ТФЯиТ. Coding

Николай Пономарев, группа 21.Б10-мм

5 ноября 2023 г.

Примеры работы программы на корректном вводе:

```
> cabal run coding -- examples/coding/expression.txt
11 1 2 12 3 5 2 51 3 4
12 1 2 5 2 101 52 3 13 102 3 5 2 2 103 53 6 103 52 3 3
 ↪  4
13 1 2 2 104 54 105 55 6 105 55 2 106 54 6 2 107 56 12
 ↪  108 57 3 5 6 109 58 9 11 110 10 111 59 6 112 3 6
 ↪  113 58 11 114 59 3 9 2 115 60 6 116 61 3 117 55 10
 ↪  6 105 62 6 118 63 6 119 64 6 120 65 3 5 2 121 66 11
 ↪  122 67 3 4
1000
```

Примеры работы программы на некорректном вводе:

```
> cabal run coding -- examples/coding/expression1.txt
examples/coding/expression1.txt:14:45:
    |
 14 |            )*( $cond1 '? expression $cond2 ':' )
  ↪  .
    |                                             ^
unexpected ':'
expecting '#', '$', ''', '(', ')', '*', ',', ';', '[',
 ↪  or terminal or nonterminal
```

Код программы разделен на три файла:

`CodingParser.hs`

```haskell
{-# LANGUAGE OverloadedStrings #-}
{-# LANGUAGE RecordWildCards #-}

module CodingParser where

import Data.Char (isAlphaNum, isPrint)
import qualified Data.Text as T
import Data.Text.Lazy (Text)
import Data.Void
import Text.Megaparsec hiding (State)
import Text.Megaparsec.Char
import qualified Text.Megaparsec.Char.Lexer as L

type Parser = Parsec Void Text

sc :: Parser ()
sc = L.space space1 empty empty

lexeme :: Parser a -> Parser a
lexeme = L.lexeme sc

symbol :: Text -> Parser Text
symbol = L.symbol sc

data Rule = Rule {nonterminal :: Text, productions ::
 ↪  [Production]}
    deriving (Eq, Show)

pIdentifier :: Maybe String -> Parser Text
pIdentifier msg = lexeme (takeWhile1P msg isAlphaNum)

data Production
    = ProdParentheses [Production]
    | ProdAsterisk
    | ProdSemicolon
    | ProdComma
    | ProdHash
    | ProdBrackets [Production]
    | ProdTerminal Text
    | ProdSemantic Text
    | ProdTerminalNonterminal Text
    deriving (Eq, Show)

isPrintNotBraces c = isPrint c && c /= '\''

pTerminal :: Parser Text
pTerminal = lexeme $ between (symbol "'") (symbol "'")
 ↪  (takeWhile1P Nothing isPrintNotBraces)

pSemantic :: Parser Text
pSemantic = lexeme $ symbol "$" *> pIdentifier (Just
 ↪  "semantic")

pNonterminal :: Parser Text
pNonterminal = pIdentifier (Just "nonterminal")

pProduction :: Parser Production
pProduction =
    choice
        [ ProdParentheses <$> between (symbol "(")
         ↪  (symbol ")") (many pProduction)
        , ProdBrackets <$> between (symbol "[") (symbol
         ↪  "]") (many pProduction)
        , ProdAsterisk <$ symbol "*"
        , ProdSemicolon <$ symbol ";"
        , ProdComma <$ symbol ","
        , ProdHash <$ symbol "#"
        , ProdTerminalNonterminal <$> pIdentifier (Just
         ↪  "terminal or nonterminal")
        , ProdTerminal <$> pTerminal
        , ProdSemantic <$> pSemantic
        ]

pProductions :: Parser [Production]
pProductions = many pProduction

pRule :: Parser Rule
pRule = do
    nonterminal <- pNonterminal
    _ <- symbol ":"
```

```haskell
    productions <- pProductions
    _ <- symbol "."
    return Rule{..}

pRules :: Parser [Rule]
pRules = do
    rules <- manyTill pRule (lexeme (string "Eofgram"))
    _ <- eof
    return rules
```

