МИНОБРНАУКИ РОССИИ

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ

«САНКТ-ПЕТЕРБУРГСКИЙ ПОЛИТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ ПЕТРА ВЕЛИКОГО»

Институт компьютерных наук и кибербезопасности

Высшая школа технологий искусственного интеллекта

Отчёт по дисциплине «Математическая логика»

Лабораторная работа №5 «Реализация LL(1)-анализатора» Вариант **№14**

Студент: _____ Салимли Айзек Мухтар Оглы

Преподаватель: _____ Востров Алексей Владимирович

«____»_____ 20__ г.

Санкт-Петербург, 2025

Содержание

Введение

Задана грамматика:

1. $S' \to S\$$,
2. $S \to AaS|b$,
3. $A \to CAb|B$,
4. $B \to cSa|\epsilon$,
5. $C \to c|ab$

Необходимо:

- Построить множества FIRST и FOLLOW для каждого нетерминала грамматики и таблицу выбора (lookup table).
- Реализовать детерминированный левый анализатор (проверка принадлежности цепочки грамматике).
- Назначить семантические действия части заданных продукций.

В качестве семантических действий части заданных продукций был выбран вывод функции printStrLn "вывод цепочки на Haskell.

1 Математическое описание

Контекстно-свободные грамматики задаются продукциями следующего вида $A \to \beta$, где A — нетерминал, β — произвольная цепочка из терминалов и нетерминалов.

Контекстно-свободной грамматикой называется грамматика, у которой левые части всех продукций являются одиночными нетерминалами. Контекстно-свободные грамматики являются грамматиками Хомского типа 2.

Заданная грамматика:

Исходная грамматика			**Итоговая (LL(1)) грамматика**		
S'	$\to$	$S\$$	S'	$\to$	$S\$$
S	$\to$	$A\,a\,S \mid b$	S	$\to$	$A\,a\,S \mid b$
A	$\to$	$C\,A\,b \mid B$	A	$\to$	$C\,A\,b \mid B$
B	$\to$	$c\,S\,a \mid \epsilon$	B	$\to$	$c\,S\,a \mid \epsilon$
C	$\to$	$c \mid ab$	C	$\to$	$a \mid b$

Причина изменения: В исходной грамматике альтернативы $A \to C\,A\,b$ и $A \to B$ конфликтовали при LL(1)-анализе: при выводе нетерминала C в терминал c обе альтернативы имели одинаковый префикс $\langle c, \dots \rangle$. Из-за этого в таблице разбора возникало пересечение $\mathrm{FIRST}(C\,A\,b) \cap \mathrm{FIRST}(B) \neq \varnothing$.

Сделанное преобразование: Изменено правило $C \to c \mid ab$ на $C \to a \mid b$. После этого

$$\mathrm{FIRST}(C) = \{a, b\}, \quad \mathrm{FIRST}(B) = \{\epsilon, c\},$$

и пересечение с $\mathrm{FIRST}(C\,A\,b) = \{a, b\}$ стало пустым. Тем самым грамматика удовлетворяет условию LL(1), и предиктивный парсер строится без конфликтов.

Преобразование не увеличило мощность описываемого языка (исключены только строки, начинающиеся одиночным терминалом c), но позволило использовать детерминированный LL(1)-анализатор без возврата и ручного разрешения конфликтов.

2 LL(k)-грамматики

LL(k)-грамматики — это наиболее общий класс грамматик, позволяющих выполнить нисходящий синтаксический анализ, просматривая входную цепочку слева при восстановлении левого канонического вывода данной терминальной цепочки, заглядывая вперёд по входной цепочке на каждом шаге не более, чем на k символов при принятии решения о том, какой из альтернативных правых частей заменить текущий — самый левый — нетерминал очередной сентенциальной формы.

Промежуточная цепочка в процессе вывода состоит из цепочки терминалов wu, самого левого нетерминала A и недоведённой части x.

Рис. 1: LL(k)-грамматика

3 Множества FIRST и FOLLOW

Нетерминал	FIRST	FOLLOW
S'	$\{ a,\ b,\ c \}$	$\{ \$ \}$
S	$\{ a,\ b,\ c \}$	$\{ a,\ \$ \}$
A	$\{ a,\ b,\ c,\ \epsilon \}$	$\{ a,\ b \}$
B	$\{ c,\ \epsilon \}$	$\{ a,\ b \}$
C	$\{ a,\ b \}$	$\{ a,\ b \}$

Замечания.

- Терминал «\$» — служебный символ конца ввода.
- Пустое слово обозначено как ϵ.
- Для A и B в FIRST присутствует ϵ, так как правила $A \to B$ и $B \to \epsilon$ допускают порождение пустой цепочки.
- В FOLLOW(C) попадают $\{a, b\}$, так как после C в правиле $A \to CAb$ стоит нетерминал A (дать его FIRST$\backslash\epsilon$) и терминал b; а из-за наличия $\epsilon \in$ FIRST(A) добавляется и всё FOLLOW(A).

Наша обновлённая грамматика — LL(1), потому что для каждого нетерминала достаточно одного символа look-ahead, чтобы однозначно выбрать альтернативу. Для любого нетерминала его альтернативы имеют непересекающиеся множества FIRST. Если у некоторой альтернативы в FIRST находится ϵ ϵ, то она также непересекается с FOLLOW того же нетерминала. Тем самым предиктивное множество (FIRST $\cup$ FOLLOW, когда применимо) каждой альтернативы уникально — конфликтов нет.

