Domain Specific Languages
1: Defining and Implementing Languages

Greg Michaelson
School of Mathematical & Computer Sciences
Heriot Watt University

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.

  operator -> + | – | * | / | (|) | ; | =

  identifier -> *letter* | *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*)

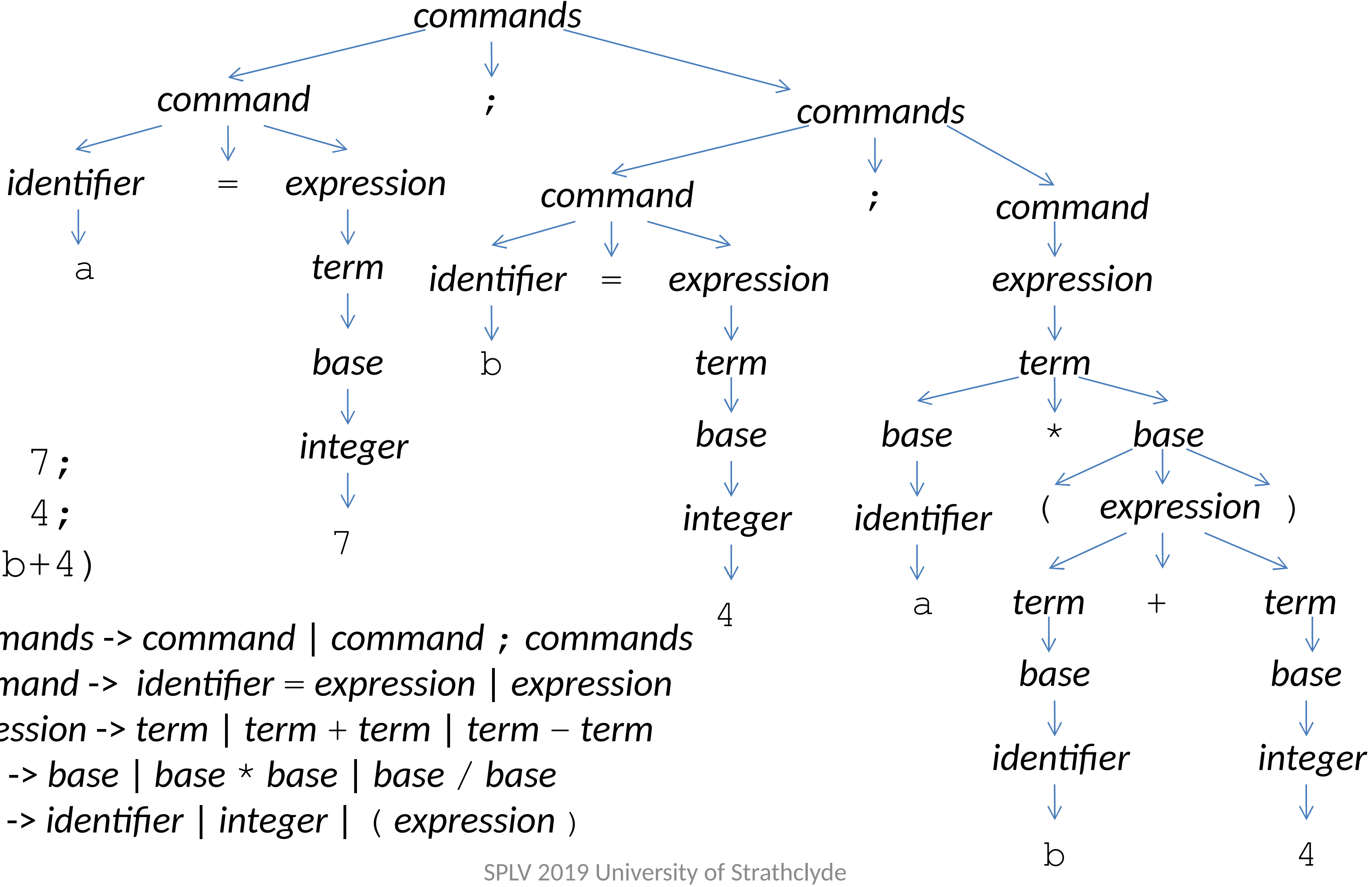
Concrete Syntax

- use grammar to:
  - parse symbol sequence
  - build internal representation
    - parse tree

e.g.

