Грамматики. Парсеры

Роль формальных грамматик в науке и в IT-сфере значительна. Этот формализм позволяет описывать языки (как множества слов) естественного или искусственного происхождения на основе синтаксических правил, т.е. фактически, на основе внешних признаков. Процесс синтаксического разбора (парсинг, eng.: parsing), другими словами, сопоставление последовательности символов (лексем, токенов, слов) формальным правилам грамматики в IT-технологиях обычно осуществляют особые алгоритмы, называемые *парсерами*, на основе формальных описаний, называемых *грамматиками*. Для создания компиляторов наиболее востребованными являются так называемые КС-грамматики.

wiki: Формальная грамматика

Грамматики с точки зрения математики

Ранее мы уже разбирали частный пример разработки простого парсера. Однако, более общая проблема разработки парсеров для *Контекстно-Свободных Грамматик* (КС-грамматик, cfg (context-free grammar)) требует и более изощрённых методов. Дадим определение КС-грамматик.

Определение 1. Пусть даны V_T — алфавит (множество терминальных символов), V_N — множество нетерминальных (служебных) символов, причем $V_T \cap V_N = \varnothing$ (т.е. пересечение данных множеств пусто), P — конечное множество правил (продукций), каждое из которых имеет вид $A \to a$, где $A \in V_N$, $a \in (V_T \cup V_N)^*$ (т.е. a является словом, состоящим из символов смешанного алфавита V_T и V_N, или даже пустым словом, обычно обозначаемым как ε), $S \in V_N$ — стартовый символ. Тогда мы говорим, что задана грамматика $G = \langle V_T, V_N, P, S \rangle$.

Более точно, мы даже задали *контекстно-свободную* грамматику, или КС-грамматику. КС-грамматики занимают особую роль в проектировании языков программирования и компиляторов. Как правило, современные языки описываются некоторой КС-грамматикой. Есть и другие виды грамматик: более общие *контекстно-зависимые* и более специализированные *регулярные*. При этом, регулярные являются подмножеством контекстно-свободных грамматик, а контекстно-свободные являются подмножеством контекстно-зависимых грамматик.

Определение 2. Пусть $(A \to \beta) \in P$ — правило, γ, δ — любые цепочки символов (слова) из множества $(V_T \cup V_N)^*$. Тогда говорят, что «из $\gamma A \delta$ непосредственно выводится $\gamma\beta\delta$ в грамматике G (при помощи данного правила)» и обозначают: $\gamma A \delta \underset{G}{\Rightarrow} \gamma\beta\delta$ (если известно, о какой G идёт речь, это обозначение под стрелкой может опускаться).

Определение 3. Пусть $\alpha_1, \alpha_2, \ldots, \alpha_m$ — слова из множества $(V_T \cup V_N)^*$ и $\alpha_1 \Rightarrow \alpha_2, \ldots, \alpha_{m-1} \Rightarrow \alpha_m$. Тогда мы пишем: $\alpha_1 \underset{*}{\Rightarrow} \alpha_m$ и говорим, что «из α_1 выводится α_m в грамматике G».

Определение 4. Язык, порождаемый грамматикой G, определим как

$$L(G) = \{w \mid w \in V_T^*,\ S \underset{*}{\Rightarrow} w\}.$$

Другими словами, язык есть множество терминальных цепочек (слов), выводимых из начального нетерминала (стартового символа) грамматики. И таким образом, первой задачей проектируемого парсера будет синтаксический анализ принадлежности слов данной грамматике, т.е. ответ на вопрос: возможно ли вывести то или иное слово из стартового символа с помощью данных грамматических правил?

Упражнение 1. Составить грамматику для языка, где $V_T = \{a, b, c\}$ и $L = \{wcw^T \mid w \in \{a, b\}^*\}$.

Упражнение 2. Составить грамматику для языка, где $V_T = \{a, b\}$ и $L = \{ww^T \mid w \in V_T^*\}$.

Упражнение 3. Составить грамматику для языка, где $V_T = \{a, b\}$ и $L = \{w \mid w = w^T, w \in V_T^*\}$.

Упражнение 4. Составить грамматику для языка сбалансированных скобок (терминалами будут только скобки). Например: `()()`, `(())`.

Решением первого упражнения будет следующая грамматика: $V_N = \{S\}$, $S = S$ и P определено следующим образом: $S \to c$; $S \to aSa$; $S \to bSb$. Часто для удобства пишут тоже самое более компактно: $S \to c \mid aSa \mid bSb$.

Пример. Вот как будет выглядеть вывод слова $aabcbaa$ в этой грамматике:

$$S \Rightarrow aSa \Rightarrow aaSaa \Rightarrow aabSbaa \Rightarrow aabcbaa$$

(на практике вывод делается справа налево)

Определение 5 [1. С. 44; 2. С. 82–83; 3. С. 197–198]. Деревом вывода D (или деревом синтаксического разбора) называется помеченное упорядоченное дерево в КС-грамматике, если

- Каждая вершина (узел) имеет метку — символ из алфавита $(V_T \cup V_N)$.
- Корень дерева D помечен символом S.
- Если узел имеет по крайней мере одно потомка, то метка этого узла должна быть помечена нетерминалом. Листья этого дерева должны быть помечены терминалами.
- Если $n_1, \ldots, n_k$ — прямые потомки узла n, перечисленные слева направо, с метками $A_1, \ldots, A_k$ соответственно, а метка узла n есть A, то $A \to A_1 \ldots A_k$ является продукцией в данной грамматике $G = \langle V_T, V_N, P, S\rangle$. При этом, если лист помечен символом ε, то он должен быть единственным дочерним узлом у вершины n, а правило $A \to \varepsilon$ должно быть в списке продукций.

«*Помеченное*» означает наличие метки у каждой вершины в виде символа из алфавита $(V_T \cup V_N)$. «*Упорядоченное*» означает, что все дочерние узлы каждого узла упорядочены (т.е., их можно занумеровать как конечную последовательность по какому-то принципу). [4. С.1221]

Упорядоченное дерево

What is an ordered tree?

srackoverflow: What is an Ordered Tree

Ordered Trees

Такое дерево (как в определении-5) называют ещё «*конкретным синтаксическим деревом*». Есть ещё и «*абстрактное синтаксическое дерево*»:

Абстрактное синтаксическое дерево отличается от дерева разбора тем, что в нём отсутствуют узлы и рёбра для тех синтаксических правил, которые не влияют на семантику программы. Классическим примером такого отсутствия являются группирующие скобки, так как в АСД группировка операндов явно задаётся структурой дерева.

wikipedia: Абстрактное синтаксическое дерево

Приведём более строгое, математическое определение в рекурсивном стиле в терминах поддеревьев, правда немного отличающееся. *Выдачей* (или *выходом*, eng: yield) дерева в определении ниже будем называть конкатенацию меток листьев, взятых слева направо.

