Haski the Robot

Final examination project for
Advanced Functional Programming 2004

Lars Otten
<ottenl@student.chalmers.se>

A robot? WTF?!

- The motivation for this project goes back to a progamming course in secondary school
  - We used "Niki der Roboter", a very stripped-down versions of Pascal
  - Originally based on "Karel - the Robot" by Richard E. Pattis, 1981
  - There are variants for C, Java and the like
- But this one is new:
  - It's all Haskell ! :-)

What is it?

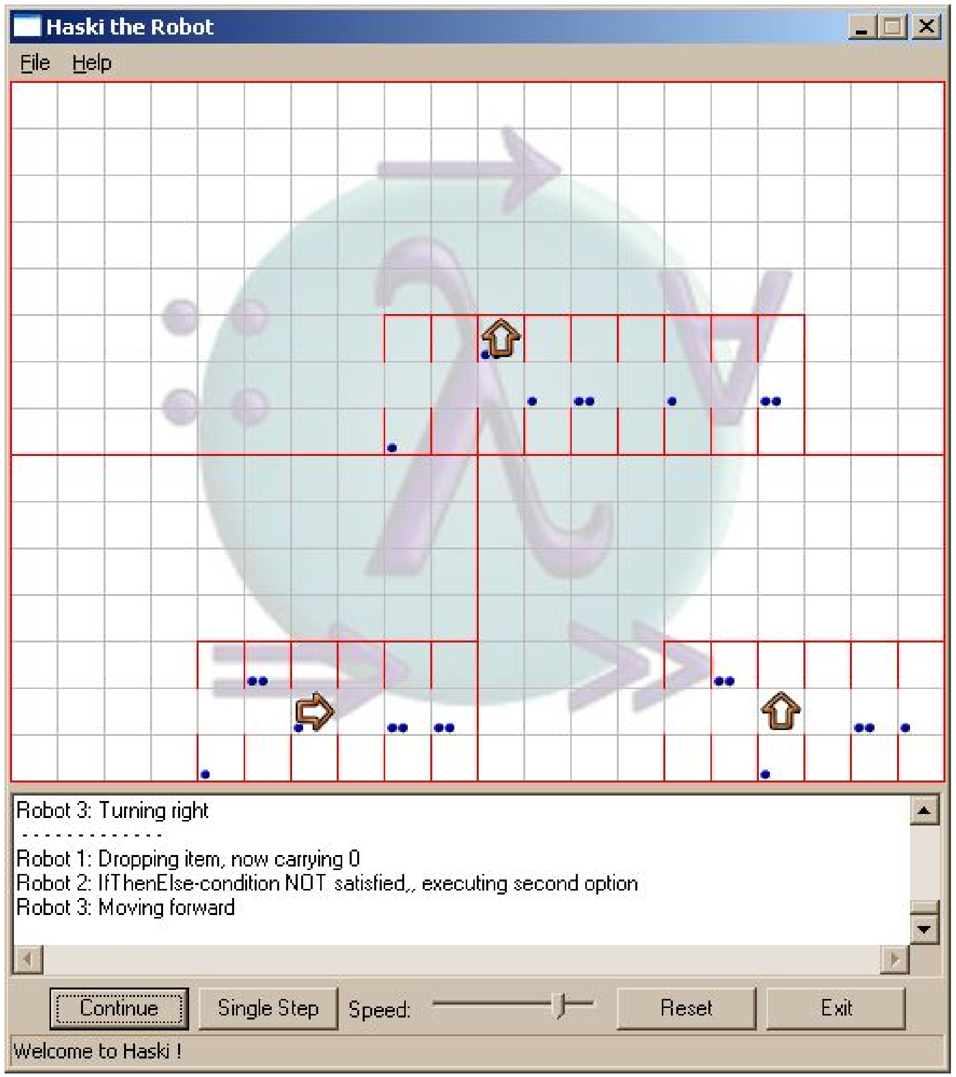
- *Haski the robot* is a complete programming environment for the very simple language "Haski"
  - The user controls a small robot on a working area, trying to accomplish various tasks
- Intended for people who have no or very little previous programming experience
- Keep it simple !
  - Very limited choice of commands
  - Only basic syntactical structures

So what does Haski offer?

- Haski's main features:
  - A fully-fledged GUI that shows the working area and the robot(s) as they execute the programs
  - A quite sophisticated lexical and syntactical parser for Haski-programs and mapfiles
    - Reads Haski-code and map definitions from files and parses them, creating a suitable data structure which can be used by the GUI and it's underlying interpreter
  - It is independent of GHCi, i.e. it can be compiled and distributed as a stand-alone executable file.
    - Important for usability wrt. unexperienced users
  - An extensive user manual with lots of examples

Let's have a look

- Working area divided into 15x20 fields
- Movement blocked by walls
- Items lying on certain fields
- One or more robots at the same time



```
Robot 3: Turning right
- - - - - - - - - - - - -
Robot 1: Dropping item, now carrying 0
Robot 2: IfThenElse-condition NOT satisfied,, executing second option
Robot 3: Moving forward
```

Welcome to Haski !

Haski commands

- Controlling a robot is simple:
  - The robot understands the following commands:
    - `go_forward`
    - `turn_right`
    - `take_item`
    - `drop_item`
    - `do_nothing`
  - It implements some boolean sensors:
    - `front_free, left_free, right_free`
    - `facing_up, facing_right, facing_down, facing_left`
    - `field_has_item, is_carrying`

Writing programs

- Commands can be combined with "**`&>`**"
  - `turn_right &> go_forward &> take_item`
- Four conditional expressions:
  - `IfThen` *<condition> <commands>*
  - `IfThenElse` *<condition> <commands> <commands>*
  - `While` *<condition> <commands>*
  - `DoWhile` *<commands> <condition>*
- Condition:
  - Built of sensors or combination of sensors
    - **`:&`** for AND, **`:|`** for OR, **`Not`** for negation

Functions

- A program consists of one or more function definitions:
  - `main = While front_free go_forward`
    - Every program must have the `main`-function
- More functions can be defined and used elsewhere:

```
– main = go_forward &>
         IfThen left_free turn_left

  turn_left = turn_right &>
              turn_right &>
              turn_right
```

Implementation

- Main steps (rough outline):
  - Implement the GUI and its underlying interpreter
    - Used embedded language in the beginning
  - Build a lexical and syntactical parser for programs
  - Add error handling and meaningful output to parser
  - Build a lexical and syntactical parser for mapfiles
  - Add error handling here as well
  - Add some refinements to the GUI like loading programs and maps and a log-window
  - Bugfixing :-p

Parsing

- Parsing consists of two steps
  - `lexer :: String -> [Token]`
    - Define list of Tokens using RegEx
    - Haskell-lexer is generated with the tool *Alex*
  - `parser :: [Token] -> Program`
    - Define a suitable CFG for the language and how it translates to the internal data structure for programs
    - Use *Happy* to generate a Haskell-parser
- Problem with this simple version:
  - On parsing errors the haskell function `error "..."` is called, which is definitely not good style

Parse-error handling

- Monadic approach:
  - Construct a monad for handling and passing errors
    - `data Parse a = Ok a | Failed String`
      `instance Monad Parse where ...`
    - `lexer :: String -> Parse [Token]`
      `parser :: [Token] -> Parse Program`
    - Final parser is `\s -> lexer s >>= parser`
  - We want meaningful error messages
    - Line/column numbers and strings have to be passed around while parsing, which gets pretty messy
    - Monad has to be integrated into the parsing, requires quite some handwork