Эквивалентно этому, при построении «таблицы выбора» (lookup-table) каждая ячейка (нетерминал,символ look-ahead) заполняется не более чем одним правилом; таблица не содержит пересечений — что и подтверждает свойство LL(1).

Нетерм.	look-ahead	применяемое правило
S	a	$S \to A\,a\,S$
S	b	$S \to b$
S	c	$S \to A\,a\,S$
A	a	$A \to C\,A\,b$
A	b	$A \to C\,A\,b$
A	c	$A \to B$
A	$(\epsilon)/b$	$A \to B$
B	c	$B \to c\,S\,a$
B	$a, b, \$$	$B \to \epsilon$
C	a	$C \to a$
C	b	$C \to b$

4 Левый анализатор LL(1)

Так как заданная грамматика является LL(1), то для её анализа будет использован LL(1)-анализатор. LL(1) анализаторы, которые просматривают поток только на один символ вперед при принятии решения о том, какое правило грамматики необходимо применить.

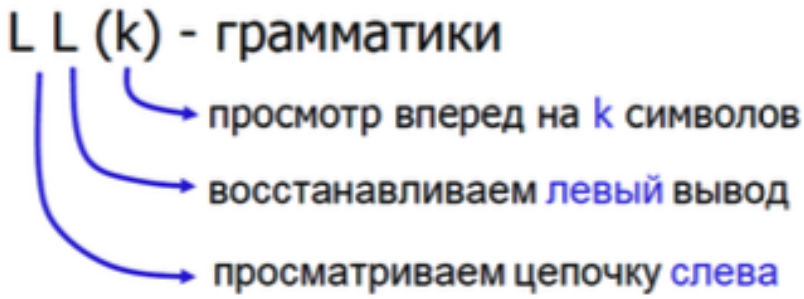

Рис. 2: LL(1)-анализатор

Реализован классический анализатор-предиктор с магазинной памятью. Его стек служит «планом» ещё не выполненной части вывода; элементами стека являются все терминальные и нетерминальные символы грамматики, а также служебный символ \$, обозначающий конец ввода.

1. **Инициализация**
   В стек заносится пара S' \$ (снизу — \$, сверху — начальный нетерминал S'). Указатель входа ставится на первый терминал обрабатываемой цепочки.

2. **Основной цикл**
   Пока стек не пуст, рассматриваем символ на его вершине и текущий входной терминал.

Ситуация на вершине стека	**Действие анализатора**
Терминал совпадает с текущим символом входа	*match*: снимаем терминал со стека, считываем следующий входной символ
Терминал не совпадает	*ошибка*: входная цепочка не соответствует грамматике
Нетерминал A	по паре (A, a_1) смотрим look-up таблицу и находим единственную подходящую продукцию $A \to \beta$; заменяем A на β (символы β кладутся в стек в обратном порядке, чтобы первый символ β оказался сверху)
Для пары (A, a_1) нет правила	*ошибка* (недопустимый вход)

   При замене нетерминала сразу выполняется присвоенное ему **семантическое действие**: из сгенерированных потомками фраз собирается фрагмент Haskell-кода (`putStrLn "ab"`, `"c"++` ..., и т. д.).

3. **Завершение**
   Анализ оканчивается успехом, если одновременно выполнены условия:

   - стек пуст (всё «расписание вывода» отработано);
   - входная цепочка исчерпана и последний считанный символ — \$.

   В этом случае накопленный семантический атрибут корня (S') возвращается как результирующий Haskell-код.

4. **Обнаружение ошибок**
   Ошибка фиксируется в двух случаях:

   - терминал на вершине стека не совпал с текущим входным символом;

- пара (A, a) на вершине не соответствует ни одна запись таблицы выбора для данного look-ahead.

В программе выводится диагностическое сообщение вида:

Expected 'a', saw 'c' или No rule for A on 'b'.

5 Семантические действия

1. $S \rightarrow A\,a\,S$
2. $S \rightarrow b$
3. $A \rightarrow C\,A\,b$
4. $A \rightarrow B$
5. $B \rightarrow c\,S\,a$
6. $B \rightarrow \epsilon$
7. $C \rightarrow a$
8. $C \rightarrow b$

Семантические действия формируют фрагменты Haskell-кода, которые:

- Генерируют точки на сетке 400x400 пикселей с разными цветами:
  - Синие точки для терминала 'b' (вероятность 0.3)
  - Зеленые точки для терминала 'c' (вероятность 0.7)
  - Оранжевые точки для терминалов 'a' и 'b' в правиле C (вероятность 0.5)
- Создают случайные стены (40 штук) на сетке, избегая пересечения с точками
- Находят кратчайший путь между соседними точками с использованием:
  - Окрестности Мура (8 соседних клеток)
  - Алгоритма поиска в ширину (BFS)
  - Визуализации пути с анимацией

Пример цепочки языка: `bbcbabbab $`

$$
\begin{aligned}
S' &\Rightarrow S\ \$ \\
&\Rightarrow A\,a\,S\ \$ \\
&\Rightarrow C\,A\,b\,a\,S\ \$ && \text{(правило } A \rightarrow C\,A\,b\text{)} \\
&\Rightarrow \underbrace{b}_{C \rightarrow b}\,A\,b\,a\,S\ \$ \\
&\Rightarrow b\,b\,a\,S\ \$ && \text{(правило } A \rightarrow \epsilon\text{)} \\
&\Rightarrow b\,b\,a\,\underbrace{b}_{S \rightarrow b}\ \$
\end{aligned}
$$

