Code Shape

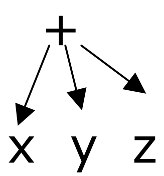
More on Three-address Code Generation

Machine Code Translation

- A single language construct can have many implementations
  - many-to-many mappings from high-level source language to low-level target machine language
  - Different implementations have different efficiency
    - Speed, memory space, register, power consumption

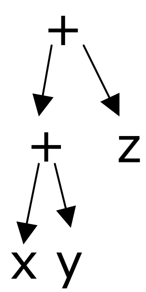
Source code

x + y + z

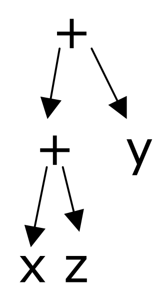


Low-level three-address code

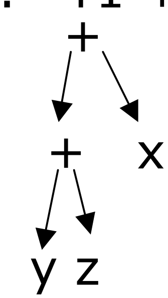
r1 := rx + ry
r2 := r1 + rz



r1 := rx + rz
r2 := r1 + ry



r1 := ry + rz
r2 := r1 + rx



Generating Three-Address Code

- No more support for structured control-flow
  - Function calls=>explicit memory management and goto jumps
- Every three-address instr=>several machine instructions
  - The original evaluation order is maintained
- Memory management
  - Every variable must have a location to store its value
    - Register, stack, heap, static storage
  - Memory allocation convention
    - Scalar/atomic values and addresses => registers, runtime stack
    - Arrays => heap
    - Global/static variables => static storage

```
void fee() {
   int a, *b, c;
   a = 0;     b = &a;   *b = 1;
   c = a + *b;
}
```

From Expressions To 3-Address

- For every non-terminal expression E
  - E.place: temporary variable used to store result
- Synthesized attributes for E
  - Bottom up traversal ensures E.place assigned before used
  - Symbol table has value types and storage for variables
- What about the value types of expressions?

```
E ::= id '=' E1 { E.place=E1.place; gen_var_store(id.entry, E1.place); }
E ::= E1 '+' E2  {E.place=new_tmp();
                  gen_code(ADD,E1.place,E2.place,E.place);}
E ::= (E1)  { E.place = E1.place; }
E ::= id  { E.place=gen_varLoad(id.entry); }
E ::= num { E.place=new_tmp(); gen_code(LOADI, num.val, 0, E.place; }
```

Example input: a = b*c+b+2

Should we reuse register for variable b?

Storing And Accessing Arrays

- Single-dimensional array
  - Accessing ith element: base + (i-low) * w
    Low: lower bound of dimension; w : element size
- Multi-dimensional arrays
  - need to locate base addr of each dimension
    - Row-major, column-major, Indirection vector
- Extend translation scheme to support array access

Row-major

(1,1)	(1,2)	(1,3)	(2, 1)	(2, 2)	(2, 3)

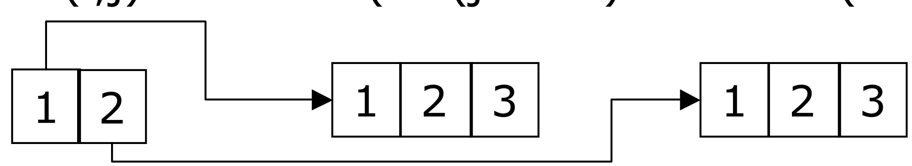
A(i,j)=value at (A+(i-low1)*len2*w+(j-low2)*w)

Column-major

(1,1)	(2,1)	(1,2)	(2, 2)	(1,3)	(2, 3)

A(i,j)=value at (A+(j-low2)*len1*w+(i-low1)*w)

Indirection vector



A(i,j)= value at (A+(i-low1)*wp+(j-low2)*w)

