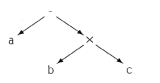
Issues in Code Generation

Code Generator for Expressions

```
expr(node) {
  int result, t1, t2;
  switch(type(node)) {
    case ×, ÷, +, -:
      t1 ← expr(LeftChild(node));
      t2 ← expr(RightChild(node));
      result ← NextRegister();
      emit(op(node), t1, t2, result);
      break;
    case IDENT:
      t1 ← base(node);
      t2 ← offset(node);
      result ← NextRegister();
      emit(loadAO, t1, t2, result);
      break;
    case NUM:
      result ← NextRegister();
      emit(loadI, val(node), none,
           result);
      break;
  }
  return result;
}
```

(a) Treewalk Code Generator



(b) Abstract Syntax Tree for

a - b x c

```
loadI  @a       ⇒ r1
loadAO rarp, r1 ⇒ r2
loadI  @b       ⇒ r3
loadAO rarp, r3 ⇒ r4
loadI  @c       ⇒ r5
loadAO rarp, r5 ⇒ r6
mult   r4, r6   ⇒ r7
sub    r2, r7   ⇒ r8
```

(c) Naive Code

■ **FIGURE 7.5** Simple Treewalk Code Generator for Expressions.

Code Shape

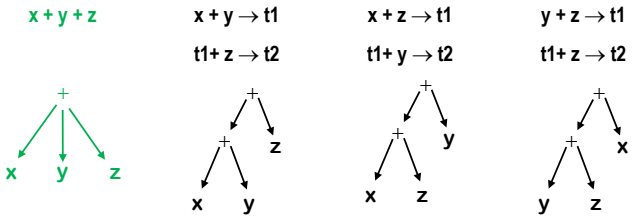
Definition

All those nebulous properties of the code that effect performance

Includes code, approach for different constructs, cost, storage requirements and mapping and choice of operations

Code shape is the end product of many decisions

Code Shape Example



The "best" shape for x+y+z depends on contextual knowledge

- There may be several conflicting options, such as data that may or may not be in registers already, especially if register space is maxed out.
- Data that may have been evaluated already, for example what if y+z was evaluated earlier?

Boolean and Relational Values

Two classic approaches

- Numerical (explicit) representation
- Positional (implicit) representation

Best choice depends on both context and instruction set architecture.

Numerical Encoding

- Explicitly represent the result of Boolean operations.
- Expression: a < b v c < d ^ e < f

```
    comp    ra, rb  ⇒ cc1     // a < b
    cbr_LT  cc1     → L1, L2
L1: loadI   true    ⇒ r1
    jumpI   → L3
L2: loadI   false   ⇒ r1
    jumpI   → L3

L3: comp    rc, rd  ⇒ cc2     // c < d
    cbr_LT  cc2     → L4, L5
L4: loadI   true    ⇒ r2
    jumpI   → L6
L5: loadI   false   ⇒ r2
    jumpI   → L6

L6: comp    re, rf  ⇒ cc3     // e < f
    cbr_LT  cc3     → L7, L8
L7: loadI   true    ⇒ r3
    jumpI           → L9
L8: loadI   false   ⇒ r3
    jumpI           → L9

L9: and     r2, r3  ⇒ r4
    or      r1, r4  ⇒ r5
```

Positional Encoding with Short-Circuit Evaluation

- Position in code represents the result of Boolean operations.
- Expression: a < b v c < d ^ e < f

```
    comp   ra,rb ⇒ cc1    // a < b
    cbr_LT cc1   → L3,L1
L1: comp   rc,rd ⇒ cc2    // c < d
    cbr_LT cc2   → L2,L4
L2: comp   re,rf ⇒ cc3    // e < f
    cbr_LT cc3   → L3,L4
L3: loadI  true  ⇒ r5
    jumpI        → L5
L4: loadI  false ⇒ r5
    jumpI        → L5
L5: nop
```

Issues

Instruction selection

Mapping *IR* into assembly code

Combining operations, using address modes

Instruction scheduling

Reordering operations to hide latencies

Changes demand for registers

Register allocation

Deciding which values will reside in registers

Changes the storage mapping, may add false sharing

Concerns about placement of data and memory operations

These three problems are tightly coupled.

Reducing Demand for Registers

Consider the expression: a – b * c

```
loadI   @a          ⇒ r1
loadAO  rarp, r1    ⇒ r1
loadI   @b          ⇒ r2
loadAO  rarp, r2    ⇒ r2
loadI   @c          ⇒ r3
loadAO  rarp, r3    ⇒ r3
mult    r2, r3      ⇒ r2
sub     r1, r2      ⇒ r2
```

(a) Example After Allocation

```
loadI   @c          ⇒ r1
loadAO  rarp, r1    ⇒ r1
loadI   @b          ⇒ r2
loadAO  rarp, r2    ⇒ r2
mult    r1, r2      ⇒ r1
loadI   @a          ⇒ r2
loadAO  rarp, r2    ⇒ r2
sub     r2, r1      ⇒ r1
```

(c) After Register Allocation