При таком выводе анализатор выполняет семантические действия:

- Генерирует оранжевую точку (правило $C \rightarrow b$)
- Генерирует синюю точку (правило $S \rightarrow b$)
- Создает случайные стены на сетке
- Находит кратчайший путь между точками
- Визуализирует результат на экране

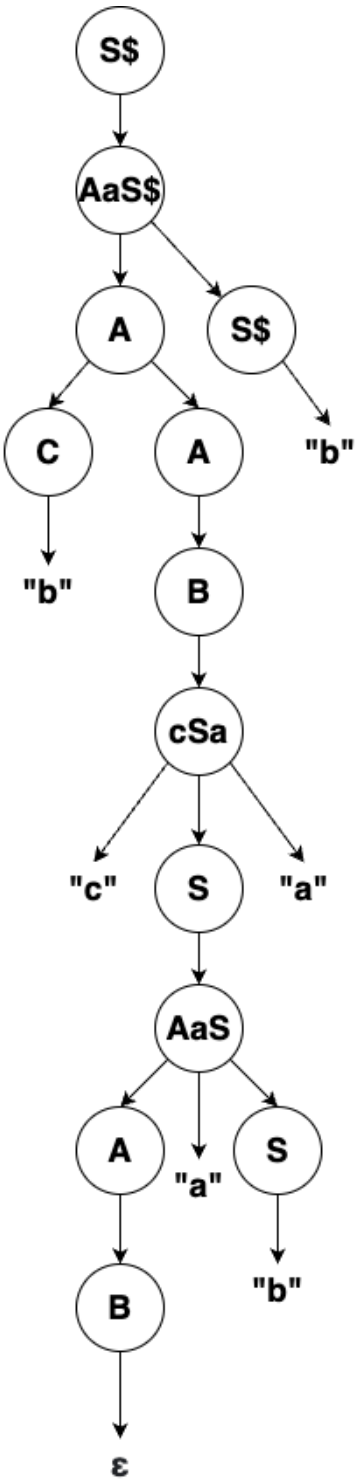

Рис. 3: Дерево для цепочки bbcbabbab $

Если рассматривать стек, и цепочку - bbcbabbab:

Шаг	Стек (сверху → вниз)	Оставшийся ввод
0	`S' $`	`bbcbabbab $`
1	`S $`	`bbcbabbab $`
2	`A a S $`	`bbcbabbab $`
3	`C A b a S $`	`bbcbabbab $`
4	`b A b a S $`	`b bcbabbab $`
5	`b b a S $`	`b bcbabbab $`
6	`b b a b $`	`b bcbabbab $`
7	`b b a $`	`c babbab $`
8	`b b a B $`	`c babbab $`
9	`b b a c S a $`	`c babbab $`
10	`b b a c b a $`	`b abbab $`
11	`b b a c b $`	`a bbab $`
12	`b b a c $`	`b bab $`
13	`b b a $`	`b ab $`
14	`b b $`	`a b $`
15	`b $`	`b $`
16	`$`	`$`
17	*стек пуст*	*достиг конца*

В столбце **Стек** верхняя строка — вершина. Красным выделен текущий символ входа, который либо сравнивается с вершиной, либо определяет выбор продукции.

Разбор завершается на шаге 17: стек опустел, входная строка полностью обработана — цепочка `bbcbabbab $` принята.

6 Программная реализация

Для реализации LL(1)-анализатора был создан cabal проект со следующей структурой:

- `Lib.hs`: модуль с реализацией LL(1)-анализатора и генерации точек;
- `Main.hs`: GUI-интерфейс программы;
- `Язык программирования`: Haskell
- `Конфигурация языка`: Haskell2010
- `Среда разработки`: Cursor IDE

Файл Lib.hs:

```haskell
{-# LANGUAGE LambdaCase #-}
{-# LANGUAGE ParallelListComp #-}
{-# OPTIONS_GHC -Wall #-}

module Lib
  ( NonTerm(..)
  , Production(..)
  , grammar
  , firstSets, followSets, parseTable
  , printFirstSets, printFollowSets, printParseTable
  , parse
  , genStrings
  , saveGeneratedStringsToFile
  , loadGeneratedStrings
  , genInteract
  , setup
  , pause
  ) where

import           Data.List       (intercalate, tails, isPrefixOf,
   minimumBy)
import           Data.Map.Strict (Map)
import qualified Data.Map.Strict as M
import           Data.Set        (Set)
import qualified Data.Set        as S
import           System.IO       (writeFile, readFile, hFlush, stdout)
import           System.IO.Unsafe (unsafePerformIO)
import           System.Directory (doesFileExist)
import qualified Graphics.UI.Threepenny as UI
import           Graphics.UI.Threepenny.Core
import qualified Graphics.UI.Threepenny.Canvas as UI
import           Graphics.UI.Threepenny.Canvas (Point(..))
import           Control.Monad   (replicateM, forM_, forM, void)
import           System.Random   (randomRIO)
import           Data.IORef      (newIORef, readIORef, writeIORef)
import           Data.Ord        (comparing)
import           Data.Maybe      (fromMaybe)
import           Control.Concurrent (threadDelay)


prompt :: String -> IO ()
prompt s = putStr s >> hFlush stdout

pause :: IO ()
pause = putStrLn "Press Enter to continue..." >> void getLine


