

Колледж космического машиностроения и технологий

КУРСОВОЙ ПРОЕКТ

По МДК.01.02 «Прикладное программирование»

Тема: «Разработка генератора блок-схем»

Выполнил студент

Богомолов Максим Вадимович

Группа П1-17

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_(Подпись)

\_\_\_\_\_\_\_\_\_\_\_ (Дата сдачи работы)

Королёв 2020 г.

# Оглавление

[Введение 4](#_Toc43825340)

[Глава 1. Теоретическая часть 5](#_Toc43825341)

[1.1 Изучение предметной области 5](#_Toc43825342)

[1.2 Изучение существующих разработок. 6](#_Toc43825343)

[Глава 2. Проектная часть 9](#_Toc43825344)

[2.1 Построение диаграммы прецедентов 9](#_Toc43825345)

[2.2 Выбор инструментов 10](#_Toc43825346)

[2.3 Проектирования сценария 11](#_Toc43825347)

[2.4 Диаграмма типов 12](#_Toc43825348)

[2.5 Описание главного модуля 13](#_Toc43825349)

[2.6 Описание спецификаций к модулям 16](#_Toc43825350)

[2.7 Описание модулей 17](#_Toc43825351)

[2.8 Описание тестовых наборов модулей 23](#_Toc43825352)

[2.9 Описание применения средств отладки 24](#_Toc43825353)

[2.10 Анализ оптимальности использования памяти и быстродействия 25](#_Toc43825354)

[Глава 3. Эксплуатационная часть 26](#_Toc43825355)

[3.1 Руководство оператора 26](#_Toc43825356)

[3.2 To-Do лист 28](#_Toc43825357)

[Заключение 29](#_Toc43825358)

[Список литературы и интернет-источников 30](#_Toc43825359)

# Введение

Целью данного курсового проекта является написание программы “Генератор блок-схем ”. Эта программа будет полезна, потому что блок-схемы сейчас используются везде, программистами любого уровня навыков. Как первый поддерживаемый язык был выбран Python на основании его популярности и простоты использования, тем самым эта программа будет полезна для большего числа людей Данная программа выполняется как консольная команда, с несколькими дополнительными параметрами. Это облегчает использование программы и позволяет программистам создавать блок-схемы не замедляя рабочий процесс.

В первой части будет рассмотрена предметная область данной темы, а также несколько аналогичных разработок по данной теме.

Во второй части будут рассмотрены инструменты, которые были использованы для создания программы, модули, которые были разработаны с их помощью, структура программы и листинги ключевых частей программных модулей.

В третьей части будет рассмотрено руководство для пользователей, а также приведен To-do лист с планами по доработке программы.

В заключительной части будут сделаны общие выводы о получившемся проекте.

# Глава 1. Теоретическая часть

## Изучение предметной области

Полезность блок-схем очевидна: они позволяют увидеть алгоритм в удобном для человека виде и понять все его нюансы и детали, даже если человек не знаком с языком программирования. С их помощью проще вносить изменения в уже готовый алгоритм. Именно поэтому умение чертить блок схемы является таким важным для любого программиста. Тем не менее рисование блок-схем требует траты сил и времени, и с помощью такой программы можно еще больше увеличить эффективность блок-схем. Эта программа также позволит создать блок-схему человеку не знакомому с программированием.

Интерфейс такой не нуждается в большом функционале, так как основное назначение программы: на основе текста произвести картинку. Соответственно для максимальной эффективности и сохранения простоты использования достаточно поля для ввода и поля с результатом.

Цели использования генератора блок-схем:

1. Автоматизация построения блок-схем по исходному коду
2. Единый стандарт блок-схем в проекте
3. Возможность видеть алгоритм проекта даже без знания языков программирования

Основные требования:

1. Простота использования
2. Высокая скорость работы

В больших проектах бывает очень много исходного кода написанного разными людьми. Часто для написания нового модуля или ввода дополнительного функционала бывает необходимо подробно изучить исходный код всего проекта, что значительно замедляет процесс разработки программы. С помощью блок-схем можно изучить алгоритм модуля в удобном для человека виде и значительно повысить скорость разработки и качество программы. А благодаря автоматизации построения блок-схем можно убрать временные затраты на их создание и предотвратить неправильно сделанные блок-схемы.

## Изучение существующих разработок.

* + 1. Flowchart

Программа “Flowchart” [8] – программа строящая блок-схемы по исходному кода на Pascal, C++ или по псевдокоду.

Достоинства данной программы:

1. Поддержка нескольких языков программирования и псевдокода
2. Настройка цветов блоков схемы и прочих параметров
3. Сохранение скриншота блок-схемы в форматах BMP, PNG и JPEG
4. Подсветка синтаксиса исходников

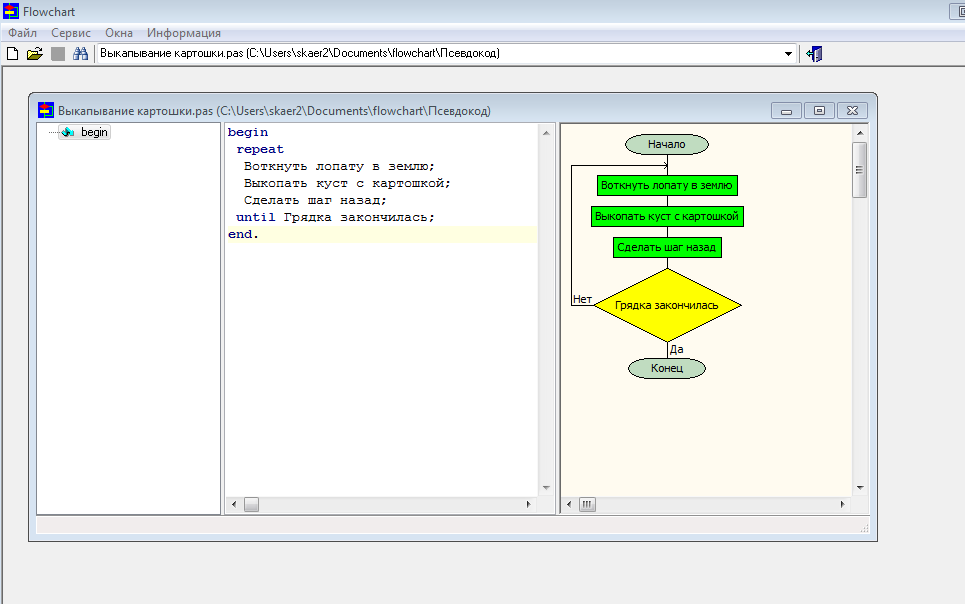


Рисунок 1 Интерфейс программы "Flowchart"

Эта программа позволяет легко создать блок-схемы по исходному коду, с настраиваемым оформлением и редактором кода в самой программе. Благодаря интерфейсу программы с ней смогут разобраться даже начинающие пользователи ПК. Возможность экспорта в несколько форматах делает программу универсальной. Тем не менее у программы есть и минусы: многооконность хоть и позволяет работать с несколькими файлами одновременно но не делает это удобным. Если целью является построение блок-схем для большого количества файлов, то программа становится неэффективна и использовать её становиться затруднительно.

* + 1. Visustin

Программа “Visustin” [9] — это генератор блок-схем для разработчиков программного обеспечения. Позволяет преобразовать исходный код в блок-схемы и диаграммы деятельности UML в автоматическом режиме. Рисуйте блок-схемы мышью. Возможно просматривать и распечатывать блок-схемы или экспортировать их в Visio.

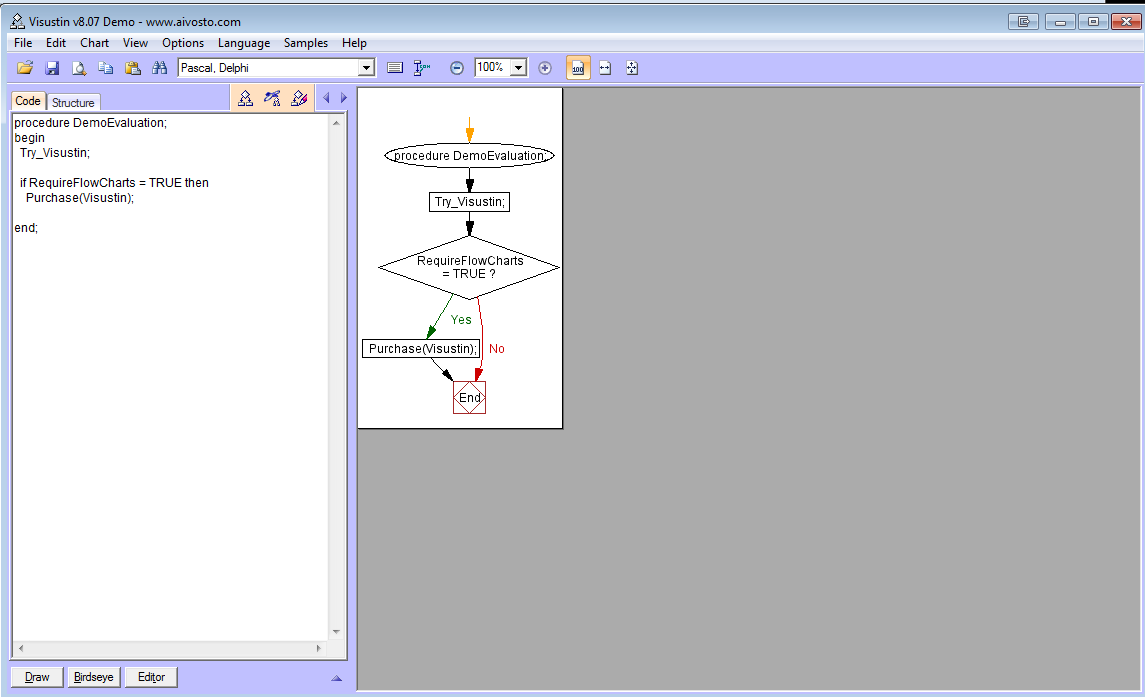


Рисунок 2 Интерфейс программы "Visustin"

