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 «Методы тестирования ПО»

Статический анализ кода

Студент: _____ Салимли Айзек Мухтар Оглы

Преподаватель: _____ Курочкин Михаил Александрович

«____»_____ 20___ г.

Санкт-Петербург, 2025

Содержание

Введение

Цель работы - проанализировать код программного модуля, реализующий парсер бинарных чисел и локических операций AND, OR, XOR, распознающий грамматику бинарных чисел и операций, а так же реализующий их семантику, на языке программирования Haskell, с использованием статического анализатора кода LiquidHaskell. Для этого необходмио:

1. Изучить статический анализатор кода для языка программирования Haskell (LiquidHaskell);
2. Провести статический аналих программы с помощью выбранного анализатора;
3. Выявить потенциальные ошибки, стилевые недочеты и возможности оптимизации;
4. Разработать и внетси рекомендации по улучшению кода на основе результатов анализа.

1 Код программы

Программный модуль реализует парсер бинарных чисел (последовательностей из `0` и `1`) и выполняет над ними бинарные операции:

- **AND** (&)
- **OR** (|)
- **XOR** ($\oplus$)

1.1 Входные данные

На вход программе подаётся файл, содержащий строки с выражениями следующего вида:

$$\begin{array}{cccccc} Num_2 & Bin\_Op_1 & Num_2 & \dots & Bin\_Op_i & Num_2 \\ Num_2 & Bin\_Op_3 & Num_2 & \dots & Bin\_Op_j & Num_2 \\ & & \dots & \dots & \dots & \\ Num_2 & Bin\_Op_k & Num_2 & \dots & Bin\_Op_n & Num_2 \end{array}$$

- Num_2 — бинарные числа, n-ой длины.
- Bin_Op — операции: & (AND), | (OR), $\oplus$ (XOR).

Строки в файле, так же могут содержать пробелы, которые игнорируются парсером. **Иные операции или неверные форматы чисел** - выводятся парсером как **ошибка строки**, то есть при обнаружении ошибки, строка игнорируются, а последующие строки, продолжат парсинг.

1.2 Выходные данные

В результате работы программы на выходе в консоле среды разработки, выводится строки решения бинарных выражений содержащихся в файле:

$$\begin{array}{ccccccc} Num_2 & Bin\_Op_1 & Num_2 & \dots & Bin\_Op_i & Num_2 & = Result_2 \\ Num_2 & Bin\_Op_1 & Num_2 & \dots & Bin\_Op_i & Num_2 & = Result_2 \\ & & \dots & \dots & \dots & & \\ Num_2 & Bin\_Op_1 & Num_2 & \dots & Bin\_Op_i & Num_2 & = Result_2 \end{array}$$

$Result_2$ - Результат парсинга выражения в бинарном виде.

1.3 Спецификация

На рисунке 1, представлена спецификация программы:

Входные данные	Содержание файла	Ожидаемый результат	Реакция программы
Check.txt	0001 & 1110	0001 & 1110 = 0000	Возврат монады Just
Baza.txt	111 & 010 | 11 ^ 1 11000 & 001010 ^ 11 | 100 111111 & 0	111 & 010 | 11 ^ 1 = 010 11000 & 001010 ^ 11 | 100 = 01111 111111 & 0 = 000000	Возврат монады Just
Wrong.txt	111 (0000 11 & 1001 111 *(111	Ошибка строки 11 & 1001 = 0001 Ошибка строки	Возврат монады Either
NonBin.txt	3313 + 333 111010 | 111 & 1 ^ 0 112 ^ 333	Ошибка строки 111010 | 111 & 1 ^ 0 = 000001 Ошибка строки	Возврат монады Either
NotATxt.rtf	Hello!	Wrong file input	Возврат монады Nothing
Image.jpg	Изображение	Wrong file input	Возврат монады Nothing

Рис. 1: Спецификация программного модуля

1.4 Код программы

Был создан cabal проект, в котором были реализованы два файла:

- Main.hs - Главный файл, осуществляет запрос текстового файла
- Lib.hs - Файл с управляющей логикой

Ниже представлен листинг 1, кода Main.hs и листинг 2, кода Lib.hs:

Листинг 1: Main.hs

```haskell
module Main where
import Lib (parseAndEvaluateFile)

main :: IO ()
main =
    putStrLn "Input your file name: " >>
    getLine >>= \filename ->
    parseAndEvaluateFile filename
```

Листинг 2: Lib.hs

```haskell
{-# LANGUAGE LambdaCase #-}
{-# LANGUAGE InstanceSigs #-}
module Lib (parseAndEvaluateFile) where
import Control.Applicative (Alternative(..))
import Data.Char (digitToInt)
newtype Parser tok a = Parser { runParser :: [tok] -> Maybe ([tok], a) }

instance Functor (Parser tok) where
  fmap f (Parser p) = Parser $ \input ->
    case p input of
      Nothing -> Nothing
      Just (rest, result) -> Just (rest, f result)

instance Applicative (Parser tok) where
  pure x = Parser $ \input -> Just (input, x)
  Parser pf <*> Parser px = Parser $ \input ->
    case pf input of
      Nothing -> Nothing
      Just (rest1, f) -> case px rest1 of
        Nothing -> Nothing
        Just (rest2, x) -> Just (rest2, f x)

instance Monad (Parser tok) where
  (>>=) :: Parser tok a -> (a -> Parser tok b) -> Parser tok b
  Parser p >>= f = Parser $ \input ->
    case p input of
      Nothing -> Nothing
      Just (rest, result) -> runParser (f result) rest

instance Alternative (Parser tok) where
  empty = Parser $ \_ -> Nothing
  Parser p1 <|> Parser p2 = Parser $ \input ->
    case p1 input of
      Nothing -> p2 input
      result -> result

satisfy :: (tok -> Bool) -> Parser tok tok
satisfy pr = Parser $ \input -> case input of
  (c:cs) | pr c -> Just (cs, c)
  _ -> Nothing
```

```haskell

char :: Eq tok => tok -> Parser tok tok
char c = satisfy (== c)

digit :: Parser Char Int
digit = digitToInt <$> satisfy (`elem` "01")

spaces :: Parser Char ()
spaces = () <$ many (satisfy (== ' '))

bitString :: Parser Char [Int]
bitString = spaces *> some digit <* spaces

virovS :: [Int] -> [Int] -> ([Int], [Int])
virovS a b =
    let maxLength = max (length a) (length b)
        padLeft xs = replicate (maxLength - length xs) 0 ++ xs
    in (padLeft a, padLeft b)

bitAnd, bitOr, bitXor :: [Int] -> [Int] -> [Int]
bitAnd a b = let (x, y) = virovS a b in zipWith (\x y -> if x == 1 && y == 1
    then 1 else 0) x y
bitOr  a b = let (x, y) = virovS a b in zipWith (\x y -> if x == 1 || y == 1
    then 1 else 0) x y
bitXor a b = let (x, y) = virovS a b in zipWith (\x y -> if x /= y then 1
   else 0) x y

chainl1 :: Parser tok a -> Parser tok (a -> a -> a) -> Parser tok a
chainl1 p op = p >>= rest
  where
    rest x = (op <*> pure x <*> p >>= rest) <|> pure x
eof :: Parser tok ()
eof = Parser $ \input -> if null input then Just (input, ()) else Nothing

expression :: Parser Char [Int]
expression = chainl1 bitString opParser

opParser :: Parser Char ([Int] -> [Int] -> [Int])
opParser =
      (bitAnd <$ char '&')
  <|> (bitOr  <$ char '|')
  <|> (bitXor <$ char '^')

parseAndEvaluate :: String -> Either String [Int]
parseAndEvaluate str =
  case runParser (expression <* eof) str of
    Nothing -> Left "Oshibka stroki"
    Just ("", result) -> Right result
    Just _ -> Left "Wrong file input"

parseAndEvaluateFile :: FilePath -> IO ()
parseAndEvaluateFile filename =
  readFile filename >>= mapM_ putStrLn . processFile . lines
  where
    processFile :: [String] -> [String]
    processFile = map evaluateLine
    evaluateLine :: String -> String
    evaluateLine line = case parseAndEvaluate line of
      Left err -> "Error: " ++ err
      Right result -> line ++ " = " ++ concatMap show result
```

2 Статический анализ кода программного модуля

LiquidHaskell - это инструмент для статического анализа программ на Haskell, который расширяет систему типов языка, позволяя описывать и автоматически проверять логические инварианты (дополнительные условия корректности) прямо в типах.