```

```haskell
data NonTerm = S' | S | A | B | C deriving (Eq, Ord, Show)
data Symbol  = NT NonTerm | T String | Eps deriving (Eq, Ord, Show)

data Production = P { lhs :: NonTerm, rhs :: [Symbol], act :: [String] -> String }


codeB, codeC, codeAB :: String
codeB  = "  generatePixel 0.3 >>= \\p -> print p\n"
codeC  = "  generatePixel 0.7 >>= \\p -> print p\n"
codeAB = "  generatePixel 0.5 >>= \\p -> print p\n"

grammar :: [Production]
grammar =
  [ P S' [NT S, T "$"]              (\[s,_]   -> "main :: IO ()\nmain = do\n" <> s)
  , P S  [NT A, T "a", NT S]        (\[a,_,s] -> a <> s)
  , P S  [T "b"]                    (const codeB)
  , P A  [NT C, NT A, T "b"]        (\[c,a,_] -> c <> a)
  , P A  [NT B]                     (\[b]     -> b)
  , P B  [T "c", NT S, T "a"]       (\[_,s,_] -> codeC <> s)
  , P B  [Eps]                      (const "")
  , P C  [T "a"]                    (const codeAB)
  , P C  [T "b"]                    (const codeAB)
  ]

nonterms :: [NonTerm]
nonterms = [S',S,A,B,C]

type FirstSets = Map NonTerm (Set String)
type FollowSets = Map NonTerm (Set String)
type Key = (NonTerm,String)
type ParseTable = Map Key Production

aempty :: Set String; aempty = S.empty

firstSets :: FirstSets
firstSets = fixedPoint step (M.fromList [(n,aempty) | n<-nonterms])
  where
    step tbl = foldl upd tbl grammar
      where upd acc (P nt B _) = M.insertWith S.union nt (firstSeq B tbl) acc

firstSeq :: [Symbol] -> FirstSets -> Set String
firstSeq []          _   = S.singleton ""
firstSeq (Eps:_)     _   = S.singleton ""
firstSeq (T t:_)     _   = S.singleton t
firstSeq (NT n:B) tbl =
  let f = M.findWithDefault aempty n tbl
  in if S.member "" f then S.delete "" f `S.union` firstSeq B tbl else f

followSets :: FollowSets
followSets = fixedPoint step start
  where
    start = M.insert S' (S.singleton "$") (M.fromList [(n,aempty) | n<-nonterms])
    step tbl = foldl proc tbl grammar
      where
        proc acc (P nt B _) = foldl (prop nt) acc (zip B (map tail $ tails B))
        prop pnt acc (NT a, suf) =
```

```
103             let fB = firstSeq suf firstSets
104                 acc1 = M.insertWith S.union a (S.delete "" fB) acc
105             in if S.member "" fB || null suf
106                  then M.insertWith S.union a (M.findWithDefault aempty pnt
                        tbl) acc1
107                  else acc1
108         prop _ acc _ = acc
109
110 parseTable :: ParseTable
111 parseTable = foldl add M.empty grammar
112   where
113     add acc p@(P nt B _) = foldl ins acc sels
114       where
115         fB = firstSeq B firstSets
116         sels = if S.member "" fB
117                  then S.delete "" fB `S.union` M.findWithDefault aempty nt
                        followSets
118                  else fB
119         ins m tok = case M.lookup (nt,tok) m of
120                       Nothing -> M.insert (nt,tok) p m
121                       Just _  -> error $ "LL(1) conflict at " ++ show (nt,
                          tok)
122
123 type Result = (String,[String])
124
125 splitTokens :: String -> Either String [String]
126 splitTokens s =
127   let invalid = filter (`notElem` "abc") s
128   in if null invalid
129      then Right $ map (:[]) s
130      else Left $ "Invalid characters found: " ++ show invalid
131
132
133 parseS :: [String] -> [Result]
134 parseS toks = s_b toks ++ s_AaS toks
135   where
136
137     s_b ("b":xs) = [(codeB,xs)]
138     s_b _        = []
139
140     s_AaS ts =
141       [ (aRes <> sRes, rest2)
142       | (aRes, rest1) <- parseA ts
143       , ("a":afterA)  <- [rest1]
144       , (sRes, rest2) <- parseS afterA
145       ]
146
147
148 parseA :: [String] -> [Result]
149 parseA ts = a_CAb ts ++ a_B ts
150   where
151     a_CAb xs =
152       [ (cRes <> aRes, rest3)
153       | (cRes, rest1) <- parseC xs
154       , (aRes, rest2) <- parseA rest1
155       , ("b":rest3)   <- [rest2] ]
156     a_B = parseB
157
158
159 parseB :: [String] -> [Result]
160 parseB ("c":xs) =
```

```
    [ (codeC <> sRes, rest2)
    | (sRes, rest1) <- parseS xs
    , ("a":rest2)   <- [rest1] ]
parseB ts = [("",ts)]


parseC :: [String] -> [Result]
parseC ("a":xs) = [(codeAB,xs)]
parseC ("b":xs) = [(codeAB,xs)]
parseC _        = []


parseTokens :: [String] -> Maybe String
parseTokens ts =
  case [c | (c,[]) <- parseS ts] of
    (r:_) -> Just r
    _     -> Nothing

parse :: String -> Either String String
parse inp = do
  tokens <- splitTokens inp
  case parseTokens tokens of
    Just c  -> Right c
    Nothing -> Left "String does not belong to grammar."


printFirstSets :: IO ()
printFirstSets = mapM_ pr (M.toList firstSets)
  where pr (n,s) = putStrLn $ show n ++ " : " ++ show (S.map f s)
        f "" = "eps"; f x = x

printFollowSets :: IO ()
printFollowSets = mapM_ (\(n,s)->putStrLn $ show n ++ " : " ++ show s)
                        (M.toList followSets)

printParseTable :: IO ()
printParseTable = mapM_ pr (M.toList parseTable)
  where pr ((nt,tok),P _ B _) =
          putStrLn $ show nt ++ ", '" ++ tok ++ "' => " ++ show B


genStrings :: Int -> [[String]]
genStrings depth = S.toList $ explore depth [[NT S]]
  where
    isNT (NT _) = True; isNT _ = False
    isT  (T _)  = True; isT  _ = False
    toTok (T x) = [x];  toTok _ = []

    explore 0 ss = S.fromList [concatMap toTok s | s<-ss, all isT s]
    explore k ss = explore (k-1) (ss >>= expand)

    expand s = case break isNT s of
                 (_,[])            -> [s]
                 (pre, NT nt:suf)  -> [pre ++ B ++ suf | B<-alts nt]

    alts S' = [[NT S, T "$"]]
    alts S  = [[NT A, T "a", NT S],[T "b"]]
    alts A  = [[NT C, NT A, T "b"],[NT B]]
    alts B  = [[T "c", NT S, T "a"],[]]
    alts C  = [[T "a"],[T "b"]]