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

Concrete Syntax



commands

command *;* *commands*

identifier = *expression* *command* *;* *command*

a *term* *identifier* = *expression* *expression*

base b *term* *term*

integer *base* *base* * *base*

7 *integer* *identifier* (*expression*)

4 a *term* + *term*

base *base*

identifier *integer*

b 4

```
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

    integer is

      base is

        term is

          expression

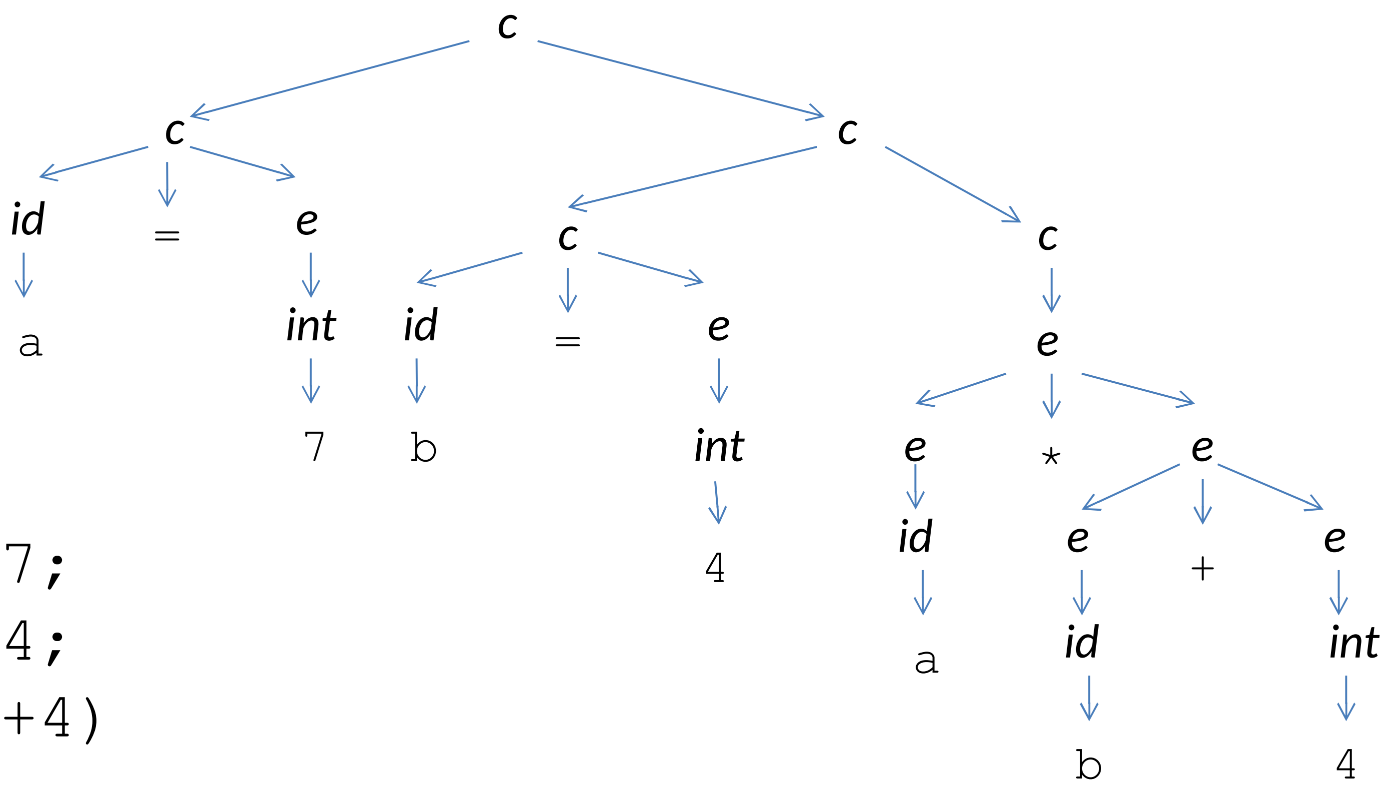
Abstract Syntax

- simplify grammar
  - to reflect key constructs
  - e.g. *commands* & *expressions* with *identifiers* & *integers*
  - drop irrelevant punctuation e.g. `;` `(...)`

  c -> c c | id = e | e

  *e -> id | int | e + e | e – e | e * e | 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 c* | *id* = *e* | *e*

e -> *id* | *int* | *e* + *e* | *e* – *e* | *e* * *e* | *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* -> integer

- state maps syntactic *identifiers* to semantic integers

me: *expression* -> state -> integer

- meaning of an *expression* given a state is an integer