Определение 6 [5. С. 123–124].

1. Одиночная вершина с меткой $a \in V_T$ образует дерево синтаксического разбора (являясь сразу и корнем, и листом в этом дереве). Выдачей такого дерева будет одиночный символ a.

2. Если $A \to \varepsilon$ является правилом в списке продукций P, то дерево вида, изображённое на рис. 1:

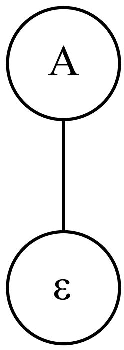

Рис. 1: Дерево с листом, содержащим пустой символ ε

   тоже будет деревом синтаксического разбора; его корнем будет вершина, помеченная A, а его единственным листом будет вершина, помеченная ε. Выдачей такого дерева будет пустой символ ε.

3. Пусть $n \geq 1, T_1, \dots T_n$ — уже деревья синтаксического разбора с корнями $A_1, \dots A_n$ и с выдачей $y_1, \dots, y_n$, соответственно на рис. 2:

   и пусть $A \to A_1 \dots A_n$ — правило в списке продукций P, тогда дерево T на рис. 3:

   также будет деревом синтаксического разбора. Его корнем будет новый узел, помеченный A, и с листьями, его листьями будут листья составляющих его поддеревьев $T_1, \dots T_n$, и его выходом будет выражение $y_1 \dots y_n$.

Приведённое рекурсивное определение будет удобно в наших практических реализациях парсеров далее, но чтобы это определение больше соответствовало ранее введённому определению 5, стоит изменить первый пункт следующим образом:

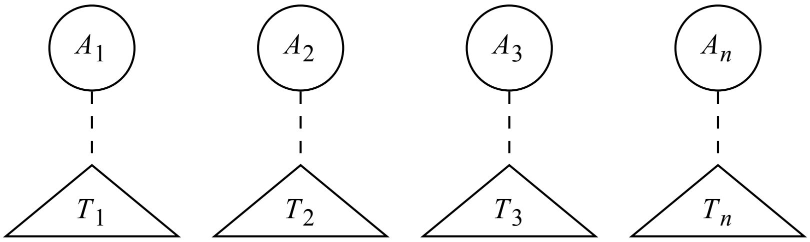

Рис. 2: Деревья $T_1, \dots T_n$ с корнями $A_1, \dots A_n$

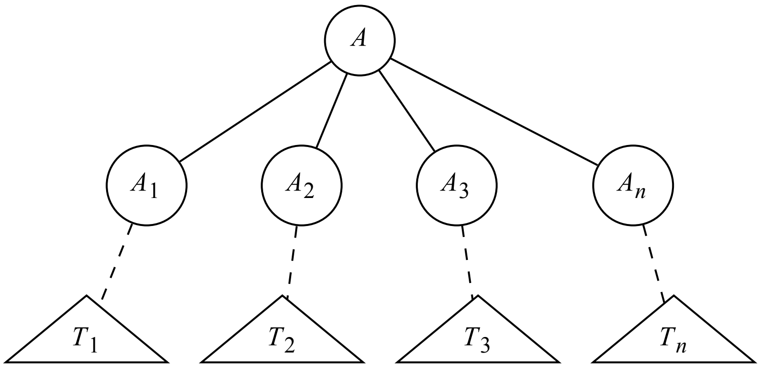

Рис. 3: Рекурсивно полученное дерево T

1. Одиночная вершина с меткой $a \in V_T$ не образует дерево синтаксического разбора. Если $a_1, \dots, a_n \in V_T$ и $A \to a_1 \dots a_k$ является правилом в списке продукций P, то дерево вида на рис. 4:

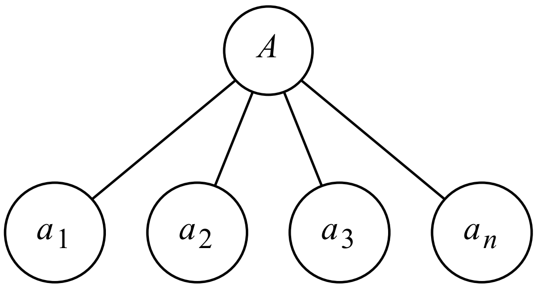

Рис. 4: Дерево с листами, содержащими терминальные символы

будет деревом синтаксического разбора; его корнем будет вершина, помеченная A, а его листьями будут вершины, помеченные $a_1 \dots a_n$. Выдачей такого дерева будет строка символов $a_1 \dots a_n$.

Отметим важность дерева синтаксического разбора. В том или ином виде оно лежит в основе таких технологий как wikipedia: DOM (при обработке HTML из JavaScript), или wikipedia: WebAssembly.

Контекстно-свободные грамматики, вывод, дерево разбора

Пример. Дерево вывода для выражения `(()())` в грамматике сбалансированных скобок:

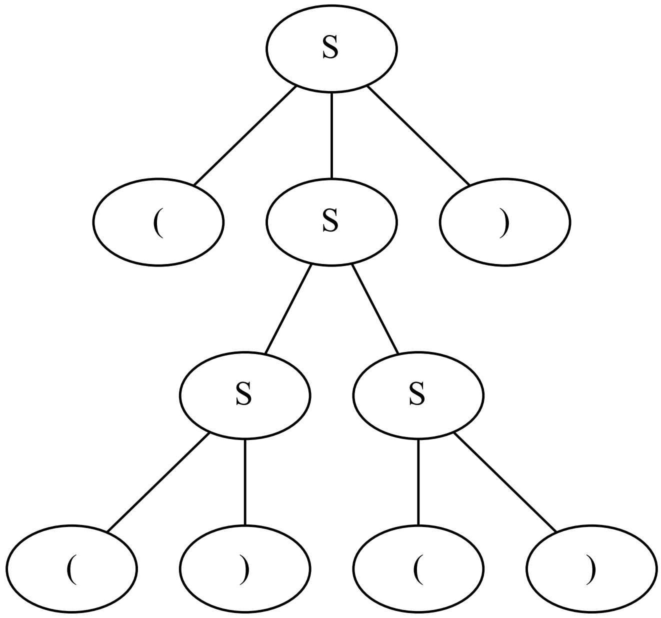

Рис. 5: Дерево вывода для выражения `(()())`

Отметим, что не все языки могут иметь КС-грамматику (даже среди языков программирования) и не все языки, задаваемые КС-грамматикой, будут иметь простые парсеры.

