Representation of programs

Primary goals:

- analysis is easy & effective
- transformations are easy
- just a few cases to handle
- general, across input languages & target machines

Additional goals:

- compact in memory
- easy to generate to and from
- tracks info for source-level debugging, profiling, etc.
- extensible (new optimizations, targets, language features)
- displayable

High-level syntax-based representation

Represent source-level control structures & expressions directly

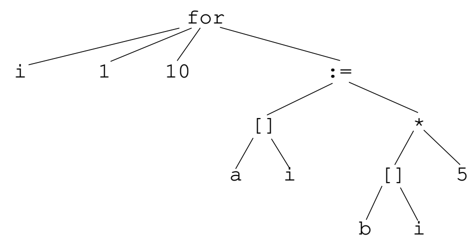
Examples

- (Attributed) AST
- Lisp S-expressions

Source:

```
for i := 1 to 10 do
  a[i] := b[i] * 5;
end
```

AST:



Pros/cons

+ analysis exploits high-level knowledge of constructs
+ easy to map to source code terms for debugging, profiling
+ supports semantics-based reasoning about analysis, correctness, etc.

– some transformations hard to express
– low-level optimizations hard/impossible
– can be many kinds of AST nodes to analyze
– limited to particular input language(s)

Suitable for a source-to-source optimizer, parallelizer

- rely on another compiler to do low-level optimizations

Low-level representation

Translate input programs into low-level primitive chunks, often close to the target machine

Examples

- assembly code, virtual machine code (e.g. stack machine)
- three address code, register transfer language (rtl's)
- lambda calculus

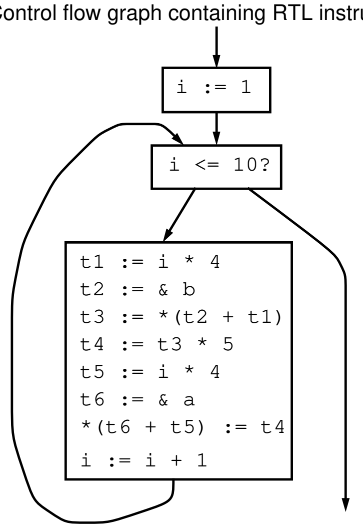
Standard RTL operators:

assignment	`x := y;`
unary op	x := *op* y;
binary op	x := y *op* z;
address-of	`p := &y;`
load	`x := *(p + o);`
store	`*(p + o) := x;`
call	`x := f(...);`
unary compare	*op* x ?
binary compare	x *op* y ?

Source:

```
for i := 1 to 10 do
  a[i] := b[i] * 5;
end
```

Control flow graph containing RTL instructions:



Pros/cons

+ can do low-level, machine-specific optimizations
  (if target-based representation)
+ supports good analysis (?), transformations
+ can have relatively few kinds of instructions to analyze
+ can be language-independent

– may have to try to recover high-level knowledge
– hard to track source for debugging, profiling, etc.
– can be more space-, time-consuming

Idea: use multiple representations at different levels

Components of representation

Operations

Control flow: sequencing of operations
- evaluation of then & else arms depends on result of test
- side-effects of statements occur in right order

Data flow: flow of values from **definitions** to **uses**
- operands computed before operation

Also called control & data **dependence**

Ideal: represent just those dependences that matter
- dependences constrain transformations
- fewest dependences ⇒ most flexibility in implementation

Representing control dependences

Option 1: high-level representation
- control flow implicit in semantics of AST nodes

Option 2: control flow graph
- nodes are basic blocks
- edges represent branches
  (control flow between basic blocks)

Fancier:
- control dependence tree,
  part of **program dependence graph** (PDG)
- convert into data dependences on explicit state,
  in **value dependence graph** [Weise *et al. 94*]
- ...

Kinds of data dependences

read-after-write (RAW): **true dependence**, **flow dependence**

- reflects real data flow, operands to operation

write-after-read (WAR): **anti-dependence**
write-after-write (WAW): **output dependence**

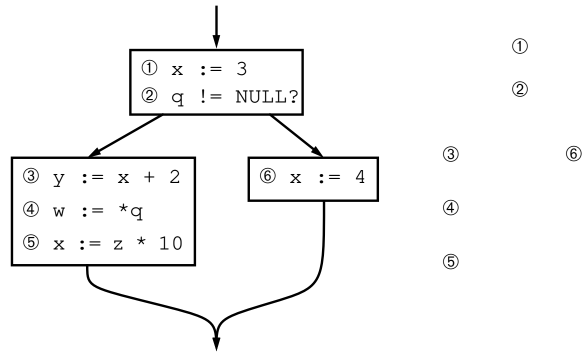
- reflects reuse of memory, not real data flow
  ⇒ can sometimes be eliminated by optimization

read-after-read (RAR): no dependence

- can occur in any order

Example

```
x := 3
if q != NULL then
  y := x + 2
  w := *q
  x := z * 10
else
  x := 4
endif
```



Representing data dependences

Within basic block:

Option 1: sequence of instructions
(represent data flow as a kind of control dependence)

+ simple, source-like
+ fixed ordering supports easy analysis
– may overconstrain order of operations

Option 2: expression tree/DAG

+ natural, abstract
+ directly captures data dependences within basic block
+ DAG supports local CSE
+ can be compact
– conceptually harder to analyze, transform
– must linearize eventually

Representing data dependences, cont.

Across basic blocks:

Option 1: implicitly through variable defs/uses

+ simple
– analysis wants important things explicit

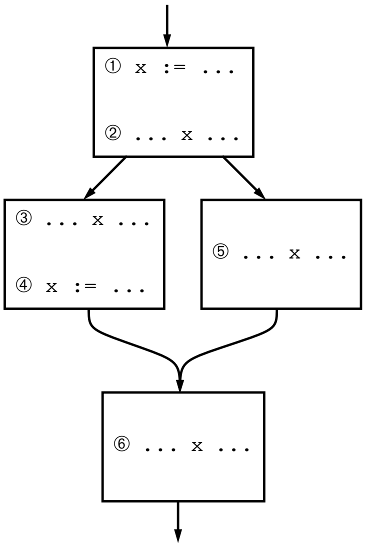
Option 2: def/use chains, linking each def with each use

+ explicit
– must be computed, maintained after transformations
– space consuming

Fancier def/use representations:

- **static single assignment** (SSA) form
- **dependence flow graphs** (DFGs)
- value dependence graphs
- ...

Example



①

②

③

⑤

④

⑥