

Representing types [33]

The lineage of a type can be represented without reference to specific type names. A bit-vector representation is convenient for construction and for comparison:

Each of the types extenders is assigned a bit pattern from k bits:

Type	Pattern
ptr	01
array	10
func	11

So that a pointer to type X is represented as

01 X

Note that this scheme does not track array index types or function parameter types.

Wisely leaving one pattern free, we now assign the base types:

Type	Pattern
void	0000
char	0001
int	0010
float	0011

Declaration	Type representation
int x	0010
int **x[]	01 01 10 0010
char ((*x())[])()	11 01 10 01 11 0001

The last entry is a function that returns a pointer to an array of pointers to functions that return characters.

Copyright ©1994 Ron K. Cytron. All rights reserved

- 109 -

SIGPLAN '94 COMPILER CONSTRUCTION TUTORIAL

Runtime storage management

Most block-structured languages require manipulation of runtime structures to maintain efficient access to appropriate data and machine resources. For our purposes, a procedure is either a named function or an inline (parameterless) block.

Let's examine the activity normally associated with invoking a procedure P :

1. Some machine state might be saved: general registers, vector registers, condition codes, interrupt masks, etc.
2. Access must be established to P 's local variables and compiler-generated temporaries.
3. Access must be established to outer scope variables (but not for C).
4. The caller of P must be recorded so that P can return when done.
5. Parameters might be received prior to executing P .
6. A return value might be prepared prior to returning from P .

Each procedure invocation causes creation of an *activation record* or *frame* to hold such runtime information.

State
Local x
Local y
⋮
Dynamic Link
Static Link
Parameters
Return Value

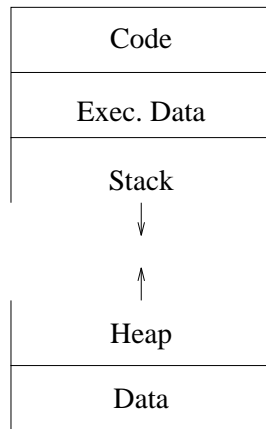
It's convenient to have each local occupy a fixed amount of storage in the frame. Therefore, arrays and other large objects are often indirectly accessed from a procedure's frame, with the actual storage allocated on stack after the frame.

Copyright ©1994 Ron K. Cytron. All rights reserved

- 110 -

SIGPLAN '94 COMPILER CONSTRUCTION TUTORIAL

A simple runtime storage layout



Since there are two dynamically growing areas, a simple scheme is to place these at opposite ends of the address space.

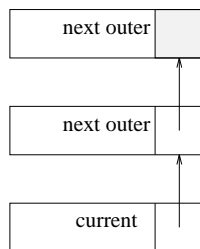
Following the contour of procedure entry and exit, activation records are usually allocated on a stack. Where languages allow suspension and resumption of procedures (*e.g.*, via *continuations*), then frames are garbage-collected from the heap when dead.

The stack can also be used for performing intermediate computations.

The heap is generally managed by some form of explicit or implicit garbage collection [4].

Access to nonlocals

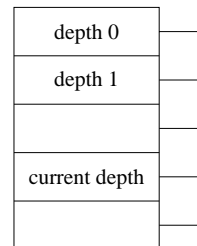
Static Links



At procedure entry, a link is inserted in the frame to a procedure's next outer scope, whose frame is linked to its outer scope, and so on. The static link is deallocated along with the frame.

Establishing the link is fast, but accessing the k th enclosing scope requires k indirections using static links. However, a good register allocator would cache these in the procedure's registers.

Displays



A display is an array of frame pointers. At procedure entry, the display is adjusted so that the frame at static depth d is accessed via entry d of the display. The display must be reset at procedure return.

Maintaining the display takes more time than with static links, but access to outer scopes is faster once the display is established.

Most programs make almost exclusive use of local and outermost scopes, with scant use of intermediate scopes. This is especially true in C, where the language offers no access to intermediate scopes except by explicit pointers.

Code Generation

As with parsing, methods for code generation can be classified:

Ad hoc.

As with semantic processing, code could be generated after the parse by traversing the AST. Typically, a combined pre- and post-order traversal suffices, where a node's type prompts the code generator to emit a parameterized template of code.

Systematic

- Grammar-based [21].
- Grammars with attributes [20].
- Tree pattern-matching [17, 19].
- By peephole processing [11].