Is it possible to make a grammar LL(1) which recognizes palindroms?

БНФ и РБНФ

Для формального описания КС-грамматик существует ряд стандартов, наиболее известные из которых БНФ и РБНФ.

Форма Бэкуса–Наура

Собственно, из всего описания нам интересен только способ записи правил (продукций) в этом формализме:

```
<определяемый символ> ::= <посл.1> | <посл.2> | . . . | <посл.n>
```

что в точности соответствует набору из n продукций вида:

$$A \to a_1, \quad A \to a_2, \quad \ldots, \quad A \to a_n,$$

где A — нетерминал, а a_i — слова из символов из смешанного алфавита терминальных и нетерминальных симвлов.

wiki: Форма Бэкуса–Наура

Расширенная форма Бэкуса–Наура

Предложена Никлаусом Виртом. Является расширенной переработкой форм Бэкуса–Наура.

Правила записываются в форме:

```
идентификатор = выражение;
```

где *идентификатор* — имя нетерминального символа, а *выражение* — соответствующая правилам РБНФ комбинация терминальных и нетерминальных символов и специальных знаков (на конце `;` или иногда `.` — специальный символ, указывающий на завершение правила).

Краткая сводка по конструированию выражений:

- `=` — описание (или `::=`);
- `'…'` или `"…"` — строка терминальных символов;
- `,` — конкатенация;
- `|` — выбор;
- `[…]` — условное вхождение;
- `{…}` — повторение (ноль или более);
- `(…)` — группировка.

Пример. Опишем простейшей грамматикой понятие идентификатора, состоящего из заданных букв (`x`, `y` или `z`) или цифр, и начинающего на букву. Входным алфавитом можно считать ASCII или даже весь Юникод.

Множеством нетерминалов будет $\{L, D, I\}$, стартовым символом I. Правила

$$L \to x|y|z, \quad D \to 0|1|2|3|4|5|6|7|8|9$$

описывают понятие допустимой буквы и цифры. Правила

$$I \to L, \quad I \to ID, \quad I \to IL$$

рекурсивно описывают понятие идентификатора.

То же самое в БНФ:

```
<letterxyz> ::= x|y|z
<digit> ::= 0|1|2|3|4|5|6|7|8|9
<id> ::= <letterxyz>
<id> ::= <id><letterxyz>
<id> ::= <id><digit>
```

Немного в ином виде, это же самое в РБНФ:

```
letterxyz = 'x'|'y'|'z';
digit = '0'|'1'|'2'|'3'|'4'|'5'|'6'|'7'|'8'|'9';
id = letterxyz, {letterxyz|digit};
```

В конце лекции мы вернёмся к этому примеру.

Расширенный Бекуса-Наура формализм (РБНФ)

en.wiki: Extended Backus–Naur form

David A. Wheeler. Don't Use ISO/IEC 14977 Extended Backus-Naur Form (EBNF)

V. Zaitsev. BNF WAS HERE: What Have We Done About the Unnecessary Diversity of Notation for Syntactic Definitions

EBNF notation from the W3C

Схема восходящего парсера (LR)

Опишем принцип действия простого восходящего парсера для валидации входной строки символов внешнего алфавита V_T некоторой КС-грамматики G.

Рассматриваем список входных символов (`Char`), магазин (обрабатываемых элементов), список правил (продукций или свёрток) в виде свёрточных функций [2. С. 301].

1. Из списка входных элементов считываем очередной символ. Сразу проверяем на вхождение его в требуемый алфавит. Символ помещаем в магазин.
2. Пытаемся применить какие-либо продукции к подспискам символов в магазине (рассматривая их с реверсом, так как в магазин помещали в обратном порядке). Если можем применить какую-либо продукцию, делаем свёртку. Потом вновь повторяем этот пункт, до тех пор пока есть возможность.
3. Как только возможность применить продукции закончилась, переходим вновь к п.1.
4. Вскоре список входных символов станет пустым. Тогда проверяем магазин. Пробуем вновь, как в п.2 применить продукции, если получается — проводим свёртки. Если уже не можем, то проверяем остаток в магазине.
5. Если в магазине после всех свёрток остался лишь один стартовый нетерминал нашей грамматики S, то считаем введённую стоку верифицированной. Если там остались несколько любых символов, то считаем введённую строку неверифицированной.

Отметим, что пустым магазин может остаться только в вырожденном случае, когда на вход был подан пустой текст. При свёртках пустых строк получиться не может.

Восходящий парсер для грамматики сбалансированных скобок

Вернёмся ещё раз к теме простой грамматики сбалансированных скобок. Рассмотрим упрощённую версию, когда и терминалы, и нетерминалы будут представлены символами из `Char`.

терминалы: ()

нетерминал: S

продукционные правила: $S \to () \mid (S) \mid SS$

Для дальнейшего описания нам потребуется техника монад.

```
import Control.Monad
import Data.Maybe
```

Проверим наличие только скобок во входной строке.

```
checksymb :: Char -> Char
checksymb  c | (c `elem` "()")     =  c
             | otherwise =
   error "There is an illegal symbol!"
```

Опишем продукционные правила как функцию свёртки:

```
prodrule :: String -> Maybe Char
prodrule "()"  = Just 'S'
prodrule "(S)" = Just 'S'
prodrule "SS"  = Just 'S'
prodrule _     = Nothing
```

Воспользуемся общей схемой «восходящего анализатора».

0. На вход для анализа получаем строку символов. В нашем представлении это будет список `[Char]`, его символизирует переменная `ss`. Проектируемый парсер будет работать с парой типа `(String,String)` у которой первый элемент `ss` — входная (точнее — укорачиваемая текущая строка), а второй элемент — строка-магазин `mss` обрабатываемых на каждом шагу нетерминалов.

1. Отщепляем первый элемент `s` списка. И, прежде чем поместить его в магазин `mss`, проверяем его на корректность.

```
transfer :: (String,String) -> (String,String)
transfer ((s:ss),mss) = (ss, ((checksymb s):mss))
transfer ([],m)       = ([], m)
```

Поместим его в магазин.

