CS 322: Languages and Compiler Design II

Mark P Jones, Portland State University

Spring 2015

Week 2: Introduction to Code Generation

Repeat Warning: Code alert!

- Lots of code in the slides ahead
- Don't try to take it all in now
- Do try to understand the key details
- Do stop me and ask questions along the way!
- And note that the code is available on D2L so that you can study it more carefully later ... (e.g., in your lab session)

Review from Last Week

A simple source language

IExpr	::=	Var(String)
	|	Int(int)
	|	Plus(IExpr, IExpr)
	|	Minus(IExpr, IExpr)
BExpr	::=	LT(IExpr, IExpr)
	|	EqEq(IExpr, IExpr)
Stmt	::=	Seq(Stmt, Stmt)
	|	Assign(String, IExpr)
	|	While(BExpr, Stmt)
	|	If(BExpr, Stmt, Stmt)
	|	Print(IExpr)

Abstract syntax!

Type correct!

Abstract syntax in Java

As

```java
abstract class IExpr {
}

class Var extends IExpr {
  private String name;
  Var(String name) { this.name = name; }
}

class Int extends IExpr {
  private int num;
  Int(int num) { this.num = num; }
}

class Plus extends IExpr {
  private IExpr l, r;
  Plus(IExpr l, IExpr r) { this.l = l; this.r = r; }
}
```


Adding printing code

Ps

```java
abstract class IExpr {
  abstract String show();
}

class Var extends IExpr {
  ...
  String show() { return name; }
}

class Int extends IExpr {
  ...
  String show() { return Integer.toString(num); }
}

class Plus extends IExpr {
  ...
  String show() { return "(" + l.show() + " + " + r.show() + ")"; }
}
```

Evaluating expressions

ls

```
abstract class IExpr { ...
  abstract int eval(Memory mem);
}
class Var extends IExpr { ...
  int eval(Memory mem) { return mem.load(name); }
}
class Int extends IExpr { ...
  int eval(Memory mem) { return num; }
}
class Plus extends IExpr { ...
  int eval(Memory mem) { return l.eval(mem) + r.eval(mem); }
}
class Minus extends IExpr { ...
  int eval(Memory mem) { return l.eval(mem) - r.eval(mem); }
}
```

Is there anything special about our source language that allows us to do this?

Interpreting a Target Language

A simple target language

Program	::=	a collection of "Basic Blocks"
Block	::=	Block(Label, Code)
Reg	::=	... register ...
Code	::=	Stop
	|	Goto(Block)
	|	Cond(Reg, Block, Block)
	|	Load(Reg, Var, Code)
	|	Store(Var, Reg, Code)
	|	Immed(Reg, Int, Code)
	|	Op(Reg, Reg, Op, Reg, Code)
	|	Pcode(Reg, Code)

A simple target language

Program	::=	a collection of "Basic Blocks"
Block	::=	Block(Label, Code)
Reg	::=	... register ...
Code	::=	Stop
	|	Goto(Block)
	|	Cond(Reg, Block, Block)
	|	Load(Reg, Var, Code)
	|	Store(Var, Reg, Code)
	|	Immed(Reg, Int, Code)
	|	Op(Reg, Reg, Op, Reg, Code)
	|	Pcode(Reg, Code)

Stop
Terminate the program

A simple target language

Program	::=	a collection of "Basic Blocks"
Block	::=	Block(Label, Code)
Reg	::=	... register ...
Code	::=	Stop
	|	Goto(Block)
	|	Cond(Reg, Block, Block)
	|	Load(Reg, Var, Code)
	|	Store(Var, Reg, Code)
	|	Immed(Reg, Int, Code)
	|	Op(Reg, Reg, Op, Reg, Code)
	|	Pcode(Reg, Code)

Goto(b)
Branch to block **b**

A simple target language

Program	::=	a collection of "Basi…
Block	::=	Block(Label, Code)
Reg	::=	... register ...
Code	::=	Stop
	|	Goto(Block)
	|	Cond(Reg, Block, Block)
	|	Load(Reg, Var, Code)
	|	Store(Var, Reg, Code)
	|	Immed(Reg, Int, Code)
	|	Op(Reg, Reg, Op, Reg, Code)
	|	Pcode(Reg, Code)

Cond(r, t, f)
Branch to block **t** (if register **r** is non-zero) or to block **f** (if register **r** is zero)
Also written **r** → **t, f**
(the *"McCarthy conditional"*)

A simple target language

Program ::= a collection of "Basic Blocks"

Block ::= Block(Label, Code)

Reg ::= ... register ...

