What we're going to cover

Representations for **I**ntermediate **R**epresentation

- also the implementation

Building IR (in PL/0)

Generating IR from ASTs

Intermediate Representation (**IR**):

- language-independent
  - allows multiple frontends
- machine-independent
  - facilitates retargeting to multiple architectures:

Common representations:

** **three-address code**
- syntax trees & DAGs
- postfix notation

Syntax trees & postfix notation

```
a := b * -c + b * -c
```

Syntax trees & **DAGs**: reflect hierarchical structure of the source program

Postfix notation: linearized representation of AST

```
a b c uminus * b c uminus * + assign
```

Implementation:

- record for each node (operator, operand)
- pointers to connect nodes

Three-address code

Sequence of three-address statements of the form:

```
x := y op z
```

for: `a := b * -c + b * -c`

```
t1 := - c                t1 := - c
t2 := b * t1             t2 := b * t1
t3 := - c                t5 := t2 + t2
t4 := b * t3             a := t5
t5 := t2 + t4
a  := t5
```

Types of 3-address code

Assignment statements:

```
x := y op z
x := op y
x := y
x := y[i]
x[i] := y
x := &y
x := *y            (y = address)
x := *(a + o)      (a = address, o = offset)
*x := y
*(a + o) := y
```


Types of 3-address code

Unconditional jumps: `goto label`

Conditional jumps: `if x relop y goto label`

Param, call, return: `p(x1, x2, ..., xn)`

```
param x1           (push parameter on stack)
...
param xn
call p, n

return y
```

Three-address code

Advantages:

+ simple
+ machine code-like statements
  - operations similar to opcodes
  - operands = 2 sources, 1 destination
  - statements can have labels

  ⇒ easy conversion to target code
+ explicit names for intermediate values

  ⇒ easy to perform optimizations that rearrange or eliminate statements
+ control flow becomes explicit

  ⇒ optimizations

Used in gcc

Implementation of 3-address code

Quadruples

	op	arg1	arg2	result
(0)	uminus	c		t1
(1)	*	b	t1	t2
(2)	uminus	c		t3
(3)	*	b	t3	t4
(4)	+	t2	t4	t5
(5)	:=	t5		a

```
a := b * -c + b * -c
```

beq	x	y	label
param	x		

- temporary names are in the symbol table
- all operands are pointers to symbol table

Implementation of 3-address code, cont'd.

Triples

	op	arg1	arg2
(0)	uminus	c	
(1)	*	b	(0)
(2)	uminus	c	
(3)	*	b	(2)
(4)	+	(1)	(3)
(5)	:=	a	(4)

- pointer to a triple instead of a temporary name
- only programmer-defined names are in symbol table

Implementation of 3-address code, cont'd.

Indirect triples

	statement
(0)	(14)
(1)	(15)
(2)	(16)
(3)	(17)
(4)	(18)
(5)	(19)

	op	arg1	arg2
(14)	uminus	c	
(15)	*	b	(14)
(16)	uminus	c	
(17)	*	b	(16)
(18)	+	(15)	(17)
(19)	:=	a	(18)

Comparison

Space

\+ triples

• indirect triples

\- quadruples

Optimizations

\+ quadruples: computation of a value & its use are separate

\+ indirect triples: change statement list

\- triples: optimizations that move a temporary value definition require changing all its uses

Allocation of **storage** for temporaries

\+ quadruples: can access temporaries immediately via symbol table

\- indirect triples & triples: calculation deferred to code generation

Generating IR

How:

- tree walk of the AST, bottom up, left to right
- assign to a new temporary for each result

Illustrate using pseudo-PI/0 code

Generating IR for variable references

Two cases:

- if want l-value: get an address
- if want r-value: get the value @ address

To compute l-value:

```
Name VarRef::codegen_addr(s, int& offset) {
  ste = s->lookup(_ident, foundScope);
  if (ste == NULL) ... // fatal error
  if (!ste->isVariable()) ... // fatal error

  Name base = s->getFPOf(foundScope);
  offset = ste->offset();
  // base + offset = address of variable

  return base;
}
```


IR for variable references, cont'd.

