Compilers - Chapter 3: Code generation

- Lecturers:
  - Paul Kelly (p.kelly@imperial.ac.uk)

  - Naranker Dulay (n.dulay@imperial.ac.uk)

- Materials:
  - materials.doc.ic.ac.uk, Panopto
  - Textbook
  - Course web pages (http://www.doc.ic.ac.uk/~phjk/Compilers)
  - Piazza (https://piazza.com/class/kf7uelkyxk7aa)

The plan

- A simple language with assignments, loops etc.
- A stack-based instruction set and its code generator
- Code generation for a machine with registers:
  - an unbounded number of registers
  - a fixed number of registers
  - avoiding running out of registers
  - register allocation across multiple statements

This will lead us on to dataflow analysis and optimisation

A simple programming language with statements and loops

- Concrete syntax:

```
stat → ident ':=' exp |
       stat ';' stat |
       'for' ident 'from' exp 'to' exp 'do' stat 'od'
exp → exp binop exp |
      unop exp |
      ident |
      num
binop → '+' | '-' | '*' | '/'
unop → '-'
```

(Language based on Maple)

Abstract syntax tree data type:

- data Stat = Assign name Exp |
  Seq Stat Stat |
  ForLoop Name Exp Exp Stat
- data Exp = Binop Op Exp Exp |
  Unop Op Exp |
  Ident Name |
  Const Int
- data Op = Plus | Minus |
  Times | Divide | Minus

- type Name = [Char]

Target machine: stack machine

- To begin with we consider a computer consisting of a main store, addressed from zero up to some limit, together with a program counter, a current instruction register, a pointer to the topmost item on the stack, and a temporary register.

- We can specify what the machine does by giving an interpreter for its instruction set.

```
PROCEDURE stackmachine()
VAR store : ARRAY [0..maxmem] OF BYTE;
PC, IR, SP, T : BYTE;
BEGIN
   PC := 0;   SP := maxmem;          (* stack grows downwards *)
   REPEAT
    IR := store[PC];
    PC := PC + 4;
    CASE opcode(IR) OF
        ADD: ....action for ADD...
        SUB: ....action for SUB...
        PUSHIMM: ....action for PUSHIMM...
        PUSHABS: ....action for PUSHABS...
        COMPEQ: ....action for COMPEQ...
        JTRUE: ....action for JTRUE...
   FOREVER
END
```

This is a description of how the machine executes instructions

The function `opcode` selects the opcode part of the instruction word

Actions for each instruction defined shortly

Instruction set for stack machine

```
data Instruction
  = Add | Sub | Mul | Div              (as before)
    | PushImm Int      (push constant onto stack)
    | PushAbs Name     (push variable at given location onto stack)
    | Pop Name         (remove top of stack & store it at given loc'n)
    | CompEq           (subtract top two elements of stack, and
                        replace with 1 if the result was zero, 0 otherwise)
    | JTrue Label      (remove top item from stack; if 1 jump to label)
    | JFalse Label     (jump if stack top is 0)
    | Define Label     (set up destination for jump)
```

Note that Define is an assembler directive, not an executable instruction

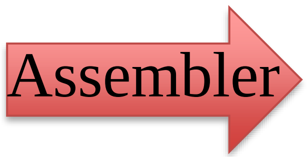
What exactly do these instructions do?

```
CASE opcode(IR) OF
  ADD:
    T:=store[SP];
    SP := SP+4;
    T:=store[SP]+T;
    store[SP]:=T;
  PUSHIMM:
    SP:=SP-4;
    store[SP]:=operand(IR);
  PUSHABS:
    T:=store[operand(IR)];
    SP:=SP-4;
    store[SP]:=T;
  POP:
    T:=store[SP];
    SP:=SP+4;
    store[operand(IR)]:=T;
  COMPEQ:
    T:=store[SP];
    SP := SP+4;
    T:=store[SP]-T;
    store[SP]:=IF T=0 THEN 1 ELSE 0;
  JTRUE:
    T:=store[SP];
    SP:=SP+4;
    IF T=1 THEN PC:=operand(IR);
```