Достоинства данной программы:

1. Данная программа поддерживает большое количество языков
2. Возможность распечатывать блок-схемы и экспортировать в разные форматы
3. Возможность редактировать сгенерированную блок-схему
4. Возможность рисовать блок-схемы в ручную

Это автоматизированная программа создания блок-схем для разработчиков программ и авторов документов. Visustin выполняет обратный инжиниринг исходного кода для его разбиения на блок-схемы или диаграммы деятельности UML (Activity Diagram). Visustin считывает операторы if и else, операторы цикла и операторы перехода и создает блок-схему — в полностью автоматическом режиме. Данная программа также позволяет выбирать на какие процедуры нужно генерировать блок-схему. Имеет встроенный тестовый редактор. Позволяет выбрать цветовую схему, шрифт.

Программа имеет сложный интерфейс.

Данная программа является платной.

# Глава 2. Проектная часть

## Построение диаграммы прецедентов

В этом разделе представлена диаграмма прецедентов. На диаграмме показаны все возможные функциональные отношения.

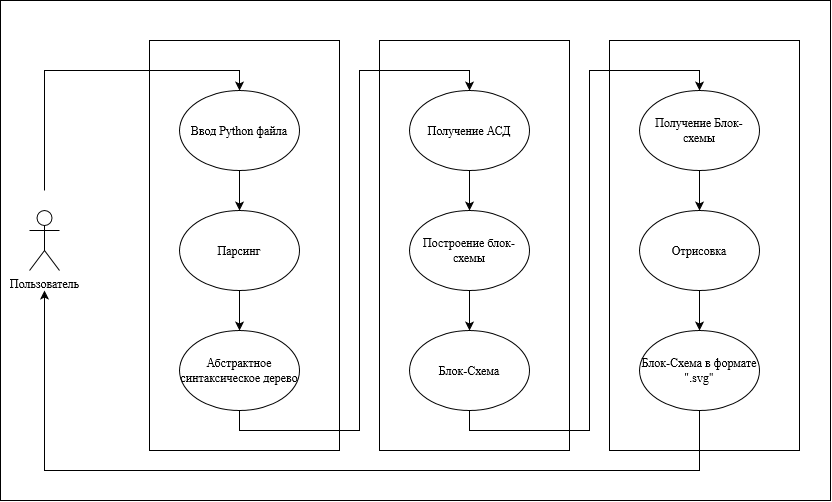


Рисунок 3 Диаграмма прецедентов работы генератора блок-схем

## Выбор инструментов

При выборе инструментов я исходил из некоторых критериев.

Важность критерия выбрана из: низкая, ниже средней, средняя, ниже высокой, высокая

Таблица 1

Критерии выбора инструментов

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Критерий | Участие в корпоративном проекте | Простота сопровождения | Наличие библиотек | Понятный синтаксис | Скорость разработки |
| Важность критерия | Ниже средней | Выше Средней | Средняя | Высокая | Высокая |

Исходя из этих критериев, я сравнил 3 языка программирования от 0 до 10 баллов за критерий.

Таблица 2

Сравнение языков программирования

|  |  |  |  |
| --- | --- | --- | --- |
| Критерий/Язык программирования | C++ | Python | Haskell |
| Участие в корпоративном проекте | 5 | 3 | 10 |
| Простота сопровождения | 2 | 2 | 10 |
| Наличие библиотек | 10 | 9 | 8 |
| Понятный синтаксис | 3 | 8 | 10 |
| Скорость разработки | 1 | 1 | 10 |
| Итого баллов | 21 | 23 | 48 |

По результатам сравнения был выбран язык программирования Haskell.

## Проектирования сценария

В данном разделе приведен сценарий использования программы пользователем.

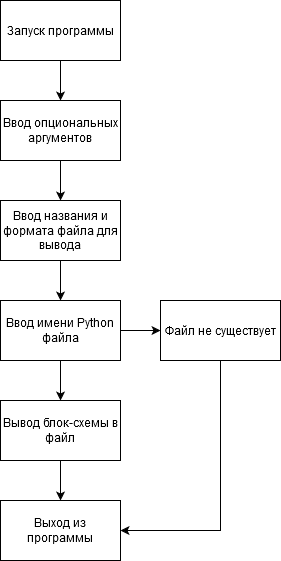


Рисунок 4 Сценарий использования

Пользователь вводит необязательные аргументы программы, например ширину и/или высоту конечного изображения. Дальше вводится имя файла и формат для вывода изображения и имя файла с исходным кодом. После этого программа проверяет существование указанного файла, если файл не существует, выводится ошибка. Если файл существует то программа выводит блок-схему в указанный файл, в выбранном формате.

## Диаграмма типов

Здесь представлены все типы данных описанные в проекте.

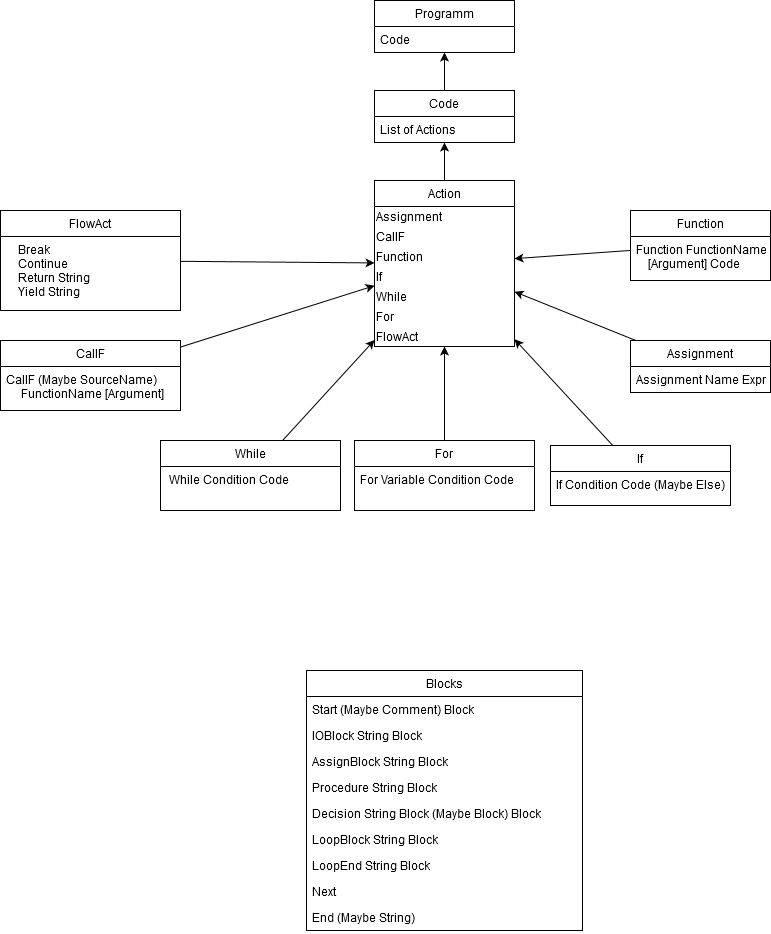


Рисунок 5 Диаграмма типов проекта

## Описание главного модуля

В главный модуль подключаются 3 основных модуля: модуль с парсером, модуль с типом данных блок-схемы, и модуль с отрисовкой блок-схемы.

Также туда подключается вспомогательный модуль с универсальными функциями не зависящими от остальных модулей программы.

Далее в главном модуле считывается файл, производится его парсинг, и в случае успешного парсинга, с помощью функции readyProgramm АСД подготавливается для построение блок-схемы, затем строится блок-схема, и наконец отрисовывается в выходной файл.

Главный модуль является связующим звеном между главными модулями программы и позволяет инкапсулировать в себе всю работу с пользователем.

Данный модуль состоит из одного кода, который приведен в листинге 1.