2. Пара «технических» функций. Функция `justadd2list` передаст результат удачно выполненной продукции обратно в магазин (или выставит флаг `Nothing` в случае неудачи) и обернёт его монадой `Maybe`. Функция `mmplus` действует немного подобно своей тёзке `mplus` из `MonadPlus`: соединяя два аргумента — пары (входная строка, магазин), причём в первом аргументе-паре магазин обёрнут монадой `Maybe`. В соответствии с обёрткой, если она `Nothing`, то возвращаем второй аргумент, если `(Just mss)`, то снимаем обёртку и возвращаем значение первого аргумента.

```
justadd2list :: Maybe a -> [a] -> Maybe [a]
(Just x) `justadd2list` mss   = Just (x:mss)
Nothing  `justadd2list` _     = Nothing

mmplus :: (t, Maybe t1) -> (t,t1) -> (t,t1)
(_,  Nothing)  `mmplus` t  = t
(ss, Just mss) `mmplus` _  = (ss,mss)
```

3. Если в магазине есть два или три символа, извлечём их из него и попытаемся применить продукционное правило. Обратим внимание, что применяя продукционное правило, меняем порядок символов, извлечённых из магазина — располагая их

именно так как они шли в исходной строке (учитывая свёртку). Полученный нетерминал `S` вернём в магазин `mss` (но потом и без **`Just`**).

```
use2symbols :: [Char] -> Maybe [Char]
use2symbols (m1:m2:mss)    =
  prodrule [m2,m1] `justadd2list` mss
use2symbols _              = Nothing

use3symbols :: [Char] -> Maybe [Char]
use3symbols (m1:m2:m3:mss) =
  prodrule [m3,m2,m1] `justadd2list` mss
use3symbols _              = Nothing
```

Если продукцию применить не можем (выставляем флаг **`Nothing`**), переходим к следующему шагу.

4. Если в шаге 3 мы не смогли применить продукции, переходим к шагу 5. Если на шаге 3 мы смогли применить продукцию, то вновь возвращаемся к началу шага 3.

```
step :: (String,String) -> (String,String)
step ([],mss) | (isNothing res) = ([],mss)
              | otherwise       = step ([],(fromJust res))
  where res :: Maybe String
        res = (use2symbols mss `mplus` use3symbols mss)
step (ss,mss) = let
  t = (ss,(use2symbols mss `mplus` use3symbols mss))
        `mmplus`
      (transfer (ss,mss)) in step t
```

5. Кончились ли элементы во входном списке? Если да, то смотрим результаты парсинга: получился единственный нетерминал `S` — значит, все распозналось, если осталось что-то ещё, то не распозналось.. Если элементы во входном списке еще остались, то переходим к шагу 1.

Ниже применяем полученные функции для создания парсера `parse`.

```
parse teststr = if (mss == "S")
  then teststr ++ " is valid"
  else teststr ++ " isn't valid"
     where (_,mss) = step (teststr,[])
```

Проверяем результат:

Для функции `step`:

```
> step ("()(())", "")
("","S")
> step ("()(", "")
("","(S")
```

и для функции `parse`:

```
> parse "()(())"
"()(())" is valid
> parse "()("
"()(" isn't valid
```

Парсинг булевых формул

Опишем восходящий парсер для грамматики:

терминалы: 0, 1, &, +, !, (,)

нетерминал: S

продукционные правила:

$$
\begin{aligned}
&S \to 0 \mid 1 \\
&S \to !S \\
&S \to (S\&S) \mid (S{+}S)
\end{aligned}
$$

Как и прежде, используем технику монад:

```
import Control.Monad
import Data.Maybe
```

В этот раз парсер будет строить дерево синтаксического разбора:

```
data PrsdTr = Leaf Char | Node1 Char PrsdTr |
  Node2 Char PrsdTr PrsdTr
  deriving (Show,Read,Eq,Ord)
```

Отметим, что мы будем строить не «конкретное синтаксическое дерево», а упрощённую форму абстрактного, размещая в нодах вместо нетерминалов сами операции. Например, для формулы $(!0 + (1\&0))$ вместо конкретного дерева мы построим абстрактное, см. рис. 6 и 7.

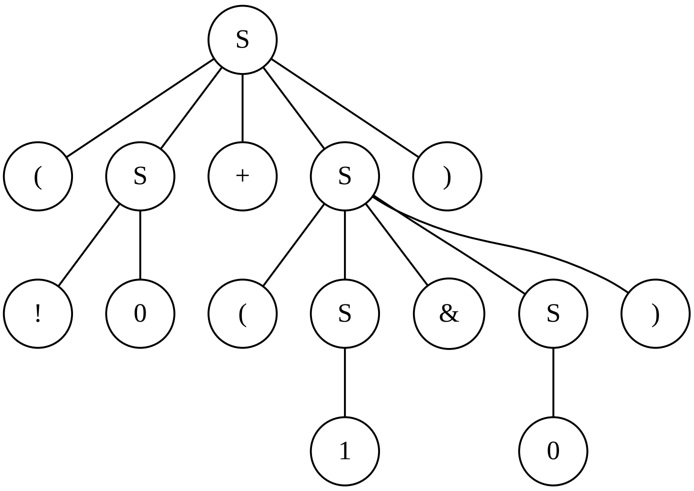

Рис. 6: Конкретное дерево

Введем ограничение на тип используемых элементов:

```
listVT :: [Char]
listVT = "01!&+()"
```

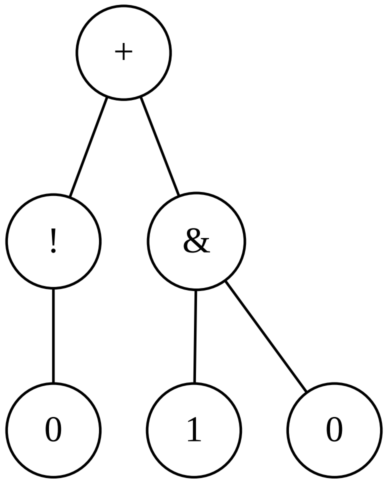

Рис. 7: Абстрактное дерево

(в будущем, лучшим решением было бы завести особый тип).

