Parsing

# LECTURE 5


© 2011 University of Tuebingen

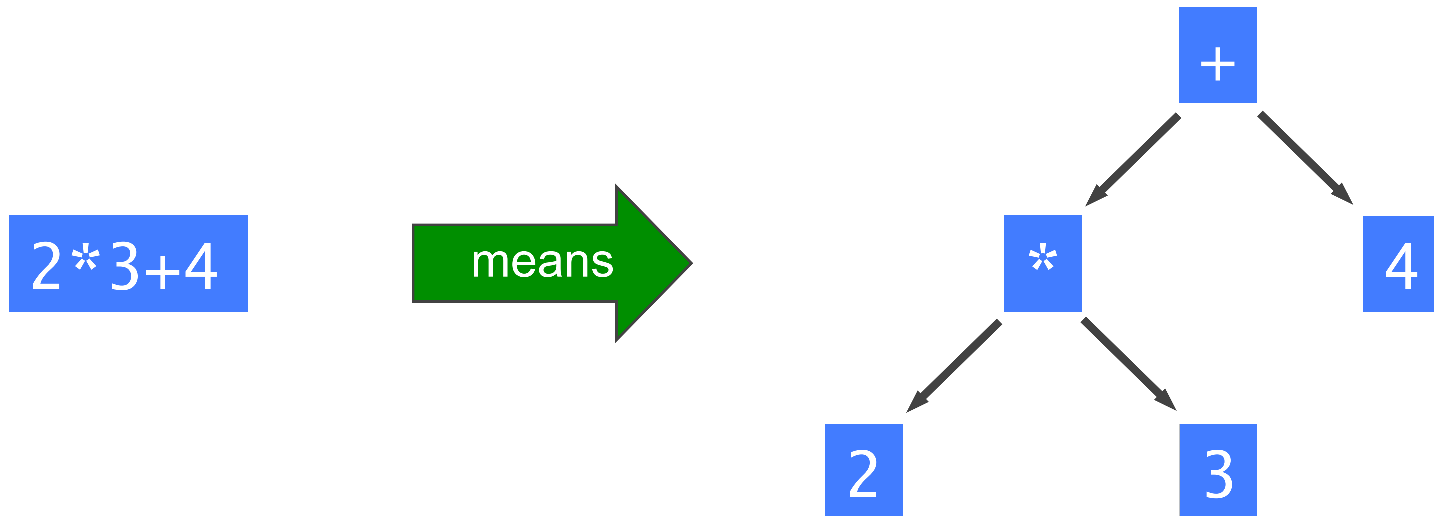
With some slides of Graham Hutton

# CREATING PARSERS


© 2011 University of Tuebingen

# What is a Parser?

A parser is a program that analyses a piece of text to determine its syntactic structure.

