Intermediate code generation

- Purpose: translate ASTs into linear sequence of simple statements called intermediate code
- Can optimize intermediate code in place
- Intermediate code is machine-independent
- Don’t worry about details of the target machine (e.g., number of registers, kinds of instruction formats)
- Intermediate code generator and optimizer are portable across target machines
- Intermediate code is simple and explicit
- Decomposes code generation problem into simpler pieces
- Constructs implicit in the AST become explicit in the intermediate code

PL/0

- Our PL/0 compiler merges intermediate and target code generation for simplicity of coding

Three-address code: a simple intermediate language

- Each statement has at most one operation in its right-hand side
- Introduce extra temporary variables if needed
- Control structures are broken down into (conditional) branch statements
- Pointer and address calculations are made explicit

Examples

- \[ x := y \times z + q / r \]
- \[ t1 := y \times z \]
- \[ t2 := q / r \]
- \[ x := t1 + t2 \]
- \[ i := 0 \] to 10 do…
- \[ i := 0 \]
- \[ \text{loop: if } i > 10 \text{ goto done; } \]
- \[ i := i + 1 \]
- \[ \text{goto loop; } \]
- \[ \text{done: } ]
- \[ x := a[i] \]
- \[ t1 := i \times 4 \]
- \[ x := * (a + t1) \]
Available operations

- \( \text{var} := \text{constant} \)
- \( \text{var} := \text{var} \)
- \( \text{var} := \text{unop var} \)
- \( \text{var} := \text{binop var} \)
- \( \text{var} := \text{proc} ( \text{var}, \ldots ) \)
- \( \text{var} := \text{unaryvar} \)
- \( \text{var} := \text{var} \text{ binop } \text{var} \)
- \( \text{var} := \text{post label} \)
- \( \text{var} := \text{return var} \)

ICG (Intermediate code generation) from ASTs

- Generally one operation per statement, not arbitrary expressions, etc.
- \( \text{Once again (like type checking), we'll do a tree traversal} \)
- \( \text{Cases} \) (likewise type checking, we'll do a tree traversal)
  - \( \text{expressions} \)
  - \( \text{assignment statements} \)
  - \( \text{control statements} \)
  - \( \text{declarations are already done} \)

ICG for expressions

- \( \text{How: tree walk, bottom-up, left-right, (largely postorder) assigning a new temporary for each result} \)
- \( \text{Pseudo-code} \)
  
  ```
  \text{Name IntegerLiteral::codegen(STS* s)}
  \{ 
    \text{int offset;}
    \text{Name base = codegen_address(s, offset);}
    \text{Name dest = new Name;}
    \text{emit(dest := base + offset);}
    \text{return dest;}
  \}
  ```

Another pseudo-example

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

ICG for variable references

- \( \text{Two cases} \)
  - \( \text{if we want l-value, compute address} \)
  - \( \text{if we want r-value, load value at that address} \)

r-value

\( \text{Name LValue::codegen(SymTabScope* s)} \) \{
  \text{int offset;}
  \text{Name base = codegen_address(s, offset);}
  \text{Name dest = new Name;}
  \text{emit(dest := base + offset);}
  \text{return dest;}
\}

\( \text{Name VarRef::codegen(SymTabScope* s)} \) \{
  \text{SymTabEntry* ste = s->lookup(_ident, foundScope);}
  \text{if (ste->isConstant())} \{
    \text{Name dest = new Name;}
    \text{emit(dest := ste->value());}
    \text{return dest;}
  \}
  \text{return LValue::codegen(s);}
\}
**ICG for assignments**

```c
AssignStmt::codegen(SymTabScope* s) {
    int offset;
    Name base = _value->codegen_addr(s, offset);
    Emit((base + offset) := result);
}
```

**ICG for function calls**

```c
FunCall::codegen(SymTabScope* s) {
    forall arguments, from right to left {
        if (arg is byValue) {
            Name name = arg->codegen(s);
            Emit(push name);
        } else {
            int offset;
            Name base = arg->codegen_addr(s, offset);
            Name ptr = new Name;
            Emit(ptr := base + offset);
            Emit(push ptr);
        }
    }
}
```

---

**Accessing call-by-ref params**

```c
Formal parameter is address of actual, not the value, so we need an extra load statement
```
ICG for array accesses

- AST:
  array_expr[index_expr]

- Code generated:
  (array_b, array_o) := <base, offset of array_expr>
  i := <value of index_expr>
  delta := i * <size of element type>
  (elm_b, elm_o) := (array_b + delta, array_o)

- 2D Arrays? Not really:

  MyArray[i][j];

ICG for if statement

  void IfStmt::codegen(SymTabScope* s) {
    Name t = _test->codegen(s);
    Label else_lab = new Label;
    emit(if t == 0 goto else_lab);
    _then_stmts->codegen(s);
    Label done_lab = new Label;
    emit(goto done_lab);
    _else_stmts->codegen(s);
    emit(done_lab);
  }

