Introduction

• Domain Specific Language (DSL)
  – notation oriented to some problem domain
  – specialised types & control structures

• DSL is a library + syntax
  – library: what it can do
  – syntax: how to tell it what to do
Where do DSLs come from?

• tiresome writing lots of small programs for same problem area
• end up using same set of:
  – programming tropes
  – abstractions
    • types
    • control
Where do DSLs come from?

• construct an ad hoc command based framework
  – scripts to invoke and configure individual program components
• nice to deploy:
  – consistent notation oriented to problem area
  – built in constructs capturing specialised abstractions
• how to define & implement languages?
Language

• symbolic system for communicating state changing meanings
• components:
  – alphabet
  – symbols/lexicon
  – syntax
  – semantics
Example

• calculator language
  – integers
  – define variables
  – expressions output values
• e.g.
  
a = 7;
b = 4;
a*(b+4) ==> 56
Medium

• how utterances are conveyed
• must be capable of bearing distinguishable units of difference
  – sounds in air
  – marks on paper
  – electro-magnetic waves
  – charges in transistors
• fundamental but not relevant...
Alphabet

• basic distinguishable units of expression
• not meaningful in themselves
• e.g.

  * letter: a b c d ... z
  * digit: 0 1 2 3 ... 9
  * punctuation: + - * / ( ) = ;
Symbols/lexicon

- “words” in “dictionary”
- alphabet sequences
- smallest meaningful units
- list them
- regular expressions/Chomsky Type 3
- e.g.

  \( \text{operator} \rightarrow + | - | * | / | ( | ) | ; | = \)

  \( \text{identifier} \rightarrow \text{letter} | \text{letter identifier} \)

  e.g. a be sea
Symbols/lexicon

• e.g.

integer -> digit | positive int
positive -> 1 | 2 ... | 9
int -> digit | digit int

e.g. 0 12 45700 but not 00 012
Syntax

• grammar
• well formed symbol sequences
• not necessarily meaningful
• concrete syntax
  - representational structure
• abstract syntax
  - meaningful structure
Concrete Syntax

• context free/Chomsky Type 2
• Backus Naur Form (BNF)
• e.g.

```
commands -> command | command ; commands
command -> identifier = expression | expression
expression -> term | term + term | term - term
term -> base | base * base | base / base
base -> identifier | integer | ( expression )
```
Concrete Syntax

• use grammar to:
  – parse symbol sequence
  – build internal representation
  • parse tree
e.g.
  
  a = 7;
  b = 4;
  a*(b+4)
Concrete Syntax

```
commands

command

identifier = expression

command ;

commands

expression

term

base

integer

a = 7;
b = 4;
a*(b+4)
```

```
commands -> command | command ; commands
command -> identifier = expression | expression
expression -> term | term + term | term - term
term -> base | base * base | base / base
base -> identifier | integer | ( expression )
```
Abstract Syntax

• concrete syntax contains irrelevant information for meaning
• e.g. don’t care that:
  7 is
    \textit{integer} is
      \textit{base} is
        \textit{term} is
          expression
Abstract Syntax

• simplify grammar
  – to reflect key constructs
  – e.g. commands & expressions with identifiers & integers
  – drop irrelevant punctuation e.g. ; ( . . . )
  \[ c \rightarrow c\ c \mid id = e \mid e \]
  \[ e \rightarrow id \mid int \mid e + e \mid e - e \mid e \ast e \mid e / e \]
• doesn’t matter that grammar is ambiguous
• use to derive structure of abstract syntax tree
Abstract Syntax

\[
a = 7; \\
b = 4; \\
a*(b+4)
\]

\[
c -> c \mid id = e \mid e \\
e -> id \mid int \mid e + e \mid e - e \mid e * e \mid e / e
\]
Semantics

• what constructs mean
• express in meta-language
  – informal – natural language
  – formal – some theory of computability
    • e.g. number theoretic predicate calculus
    • e.g. set theory
  – static semantics: types
  – dynamic semantics: run time behaviour
Semantics

• dynamic
• denotational semantics
• Scott-Strachey
  – meanings expressed as functions
  – λ calculus with syntactic sugar
  – compositional on abstract syntax
Semantics

state: $identifier \rightarrow integer$
• state maps syntactic $identifier$s to semantic integers

me: $expression \rightarrow state \rightarrow integer$
• meaning of an $expression$ given a state is an integer

mc: $command \rightarrow state \ast output \rightarrow state \ast output$
• meaning of a $command$ given a old state and old output is a new state and new output
Semantics

me [int] state = value([int])

• value == valuation function from syntactic int to semantic integer

me [id] state = state([id])

• apply state to id to return associated integer

me [e₁+e₂] state = m [e₁] state + m [e₂] state
me [e₁−e₂] state = m [e₁] state − m [e₂] state
me [e₁*e₂] state = m [e₁] state * m [e₂] state
me [e₁/e₂] state = m [e₁] state / m [e₂] state
Semantics

\[
\text{mc} [c_1 c_2] (\text{state}, \text{output}) = \\
\text{mc} [c_2] (\text{mc} [c_1] (\text{state}, \text{output}))
\]

- meaning of command sequence given old state and output is:
  - meaning of \(c_2\) using state and output from...
  - meaning of \(c_1\) with old state and output
Semantics

\[ mc [id = e] (\text{state}, \text{output}) = \]
\[ (\text{state}\{id/(\text{me}\ [e] \text{state})\}, \text{output}) \]

• meaning of definition given old state and output is:
  – old state updated with \( id \) associated with value of \( e \) in old state
  – old output
Semantics

mc \[e\] (state,\ output) = (state, \ output++me \ [e]\ state)