mc: *command* -> state * output -> state * 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 [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*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

Semantics

mc [c_1 c_2] (state,output) =

mc [c_2] (mc [c_1] (state, 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*] (state,output) =

(state{*id*/(me [*e*] state)}, 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

ε == empty state

{} == empty output

mc [`a=7;b=4;a*(b+4)`] (ε,{}) ㅍ

mc [`b=4;a*(b+4)`] (mc [`a=7`] (ε,{}))

mc [`a*(b+4)`](mc [`b=4`] (mc [`a=7`] (ε,{}))) ㅍ

mc [`a*(b+4)`](mc [`b=4`] ({`a`->7},{})) ㅍ

mc [`a*(b+4)`] ({`b`->4, `a`->7},{})) ㅍ

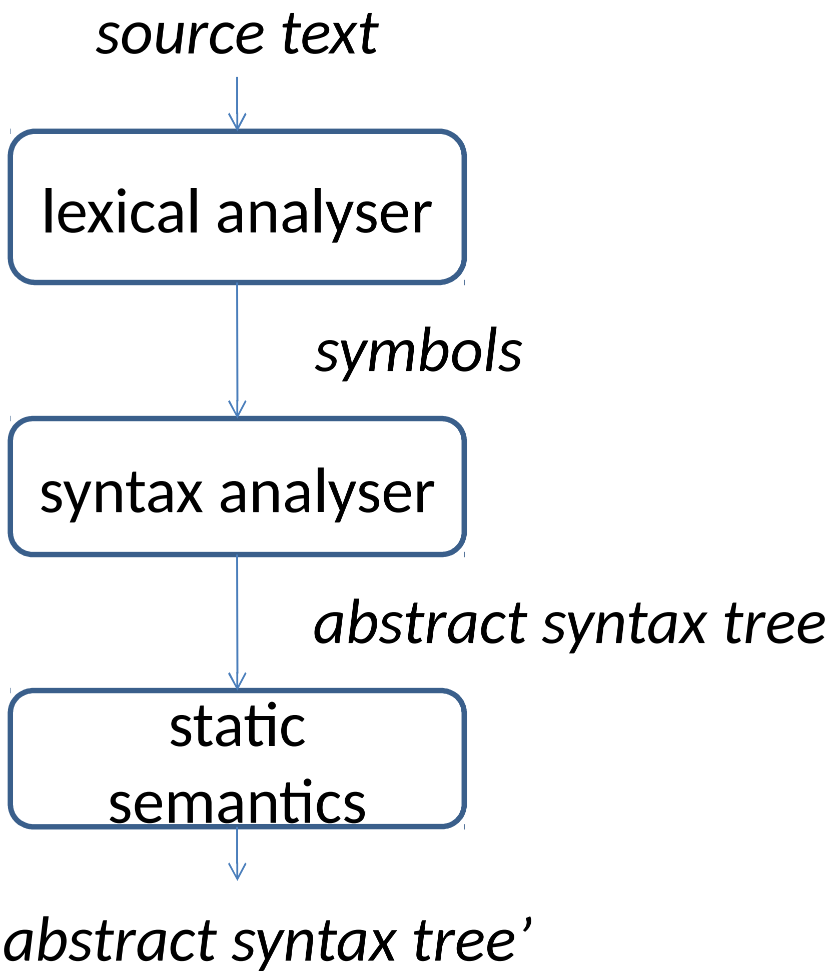
({`b`->4, `a`->7},{me [`a*(b+4)`] {`b`->4, `a`->7}) ㅍ

Semantics

({b->4, a->7},{me [a] {b->4, a->7} *
me [b+4] {b->4, a->7}) ㅍ
({b->4, a->7},{me [a] {b->4, a->7} *
(me [b] {b->4, a->7} +
me [4] {b->4, a->7})} ㅍ
({b->4, a->7},{7*(4+4)}) ㅍ
({b->4, a->7},{56})

Implementation

- front end



source text

lexical analyser

symbols

syntax analyser

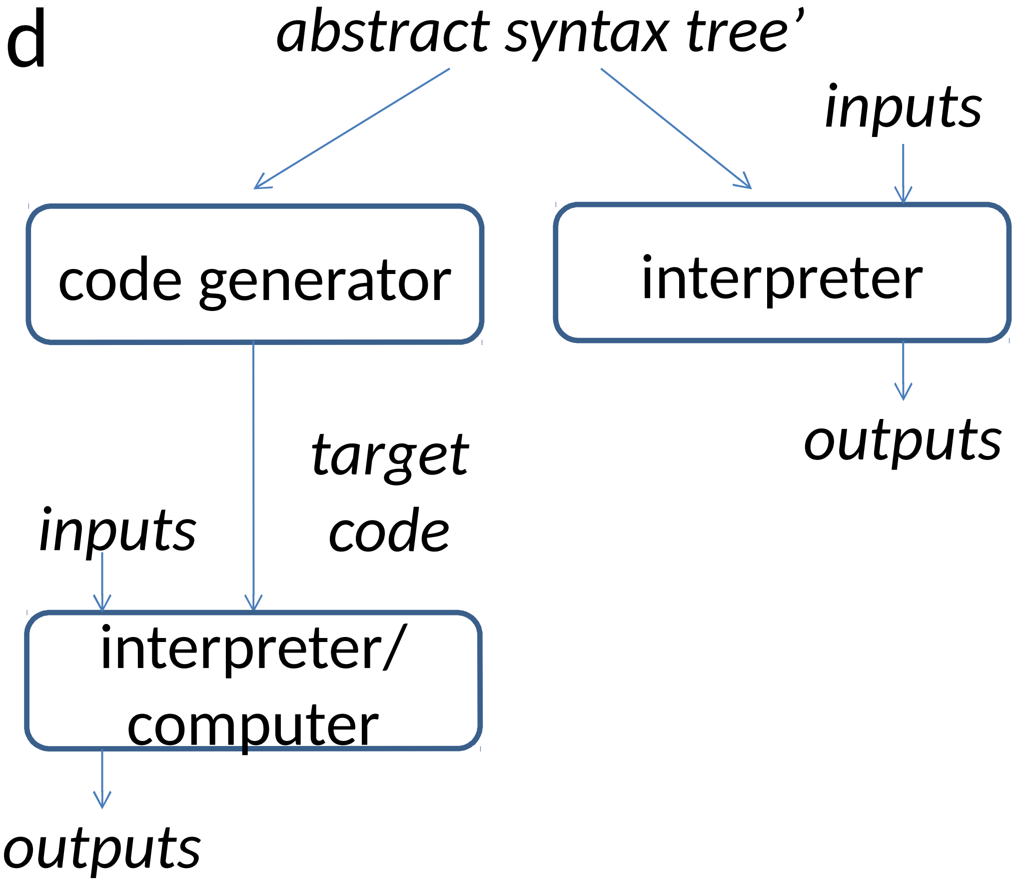
abstract syntax tree

static semantics

abstract syntax tree’

Implementation

- back end



abstract syntax tree'

code generator

interpreter

inputs

outputs

target code

inputs

interpreter/ computer

outputs

Interpreter