Проверим наличие разрешенных символов во входной строке.

```
checksymb :: Char -> Char
checksymb  c | (c `elem` listVT)     =  c
             | otherwise = error "There is an illegal symbol!"
```

Трансфер символов из входной строки совместим с построением листа дерева и помещением его в магазин:

```
transfer :: ([Char], [PrsdTr]) -> ([Char], [PrsdTr])
transfer ((s:ss),mss) = (ss, ((Leaf $ checksymb s):mss))
transfer ([],m)       = ([], m)
```

Опишем продукционные правила как функции свёртки (из списка деревьев в «возможно» дерево):

```
prodrule :: [PrsdTr] -> Maybe PrsdTr
prodrule [(Leaf '!'), s]  = Just $ Node1 '!' s
prodrule [(Leaf '('),s1,(Leaf '&'),s2,(Leaf ')')] =
   Just $ Node2 '&' s1 s2
prodrule [(Leaf '('),s1,(Leaf '+'),s2,(Leaf ')')] =
   Just $ Node2 '+' s1 s2
prodrule _                                        =
   Nothing
```

Вновь определим технические функции:

```
justadd2list :: Maybe a -> [a] -> Maybe [a]
(Just x) `justadd2list` mss   = Just (x:mss)
Nothing  `justadd2list` _     = Nothing

mmplus :: (a, Maybe b) -> (a, b) -> (a, b)
(_,  Nothing)  `mmplus` t  = t
(ss, Just mss) `mmplus` _  = (ss,mss)
```

Далее. Если в магазине есть 2 или 5 помещённых туда дерева, извлечем их из него и попытаемся применить продукционное правило. Обратим внимание, что применяя продукционное правило, меняем порядок символов, извлечённых из магазина — располагая их именно так как они шли в исходной строке (учитывая свёртку). Полученное дерево вернём в магазин mss с тегом **Just** на всём магазине (или **Nothing** при неудаче).

```
use2symbols :: [PrsdTr] -> Maybe [PrsdTr]
use2symbols (m1:m2:mss)    = prodrule [m2,m1] `justadd2list` mss
use2symbols _              = Nothing

use5symbols :: [PrsdTr] -> Maybe [PrsdTr]
use5symbols (m1:m2:m3:m4:m5:mss) =
        prodrule [m5,m4,m3,m2,m1] `justadd2list` mss
use5symbols _                    = Nothing
```

Вот теперь опишем наш цикл step. Если входной список символов пуст и применение правил к списку, оставшемуся в магазине даёт **Nothing**, то возвращаем ([],mss). Если не **Nothing**, то пытаемся ещё раз применить наш цикл.

Если входной список ss не пуст, то пытаемся применить к нему правила в 2 и в 5 обрабатываемых деревьев — через use2symbols илиuse5symbols (сработает первое из возможных), и потом передаём с помощью функции mmplus, которая заодно снимет тег **Just** у магазина, пару «(входной список, обработанный магазин)» на новую итерацию в функцию step. Или, если в результате работы use2symbols илиuse5symbols получится **Nothing** — передаём результат свёртки в функцию transfer.

```
step :: ([Char], [PrsdTr]) -> ([a], [PrsdTr])
step ([],mss) | (isNothing res) = ([],mss)
              | otherwise       = step ([],(fromJust res))
                  where res :: Maybe [PrsdTr]
                        res = (use2symbols mss `mplus` use5symbols mss)
step (ss,mss) =
  let
    t = (ss,(use2symbols mss `mplus` use5symbols mss))
          `mmplus` (transfer (ss,mss))
  in step t
```

В целом, для работы уже всё задано.

```
*Main> step ("(1&0)",[])
([],[Node2 '&' (Leaf '1') (Leaf '0')])
*Main> step ("(1&&0)",[])
([],[Leaf ')',Leaf '0',Leaf '&',Leaf '&',Leaf '1',Leaf '('])
*Main> step ("(10)",[])
([],[Leaf ')',Leaf '0',Leaf '1',Leaf '('])
*Main> step ("10",[])
([],[Leaf '0',Leaf '1'])
*Main> step ("&",[])
([],[Leaf '&'])
*Main> step ("!0",[])
([],[Node1 '!' (Leaf '0')])
```

Видим, что если выражение было валидно, то на выходе будет список из одного дерева. Если выражение было не валидно, то получим список, возможно и из одного элемента: `[Leaf '&']`. Таким образом, эту ситуацию надо отсекать, например, удаляя плохие варианты (их немного) в конце работы. Кстати, одна из причин нашей проблемы в том, что мы решили строить упрощённую форму AST — если бы в узловых нодах вместо символов операций мы бы хранили S, то для правильно построенных деревьев такого бы не возникло. Другая причина такого поведения в том, что мы взяли для простоты реализации определение 6 из теоретического начала лекции, в котором отдельные вершины-листья, содержащие какие-то терминалы, могут считаться деревьями разбора.

Для удобства работы (с заведомо корректными выражениями) заведём функцию, которая будет извлекать единственный элемент из списка:

```
fromLst :: [a] -> a
fromLst [x] = x
fromLst  _  = error "List isn't single!"
```

Работа теперь будет выглядеть примерно так:

```
*Main> fromLst $  snd $ step ("(1&(1+0))",[])
Node2 '&' (Leaf '1') (Node2 '+' (Leaf '1') (Leaf '0'))
```

Обработка синтаксического дерева

Синтаксическое дерево интересно и как самостоятельное представление структуры данных, оно довольно удобно для дальнейшей обработки.

Для начала, заведем функцию `eval`, которая будет вычислять значение исходного выражения с помощью построенного дерева:

```
eval :: PrsdTr -> Bool
eval (Leaf '0') = False
eval (Leaf '1') = True
eval (Node1 '!' tr) = not $ eval tr
eval (Node2 '&' tr1 tr2) =
  (eval tr1) && (eval tr2)
eval (Node2 '+' tr1 tr2) =
  (eval tr1) || (eval tr2)
eval _ = error "Tree isn't correct!"
```

Для корректного логического выражения (формулы) это будет примерно так:

```
> eval $ fromLst $ snd $ step ("(1&(1+0))",[])
True
```

Можно сделать преобразования и самого дерева, и исходного выражения. Рассмотрим так называемую *обратную польскую запись*.

wikipedia: Обратная польская запись

Обработка входных аргументов функцией `reversePolish` идёт по рекурсии, аналогично `eval`, но с учётом требуемой задачи.

```
reversePolish :: PrsdTr -> String
reversePolish (Leaf '0') = "0"
reversePolish (Leaf '1') = "1"
reversePolish (Node1 '!' tr) =
  (reversePolish tr) ++ "!"
reversePolish (Node2 '&' tr1 tr2) =
  (reversePolish tr1) ++ (reversePolish tr2) ++ "&"
reversePolish (Node2 '+' tr1 tr2) =
  (reversePolish tr1) ++ (reversePolish tr2) ++ "+"
reversePolish _ = error "Tree isn't correct!"
```

На выходе получаем вполне ожидаемо:

```
> reversePolish $ fromLst $ snd $ step ("(1&(1+0))",[])
"110+&"
```