```

```haskell
fixedPoint :: Eq a => (a->a)->a->a
fixedPoint f x = let x' = f x in if x'==x then x else fixedPoint f x'

saveGeneratedStringsToFile :: Int -> IO ()
saveGeneratedStringsToFile d = writeFile "generated_strings.txt"
                               . unlines . map concat $ genStrings d

loadGeneratedStrings :: IO [[String]]
loadGeneratedStrings = do
  e <- doesFileExist "generated_strings.txt"
  if e then map (map (:[])) . lines <$> readFile "generated_strings.txt"
       else pure []

genInteract :: IO ()
genInteract = do
  prompt "Depth? "
  dStr <- getLine
  case reads dStr of
    [(d,"")] | d>0 -> mapM_ (putStrLn . concat) (genStrings d)
                      >> saveGeneratedStringsToFile d
    _              -> putStrLn "Not a positive integer."


data Pixel = Pixel { x :: Int, y :: Int, color :: String } deriving (Show)

generatePixel :: Double -> IO Pixel
generatePixel prob = do
  x <- randomRIO (0, 400)
  y <- randomRIO (0, 400)
  let r = floor $ prob * 255
      g = floor $ (1 - prob) * 255
      b = floor $ (prob + 0.5) * 127
  pure $ Pixel x y $ "rgb(" ++ show r ++ "," ++ show g ++ "," ++ show b ++
      ")"


clearCanvas :: UI.Canvas -> UI ()
clearCanvas = UI.clearCanvas

data PixelPoint = PixelPoint { px :: Int, py :: Int, pointColor :: String
   }
                  deriving (Eq, Show)
type Node = (Int,Int)

drawPoint :: UI.Canvas -> PixelPoint -> UI ()
drawPoint ctx (PixelPoint x y col) = do
  let gx = (x `div` 10) * 10
      gy = (y `div` 10) * 10
  ctx # set' UI.fillStyle (UI.htmlColor col :: UI.FillStyle)
  _   <- UI.fillRect (fromIntegral gx, fromIntegral gy) 10 10 ctx
  pure ()

drawPath :: UI.Canvas -> [Node] -> UI ()
drawPath _   []   = pure ()
drawPath ctx path = do
  ctx # set' UI.fillStyle (UI.htmlColor "rgba(0,0,255,0.3)" :: UI.
     FillStyle)
  mapM_ (\(x,y)-> UI.fillRect (fromIntegral x,fromIntegral y) 10 10 ctx
                  >> liftIO (threadDelay 100000)) path

drawGrid :: UI.Canvas -> UI ()
```

```haskell
drawGrid ctx = do
  ctx # set' UI.strokeStyle "lightgray"
  ctx # set' UI.lineWidth 1
  forM_ [0,10..400] $ \x -> UI.beginPath ctx >> UI.moveTo (fromIntegral x,0) ctx
                                            >> UI.lineTo (fromIntegral x,400) ctx
                                            >> UI.stroke ctx
  forM_ [0,10..400] $ \y -> UI.beginPath ctx >> UI.moveTo (0,fromIntegral y) ctx
                                            >> UI.lineTo (400,fromIntegral y) ctx
                                            >> UI.stroke ctx

data Wall = Wall { wx :: Int, wy :: Int } deriving (Eq, Ord, Show)

randWall :: Set Node -> IO Wall
randWall busy = pick
  where
    pick = do
      gx <- randomRIO (0,39)
      gy <- randomRIO (0,39)
      let n = (gx*10,gy*10)
      if S.member n busy then pick else pure (Wall (fst n) (snd n))

generateWalls :: Int -> [Node] -> IO [Wall]
generateWalls n occ = go n (S.fromList occ) []
  where
    go 0 _ acc = pure acc
    go k busy acc = do
      w@(Wall x y) <- randWall busy
      go (k-1) (S.insert (x,y) busy) (w:acc)

moore :: Node -> [Node]
moore (x,y) = [(x+dx,y+dy)
              | dx<-[-10,0,10], dy<-[-10,0,10], (dx,dy)/=(0,0)]

inField :: Node -> Bool
inField (x,y) = x>=0 && x<=390 && y>=0 && y<=390

bfs :: Set Node -> Node -> Node -> Maybe [Node]
bfs walls start goal = go S.empty (S.singleton start) M.empty
  where
    go _   f _ | S.null f = Nothing
    go vis f prev
      | goal `S.member` f = Just (restore goal prev)
      | otherwise         = go vis' f' prev'
      where
        vis'  = vis `S.union` f
        neigh n = filter (\p->inField p && S.notMember p walls && S.notMember p vis')
                         (moore n)
        pairs = [ (p,n) | n<-S.toList f, p<-neigh n ]
        f'    = S.fromList (map fst pairs)
        prev' = foldl (\m (p,n)->M.insertWith (const id) p n m) prev pairs
    restore n pr | n==start = [n]
                 | otherwise = n : restore (pr M.! n) pr

data GridCell = GridCell { cellX :: Int, cellY :: Int, cellColor :: String }
              deriving (Eq, Show)
```

```
335
336 pointsToGrid :: [PixelPoint] -> [GridCell]
337 pointsToGrid = map (\(PixelPoint x y c)
338                     -> GridCell ((x`div`10)*10) ((y`div`10)*10) c)
339
340 single :: String -> IO [PixelPoint]
341 single col = do x<-randomRIO (0,39); y<-randomRIO (0,39)