• meaning of expression given old state and output is:
  – old state, and
  – old output augmented with value of expression in old state
Semantics

\[ \varepsilon == \text{empty state} \]
\[ \{\} == \text{empty output} \]

\[ \text{mc} \left[ a=7 ; b=4 ; a \times (b+4) \right] (\varepsilon,\{\}) \]
\[ \text{mc} \left[ b=4 ; a \times (b+4) \right] (\text{mc} \left[ a=7 \right] (\varepsilon,\{\})) \]
\[ \text{mc} \left[ a \times (b+4) \right] (\text{mc} \left[ b=4 \right] (\text{mc} \left[ a=7 \right] (\varepsilon,\{\}))) \]
\[ \text{mc} \left[ a \times (b+4) \right] (\text{mc} \left[ b=4 \right] (\{a->7\},\{\})) \]
\[ \text{mc} \left[ a \times (b+4) \right] (\{b->4, a->7\},\{\}) \]
\[ (\{b->4, a->7\},\{\text{me} \left[ a \times (b+4) \right] \{b->4, a->7\}\}) \]
Semantics

$$\{\text{b->4, a->7}\}, \{\text{me [a] \{b->4, a->7\}} \ast \text{me [b+4] \{b->4, a->7\}}\} \not\models \{\text{b->4, a->7}\}, \{\text{me [a] \{b->4, a->7\}} \ast (\text{me [b] \{b->4, a->7\}} + \text{me [4] \{b->4, a->7\}})\} \not\models \{\text{b->4, a->7}\}, \{7*(4+4)\} \not\models \{\text{b->4, a->7}\}, \{56\}$$
Implementation

• front end

source text

lexical analyser

symbols

syntax analyser

abstract syntax tree

static semantics

abstract syntax tree’
Implementation

• back end

abstract syntax tree’

inputs
code generator

target code

inputs

interpreter

outputs

interperter/computer

outputs
Interpreter

• implement in Haskell
• choose AST representation

\[
c \rightarrow c \cdot c | id = e | e
\]

\[
e \rightarrow id | int | e + e | e - e | e \cdot e | e / e
\]

```
data SYMBOL = SADD | SSUB | SMULT | SDIV | ...
data AST = ID String | INT Int |
          DEF(AST,AST) |
          EXP(SYMBOL,AST,AST)
```

\[
c \cdot c == [AST]
\]
Interpreter

data SYMBOL = SADD | SSUB | SMULT | SDIV | ... 
data AST = ID String | INT Int | 
    DEF(AST,AST) | 
    EXP(SYMBOL,AST,AST)

a = 7;
b = 4;
a*(b+4) = 
[DEF(ID “a”,INT 7),
  DEF(ID “b”,INT 4),
  EXP(SMULT,ID “a”,EXP(SADD,ID “b”,INT 4))]
Interpreter

• choose representations for semantic entities
  – \textit{identifier} == \texttt{String}
  – integer == \texttt{Int}

• could model state as higher order function

• more flexible to use data structure + look up

• e.g. list

state: \textit{identifier} -> integer \implies [(\texttt{String}, \texttt{Int})]

e.g. \{(b\rightarrow 4, a\rightarrow 7) \implies [(“b”, 4), (“a”, 7)]
Interpreter

\[ mc: command \rightarrow \text{state} \times \text{output} \rightarrow \text{state} \times \text{output} \]
\[ mc \ [c_1 \ c_2] \ (\text{state}, \text{output}) = \ mc \ [c_2] \ (mc \ [c_1] \ (\text{state}, \text{output})) \]

\[ mCs :: \text{[AST]} \rightarrow \]
\[ \ (([\text{String}, \text{Int}]], [[\text{Int}]]) \rightarrow \]
\[ \ (([\text{String}, \text{Int}]], [[\text{Int}]]) \]
\[ mCs \ [] \ (\text{state}, \text{output}) = \ (\text{state}, \text{output}) \]
\[ mCs \ \ (c1:c2) \ \ (\text{state}, \text{output}) = \]
\[ \ mCs \ c2 \ \ (mc \ c1 \ (\text{state}, \text{output}))  \]
mc [id = e] (state, output) =
(state{id/(me [e] state)}, output)
mc [e] (state, output) = (state, output++me [e] state)

mC :: AST -> ([(String, Int)], [Int]) ->
    ([(String, Int)], [Int])
mC (DEF(ID i, e)) (state, output) =
    ((i, mE e state): state, output)
mC e (state, output) =
    (state, output++[mE e state])
Interpreter

me: expression -> state -> integer
me [int] state = value([int])
me [id] state = state([id])

mE :: AST -> [(String,Int)] -> Int
mE (INT n) _ = n
mE (ID i) state = lookUp i state

lookUp v [] = error ("can't find "+v++"\n")
lookUp v ((v1,i1):t) = if v==v1
  then i1
  else lookUp v t
Interpreter

me \[e_1+e_2\] state = m \[e_1\] state + m \[e_2\] state
me \[e_1-e_2\] state = m \[e_1\] state − m \[e_2\] state
me \[e_1\times e_2\] state = m \[e_1\] state * m \[e_2\] state
me \[e_1/e_2\] state = m \[e_1\] state / m \[e_2\] state

mE \((\text{EXP}(\text{SADD}, e_1, e_2))\) state =
(mE e_1 state) + (mE e_2 state)
mE \((\text{EXP}(\text{SSUB}, e_1, e_2))\) state =
(mE e_1 state) − (mE e_2 state)
mE \((\text{EXP}(\text{SMULT}, e_1, e_2))\) state =
(mE e_1 state) * (mE e_2 state)
mE \((\text{EXP}(\text{SDIV}, e_1, e_2))\) state =
(mE e_1 state) `div` (mE e_2 state)