Assembly code

- A typical assembly language sequence:

```
PushAbs i
PushImm 1
Sub
Pop i
PushAbs i
PushImm 100
CompEq
JTrue start
```



- Corresponding binary encoding:

Bits: 31 30 29 28 27 26 25 24 25 23 22 21 20 19 18 17 16 15 14 13 12 11 10 09 08 07 06 05 04 03 02 01 00

Location	Opcode	Operand
128	00001	000000000000000000000100000
132	00010	000000000000000000000000001
136	00011	000000000000000000000000000
140	00100	000000000000000000000100000
144	00001	000000000000000000000100000
148	00010	00000000000000000001100100
152	00101	000000000000000000000000000
156	00110	000000000000000000010000000

(assuming variable i is stored at address 32 (=10000 in binary))

Assembly code

- A typical assembly language sequence:

```
PushAbs i
PushImm 1
Sub
Pop i
PushAbs i
PushImm 100
CompEq
JTrue start
```

Assembler

- Corresponding binary encoding:

Bits: 31 30 29 28 27 26 25 24 25 23 22 21 20 19 18 17 16 15 14 13 12 11 10 09 08 07 06 05 04 03 02 01 00

Location	Opcode	Operand
128	00001	000000000000000000000100000
132	00010	000000000000000000000000001
136	00011	000000000000000000000000000
140	00100	000000000000000000000100000
144	00001	000000000000000000000100000
148	00010	00000000000000000001100100
152	00101	000000000000000000000000000
156	00110	000000000000000000010000000

(assuming variable i is stored at address 32 (=10000 in binary)

How does the assembler know what address "start" is?

Labels

- A typical assembly language sequence:

```
start:
    PushAbs i
    PushImm 1
    Sub
    Pop i
    PushAbs i
    PushImm 100
    CompEq
    JTrue start
```

Assembler

- Corresponding binary encoding:

Location	Opcode (Bits: 31 30 29 28 27)	Operand (Bits: 26 25 24 25 23 22 21 20 19 18 17 16 15 14 13 12 11 10 09 08 07 06 05 04 03 02 01 00)
128	00001	000000000000000000000100000
132	00010	000000000000000000000000001
136	00011	000000000000000000000000000
140	00100	000000000000000000000100000
144	00001	000000000000000000000100000
148	00010	00000000000000000001100100
152	00101	000000000000000000000000000
156	00110	000000000000000000010000000

(assuming variable i is stored at address 32 (=10000 in binary))

The label tells the assembler to associate the symbol "start" with the address of the next instruction

Representation in Haskell

- A typical assembly language sequence:

```
PushAbs i
PushImm 1
Sub
Pop i
PushAbs i
PushImm 100
CompEq
JTrue start
```

- In our code generator, assembly code is represented by the Haskell list:

```
[
 PushAbs "i",
 PushImm 1,
 Sub,
 Pop "i",
 PushAbs "i",
 CompEQ,
 Jtrue "start“
]
```

Representation in Haskell: labels

- A typical assembly language sequence:

```
start:
   PushAbs i
   PushImm 1
   Sub
   Pop i
   PushAbs i
   PushImm 100
   CompEq
   JTrue start
```

- In our code generator, assembly code is represented by the Haskell list:

```
[Define “start”,
 PushAbs "i",
 PushImm 1,
 Sub,
 Pop "i",
 PushAbs "i",
 CompEQ,
 Jtrue "start“
]
```

- This is a Haskell representation of assembly language: we add a pseudo-instruction “Define” in the instruction data type
- In assembly language, cross-references are represented using labels which are resolved by the linker