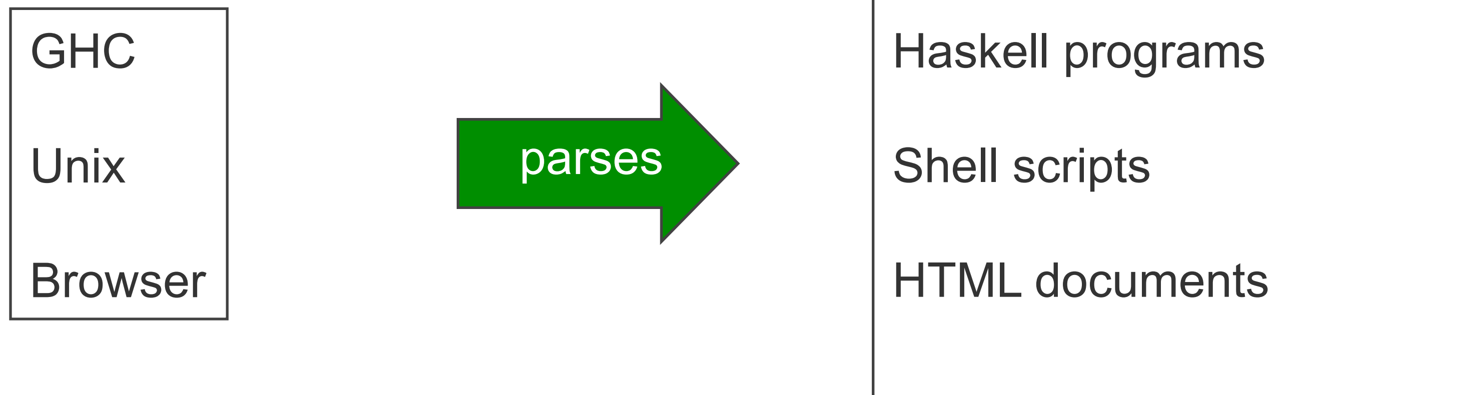





© 2011 University of Tuebingen

# Where Are They Used?

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



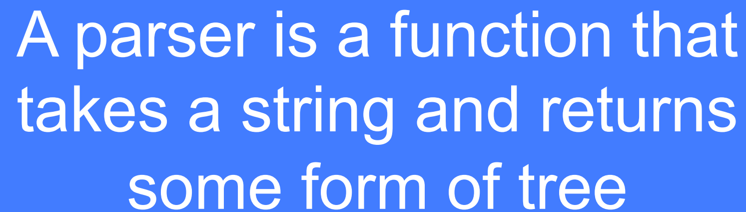


© 2011 University of Tuebingen

# The Parser Type

In a functional language such as Haskell, parsers can naturally be viewed as <u>functions</u>.

```
type Parser = String → Tree
```






© 2011 University of Tuebingen

# The Parser Type

However, a parser might not require all of its input string, so we also return any unused input:

```
type Parser = String → (Tree, String)
```


© 2011 University of Tuebingen

# The Parser Type

A string might be parsable in many ways, including none, so we generalize to a list of results:

```
type Parser = String → [(Tree, String)]
```



© 2011 University of Tuebingen

# The Parser Type

A parser might not always produce a tree, so we generalize to a value of <u>any type</u>.

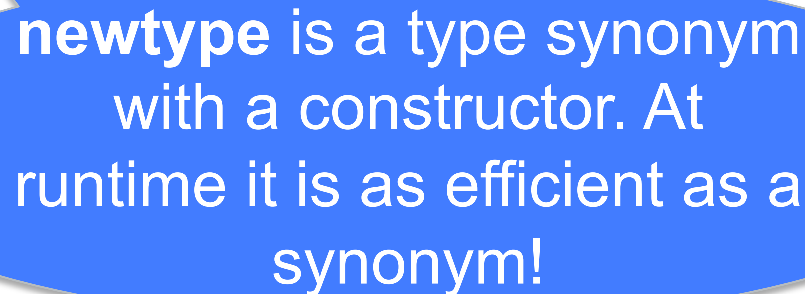
```
type Parser a = String → [(a, String)]
```


© 2011 University of Tuebingen

# The Parser Type

We cannot define class instance for type synonyms, and a data type seems to be a bit heavy:

```
newtype Parser a = P (String → [(a, String)])
```





© 2011 University of Tuebingen

*“A parser for things*
*Is a function from strings*
*To lists of pairs*
*Of things and strings”*


© 2011 University of Tuebingen

# Basic Parsers

We first define a parser that always succeeds with a given result value.

```
succeed :: a → Parser a
succeed v = P $ \inp → [(v, inp)]
```


© 2011 University of Tuebingen

# Basic Parsers

We can also define a parser that always fails

```
failure :: Parser a
failure = P $ \_ → []
```


© 2011 University of Tuebingen

# Basic Parsers

A parser that can consume the first element of a string:

```
item :: Parser Char
item = P $ \inp → case inp of
                    [] → []
                    (x:xs) → [(x,xs)]
```



© 2011 University of Tuebingen

# Running a Parser

The function parse applies a parser to a string:

```
parse :: Parser a → String → [(a, String)]
parse (P p) inp = p inp
```


© 2011 University of Tuebingen

# Running a Parser

We can now use our parsers:

```
> parse (succeed 1) “abc”
[(1, “abc”)]
> parse item “abc”
[(‘a’, “bc”)]
> parse item “”
[]
> parse failure “abc”
[]
```


© 2011 University of Tuebingen

# Changing Parser Results

Usually we do not just want the character that we parsed but do something with it.

```
(<$>) :: (a → b) → Parser a → Parser b
f <$> p = P $ \inp →
                [(f v, out)
                |(v, out) ← parse p inp ]
```


