Domain Specific Embedded Languages

Lecture 2,

Designing and Using Combinators

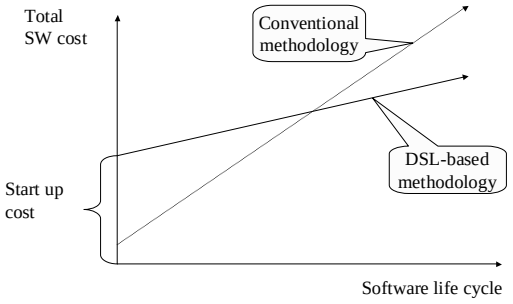
John Hughes

What is a Domain Specific Language?

A programming language tailored for a particular application domain, which captures precisely the semantics of the application domain -- no more, no less.

> *A DSL allows one to develop software for a particular application domain quickly, and effectively, yielding programs that are easy to understand, reason about, and maintain.*
>
> Hudak

The Cost Argument



The Problem with DSLs

- DSLs tend to grow: adding procedures, modules, data structures…
- Language design is *difficult* and *time-consuming*; large parts are not domain specific.
- Implementing a compiler is *costly* (code-generation, optimisation, type-checking, error messages…)

Start up costs may be substantial!

Domain Specific *Embedded* Languages

Why not *embed* the DSL as a library in an existing *host* language?

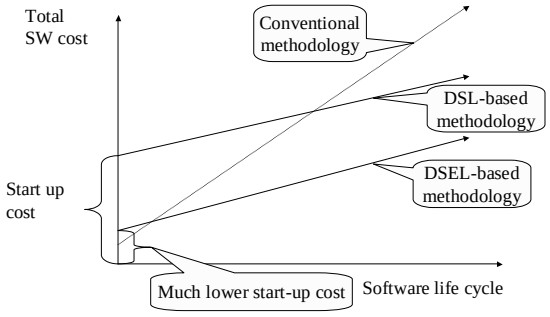
+

- Inherit non-domain-specific parts of the design.
- Inherit compilers and tools.
- Uniform "look and feel" across many DSLs
- DSLs integrated with full programming language, and with each other.

-

- Constrained by host language syntax and type system.

The Cost Argument Again



What Makes Haskell a Suitable Host?

- Higher-order functions.

  DSL library *constructs programs*, naturally represented as functions.

- Lazy evaluation.

  Permits recursive definitions of DSL programs, "**if-then-else**" as a function.

- Polymorphism and class system.

  Powerful type system to embed DSL's types in. Classes permit definitions by "recursion over types".

Example: DSL vs DSEL for Parsing

- YACC is a well-known DSL for parsers.
- HAPPY is a YACC-like tool for Haskell.
- Parsing combinators are a classic DSEL for Haskell.
- Compare approaches in an evaluator for arithmetic expressions.

Expression BNF

```
expr ::= term | term + term
term ::= factor | factor * factor
factor ::= constant | (expr)
constant ::= digit*
```

Examples

1+2, 1+2*3, (1+2)*3, but *not* 1+2+3

Lexical Analyser

Happy requires an external lexical analyser:

```
data Token = TokenInt Int
           | TokenPlus
           | TokenTimes
           | TokenBra
           | TokenKet
  deriving (Show,Eq)

lexer :: String -> [Token]
```

Happy Grammar: Tokens

```
%name calc
%tokentype { Token }
%token
  int {TokenInt $$}
  '+' {TokenPlus}
  '*' {TokenTimes}
  '(' {TokenBra}
  ')' {TokenKet}
%%
```

Name of function to generate

Relate token names to Haskell values

Happy Grammar: Syntax

```
Exp   : Term '+' Term     {$1+$3}
      | Term              {$1}
Term  : Factor '*' Factor {$1*$3}
      | Factor            {$1}
Factor: int               {$1}
      | '(' Exp ')'       {$2}
```