A Naive code generator for a stack machine

- We now present a syntax-directed code generator for the language with assignment and 'for' loops
- The structure of the translator is derived directly from the AST data type: we deal with each of the alternatives using a separate rule
- Begin with assignment:

```
transStat :: stat -> [instruction]
transStat (Assign (Ident id) exp) = ...
transStat (Seq s1 s2) = ...
transStat (ForLoop id e1 e2 body) = ...
```

Assignment:

```
transStat (Assign id exp)
  = transExp exp ++ [Pop id]
```

- The output code consists of instructions generated by `transExp` (see later), joined to the one element list '`[Pop id]`'.
- '`transExp exp`' yields a list of instructions, which, when executed, leave the value of the RHS of the assignment on the top of the stack.
- When the '`Pop id`' instruction is executed, it removes the value from the stack and stores it at the location specified by the name `id`.

Statement sequence:

```
transStat (Seq s1 s2)
  = transStat s1 ++ transStat s2
```

For loop:

The ‘for’ statement is a bit more complicated…

- Basic idea—given the source code:

```
for x := e1 to e2 do
  body
od
next statement
```

- we want the output code to look like:

```
  x := e1
label1:
  if x>e2 then goto label2
  body
  x := x + 1
  goto label1
label2:
  code for next statement
```

This is our “**code template**” for the ‘for’ loop

For loops…

- **Example**: Source code:

```
for x := 1 to 10 do
  a := a+x;
od
```

- … Resulting code:

```
[PushImm 1,   (initialisation)
Pop "x",
Define L1,
PushImm 10,   (test)
PushAbs "x",
CompGt.
JTrue L2,
PushAbs "a",   (body)
PushAbs "x",
Add,  Pop "a",   (store a+x in a)
PushAbs "x",   (increment)
PushImm 1,
Add,
Pop "x",        (store x+1 in x)
Jump L1,
Define L2]
```

- From the template, write down the translator:

```
transStat (ForLoop id e1 e2 body)
= transExp e1 ++ [Pop id] ++
   [Define label1] ++
   transExp e2 ++ [PushAbs id] ++ [CompGt] ++
   [JTrue label2] ++
   transStat body ++
   [PushAbs id] ++ [PushImm 1] ++ [Add] ++ [Pop id]++
   [Jump label1] ++
   [Define label2]
```

(initialisation)

(test)

(increment)

where label1 and label2 are fresh labels which have not been used so far

Expressions:

- This completes the statement part of the code generator; all that remains is to deal with expressions—which are handled just as they were in the introductory example:

```
transExp :: Exp -> [Instruction]
transExp (Binop op e1 e2)
= transExp e1 ++
  transExp e2 ++
  transOp op
transExp (Unop op e)
= transExp e ++
  transUnop op
transExp (Ident id) = [PushAbs id]
transExp (Const v) = [PushImm v]
transOp Plus = [Add]
transOp Minus = [Sub]
transOp Times = [Mul]
transOp Divide = [Div]
transUnop Minus = [Negate]
```

Conclusion

- This chapter has shown how a code generator can be written, which takes an AST as input and produces a working assembler program as output.
- We divided the problem into two parts: code generation for statements (e.g. assignment, if-then-else, while, for etc), and code generation for expressions.
- For each statement type, the code generator uses a standard "template"; the details of the statement determine how the gaps are filled in.
- For expressions we used a very simple, stack-based scheme; we will study better ways very shortly
- We haven't looked at procedures, declarations, records, etc.