To compute r-value:

```
Name LValue::codegen(s) {
  int offset;
  Name base = codegen_addr(s, offset);
  Name dest = new Name;
  emit(dest := *(base+offset));
  return dest;
}
```

Shared by all r-value syntax nodes (vars and arrays)

VarRef::codegen handles constants

IR for literals

```
Name IntegerLiteral::codegen(s) {
  result = new Name;
  emit(result := _value);
  return result;
}
```


IR for expressions

```
Name BinOp::codegen(s) {
  Name e1 = _left->codegen(s);
  Name e2 = _right->codegen(s);
  result = new Name;
  emit(result := e1 _op e2);
  return result;
}
```

Also unary operations

IR for assignments

```
AssignStmt::codegen(s) {
  // compute address of l.h.s.:
  int offset;
  Name base = _lvalue->codegen_addr
            (s, offset);

  // compute value of r.h.s.:
  Name result = _expr->codegen(s);

  // do assignment:
  emit(*(base + offset) := result);
}
```


IR for array accesses

Source code:

array_expr[*index_expr*]

Generated IR code:

```
// address of location = a + offset
a := <addr of array_expr>
i := <value of index_expr>
offset := i * <size of element type>
result := a + offset
```

Implementation of array access

```
Name ArrayRef::codegen_addr(s, int& offset){
  // compute address of array:
  Name base =
    _array->codegen_addr(s, offset);

  // compute value of index:
  Name i = _index->codegen(s);

  // scale index by elem size to get array offset
  int esize =
    _array_type->elem_type()->size();
  Name arrayOffset = new Name;
  emit(arrayOffset := i * esize);

  // compute final base address:
  Name result = new Name;
  emit(result := base + arrayOffset);

  return result; // + offset!
}
```


Calling functions

Push arguments, static link, call function

Return a value

```
Name FuncCall::codegen(s) {
  forall arguments, from left to right {

    if (arg is byValue) {
      // pass value of argument:
      arg = arg->codegen(s);
      emit(push arg);
    }

      else {

      // pass address of argument (NEW):
      int offset;
      base = arg->codegen_addr(s, offset);
      arg = new Name;
      emit(arg := base + offset);
      emit(push arg);
    }
  }

  ...
```

```
  ...

  // compute & push static link:
  s->lookup(_ident, foundScope);
  Name staticLink = s->getFPOf(foundScope);
  emit(push staticLink);

  ...

  // generate call:
  emit(call _ident);

  ...

  staticLink// handle result (NEW):
  Name result = new Name;
  emit(result := RET0);
  return result;
}
```


Accessing call-by-reference parameters

Formal parameter is **address** of actual, not value
⇒ need extra load

```
Name VarRef::codegen_address(s, int& offset){
  ste = s->lookup(_ident, foundScope);
  // check for errors; defensive programming
  ...

  Name base = s->getFPOf(foundScope);
  offset = ste->offset();

  if (ste->isFormalByRef()) {
    Name result = new Name;
    emit(result := *(base + offset);
    offset = 0;
    base := result;
  }

  return base;
}
```

Control structures

Rewrite control structures using:
- explicit labels and
- conditional & unconditional branch IR instructions

E.g. **if** statement:

```
if test then stmts1 else stmts2 end;
    ⇒
  t1 := test
  if t1 = 0 goto _else  // conditional branch
  stmts1
  goto _done
_else:
  stmts2
_done:
```


Code for if codegen

```
void IfStmt::codegen(s) {
  // generate test expr into temp:
  Name t = _test->codegen(s);

  // generate conditional branch:
  Label else_lab = new Label;
  emit(if t = 0 goto else_lab);

  // generate then part:
  _then_stmts->codegen(s);

  // generate branch over else part:
  Label done_lab = new Label;
  emit(goto done_lab);

  // generate else part, with leading label:
  emit(else_lab:);
  _else_stmts->codegen(s);

  // finish up:
  emit(done_lab:);
}
```

`while` statement

```
while test do stmts end;
   ⇒
_loop:
  t1 := test
  if t1 = 0 goto _done
  stmts
  goto _loop
_done:
```


IR codegen for break stmt

```
...
while ... do
  ...
  if ... then
    ...
    break;
  end;
  ...
end;
...
```

Short-circuiting boolean expressions

How to support short-circuit evaluation of `and` and `or`

Example:

```
if x <> 0 and y / x > 5 then
  b := y < x;
end;
```

Treat as control structure, not as operator:

```
  expr1 and expr2
    ⇒
  result := expr1
  if result = 0 goto _done
  result := expr2
_done:
```


Case statements

```
switch expr
  begin
    case value1:stmt
    case value2:stmt
    ...
    case valuen:stmt
    default: stmt
end
```

Implementation

- evaluate the expression
- find the matching value
  - conditional goto's
  - jump table
- execute the associated statement

Case statements

value	label
$value_1$	L1
$value_2$	L2
...	...
$value_n$	Ln
default	Ln+1

Implementation considerations:

- small number of values ⇒ **conditional goto**'s
- > 10 values ⇒ **jump table**
  - values not consecutive:
    value part of table & seach on value
  - values consecutive & $value_1$ <= value <= $value_n$:
    index via value-$value_1$
- >> 10 values ⇒ **hash table**