Actions to evaluate

End Result

```
Calc> calc (lexer "(1+2)*3")
9
Calc>
```

A Combinator Parser

```
expr = do t <- term
          literal TokenPlus
          t' <- term
          return (t+t')
       ||| term

term = do f <- factor
          literal TokenTimes
          f' <- factor
          return (f*f')
       ||| factor
```

```
factor =
    do  literal TokenBra
        e <- expr
        literal TokenKet
        return e
 ||| do TokenInt n <-
           satisfy isTokenInt
        return n

calc = runParser expr
```

A Combinator Parser

Sequencing using **do**

```
expr = do t <- term
          literal TokenPlus
          t' <- term
          return (t+t')
       ||| term

term = do f <- factor
          literal TokenTimes
          f' <- factor
          return (f*f')
       ||| factor
```

```
factor =
    do  literal TokenBra
        e <- expr
        literal TokenKet
        return e
 ||| do TokenInt n <-
           satisfy isTokenInt
        return n

calc = runParser expr
```

tor Parser

Sequencing using **do**

Results named in the usual way

```
expr = do t <- term
          literal TokenPlus
          t' <- term
          return (t+t')
       ||| term

term = do f <- factor
          literal TokenTimes
          f' <- factor
          return (f*f')
       ||| factor
```

```
factor =
    do  literal TokenBra
        e <- expr
        literal TokenKet
        return e
 ||| do TokenInt n <-
           satisfy isTokenInt
        return n

calc = runParser expr
```

to

Sequencing using **do**

Results named in the usual way

Literal tokens (cf %token section)

```
expr = do t <- term
          literal TokenPlus
          t' <- term
          return (t+t')
       ||| term

term = do f <- factor
          literal TokenTimes
          f' <- factor
          return (f*f')
       ||| factor
```

```
factor =
    do  literal TokenBra
        e <- expr
        literal TokenKet
        return e
 ||| do TokenInt n <-
           satisfy isTokenInt
        return n

calc = runParser expr
```

to

Sequencing using **do**

Results named in the usual way

Literal tokens (cf %token section)

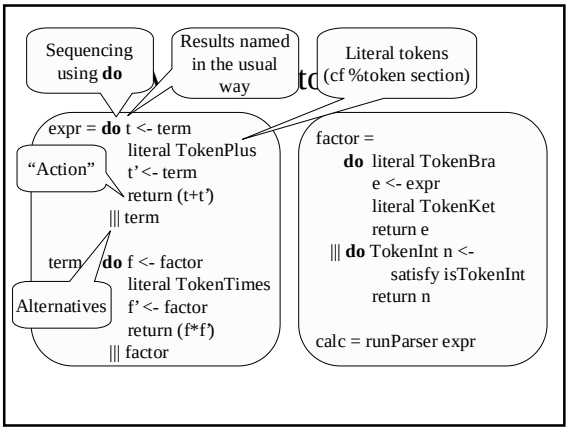
"Action"

```
expr = do t <- term
          literal TokenPlus
          t' <- term
          return (t+t')
       ||| term

term = do f <- factor
          literal TokenTimes
          f' <- factor
          return (f*f')
       ||| factor
```

```
factor =
    do  literal TokenBra
        e <- expr
        literal TokenKet
        return e
 ||| do TokenInt n <-
           satisfy isTokenInt
        return n

calc = runParser expr
```



Abstracting Common Patterns

- The combinator parser is not quite as concise as the DSL one -- we cannot invent new syntax!
- But we can exploit the power of Haskell to *abstract* common patterns!

Example: Binary operators

```
binOp oper rand =
   do   x <- rand
        op <- oper
        y <- rand
        return (x `op` y)
||| rand
```

```
expr =
   binOp (do literal TokenPlus
             return (+))
         term
```

Further Abstraction

```
expr =
   binOp (do literal TokenPlus
             return (+))
         term
```

Appears for every operator

Further Abstraction

```
expr =
   binOp (literal TokenPlus `giving` (+))
         term
```

Further Abstraction

```
expr = binOp (literal TokenPlus `giving` (+)) term
term = binOp (literal TokenTimes `giving` (*)) factor
```

Extending the Operators

```
expr = binOp (literal TokenPlus `giving` (+)
              ||| literal TokenMinus `giving` (-)) term
term = binOp (literal TokenTimes `giving` (*)
              ||| literal TokenDivide `giving` div) factor
```

Thanks to the embedding in Haskell, we can "extend the language" to obtain concise parsers in each particular case.