Textbooks

- EaC
  - Section 4.4: ad-hoc syntax-directed translation
  - especially Figure 4.14 (pg 198)
  - Section 4.3: Attribute grammars
  - See also section 11.1
- Appel
  - Section 11.4: Expression trees, register allocation
  - Section 9: Instruction selection (Appel skips simple code generation and concentrates on finding the best instruction to match the context).
- Dragon Book
  - Chapter 2: introduction to code generation
  - Chapter 8, esp 8.1 and 8.6

This course vs the textbooks

- In this course, we translate the text into the AST, then translate the AST to assembler. Modern compilers tend to use an more than one *Intermediate Representation* (IR)
- **See EaC Chapter 5**
- The first IR is often a tree, the Abstract Syntax Tree
  - But may include statement operations and expressions uniformly
  - This is useful for more sophisticated instruction selection and register allocation techniques
- This tree is typically "flattened" into a control-flow graph or linear IR, that makes branches/jumps explicit
  - A data structure representing the assembler-level code
  - Useful for control-flow – sensitive optimisations like loop-invariant code motion
- Modern compilers often also use dependence-based graph representations, and "static single assignment" form

Appendix A

- To help clarify what is going on for students less familiar with Haskell, the next few slides offer a sketch of how to do this is Java.
- You can find the code at

  http://www.doc.ic.ac.uk/~phjk/CompilersCourse/SampleCode/Ex2-CodeGenInJava/

Step 1: define abstract syntax tree

```
public abstract class StatementTree {
    public abstract void Accept(StatementTreeVisitor v);
}

public class AssignNode extends StatementTree {
    String lhs; ExpressionTree rhs;
    AssignNode(String _lhs, ExpressionTree _rhs) {
        lhs = _lhs; rhs = _rhs;
    }
    public void Accept(StatementTreeVisitor v) {
        v.visitAssignNode(lhs, rhs);
    }
}
```

Each AST node type extends StatementTree abstract class
Each node has members, constructor, and accepts a visitor

Step 1: define abstract syntax tree

```
public class CompoundNode extends StatementTree {
    Vector body; // Vector of StatementTree
    CompoundNode(Vector _body) {
        body = _body;
    }
    public void Accept(StatementTreeVisitor v) {
        v.visitCompoundNode(body);
    }}
public class IfThenNode extends StatementTree {
    ExpressionTree cond; StatementTree body;
    IfThenNode(ExpressionTree _cond, StatementTree _body) {
        cond = _cond; body = _body;
    }
    public void Accept(StatementTreeVisitor v) {
        v.visitIfThenNode(cond, body);
    }}
```

For this example we define an AST with three node types:

- Assignment statement
- Compound statement
- If-Then statement

Each AST node type extends StatementTree abstract class
Each node has members, constructor, and accepts a visitor

Step 2: define the Visitor class

```
public abstract class StatementTreeVisitor {
   abstract void visitCompoundNode(Vector body);
   abstract void visitAssignNode(String lhs, ExpressionTree rhs);
   abstract void visitIfThenNode(ExpressionTree cond, StatementTree body);
}
```

To define a function to walk the AST, create a Visitor like this:

```
public class ExampleVisitor extends StatementTreeVisitor {
   void visitCompoundNode(Vector body) {
      // case for Compound statement node
   }
   void visitAssignNode(String lhs, ExpressionTree rhs) {
      // case for Assign statement node
   }
   void visitIfThenNode(ExpressionTree cond, StatementTree body) {
      // case for If-Then statement node
}}
```

Step 3: define a Visitor that generates code

public class TranslateVisitor extends StatementTreeVisitor {

We implement the code generator as a visitor.
We define a "visit" method for each node type

Assign node case:

```
void visitAssignNode(String lhs, ExpressionTree rhs) {
    // print instructions which, when executed, will leave
    // expression value at top of stack
    rhs.Accept(new TranslateExpVisitor());
    System.out.println("pop "+lhs);
  }

  .....
```

Step 3: define a Visitor that generates code

Compound statement node case:

```
void visitCompoundNode(Vector body) {
   // Visit each statement in the list of statements
   // that make up the Compound statement body
   for (int i=0; i<body.size(); i++)
      ((StatementTree)body.elementAt(i)).Accept(this);
}
```