- implement in Haskell
- choose AST representation

c -> *c c* | *id* = *e* | *e*

e -> *id* | *int* | *e* + *e* | *e* – *e* | *e* * *e* | *e* / *e*

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

c 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) ㅍ
[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
  - *identifier* == `String`
  - integer == `Int`
- could model state as higher order function
- more flexible to use data structure + look up
- e.g. list

state: *identifier* -> integer ㅍ `[(String,Int)]`

e.g. ({`b`->4, `a`->7} ㅍ `[(“b”,4),(“a”,7)]`

Interpreter

mc: *command* -> state * output -> state * output

mc [c_1 c_2] (state,output) = mc [c_2] (mc [c_1] (state, output))

```
mCs :: [AST] ->
       ([(String,Int)],[Int]) ->
       ([(String,Int)],[Int])
mCs [] (state,output) = (state,output)
mCs (c1:c2) (state,output) =
  mCs c2 (mC c1 (state,output))
```

Interpreter

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*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 (EXP(SADD,e1,e2)) state =
 (mE e1 state) + (mE e2 state)
mE (EXP(SSUB,e1,e2)) state =
 (mE e1 state) - (mE e2 state)
mE (EXP(SMULT,e1,e2)) state =
 (mE e1 state) * (mE e2 state)
mE (EXP(SDIV,e1,e2)) state =
 (mE e1 state) `div` (mE e2 state)
```