Character Strings

- Languages provide different support for strings
  - C/C++/Java: through library routines
  - PLI/Lisp/ML/Perl/python: through language implementation
  - Important string operations
    - Assignment, concatenation
- Representing strings
  - Null-terminated vs. explicit length field
- Treat strings as arrays of bytes
  - More complex if hardware does not support operating on bytes
  - Translate collective string operations to array operations before three-address translation

a	s	t	r	i	n	g	\0	

Null-termination

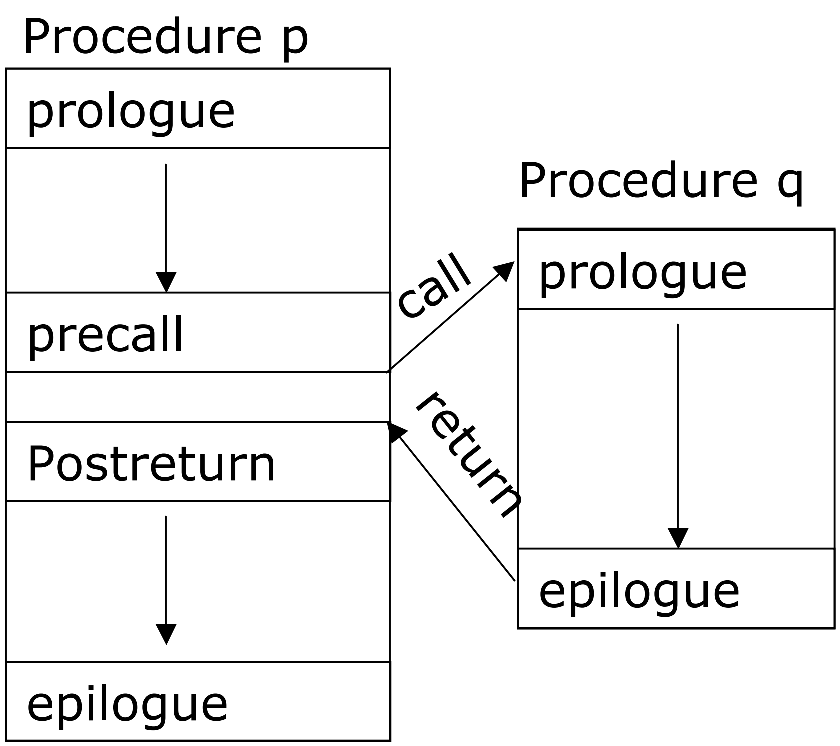
7	a	s	t	r	i	n	g	

Explicit length field

String assignment
a[1] = b[2]

```
loadI @b => r1
cloadAI r1, 2 => r2
loadI @a => r3
cstoreAI r2 => r3, 1
```

Translating Procedural calls



- Function/procedural calls need to be translated into calling sequences
- Side-effect of procedural calls
  - Determined by linkage convention
  - If function call has side effects, Orig. evaluation order need be preserved
- Saving and restoring registers
  - Expensive for large register sets
  - Use special routines or operations to speed it up
  - Combine responsibility of caller and callee
- Optimizing small procedures that don't call others
  - Reduce precall and prologue
  - Reduce number of registers need to be saved

Passing Arrays As Parameters

- Arrays are pointers to data areas
  - Mostly treated as addresses (pointers)
  - Must know dimension & size to support element access
  - Must have type info when passed as parameters
    - Handled either by compilers or programmers
- Compiler support for dynamic arrays
  - Arrays passed as parameters or dynamically allocated
  - Must save type information at runtime to be type safe
- Dope vector: runtime descriptor of arrays
  - Saves starting address, number of dimensions, lower/upper bound and size of each dimension
  - Build a dope vector for each array
  - Can support runtime checking of each element access
    - Before accessing the element, is it a valid access?

Translating Boolean Expressions

- Two approaches
  - Same as translating regular expressions: true → 1/non-zero; false → 0
  - Translate into control-flow branches

    For every boolean expression E

    E.true/E.false: the labels to goto if E is true/false

Numerical translation:

```
c := (a < b)
```

→

```
Cmp_LT ra, rb => rc
```

Position-based translation:

```
c := a < b
```

→

```
     if a < b goto Et
     else goto Ef
Et: c := true
     goto next
Ef: c := false
next:
```

→

```
     cmp ra, rb => cc1
     cbr_LT cc1 =>L1, L2
L1: loadI true => rc
     jumpI  => L3
L2: loadI false=> rc
L3:
```

Short-Circuit Evaluation

- Evaluate only expressions required to determine the final result
  - E: a < b && c < d

    if a >= b, there is no need to evaluate whether c < d
- For every boolean expression E
  - E.true/E.false: the labels to goto if E is true/false

```
E: a < b && c < d
```

→

```
if a < b goto L1
else goto E.false
L1: if c < d goto E.true
else goto E.false
```