Step 3: define a Visitor that generates code

Assign node case:

```
…
void visitIfThenNode(ExpressionTree cond, StatementTree body) {
   // print instructions which, when executed, will leave
   // expression value at top of stack
   UniqueLabel skiplabel = new UniqueLabel();
   cond.Accept(new TranslateExpVisitor());
   System.out.println("JFalse "+skiplabel.toString());
   body.Accept(this);
   System.out.println("Define "+skiplabel.toString());
}
```

(to complete this code you need to add an AST for expressions)

If you don’t use a visitor…

```
public class TurnNode extends StatementTree {
  int degrees;

  TurnNode(int d) {
    degrees = d;
  }
  public void print() {
    System.out.println("turn "+degrees+" degrees");
  }
  public void interpret() {
    System.out.println("please turn "+degrees);
  }
  public void orientation() {
    pose.setHeading(pose.getHeading)+degrees);
  }
}
```

- *You need to add a method for each operation that involves a traversal of the AST*
- *For every StatementTree subclass*

```
public class InterpretVisitor extends TreeVisitor {
   void visitStatementList(StatementTree first,
                           StatementTreeList rest) {
      first.Accept(this);
      if (rest != null) {
         rest.Accept(this);
      }
   }
   void visitTurnNode(int degrees) {
      System.out.println("Please turn "+degrees+" degrees");
   }
   void visitForwardNode(int distance) {
      System.out.println("Please move forward "+distance);
   }
   void visitTimesNode(int count, StatementTree body) {
      for (int i=0; i<count; ++i) {
         body.Accept(this);
      }
   }
   void visitBeginNode(StatementTreeList body) {
      body.Accept(this);
   }
}
```

- Now we can encapsulate all the interpreter code in a single file
- And we can write a “print” traversal in a similar, single file

Appendix B: Syntax-directed translation and attribute grammars

- The structure of our translator is derived *systematically* from the AST data type—which in turn is derived from the language's grammar. Thus translation is "syntax-directed".

- In fact some textbooks (eg EaC and The Dragon book) make this link explicit -
  - *attribute grammars* express syntax-directed translation directly in terms of the grammar
  - we use Haskell to traverse the AST. The principle is the same
  - In Java a common approach is to use a Visitor pattern, see example at the end of these notes
- (Using attribute grammars leads to interesting possibilities for automatically-generating the syntax-directed translator)

Ad-hoc syntax-directed translation

- Attribute grammars are a neat theory (see Appendix B of these notes)
  - For example, supports *incremental* calculation of attributes, so you can update them when small changes are made to the tree
  - Lots of academic researchers have developed compiler-construction tools based on attribute grammars

- In most cases it's just as easy to build your own syntax-directed translator directly (see EaC section 4.4)
- Especially if you use a nice functional language like Haskell… (if you want to see how it's done in Java see Appendix A).

Attribute grammars: example

Example grammar

Number	→	Sign List
Sign	→	+
	|	-
List	→	List Bit
	|	Bit
Bit	→	0
	|	1

This grammar describes signed binary numbers

We would like to augment it with rules that compute the decimal value of each valid input string

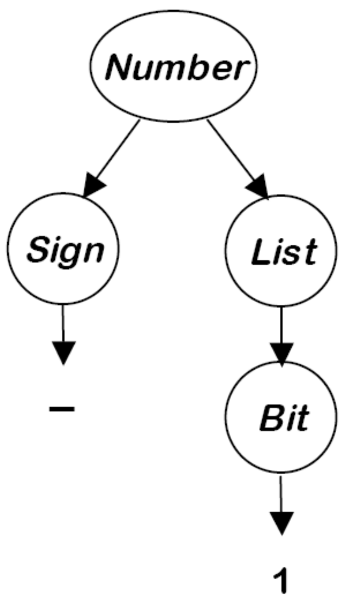
- Attribute grammars are a formal technique for specifying syntax-directed computation
- Invented by Knuth in 1968 – see EaC Section 4.3
- A kind of functional programming…