Code ::= Stop
| Goto(Block)
| Cond(Reg, Block, Block)
| Load(Reg,Var, Code)
| Store(Var, Reg, Code)
| Immed(Reg, Int, Code)
| Op(Reg, Reg, Op, Reg, Code)
| Pcode(Reg, Code)

Load(r, v, c)
Load the contents of variable **v** into register **r** and then execute code **c**

A simple target language

Program ::= a collection of "Basic Blocks"

Block ::= Block(Label, Code)

Reg ::= ... register ...

Code ::= Stop
| Goto(Block)
| Cond(Reg, Block, Block)
| Load(Reg,Var, Code)
| Store(Var, Reg, Code)
| Immed(Reg, Int, Code)
| Op(Reg, Reg, Op, Reg, Code)
| Pcode(Reg, Code)

Store(v, r, c)
Store the value in register **r** in the variable **v** and then execute code **c**

A simple target language

Program ::= a collection of "Basic Blocks"

Block ::= Block(Label, Code)

Reg ::= ... register ...

Code ::= Stop
| Goto(Block)
| Cond(Reg, Block, Block)
| Load(Reg,Var, Code)
| Store(Var, Reg, Code)
| Immed(Reg, Int, Code)
| Op(Reg, Reg, Op, Reg, Code)
| Pcode(Reg, Code)

Immed(r, n, c)
Set register **r** to the integer **n** and then execute code **c**

A simple target language

Program ::= a collection of "Basic Blocks"

Block ::= Block(Label, Code)

Reg ::= ... register ...

Code ::= Stop
| Goto(Block)
| Cond(Reg, Block, Block)
| Load(Reg,Var, Code)
| Store(Var, Reg, Code)
| Immed(Reg, Int, Code)
| Op(Reg, Reg, Op, Reg, Code)
| Pcode(Reg, Code)

Op(r_1, r_2, op, r_3, c)
Combine the values in registers **r_2** and **r_3** using the operator **op**, save the result in **r_2**, and then execute code **c**

A simple target language

Program ::= a collection of "Basic Blocks"

Block ::= Block(Label, Code)

Reg ::= ... register ...

Code ::= Stop
| Goto(Block)
| Cond(Reg, Block, Block)
| Load(Reg,Var, Code)
| Store(Var, Reg, Code)
| Immed(Reg, Int, Code)
| Op(Reg, Reg, Op, Reg, Code)
| Pcode(Reg, Code)

Pcode(r, c)
Print the value in register **r** and then execute code **c**

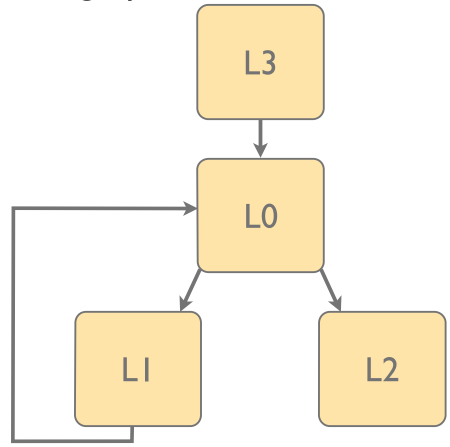
Simple example: `i = j + 1`

"Assembly"	"Abstract syntax"
`r1  <- [j]`	`new Load(r1, j,`
`r2  <- 1`	`new Immed(r2, 1,`
`r3  <- r1 + r2`	`new Op(r3, r1, '+', r2,`
`[i] <- r3`	`new Store(i, r3,`
`...`	`...))))`

