Intermediate Code Generation

Purpose:
translate ASTs into linear sequence of simple statements called **intermediate code**

- can optimize intermediate code "in place"
- a later pass translates intermediate code into **target code**

Intermediate code is machine-independent

- don't worry about details of target machine (e.g. # of registers, kinds of instruction formats)
- intermediate code generator & optimizer are portable across target machines

Intermediate code is simple and explicit

- decomposing whole code generation problem into simpler pieces
- constructs implicit in AST become explicit in intermediate code

PL/0 compiler merges intermediate and target code gen. for simplicity of coding

Three-address code

A simple intermediate "language"

Each statement has at most one operation in right-hand side

- introduce extra temporary variables for all "internal" values

```
Not OK: x  := y * z + q / r
OK:     t1 := y * z
        t2 := q / r
        x  := t1 + t2
```

Control structures broken down

```
Not OK: for i := 0 to 10 do ... end
OK:     i := 0
      loop:
        if i < 10 goto done;
        ...
        goto loop;
      done:
```

Pointers and address calculations made explicit

```
Not OK: x  := a[i]
OK:     t1 := i * 4
        x  := *(a+t1)
```


Available operations

var := *constant*
var := *var*
var := *unop* *var*
var := *var* *binop* *var*
var := *proc*(*var*, ...)
proc(*var*, ...)

var := &*var*
var := *(*var* + *constant*)
*(*var* + *constant*) := *var*

if *var* goto *label*
goto *label*
label: ...

return *var*
return

Generating IR from AST's

Cases:

- expressions
- assignment statements
- control statements
- declarations ⇒ already done

Generating IR for expressions

How:

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

Pseudo-code:

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

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


IR for variable references

Two cases:

- if want l-value: load address
- if want r-value: load value @ address

To compute l-value:

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

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

   return base;
}
```


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 lvalue syntax nodes (vars and arrays)

`VarRef::codegen` handles constants

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);
}
```

Accessing call-by-reference parameters

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

```
Reg VarRef::codegen_address(s, int& offset){
  ste = s->lookup(_ident, foundScope);
  ... // error checking
  Name base = s->getFPOf(foundScope);
  offset = ste->offset();

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

  return base;
}
```


Calling functions

New: by-ref arguments, return values

```
Name FunCall::codegen(s) {
  forall arguments, from left to right {
    if (arg is byValue) {
      // pass value of argument:
      name = arg->codegen(s);
      emit(push name);
    } else {
      // pass address of argument (NEW):
      int offset;
      base = arg->codegen_addr(s, offset);
      name = new Name;
      emit(name := base + offset);
      emit(push name);
    }
  }

  ...
```



```
  ...

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

  // generate call:
  emit(call _ident);

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


IR for array accesses

Source code:

array_expr[*index_expr*]

Generated IR code:

```
a_addr := <addr of array_expr>
i := <value of index_expr>
offset := i * <size of element type>
elem_addr := a_addr + 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 offset:
  int esize =
    _array_type->elem_type()->size();
  Name o = new Name;
  emit(o := i * esize);

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

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


Example

```
module main;
  var z:int;
  var a: array[5] of array[10] of int;
  procedure p(var q:int);
    var b: int;
  begin
    b := 1 + 2;
    b := b + z;
    q := q + 1;
    a[3][8] := b;
  end p;
begin
  z := 2;
  p(z);
end main.
```


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:
```


Short-circuiting

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:
```


IR codegen for break stmt

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