другой альтернативы, начиная с $m + 1$.

3. Если правило, в соответствии с которым выполняется разбор, имеет вид

$$A \to \omega_1|\omega_2|...|\omega_n,$$

где $\omega_i \in (T \cup N)^*$, $i = \overline{1,n}$, то оно интерпретируется как группа альтернатив, и его анализ выполняется в соответствии с пунктом 1.

Разбор считается успешным при одновременном завершении анализа текста исходной программы и обхода псевдодерева разбора.

Последняя анализируемая строка считается ошибочной, если

- рассмотрены все возможные варианты разбора текущей строки, и ни один из них не позволяет корректно проанализировать программу;
- завершен обход псевдодерева разбора, но программа содержит еще неразобранные строки;
- завершен разбор всех строк исходной программы, но обход псевдодерева разбора не был завершен.

Описанный алгоритм возврата имеет экспоненциальную сложность.

Заключение

Разработанные алгоритмы синтаксического разбора позволяют реализовать в языковом инструментарии возможности многоуровневого моделирования и динамического изменения описания разрабатываемых DSL.

В качестве промежуточного представления для хранения результатов анализа расширенной грамматики DSL используется псевдодерево разбора, что позволяет повысить эффективность поиска конструкций языка на этапе проверки синтаксической корректности программы.

Функции подсветки синтаксиса языка и автодополнения кода текстового редактора DSL также используют в своей работе псевдодерево разбора. Эти функции значительно упрощают процесс построения и чтения модели, а также сокращают число ошибок в ней. Подсветка синтаксиса выполняется на основе списков нетерминальных символов и метасимволов. Составление списков проводится на этапе построения псевдодерева разбора для того, чтобы в дальнейшем каждый раз не выполнять поиск по псевдодереву. Для автодополнения кода необходимо для каждого нетерминального символа сформировать список его стартовых символов.

Практическая значимость работы заключается в том, что рассмотренные алгоритмы являются основой для реализации компонентов описания и проверки синтаксиса языкового инструментария, позволяющего создавать текстовые динамически настраиваемые предметно-ориентированные языки.

В дальнейшем предполагается разработать алгоритмы трансформации моделей, созданных с помощью DSL, в код на целевом языке программирования.

Список литературы

1. **1С: Предприятие 8.** Встроенный язык. URL: http://v8.1c.ru/ overview/Term_000000020.htm (дата обращения: 21.11.2017).

2. **Pegden C. D.** The Evolution of Simulation Languages // Advances in Modeling and Simulation. Simulation Foundations, Methods and Applications. Cham: Springer, 2017. P. 81—96.

3. **Buch J. P., Laursen J. S., Soensen L. C., Ellekilde L.-P., Kraft D., Schultz U. P., Petersen H. G.** Applying Simulation and a Domain-Specific Language for an Adaptive Action Library // Simulation, Modeling, and Programming for Autonomous Robots. SIMPAR 2014. Lecture Notes in Computer Science. Cham: Springer. 2014. Vol. 8810. P. 86—97.

4. **Дерябин Н. Б., Жданов Д. Д., Соколов В. Г.** Внедрение языка сценариев в программные комплексы оптического моделирования // Программирование. 2017. № 1. С. 40—53.

5. **Сухов А. О., Медведева Е. Ю.** Подход к разработке языкового инструментария для создания текстовых пред-

метно-ориентированных языков // Информационные технологии. 2018. Т. 24, № 1. С. 10—16.

6. **Haase A.** Introduction to OpenArchitectureWare 4.1.2 // Model-Driven Development Tool Implementers Forum 2007. URL: https://pdfs.semanticscholar.org/a6fa/d330ed634810f3f002658443ab361eb3c423.pdf (дата обращения: 21.11.2017).

7. **Voelter M.** Language and IDE Modularization and Composition with MPS // Generative and Transformational Techniques in Software Engineering IV. Lecture Notes in Computer Science. Berlin: Springer. 2013. Vol. 7680. P. 383—430.

8. **Charles Ph., Fuhrer R. M., Sutton S. M.** IMP: a Meta-Tooling Platform for Creating Language-Specific IDEs in Eclipse // Proceedings of the Twenty-Second IEEE/ACM International Conference on Automated Software Engineering. NY. 2007. P. 485—488.

9. **Holldobler K., Roth A., Rumpe B., Wortmann A.** Advances in Modeling Language Engineering // Model and Data Engineering. MEDI 2017. Lecture Notes in Computer Science. Cham: Springer. 2017. Vol. 10563. P. 3—17.

10. **Kats L. C. L., Visser E.** The Spoofax Language Workbench: Rules for Declarative Specification of Languages and IDEs // Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications. NY. 2010. P. 444—463.

11. **Nijholt A.** Context-Free Grammars. Covers, Normal Forms, and Parsing. Berlin: Springer, 1980. 253 p.

12. **Mogensen T.** Introduction to Compiler Design. Berlin: Springer, 2017. 258 p.