© 2011 University of Tuebingen

# Changing Parser Results

The type of <$> looks a bit like the signature of map:

```
(<$>) :: (a → b) → Parser a → Parser b
map   :: (a → b) → [a]      → [b]
```

We can define a generalised version of map of type:

```
fmap :: SomeClass f => (a → b) → f a → f b
```


© 2011 University of Tuebingen

# The Functor Class

In the prelude there is a class Functor that does just that!

```
class Functor f where
      fmap :: (a → b) → f a → f b
```

“The Functor class is used for types that can be mapped over”.

It is defined for a lot of types in the prelude ([], Maybe, Either a, (→) a, etc.


© 2011 University of Tuebingen

# The Functor Class

We can turn our Parser type into a Functor!

```
instance Functor Parser where
  fmap f p =
     P $ \inp → [(f v, out)
                |(v, out) ← parse p inp ]
```


© 2011 University of Tuebingen

# The Functor Class

Instances of the class Functor must adhere to some laws (not enforced):

```
fmap id == id
fmap (f . g) == fmap f . fmap g
```

It is the programmers responsibility to make sure these laws hold, other functions rely on these laws!


© 2011 University of Tuebingen

# The Functor Class

If we imports Data.Functor we get two extra functions that can be used:

```
(<$>) :: Functor f => (a → b) → f a → f b
```

(<$>) == fmap

```
(<$) :: Functor f => b → f a → f b
```

Replace all elements in structure f by an element of type b


© 2011 University of Tuebingen

# Examples

We can now write the following programs:

```
> parse (id <$> item) “abc”
[(‘a’, “bc”)]
> parse (toUpper <$> item )“abc”
[(‘A’, “bc”)]
> parse (1 <$ item) “abc”
[(1, “bc”)]
> parse (‘d’ <$ failure) “abc”
[]
> :t ((,) <$> item)
Parser (b -> (Char, b))
```


© 2011 University of Tuebingen

# Sequencing parsers

Using a sequencing combinator we can combine multiple parsers into one parser that consumes more than one element. For example:

```
expr = Parser (Char, Char, Char)
expr = (,,) <$> item <*> item <*> item
```

((,,) <$> item) ::
Parser (Char → Char
→ (Char,Char,Char)

(<*>) ::
Parser (a → b) →
Parser a → Parser b


© 2011 University of Tuebingen

# Sequencing parsers

Using a sequencing combinator we can combine multiple parsers into one parser that consumes more than one element.

```
(<*>) :: Parser (a → b) → Parser a → Parser b
f <*> g = P $ \inp → concat
                  [ parse (v <$> g) out
                  | (v, out) ← parse f inp]
```


© 2011 University of Tuebingen

# Applicative Functors

pure = succeed

…tations in this ways is captured by the class …plicative (module Control.Applicative).

```
class Functor f => Applicative f where
     pure  :: a -> f a
     (<*>) :: f (a -> b) -> f a -> f b
```

The class actually has more functions but you don’t have to provide them yourself!



© 2011 University of Tuebingen

# Applicative Functors

Just like Functors instances of Applicative Functors must adhere to certain laws:

```
pure id <*> v == v
pure (.) <*> u <*> v <*> w
     == u <*> (v <*> w)
pure f <*> pure x == pure (f x)
u <*> pure y == pure ($ y) <*> u
```


© 2011 University of Tuebingen

# Applicative Functors

When we have an instance of applicative we get a few more functions:

```
(*>) :: Functor f => f a → f b → f b
```

Perform two computations, but ignore the result of the first computation

```
(<*) :: Functor f => f a → f b → f a
```



© 2011 University of Tuebingen

# Examples

We can now write the following programs:

```
> parse ((,) <$> item <*> item) “abc”
[((‘a’, ‘b’), “c”)]
> parse (toUpper <$> item *> item)“abc”
[(‘A’, “bc”)]
> parse (toUpper <$ item <*> item) “abc”
[(‘B’, “c”)]
> parse (toUpper <$> item *> item) “abc”
[(‘b’, “c”)]
> parse ((,) <$> item <*> failure) “abc”
[]
```