Листинг 1.

module Main where

import Blocks

import HelperFunctions

import PythonParser

import DrawBlocks

import Diagrams.Backend.SVG.CmdLine

import Control.Applicative ((<|>))

import Data.Char

import Text.ParserCombinators.ReadP

main :: IO ()

main = mainWith makeDiagram

makeDiagram :: FilePath -> IO (Diagram B)

makeDiagram file = do

contents <- readFile file

let result = parsePython contents

case result of

Left s -> putStrLn "Error\nUnread string:\n" >> putStrLn s >> return (mempty)

Right p -> do

putStrLn "Parsed"

return $ convertCodeToDiagram p

convertCodeToDiagram :: Programm -> Diagram B

convertCodeToDiagram p = blocksToDiagram blocks

where

blocks = programmToBlock left : (functionsToBlock right)

left = fst $ readyProgramm p

right = snd $ readyProgramm p

Блок-схема главного модуля:

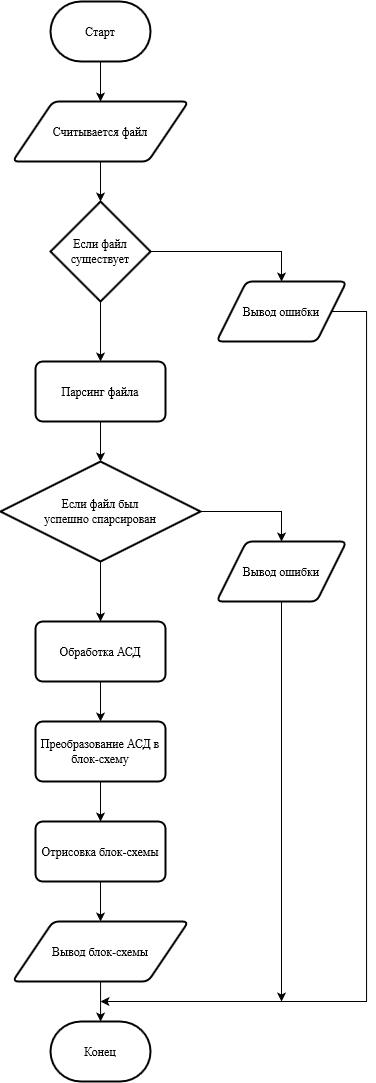


Рисунок 6 блок-схема главного модуля

## Описание спецификаций к модулям

Разработаны 6 модулей, не включая главный:

* Функции парсера
* Типы данных АСД
* Вспомогательные функции для парсинга отступов
* Типы данных блок-схемы и функции для построения блок-схемы
* Функции для отрисовки блок-схемы
* Вспомогательные функции, не зависящие от остальных модулей

Структура подключения модулей изображена ниже

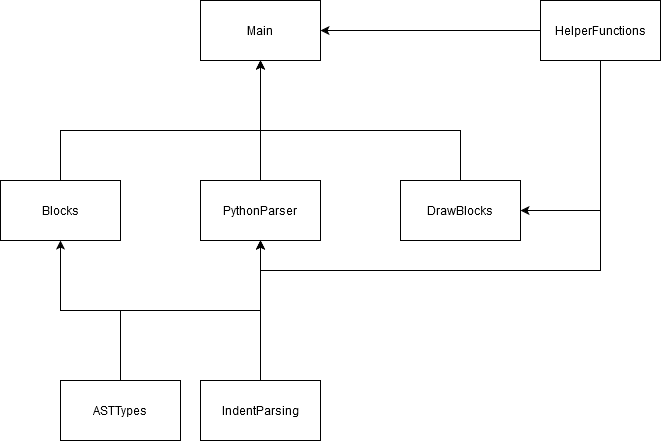


Рисунок 7 Структура подключения модулей

## Описание модулей

Модуль 1. Функции парсера

В этом модуле описаны функции парсинга отдельных грамматический конструкций, например nameParser, argsParser, functionParser.

Эта функция принимает уровень отступов и возвращает тип данных “функция”.

Листинг 1. Функция functionParser

functionParser :: Int -> ReadP Function

functionParser n = do

string "def"

skipSpaces

functionName <- nameParser

skipSpaces

char '('

arguments <- argsParser

char ')'

char ':'

codeEnd

code <- codeBlock n

return $ Function functionName arguments code

Листинг 2.

codeBlock :: Int -> ReadP Code

codeBlock oldN = do

ind <- begin oldN

actions <- sepBy1 (indentN ind >> actionParser ind) (codeEnd)

optional codeEnd

return $ Code actions

where

begin oldN = do

ind@(newN, \_) <- lookIndent

if newN <= oldN

then pfail

else return ind

Этот модуль экспортирует только две функции и реэкспортирует тип данных Programm:

parsePython – функция принимающая строку и возвращающая либо ошибку если парсинг не удался, либо тип данных Programm

readyProgramm – функция принимающая тип данных Programm, а возвращает разделённую АСД на основной алгоритм и фукнции использующиеся в алгоритме.

Листинг 3. Функция parsePython

parsePython :: String -> Either String Programm

parsePython s =

case (result) of

Nothing -> Left "unexpected error"

Just x ->

case x of

(\_, []) -> Right $ fst x

\_ -> Left $ snd x

where

result = lastMaybe $ readP\_to\_S pythonParser s

Листинг 4. Функция readyProgramm

readyProgramm :: Programm -> (Programm, [Action])

readyProgramm (Programm (Code acts)) = (wrappedLeft, wrappedRight)

where

wrappedLeft = Programm (Code (fst splitClean))

wrappedRight = snd splitClean

splitClean :: ([Action], [Action])

splitClean = (removeEmptyCodes left, removeEmptyCodes right)

right = map (Def) $ fst $ removeDefs acts

left = snd $ removeDefs acts

Модуль 2. Типы данных АСД

В данном модуле описываются и экспортируются все типы данных используемые в парсере.

Тип данных Programm, является конечным типом АСД.

Листинг. 5 Тип данных Programm и Code

data Programm =

Programm Code

deriving (Show, Eq)

data Code =

Code [Action]

deriving (Show, Eq)

В этом модуле также описан тип данных Action, отображающим разные виды команд на языке Python.

Листинг 6. Тип данных Action

data Action

= Assign Assignment

| Call CallF

| Def Function

| IfBlock If

| LoopW While

| LoopF For

| Flow FlowAct

deriving (Show, Eq)

Также есть вспомогательные синонимы типов для улучшения читаемости кода.

Листинг 7. Вспомогательные синонимы типов

type Name = String

type Expr = String

type Condition = String

type Variable = String

type SourceName = String

type FunctionName = String

type Argument = String

Модуль 3. Вспомогательные функция для парсинга отступов

Даный модуль содержит в себе функции позволяющие парсеру читать отступы.

Функция lookIndent используется для проверки размера отступа и типа отступа в начале блока кода.

Листинг 8. Функция lookIndent

lookIndent :: ReadP Indent

lookIndent = do

spaceInd@(nSpaces, \_) <- lookSpaceIndent

if nSpaces == 0

then do

tabInd <- lookTabIndent

return tabInd

else return spaceInd

Функция indentN читает отступ по полученным параметрам отступа.

Листинг 9. Функция indent

indentN :: Indent -> ReadP [Char]

indentN (n, t) = count n $ char $ indentChar t

Модуль 4. Типы данных блок-схемы и функции для построения блок-схемы

Этот модуль описывает тип данных Block – разные блоки используемые в блок-схемах по ГОСТ. Каждый конструктор этого типа имеет свои аргументы в зависимости от назначения блока. Все конструкторы, кроме конца получают следующий за ними блок.

Экспортируются функции programmToBlock, для преобразования АСД в блок-схему, и functionsToBlock, для преобразования функция описанных в программе в блок-схему. Также экспортируется тип данных блок-схемы.

Листинг 10. Тип данных Block

data Block

= Start (Maybe Comment) Block

| IOBlock String Block

| AssignBlock String Block -- Mod

| Procedure String Block

| Decision String Block (Maybe Block) Block

| LoopBlock String Block

| LoopEnd String Block

| Next

| End (Maybe String)

deriving (Show)

Функция actionToBlock отвечает за преобразования команды АСД в блок для блок-схемы.

Листинг 11. Функция actionToBlock

actionToBlock :: Action -> [String] -> Block -> Block

actionToBlock (Assign (Assignment n e)) \_ = AssignBlock $ combineNE n e

actionToBlock (Call (CallF sn fn args)) \_ =

if isIOAction fn

then IOBlock s

else if isExitAction fn

then (\\_ -> End Nothing)

else Procedure s

where

s = combineSnFnArgs sn fn args

actionToBlock (Def (Function fn args c)) names = \\_ -> Start (Just $ combineFnArgs fn args) (codeToBlock c (End Nothing) defaultNames)

actionToBlock (IfBlock (If cond c e)) names =

Decision cond (codeToBlock c Next names) $ elseToBlock e names

actionToBlock (LoopW (While cond c)) (name:names) =

LoopBlock (name ++ " While " ++ cond) . codeToLoop c name names

actionToBlock (LoopF (For var cond c)) (name:names) =

LoopBlock (name ++ " For " ++ var ++ " in " ++ cond) . codeToLoop c name names

actionToBlock (Flow flAct) \_ = flowActToBlock flAct

Модуль 5. Функции для отрисовки блок-схемы

В данном модуле находятся функции для преобразования блок-схемы в графический вид. В нём используется библиотека Diagrams позволяющая строить графику и выводить её в разные форматы.

Этот модуль экспортирует функцию blocksToDiagram, которая отрисовывает несколько блок-схем рядом друг с другом. А также реэкспортирует тип данных графики.