ICG for while statement

ICG for continue statement

Short-circuiting of and & or

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

  - Treat as control structure, not operator:
    t0 := 0
    t1 := e1
    iffalse t1 goto 1
    t0 := e2
    l: //value in t0

Example

module main;
  var z:int;
  procedure p(var q:int);
  var a array[5] of int;
  var b:int;
  begin
    b := 1 + 2;
    b := b + 3;
    q := q + 1;
    b := a[3][4];
    if b > 0 then b:=0 end
    p(b);
  end p;
end main.

t1 := 5
*{fp+2+offset} := t1
Example

```plaintext
module main;
  var z: int;
  procedure p(var q: int);
  var a[5] of int;
  var b: int;
begin
  b := 5 + 2;
  b := b + 1;
  q := q + 1;
  b := a[4][8];
  if b = 0 then
    p := q;
  begin
    z := 5;
  end;
end main.
```

Example

```plaintext
module main;
  var z: int;
  procedure p(var q: int);
  var a[5] of int;
  var b: int;
begin
  b := 5 + 2;
  b := b + 1;
  q := q + 1;
  b := a[4][8];
  if b = 0 then
    p := q;
  begin
    z := 5;
  end;
end main.
```

Example

```plaintext
module main;
  var z: int;
  procedure p(var q: int);
  var a[5] of int;
  var b: int;
begin
  b := 5 + 2;
  b := b + 1;
  q := q + 1;
  b := a[4][8];
  if b = 0 then
    p := q;
  begin
    z := 5;
  end;
end main.
```

Example

```plaintext
module main;
  var z: int;
  procedure p(var q: int);
  var a[5] of int;
  var b: int;
begin
  b := 5 + 2;
  b := b + 1;
  q := q + 1;
  b := a[4][8];
  if b = 0 then
    p := q;
  begin
    z := 5;
  end;
end main.
```
### Prototype compiler structure

- **Source Program**
  - Stream of characters
  - Lexical analysis
  - Sequence of tokens
  - Syntactic analysis
  - Abstract syntax tree (AST)
  - Semantic analysis
  - AST and symbol table

- **Target Program**
  - Executable code
  - Intermediate code
  - Optimization
  - Intermediate representation
  - Intermediate code generation
  - AST and symbol table

### Target Code Generation
- **Input**: intermediate representations (IR)
  - Ex: three-address code
- **Output**: target language program
  - Absolute binary code
  - Relocatable binary code
  - Assembly code
  - C

### Task of code generator
- Bridge the gap between intermediate code and target code
  - Intermediate code: machine independent
  - Target code: machine dependent
- Two jobs
  - Instruction selection: for each IR instruction (or sequence), select target language instruction (or sequence)
  - Register allocation: for each IR variable, select target language register/stack location

### Difficulty depends on instruction set
- **RISC**: easy
  - Usually only one way to do something
  - Closely resembles IR instructions
- **CISC**: hard
  - Lots of alternative instructions with similar semantics
  - Lots of tradeoffs among speed, size
  - Simple RISC-like translation may be inefficient
- **C**: easy, as long as C is appropriate for desired semantics
  - Can leave optimizations to the C compiler

### Instruction selection
- Given one or more IR instructions, pick the “best” sequence of target machine instructions with the same semantics
  - “best” = fastest, shortest
- Correctness is a big issue, especially if the code generator (codegen) is complex

### Example
- **IR code**
  - \( t3 := t1 + t2 \)
- **Target code for MIPS**
  - add \$3,\$1,\$2
- **Target code for SPARC**
  - add \%1,\%2,\%3
- **Target code for 68k**
  - mov.l \( d1,d3 \)
  - add.l \( d2,d3 \)
  - Note that a single IR instruction may expand to several target instructions
Example

IR code
- \( t1 := t1 + 1 \)
- Target code for MIPS
  - add $1,$1,$1
- Target code for SPARC
  - add %1,%1,%1
- Target code for 68k
  - add.l #1,dl or
  - inc.l dl

Can have choices
This is a pain, since choices imply you must make decisions

Example

IR code (push x onto stack)
- \( sp := sp - 4 \)
  - \( *sp := t1 \)
- Target code for MIPS
  - sub $sp,$sp,4
  - sw $1,0($sp)
- Target code for SPARC
  - sub %sp,4,%sp
  - st %1,[%sp+0]
- Target code for 68k
  - mov.l dl,-(sp)

Note that several IR instructions may combine to a single target instruction
This is hard!