Extending the Grammar

- What about accepting 1+2+3?
- Operators associate *to the left*: 1-2-3 = (1-2)-3
- Easy with Happy!

```
Exp   : Exp '+' Term     {$1+$3}
      | Term             {$1}
Term  : Term '*' Factor  {$1*$3}
      | Factor           {$1}
Factor: int              {$1}
      | '(' Exp ')'      {$2}
```

Extending the Combinator Parser?

```
expr = do t <- expr
          literal TokenPlus
          t' <- term
          return (t+t')
       ||| term
```

Left recursion makes the parser loop!

Happy works because it *compiles* the left recursion away.

Extending the Combinator Parser: The Right Way

Transform the grammar: parse as *right* recursion, convert the result to associate left!

E ::= E Op T | T ⇨ E ::= T {Op T}*

```
binOp oper rand =
  do x <- rand
     op_ys <- many (oper `pair` rand)
     return
       (foldl (\z (op,y) -> z `op` y) x op_ys)
```

```
p `pair` q =
  do x <- p
     y <- q
     return (x,y)
```

expr and term need not change at all!

What is the DSEL?

```
p ::= do {x <- p}* p
    | return e
    | literal tok
    | satisfy pred
    | p ||| p
    | many p
    | p `pair` p
    | binOp p p
    | p `giving` e
```

It is very easy to extend the language further!

Comparison

- DSL (Happy) is syntactically convenient.
- DSL can analyse and transform the program; harder with combinators.

- DSEL (combinators) achieves brevity by *extension*, using Haskell's power of abstraction.
- DSEL is *strongly typed*, which Happy is not! (Inherits existing type-checker)
- DSEL permits *experimental* language design.

What are DSELs used for?

A *wide variety* of applications!

3 Examples:

- QuickCheck -- software testing
- Pretty -- pretty-printing data-structures
- Wash/Cgi -- server side web scripting

QuickCheck

The programmer writes *properties* in the source code, which are tested by a testing tool.

The property language is a DSEL!

```
prop_Insert :: Integer -> [Integer] -> Property
prop_Insert x xs =
   ordered xs ==> ordered (insert x xs)
```

```
prop_Insert :: Integer -> Property
prop_Insert x =
   forAll orderedList $ \xs -> ordered (insert x xs)
```

Pretty

A *pretty-printer* for binary trees:

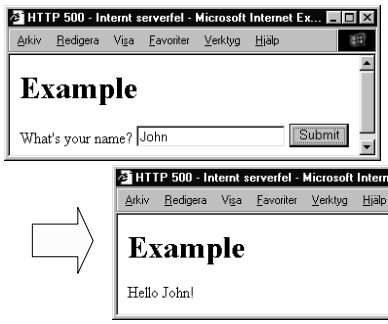
```
data Tree a = Leaf | Node a (Tree a) (Tree a)
```

```
prettyTree Leaf = text “Leaf”
prettyTree (Node a left right) =
   text (“Node ”++show a++“ ”)
     <> sep [prettyTree left, prettyTree right]
     <> text “)”
```

Example: outputs

```
(Node 2 (Node 1 Leaf Leaf)
        (Node 1 Leaf Leaf))
```

Wash/CGI



A language for defining and processing HTML forms.

Wash/CGI

```
main = run $ ask $ page $ makeForm $
  do text "What's your name? "
     name <- textInputField empty
     submitField (hello (value name))
       (fieldVALUE "Submit")

hello (Just name) = htell $ page $
   text ("Hello "++name++"!")

page x = standardPage "Example" x
```

```
f $ g $ h $ e
=
f (g (h e))
```

Callback function

Callback happens in an *entirely separate run*!

Summary

•DSLs make applications easy to write, but are (a) costly to design and implement, (b) often rather weak. *(e.g. Happy has no function abstraction or type checking)*

•DSELs are *almost* as convenient to use, but also inherit all the power of Haskell.

•DSELs are very easy to extend, easy to design and implement, easy to experiment with.

•DSELs can address a *wide variety* of application areas.