These methods spend more time on instruction selection than can be afforded or managed by *ad hoc.* methods.

In a compiler course, the choice of code generation strategy is key to a successful experience. Many courses stop just before code generation, in which case the students do not experience the elation of watching their compilers actually work.

*If the target of translation is reasonably high-level (e.g., a LISP-like intermediate language), then *ad hoc.* methods are feasible. In this case, an interpreter should be provided to execute the translated programs.*

Otherwise, experience with an automatic code generator is more beneficial. Watch for developments in the `lcc` system [18], which can be obtained by contacting Dave Hanson (drh@princeton.edu). If the MIPS instruction set were targeted, then Larus's SPIM simulator [30](Appendix A) can greatly facilitate debugging the generated code.

Example of a high-level intermediate language

The language *FRIL* [10] was developed to ease code generation, primarily by resembling *LISP* and by offering a declarative mechanism for storage association. Each symbol *FRIL* is declared at most once as any procedure's local or parameter. Each "expression" declares the static depth of its frame, and provides a pointer to its outer scope.

```
int a1;
extern int a2;
int one;
void main() {
  int i;

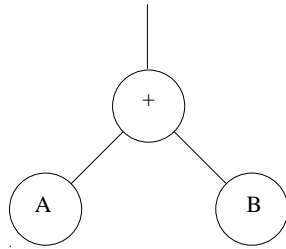
  int factorial(X)
  int X;
  {
    int Y;
    Y = X;
    if (Y > 0) Y*factorial(X-1);
    else one;
  }

  one = 1;
  a1 = factorial(i=5);
  a2 = factorial(3);
}
```

```
(Expression 1 /* factorial */
  (PushLevel 5 (LinkExpressionID 2)
    (Args (SymbolID 7) /* X */)
    (Locals (SymbolID 8) /* Y */)
  )
  (Def (SymbolID 8) /* Y */
    (Use (SymbolID 7) /* X */)
  )
  (-> 0
    (CHOOSE
      (
        (NE 0
          (GT (Use (SymbolID 8) /* Y */) 0)
        )
        (TIMES
          (Use (SymbolID 8) /* Y */)
          (-> 1
            (MINUS
              (Use (SymbolID 7) /* X */)
              1
            )
          )
        )
      )
    )
  )
  (1 (Use (SymbolID "one")))
)
```

Ad hoc. methods

For example, for a binary $+$ node, the code generator would be called recursively to place the result of the left and right subtrees in two known locations (say, registers R_1 and R_2). Code would then be emitted to form the sum, placing the result in yet another known location.



```
(PLUS
  /* Code for A */
  /* Code for B */
)
```

I usually provide procedures for generating FRIL's symbol table, for generating a PushLevel, and for indenting and formatting the output. The students must decide what constitutes an expression. For example, FRIL has only one control transfer operator: the procedure call. Thus, the body of an iterative loop must be invoked recursively to achieve iteration.

Students write some 200 lines of code to complete the *ad hoc.* code generator for FRIL.

A systematic method — Tree Rewriting

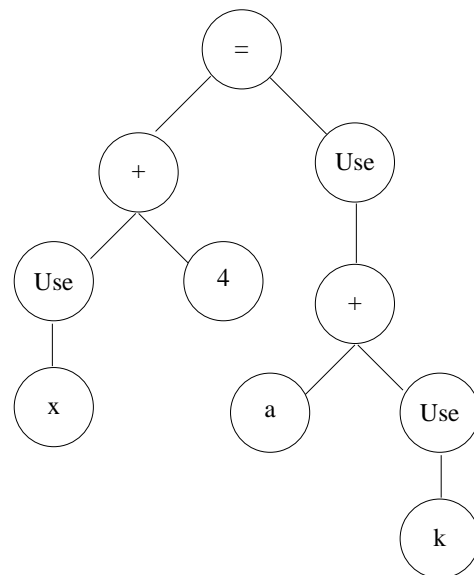
While the *ad hoc.* method services the AST a node at a time, tree rewriting systems can examine larger subtrees and searching for more optimal instruction sequences.

The AST shown to the right is representative of the code fragment

```
*(x+4)=a[k];
```

Note that left and right value analysis has already taken place.

