Compiler Construction

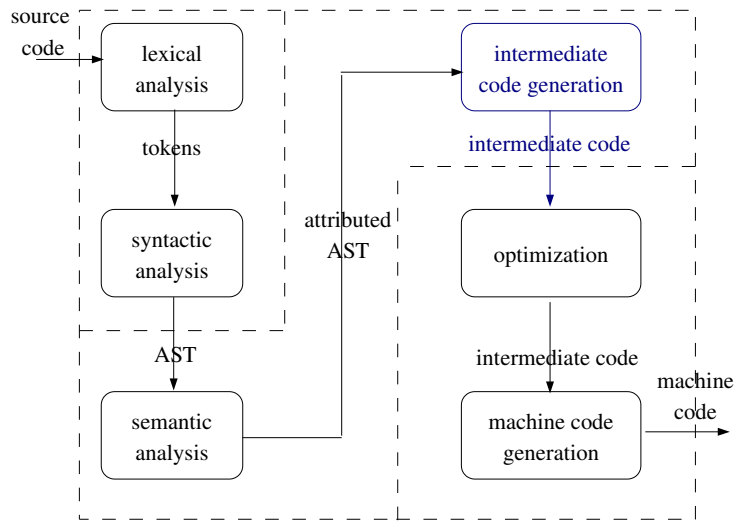
Intermediate Code

Lennart Andersson

Revision 2012-02-21

2012

Front end / Back end



Why intermediate code?

Separate front end from back end

- The front end is (reasonably) independent of the target machine
- The back end is (reasonably) independent of the source language
- For M languages and N machines
  we need M front ends + N back ends
  to construct M*N compilers

Intermediate code for portability

Portability

- Implement the back end as an interpreter (virtual machine) in some lower level language (e.g., C) that is available on many platforms.
- Easy to port the back end
- Slower execution (than compilation to native machine code)

Examples

- Pascal (PCode)
- Java (Java bytecode)

Intermediate code for interoperability

Language interoperability

- develop front ends for many languages to the same intermediate language and run-time system
- a program can be composed of different parts implemented in different languages
- non-trivial for OO languages because of run-time type information and garbage collection

Examples

- The .NET platform with MSIL (Microsoft Intermediate Language): Visual C++, Visual Basic, C#, J#, Haskell, ...
- The JVM platform with Java bytecode Java, Lisp, Scheme, Ada, Prolog, Eiffel, Scala, ...
- ...

Common types of intermediate code

3-address code, e.g.

ADD a1 a2 a3

add the values at addresses a1 and a2 and store the result at address a3

Close to an ordinary register-based machine. Good for optimization.

Stack code (stack for temporary values), e.g.

ADD

pop the two topmost values off the stack, add, and push the result

Slightly higher-level. Common for virtual machines.

Example

x = a + b * c + d

3-address code
MUL b c #0
ADD a #0 #1
ADD #1 d x

stack code
PUSH a
PUSH b
PUSH c
MUL
ADD
PUSH d
ADD
POP x

The above is pseudocode.
Variable addresses are more complex for real intermediate code

ICode — 3-address code for this course

Instructions work directly on memory, rather than on registers

- no load/store instruction

unlimited number of temporary varuables ("virtual registers")

- can be mapped to real registers or/and locations in activation records at machine code translations (register allocation)

special addressing modes that fit our run-time model for procedural languages

- access to local variables
- access to non-local variables (via the static link)
- assume that all variables have the same size (32-bit word)

special instructions for method calls

- CALL for method call
- ENTER for creating an activation record
- RETURN for returning to the caller
- ALLOC/DEALLOC for allocation of memory for arguments

ICode abstract grammar

```
Code ::= Instruction*;
abstract Instruction;
Move: Instruction ::= Operand Address;
Call: Instruction ::= <Label: String> <Levels: int>;
Enter: Instruction ::= <Vars: int> <Temps: int>
Return: Instruction;
Alloc: Instruction ::= <Size: int>;
Dealloc: Instruction ::= <Size: int>;
LabelDecl: Instruction ::= <Label: String>;
```


. . . ICode abstract grammar. . .

```
abstract Operand;
IntConst: Operand ::= <Value: String>;
abstract Address: Operand;
Temp: Address ::= <Number: int>;
Argument: Address ::= <Number: int>;
Variable: Address ::= <Levels: int> <Number: int>;
Parameter: Variable;
Result: Address;
Returned: Address;
```


. . . ICode abstract grammar. . .

```
abstract BinOpr: Instruction ::=
       Operand1:Operand Operand2:Operand Address;
abstract IntOpr: BinOpr;
IntAdd: IntOpr;
IntSub: IntOpr;
IntMul: IntOpr;
IntDiv: IntOpr;
```


. . . ICode abstract grammar

```
abstract BoolOpr: BinOpr;
IntEq: BoolOpr;
IntNe: BoolOpr;
IntLt: BoolOpr;
IntLe: BoolOpr;
IntGt: BoolOpr;
IntGe: BoolOpr;

Jmp: Instruction ::= <Label: String>;
abstract JmpCond: Jmp ::= Operand <Label: String>;
JmpF: JmpCond;
JmpT: JmpCond;
```

ICode abstract grammar

An ICode program is a sequence of instructions.