© 2011 University of Tuebingen

# Recognizing Specific Characters

With a parser we want to process specific languages, the parser *sat* only succeeds if the parsed item satisfies a certain predicate.

```
sat :: (Char → Bool) → Parser Char
sat p = P $ \inp → case inp of
                 [] → []
                 (x:xs) → if p x then [(x, xs)]
                                 else []
```


© 2011 University of Tuebingen

# Derived Parsers

```
letter :: Parser Char
letter = sat isAlpha

alphanum :: Parser Char
alphanum = sat isAlphaNum

char :: Char → Parser Char
char c = sat (==c)

digit, lower, upper :: Parser Char
```


© 2011 University of Tuebingen

# Combining Parsers

We can now construct parsers that recognize specific words by <u>combining</u> other parsers.

```
string :: String → Parser String
string []     = succeed []
string (x:xs) = (:) <$> char x <*> string xs
```


© 2011 University of Tuebingen

# Choice

Sometimes there are more valid parse options. Parsing with both parser f and g would be possible.

```
(<|>) :: Parser a → Parser a → Parser a
f <|> g = P $ \inp → parse f inp
                         ++ parse g inp
```


© 2011 University of Tuebingen

# The Alternative Class

In the module Control.Applicative a class Alternative is defined that defines a function (<|>). All instances of Alternative must also be Applicative Functors.

empty = failure

```
class Applicative f => Alternative f where
      empty :: f a
      (<|>) :: f a → f a → f a
```


© 2011 University of Tuebingen

# Parsing Many Elements

Sometimes we want to apply one parser as many times as possible.

```
many :: Parser a → Parser [a]
many p = ((:) <$> p <*> many p) <|> succeed []
```

Actually many is defined for all instance of Alternative:

```
many :: Alternative f => f a → f [a]
many p = ((:) <$> p <*> many p) <|> pure []
```


© 2011 University of Tuebingen

# Examples

We can now write the following programs:

```
> parse (many item) “abc”
[("abc",""),("ab","c"),("a","bc"),("","abc")]
> parse (many $ char ‘a’) “abc”
[("a","bc"),("","abc")]
> parse ((++) <$> many (char ‘a’)
              <*> many (char ‘b’)) “aaabc”
[("aaab","c"),("aaa","bc"),("aa","abc"),
 ("a","aabc"),("","aaabc")]
> parse (many $ char ‘a’) “bcd”
[(“”, “bcd”)]
```


© 2011 University of Tuebingen

# An Optional Parser

Sometimes we want to parse something if it is there but not fail if it is not there…

```
try :: Parser a → Parser (Maybe a)
try p = Just <$> p <|> succeed Nothing
```



© 2011 University of Tuebingen

# Quiz

Define a parser that consumes as much as it can (like many) but consumes at least one element.

```
some :: Parser a → Parser [a]
some p = (:) <$> p <*> many p
```


© 2011 University of Tuebingen

# Parsing Numbers

With all the basic parser combinators in place we can define a parser that recognises numbers (consisting out of multiple digits).

```
nat :: Parser Int
nat = read <$> some digit
```