Let's assume that from the perspective of code generation, the nodes x , a , and k represent constants. This would be the case had the compiler assigned storage to these variables. If not, then the AST should reflect a level of indirection (probably off a popular register) to reach those variables.



Given the richness of most instruction sets, trying all combinations of instructions to cover the tree would be prohibitively expensive. Most tree matching algorithms use *dynamic programming*, so that results previously holding for some subtree can be reused without additional cost.

Tree rewriting

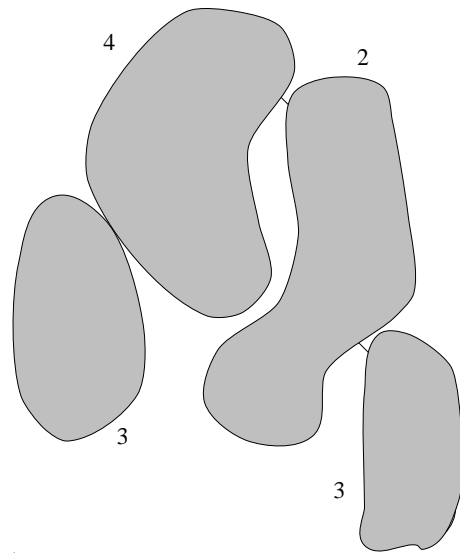
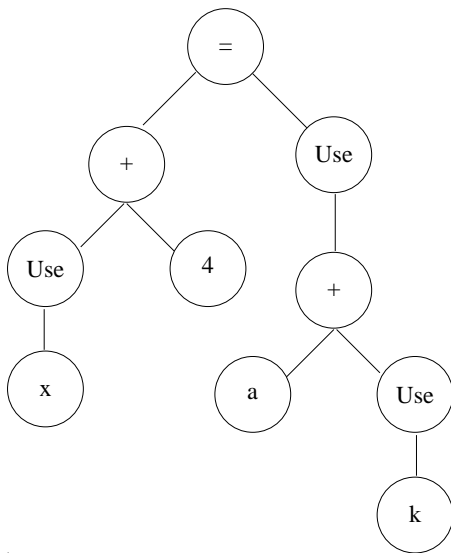
Rule	Rewrite	Instruction	Cost
1		$R_i \leftarrow R_i + \text{const}$	1
2		$R_i \leftarrow M[R_i + \text{const}]$	5
3		$R_i \leftarrow M[\text{const}]$	3

Not shown are the rules that account for the symmetry of addition.

Tree rewriting

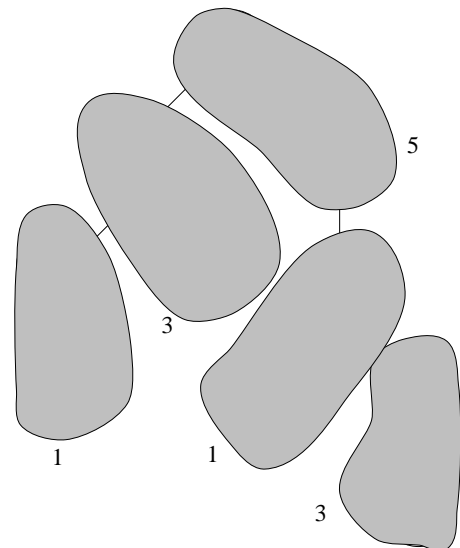
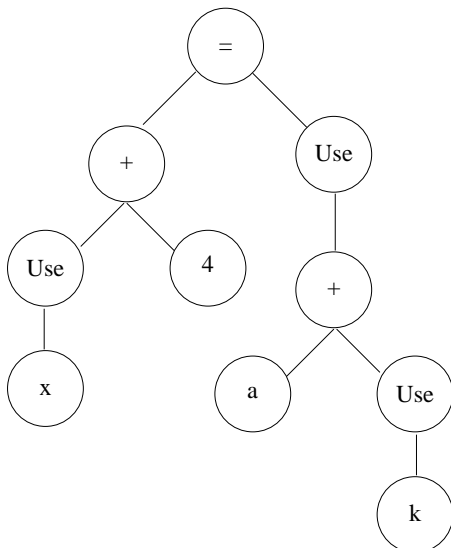
Rule	Rewrite	Instruction	Cost
4		$M[R_i + \text{const}] \leftarrow R_j$	5
5		$M[R_i] \leftarrow M[R_j]$	6
6		$R_i \leftarrow \text{const}$	1
7		$R_i \leftarrow R_i + R_j$	1