Парсинг регулярных выражений

Вместо стандартного алгоритма Томпсона, рассмотрим создание абстрактного синтаксического дерева и преобразование с его помощью в действующий конечный автомат. Для этого будем использовать модуль `NDFAFull` из прошлой лекции.

```
import Control.Monad
import Data.Maybe
import NDFAFull
```

Опишем восходящий парсер для очень упрощённой грамматики регулярных выражений:

терминалы: $a, b, |, *, (,)$

нетерминал: S

продукционные правила:

$$
\begin{aligned}
S &\to a \\
S &\to b \\
S &\to (SS) \\
S &\to (S \mid S) \\
S &\to S^*
\end{aligned}
$$

Для упрощения ситуации, конкатенация хоть и не подразумевает явного символа, но в нашем случае будет ограничена скобками и парой аргументов, напр.:

```
(ab)
((ab)a)
((a*b)(aa))
```

Объединение тоже будет явно в скобках: `(a|b)`, `((ab)|a)`

Создаваемый парсер будет строить в качестве результата следующее дерево синтаксического разбора:

```
data PrsdTr = Leaf Char | Node1 Char PrsdTr |
  Node2 Char PrsdTr PrsdTr
  deriving (Show,Read,Eq,Ord)
```

Введем ограничение на тип используемых элементов:

```
listVT = "ab|*()"
```

Проверим наличие разрешённых символов во входной строке.

```
checksymb :: Char -> Char
checksymb  c | (c `elem` listVT)     =  c
             | otherwise = error "There is an illegal symbol!"
```

Функция переноса:

```
transfer :: ([Char], [PrsdTr]) -> ([Char], [PrsdTr])
transfer ((s:ss),mss) = (ss, ((Leaf $ checksymb s):mss))
transfer ([],m)       = ([], m)
```

Технические функции:

```
justadd2list :: Maybe a -> [a] -> Maybe [a]
(Just x) `justadd2list` mss   = Just (x:mss)
Nothing  `justadd2list` _     = Nothing

mmplus :: (a, Maybe b) -> (a, b) -> (a, b)
(_,  Nothing)  `mmplus` t  = t
(ss, Just mss) `mmplus` _  = (ss,mss)
```

Вновь опишем продукционные правила как функции свёртки:

```
prodrule :: [PrsdTr] -> Maybe PrsdTr
prodrule [s, (Leaf '*')]  =
  Just $ Node1 '*' s
prodrule [(Leaf '('), s1,s2, (Leaf ')')]  =
  Just $ Node2 '.' s1 s2
prodrule [(Leaf '('), s1, (Leaf '|'), s2, (Leaf ')')]  =
  Just $ Node2 '|' s1 s2
prodrule _  = Nothing
```

Использование продукционных правил. Добавлен вариант для использования 4 символов. (помним про реверсное использование!)

```
use2symbols :: [PrsdTr] -> Maybe [PrsdTr]
use2symbols (m1:m2:mss)    =
  prodrule [m2,m1] `justadd2list` mss
use2symbols _  = Nothing

use4symbols :: [PrsdTr] -> Maybe [PrsdTr]
use4symbols (m1:m2:m3:m4:mss) =
  prodrule [m4,m3,m2,m1] `justadd2list` mss
use4symbols _  = Nothing

use5symbols :: [PrsdTr] -> Maybe [PrsdTr]
```

```
use5symbols (m1:m2:m3:m4:m5:mss) =
  prodrule [m5,m4,m3,m2,m1] `justadd2list` mss
use5symbols _  = Nothing
```

Основной цикл обработки.

```
step :: ([Char], [PrsdTr]) -> ([a], [PrsdTr])
step ([],mss) | (isNothing res) = ([],mss)
              | otherwise       = step ([],(fromJust res))
                   where res :: Maybe [PrsdTr]
                         res =
  (use2symbols  mss `mplus`
     use4symbols mss `mplus` use5symbols mss)

step (ss,mss) =
  let
    t =
      (ss,(use2symbols  mss `mplus`
         use4symbols mss `mplus` use5symbols mss))
          `mmplus` (transfer (ss,mss))
  in step t
```

Вновь функция распаковки списка из 1 элемента:

```
fromLst :: [a] -> a
fromLst [x] = x
fromLst  _  = error "List isn't single!"
```

Построение по дереву конечных автоматов:

```
eval :: PrsdTr -> NDFinStAutomata
eval (Leaf 'a') = lit 'a'
eval (Leaf 'b') = lit 'b'
eval (Node1 '*' tr) = kstar $ eval tr
eval (Node2 '.' tr1 tr2) = (eval tr1) `conc` (eval tr2)
eval (Node2 '|' tr1 tr2) = (eval tr1) `uni` (eval tr2)
eval _ = error "Tree isn't correct!"
```

Тестирование

```
> ncheck (eval $ fromLst $ snd $ step ("((aa)|b*)",[])) "bbb"
True
> ncheck (eval $ fromLst $ snd $ step ("((aa)|b*)",[])) "aa"
True
> ncheck (eval $ fromLst $ snd $ step ("(a*|b*)",[])) "aa"
True
> ncheck (eval $ fromLst $ snd $ step ("(a*|b*)",[])) "aab"
False
> ncheck (eval $ fromLst $ snd $ step ("(a*|b*)",[])) "bbb"
True
> ncheck (eval $ fromLst $ snd $ step ("((aa)*|b*)",[])) "bbb"
True
> ncheck (eval $ fromLst $ snd $ step ("((aa)*|b*)",[])) "aa"
True
```

```
> ncheck (eval $ fromLst $ snd $ step ("((aa)*|b*)",[])) "aaa"
False
> ncheck (eval $ fromLst $ snd $ step ("((aa)*|b*)",[])) "aaaa"
True
```

Дополнения

Рассмотренный самым первым парсер сбалансированных скобок может использовать состояния `State`, чтобы скрыть, например, входную строку, и работать только с магазином в аргументах функций и возвращаемых ими значений. Но из-за «лапши» вложенных **`if-then-else`** полученный код не стал более компактным, выигрыш получился небольшой. Однако, код стал более наглядным и в более императивном стиле:

```
import Control.Monad
import Data.Maybe
import Control.Monad.State

checksymb :: Char -> Char
checksymb  c | (c `elem` "()") =  c
             | otherwise =
                error "There is an illegal symbol!"

prodrule :: String -> Maybe Char
prodrule "()"  = Just 'S'
prodrule "(S)" = Just 'S'
prodrule "SS"  = Just 'S'
prodrule _     = Nothing

justadd2list :: Maybe a -> [a] -> Maybe [a]
(Just x) `justadd2list` mss   = Just (x:mss)
Nothing  `justadd2list` _     = Nothing

use2symbols :: [Char] -> Maybe [Char]
use2symbols (m1:m2:mss)    = prodrule [m2,m1] `justadd2list` mss
use2symbols _              = Nothing

use3symbols :: [Char] -> Maybe [Char]
use3symbols (m1:m2:m3:mss) = prodrule [m3,m2,m1] `justadd2list` mss
use3symbols _              = Nothing

step :: String -> State String String
step mss = do
  let res = use2symbols mss
             `mplus`
            use3symbols mss
  if isJust res
     then step $ fromJust res
     else do
           ss <- get
           if null ss then return mss
                      else do let s  = head ss
```

```
                                    let s' = tail ss
                                    put s'
                                    step $ (checksymb s):mss

parse :: String -> String
parse teststr = if (mss == "S")
  then teststr ++ " is valid"
  else teststr ++ " isn't valid"
     where mss = evalState (step []) teststr
```