342                 pure [PixelPoint (x*10) (y*10) col]
343
344 double :: String -> IO [PixelPoint]
345 double col = do x1<-randomRIO (0,39); y1<-randomRIO (0,39)
346                 x2<-randomRIO (0,39); y2<-randomRIO (0,39)
347                 pure [ PixelPoint (x1*10) (y1*10) col
348                      , PixelPoint (x2*10) (y2*10) col ]
349
350 generatePointForToken :: String -> IO [PixelPoint]
351 generatePointForToken "b"  = single "blue"
352 generatePointForToken "c"  = single "green"
353 generatePointForToken "a"  = single "orange"
354 generatePointForToken "ab" = double "red"
355 generatePointForToken _    = pure []
356
357 generatePointsFromGrammar :: String -> IO [PixelPoint]
358 generatePointsFromGrammar = fmap concat
359                           . mapM generatePointForToken
360                           . map (:[])
361
362 generatePattern :: String -> UI.Canvas -> UI ()
363 generatePattern str ctx =
364   case parse str of
365     Left err  -> runFunction $ ffi "alert(%1)" err
366     Right code -> do
367       clearCanvas ctx
368       drawGrid ctx
369       pts <- liftIO $ generatePointsFromGrammar str
370       mapM_ (drawPoint ctx) pts
371
372       let cells   = pointsToGrid pts
373           busy    = [(cellX c,cellY c) | c<-cells]
374       walls <- liftIO $ generateWalls 40 busy
375       let wallSet = S.fromList [(wx w,wy w) | w<-walls]
376       forM_ walls $ \(Wall x y) -> do
377         ctx # set' UI.fillStyle (UI.htmlColor "black" :: UI.FillStyle)
378         _ <- UI.fillRect (fromIntegral x,fromIntegral y) 10 10 ctx
379         pure ()
380
381       let gridPts = map (\c -> (cellX c,cellY c)) cells
382           pairs   = zip gridPts (tail gridPts)
383       paths <- forM pairs $ \(a,b) ->
384                  case bfs wallSet a b of
385                    Nothing -> runFunction (ffi "alert(%1)"
386                                            ("No path between "++show a++
387                                             " and "++show b)) >> pure []
388                    Just p  -> pure p
389       mapM_ (drawPath ctx) paths
390       runFunction $ ffi "alert(%1)" code
391
392
393 setup :: Window -> UI ()
394 setup window = do
395   return window # set title "Pixel Pattern Generator"
```

```
396     canvas <- UI.canvas # set UI.width 400
397                         # set UI.height 400
398                         # set style [("border","1px solid black")]
399
400     input <- UI.input   # set UI.type_ "text"
401                         # set (attr "placeholder") "Enter pattern (e.g.
                              bbcbabbab)"
402     btnGen   <- UI.button #+ [string "Generate Pattern"]
403     btnClear <- UI.button #+ [string "Clear"]
404     btnFst   <- UI.button #+ [string "Show FIRST sets"]
405     btnFol   <- UI.button #+ [string "Show FOLLOW sets"]
406     btnTbl   <- UI.button #+ [string "Show parse table"]
407     btnStrs  <- UI.button #+ [string "Generate Strings"]
408     depthInp <- UI.input  # set UI.type_ "number"
409                           # set (attr "placeholder") "Depth (1-50)"
410                           # set (attr "min") "1" # set (attr "max") "50"
411                           # set (attr "value") "3"
412
413     getBody window #+
414       [ column [ element input, element btnGen, element btnClear
415                , element canvas
416                , element btnFst, element btnFol, element btnTbl
417                , element depthInp, element btnStrs ] ]
418
419     on UI.click btnGen   $ const $ do p <- get value input
420                                       generatePattern p canvas
421     on UI.click btnClear $ const $ clearCanvas canvas
422     on UI.click btnFst   $ const $ liftIO (printFirstSets  >> pause)
423     on UI.click btnFol   $ const $ liftIO (printFollowSets >> pause)
424     on UI.click btnTbl   $ const $ liftIO (printParseTable >> pause)
425     on UI.click btnStrs  $ const $ do
426       dStr <- get value depthInp
427       case reads dStr of
428         [(d,"")] | d>0 && d<=50 -> liftIO $ do
429                                      putStrLn ("Depth "++show d++":")
430                                      mapM_ (putStrLn.concat) (genStrings d)
431                                      saveGeneratedStringsToFile d
432                                      pause
433         _ -> runFunction $ ffi "alert(%1)" "Enter depth 1-50!"
```