Example – one way to cover the nodes



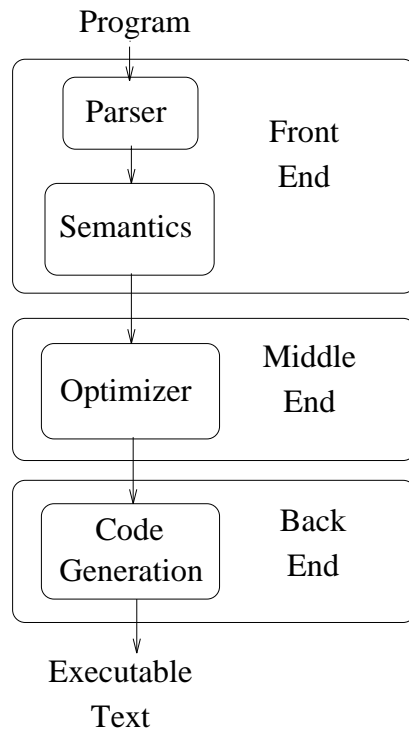
Rule	Instr	Cost
3	$R_i \rightarrow M[R_i + \text{const}]$	5
2	$R_i \rightarrow M[\text{const}]$	3
3	$R_i \rightarrow M[\text{const}]$	3
4	$M[R_i + \text{const}] \rightarrow R_j$	5
		16

Example – another way to cover the nodes



Rule	Instr	Cost
3	$R_i \rightarrow M[R_i + \text{const}]$	3
1	$R_i \rightarrow R_i + \text{const}$	1
3	$R_i \rightarrow M[\text{const}]$	3
1	$R_i \rightarrow R_i + \text{const}$	1
5	$M[R_i] \rightarrow M[R_j]$	6
		14

Compiler organizations



Front end: Operator and storage abstractions, alias mechanisms.

Middle end:

- Dead code elimination
- Code motion
- Reduction in strength
- Constant propagation
- Common subexpression elimination
- Fission
- Fusion
- Strip mining
- Jamming
- Splitting
- Collapsing

Back end: Finite resource issues and code generation.

Some thoughts

Misconceptions

Optimization optimizes your program.

There's probably a better algorithm or sequence of program transformations. While optimization hopefully improves your program, the result is usually not optimal.

Optimization requires (much) more compilation time.

For example, dead code elimination can reduce the size of program text such that overall compile time is also reduced.

A clever programmer is a good substitute for an optimizing compiler.

While efficient coding of an algorithm is essential, programs should not be obfuscated by "tricks" that are architecture- (and sometimes compiler-) specific.

All too often...

Optimization is disabled by default.

Debugging optimized code can be treacherous [45, 23]. Optimization is often the primary suspect of program misbehavior—sometimes deservedly so. "No, not the third switch!"

Optimization is slow.

Transformations are often applied to too much of a program. Optimizations are often textbook recipes, applied without proper thought.

Optimization produces incorrect code.

*Although recent work is encouraging [42], optimizations are usually developed *ad hoc*.*

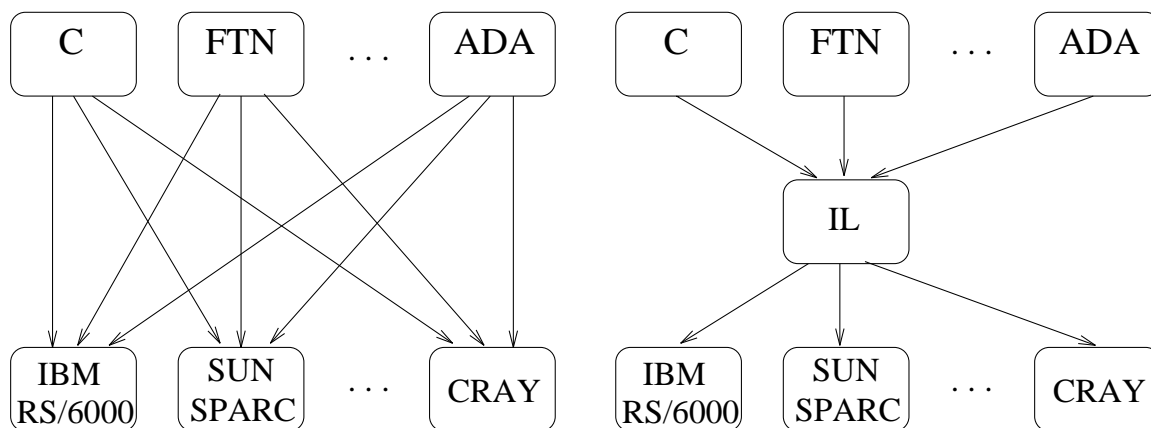
Programmers are trained by their compilers.