Instruction selection in PL/0

Very simple instruction selection
- As part of generating code for an AST node
- Merged with intermediate code generation, because it’s so simple
- Interface to target machine: assembler class
  - Function for each kind of target instruction
  - Hides details of assembly format, etc.
  - Two assembler classes (MIPS and x86), but you only need to extend MIPS

Resource constraints

Intermediate language uses unlimited temporary variables
- This makes intermediate code generation easy
- Target machine, however, has fixed resources for representing “locals”
  - MIPS, SPARC: 31 registers minus SP, FP, RetAddr, Arg1-4, ...
  - 68k: 16 registers, divided into data and address registers
  - x86: 4(?) general-purpose registers, plus several special-purpose registers

Register allocation

Using registers is
- Necessary: in load/store RISC machines
- Desirable: since much faster than memory
- So...
  - Should try to keep values in registers if possible
  - Must reuse registers for many temp variables, so we must free registers when no longer needed
  - Must be able to handle out-of-registers condition, so we must spill some variables to stack locations
  - Interacts with instructions selection, which is a pain, especially on CISCs

Classes of registers

What registers can the allocator use?
- Fixed/dedicated registers
  - SP, FP, return address, ...
  - Claimed by machine architecture, calling convention, or internal convention for special purpose
  - Not easily available for storing locals
- Scratch registers
  - A couple of registers are kept around for temp values
    - E.g., loading a spilled value from memory to operate upon it
  - Allocatable registers
  - Remaining registers are free for the allocator to allocate
    - PL/0 on MIPS: $8-$25
Which variables go in registers?

- Temporary variables: easy to allocate
  - Defined and used exactly once, during expression evaluation
  - The allocator can free the register after use easily
  - Usually not too many in use at one time
  - So less likely to run out of registers
- Local variables: hard, but doable
  - Need to determine last use of variable to free register
  - Can easily run out of registers, so need to make decisions
  - What about load/store to a local through a pointer?
  - What about the debugger?
- Global variables, procedure proarams, across calls, ...:
  - Really hard. A research project?

Keep set of allocated registers as codegen proceeds

- RegisterBank class
- During codegen, allocate one from the set
  - Reg reg = rb->getNew();
  - Side-effects register bank to note that reg is taken
  - What if no registers are available?
- When done with a register, release it
  - rb->free(reg);
  - Side-effects register bank to note that reg is free

Connection to ICG

- In the last lecture, the pseudo-code often create a new Name
- Since PL/0 merges intermediate code generation (ICG) with target generation, these new NameS are equivalent to allocating registers in PL/0

Example

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

```c
vs
```

```c
Reg IntegerLiteral::codegen(SymTabScope* s, RegisterBank* rb) {
  Reg r = rb->newReg();
  TheAssembler->moveImmediate(r, _value);
  return r;
}
```

Exanple 1

```c
AssignStmt::codegen(SymTabScope* s) a) {
  int offset;
  Name base = _value->codegen_addr(s, offset);
  Name result = _expr->codegen(s);
  emit('*(base + offset) := result);
}
```

```c
vs
```

```c
void AssignStmt::codegen(SymTabScope* s, RegisterBank* rb) {
  int offset;
  Reg base = _value->codegen_addr(s, rb, offset);
  Reg result = _expr->codegen(s, rb);
  TheAssembler->store(result, base, offset);
  rb->freeReg(base);
  rb->freeReg(result);
}
```

Codegen for if statements

```c
void IfStmt::codegen(SymTabScope* s, RegBank* rb) {
  Reg test = _test->codegen(s, rb);
  char* elseLabel = TheAssembler->newLabel();
  TheAssembler->branchFalse(test, elseLabel);
  rb->freeReg(test);
  for (int i=0; i < _then_stmts->length(); i++) {
    _then_stmts->fetch[i]->codegen(s, rb);
  }
  TheAssembler->insertLabel(elseLabel);
}
```
Codegen for call statements

void CallStmt::codegen(SymTabScope* s, RegBank* rb) {
    for (int i = _args->length() - 1; i >= 0; i--) {
        Reg areg = _args->fetch(i)->codegen(s, rb);
        TheAssembler->push(areg); rb->freeReg(areg);
    }
    SymTabScope* enclScope = s->lookup(_ident, enclScope);
    SymTabEntry* ste = s->lookup(_ident, enclScope);
    TheAssembler->push(staticLink);
    rb->freeReg(staticLink);
    TheAssembler->call(_ident);
    rb->restoreReg(s);
    TheAssembler->popMultiple(_args->length() + 1) *
        TheAssembler->wordSize());
}

Another example

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

Example

x := x + 2 * (x - 1)

Free after use: 5 regs

Example, cont'

x := x + 2 * (x - 1)

Free before use: 4 regs

Example

module main;
var z: int;
begin
  b := 1 + 2;
  b := b + 1;
  q := q + 4;
if b > 1 then b := 0
end;
begin
  b := 5;
  print;
end main.

Example

module main;
var z: int;
var b: int;
begin
  b := 1 + 2;
  b := b + 1;
  q := q + 1;
if b > 1 then b := 0
end;
begin
  b := 5;
  print;
end main.
Example module main;
var z:int;
procedure p(var q:int);
var b:int;
beg
b := b + 1;
q := q + 1;
b := a[4][8];
if b > 1 then b := 0
end
p;
beg
z := 5;
p(z);
end main.

Example module main;
var z:int;
procedure p(var q:int);
var b:int;
beg
b := b + 1;
q := q + 1;
b := a[4][8];
if b > 1 then b := 0
end
p;
beg
z := 5;
p(z);
end main.