→

```
    cmp ra, rb => cc1
    cbr_LT cc1 => L1,Ef
L1: cmp rc, rd => cc2
    cbr_LT cc2 => Et,Ef
Et: …
    jumpI next
Ef: …
Next:
```

Translating control-flow statements

S::= if E THEN S1 →

	E.code
E.true:	S1.code
E.false:	……

S::= if E THEN S1 else S2 →

	E.code
E.true:	S1.code
	goto S.next
E.false:	S2.code
	……

S::= While E DO S1 →

S.begin:	E.code
E.true:	S1.code
	goto S.begin
E.false:	……

Example
Translating control-flow statements

```
if (a < b && c < d)
    x = a;
else
    x = d;
```

```
    cmp ra, rb => cc1
    cbr_LT cc1 => L1,Ef
L1: cmp rc, rd => cc2
    cbr_LT cc2 => Et,Ef
Et: move ra => rx
    jumpI next
Ef: move rx => rd
Next:
```

```
void fee(int x, int y) {
      int I = 0;
      int z = x;
      while (I < 100) {
        I = I + 1;
        if (y < x) z = y;
        A[I] = I;
      }
}
```

More On Control-flow Translation

- If-then-else conditional
  - Use predicated execution vs. conditional branches
- Different forms of loops
  - While, for, until, etc.
  - Optimizations on loop body, branch prediction
- Case statement
  - Evaluate controlling expression
  - Branch to the selected case
    - Linear search : a sequence of if-then-else
    - Binary search or direct jump table
      - Build an ordered table that maps case values to branch labels
  - Execute code of branched case
  - Break to the end of switch statement

Appendix
Translating control-flow statements

- For every statement S, add two additional attributes
  - S.begin: the label of S
  - S.next: the label of statement following S

```
S ::= {if (S.begin != 0) gen_label(S.begin); } E ';'
                                   {S.next=merge(E.true,E.false); }
S ::= WHILE { if (S.begin==0) S.begin=new_label();
                  gen_label(S.begin); }
          '(' E ')' { S1.begin=E.true; } S1
            { S.next=E.false; merge_label(S1.next,S.begin);
             gen_code(jumpI,0,0,S.begin); }
S ::= LBRACE {stmts.begin = S.begin; } stmts RBRACE
          { S.next=stmts.next; }
stmts ::= {S.begin=stmts.begin;} S { stmts.next = S.next; }
stmts ::= {S.begin=stmts.begin; } S
             {stmts1.begin = S.next; } stmts1
             {stmts.next = stmts1.next; }
```

Appendix: Translating Boolean Expressions

- Every boolean expression E has two attributes
  - E.true/false: the label to goto if E is true/false
- Evaluate E.true and E.false as synthesized attribute
  - Create a new label for every unknown jump destination
  - Set destination of created jump labels later
- Usually evaluated by traversing the AST instead of during parsing
  - Issue: creation/merging/insertion of instruction labels

```
E::= true { E.true = new_label(); E.false=0;
                gen_code(jumpI,0,0,E.true); }
E::= false { E.false = new_label(); E.true=0;
                gen_code(jumpI,0,0,E.false); }
E::= E1 relop E2 {E.true= new_label(); E.false=new_label();
                         r=new_tmp(); gen_code(cmp,E1.place,E2.place,r);
                         gen_code(relop.cbr, r, E.true, E.false);}
```

Appendix: Hardware Support For Relational Operations

- Straight conditional code
  - Special condition-code registers interpreted only by conditional branches
- Conditional move
  - Add a special conditional move instruction
- Boolean valued comparisons
  - Store boolean values directly in registers
- Predicated evaluation
  - Conditionally executing instructions

Translating a := x < y

```
Comp rx, ry => cc1
Cbr_LT cc1 -> L1, L2
L1: loadI true => ra
     …
L2: loadI false => ra
      …
```

Straight conditional code

```
Comp rx, ry => cc1
i2i_LT cc1,true,false
                 =>ra
```

Conditional move

```
cmp_LT rx, ry => ra
Cbr ra -> L1, L2
L1: …
L2: …
```

Bool valued comparison

```
Cmp_LT rx, ry => r1
Not  r1 => r2
(r1)? …
(r2)? …
```

Predicated eval.