Things to note:

- Each instruction has at most 3 operands.
- An operand is an address or an integer constant
- The are six kinds of addresses: Variable, Parameter, Argument, Temp, Result and Returned.

See F11b for a detailed definition of ICode.

Example: Simple computation

```
void p() {
  int a;
  int b;
  a = 1;
  b = a + 1;
}
```

```
p:ENTER vars(2) temps(0)
  MOV 1 var(0,0)
  INTADD  var(0,0) 1 var(0,1)
  RETURN
```

- This activation has 2 variabels and uses no temporaries.
- levels in var(levels, offset) is the number of statics links that must be followed.
- in this example both variables are local and levels=0.
- The first variable has offset 0.

Computation with temporary variables

```
void p() {
  int x;
  int y;
  x = 2;
  y = 3 + 4*x;
}
```

```
p: ENTER vars(2) temps(1)
   MOV 2 var(0,0)
   INTMUL 4 var(0,0) #0
   INTADD 3 #0 var(0,1)
   RETURN
```


Non-local access

```
void p1() {
  int a, b, c;
  void p2() {
    int x;
    x = c;
  }
  ...
  p2();
}
```

```
p1:     ENTER vars(3) temps()
        ...
        RETURN
p1.p2:  ENTER vars(1) temps(0)
        MOV var(1,2) var(0,0)
        RETURN
```

NB! In the intermediate code all labels are global.

Different methods that have the same name in the user program must be given unique names.

This is an example of *name mangling*:

- a systematic scheme for changing overloaded user-defined names so that they become unique.

Endless loop

```
void p() {
  int a = 0;
  loop forever {
    a++;
  }
}
```

```
p:  ENTER vars(1) temps(0)
    MOV 0 var(0,0)
p.1:INTADD var(0,0) 1 var(0,0)
    JMP p.1
    RETURN
```

to make all label names unique. E.g.,

- p.q for a method q inside p
- p.1, p.2, p.3, ... for labels inside the code for p.

Conditional statement

```
void p() {
  int val;
  int absVal;
  val = ...;
  if (val > 0)
    absVal = val;
  else
    absVal = -val;
  ...
}
```

```
p:  ENTER vars(2) temps(1)
    MOV ... var(0,0)
    INTGT var(0,0) 0 #0
    JMPF #0 p.1
    MOV var(0,0) var(0,1)
    JMP p.2
p.1:INTSUB 0 var(0,0) var(0,1)
p.2:...
    RETURN
```


For loop

```
void p() {
  int sum = 0;
  for (int k = 1;
       k<=10;
       k++) {
    sum = sum + k;
  }
}
```

```
p:  ENTER vars(2) temps(1)
    MOV 0 var(0,0)
    MOV 1 var(0,1)
p.2:INTLE var(0,1) 10 #0
    JMPF #0 p.1
    INTADD var(0,0) var(0,0) var(0,1)
    INTADD var(0,1) 1 var(0,0)
    JMP p.2
p.1:RETURN
```


Simple method call

```
void p0(){
  void p1() {
    int a;
    int b;
    ...;
    p2()
    ...
  }
  void p2() {
    int x;
    ...
  }
  p1();
}
```

```
p1:ENTER vars(2) temps(0)
   ...
   CALL p2 levels(1)
   ...
   RETURN
p2:ENTER vars(1) temps(0)
   ...
   RETURN
```

Calling a method at another block level

```
void p1() {
  void p2() {
    void p3() {
      ...
    }
    p1();
    p2();
    p3();
  }
  ...
}
```

```
p1.p2:ENTER vars(0) temps(0)
      CALL p1 levels(2)
      CALL p1.p2 levels(1)
      CALL p1.p2.p3 levels(0)
      RETURN
```


Method with parameters

```
int f(int x, int y) {
  return x + y * 2
}
void p() {
  int a;
  a = f(3,7);
}
```

```
f: ENTER vars(0) temps(1)
   INTMUL par(0,1) 2  #0
   INTADD par(0,0) #0 result
   RETURN
p: ENTER vars(1) temps(0)
   ALLOC size(2)
   MOV 3 arg(0)
   MOV 7 arg(1)
   CALL f levels(1)
   MOV returned var(0,0)
   DEALLOC size(2)
   RETURN
```