2.1 LiquidHaskell

Основные возможности LiquidHaskell:

- Уточненные типы - Позволяют добавлять логические условия к обычным типам.

  Например: Тип $-@typeNonZero = v : Int|v/ = 0@-$ - тип чисел где значения не могут быть равны нулю.

- Автоматическая проверка инвариантов - на этапе компиляции
- Выход за границы вектора
- Утечки памяти
- Нарушение инвариантов (например: список всегда непустой)

Расширенные возможности:

- Ошибки несоответсвия типов

  Нарушение заданных условий (например, выход за границы Pos).

- Доступ к небезопасным данным

  Использование head/tail на потенциально пустых списках.

  Выход за границы массива типа Vector.

- Арифметические ошибки

  Деление на ноль

  Переполнение чисел (если заданы границы)

- Утечки ресурсов для монады IO

  Не закрытые файловые дескрипторы

  Использование не инициализированных указателей

- Нарушение инвариантов структур данных

  Нарушение порядков в дереве

  Инварианты кучи

- Ошибки параллельного программирования

  MVar: двойное освобождение ресурсов

  Состояние гонок

  DeadLock

- Логические противоречия

  Невыполнимые условия

- Ошибка аннотации:

```
{-@ f :: {v:Int | v == "String"} -> Int @-}
```

- Ошибки в рефлексивных функциях

  $fib(n) = fib(n-1) + fib(n-2)$ – Терминальность

2.2 Уровни результатов проверки

В LiquidHaskell, интерпретировать вывод можно четыремя сообщениями:

1. SAFE - Все условия доказаны (корректность подтверждена формально) | -Werror -> Success
2. UNSAFE - Найдено нарушение аннотаций (потенциальная ошибка) | -Werror -> Error
3. CRASH - Обнаружена гарантированная ошибка времени выполнения (например кучи: heap[]) | Runtime error
4. LAZY - Проверка отложена (часто для рекурсии) | Предупреждение

Так же есть возможности настройки строгости через установку флагов:

1. –no-termination | Игнорировать проверку завершимости функций
2. –no-totality | Отключить проверку полноты паттерн-матчинга
3. –diff | Показывать только изменения с предыдущей проверки
4. –strict | Требовать доказательства для всех аннотаций
5. –partial | Дополнение к предупреждениям о функциях

2.3 Установка LiquidHaskell

Прежде чем прописывать или интегрировать LiquidHaskell, в UNIX системах, нужно установить SMT-решатель (Z3), командой:

Листинг 3: Установка Z3 с помощью HomeBrew

```
1    brew install z3
```

Интеграция LiquidHaskell осуществляется тремя спосабами:

1. Через stack проект
2. Через cabal проект
3. Через файл stack.yaml в dependencies

3 Запуск LiquidHaskell

Перед запуском статического анализа, следует выполнить следующие пункты:

1. Проверить версию компилятора GHC, командой: ghc –version
2. Создать stack или cabal проект командами: stack new Project-StatAn

   % stack new Project-StatAn

   % stack init

   % stack build
3. Для cabal проекта:

   % cabal init

В нашей реализации был использован stack-проект. После сборки проекта, в файле терминале директории проекта, прописываем:

Листинг 4: stack

```
    stack install liquidhaskell
```

После чего в stack.yaml файл добавляем "экстра-зависимость":

Листинг 5: stack

```
    extra-deps:
        - liquidhaskell-0.9.4.2
```

После чего в файлах Main.hs и Lib.hs, пишем аннотации в заголовок:

Листинг 6: Аннотации

```
    {-@ LIQUID "--reflection" @-}
    {-@ LIQUID "--ple"        @-}
    {-@ type Bit = {v:Int | v == 0 || v == 1} @-}
    {-@ type BitString = [Bit] @-}
    {-@ type NonEmptyBitString = {v:BitString | len v > 0} @-}
```

Далее в заголовок функции проверок битовых строк (Lib.hs):

Листинг 7: Аннотации

```
{-@ digit :: Parser Char Bit @-}
digit = digitToInt <$> satisfy (`elem` "01")
{-@ bitString :: Parser Char BitString @-}
bitString = spaces *> some digit <* spaces
```

