Designing intermediate representations

CS448h
Oct. 6, 2015

Programming languages are all about **representations of computation**

The right representations are
what give DSLs their power

DSLs are often best designed
from the IRs out

For example: linear algebra $x' = ABx$

$A : L \times M$
$B : M \times N$
$x : N \times 1$

For example: linear algebra $x' = ABx$

$A : L \times M$
$B : M \times N$
$x : N \times 1$

```
C : LxN
for l in L:
  for m in M:
    for n in N:
      C[l,n] += A[l,m]*B[m,n]
for l in L:
  for n in N:
    x'[l] += C[l,n]*x[n]
```

For example: linear algebra $x' = ABx$

$A : L \times M$
$B : M \times N$
$x : N \times 1$

```
C : LxN
for l in L:
  for m in M:
    for n in N:
      C[l,n] += A[l,m]*B[m,n]
for l in L:
  for n in N:
    x'[l] += C[l,n]*x[n]
```

$x' = mul(mul(A, B), x)$

For example: linear algebra $x' = ABx$

$A : L \times M$
$B : M \times N$
$x : N \times 1$

```
C : LxN
for l in L:
  for m in M:
    for n in N:
      C[l,n] += A[l,m]*B[m,n]
for l in L:
  for n in N:
    x'[l] += C[l,n]*x[n]
```

$x' = mul(\, mul(\, A, B \,), x \,)$

$x' = mul(\, A, mul(\, B, x \,) \,)$

For example: linear algebra $x' = ABx$

$A : L \times M$
$B : M \times N$
$x : N \times 1$

```
C : LxN
for l in L:
  for m in M:
    for n in N:
      C[l,n] += A[l,m]*B[m,n]
for l in L:
  for n in N:
    x'[l] += C[l,n]*x[n]
```

?!?

$x' = mul(mul(A, B), x)$

simple rewrite

$x' = mul(A, mul(B, x))$

What makes a good IR?

simplicity

as few types as possible

generality / expressive power

analyzability / transformability

restriction

Different representations are best for different problems.

across domains

why we make DSLs!

for different compilation problems in a single domain

not 1 IR per compiler/DSL, but many!

What makes a good IR? (take 2)

Easy *target* to generate
from what came before

Easy *source* from which to
generate what comes after

What makes a good IR? (take 2)

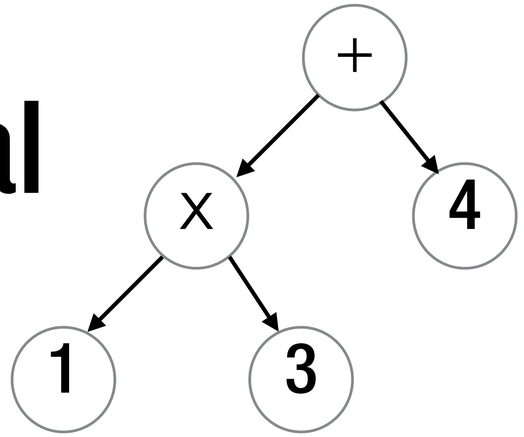
Easy *target* to generate from what came before

at the front-end: easy for a human to write!

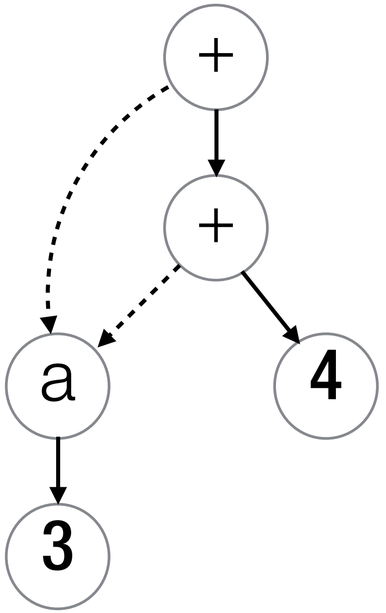
Easy *source* from which to generate what comes after