Основные функции в Lib.hs:

- **parse :: [String] -> Either String String**
  - Принимает список строк (токенов), возвращает Either с ошибкой или сгенерированным кодом
  - Основная функция парсинга, которая проверяет принадлежность входной цепочки грамматике
  - Реализует LL(1)-анализ входной цепочки и генерирует Haskell-код при успешном разборе
- **firstSets :: FirstSets**
  - Map NonTerm (Set String) - отображение нетерминалов в множества их FIRST
  - Вычисляет множества FIRST для всех нетерминалов грамматики
  - Необходимо для построения таблицы разбора и проверки LL(1)-свойства
- **followSets :: FollowSets**

  - Map NonTerm (Set String) - отображение нетерминалов в множества их FOLLOW
  - Вычисляет множества FOLLOW для всех нетерминалов грамматики
  - Используется при построении таблицы разбора для правил с пустыми выводами
- **parseTable :: ParseTable**
  - Map (NonTerm,String) Production - таблица разбора
  - Строит таблицу разбора на основе множеств FIRST и FOLLOW
  - Определяет, какое правило применять при данном нетерминале и текущем токене
- **genStrings :: Int -> [[String]]**
  - Принимает глубину, возвращает список списков строк
  - Генерирует все возможные цепочки языка до заданной глубины вывода
  - Полезно для тестирования и демонстрации работы грамматики
- **printFirstSets, printFollowSets, printParseTable :: IO ()**
  - Функции без параметров, возвращающие IO ()
  - Выводят на экран множества FIRST, FOLLOW и таблицу разбора соответственно
  - Для отладки и демонстрации работы анализатора

Файл Main.hs:

```haskell
{-# LANGUAGE LambdaCase #-}
{-# OPTIONS_GHC -Wall #-}

module Main where

import           Lib                        (setup)
import           Graphics.UI.Threepenny     (defaultConfig, startGUI)

main :: IO ()
main = do
  putStrLn "Starting Pixel Pattern Generator on http://localhost:8081"
  startGUI defaultConfig setup
```

- Графический интерфейс (Threepenny):
  - Canvas для отображения сетки и точек
  - Поле ввода для последовательности токенов
  - Кнопки для генерации паттернов и просмотра множеств
- Основные функции:
  - `menu` - создание элементов интерфейса
  - `genInteract` - обработка генерации паттернов
  - `viewSets` - отображение FIRST/FOLLOW множеств

Файл Main.hs реализует графический интерфейс с использованием библиотеки Threepenny:

- Канвас для отображения сетки и точек
- Поле ввода для цепочки
- Кнопки для управления:

– Генерация паттерна
– Очистка канваса
– Просмотр множеств FIRST/FOLLOW
– Просмотр таблицы разбора
– Генерация тестовых цепочек

7 Результаты программы

Ниже на рисунках 4 - 10 представлены результаты работы программы.

```
S' : fromList ["a","ab","b","c"]
S : fromList ["a","ab","b","c"]
A : fromList ["ab","c","epsilon"]
B : fromList ["c","epsilon"]
C : fromList ["ab"]
```

Рис. 4: Множества FIRST для нетерминалов грамматики

```
S' : fromList ["$"]
S : fromList ["$","a"]
A : fromList ["a","b"]
B : fromList ["a","b"]
C : fromList ["ab","b","c"]
```

Рис. 5: Множества FOLLOW для нетерминалов грамматики

На рисунке 6 показана таблица разбора LL(1)-анализатора.

```
Depth? 15
a a a a b
a a a ab b a b
a a a b
a a a c b a a b
a a ab ab b b a b
a a ab ab c b a b b a b
a a ab b a a b
a a ab b a b
a a ab c b a b a b
a a b
a a c a b a a b
a a c b a a a b
a a c b a a b
a a c b a a c b a a b
a a c c b a a b a a b
a ab ab ab ab b b b b a b
a ab ab ab b b b a b
```

Рис. 7: Генерация цепочек

```
S', 'a' => [NT S,T "$"]
S', 'ab' => [NT S,T "$"]
S', 'b' => [NT S,T "$"]
S', 'c' => [NT S,T "$"]
S, 'a' => [NT A,T "a",NT S]
S, 'ab' => [NT A,T "a",NT S]
S, 'b' => [T "b"]
S, 'c' => [NT A,T "a",NT S]
A, 'a' => [NT B]
A, 'ab' => [NT C,NT A,T "b"]
A, 'b' => [NT B]
A, 'c' => [NT B]
B, 'a' => [Eps]
B, 'b' => [Eps]
B, 'c' => [T "c",NT S,T "a"]
C, 'ab' => [T "ab"]
```