A style is inevitably developed that is conducive to optimization.

Optimization is like sex:

- *Everybody claims to get good results using exotic techniques;*
- *Nobody is willing to provide the details.*

Multilingual systems



Architecting an *intermediate language* reduces the incremental cost of accommodating new source languages or target architectures [5]. Moreover, many optimizations can be performed directly on the intermediate language text, so that source- and machine-independent optimizations can be performed by a common middle-end.

Intermediate languages

It's very easy to devote much time and effort toward choosing the "right" IL. Below are some guidelines for choosing or developing a useful intermediate language:

- The IL should be a *bona fide* language, and not just an aggregation of data structures.
- The semantics of the IL should be cleanly defined and readily apparent.
- The IL's representation should not be overly verbose:
 - Although some expansion is inevitable, the IL-to-source token ratio should be as low as possible.
 - It's desirable for the IL to have a verbose, human-readable form.
- The IL should be easily and cleanly extensible.
- The IL should be sufficiently general to represent the important aspects of multiple front-end languages.
- The IL should be sufficiently general to support efficient code generation for multiple back-end targets.

A sampling of difficult issues:

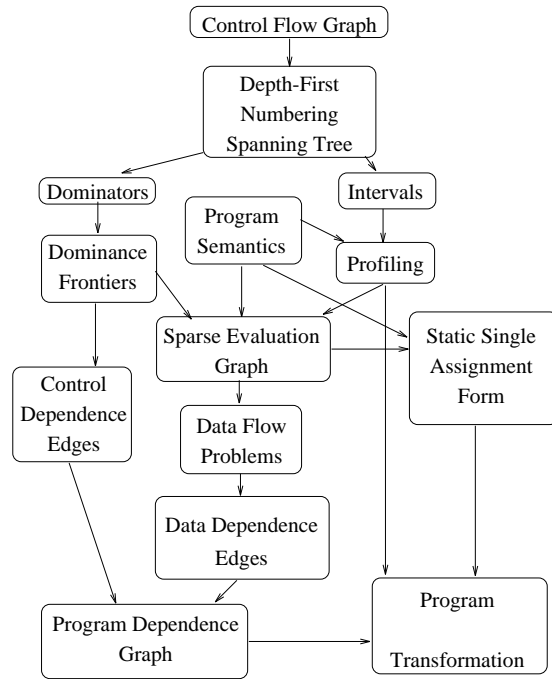
- How should a string operation be represented (intact or as a "loop")?
- How much detail of a procedure's behavior is relevant?

Ideally, an IL has *fractal* characteristics: optimization can proceed at a given level; the IL can be "lowered"; optimization is then applied to the freshly exposed description.

What happens in the middle end?

Essentially, the program is transformed into an observably equivalent while less resource-consuming program. Such transformation is often based on:

- Assertions provided by the program author or benefactor.
- The program dependence graph [29, 15, 6].
- Static single assignment (SSA) form [8, 3, 44, 9].
- Static information gathered by solving data flow problems [25, 34, 35, 36, 22, 37, 38, 27].
- Run-time information collected by *profiling* [40].



Let's take a look at an example that benefits greatly from optimization...

Unoptimized matrix multiply

```

for i = 1 to N do
  for j = 1 to N do
    A[i, j] ← 0
    for k = 1 to N do
      A[i, j] ← A[i, j] + B[i, k] × C[k, j]
    od
  od
od

```

Note that $A[i, j]$ is really

$$\text{Addr}(A) + ((i - 1) \times K_1 + (j - 1)) \times K_2$$

which takes 6 integer operations.