A sample program

```
L0:                       goto L0
  r6 <- [i]
  r1 <- 11              L2:
  r1 <- r6<r1             r0 <- [t]
  r1 -> L1, L2            print r0
                          stop
L1:
  r5 <- [t]             L3:
  r4 <- [i]               r8 <- 0
  r4 <- r5+r4             [t] <- r8
  [t] <- r4               r7 <- 0
  r3 <- [i]               [i] <- r7
  r2 <- 1                 goto L0
  r2 <- r3+r2
  [i] <- r2             Entry is L3
```


As a "flowgraph"



Observations

- Each basic block contains a sequence of instructions
- We only enter at the top of a block
- We only leave at the end of a block
- There are no high-level control constructs, just a flowgraph of edges between the basic blocks
- Arithmetic operators work only on registers

  Separate instructions are needed to load/store values in memory, load immediate (constant) values, etc.
- Overall, this language is very similar to the assembly code for a RISC (reduced instruction set) computer

It's just another language ...

At

Abstract syntax!

```
abstract class Code { }

class Stop extends Code {
  Stop() {}
}

class Goto extends Code {
  private Block block;
  Goto(Block block) { this.block = block; }
}

class Load extends Code {
  private Reg reg;
  private String name;
  private Code next;
  Load(Reg reg, String name, Code next) {
    this.reg  = reg; this.name = name; this.next = next;
  }
}
```


... albeit a little unusual ...

At

```
class Program {
  private Vector<Block> blocks
     = new Vector<Block>();
  Block block(Code code) {
    Block l
     = new Block(blocks.size(),
                 code);
    blocks.add(l);
    return l;
  }
  Block block() {
    return block(null);
  }
}
```

```
class Reg {
  private static int count = 0;
  private int num;

  Reg() { num = count++; }
}
```

```
class Block {
  private int num;
  private Code code;

  Block(int num, Code code) {
    this.num  = num;
    this.code = code;
  }

  void set(Code code) {
    this.code = code;
  }

  Code code() {
    return code;
  }
}
```


We can define print methods ...

Pt

```
class Program { ...
  void show() {
    for (int i=0; i<blocks.size(); i++) {
      blocks.elementAt(i).print();
      System.out.println();
    }
  }
}

abstract class Code { abstract void print(); }

class Stop extends Code {
  void print() { System.out.println("  stop"); }
}
class Load extends Code { ...
  void print() {
    System.out.println("  " + reg + " <- [" + name + "]");
    next.print();
  }
}
```

We can define run methods ...

```
abstract class Code {
  ...
  abstract Block run(Memory mem);
}

class Stop extends Code {
  ...
  Block run(Memory mem) { return null }
}

class Goto extends Code {
  ...
  Block run(Memory mem) { return block; }
}

class Cond extends Code {
  ...
  Block run(Memory mem) { return reg.getBool() ? t : f; }
}
```


... continued

```
class Load extends Code { ...
  Block run(Memory mem) {
    reg.set(mem.load(name));
    return next.run(mem);
  }
}
class Op extends Code { ...
  Block run(Memory mem) {
    switch (op) {
      case + : r.set(x.get() + y.get());
               break;
      case - : r.set(x.get() - y.get());
               break;
      case < : r.setBool(x.get() < y.get());
               break;
      case = : r.setBool(x.get() == y.get());
               break;
    }
    return next.run(mem);
  }
}
```


Building a Compiler

Compilation schemes

`e.compileTo(reg, next)`

generates Code that will:

- evaluate the expression `e`
- leave the result in register `reg`
- and then execute the code given by `next`

`s.compile(prog, next)`

generates Code that will:

- execute the statement `s`
- and then execute the code given by `next`
- adding any new blocks that are required to `prog`

Compilation of expressions

```
abstract class IExpr { ...
  abstract Code compileTo(Reg reg, Code next);
}
class Var extends IExpr { ...
  Code compileTo(Reg reg, Code next) {
    return new Load(reg, name, next);
  }
}
class Int extends IExpr { ...
  Code compileTo(Reg reg, Code next) {
    return new Immed(reg, num, next);
  }
}
class Plus extends IExpr { ...
  Code compileTo(Reg reg, Code next) {
    Reg tmp = new Reg();
    return l.compileTo(tmp,
             r.compileTo(reg,
               new Op(reg, tmp, '+', reg, next)));
  }
}
```