Nested calls

```
void p() {
int a;
a = f(f(4,5),f(6,7));
}
```

```
p:  ENTER vars(1) temps(2)
    ALLOC size(2)
    MOV 4 arg(0)
    MOV 5 arg(1)
    CALL f levels(1)
    MOV returned #0
    MOV 6 arg(0)
    MOV 7 arg(1)
    CALL f levels(1)
    MOV returned #1
    MOV #0 arg(0)
    MOV #1 arg(1)
    CALL f levels(1)
    MOV returned var(0,0)
    DEALLOC size(2)
    RETURN
```


More about temporary variables

Temporary variables — used for storing intermediate results in computations of expressions

- assigned once, used once
- could in principle be reused, e.g., by keeping track of them in a stack

But reuse is better to handle late (after optimization)

- Optimization will typically introduce new temps
- A simple optimization step can implement reuse of the temps

Without reuse of temporary variables

```
int p(int x) {
  return 3 + 4*x + 5 * x * x;
}
```

```
p: ENTER vars(0) temps(4)
   INTMUL 4 par(0,0) #0
   INTADD 3 #0 #1
   INTMUL 5 par(0,0) #2
   INTMUL #2 par(0,0) #3
   INTADD #1 #3 result
   RETURN
```


With reuse of temporary variables

```
int p(int x) {
  return 3 + 4*x + 5 * x * x;
}
```

```
p: ENTER vars(0) temps(2)
   INTMUL 4 par(0,0) #0
   INTADD 3 #0 #0
   INTMUL 5 par(0,0) #1
   INTMUL #1 par(0,0) #1
   INTADD #0 #1 result
   RETURN
```


Generation of intermediate code

- Use jadd aspects on the program AST to generate the ICode AST (assignment 6)
- Use jadd aspects on the ICode AST to generate the assembly code (in the project)
- ICode abstract grammar
  - The generated AST classes include constructors so that you can easily create an ICode AST
  - The assignment example includes an aspect for prettyprinting an ICode AST as text.

Support computations in the program AST

compute the level for each block

the main procedure name is defined at level 0

the body of the main procedure is at level 1

and so on

enumerate the declared variables in each block

within each block, local variables are enumerated according to their declaration order, starting at 0

each variable declaration keeps track of its order number and the block level where it is declared

enumerate the parameters in each procedure in the same way

API for IdDecl

```
class IdDecl {
  // returns the order number for the variable
  int getVarNum() {

  }

  // returns the level where the variable is declared.
  int getBlockLevel() {

  }
}
```


Example – Abstract grammar

```
Program ::= Decl* Stmt*;
abstract Decl;
...
abstract Stmt;
AssignStmt: Stmt ::= IdUse Expr;
...
abstract Expr;
AddExpr: Expr ::= Left:Expr Right:Expr;
...
IdUse: Expr ::= <ID>
```


Code generation

```
aspect CodeGeneration {
  // Generates instructions and adds them to code.
  void Stmt.genCode(Code code, TempFactory f,
                    int blockLevel) {error("not implemented");}

  // Generates instructions and adds them to code.
  // Returns the operand holding the resulting value.
  Operand Expr.genCode(Code code, TempFactory f,
                       int blockLevel) {error("not implemented");

  void AssignStmt.genCode(Code code, ... ) {
    Operand op = getExpr().genCode(code, f, blockLevel);
    Instruction instr =
       new Move(op, getIdUse().genAddress(blockLevel));
    code.addInstruction(instr);
  }
  Address IdUse.genAddress(int blockLevel) {
    return decl.genVariable(blockLevel);
  }
}
```


Code generation

```
Variable IdDecl.genVariable(int blockLevel) {
  return new Variable(getVarNum(), blockLevel-getBlockLevel());
}

Operand AddExpr.genCode(Code code, TempFactory f,
                        int blockLevel) {
  Operand op1 = getLeft().genCode(code, blockLevel);
  Operand op2 = getRight().genCode(code, blockLevel);
  Temp t = f.create();
  code.addInstruction(new IntAdd(op1, op2, t));
  return t.copy();
}

Operand IdUse.genCode(Code code, TempFactory f,
                      int blockLevel) {
  return genAddress(blockLevel);
}
```

Possible extensions of ICode

Boolean operators
- AND, OR, NOT

Floating point instructions
- FLOATADD, FLOATSUB, ...
- FLOATLT, FLOATLE, ...

Instructions for OO languages
- new
- virtual method calls
- this
- instanceof
- ...

...

Summary

Two main kinds of intermediate code:
- 3-address code (like ICode)
- Stack code (like Pcode and Java bytecode)

ICode
- simple expressions and assignments
- loops and conditionals
- temporary variables
- (non-virtual) method calls

Code generation
- Straight-forward using methods in the AST