Coding.hs

```haskell
{-# LANGUAGE OverloadedStrings #-}

module Coding where

import CodingParser
import Control.Monad.State
import qualified Data.HashMap.Strict as HashMap
import Data.Text.Lazy (Text)
import Data.Text.Lazy.Builder
import Text.Megaparsec hiding (State)
import Text.Megaparsec.Char

data CodingState = CodingState
    { nonterminalCount :: Int
    , nonterminalBound :: Int
    , terminalCount :: Int
    , terminalBound :: Int
    , semanticCount :: Int
    , semanticBound :: Int
    , nonterminals :: [Text]
    , nonterminalMap :: HashMap.HashMap Text Int
    , terminalMap :: HashMap.HashMap Text Int
    , semanticMap :: HashMap.HashMap Text Int
    }

encodeProduction :: Production -> State CodingState
 ↪   Builder
encodeProduction (ProdParentheses productions) = do
```

```
    innerProductionsWithState <- mapM
     ↪  (encodeProduction) productions
    let innerProductions = foldr (<>) ""
     ↪  innerProductionsWithState
    return $ fromLazyText "2 " <> innerProductions <>
     ↪  fromLazyText "3 "
encodeProduction ProdAsterisk = return $ fromLazyText
 ↪  "5 "
encodeProduction ProdSemicolon = return $ fromLazyText
 ↪  "6 "
encodeProduction ProdComma = return $ fromLazyText "7 "
encodeProduction ProdHash = return $ fromLazyText "8 "
encodeProduction (ProdBrackets productions) = do
    innerProductionsWithState <- mapM
     ↪  (encodeProduction) productions
    let innerProductions = foldr (<>) ""
     ↪  innerProductionsWithState
    return $ fromLazyText "9 " <> innerProductions <>
     ↪  fromLazyText "10 "
encodeProduction (ProdTerminal text) = do
    state <- get

    case HashMap.lookup text (terminalMap state) of
        Just num -> return $ fromString (show num ++ "
         ↪  ")
        Nothing -> do
            let num = terminalCount state
            put $ state{terminalCount = terminalCount
             ↪  state + 1, terminalMap = HashMap.insert
             ↪  text num (terminalMap state)}
            return $ fromString (show num ++ " ")
encodeProduction (ProdSemantic text) = do
    state <- get

    case HashMap.lookup text (semanticMap state) of
        Just num -> return $ fromString (show num ++ "
         ↪  ")
        Nothing -> do
            let num = semanticCount state
            put $ state{semanticCount = semanticCount
             ↪  state + 1, semanticMap = HashMap.insert
             ↪  text num (semanticMap state)}
            return $ fromString (show num ++ " ")
encodeProduction (ProdTerminalNonterminal text) = do
    state <- get

    if text `elem` nonterminals state
        then case HashMap.lookup text (nonterminalMap
         ↪  state) of
            Just num -> return $ fromString (show num
             ↪  ++ " ")
            Nothing -> do
                let num = nonterminalCount state
                put $
                    state{nonterminalCount =
                     ↪  nonterminalCount state + 1,
                     ↪  nonterminalMap = HashMap.insert
                     ↪  text num (nonterminalMap
                     ↪  state)}
                return $ fromString (show num ++ " ")
        else case HashMap.lookup text (terminalMap
         ↪  state) of
            Just num -> return $ fromString (show num
             ↪  ++ " ")
            Nothing -> do
                let num = terminalCount state
                put $ state{terminalCount =
                 ↪  terminalCount state + 1,
                 ↪  terminalMap = HashMap.insert text
                 ↪  num (terminalMap state)}
                return $ fromString (show num ++ " ")

encodeNonterminal :: Text -> State CodingState Builder
encodeNonterminal text = do
    state <- get
    case HashMap.lookup text (nonterminalMap state) of
        Just num -> return $ fromString (show num ++ "
         ↪  ")
        Nothing -> do
```

```
            let num = nonterminalCount state
            put $
                state
                    { nonterminalCount =
                     ↪  nonterminalCount state + 1
                    , nonterminalMap = HashMap.insert
                     ↪  text num (nonterminalMap state)
                    }
            return $ fromString (show num ++ " ")

encodeRule :: Rule -> State CodingState Builder
encodeRule rule = do
    state <- get
    nonterminalNum <- encodeNonterminal (nonterminal
     ↪  rule)
    productionsWithState <- mapM (encodeProduction)
     ↪  (productions rule)
    let productions = foldr (<>) ""
     ↪  productionsWithState

    return $
        nonterminalNum
            <> fromLazyText "1 "
            <> productions
            <> fromLazyText "4\n"

encodeRules :: [Rule] -> State CodingState Text
encodeRules rules = do
    rulesWithState <- mapM (encodeRule) rules
    let rules' = foldr (<>) "" rulesWithState
    return $ toLazyText $ rules' <> fromLazyText
     ↪  "1000\n"

codingWrapper :: [Rule] -> Text
codingWrapper rules =
    evalState
        (encodeRules rules)
        CodingState
            { nonterminalCount = 11
            , nonterminalBound = 51
            , terminalCount = 51
            , terminalBound = 101
            , semanticCount = 101
            , semanticBound = 151
            , nonterminals = map nonterminal rules
            , nonterminalMap = HashMap.fromList []
            , terminalMap = HashMap.fromList []
            , semanticMap = HashMap.fromList []
            }
```

Main.hs

```
{-# LANGUAGE OverloadedStrings #-}

module Main where

import Coding
import CodingParser (pRules)
import Control.Monad.State
import qualified Data.HashMap.Strict as HashMap
import Data.Text.Lazy (Text)
import qualified Data.Text.Lazy as T
import Data.Text.Lazy.IO
import Options.Applicative
import Text.Megaparsec hiding (State)
import Prelude hiding (putStrLn, readFile)

inputFile :: Parser String
inputFile = argument str (metavar "FILE" <> value
 ↪  "expression.txt" <> help "File to read")

coding :: String -> IO ()
coding file = do
    text <- readFile file
    let rules = runParser pRules file text
    case rules of
        Left bundle -> putStrLn $ T.pack
         ↪  (errorBundlePretty bundle)
        Right result -> putStrLn $ codingWrapper result

main :: IO ()
```

```haskell
main = coding =<< execParser opts
  where
    opts = info (inputFile <**> helper) (fullDesc <>
     ↪  header "Coding")
```