Функция blockToDiagrams отвечает за конвертацию блока в его графическое представление.

Листинг 12. Функция blockToDiagrams

blockToDiagrams :: Block -> Int -> [Diagram B]

blockToDiagrams (Blocks.Start (Just s) next) \_ =

(terminator "Start" # comment s) : blockToDiagrams next 1

blockToDiagrams (Blocks.Start (Nothing) next) \_ = (terminator "Start") : blockToDiagrams next 1

blockToDiagrams (IOBlock s next) level = (ioScheme s) : blockToDiagrams next level

blockToDiagrams (AssignBlock s next) level = (anyAction s) : blockToDiagrams next level

blockToDiagrams (Procedure s next) level = (callBlock s) : blockToDiagrams next level

blockToDiagrams (LoopBlock s next) level = (loopStart s) : blockToDiagrams next level

blockToDiagrams (LoopEnd s next) level = (loopEnd s) : blockToDiagrams next level

blockToDiagrams (Next) \_ = []

blockToDiagrams (Blocks.End (Just s)) \_ = (terminator s) : []

blockToDiagrams (Blocks.End (Nothing)) \_ = (terminator "End") : []

blockToDiagrams (Decision s left (Just right) next) level =

(combWithElse

level

(branchYesElse $ branchBlock s)

(blockToDiagram left (level + 1) "L")

(blockToDiagram right (level + 1) "R")) :

blockToDiagrams next level

blockToDiagrams (Decision s left (Nothing) next) level =

(comb level (branchYesNo $ branchBlock s) (blockToDiagram left (level + 1) "L")) :

blockToDiagrams next level

Здесь описаны функция определяющие графический вид каждого блока.

Листинг 13. Функция блоков.

terminator :: String -> Diagram B

terminator s = roundedRect 3 m 0.5 <> d

where

d = snd $ textC s

m = max (fst $ textC s) 1

ioScheme :: String -> Diagram B

ioScheme s = rect 3 m # shearX 0.3 <> d

where

d = snd $ textC s

m = max (fst $ textC s) 1

anyAction :: String -> Diagram B

anyAction s = rect 3 m <> d

where

d = snd $ textC s

m = max (fst $ textC s) 1

branchBlock :: String -> Diagram B

branchBlock s = square 1 # rotate (45 @@ deg) # scaleX mX # scaleY m <> d

where

d = snd $ textC' 15 s

mX = 2 + (m \* 0.2)

m = max (fst $ textC' 15 s) 1

Модуль 6. Вспомогательные функции

Этот модуль содержит в себе разнообразные функции, которые используются в разных модулях проекта. Они описаны в отдельном файле, потому что не используют других модулей программы и потому что из-за их универсальности могут быть использованы не только в данном проекте.

Функция dotOr получает два предиката и значение для проверки этими предикатами и возвращает True если один из предикатов вернул True.

Листинг 14. Функция dotOr

dotOr :: (a -> Bool) -> (a -> Bool) -> a -> Bool

dotOr f1 f2 = (||) <$> f1 <\*> f2

Функция lineWrap используется для разделения текста на строки заданной длины.

Листинг 15. Функция lineWrap

lineWrap :: String -> String

lineWrap s = unwords $ helper maxSpace $ words s

where

maxSpace = 33

spaceWidth = 1

helper \_ [] = []

helper spaceLeft (w1:ws) =

if (length (w1) + spaceWidth) > spaceLeft

then ('\n':w1) : (helper (maxSpace - length(w1)) ws)

else w1 : (helper (spaceLeft - length (w1) - spaceWidth) ws)

## Описание тестовых наборов модулей

В каждом модуле требуется вводить определенные значения. Там, где нужно ввести имя файла, идет проверка на существование такого файла, после идет проверка на правильность кодировки, далее на валидность Python файла парсером.

Тест 1. Ввод неправильного имени

Если файл не существует, то выведется ошибка

C:\Users\skaer2\Documents\Haskell\Block-Scheme-Creator\screenshots\doesNotExist.PNG

Рисунок 8 Ошибка при вводе неверного пути к файлу

Тест 2. Выбор неправильного формата для вывода.

Если ввести неправильное расширения файла вывода, то программа выдаст ошибку.

C:\Users\skaer2\Documents\Haskell\Block-Scheme-Creator\screenshots\WrongFormat.PNG

Рисунок 9 Ошибка при выборе неправильном формата

Тест 3. Ввод файла с ошибкой

При вводе файла с ошибкой программа выдаст ошибку и выведет остаток строки который не удалось считать.

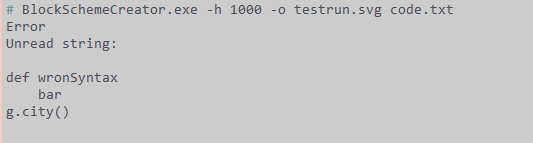


Рисунок 10 Ввод файла с ошибкой

Тест 4. Отсутствие обязательных аргументов

Если при использовании программы не ввести имя входного файла или имя файла на вывод, то программа выдаст ошибку.

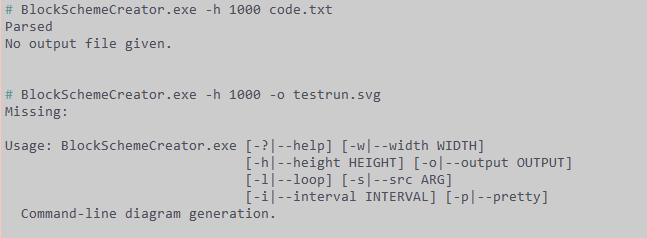


Рисунок 11 Отсутствие обязательных аргументов

## Описание применения средств отладки

В ходе написания курсового проекта при попытке запустить программу были получены ошибки и предупреждения:

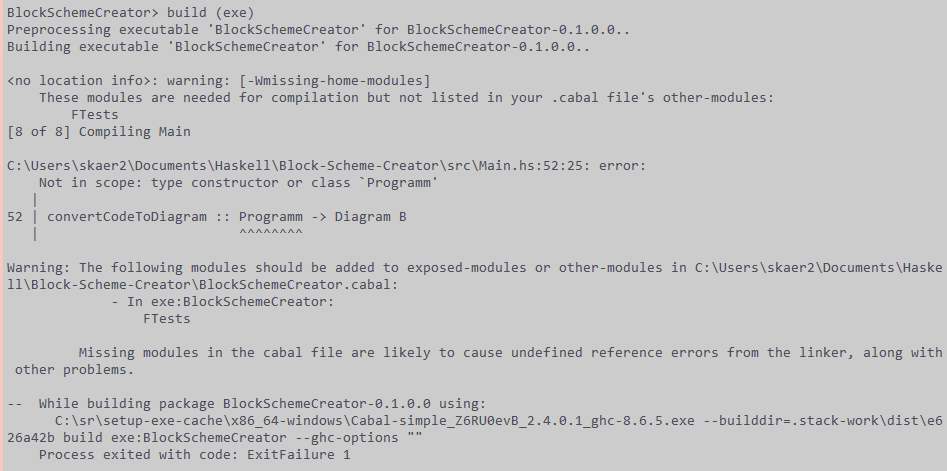


Рисунок 12 Ошибки и предупреждения

При проверке кода были исправлены найденные ошибки, в результате при запуске программы ошибок не было:

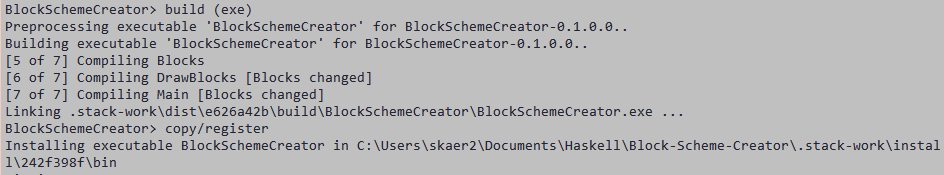


Рисунок 13 Отсутствие ошибок и предупреждений

## Анализ оптимальности использования памяти и быстродействия

Сложность рекурсивного алгоритма равна O(n), что при очень больших кодах заметно тормозит работу программы.

# Глава 3. Эксплуатационная часть

## Руководство оператора

* + 1. Назначение программы

Разработанное приложение производит парсинг Python кода в АСД, преобразование АСД в блок-схему и отрисовку блок-схемы в формате векторной графики SVG. Благодаря блок-схеме воспринимать код становится намного проще. Благодаря тому что SVG графику способны отображать большинство браузеров полученные блок-схемы можно просмотреть на любом компьютере, а также вставить на любой сайт.

* + 1. Условия выполнения программы

Операционная система: Windows 98/Vista/XP/7/8/10.

Процессор (CPU): Любой.

Оперативная память (RAM): 512 МБ.

Видеоадаптер: Любой.

Свободное место на жёстком диске: ~ 31Мб.

* + 1. Выполнение программы

Программа запускается через консоль, требуется указать аргументами “-w” и “-h” ширину и высоту выходной схемы в формате “.svg”, а также с помощью аргумента “-o” указать название файла, в который сгенерируется нарисованная блок-схема. На вход подаётся 1 текстовый файл (или .py), содержащий в себе код на языке Python.

Также доступна справка с помощью аргумента “-?” (“--help”).