Возможно, большей компактности удастся получить, если работать со вложенными монадам и использовать `State String (Maybe String)` — но это уже в следующий год :))

Parsec & Co

Отдельной и интересной темой является рассмотрение возможностей библиотеки Parsec, за многие годы ставшей жемчужиной языка `Haskell` и даже в некотором смысле его «визитной карточкой». Его возможности легко (и без «велосипедостроительства»:) позволяют реализовать КС-грамматики. Рассмотрим пример из начала лекции. Вот соответствующий код:

```
import Text.ParserCombinators.Parsec

xyz  :: Parser Char
xyz = (char 'x') <|> (char 'y') <|> (char 'z')
idn :: Parser [Char]
idn = xyz >> many(xyz <|> digit) >> eof >> return "ok"
```

Здесь комбинаторы `>>` (из монад, кстати) соответствуют последовательному выбору (в этом случае — конкатенации), `<|>` — альтернативному (параллельному) выбору из двух вариантов, `many` — повторению, `char` и `digit` помогают нам описать символы и цифры.

и вот результаты его работы:

```
*Main> parseTest idn "x"
"ok"
*Main> parseTest idn "x23"
"ok"
*Main> parseTest idn "2x23"
parse error at (line 1, column 1):
unexpected "2"
expecting "x", "y" or "z"
*Main> parseTest idn "xy2"
"ok"
*Main> parseTest idn "xQw2"
parse error at (line 1, column 2):
unexpected 'Q'
expecting "x", "y", "z", digit or end of input
```

Код на Haskell почти точно соответствует описанию в формате РБНФ. Но даже тут есть нюансы! Скажем, внезапно потребовалось указать для идентификатора конец строки (или файла) — `eof`, без которого строки вида `xyQ` проходили верификацию!

Но и это не самое страшное, в конце-концов тонкости работы описаны в отличной документации и ряде статей. Значительно худшим (или лучшим, смотря как рассматривать) является то, что `Parsec` строит нисходящий анализатор и это помимо хорошей скорости оборачивается необходимостью сильно модифицировать исходную грамматику, иногда очень нелогичным образом для избежания неоднозначности и «левой» рекурсии. Не вдаваясь в детали, скажем, что правило $S \to SS$ для грамматики сбалансированных скобок работать не будет — приведёт к бесконечному зацикливанию при анализе.

Ambiguous grammar?

wiki: Left recursion

wiki: LL parser/Left recursion removal

Придётся правила изменить, например, таким образом:

$$R \to ()|(R)|SR, \quad S \to R|\varepsilon$$

или в РБНФ (опять же, несколько изменив, за счёт наличия конструкции для повторов)

```
R = '()'|'('{R}')'
S = {R}
```

Реализация на Haskell версии РБНФ примерно будет такая:

```haskell
import Text.ParserCombinators.Parsec

r = try(string "()") <|> (string "(" >> r >> string ")")
s = many r >> eof >> return "ok"
```

(о функции **`try`** будет страницей ниже)

И каждый раз — это нетривиальный квест! :)

Компиляция. 5: нисходящий разбор

Вот как бы выглядела программа для валидации булевых формул, рассмотренных выше:

```haskell
import Text.ParserCombinators.Parsec

lit  :: Parser ()
lit = ((char '0') <|> (char '1')) >> return ()

ot :: Parser ()
ot = (char '!') >> bf >> return ()

kon :: Parser ()
kon = char '(' >> bf >> char '&' >> bf >> char ')' >> return ()

diz :: Parser ()
diz = char '(' >> bf >> char '+' >> bf >> char ')' >> return ()

bf  :: Parser ()
bf = (lit <|> ot <|> try kon <|> diz) >> return ()
```

```
test x = parseTest bf x
```

Здесь нам тоже три продукции пришлось заменить на такую коллекцию:

```
lit = '0'|'1'
ot  = '!',bf
kon = '(',bf,'\&',bf,')'
diz = '(',bf,'+',bf,')'
bf  = lit|ot|kon|diz
```

Здесь `lit` задаёт литералы, остальные правила работают взаимно-рекурсивно и задают `ot` — отрицания, `kon` — конъюнкции, `diz` — дизъюнкции, `bf` — готовые формулы.

Но и это оказалось ещё не всё! Так как конъюнкции и дизъюнкции начинаются одинаково, то из-за особенности реализации модуля (для скорости) парсер «съест» скобку и следующий символ и, наткнувшись на `+`, сообщит об ошибке. Необходимо установить перед проверкой комбинатор **`try`**, чтобы при неудаче был сделан откат и попытка использовать следующее правило.

Вот как будет выглядеть работа модуля в стиле монадного описания для построения дерева синтаксического разбора для наших булевых формул:

```
import Text.ParserCombinators.Parsec

data PrsdTr = Leaf Char | Node1 Char PrsdTr |
  Node2 Char PrsdTr PrsdTr
  deriving (Show,Read,Eq,Ord)

-- lit = '0'|'1'
lit  :: Parser PrsdTr
lit = do
   c <- (char '0' <|> char '1')
   return (Leaf c)

-- ot = '!',bf
ot :: Parser PrsdTr
ot = do
  char '!'
  b <- bf
  return (Node1 '!' b)

-- kon = '(',bf,'&',bf,')'
kon :: Parser PrsdTr
kon = do
   char '('
   b1 <- bf
   char '&'
   b2 <- bf
   char ')'
   return (Node2 '&' b1 b2)

-- diz = '(',bf,'+',bf,')'
diz :: Parser PrsdTr
```

```
diz = do
   char '('
   b1 <- bf
   char '+'
   b2 <- bf
   char ')'
   return (Node2 '+' b1 b2)

-- bf = lit | ot | kon | diz
bf  :: Parser PrsdTr
bf = (lit <|> ot <|> try kon <|> diz) >>= return

ready  :: Parser PrsdTr
ready = do
   b <- bf
   eof
   return b

-- test :: String -> PrsdTr
ptest = parseTest ready

test = parse ready ""

main = do
  ptest "((0&1)+!(1&0))"
  ptest "((0&1)+!(1&0))0"
  print $ test "((0&1)+!(1&0))"
  print $ test "((0&1)+!(1&0))0"
```

