Скриптуем и автоматизируем безопасно с Haskell

Алексей Пирогов, “Typeable.IO”

http://www.devconf.ru

Содержание

- Автоматизация (shell scripting) на языках высокого уровня.
- Скрипты и управление зависимостями.
- Библиотека **Turtle.**
- Заключение.

Автоматизация (shell scripting) на языках высокого уровня

Зачем использовать языки высокого уровня?

- **Абстракция**: свыше нам уже даны структуры данных и типы, позволяющие сразу приступать к решению задач.
- **Гибкость**: для языков высокого уровня доступны как высокоуровневые, так и низкоуровневые библиотеки.
- **Масштабируемость**: растущий проект мы всегда можем разделить на модули и тем самым управлять сложностью.
- **“Прочность”**: всё вышеперечисленное упрощает рефакторинг и повышает надёжность программ в целом.

Зачем нам статическая типизация?

- Языки с динамической типизацией довольно популярны в мире автоматизации, благодаря своей **простоте** и **низкому порогу вхождения**.
- Однако, все подобные языки обладают общим недостатком: увеличиваясь в размерах, **программы становятся сложными** для понимания.
- Статически типизированные программы **проще изменять и расширять**.

Почему **Haskell**?

- **Лаконичный синтаксис** без излишнего визуального “шума”
- **Хороший набор** полезных для написания скриптов **библиотек**: описание **CLI**, **парсеры**, работа с **CSV**, **JSON**, **YAML**, работа с сетью и на низком и на высоком уровне.
- Возможность запускать программы без предварительной сборки (в режиме интерпретации*).

Скрипты и управление зависимостями

Всё, что вам нужно, это **Stack**

Stack (haskellstack.org) - настоящий швейцарский нож, ведь он

- устанавливает **GHC** (изолировано от остальной системы),
- устанавливает необходимые **пакеты** (зависимости),
- **собирает** проекты,
- запускает **тесты**,
- запускает **benchmarks**.

(а также создаёт проекты на основе шаблонов, генерирует документацию, собирает docker-образы и проч.)

"Hello World"

Типичный скрипт на Haskell выглядит так:

```
#!/usr/bin/env stack
-- stack --install-ghc runghc

main = putStrLn "Hello World!"
```

Воспроизводимость результата

Переносимый скрипт обычно выглядит так:

```
#!/usr/bin/env stack
-- stack
--   --resolver=6.30
--   script
--   --package=turtle
--   --package=containers

import Turtle (echo)

main :: IO ()
main = echo "Hello World!"
```

Библиотека **Turtle**

Библиотека **Turtle**

- **Turtle**, это реализация утилит командной строки **UNIX** на **Haskell**
- Предоставляет набор узнаваемых функций для работы с
  - **файловой системой**,
  - **потоками (pipes)**,
  - **подзадачами** (подпроцессами).
- Позволяет **шаблонизировать строки** и работать с **типизированными регулярными выражениями**.

Привычный инструментарий

```
echo :: Text -> IO ()
pwd  :: IO FilePath
cd   :: FilePath -> IO ()
cp   :: FilePath -> FilePath -> IO ()

script :: IO ()
script = do
  echo "Hello!"
  cd "Documents" -- (*)
  here <- pwd
  cp "tasks.txt" "tasks.bak"
  echo (format ("You're here: "%fp) here)
```

Типизированные потоки (pipes)

```
stdin  :: Shell Text
stdout :: Shell Text -> IO ()

input  :: FilePath -> Shell Text
output :: FilePath -> Shell Text -> IO ()

grep :: Pattern a    -> Shell Text     -> Shell Text
sed  :: Pattern Text -> Shell Text     -> Shell Text
find :: Pattern a    -> Shell FilePath -> Shell FilePath

fmap :: (a -> b)     -> Shell a        -> Shell b

-- cat foo.txt | grep PATTERN > bar.txt
output "bar.txt" (grep PATTERN (input "foo.txt"))
-- или
input "foo.txt" & grep PATTERN & output "bar.txt"
```

Подпроцессы

```
inshell :: Text ->           Shell Text -> Shell Text
-- ishell "tail -f" stdin
inproc  :: Text -> [Text] -> Shell Text -> Shell Text
-- ishell "tail" ["-f"] stdin

-- === Exit Codes ===
proc :: Text -> [Text] -> Shell Text -> IO ExitCode

script = do
  ec <- proc "ping" ["google.com"] empty
  case ec of
    ExitSuccess      -> echo "Done!"
    ExitFailure code -> echo (format ("Error: "%d) code)

proc1 .||. proc2 -- аналог привычного "||"
proc3 .&&. proc4 -- аналог привычного "&&"
```

Заключение

Вопросы?