CSE401: Code Generation

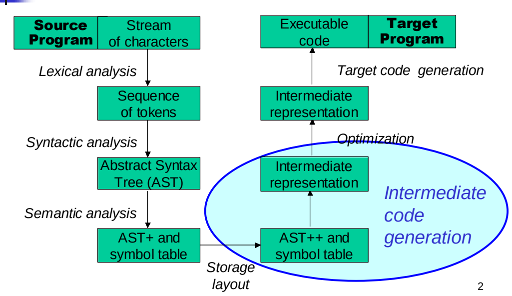
Andrei Alexandrescu

Autumn 2002

Slides by Chambers, Eggers, Notkin, Ruzzo, and others

1

Prototype compiler structure



2

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 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

3

PL/0

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

4

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

5

Examples

A.
```
x := y * z + q / r
```
A.
```
t1 := y * z
t2 := q / r
x := t1 + t2
```

B.
```
for i := 0 to 10 do …
end
```
B.
```
i := 0
loop:
  if i < 10 goto done;
  …
  i := i + 1
  goto loop;
done:
```

C.
```
x := a[i]
```
C.
```
t1 := i * 4
x  := *(a + t1)
```

6

Available operations

- `var := constant`
- `var := var`
- `var := unop var`
- `var := var binop var`
- `var := proc(var, …)`
- `var := &var`
- `var := *(var + constant)`
- `*(var + constant) := var`
- `if var goto label`
- `goto label`
- `label:`
- `return var`
- `return`

generally *one* operation per statement, not arbitrary expressions, etc.

7

ICG (Intermediate code generation) from ASTs

- Once again (like type checking), we'll do a tree traversal
- Cases
  - expressions
  - assignment statements
  - control statements
  - declarations are already done

8

ICG for expressions

- How: tree walk, bottom-up, left-right, (largely postorder) assigning a new temporary for each result
- Pseudo-code

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

Temps: just suppose we had infinitely many registers

9

Another pseudo-example

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

10

ICG for variable references

- Two cases
  - if we want l-value, compute address
  - if we want r-value, load value at that address

11

r-value

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

Name VarRef::codegen(SymTabScope* s) {
  STE* ste = s->lookup(_ident,foundScope);
  if (ste->isConstant()) {
    Name dest = new Name;
    emit(dest := ste->value());
    return dest;
  }
  return Lvalue::codegen(s);
}
```

12

l-value

```
Name VarRef::codegen_address(STS* s, int& offset)
{
  STE* 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;
}
```

returning two things

13

Compute address of frame containing variable

```
Name SymTabScope::getFPOf(foundScope) {
  Name curFrame = FP;
  SymTabScope* curscope = this;
  while (curScope != foundScope) {
    Name newFrame = new Name; // load static link
    int offset = curScope->staticLinkOffset();
    emit(newFrame := *(curFrame + offset));
    curScope = curScope->parent();
    curFrame = newFrame;
  }
  return curFrame;
}
```

14

ICG for assignments

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

15

ICG for function calls

```
Name 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);
    }
  }
```

…continued

16

ICG for function calls, con't

```
  s->lookup(_ident,foundScope);
  Name link = s->getFPOf(foundScope);
  emit(push link);      // callee's static link

  emit(call _ident)

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

17

Accessing call-by-ref params

- Formal parameter is address of actual, not the value, so we need an extra load statement
- ```
  Name VarRef::codegen_address(STS* s, int& offset){
    ste = s->lookup(_ident,foundScope);
    Name base = s->getFPOf(foundScope);
    offset = ste->offset();
    if (ste->isFormalByRef()) {
      Name ptr = new Name;
      emit(ptr := *(base + offset));
      offset = 0;
      return ptr;
    }
    return base;
  }
  ```

18
  ```

ICG for array accesses

- AST:

```
array_expr[index_expr]
```

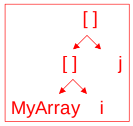
- Code generated:

```
(array_b,array_o):=<base,offset of array_expr>
i := <value of index_expr>
delta := i * <size of element type>
(elmt_b, elmt_o) := (array_b + delta, array_o)
```

- 2D Arrays? Not really:

```
var MyArray array[10] of
            array[5] of int;
MyArray [i] [j];
```

array_expr



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);
  emit(else_lab:);
  _else_stmts->codegen(s);
  emit(done_lab:);
}
```