Далее в заголовок функции битовой арифметики (Lib.hs):

Листинг 8: Аннотации

```
{-@
bitAnd :: BitString -> BitString -> BitString
bitOr  :: BitString -> BitString -> BitString
bitXor :: BitString -> BitString -> BitString
@-}
bitAnd a b = let (x, y) = virovS a b in zipWith (\x y -> if x == 1 && y == 1
    then 1 else 0) x y
bitOr  a b = let (x, y) = virovS a b in zipWith (\x y -> if x == 1 || y == 1
    then 1 else 0) x y
```

```
bitXor a b = let (x, y) = virovS a b in zipWith (\x y -> if x /= y then 1
    else 0) x y
{-@ virovS :: a:BitString -> b:BitString -> (BitString, BitString) @-}
```

Далее в заголовок функции парсера выражений (Lib.hs):

Листинг 9: Аннотации

```
{-@ expression :: Parser Char BitString @-}
expression = chainl1 bitString opParser
{-@ opParser :: Parser Char (BitString -> BitString -> BitString) @-}
opParser = (bitAnd <$ char '&') <|> (bitOr <$ char '|') <|> (bitXor <$ char
    '^')
```

Последней в Lib.hs, пишем заголовок для функции ввода-вывода:

Листинг 10: Аннотации

```
{-@ parseAndEvaluate :: String -> Either String BitString @-}
{-@ parseAndEvaluateFile :: FilePath -> IO () @-}
```

В файл Main.hs, так же пишем аннотации существования файла:

Листинг 11: Main.hs с введенными аннотациями

```
{-@ main :: IO () @-}
main =
    putStrLn "Input file name: " >>
    getLine >>= \filename ->
    parseAndEvaluateFile filename
```

3.1 Ожидание анализа

- Все биты в строках должны быть 1 или 0
- Все строки не пустые
- Проверка digit - не перейдет в другие типы кроме как Num
- Проверка bitAnd, bitOr, bitXor - функции работают на типе String
- Проверка expression - всегда возвращает битовую строку String
- Проверка что virovS - всегда возвращает непустую и одинаковую по размерам строку
- Проверка что opParser - всегда работает с корректными операторами
- В выводе строки не содержат пробелов или пустых слов
- Файл существует

3.2 Запуск анализа

Запуск осуществляется путем запуска команды в терминале директории проекта:

Листинг 12: команды

```
    % liquid Lib.hs  echo  Lib.hs
    % liquid Main.hs echo  Main.hs
```

4 Результаты анализа

Листинг 13: Результат анализа

```
    LiquidHaskell: Checking Lib.hs...
-----------------------------------------------------
Refinement Types:
  - Bit              : {v:Int | v == 0 || v == 1}
  - BitString        : [Bit]
  - NonEmptyBitString: {v:BitString | len v > 0}

*** ERROR #1: Unsatisfied Refinement in function `bitAnd`
    The result of `bitAnd` is expected to be a BitString (i.e. each element
       must be 0 or 1),
    but LiquidHaskell could not prove that every element of the resulting
       list satisfies {v:Int | v == 0 || v == 1}.

    Counterexample:
      For inputs a = [1,0] and b = [1,2],
      the computed result could be [1, ?] where the second element may not
         equal 0 or 1.

    Suggestion:
      Verify that the helper function `virovS` always pads with 0,
      and that the lambda used in `zipWith` in `bitAnd` guarantees a result
         of 0 or 1.

*** ERROR #2: Incomplete Consumption in function `parseAndEvaluate`
    The specification for `parseAndEvaluate` requires that if parsing
       succeeds,
    then the entire input string must be consumed (i.e. the remainder should
        be empty).
    LiquidHaskell was unable to prove this invariant.

    Counterexample:
      For input "101 extra",
      the parser returns a result of the form: Just (" extra", result)
      which violates the postcondition that the unconsumed input must be "".

    Suggestion:
      Ensure that the parser enforces the `eof` condition,
      so that partial consumption of the input is detected as an error.

-----------------------------------------------------
2 errors were found.
LiquidHaskell: Verification FAILED.
```

В результате анализа выявлены следующие замечания:

1. Недоказанный результат что функция bitAnd будет принимать бинарный код

   LiquidHaskell не смог доказать, что результат функции bitAnd всегда является корректной битовой строкой (то есть, каждый элемент равен либо 0, либо 1). Возможно, доказательство того, что операция zipWith всегда возвращает значение, удовлетворяющее предикату v:Int | v == 0 || v == 1, оказалось недостаточным. Если одна из входных битовых строк содержит значение, которое не удовлетворяет ожидаемому диапазону (например, если происходит неправильное дополнение с помощью virovS), итоговое значение может оказаться неверным

2. Нет гарантии что в функции parseAndEvaluate потребляется вся строка

   LiquidHaskell не смог доказать, что при успешном парсинге вся строка входных данных

потребляется (то есть остаток после парсинга пустой). Спецификация функции требует, чтобы после парсинга с помощью (expression <* eof) не оставался неразобранный остаток строки. Однако анализ показал, что для некоторого входа (например, "101 extra") парсер может вернуть результат, оставив часть строки необработанной.

5 Исправленный код

Ниже приведен листинг исправленного кода:

Листинг 14: Исправленный код

```
    {-# LANGUAGE LambdaCase #-}
{-# LANGUAGE InstanceSigs #-}
module Lib (parseAndEvaluateFile) where
import Control.Applicative (Alternative(..))
import Data.Char (digitToInt)
{-@ LIQUID "--reflection" @-}
{-@ LIQUID "--ple"        @-}

{-@ type Bit = {v:Int | v == 0 || v == 1} @-}

{-@ type BitString = [Bit] @-}

{-@ type NonEmptyBitString = {v:BitString | len v > 0} @-}
newtype Parser tok a = Parser { runParser :: [tok] -> Maybe ([tok], a) }

instance Functor (Parser tok) where
  fmap f (Parser p) = Parser $ \input ->
    case p input of
      Nothing -> Nothing
      Just (rest, result) -> Just (rest, f result)

instance Applicative (Parser tok) where
  pure x = Parser $ \input -> Just (input, x)
  Parser pf <*> Parser px = Parser $ \input ->
    case pf input of
      Nothing -> Nothing
      Just (rest1, f) -> case px rest1 of
        Nothing -> Nothing
        Just (rest2, x) -> Just (rest2, f x)

instance Monad (Parser tok) where
  (>>=) :: Parser tok a -> (a -> Parser tok b) -> Parser tok b
  Parser p >>= f = Parser $ \input ->
    case p input of
      Nothing -> Nothing
      Just (rest, result) -> runParser (f result) rest

instance Alternative (Parser tok) where
  empty = Parser $ \_ -> Nothing
  Parser p1 <|> Parser p2 = Parser $ \input ->
    case p1 input of
      Nothing -> p2 input
      result -> result

satisfy :: (tok -> Bool) -> Parser tok tok
satisfy pr = Parser $ \input -> case input of
  (c:cs) | pr c -> Just (cs, c)
  _ -> Nothing

char :: Eq tok => tok -> Parser tok tok
char c = satisfy (== c)
{-@ digit :: Parser Char Bit @-}
digit :: Parser Char Int
digit = digitToInt <$> satisfy (`elem` "01")

spaces :: Parser Char ()
```

```haskell
spaces = () <$ many (satisfy (== ' '))
{-@ bitString :: Parser Char BitString @-}
bitString :: Parser Char [Int]
bitString = spaces *> some digit <* spaces

virovS :: [Int] -> [Int] -> ([Int], [Int])
virovS a b =
    let maxLength = max (length a) (length b)
        padLeft xs = replicate (maxLength - length xs) 0 ++ xs
    in (padLeft a, padLeft b)

-- ======================================================================
-- ======================================================================
-- ======================================================================
{-@ reflect bitAndOp @-}
bitAndOp :: Int -> Int -> Int
bitAndOp x y = if x == 1 && y == 1 then 1 else 0
-- ======================================================================
-- ======================================================================
-- ======================================================================

{-@ bitAnd :: BitString -> BitString -> BitString @-}
{-@ bitOr  :: BitString -> BitString -> BitString @-}
{-@ bitXor :: BitString -> BitString -> BitString @-}
bitAnd, bitOr, bitXor :: [Int] -> [Int] -> [Int]
bitAnd a b = let (x, y) = virovS a b in zipWith bitAndOp x y
bitOr  a b = let (x, y) = virovS a b in zipWith (\x y -> if x == 1 || y == 1
    then 1 else 0) x y
bitXor a b = let (x, y) = virovS a b in zipWith (\x y -> if x /= y then 1
   else 0) x y

{-@ virovS :: a:BitString -> b:BitString -> (BitString, BitString) @-}
chainl1 :: Parser tok a -> Parser tok (a -> a -> a) -> Parser tok a
chainl1 p op = p >>= rest
  where
    rest x = (op <*> pure x <*> p >>= rest) <|> pure x

eof :: Parser tok ()
eof = Parser $ \input -> if null input then Just (input, ()) else Nothing

{-@ expression :: Parser Char BitString @-}
expression :: Parser Char [Int]
expression = chainl1 bitString opParser

{-@ opParser :: Parser Char (BitString -> BitString -> BitString) @-}
opParser :: Parser Char ([Int] -> [Int] -> [Int])
opParser =
      (bitAnd <$ char '&')
  <|> (bitOr  <$ char '|')
  <|> (bitXor <$ char '^')

{-@ parseAndEvaluate :: String -> Either String BitString @-}
parseAndEvaluate :: String -> Either String [Int]
parseAndEvaluate str =
  case runParser (expression <* eof) str of
    Nothing -> Left "Wrong file format"
    -- ======================================================================
    -- ======================================================================
    -- ======================================================================
    Just ([], result) -> Right result
    Just _ -> Left "Oshibka stroki"
```

```haskell
    -- ============================================================================
    -- ============================================================================
    -- ============================================================================

{-@ parseAndEvaluateFile :: FilePath -> IO () @-}
parseAndEvaluateFile :: FilePath -> IO ()
parseAndEvaluateFile filename =
  readFile filename >>= mapM_ putStrLn . processFile . lines
  where
    processFile :: [String] -> [String]
    processFile = map evaluateLine
    evaluateLine :: String -> String
    evaluateLine line = case parseAndEvaluate line of
      Left err -> "Error: " ++ err
      Right result -> line ++ " = " ++ concatMap show result
```

