Source code → … → Semantic analysis → IR → And then a miracle happens → Assembly

(notes based on A.W. Appel's *Modern Compiler Implementation in Java*)

IR is the input for the back-end of the compiler. The output is, of course, machine language code.

Canonical trees

For the purposes of code optimization, the compiler should be able to evaluate parts of the IR tree in any order it sees fit.

Issues:

- subexpressions (parts of the IR tree) may have side effects
- different order of evaluating functions can have different end effects

Solution:

- an IR tree is rewritten as (broken down into) an equivalent list of canonical trees

Canonical tree

- tree where any subtree can be evaluated in any order (informal definition)

Rewriting (transformation) consists of:

- subexpression extraction
- subexpression insertion
- subexpression commutation, if possible: $(A + B) + C = A + (B + C)$
- moving procedure calls to the root of the canonical tree

A list of canonical trees can be linearized (rewritten as a sequence of instructions).

However, this is only the beginning…

Instruction selection

A specific fragment of the IR tree (tree pattern) can be expressed as an assembly instruction.

For example:

```
A ← B + C
```

High-level Language

```
        :=
      /    \
     A      +
          /   \
         B     C
```

Intermediate Represenation

```
ADD $r1, $r2, $r3
```

MIPS64 Assembly

Algorithms for instruction selection:

- **MAXIMAL MUNCH**
  (local optimization starting from the root of the tree and proceeding to the subtrees)

- **DYNAMIC PROGRAMMING**
  (global optimization, finds a solution based on optimum solutions of the subproblems)

By the way, two ISA's (instruction set architectures) are popular today, very different in their philosophies:

RISC (Reduced Instruction Set Computer)	CISC (Complex Instruction Set Computer)
32 registers	6, 8 or 16 registers
all registers are created equal	specialized registers
arithmetic only between registers	arithmetic can access memory
3-address instructions	2-address instructions
register + constant offset addressing mode	several different addressing modes
fixed length instructions (32-bit)	variable length instructions
no side effects	instructions can have side effects

However, not everything translates so well…

Basic blocks and traces

Issue:
IR branching does not translate well into Assembly branching

```
if CONDITION then
    BLOCK 1
else
    BLOCK 2
```

High-level Language

```
            BNE $r1, $r2, label_1
            BLOCK 2
label_1:    BLOCK 1
```

MIPS64 Assembly

Solution:

- transform list of canonical trees into a series of basic blocks (**PRIMA PARS**)
- order basic blocks into a trace (**PARS SECUNDA**)

Basic block

- sequence of statements always entered at the beginning and exited at the end (i.e. there is not way to jump into the middle of a basic block, nor to jump out of it before the end of the block has been reached)

PRIMA PARS: how to form basic blocks?

- analyze the control flow
- lump together any sequence of non-branching instructions

Control flow

- sequence of instructions in a program

PARS SECUNDA: rearrange basic blocks in such a way that:

- for all conditional branches, false block immediately follows the branching instruction
- unconditional jumps are followed by their target labels (so they can be eliminated)

Thus a trace is formed.

Trace

- sequence of statements that could be consecutively executed in a program execution

The set of traces covers the program (each basic block is in an exactly one trace, all basic blocks are covered).

Liveness analysis

Getting closer to hardware…

Issue:
Code generated so far does not worry about the resources, which, as it happens, are scarce. Specifically, the number of registers in a processor is 32 or less.

Solution:
If variables *a* and *b* are not used at the same time, we can fit them in the same register at different times. Analysis of variables being "used at the same time" is called **LIVENESS ANALYSIS**.

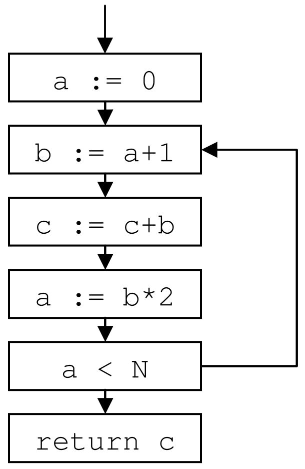
Live (variable)

- a variable is said to be live if it holds a value that may be needed in the future

Example program:

```
1                 a := 0
2     Label:      b := a + 1
3                 c := c + b
4                 a := b * 2
5                 if (a < N) goto Label
6                 return c
```

The control flow graph of this program would look like this:



Intro to flow-graph terminology: **OUT-EDGES** lead to **SUCCESSOR** nodes, **IN-EDGES** come from **PREDECESSOR** nodes. An assignment to a variable **DEFINES** the variable, occurrence of a variable on the right-hand side **USES** the variable. So, *pred*(2) = {1, 5}, *def*(3) = {*c*}, *use*(3) = {*b, c*}

Liveness

- a variable is **LIVE** on an edge if there is a directed path from that edge to a **USE** of the variable that does not go through any **DEF**.

Dataflow equations

Register allocation

Graph coloring algorithms

Coloring by simplification

Coalescing