© 2011 University of Tuebingen

# Quiz

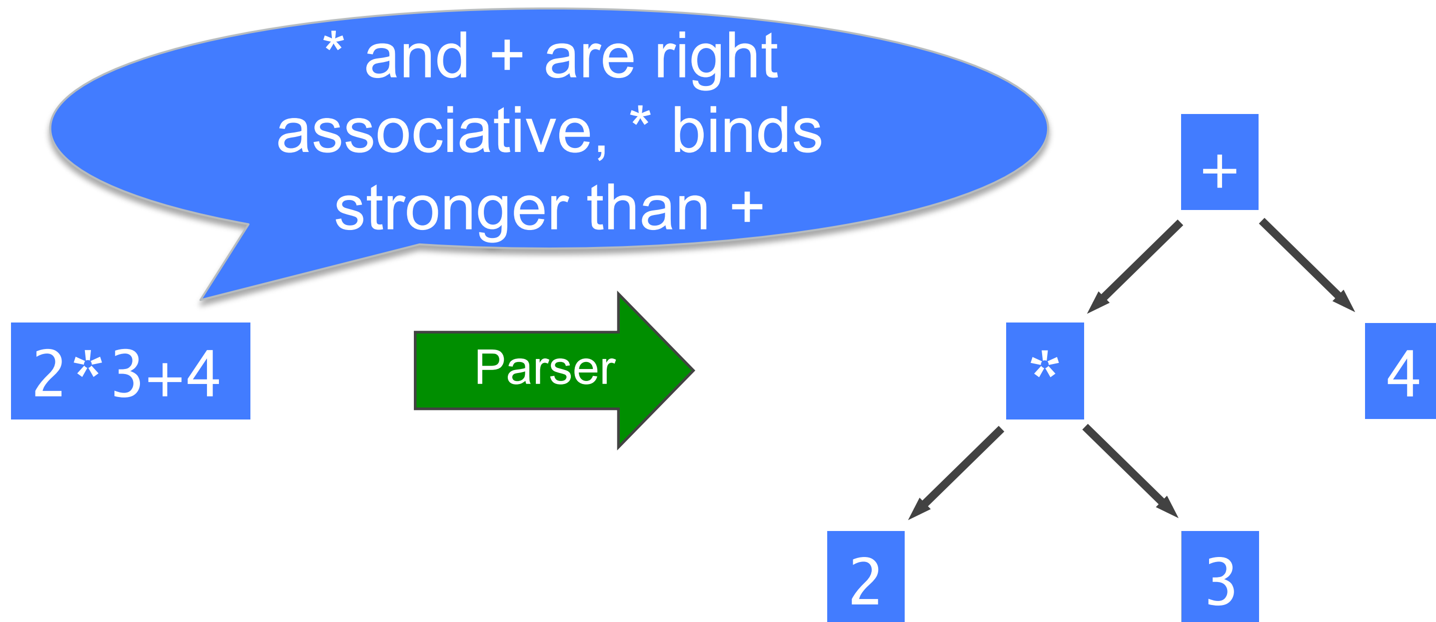
Define a parser that parses the name of a variable. A variable name consists out of alpha numeric characters starting with a lowercase letter.

```
var :: Parser String
var = (:) <$> lower <*> many alphanum
```


© 2011 University of Tuebingen

# Parsing Expressions

With the parsers we have defined so far we can parse simple arithmetic expressions.





© 2011 University of Tuebingen

# The Grammar of Expressions

Formally, the syntax of such expressions is defined by the following context free grammar:

```
expr   → term '+' expr | term
term   → factor '*' term | factor
factor → nat | '(' expr ')'
nat    → digit nat | digit
digit  → '0' | … | '9'
```


© 2011 University of Tuebingen

# The Grammar of Expressions

However, for reasons of efficiency, it is important to <u>factorise</u> the rules for *expr*, *term* and *nat*:

```
expr   → term (‘+’ expr | ε)

term   → factor (‘*’ term | ε)

factor → nat | ‘(‘ expr ‘)’

nat    → digit (nat | ε)

digit  → ‘0’ | … | ‘9’
```


© 2011 University of Tuebingen

# The Expr Type

To represent the expression tree we define the following type:

```
data Expr = Nat Int
          | Add Expr Expr
          | Mul Expr Expr
```

With an evaluation function:

```
eval :: Expr → Int
eval (Nat i)       = i
eval (Add e1 e2) = eval e1 + eval e2
eval (Mul e1 e2) = eval e1 * eval e2
```


© 2011 University of Tuebingen

# A Parser for Factors

```
factor → nat | ‘(‘ expr ‘)’
```

We first write a parser that parsing something between parentheses.

```
paren :: Parser a → Parser a
paren p = char ‘(‘ *> p <* char ‘)’
```

Assuming we have a parser *expr* we can now define:

```
factor :: Parser Expr
factor = (Nat <$> nat) <|> paren expr
```


© 2011 University of Tuebingen

# A Parser for Terms

```
term → factor ('*' term | ε)
```

A term is either a factor or a factor multiplied by another term.

```haskell
term :: Parser Expr
term = t <$> factor <*> try (char '*' *> term)
  where
   t fac Nothing       = fac
   t fac (Just term) = Mul fac term
```