The innermost loop of this "textbook" program takes

24 integer ops
3 loads
1 floating add
1 floating mpy
1 store
30 instructions

Optimizing matrix multiply

```
for  $i = 1$  to  $N$  do
  for  $j = 1$  to  $N$  do
     $a \leftarrow \&(A[i, j])$ 
    for  $k = 1$  to  $N$  do
       $\star a \leftarrow \star a + B[i, k] \times C[k, j]$ 
    od
  od
od
```

```
for  $i = 1$  to  $N$  do
   $b \leftarrow \&(B[i, 1])$ 
  for  $j = 1$  to  $N$  do
     $a \leftarrow \&(A[i, j])$ 
    for  $k = 1$  to  $N$  do
       $\star a \leftarrow \star a + \star b \times C[k, j]$ 
       $b \leftarrow b + K_B$ 
    od
  od
od
```

The expression $A[i, j]$ is *loop-invariant* with respect to the k loop. Thus, *code motion* can move the address arithmetic for $A[i, j]$ out of the innermost loop.

The resulting innermost loop contains only 12 integer operations.

As loop k iterates, addressing arithmetic for B changes from $B[i, k]$ to $B[i, k + 1]$. *Induction variable analysis* detects the constant difference between these expressions.

The resulting innermost loop contains only 7 integer operations.

Similar analysis for C yields only 2 integer operations in the innermost loop, for a speedup of nearly 5. We can do better, especially for large arrays.

Copyright © 1994 Ron K. Cytron. All rights reserved

- 127 -

SIGPLAN '94 COMPILER CONSTRUCTION TUTORIAL

If optimization is...

so great because:

A good compiler can sell (even a slow) machine. *Optimizing compilers easily provide a factor of two in performance. Moreover, the analysis performed during program optimization can be incorporated into the "programming environment" [29, 7, 43].*

New languages and architectures motivate new program optimizations. *Although some optimizations are almost universally beneficial, the advent of functional and parallel programming languages has increased the intensity of research into program analysis and transformation.*

Programs can be written with attention to clarity, rather than performance. *There is no substitute for a good algorithm. However, the expression of an algorithm should be as independent as possible of any specific architecture.*

then:

Why does it take so long? *Compilation time is usually 2–5 times slower, and programs with large procedures often take longer. Often this is the result of poor engineering: better data structures or algorithms can help in the optimizer.*

Why does the resulting program sometimes exhibit unexpected behavior? *Sometimes the source program is at fault, and a bug is uncovered when the optimized code is executed; sometimes the optimizing compiler is itself to blame.*

Why is "no-opt" the default? *Most compilations occur during the software development cycle. Unfortunately, most debuggers cannot provide useful information when the program has been optimized [45, 23]. Even more unfortunately, optimizing compilers sometimes produce incorrect code. Often, insufficient time is spent testing the optimizer, and with no-opt the default, bugs in the optimizer may remain hidden.*

Ingredients in a data flow framework

Data flow graph

$$\mathcal{G}_{df} = (\mathcal{N}_{df}, \mathcal{E}_{df})$$

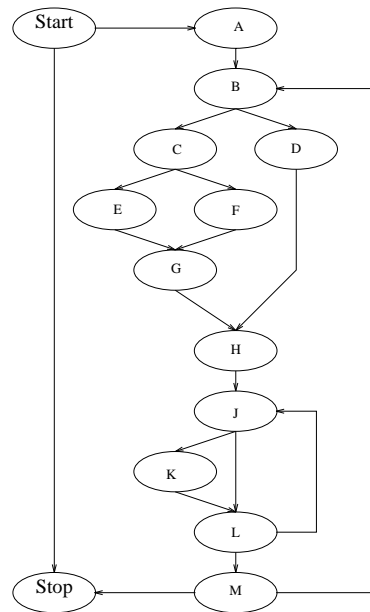
which is based on a directed *flow graph* $\mathcal{G}_f = (\mathcal{N}_f, \mathcal{E}_f)$, typically the *control flow graph* of a procedure.

A data flow problem is

forward if the solution at a node may depend only on the program's past behavior;

backward if the solution at a node may depend only on a program's future behavior;

bidirectional if both past and future behavior is relevant [12, 13, 14].



- We'll assume the data flow graph is augmented with a *Start* and *Stop* node, and an edge from *Start* to *Stop*.
- We'll limit our discussion to non-bidirectional problems, and assume that edges in the data flow graph are oriented in the direction of the data flow problem.