Пример комманды:

C:\Users\skaer2\Documents\Haskell\Block-Scheme-Creator\screenshots\ExampleCommand.png

Рисунок 14 Запуск программы

Пример входного файла:

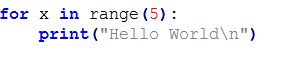


Рисунок 15 Пример входного файла

При успешном выполнении программы вы увидите надпись:

C:\Users\skaer2\Documents\Haskell\Block-Scheme-Creator\screenshots\SucessfulRun.png

Рисунок 16 Успешное выполнение программы

При наличии ошибок в коде, вы увидите строку которую не удалось считать парсеру.

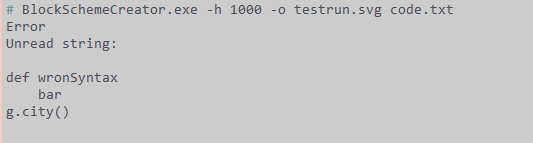


Рисунок 17 Вывод несчитанной строки

При успешном выполнении программы будет создан файл с указанным именем и форматом.C:\Users\skaer2\Documents\Haskell\Block-Scheme-Creator\screenshots\sucessfuloutput.PNG

Рисунок 18 Успешный вывод файла

* + 1. Сообщение оператору

Функция parsePython может вернуть “unexpected error”.

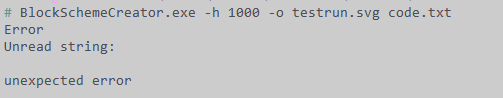


Рисунок 19 Ошибка при вводе пустого файла

ПРИЧИНА: Файл поданный на ввод пуст.

ДЕЙСТВИЯ ПРОГРАММЫ: Программа завершает работу

## To-Do лист

* Добавить поддержку большего количества языков программирования.
* Добавить поддержку большего количества форматов вывода блок-схемы.
* Добавить возможность выбрать стиль отрисовки блок-схемы.
* Добавить возможность редактировать сгенерированную блок-схему.
* Добавить опциональный графический интерфейс.

# Заключение

В результате выполнения курсового проекта было разработано приложение “Генератор блок-схем” для быстрой генерации блок-схем по исходному коду на языке Python.

В ходе работы были проанализированы существующие разработки посвященные данному направлению и изучены следующие темы: «Работы с графикой на Haskell» и «Разработка парсера».

Разработанная программа выполнена в соответствии с требованиями технического задания.

# Список литературы и интернет-источников

1. ГОСТ Схемы алгоритмов, программ данных и систем <http://cert.obninsk.ru/gost/282/282.html>
2. Блок-схемы алгоритмов. ГОСТ. Примеры <https://pro-prof.com/archives/1462>
3. Документация библиотеки Diagrams <https://archives.haskell.org/projects.haskell.org/diagrams/doc/cmdline.html#clock-example>
4. Аппликативные парсеры на Haskell <https://habr.com/ru/post/436234/>
5. Parsing - Write You a Haskell <http://dev.stephendiehl.com/fun/002_parsers.html>
6. Parser Combinators: Parsing for Haskell Beginners <https://two-wrongs.com/parser-combinators-parsing-for-haskell-beginners.html>
7. Hoogle API search engine <https://hoogle.haskell.org/>
8. Ссылка на программу “Flowchart”: <https://almiur.ru/show_prog_9.html>
9. Ссылка на программу “Visustin”: <https://www.aivosto.com/visustin-ru.html>

Приложение 1. Код главного модуля

module Main where

import Blocks

import HelperFunctions

import PythonParser

import DrawBlocks

import Diagrams.Backend.SVG.CmdLine

import Control.Applicative ((<|>))

import Data.Char

import Text.ParserCombinators.ReadP

main :: IO ()

main = mainWith makeDiagram

makeDiagram :: FilePath -> IO (Diagram B) -- reads a file and makes a diagram from blocks

makeDiagram file = do

contents <- readFile file

let result = parsePython contents

case result of

Left s -> putStrLn "Error\nUnread string:\n" >> putStrLn s >> return (mempty)

Right p -> do

putStrLn "Parsed"

return $ convertCodeToDiagram p

convertCodeToDiagram :: Programm -> Diagram B -- converts parsed code to diagram

convertCodeToDiagram p = blocksToDiagram blocks

where

blocks = programmToBlock left : (functionsToBlock right)

left = fst $ readyProgramm p

right = snd $ readyProgramm p

Приложение 2. Код модуля PythonParser

module PythonParser

( parsePython -- String -> Either String Programm

-- this function either returns an unread string or programm. Also can return an error string

, readyProgramm -- Programm -> (Programm, [Action])

-- splits a programm into a main algorithm and functions defined in that algorithm

, Programm

) where

--

-- This module describes functions to parse Python code

-- It uses a standart parser combinators library

--

--

--

--TODO add assignment with modifiers:

-- data Modifier = ...

--

-- data Assignment = ... Modifier ...

--

-- ... modifierParser ...

-- mod <- modifierParser

-- return Assignment ... mod ...

import Control.Applicative ((<|>))

import Data.Char

import Data.List

import Text.ParserCombinators.ReadP

import ASTTypes

import HelperFunctions

import IndentParsing

-- Code = List Actions

--

-- Actions = Assignment | Function | Call | Keywords

--

-- Assignment = Name Assign Expr

-- | Name BinOperation Assign Expr

--

-- Assign = '='

-- Name = String starting with a letter

--

-- Expr = String

--

-- Function = Def Name '(' Arguments ')' ':'

-- BlockBegin Code BlockEnd

-- BlockBegin = IndentIn

-- BlockEnd = IndentOut

--

parsePython :: String -> Either String Programm -- this function either returns an unread string or programm

-- also can return an error string

parsePython s =

case (result) of

Nothing -> Left "unexpected error"

Just x ->

case x of

(\_, []) -> Right $ fst x

\_ -> Left $ snd x

where

result = lastMaybe $ readP\_to\_S pythonParser s

pythonParser :: ReadP Programm -- a Python parser combinator

pythonParser = fmap Programm $ (skipMany importParser >> codeBlock (-1))

flowActParser :: ReadP FlowAct -- a Combinator which reads actions like return, continue break

flowActParser = breakParser <|> continueParser <|> returnParser <|> yieldParser

where

breakParser = string "break" >> return Break

continueParser = string "continue" >> return Continue

returnParser = xParser "return" (Return)

yieldParser = xParser "yield" (Yield)

xParser x f = -- parsers an action string and a argument string

string x >> skipSpaces >> do

s <- stringParser

return $ f s

importParser :: ReadP () -- reads imports

importParser = do

string "import"

void stringParser

codeEnd

nameParser :: ReadP String -- reads a name. a name can't start with a number

nameParser = do

firstLetter <- letter

everythingElse <- many $ satisfy isAlowed

return $ firstLetter : everythingElse

where

isAlowed = (||) <$> isAlphaNum <\*> isUnderscore

isUnderscore = (==) '\_'

commentParser :: ReadP String -- reads "#" comments

commentParser = do

char '#'

commentContent <- option [] stringParser

endOfLine

return commentContent

codeEnd :: ReadP () -- a parser combinator for code end

codeEnd = (endOfLine <|> (void commentParser)) >> skipLines

where

skipLines = skipMany emptyLine

emptyLine = do

munch (\x -> elem x " \t")

endOfLine <|> (void commentParser)

codeBlock :: Int -> ReadP Code -- a parser combinator for code blocks

-- takes a indent level

codeBlock oldN = do

ind <- begin oldN

actions <- sepBy1 (indentN ind >> actionParser ind) (codeEnd)

optional codeEnd

return $ Code actions

where

begin oldN = do

ind@(newN, \_) <- lookIndent

if newN <= oldN

then pfail

else return ind

actionParser :: Indent -> ReadP Action -- reads an action. takes indent level

actionParser ind = (fmap Flow flowActParser) <|> simpleActParser <|> (compoundActParser ind)

simpleActParser :: ReadP Action -- reads an action which cannot have actions in it

simpleActParser = (fmap Assign assigmentParser) <|> (fmap Call callParser)

compoundActParser :: Indent -> ReadP Action -- reads an action which can have actions in it

compoundActParser ind@(n, \_) =

(fmap Def $ functionParser n) <|> (fmap IfBlock $ ifParser ind) <|> (fmap LoopW $ whileParser n) <|>

(fmap LoopF $ forParser n)

assigmentParser :: ReadP Assignment -- reads an "=" action

assigmentParser = do

name <- nameParser

skipSpaces

char '='

skipSpaces

expr <- stringParser

return $ Assignment name expr

argsParser :: ReadP [Argument] -- reads arguments like "x,y,z"

argsParser = sepBy nameParser argsSeparator

argsSeparator :: ReadP () -- reads a separation between arguments

argsSeparator = skipSpaces >> char ',' >> skipSpaces

callParser :: ReadP CallF -- reads a function call like "print(x)" or "this.do()"

callParser = do

sourceName <- option Nothing source

functionName <- nameParser

char '('

arguments <- sepBy callArgP argsSeparator

char ')'

return $ CallF sourceName functionName arguments

where

callArgP = munch1 $ not . (flip elem) "\n(),"

source = do

name <- nameParser

char '.'