Note: we are adding methods to the abstract syntax classes for the source ... that produce values using the abstract syntax classes of the target

Compilation of statements

```
abstract class Stmt { ...
  abstract Code compile(Program prog, Code next);
}

class Seq extends Stmt { ...
  Code compile(Program prog, Code next) {
    return l.compile(prog, r.compile(prog, next));
  }
}

class Assign extends Stmt { ...
  Code compile(Program prog, Code next) {
    Reg tmp = new Reg();
    return rhs.compileTo(tmp, new Store(lhs, tmp, next));
  }
}
```

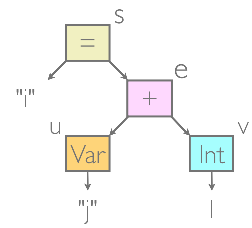
Compilation of statements

```
abstract class Stmt { ...
  abstract Code compile(Program prog, Code next);
}

class Seq extends Stmt { ...
  Code compile(Program prog, Code next) {
    return l.compile(prog,
           r.compile(prog,
           next));
  }
}

class Assign extends Stmt { ...
  Code compile(Program prog, Code next) {
    Reg tmp = new Reg();
    return rhs.compileTo(tmp,
           new Store(lhs, tmp,
           next));
  }
}
```


Example

Concrete syntax:

```
i = j + 1
```



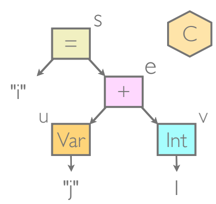
Abstract syntax:

```
s = new Assign("i",
      new Plus(new Var("j"),
               new Int(1)  )  )
```

s u e v

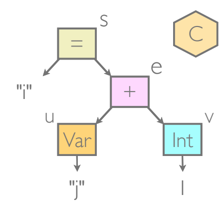

```
s.compile(prog, next)

  = e.compileTo(t₁,
      new Store("i", t₁, next))
```

```
class Assign extends Stmt { ...
  Code compile(Program prog, Code next) {
    Reg tmp = new Reg();
    return rhs.compileTo(tmp,
           new Store(lhs, tmp, next));
  }
}
```




```
s.compile(prog, next)

  = e.compileTo(t₁,
      new Store("i", t₁, next))
```

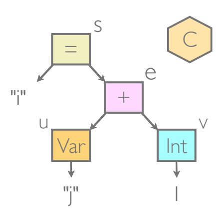
```
class Plus extends IExpr { ...
  Code compileTo(Reg reg, Code next) {
    Reg tmp = new Reg();
    return l.compileTo(tmp,
           r.compileTo(reg,
           new Op(reg, tmp, '+', reg, next)));
  }
}
```




```
s.compile(prog, next)

  = e.compileTo(t₁,
      new Store("i", t₁, next))
```

```
class Plus extends IExpr { ...
  Code compileTo(Reg reg, Code next) {
    Reg tmp = new Reg();
    return l.compileTo(tmp,
           r.compileTo(reg,
           new Op(reg, tmp, '+', reg, next)));
  }
}
```

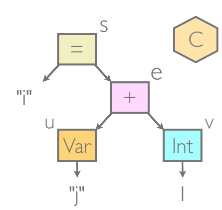
```
  = u.compileTo(t₂,
      v.compileTo(t₁,
      new Op(t₁, t₂, '+', t₁,
      new Store("i", t₁, next))))
```

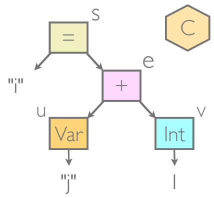



```
s.c
```

```
class Var extends IExpr { ...
  Code compileTo(Reg reg, Code next) {
    return new Load(reg, name, next);
  }
}
```

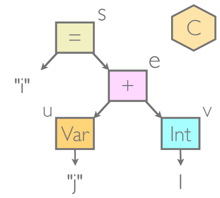
```
    ew Store("i", t₁, next))

  = u.compileTo(t₂,
      v.compileTo(t₁,
      new Op(t₁, t₂, '+', t₁,
      new Store("i", t₁, next))))
```