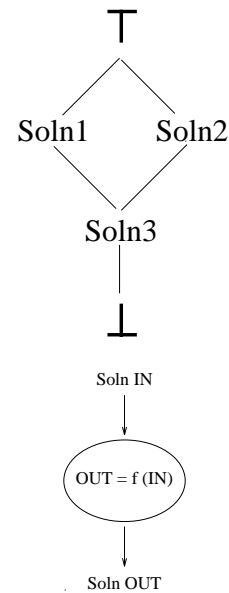
Ingredients in a data flow framework (cont'd)

Meet lattice which determines the outcome when disparate solutions combine. The lattice is specified with distinguished elements

\top which represents the best possible solution, and

\perp which represents the worst possible solution.

Transfer Functions which transform one solution into another.



We'll use the meet lattice to summarize the effects of convergent paths in the data flow graph, and transfer functions to model the effects of a data flow graph path on the data flow solution.

We'll begin with some simple *bit-vectoring* data flow problems, classically solved as operations on bit-vectors. For ease of exposition, we'll associate data flow solutions with the edges, rather than the nodes, of the data flow graph.

Available expressions

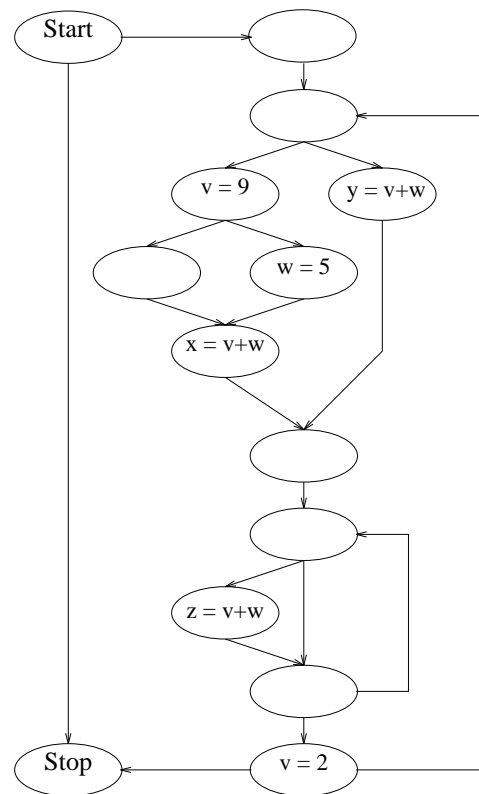
An expression $expr$ is *available* (*Avail*) at flow graph edge e if any past behavior of the program includes a computation of the value of $expr$ at e .

Consider the expression $(v + w)$ in the flow graph shown to the right. If the expression is available at the assignment to z , then it need not be recomputed.

- This is a forward problem, so the data flow graph will have the same edges and *Start* and *Stop* nodes as the flow graph.
- The solution for any given $expr$ is either *Avail* or \overline{Avail} .
- The "best" solution for an expression is *Avail*. We thus obtain the two-level lattice:

\top is *Avail*.

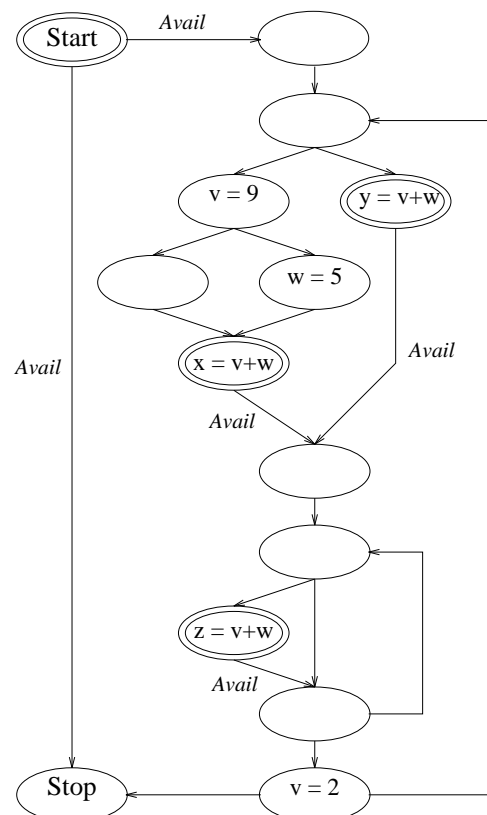
\perp is \overline{Avail} .