Example attributed to Scott K Warren

Numbers represented in our example grammar

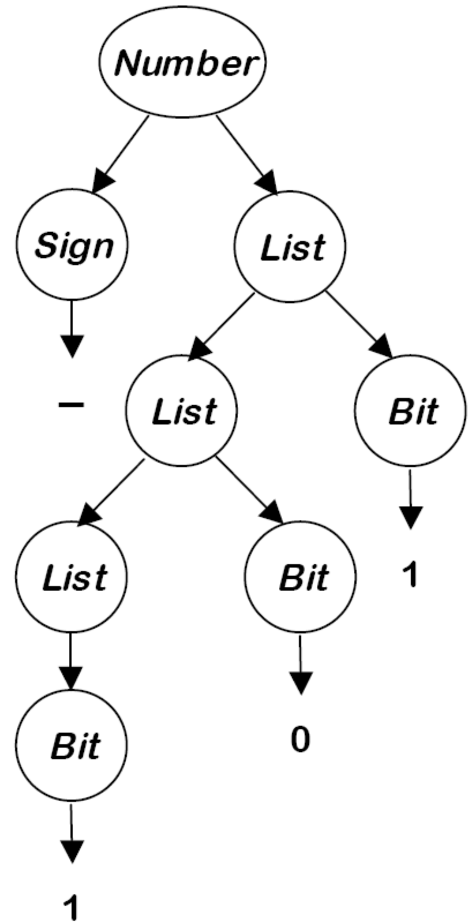
For "–1"

Number → *Sign List*
→ – *List*
→ – *Bit*
→ – *1*



For "–101"

Number → *Sign List*
→ *Sign List Bit*
→ *Sign List 1*
→ *Sign List Bit 1*
→ *Sign List 1 1*
→ *Sign Bit 0 1*
→ *Sign 1 0 1*
→ – *101*