© 2011 University of Tuebingen

# A Parser for Exprs

```
expr → term ('+' expr | ε)
```

An expr is either a term or a term added to another expr.

```
expr :: Parser Expr
expr = e <$> term <*> try (char '+' *> expr)
  where
   e term Nothing      = term
   e term (Just expr) = Add term expr
```


© 2011 University of Tuebingen

# Examples

We can now write the following expressions:

```
> parse expr “1”
[(Nat 1, [])]
> Parse expr “1+2”
[(Add (Nat 1) (Nat 2),""),(Nat 1,"+2")]
> parse expr “2*3+4”
[(Add (Mul (Nat 2) (Nat 3)) (Nat 4),""),
 (Mul (Nat 2) (Nat 3),"+4"),(Nat 2,"*3+4")]
> parse expr “1 + 2”
[(Nat 1, “ + 2”)]
```


© 2011 University of Tuebingen

# White Space

We want to ignore white space but use it to indicate the start of a new word.

```
space :: Parser ()
space = () <$ many (sat isSpace)
```

```
token :: Parser a → Parser a
token p = space *> p <* space
```


© 2011 University of Tuebingen

# White Space

Using the *token* parser we can define:

```
identifier :: Parser String
identifier = token var

natural :: Parser Int
natural = token nat

symbol :: String → Parser String
symbol s = token (string s)
```

We can use these to rewrite our expression parser and add support for white space.


© 2011 University of Tuebingen

```
expr :: Parser Expr
expr = e <$> term <*> try (symbol “+” *> expr)
  where
   e term Nothing = term
   e term (Just expr) = Add term expr

term :: Parser Expr
term = t <$> factor <*> try (symbol “*” *> term)
  where
   t fac Nothing = fac
   t fac (Just term) = Mul fac term

factor :: Parser Expr
factor = (Nat <$> natural) <|> paren expr

paren :: Parser a -> Parser a
paren p = symbol “(“ *> p <* symbol “)”
```

# A Word of Warning

The parsers defined in this lecture are very simple and elegant. And they actually work!

But, they are inefficient and slow.

A very efficient cutting-edge parser library is the uu-parsinglib.

The most widely used parser library is Parsec.


© 2011 University of Tuebingen

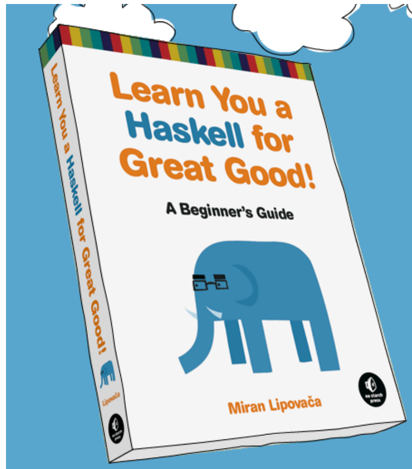
# Parsing a Small Language

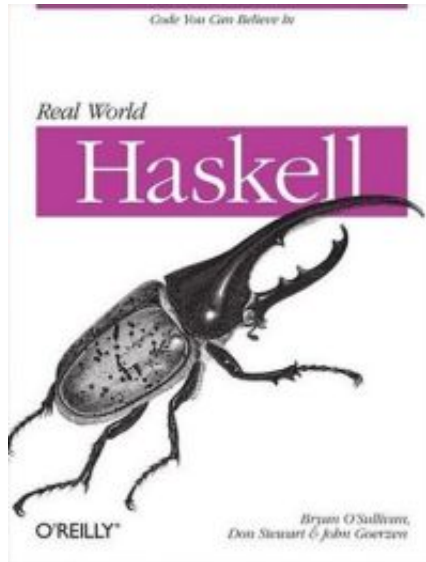
With the parsers defined in this lecture we can parse a small language.

In this weeks set of exercises you will use these parsers for the questions.


© 2011 University of Tuebingen

# Reading material




- Learn you a Haskell:
  - Chapter 8, section the Functor typeclass
  - Chapter 11
- Parallel Parsing Processes
  - Koen Claessen
- Parsec Documentation & Tutorial
- Real world Haskell
  - Chapter 16 (about old Parsec)


© 2011 University of Tuebingen