13. **Koranne S.** Compiler Construction // Handbook of Open Source Tools. Boston: Springer. 2010. P. 241—284.

L. N. Lyadova, Associate Professor, e-mail: LLyadova@hse.ru,
A. O. Sukhov, Associate Professor, e-mail: ASuhov@hse.ru,
E. Yu. Medvedeva, Master Student, e-mail: medvedevaeyu@mail.ru,
National Research University Higher School of Economics, Perm, 614070, Russian Federation

Syntactic Parsing Algorithms for Textual Domain-Specific Languages

Over time an increasing number of software packages have provided users with domain-specific languages designed to customize the system in accordance with changing operating conditions and user needs. However, it is necessary to create language workbenches for such languages. The authors suggested an approach to developing toolkits for creating textual dynamically customizable domain-specific languages that allow eliminating the limitations of most existing language workbenches. The purpose of the study is to develop algorithms for parsing language grammar and checking syntactic correctness of the programs. These algorithms are necessary for implementing a language workbench.

As a means of representing the transitional results of the domain-specific language grammar analysis a parsing pseudo tree is used, which makes it possible to improve the efficiency of language constructions searching at the stage of checking the program's syntactic correctness.

The parsing algorithm is based on the left-recursive descent method with backtracking. Despite the exponential complexity, the algorithm makes it possible to provide the possibility of dynamically changing the language description.

Functions of language syntax highlighting and code completing also use the parsing pseudo tree in their work. These functions simplify the process of model building and reading and reduce the number of errors in it.

Keywords: *grammar analysis, syntax checking, domain-specific languages, textual languages, language workbench, left-recursive descent parser, formal grammars, parse tree, metalanguage, model-oriented approach*

DOI: 10.17587/it.24.648-656

References

1. **1S:Predprijatie 8.** Vstroennyj jazyk, available at: http://v8.1c.ru/overview/Term_000000020.htm (Access at: 21.11.2017) (in Russian).

2. **Pegden C. D.** The Evolution of Simulation Languages, *Advances in Modeling and Simulation. Simulation Foundations, Methods and Applications*, Cham, Springer, 2017, pp. 81—96.

3. **Buch J. P., Laursen J. S., Soensen L. C., Ellekilde L.-P., Kraft D., Schultz U. P., Petersen H. G.** Applying Simulation and a Domain-Specific Language for an Adaptive Action Library, *Simulation, Modeling, and Programming for Autonomous Robots. SIMPAR 2014. Lecture Notes in Computer Science*, Cham, Springer, 2014, vol. 8810, pp. 86—97.

4. **Derjabin N. B., Zhdanov D. D., Sokolov V. G.** Vnedrenie jazyka scenariev v programmnye kompleksy opticheskogo modelirovanija (Embedding the Script Language into Optical Simulation Software), *Programmirovanie*, 2017, no. 1, pp. 40—53 (in Russian).

5. **Suhov A. O., Medvedeva E. Ju.** Podhod k razrabotke jazykovogo instrumentarija dlja sozdanija tekstovyh predmetno-orientirovannyh jazykov (Approach to Language Workbench Development for Textual Domain-Specific Languages Creation), *Informacionnye Tehnologii*, 2018, vol. 24, no. 1, pp. 10—16 (in Russian).

6. **Haase A.** Introduction to OpenArchitectureWare 4.1.2, Model-Driven Development Tool Implementers Forum 2007, available at: https://pdfs.semanticscholar.org/a6fa/d330ed634810f3f002658443ab361eb3c423.pdf (Access at: 21.11.2017).

7. **Voelter M.** Language and IDE Modularization and Composition with MPS, *Generative and Transformational Techniques in Software Engineering IV. Lecture Notes in Computer Science*, Berlin, Springer, 2013, vol. 7680, pp. 383—430.

8. **Charles Ph., Fuhrer R. M., Sutton S. M.** IMP: a Meta-Tooling Platform for Creating Language-Specific IDEs in Eclipse, *Proceedings of the Twenty-Second IEEE/ACM International Conference on Automated Software Engineering*, NY, 2007, pp. 485—488.

9. **Holldobler K., Roth A., Rumpe B., Wortmann A.** Advances in Modeling Language Engineering, *Model and Data Engineering. MEDI 2017. Lecture Notes in Computer Science*, Cham, Springer, 2017, vol. 10563, pp. 3—17.

10. **Kats L. C. L., Visser E.** The Spoofax Language Workbench: Rules for Declarative Specification of Languages and IDEs, *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, NY, 2010, pp. 444—463.

11. **Nijholt A.** Context-Free Grammars. Covers, Normal Forms, and Parsing, Berlin, Springer, 1980, 253 p.

12. **Mogensen T.** Introduction to Compiler Design, Berlin, Springer, 2017, 258 p.

13. **Koranne S.** Compiler Construction, *Handbook of Open Source Tools*, Boston, Springer, 2010, pp. 241—284.