return $ Just name

functionParser :: Int -> ReadP Function -- reads a function definition

functionParser n = do

string "def"

skipSpaces

functionName <- nameParser

skipSpaces

char '('

arguments <- argsParser

char ')'

char ':'

codeEnd

code <- codeBlock n

return $ Function functionName arguments code

conditionParser :: ReadP Condition -- reads an boolean condition

conditionParser = munch1 $ not . (flip elem) ":"

elseParser :: Int -> ReadP (Maybe Else) -- reads an "else" structure

elseParser n = do

string "else:"

codeEnd

code <- codeBlock n

return $ Just $ Else code

elifParser :: Indent -> ReadP (Maybe Else) -- reads an "elif" structure

-- takes indent type

elifParser ind = do

string "el"

ifCode <- ifParser ind

return $ Just $ Else $ Code [IfBlock ifCode]

ifParser :: Indent -> ReadP If -- reads "if" structure

-- takes indent type

ifParser ind@(n, \_) = do

string "if"

skipSpaces

condition <- conditionParser

skipSpaces

char ':'

codeEnd

code <- codeBlock n

elseCode <- option Nothing (indentN ind >> (elseParser n <|> elifParser ind))

return $ If condition code elseCode

whileParser :: Int -> ReadP While -- reads a "while" loop

whileParser n = do

string "while"

skipSpaces

condition <- conditionParser

skipSpaces

char ':'

codeEnd

code <- codeBlock n

return $ While condition code

forParser :: Int -> ReadP For -- reads a "for" loop

forParser n = do

string "for"

skipSpaces

variable <- nameParser

skipSpaces

string "in"

skipSpaces

condition <- conditionParser

skipSpaces

char ':'

codeEnd

code <- codeBlock n

return $ For variable condition code

getInnerActions :: Action -> (Maybe [Action], Maybe [Action]) -- extracts actions from actions with code in them

getInnerActions (Def (Function \_ \_ (Code ias))) = (Just ias, Nothing)

getInnerActions (LoopW (While cond (Code ias))) = (Just ias, Nothing)

getInnerActions (LoopF (For var cond (Code ias))) = (Just ias, Nothing)

getInnerActions (IfBlock (If cond (Code ias) e)) =

case e of

Nothing -> (Just ias, Nothing)

Just (Else (Code eas)) -> (Just ias, Just eas)

getInnerActions \_ = (Nothing, Nothing)

mapInnerActions :: ([Action] -> [Action]) -> Action -> Action -- maps a function over inner actions

mapInnerActions f (Def (Function fn args (Code ias))) = Def (Function fn args (Code $ f ias))

mapInnerActions f (LoopW (While cond (Code ias))) = LoopW (While cond (Code $ f ias))

mapInnerActions f (LoopF (For var cond (Code ias))) = LoopF (For var cond (Code $ f ias))

mapInnerActions f (IfBlock (If cond (Code ias) e)) =

case e of

Nothing -> IfBlock (If cond (Code $ f ias) Nothing)

Just (Else (Code eas)) -> IfBlock (If cond (Code $ f ias) (Just (Else (Code $ f eas))))

mapInnerActions \_ a = a

removeEmptyCodes :: [Action] -> [Action] -- removes actions with code if that code is empty

removeEmptyCodes [] = []

removeEmptyCodes (a:as) =

case (getInnerActions a) of

(Just [], \_) -> removeEmptyCodes as

(Nothing, \_) -> a : removeEmptyCodes as

(Just (xs), \_) -> (mapInnerActions removeEmptyCodes a) : removeEmptyCodes as

removeDefs :: [Action] -> ([Function], [Action]) -- extracts function definitions

removeDefs [] = ([], [])

removeDefs ((Def (Function fn args (Code ias))):as) =

( (Function fn args (Code $ snd newIas)) : (fst $ newIas) ++ (fst $ removeDefs as)

, snd $ removeDefs as)

where

newIas = removeDefs ias

removeDefs ((LoopW (While cond (Code ias))):as) =

( (fst $ removeDefs as) ++ (fst newIas)

, (LoopW (While cond (Code (snd newIas)))) : (snd $ removeDefs as))

where

newIas = removeDefs ias

removeDefs ((LoopF (For var cond (Code ias))):as) =

( (fst $ removeDefs as) ++ (fst newIas)

, (LoopF (For var cond (Code (snd newIas)))) : (snd $ removeDefs as))

where

newIas = removeDefs ias

removeDefs ((IfBlock (If cond (Code ias) e)):as) =

( (fst $ removeDefs as) ++ (fst newIas) ++ (fst newE)

, (IfBlock (If cond (Code (snd newIas)) wrappedNewE)) : (snd $ removeDefs as))

where

newE =

case e of

Nothing -> ([], [])

Just (Else (Code eas)) -> removeDefs eas

wrappedNewE =

case e of

Nothing -> Nothing

Just (Else (Code eas)) -> Just (Else $ Code (snd $ removeDefs eas))

newIas = removeDefs ias

removeDefs (a:as) = (fst $ removeDefs as, a : (snd $ removeDefs as))

readyProgramm :: Programm -> (Programm, [Action]) -- readys Programm

readyProgramm (Programm (Code acts)) = (wrappedLeft, wrappedRight)

where

wrappedLeft = Programm (Code (fst splitClean))

wrappedRight = snd splitClean

splitClean :: ([Action], [Action])

splitClean = (removeEmptyCodes left, removeEmptyCodes right)

right = map (Def) $ fst $ removeDefs acts

left = snd $ removeDefs acts

Приложение 3. Код модуля ASTTypes

module ASTTypes where

data Programm =

Programm Code

deriving (Show, Eq)

data Code =

Code [Action]

deriving (Show, Eq)

type Name = String

type Expr = String

data Action

= Assign Assignment

| Call CallF

| Def Function

| IfBlock If

| LoopW While

| LoopF For

| Flow FlowAct

deriving (Show, Eq)

data FlowAct

= Break

| Continue

| Return String

| Yield String

deriving (Show, Eq)

type Condition = String

data If =

If Condition Code (Maybe Else)

deriving (Show, Eq)

data Else =

Else Code

deriving (Show, Eq)

data While =

While Condition Code

deriving (Show, Eq)

type Variable = String

data For =

For Variable Condition Code

deriving (Show, Eq)

data Assignment =

Assignment Name Expr

deriving (Show, Eq)

type SourceName = String

type FunctionName = String

type Argument = String

data CallF =

CallF (Maybe SourceName) FunctionName [Argument]

deriving (Show, Eq)

data Function =

Function FunctionName [Argument] Code

deriving (Show, Eq)

Приложение 4. Код модуля IndentParsing

module IndentParsing where

import Text.ParserCombinators.ReadP

type Indent = (Int, IndentType)

data IndentType

= Spaces

| Tabs

deriving (Show, Eq)

indentChar :: IndentType -> Char -- returns corresponding character for every indent type

indentChar Spaces = ' '

indentChar Tabs = '\t'

indentN :: Indent -> ReadP [Char] -- reads indent

indentN (n, t) = count n $ char $ indentChar t

lookTypeIndent :: IndentType -> ReadP Indent

lookTypeIndent t = do

s <- look

return (indent s, t)

where

indent (c:s)

| (c == indentChar t) = 1 + indent s

indent \_ = 0

lookSpaceIndent :: ReadP Indent

lookSpaceIndent = lookTypeIndent Spaces

lookTabIndent :: ReadP Indent

lookTabIndent = lookTypeIndent Tabs

lookIndent :: ReadP Indent -- checks for indent size and type

lookIndent = do

spaceInd@(nSpaces, \_) <- lookSpaceIndent

if nSpaces == 0

Приложение 5. Код модуля Blocks

module Blocks

( Block(..)

, programmToBlock

, functionsToBlock

) where

import ASTTypes

import Data.List

type Comment = String

data Block -- Block-scheme data type

= Start (Maybe Comment) Block

| IOBlock String Block

| AssignBlock String Block -- Mod

| Procedure String Block

| Decision String Block (Maybe Block) Block

| LoopBlock String Block

| LoopEnd String Block

| Next

| End (Maybe String)

deriving (Show)

-- names for loops

defaultNames = map (\c -> c : []) ['A' .. 'Z']

isIOAction :: String -> Bool -- checks if function name corresponds to an IO function

isIOAction s = s `elem` ioActions

where

ioActions = ["print", "input"]

isExitAction :: String -> Bool -- checks if function name corresponds to an exit function

isExitAction s = s `elem` exitActions

where

exitActions = ["exit"]

functionsToBlock :: [Action] -> [Block] -- converts function definitions to blocks

functionsToBlock as = map (\a -> actionsToBlock [a] (End Nothing) defaultNames) as

programmToBlock :: Programm -> Block -- converts whole AST to block

programmToBlock (Programm c) = Start Nothing $ codeToBlock c (End Nothing) defaultNames

codeToBlock :: Code -> Block -> [String] -> Block -- converts code to block

-- takes code, block to end the code, and names string

codeToBlock (Code as) = actionsToBlock as

actionsToBlock :: [Action] -> Block -> [String] -> Block -- converts a list of actions to block