20

ICG for while statement

21

ICG for continue statement

22

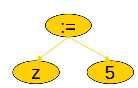
Short-circuiting of and & or

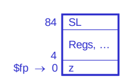
- Example
  - `if x <> 0 and y / x > 5 then`
`b := y < x;`
`end;`
- Treat as control structure, not operator:
  - e1 and e2 →

```
  t0 := 0
  t1 := e1
  iffalse t1 goto 1
  t0 := e2
1: //value in t0
```

23

Example





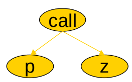

```
module main;
  var z:int;
  procedure p(var q:int);
    var a:array[5] of
          array[10] of int;
    var b:int;
  begin
    b := 1 + 2;
    b := b + z;
    q := q + 1;
    b := a[4][8];
    if b>1 then b:=0 end
  end p;
begin
  z := 5;
  p(z);
end main.
```

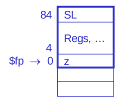
t1 := 5

$*(fp+z_{offset})$:= t1

24

Example



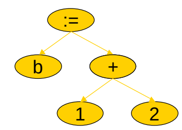


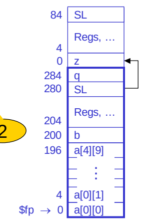

```
module main;
  var z:int;
  procedure p(var q:int);
    var a:array[5] of
          array[10] of int;
    var b:int;
  begin
    b := 1 + 2;
    b := b + z;
    q := q + 1;
    b := a[4][8];
    if b>1 then b:=0 end
  end p;
begin
  z := 5;
  p(z);
end main.
```

```
t1 := z_offset
t2 := fp+t1
*(sp) := t2
sp := sp-4
*(sp) := fp
sp := sp-4
call p
```

25

Example



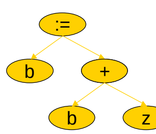


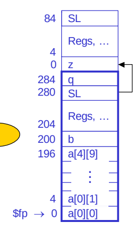

```
module main;
  var z:int;
  procedure p(var q:int);
    var a:array[5] of
          array[10] of int;
    var b:int;
  begin
    b := 1 + 2;
    b := b + z;
    q := q + 1;
    b := a[4][8];
    if b>1 then b:=0 end
  end p;
begin
  z := 5;
  p(z);
end main.
```

```
t1 := 1
t2 := 2
t3 := t1 + t2
*(fp+b_offset) := t3
```

26

Example



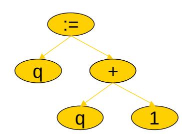


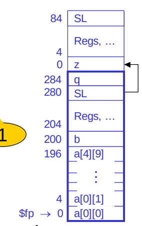

```
module main;
  var z:int;
  procedure p(var q:int);
    var a:array[5] of
          array[10] of int;
    var b:int;
  begin
    b := 1 + 2;
    b := b + z;
    q := q + 1;
    b := a[4][8];
    if b>1 then b:=0 end
  end p;
begin
  z := 5;
  p(z);
end main.
```

```
t1 := *(fp+b_offset)
t2 := *(fp+SL_offset)
t3 := *(t2+z_offset)
t4 := t2 + t3
*(fp+b_offset) := t4
```

27

Example



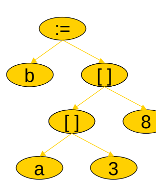


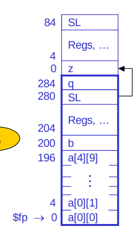

```
module main;
  var z:int;
  procedure p(var q:int);
    var a:array[5] of
          array[10] of int;
    var b:int;
  begin
    b := 1 + 2;
    b := b + z;
    q := q + 1;
    b := a[4][8];
    if b>1 then b:=0 end
  end p;
begin
  z := 5;
  p(z);
end main.
```

```
t1 := *(fp+q_offset)
t2 := *(fp+q_offset)
t3 := *(t2+0)
t4 := 1
t5 := t3 + t4
*(t1+0) := t5
```