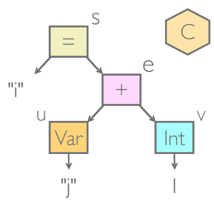

```
s.c
  = u.compileTo(t2,
    v.compileTo(t1,
    new Op(t1, t2, '+', t1,
    new Store("i", t1, next))))

  = new Load(t2, "j",
    v.compileTo(t1,
    new Op(t1, t2, '+', t1,
    new Store("i", t1, next))))
```

```
class Var extends IExpr { ...
  Code compileTo(Reg reg, Code next) {
    return new Load(reg, name, next);
  }
}
```

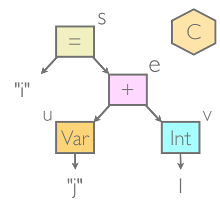

```
s.compile(prog, next)

  = e.compileTo(t1,
    new Store("i", t1, next))

  = u.compileTo(t2
                                    ))))

  = new Load(t2, "j",
    v.compileTo(t1,
    new Op(t1, t2, '+', t1,
    new Store("i", t1, next))))
```

```
class Int extends IExpr { ...
  Code compileTo(Reg reg, Code next) {
    return new Immed(reg, num, next);
  }
}
```


```
s.compile(prog, next)

  = e.compileTo(t1,
    new Store("i", t1, next))

  = u.compileTo(t2
                                    ))))

  = new Load(t2, "j",
    new Immed(t1, 1,
    new Op(t1, t2, '+', t1,
    new Store("i", t1, next))))
```

```
class Int extends IExpr { ...
  Code compileTo(Reg reg, Code next) {
    return new Immed(reg, num, next);
  }
}
```

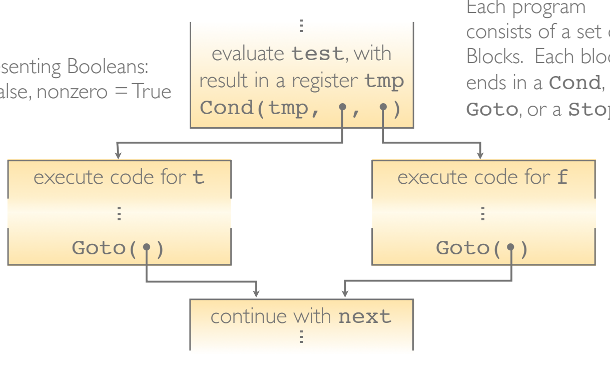

```
s.compile(prog, next)

  = e.compileTo(t1,
    new Store("i", t1, next))

  = u.compileTo(t2,
    v.compileTo(t1,
    new Op(t1, t2, '+', t1,
    new Store("i", t1, next))))

  = new Load(t2, "j",
    new Immed(t1, 1,
    new Op(t1, t2, '+', t1,
    new Store("i", t1,
    next))))
```

```
[ t2   <-  [j]
  t1   <-  1
  t1   <-  t2 + t1
  [i]  <-  t1
  next              ]
```


Compilation of `If`

What target code should we generate to execute a statement of the form `if test then t else f`, and then continue to execute the `next` code?

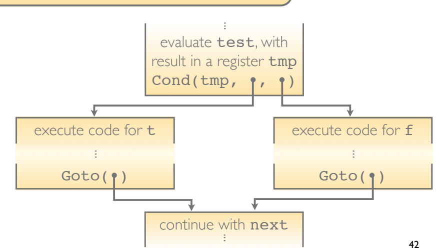
Representing Booleans:
0 = False, nonzero = True



Each program consists of a set of Blocks. Each block ends in a `Cond`, a `Goto`, or a `Stop`.

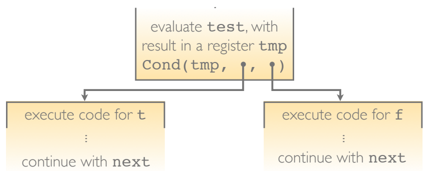
Compilation of `If`

```
class If extends Stmt { ...
  Code compile(Program prog, Code next) {
    Reg  tmp = new Reg();
    Goto got = new Goto(prog.block(next));
    return test.compileTo(tmp,
             new Cond(tmp,
                       prog.block(t.compile(prog, got)),
                       prog.block(f.compile(prog, got))));
  }
}
```



What's wrong with this alternative?