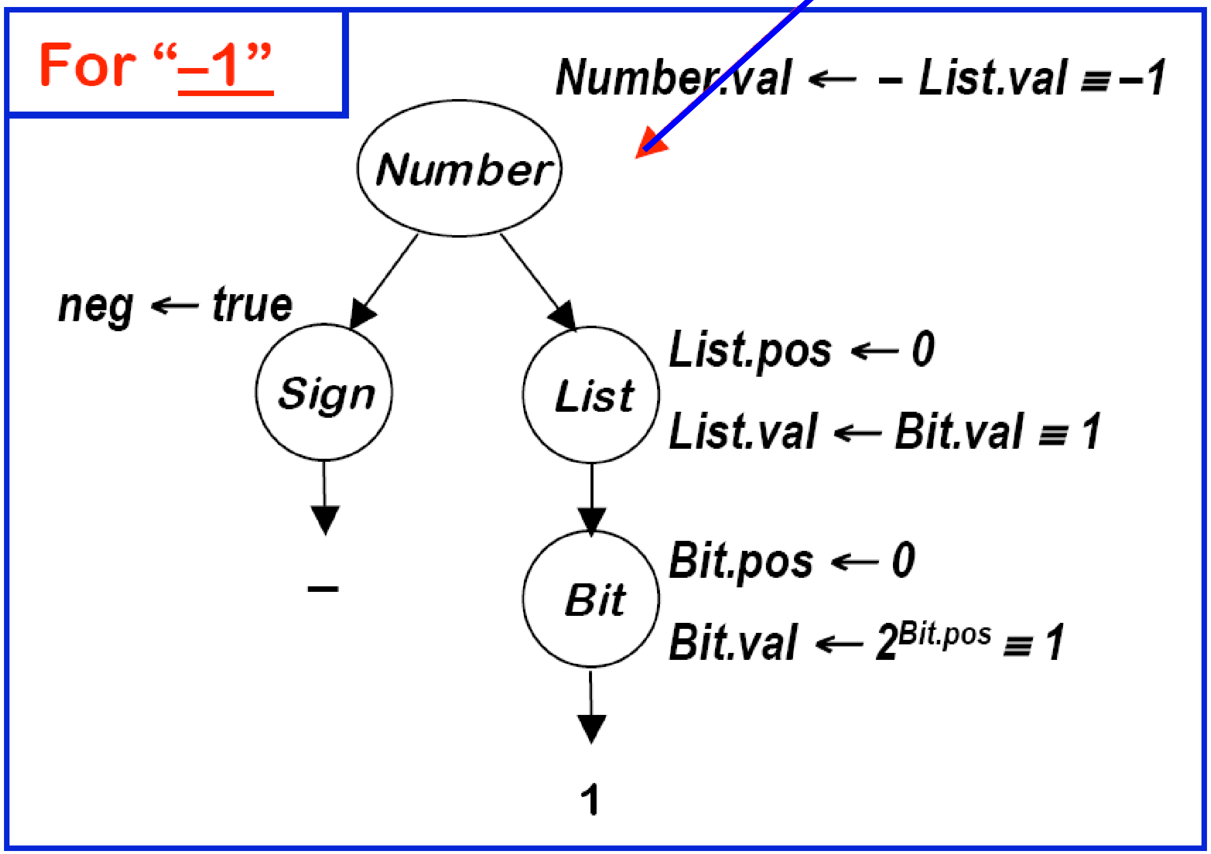
Extending the grammar with attributes

Productions			Attribution Rules
Number	→	*Sign List*	*List.pos* ← *0* *If Sign.neg* *then Number.val* ← – *List.val* *else Number.val* ← *List.val*
Sign	→	+	*Sign.neg* ← *false*
	|	–	*Sign.neg* ← *true*
$List_0$	→	$List_1$ *Bit*	$List_1.pos \leftarrow List_0.pos + 1$ $Bit.pos \leftarrow List_0.pos$ $List_0.val \leftarrow List_1.val + Bit.val$
	|	*Bit*	*Bit.pos* ← *List.pos* *List.val* ← *Bit.val*
Bit	→	0	*Bit.val* ← *0*
	|	1	$Bit.val \leftarrow 2^{Bit.pos}$

Symbol	Attributes
Number	val
Sign	neg
List	pos, val
Bit	pos, val

- Each non-terminal carries attributes
- Each production of the grammar is extended with rules
- The rules specify how the attributes are calculated

- Parse tree, combined with attribute rules, define functional program to calculate all the attribute values



One possible evaluation order:

1. List.pos
2. Sign.neg
3. Bit.pos
4. Bit.val
5. List.val
6. Number.val

Other orders are possible

- Evaluation order must be consistent with attribute dependence graph

(Example from Ken Kennedy's EaC-based course notes)

Feeding curiosity…

- We’re in the business of writing programs that generate programs. Can you write a program that prints out its own source code? In how many different ways? See “**Some alternative reproductive strategies in artificial molecular machines**”, Richard Laing, Journal of Theoretical Biology, 1975.
- What does a compiler look like if it’s written by someone who’s never seen a compiler textbook, nor taken this course? See “**The FORTRAN automatic coding system**”, John Backus et al, IRE-AIEE-ACM ‘57.
- Suppose your language had expressions, conditionals, functions and recursion, but no other control constructs. Can you define statements, blocks, loops, goto, exceptions, coroutines, threads, backtracking, lazy evaluation etc – *in the language*? See “**Lambda: The Ultimate Imperative**”, Guy Steele and Gerald Jay Sussman, MIT AI Memo 353, 1976.
- One page 7, I defined a processor microarchitecture in pseudocode. Using a hardware description language like Verilog or Chisel you can do this for real – see, for example https://github.com/ucb-bar/chisel-tutorial/blob/release/src/main/scala/examples/Risc.scala (one page of code).
- Verilog and Chisel need compilers too…. See https://llhd.io/