CS4450 Lecture ~~Notes~~ Notes, Page 1 29

Andrew Krall 10-26-2018

Parsing in domain specific languages is fairly easy.

Haskell has a built-in function called "parse".

Type of parse:

parse :: Parser a -> String -> [(a, String)]

Command:

parse parseExp "99"

output: [(Const 99, "")]

Second part is an empty string.

May give back one "a" or multiple.

ExpParser.hs is on Harrison's website; it contains all of the parser code.

Functions inside there are parseNeg and parseAdd, among others.

ExpParse is one of those things where you can understand how to use it without knowing exactly how it works.

parse parseExp "(- 99)"

will output

[(Neg (Const 99), "")]

Signifies that the parsing is done.

On Monday, we'll go through building a language of types.

It's interesting to note that human languages are complex, so you can create →

Andrew
Krall
10-26-2018

a sentence that has multiple meanings depending on how it is parsed.

Almost every real life program uses some form of parser to pre-process it's input.

In a functional language such as Haskell, parsers can naturally be viewed as functions.

Type Parser = String -> Tree

A parser is a function that takes a string and returns some form of tree.

Basic Building Blocks of Parsing

The parser item on page 23 in the slides fails if the input is empty, and consumes the first character otherwise.

The parser failure always fails (page 24). It's kind of like the type Nothing.

The parser (p +++ q) behaves as the parser p if it succeeds and as the parser q otherwise (page 25).

The function parse applies a parser to a string (as described in the beginning of the notes along with it's type.

→

 Andrew Krall
10-26-2018

Check the examples on page 27 for more parsing examples in Harrison's slides.

A homework will be given on <u>Monday</u> regarding parsing! We'll get a parser for a typed language and will need to extend it. We'll work up to type checking as well.