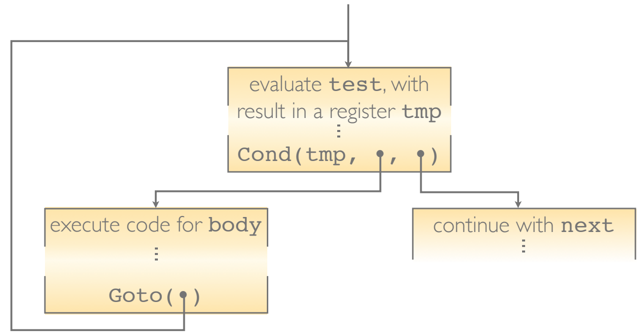
```
class If extends Stmt { ...
  Code compile(Program prog, Code next) {
    Reg tmp = new Reg();
    return test.compileTo(tmp,
           new Cond(tmp,
                    prog.block(t.compile(prog, next)),
                    prog.block(f.compile(prog, next))));
  }
}
```



Code duplication!

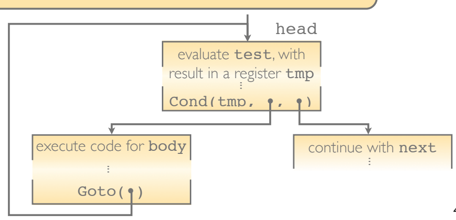
Compilation of `While`

What target code should we generate to execute a statement of the form `while (test) body`, and then continue to execute the `next` code?



Compilation of `While`

```
class While extends Stmt { ...
  Code compile(Program prog, Code next) {
    Block head = prog.block();
    Code  loop = new Goto(head);
    Reg   tmp  = new Reg();
    head.set(test.compileTo(tmp,
             new Cond(tmp,
                      prog.block(body.compile(prog, loop)),
                      prog.block(next))));
    return loop;
  }
}
```



A "main" program

```
class Main {
  public static void main(String[] args) {
    ...
    System.out.println("Compiling:");
    Program p     = new Program();
    Block   entry = p.block(s.compile(p, new Stop()));
    System.out.println("Entry point is at " + entry);
    p.show();

    System.out.println("Running on an empty memory:");
    mem       = new Memory();
    Block pc = entry;
    while (pc!=null)  {
      pc = pc.code().run(mem);
    }

    System.out.println("Done!");
  }
}
```


Running the program

```
$ java Main
Complete program is:
  ...
Running on an empty memory:
Output: 55
Compiling:
Entry point is at L3
  ...
Running on an empty memory:
Output: 55
Done!
$
```

Interpreter

compilation has preserved the original meaning!

Compiler

Compiler correctness

We can capture the conditions for the correctness of a compiler by using the following "commuting diagram":

$$\begin{tikzcd} \text{source} \arrow[r, "\text{compile}_{S\to T}"] \arrow[d, "\text{evaluate}_S"'] & \text{target} \arrow[d, "\text{evaluate}_T"] \\ \text{result}_S \arrow[r, leftrightarrow, "="] & \text{result}_T \end{tikzcd}$$

Or, in symbols: $\forall p.\ \text{evaluate}_S\,(p) = \text{evaluate}_T\,(\text{compile}_{S\to T}\,(p))$

Version summary

Compilation	C	
Interpreter	Is	It
Printing	Ps	Pt
Abstract syntax	As	At
	Source	Target

What's missing?

- Our **source language** has a very limited feature set
  - How do we represent a broader range of values, from floats, arrays, & pointers, to objects & first-class functions?
  - How do we translate a broader range of constructs, from switch statements and function calls to class definitions?
- Our **target language** has an idealized instruction set and no limit on the number of registers
  - How do we work around the limitations of real machines?
  - How do we make good use of what they do provide?
- Our **code generator** doesn't always produce efficient code
  - How do we optimize generated code?

Summary

- Abstract syntax provides a way to represent the structure of programs
- We can use an interpreter/evaluator to document the semantics of a language
- Compilation to a particular target is just one of many interesting functions that we can define on the abstract syntax of a language
- The goal of compilation is to translate between two different languages while preserving semantics
- The example in this lecture illustrates many key concepts … but there are still plenty of details for us to explore!