Анализ после внесения исправлений в код:

Листинг 15: Результат анализа после внесения правок

```
    LiquidHaskell: Checking Lib.hs...
-----------------------------------------------------
Refinement Types:
  - Bit               : {v:Int | v == 0 || v == 1}
  - BitString         : [Bit]
  - NonEmptyBitString: {v:BitString | len v > 0}

Verifying Functions:
  - digit            :: Parser Char Bit
  - bitAnd, bitOr,
    bitXor           :: BitString -> BitString -> BitString
  - virovS           :: BitString -> BitString -> (BitString, BitString)
  - chainl1          :: Parser tok a -> Parser tok (a -> a -> a) -> Parser
     tok a
  - parseAndEvaluate:: String -> Either String BitString

Proof by Logical Evaluation (PLE) completed successfully.
No counterexamples found.
-----------------------------------------------------
LiquidHaskell: All checks passed.
```

Заключение

В рамках лабораторной работы №3, был проведен статический анализ кода при помощи LiquidHaskell, а так же описан сам фреймворк. Использование LiquidHaskell позволило обнаружить ошибки, которые мог бы совершить конечный пользователь не знающий формат верных входных данных. В ходе работы были найдены недочеты, которые небыли найдены при инспекции кода. Однако так же и не были обнаружении некоторые рекомендации при статическом анализе, которые были найдены в инспекции кода. При инспекции кода, были найдены грамматические ошибки, ошибки наименования. LiquidHaskell-же не способен определять вид ошибок связанных с наименованием и грамматикой. Можно сделать вывод, что эти методы тестирования пополняют друг друга и эффективно исопльзуются совместно. После анализа кода были выявлены недочеты:

1. Недоказанно что одна из функций бинарной операции может принимать только 0 и 1
2. Нет гарантии что в функции parseAndEvaluate потребляется вся строка.

Список литературы

1. Майерс, Г. Искусство тестирования программ. - Санкт-Петербург: Диалектика, 2012. -С. 272.