результат:

```
>runghc bfTree.hs
Node2 '+' (Node2 '&' (Leaf '0') (Leaf '1'))
 (Node1 '!' (Node2 '&' (Leaf '1') (Leaf '0')))
parse error at (line 1, column 15):
unexpected '0'
expecting end of input
Right (Node2 '+' (Node2 '&' (Leaf '0') (Leaf '1'))
 (Node1 '!' (Node2 '&' (Leaf '1') (Leaf '0'))))
Left (line 1, column 15):
unexpected '0'
expecting end of input
```

И кстати, дерево вывода, получаемое в результате работы модуля, является просто результатом, не участвующим в реальной работе «механики» модуля `Parsec`, поэтому, в случае обработки формулы & проблемы не возникает:

```
*Main> ptest "&"
parse error at (line 1, column 1):
unexpected "&"
expecting "0", "1", "!" or "("
```

Общий «вес» добавки модуля при статической линковке в среде Windows-7 (64 bit) примерно 6 мегабайт (после обработки утилитой `strip -s`).

Помимо монадного стиля, в современной версии пакета есть возможность работать в более легковесном стиле аппликативных функторов.

Также созданы более современные пакеты на базе или «с оглядкой» на Parsec: attoparsec, megaparsec, trifecta.

Более «хаскеловский вариант» последнего примера

Изменим алгебраический тип данных получаемого абстрактного дерева на вариант, менее зависимый от строкового представления, в форме, как это обычно принято при программировании на Haskell с помощью `Parseq`:

```
data BTr = T | F | N BTr | BTr :&: BTr |
   BTr :+: BTr  deriving (Show,Read,Eq,Ord)
```

Тогда наши парсеры примут вид:

```
-- lit = ’0’|’1’
lit  :: Parser BTr
lit = do
        c <- char '0'
        return F
      <|>
      do
        c <- char '1'
        return T

-- ot = ’!’,bf
ot :: Parser BTr
ot = do
  char '!'
  b <- bf
  return (N b)

-- kon = ’(’,bf,’&’,bf,’)’
kon :: Parser BTr
kon = do
   char '('
   b1 <- bf
   char '&'
   b2 <- bf
   char ')'
   return (b1 :&: b2)

-- diz = ’(’,bf,’+’,bf,’)’
diz :: Parser BTr
diz = do
   char '('
```

```haskell
    b1 <- bf
    char '+'
    b2 <- bf
    char ')'
    return (b1 :+: b2)

-- bf = lit | ot | kon | diz
bf  :: Parser BTr
bf = (lit <|> ot <|> try kon <|> diz) >>= return

ready  :: Parser BTr
ready = do
    b <- bf
    eof
    return b
```

И можно добавить следующую функцию `eval`:

```haskell
eval :: BTr -> Bool
eval T = True
eval F = False
eval (N b) = not $ eval b
eval (b1 :&: b2) = (eval b1) && (eval b2)
eval (b1 :+: b2) = (eval b1) || (eval b2)
```

И их комбинацию, для первичной интерпретации строки:

```haskell
runMe :: String -> Either ParseError Bool
runMe str = do
              t <- parse ready "" str
              return (eval t)
```

Использование `Parsec` совместно с `LLVM`-бэкендом

Отличная статья-пример по созданию «игрушечного» языка и компилятора на `Parsec` с построением аннотированного AST-дерева и последующей кодогенерации `LLVM IR` и бинарного представления. Рекомендуется для первого шага всем, кто хочет постигнуть тонкости «компиляторостроения».

Литература

1. *Касьянов В.Н.* Лекции по теории формальных языков, автоматов и сложности вычислений: Учеб. пособ. / Новосиб. гос. ун-т. Новосибирск, 1995.

2. *Ахо А.В., Лам М.С., Сети Р., Ульман Дж.Д.* Компиляторы: принципы, технологии и инструментарий. 2-е изд.: Пер. с англ. М.: ООО «И.Д. Вильямс», 2008.

3. *Хопкрофт Д., Мотвани Р., Ульман Д.* Введение в теорию автоматов, языков и вычислений. 2-е изд.: Пер. с англ. М.: Изд-во «Вильямс», 2002.

4. *Кормен Т., Лейзерсон Ч. и др.* Алгоритмы: построение и анализ. 2-е изд.: Пер. с англ. М.: Изд-во «Вильямс», 2005

5. *Lewis H.*, Papadimitriou Chr. Elements of the theory of computation. 2nd. ed. Prentice-Hall, Inc., 1997.

6. *Фоккер Е.* Функциональные парсеры. Eng.: *Jeroen Fokker,* Functional Parser.

7. *Hutton G., Meijer E.* Monadic Parsing in Haskell.

8. *Hutton G., Meijer E.* Monadic Parser Combinators. Перевод: Монадические комбинаторы парсеров или тут Монадические комбинаторы парсеров.

9. *Leijen D.* Parsec, a Fast Combinator Parser.

10. *Jake Wheat.* Intro to Parsing with Parsec in Haskell

11. An Introduction to the Parsec Library.

12. *James Wilson.* An introduction to parsing text in Haskell with Parsec.

13. wiki.haskell: Parsing a simple imperative language.

14. Write Yourself a Scheme in 48 Hours (частичный перевод).

15. Создаём парсер для ini-файлов на Haskell.

16. Аппликативные парсеры на Haskell.

17. *Ангэ Эсэф (Иван Веселов)*, Haskell: комбинаторные парсеры Parsec.

18. Пишем простой интерпретатор на Haskell.

19. Пишем (недо)интерпретатор на Haskell с помощью alex и happy.

20. *Marlow S., Gill A.* Happy User Guide.

21. Using Parsec with Data.Text.

22. Компиляция. 5: нисходящий разбор.

23. Компиляция. 2: грамматики.

24. Пузырьковый вычислитель выражений: простейший синтаксический LR-анализатор вручную.

25. Про LL-парсинг: Подход к синтаксическому анализу через концепцию нарезания строки.

26. Написание компилятора на Haskell + LLVM