28

Example



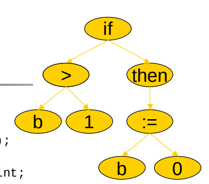


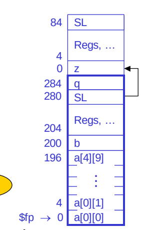

```
module main;
  var z:int;
  procedure p(var q:int);
    var a:array[5] of
          array[10] of int;
    var b:int;
  begin
    b := 1 + 2;
    b := b + z;
    q := q + 1;
    b := a[3][8];
    if b>1 then b:=0 end
  end p;
begin
  z := 5;
  p(z);
end main.
```

```
t1 := 3
t2 := 40
t3 := t1 * t2
t5 := fp + t3
t6 := 8
t7 := 4
t8 := t6 * t7
t9 := t5 + t8
t10:= *(t9+a_offset)
*(fp+b_offset) := t10
```

29

Example



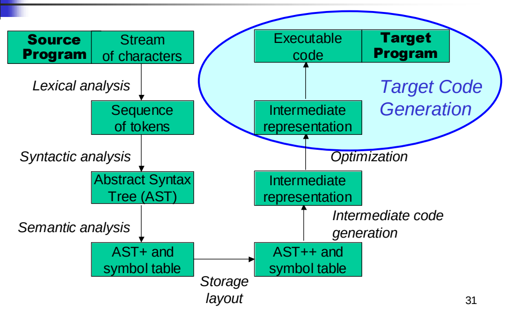



```
module main;
  var z:int;
  procedure p(var q:int);
    var a:array[5] of
          array[10] of int;
    var b:int;
  begin
    b := 1 + 2;
    b := b + z;
    q := q + 1;
    b := a[3][8];
    if b>1 then b:=0 end
  end p;
begin
  z := 5;
  p(z);
end main.
```

```
  t1 := *(fp+b_offset)
  t2 := 1
  t3 := t1 > t2
  iffalse t3 goto 1
  t4 := 0
  *(fp+b_offset) := t4
1:
```

30

Prototype compiler structure



31

Target Code Generation

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

32

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

33

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

34

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

35

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

36

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,d1` **or**
  - `inc.l d1`

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

37

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 d1,-(sp)`

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

38

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

39

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

40

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

41

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)

42

Which variables go in registers?

- Temporary variables: easy to allocate
  - Defined and used exactly once, during expression eval
    - So 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 params, across calls, …:
  - Really hard. A research project?

PL/0

43

PL/0's simple allocator design

- 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

44

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

45

Example

ICG

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

vs

PL/0

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

46

Codegen for assignments

ICG

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

vs

PL/0

```
void AssignStmt::codegen(SymTabScope* s, RegBank* rb) {
  int offset;
  Reg base = _lvalue->codegen_address(s, rb, offset);
  Reg result = _expr->codegen(s, rb);
  TheAssembler->store(result, base, offset);
  rb->freeReg(base);
  rb->freeReg(result);
}
```

47

Codegen for if statements

PL/0

```
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);
}
```

48

Codegen for call statements

PL/0

```
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;
  SymTabEntry* ste = s->lookup(_ident, enclScope);
  Reg staticLink = s->getFPOf(enclScope, rb);
  TheAssembler->push(staticLink);
  rb->freeReg(staticLink);
  rb->saveRegs(s);
  TheAssembler->call(_ident);
  rb->restoreRegs(s);
  TheAssembler->popMultiple((_args->length() + 1) *
                            TheAssembler->wordSize());
}
```

49

Another example

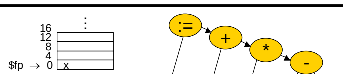
ICG

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

PL/0

```
Reg BinOp::codegen(SymTabScope* s, RegBank* rb) {
  Reg expr1 = _left->codegen(s, rb);
  Reg expr2 = _right->codegen(s, rb);
  rb->freeReg(expr1);
  rb->freeReg(expr2);
  Reg dest = rb->newReg();
  TheAssembler->binop(_op, dest, expr1, expr2);
  return dest;
}
```

50



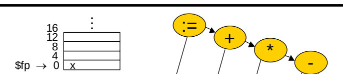
Example

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

		8	9	10	11	12	13
lw	$8, 0($fp)						
li	$9, 2						
lw	$10, 0($fp)						
li	$11, 1						
sub	$12, $10, $11						
mul	$10, $9, $12						
add	$9, $8, $10						
sw	$9, 0($fp)						