-- takes actions, block to end the code, and names string

actionsToBlock [] lastB \_ = lastB

actionsToBlock (a:as) lastB names = actionToBlock a names $ actionsToBlock as lastB names

combineSnFnArgs :: Maybe String -> String -> [String] -> String -- combines source name, function name and arguments into one string

combineSnFnArgs Nothing fn args = combineFnArgs fn args

combineSnFnArgs (Just sn) fn args = sn ++ '.' : (combineFnArgs fn args)

combineFnArgs :: String -> [String] -> String -- combines function name and arguments into one string

combineFnArgs fn args = fn ++ ('(' : combArgs ++ ")")

where

combArgs = (intercalate ", " args)

combineNE :: String -> String -> String -- combines name and expression into one assignment string

combineNE s1 s2 = s1 ++ " = " ++ s2

elseToBlock :: Maybe Else -> [String] -> Maybe Block -- converts else construction into block

elseToBlock Nothing \_ = Nothing

elseToBlock (Just (Else c)) names = Just (codeToBlock c Next names)

codeToLoop :: Code -> String -> [String] -> Block -> Block -- converts code into a loop section block

codeToLoop c s names b = codeToBlock c (LoopEnd s b) names

flowActToBlock :: FlowAct -> Block -> Block -- converts actions like return, continue, brake into block

flowActToBlock Break \_ = Next

flowActToBlock Continue \_ = Next

flowActToBlock (Return s) \_ = End (Just s)

flowActToBlock (Yield s) \_ = End (Just s)

actionToBlock :: Action -> [String] -> Block -> Block -- converts action of every type into block

-- takes action, names, and next block

actionToBlock (Assign (Assignment n e)) \_ = AssignBlock $ combineNE n e

actionToBlock (Call (CallF sn fn args)) \_ =

if isIOAction fn

then IOBlock s

else if isExitAction fn

then (\\_ -> End Nothing)

else Procedure s

where

s = combineSnFnArgs sn fn args

actionToBlock (Def (Function fn args c)) names =

\\_ -> Start (Just $ combineFnArgs fn args) (codeToBlock c (End Nothing) defaultNames)

actionToBlock (IfBlock (If cond c e)) names =

Decision cond (codeToBlock c Next names) $ elseToBlock e names

actionToBlock (LoopW (While cond c)) (name:names) =

LoopBlock (name ++ " While " ++ cond) . codeToLoop c name names

actionToBlock (LoopF (For var cond c)) (name:names) =

LoopBlock (name ++ " For " ++ var ++ " in " ++ cond) . codeToLoop c name names

actionToBlock (Flow flAct) \_ = flowActToBlock flAct

Приложение 6. Код модуля DrawBlocks

{-# LANGUAGE FlexibleContexts #-}

module DrawBlocks

( blocksToDiagram

, Diagram

, B

) where

import Blocks

import HelperFunctions

import Data.List

import Diagrams.Backend.SVG

import Diagrams.Backend.SVG.CmdLine

import Diagrams.Prelude

import Diagrams.Trail

import Diagrams.TwoD.Layout.Grid

import Diagrams.TwoD.Text

branchYes b = cat unit\_X [b, textLeft]

branchYesElse b = cat unitX [branchYes b, textRight]

branchYesNo b = cat unit\_Y [branchYes b, textBottom]

textRight :: Diagram B -- a text placed to the right of a block

textRight = alignBL $ text "No" # fontSize (local 0.5) <> pha

where

pha = setTrace emptyTrace $ phantom (rect 1.5 0.7 :: Diagram B)

emptyTrace = getTrace $ (strutX 1 :: Diagram B)

textBottom :: Diagram B -- a text placed to the bottom of a block

textBottom = alignTL $ text "No" # fontSize (local 0.5) <> pha

where

pha = setTrace emptyTrace $ phantom (rect 1.5 0.7 :: Diagram B)

emptyTrace = getTrace $ (strutY 1 :: Diagram B)

textLeft :: Diagram B -- a text placed to the left of a block

textLeft = alignBR $ text "Yes" # fontSize (local 0.5) <> pha

where

pha = setTrace emptyTrace $ phantom (rect 1.5 0.7 :: Diagram B)

emptyTrace = getTrace $ (strutX 1 :: Diagram B)

blocksToDiagram :: [Block] -> Diagram B -- converts blocks to diagram with horizontally placed schemes. the only exported function

blocksToDiagram bs = hcat $ intersperse (strutX 10) ds # lwL 0.05

where

ds = map (\x -> blockToDiagram x 1 "M") bs

blockToDiagram :: Block -> Int -> String -> Diagram B -- converts and connects blocks into a diagram. takes level and names

blockToDiagram b level s = vcatConnect (blockToDiagrams b level) level s

blockToDiagrams :: Block -> Int -> [Diagram B] -- converts a block into a list of diagrams

blockToDiagrams (Blocks.Start (Just s) next) \_ =

(terminator "Start" # comment s) : blockToDiagrams next 1

blockToDiagrams (Blocks.Start (Nothing) next) \_ = (terminator "Start") : blockToDiagrams next 1

blockToDiagrams (IOBlock s next) level = (ioScheme s) : blockToDiagrams next level

blockToDiagrams (AssignBlock s next) level = (anyAction s) : blockToDiagrams next level

blockToDiagrams (Procedure s next) level = (callBlock s) : blockToDiagrams next level

blockToDiagrams (LoopBlock s next) level = (loopStart s) : blockToDiagrams next level

blockToDiagrams (LoopEnd s next) level = (loopEnd s) : blockToDiagrams next level

blockToDiagrams (Next) \_ = []

blockToDiagrams (Blocks.End (Just s)) \_ = (terminator s) : []

blockToDiagrams (Blocks.End (Nothing)) \_ = (terminator "End") : []

blockToDiagrams (Decision s left (Just right) next) level =

(combWithElse

level

(branchYesElse $ branchBlock s)

(blockToDiagram left (level + 1) "L")

(blockToDiagram right (level + 1) "R")) :

blockToDiagrams next level

blockToDiagrams (Decision s left (Nothing) next) level =

(comb level (branchYesNo $ branchBlock s) (blockToDiagram left (level + 1) "L")) :

blockToDiagrams next level

combWithElse :: Int -> Diagram B -> Diagram B -> Diagram B -> Diagram B -- makes an "if" construction with "else"

combWithElse level top left right =

comb'' # connect90deg True (level, "top") (level, "leftBottom") #

connect90deg False (level, "leftBottom") (level, "dot") #

connect90deg True (level, "top") (level, "rightBottom") #

connect90deg False (level, "rightBottom") (level, "dot")

where

comb'' = cat (r2 (0, -1)) [comb', strut (r2 (0, -1)), circle 0.01 # named (level, "dot")]

leftAndRight =

centerX $

cat

(unitX)

[left # named (level, "leftBottom"), strutX 4, right # named (level, "rightBottom")]

comb' = cat (r2 (0, -1)) [top # named (level, "top"), strutY 1, leftAndRight]

comb :: Int -> Diagram B -> Diagram B -> Diagram B -- makes an "if" construction without "else"

comb level top left =

comb'' # connect90deg True (level, "top") (level, "leftBottom") #

connect90deg False (level, "leftBottom") (level, "dot") #

connectOutsideL (level, "top") (level, "dot")

where

comb'' = cat (r2 (0, -1)) [comb', strut (r2 (0, -1)), circle 0.01 # named (level, "dot")]

comb' =

cat

(r2 (-1, -1))

[ top # named (level, "top")

, strut (r2 (-1, -1))

, alignR $ left # named (level, "leftBottom")

]

comment :: String -> Diagram B -> Diagram B -- makes a comment block

comment s d =

cat (r2 (1, 0))

[ d

, hrule 0.5 # dashingL (replicate 5 0.1) 0

, commentPart <> alignedTextC s

, phantom ((rect 2.8 1) :: Diagram B)