Common types of representation

trees reflect the hierarchical structure of programs

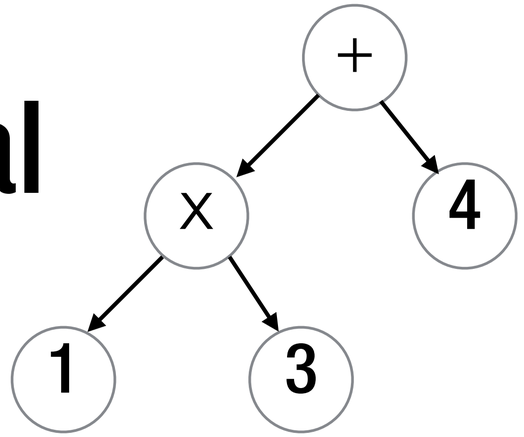


graphs reflect control and data flow

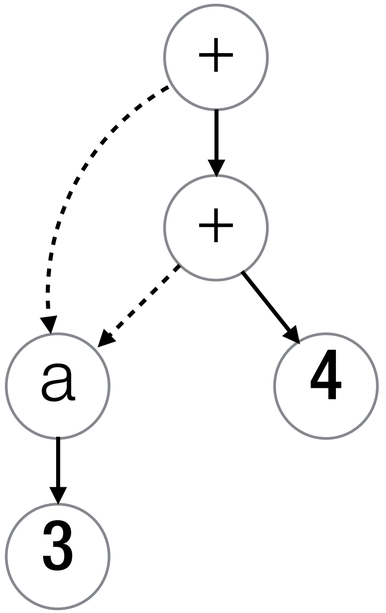


Common types of representation

trees reflect the hierarchical structure of programs



graphs reflect control and data flow



tables map identifiers to nodes, auxiliary metadata

Common types of representation

AST: *user code*

High-level: *user intent*

Low-level: *execution strategy*

Instruction-level: *machine operations*

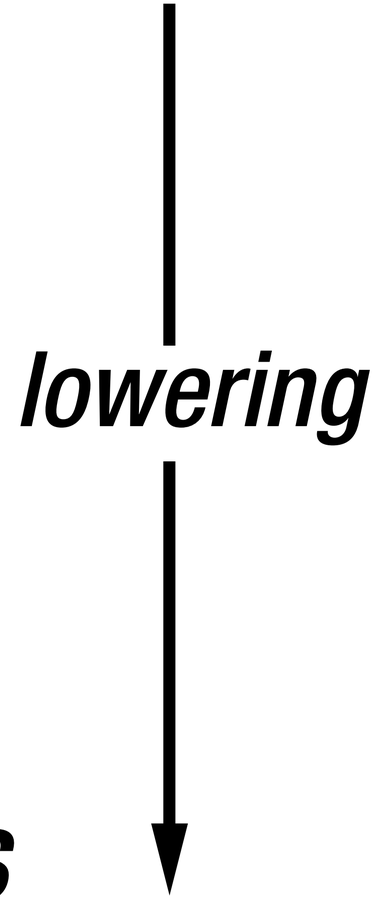
Common types of representation

AST: *user code*

High-level: *user intent*

Low-level: *execution strategy*

Instruction-level: *machine operations*



Algebraic Data Types

a notation for representations

Algebraic Data Types

a notation for representations

```
x = A | B(y) | C(x, y)
y = D(x)
```

Algebraic Data Types

a notation for representations

```
x = A | B(y) | C(x, y)
y = D(x)
```

```
B(D(C(A, D(A))))
```

Algebraic Data Types
a notation for representations

```
x = A | B(y) | C(x, y)
y = D(x)
```

```
B(D(C(A, D(A))))
```

list = Cons (val, list) | Nil

Algebraic Data Types
a notation for representations

```
x = A | B(y) | C(x, y)        B(D(C(A, D(A))))
y = D(x)
```

list = Cons (val, list) | Nil

list = Cons (val, list) | Atom (val)

Representing Regexs & NFAs

Representing Regexs & NFAs

```
re = Char ( char )
   | Seq ( re list )
   | Or ( re list )
   | Star ( re )
   | Maybe ( re )
```

Representing Regexs & NFAs

```
re = Char ( char )
   | Seq ( re list )
   | Or ( re list )
   | Star ( re )
   | Maybe ( re )
```

```
nfa = NFA ( node list, start : node )
```

Representing Regexs & NFAs

```
re = Char ( char )
   | Seq ( re list )
   | Or ( re list )
   | Star ( re )
   | Maybe ( re )
```

```
nfa = NFA ( node list, start : node )
```

```
node = Node ( edge list, accepts : bool, id : int )
```

Representing Regexs & NFAs

```
re = Char ( char )
   | Seq ( re list )
   | Or ( re list )
   | Star ( re )
   | Maybe ( re )
```

```
nfa = NFA ( node list, start : node )

node = Node ( edge list, accepts : bool, id : int )

edge = EpsEdge ( pointsTo : int )
     | CharEdge ( token : char, pointsTo : int )
```

Representing Regexs & NFAs

```
re = Char ( char )
   | Seq ( re list )
   | Or ( re list )
   | Star ( re )
   | Maybe ( re )
```

```
nfa = NFA ( node list, start : node )

node = Node ( edge list, accepts : bool, id : int )

edge = EpsEdge ( pointsTo : int )
     | CharEdge ( token : char, pointsTo : int )

nodemap = map int → node
```

Why is this a good idea?

IRs are naturally recursive data structures with variants

Concise notation to formalize what we're building

Writing down early reveals issues

Common ways to fail

Throw away information

including what's in the code vs. the programmer's head

Be too general

Turing completeness is a curse

when in doubt, restrict rather than generalize!

Expect to get your IRs wrong at first!

Design from your representations out!

Iterate until they feel right