Free after use: 5 regs

51



Example, con't

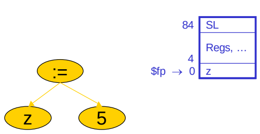
x := x + 2 * (x - 1)

		8	9	10	11	12	13
lw	$8, 0($fp)						
li	$9, 2						
lw	$10, 0($fp)						
li	$11, 1						
sub	$10, $10, $11						
mul	$9, $9, $10						
add	$8, $8, $9						
sw	$8, 0($fp)						

Free <u>before</u> use: 4 regs

52

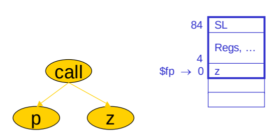
Example




```
module main;
  var z:int;
  procedure p(var q:int);
    var a:array[5] of
          array[10] of int;
    var b:int;
  begin
    b := 1 + 2;
    b := b + z;
    q := q + 1;
    b := a[4][8];
    if b>1 then b:=0 end
  end p;
begin
  z := 5;
  p(z);
end main.
```

53

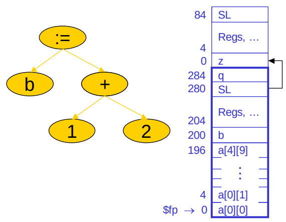
Example




```
module main;
  var z:int;
  procedure p(var q:int);
    var a:array[5] of
          array[10] of int;
    var b:int;
  begin
    b := 1 + 2;
    b := b + z;
    q := q + 1;
    b := a[4][8];
    if b>1 then b:=0 end
  end p;
begin
  z := 5;
  p(z);
end main.
```

54

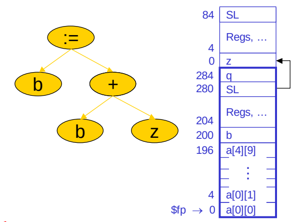
Example




```
module main;
  var z:int;
  procedure p(var q:int);
    var a:array[5] of
          array[10] of int;
    var b:int;
  begin
    b := 1 + 2;
    b := b + z;
    q := q + 1;
    b := a[4][8];
    if b>1 then b:=0 end
  end p;
begin
  z := 5;
  p(z);
end main.
```

55

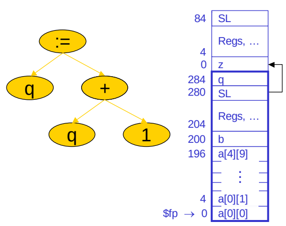
Example




```
module main;
  var z:int;
  procedure p(var q:int);
    var a:array[5] of
          array[10] of int;
    var b:int;
  begin
    b := 1 + 2;
    b := b + z;
    q := q + 1;
    b := a[4][8];
    if b>1 then b:=0 end
  end p;
begin
  z := 5;
  p(z);
end main.
```

56

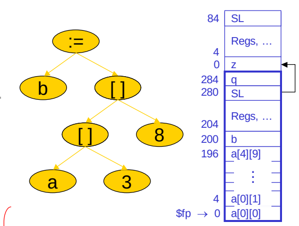
Example




```
module main;
  var z:int;
  procedure p(var q:int);
    var a:array[5] of
          array[10] of int;
    var b:int;
  begin
    b := 1 + 2;
    b := b + z;
    q := q + 1;
    b := a[4][8];
    if b>1 then b:=0 end
  end p;
begin
  z := 5;
  p(z);
end main.
```

57

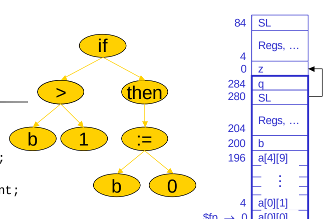
Example




```
module main;
  var z:int;
  procedure p(var q:int);
    var a:array[5] of
          array[10] of int;
    var b:int;
  begin
    b := 1 + 2;
    b := b + z;
    q := q + 1;
    b := a[3][8];
    if b>1 then b:=0 end
  end p;
begin
  z := 5;
  p(z);
end main.
```

58

Example




```
module main;
  var z:int;
  procedure p(var q:int);
    var a:array[5] of
          array[10] of int;
    var b:int;
  begin
    b := 1 + 2;
    b := b + z;
    q := q + 1;
    b := a[3][8];
    if b>1 then b:=0 end
  end p;
begin
  z := 5;
  p(z);
end main.
```

59