]

where

alignedTextC s = fontSize (local (3 / fromIntegral (length (s)))) $ alignedText 0 0.5 s

commentPart =

centerXY $ stroke $ lineFromVertices [0.2 ^& 0, 0 ^& 0, 0 ^& 0.5, 0 ^& 1, 0.2 ^& 1]

textC :: String -> (Double, Diagram B) -- makes a text with line wrap

textC s = (modifier, diagram)

where

modifier = fromIntegral (length ls) \* 0.5

ls = lines $ lineWrap s

diagram = centerXY $ vcat $ map f ls

f s = txt s <> pha

pha = setTrace emptyTrace $ phantom (rect 3 0.5 :: Diagram B)

txt = fontSizeL (0.2) . text

emptyTrace = getTrace $ (strutY 1 :: Diagram B)

textC' :: Int -> String -> (Double, Diagram B) -- makes a text with line wrap. also takes max line length

textC' n s = (modifier, diagram)

where

modifier = fromIntegral (length ls) \* 0.5

ls = lines $ lineWrap' n s

diagram = centerXY $ vcat $ map f ls

f s = txt s <> pha

pha = setTrace emptyTrace $ phantom (rect 3 0.5 :: Diagram B)

txt = fontSizeL (0.2) . text

emptyTrace = getTrace $ (strutY 1 :: Diagram B)

terminator :: String -> Diagram B -- terminator block

terminator s = roundedRect 3 m 0.5 <> d

where

d = snd $ textC s

m = max (fst $ textC s) 1

ioScheme :: String -> Diagram B -- block for IO

ioScheme s = rect 3 m # shearX 0.3 <> d

where

d = snd $ textC s

m = max (fst $ textC s) 1

anyAction :: String -> Diagram B -- block for general actions

anyAction s = rect 3 m <> d

where

d = snd $ textC s

m = max (fst $ textC s) 1

branchBlock :: String -> Diagram B -- "if" block

branchBlock s = square 1 # rotate (45 @@ deg) # scaleX mX # scaleY m <> d

where

d = snd $ textC' 15 s

mX = 2 + (m \* 0.2)

m = max (fst $ textC' 15 s) 1

callBlock :: String -> Diagram B -- block for function calls

callBlock s = rect 3 m <> rect 3 m # scaleX 0.9 <> d

where

d = snd $ textC s

m = max (fst $ textC s) 1

loopStart :: String -> Diagram B -- starting loop block

loopStart s = loopForm # scaleY m <> d

where

d = snd $ textC s

m = max (fst $ textC s) 1

loopForm :: Diagram B -- loop block shape

loopForm =

centerXY $

stroke $ closeLine $ lineFromVertices [0 ^& 0, 0 ^& 0.8, 0.2 ^& 1, 2.8 ^& 1, 3 ^& 0.8, 3 ^& 0]

loopEnd :: String -> Diagram B -- block for loops end

loopEnd s = loopForm # reflectY # scaleY m <> d

where

d = snd $ textC s

m = max (fst $ textC s) 1

-- True for connecting above

-- False for connecting belove

-- connect two blocks with a 90 degree angle

connect90deg ::

(TypeableFloat n, Renderable (Path V2 n) b, IsName n1, IsName n2)

=> Bool

-> n1

-> n2

-> QDiagram b V2 n Any

-> QDiagram b V2 n Any

connect90deg b n1 n2 d =

case intPoint of

Just p ->

(d <> intPoint' p) # connectOutsideL' (with & arrowHead .~ lineHead) n1 "intPoint" #

connectOutsideL "intPoint" n2

Nothing -> d

where

intPoint =

if b

then getIntersectionPoint n2 n1 d

else getIntersectionPoint n1 n2 d

intPoint' x = position [(x, circle 0.001 # named "intPoint")]

connect90deg' :: -- connect two blocks with a 90 degree angle. also takes arrow options

(TypeableFloat n, Renderable (Path V2 n) b, IsName n1, IsName n2)

=> ArrowOpts n

-> Bool

-> n1

-> n2

-> QDiagram b V2 n Any

-> QDiagram b V2 n Any

connect90deg' opts b n1 n2 d =

case intPoint of

Just p ->

(d <> intPoint' p) # connectOutsideL' (opts & arrowHead .~ lineHead) n1 "intPoint" #

connectOutsideL' opts "intPoint" n2

Nothing -> d

where

intPoint =

if b

then getIntersectionPoint n2 n1 d

else getIntersectionPoint n1 n2 d

intPoint' x = position [(x, circle 0.001 # applyStyle (\_shaftStyle opts) # named "intPoint")]

-- names

defaultNames = map (\c -> c : []) ['A' .. 'Z']

-- combines a list of diagrams and a list of names

zipName :: (IsName nm1, IsName nm2) => [Diagram B] -> [nm1] -> nm2 -> [Diagram B]

zipName ds nms lastName = go ds nms

where

go [] \_ = []

go (x:[]) \_ = named lastName x : []

go \_ [] = []

go (x:xs) (y:ys) = named y x : go xs ys

-- connects outside edges in a list of diagrams to eachover

connectOutsideList :: (IsName nm1, IsName nm2) => Int -> [nm1] -> nm2 -> Diagram B -> Diagram B

connectOutsideList 1 \_ \_ d = d

connectOutsideList 2 (n1:\_) lastName d = connectOutsideL n1 lastName $ d

connectOutsideList \_ [] \_ d = d

connectOutsideList \_ (\_:[]) \_ d = d

connectOutsideList n (n1:n2:ns) lastName d =

connectOutsideL n1 n2 $ connectOutsideList (n - 1) (n2 : ns) lastName d

-- connects and combines a list of diagrams into one diagram. takes a levelName and a name for the last block in the list

vcatConnect :: (IsName nm1, IsName nm2) => [Diagram B] -> nm1 -> nm2 -> Diagram B

vcatConnect ds levelName lastName =

vcat (intersperse (strutY 1) namedDs) #

connectOutsideList (length ds) namesList (levelName, lastName)

where

namedDs = zipName ds namesList (levelName, lastName)

namesList = map (\x -> (levelName, x)) [1 :: Int,2 ..]

connectOutsideL ::

(TypeableFloat n, Renderable (Path V2 n) b, IsName n1, IsName n2)

=> n1

-> n2

-> QDiagram b V2 n Any

-> QDiagram b V2 n Any -- same as connectOutside but with modified line width

connectOutsideL = connectOutside' (with & headLength .~ (local 0.4))

connectOutsideL' ::

(TypeableFloat n, Renderable (Path V2 n) b, IsName n1, IsName n2)

=> ArrowOpts n

-> n1

-> n2

-> QDiagram b V2 n Any

-> QDiagram b V2 n Any -- same as connectOutside' but with modified line width

connectOutsideL' opts = connectOutside' (opts & headLength .~ (local 0.4))

getPointX (P (V2 x \_)) = x

getPointY (P (V2 \_ y)) = y

getIntersectionPoint :: -- returns an intersection point between two diagrams.

(Semigroup m, Floating a, Ord a, IsName n1, IsName n2)

=> n1

-> n2

-> QDiagram b V2 a m

-> Maybe (Point V2 a)

getIntersectionPoint nameX nameY d =

(\x y -> P $ V2 x y) <$> (fmap getPointX point1) <\*> (fmap getPointY point2)

where

points = names d

point1 =

case lookup (toName nameX) points of

Just (p:ps) -> (Just p)

Nothing -> Nothing

point2 =

case lookup (toName nameY) points of

Just (p:ps) -> (Just p)

Nothing -> Nothing

Приложение 7. Код модуля HelperFunctions

module HelperFunctions where

import Data.Char

import Text.ParserCombinators.ReadP

lineWrap :: String -> String -- adds "\n" to split string into lines to fit into line

lineWrap s = unwords $ helper maxSpace $ words s

where

maxSpace = 33

spaceWidth = 1

helper \_ [] = []

helper spaceLeft (w1:ws) =

if (length (w1) + spaceWidth) > spaceLeft

then ('\n':w1) : (helper (maxSpace - length(w1)) ws)

else w1 : (helper (spaceLeft - length (w1) - spaceWidth) ws)

lineWrap' :: Int -> String -> String -- adds "\n" to split string into lines. also takes max line length

lineWrap' maxSpace s = unwords $ helper maxSpace $ words s

where

spaceWidth = 1

helper \_ [] = []

helper spaceLeft (w1:ws) =

if (length (w1) + spaceWidth) > spaceLeft

then ('\n':w1) : (helper (maxSpace - length(w1)) ws)

else w1 : (helper (spaceLeft - length (w1) - spaceWidth) ws)

interleave :: [a] -> [a] -> [a] -- combines two list with alternating elements

interleave (e:es) (o:os) = e : o : interleave es os

interleave \_ \_ = []

dotOr :: (a -> Bool) -> (a -> Bool) -> a -> Bool -- takes two predicates and a value to check. returns true if one of the predicates is true

dotOr f1 f2 = (||) <$> f1 <\*> f2

void :: Monad m => m a -> m () -- voids a monad action

void a = a >> return ()

showResult :: [(a, String)] -> Maybe a -- returns parsing result

showResult r =

case showResults r of

[] -> Nothing

(x:\_) -> Just $ fst x

showResults :: [(a, String)] -> [(a, String)] -- returns parsing results

showResults =

filter

(\(x, leftovers) ->

if leftovers == ""

then True

else False)

showShortOutput :: [(a, String)] -> [(a, String)] -- returns parsing results but with shortened string

showShortOutput = map (\(x, leftovers) -> (x, take 10 leftovers))

lastMaybe :: [a] -> Maybe a -- maybe returns a last element of the list

lastMaybe xs =

if null xs

then Nothing

else Just (last xs)

addLineBreaks :: String -> String -- add line breaks on commas

addLineBreaks [] = []

addLineBreaks (x:xs) =

if x == ','

then x : '\n' : (addLineBreaks xs)

else x : (addLineBreaks xs)

letter :: ReadP Char -- reads letters

letter = satisfy isAlpha

digit :: ReadP Char -- reads digits

digit = satisfy isDigit

number :: ReadP Int -- reads a number

number = fmap (read) $ many1 digit

isEndOfLine :: Char -> Bool -- end of line predicate

isEndOfLine = (==) '\n'

stringParser :: ReadP String -- reads a line

stringParser = munch1 $ not . isEndOfLine

-- reads endOfLine

endOfLine :: ReadP () -- TODO \ \n

endOfLine = void $ satisfy isEndOfLine