Рис. 6: Таблица разбора LL(1)-анализатора

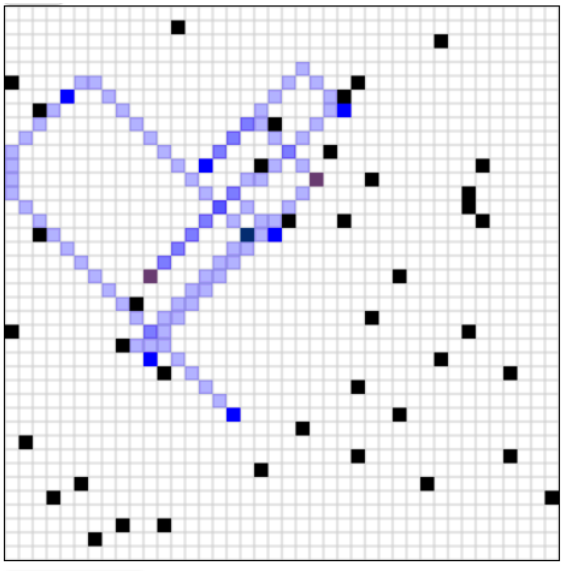

Рис. 8: Начало работы программы с цепочкой bbcbabbab

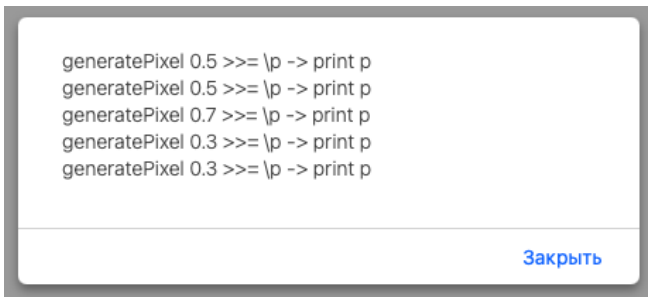

Рис. 9: Результат цепочки bbcbabbab в текстовом виде

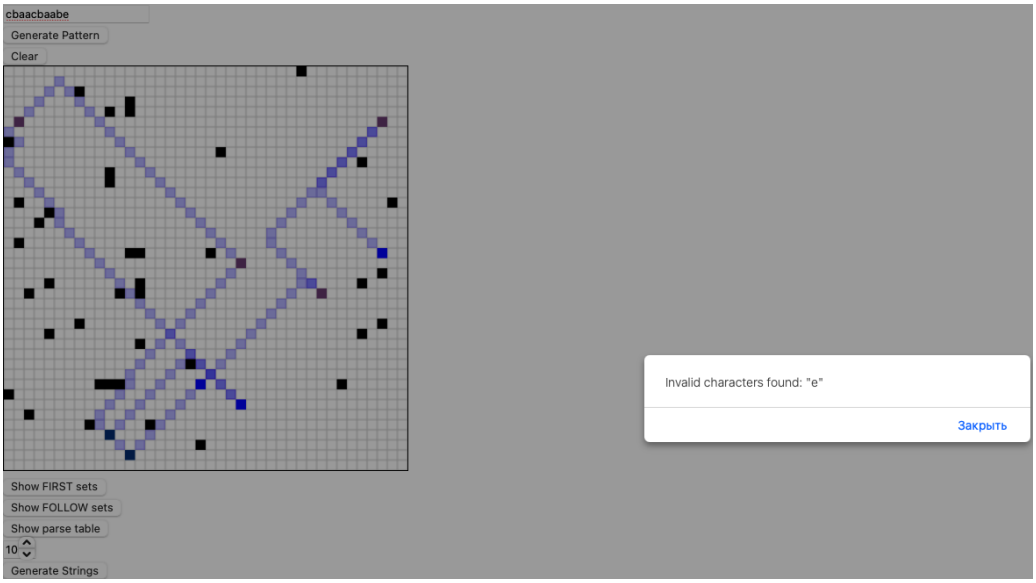

Рис. 10: Ошибка при вводе цепочки (cbaacbaab - верно, e - неверно)

Заключение

В данной лабораторной работе была рассмотрена контекстно-свободная грамматика (КС-грамматика) и реализован полный цикл построения LL(1)-анализатора для неё. Были вычислены множества **FIRST** и **FOLLOW**. На их основе сформирована таблица выбора (look-up table). Отсутствие пересечений между множествами выбора альтернатив одного нетерминала подтвердило, что грамматика принадлежит классу LL(1).

Каждой продукции назначено семантическое действие, интерпретируемое как генерация точек на сетке и поиск пути между ними: `generatePixel 0.3` для синих точек, `generatePixel 0.7` для зеленых и `generatePixel 0.5` для оранжевых точек, с последующим поиском кратчайшего пути между ними с использованием окрестности Мура. Для разбора был реализован детерминированный LL(1)-*предиктивный анализатор*.

В ходе работы был реализован LL(1) парсер для заданной грамматики.

- **Плюсы**:
  - Сохранена грамматика LL(1).
  - Разбирает входные цепочки и генерирует код для создания точек.
  - Визуализирует точки на сетке и находит кратчайший путь между ними.
- **Минусы**
  - Лексер тривиален: терминал `ab` должен вводиться как единая лексема, иначе разбор завершится ошибкой; для «настоящего» языка это потребует более гибкого токенизатора.
  - Пришлось редактировать грамматику для корректного разбора. Не был реализован парсер для LL(3).
- **Дополнения**
  1. Усовершенствовать поиск минимальних путей алгоритмом A*.
  2. Добавить возможность вводить иные грамматики.
  3. Добавить возможность добавления "черных"точек в соответствии с грамматикой.

Список литературы

1. Ю.Г. Карпов, Теория и технология программирования. Основы построения трансляторов. СПб.: БХВ-Петербург, 2005.
2. А.В. Востров, "Математическая логика и теория автоматов". Распознавание КС-языков и трансляция. [Электронный ресурс], URL: https://tema.spbstu.ru/userfiles/files/courses/2018-compilers (дата обращения: 14.05.2025)