Available expressions(cont'd)

Nodes that compute an expression make that expression available. We also assume that every expression is available from *Start*.

The transfer function for each highlighted node makes the expression $(v + w)$ *Avail*, regardless of the solution present at the node's input.



Live variables (cont'd)

If a node Y preserves v (as might a procedure call), then the node does not affect the solution.

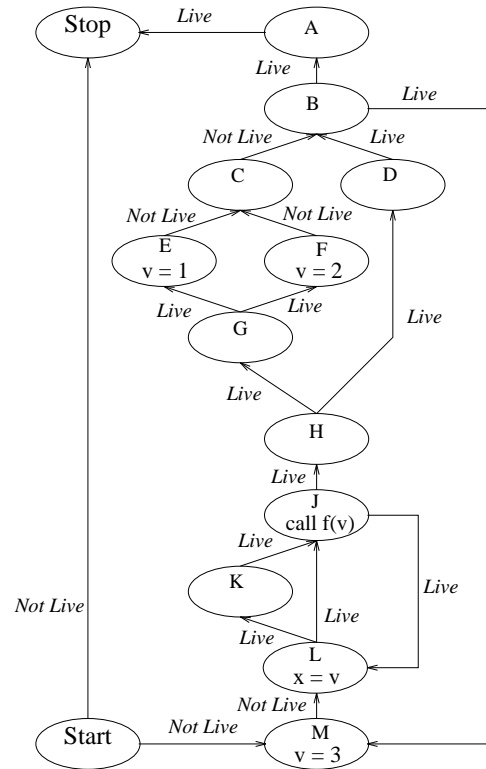
- If v is *Live* on "input" to Y , then Y cannot make v *Not Live*.
- If v is *Not Live* on "input" to Y , then Y does not make v *Live*.

Node Y 's transfer function is therefore the *identity* function:

$$f_Y(IN) = IN$$

assuming node Y does not use v .

Global solution: Live variables



Formal specification of a data flow framework

The *data flow graph*

$$\mathcal{G}_{df} = (\mathcal{N}_{df}, \mathcal{E}_{df})$$

has been described previously:

- its edges are oriented in the direction of the data flow problem;
- \mathcal{G}_{df} is augmented with nodes *Start* and *Stop* and an edge $(Start, Stop)$, suitably inserted with respect to the direction of the data flow problem.

Successors and predecessors are also defined with respect to the direction of the data flow problem:

$$Succs(Y) = \{ Z \mid (Y, Z) \in \mathcal{E}_{df} \}$$

$$Preds(Y) = \{ X \mid (X, Y) \in \mathcal{E}_{df} \}$$

The *meet semilattice* is

$$L = (A, \top, \perp, \preceq, \wedge)$$

A is a set (usually a powerset), whose elements form the domain of the data flow problem,

\top and \perp are distinguished elements of A , usually called "top" and "bottom", respectively,

\preceq is a reflexive partial order, and

\wedge is the associative and commutative *meet* operator, such that for any $a, b \in A$,

$$a \preceq b \iff a \wedge b = a$$

$$a \wedge a = a$$

$$a \wedge b \preceq a$$

$$a \wedge b \preceq b$$

$$a \wedge \top = a$$

$$a \wedge \perp = \perp$$

These rules allow formal reasoning about \top and \perp in a framework.

Formal specification (cont'd)

The set \mathcal{F} of *transfer functions*

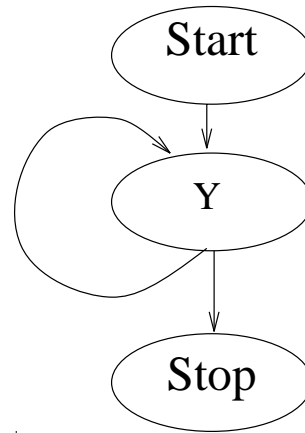
$$\mathcal{F} \subseteq \{f : L \mapsto L\}$$

has elements for describing the behavior of any flow graph node with respect to the data flow problem.

To obtain a stable solution, we'll require the functions in \mathcal{F} to be *monotone*:

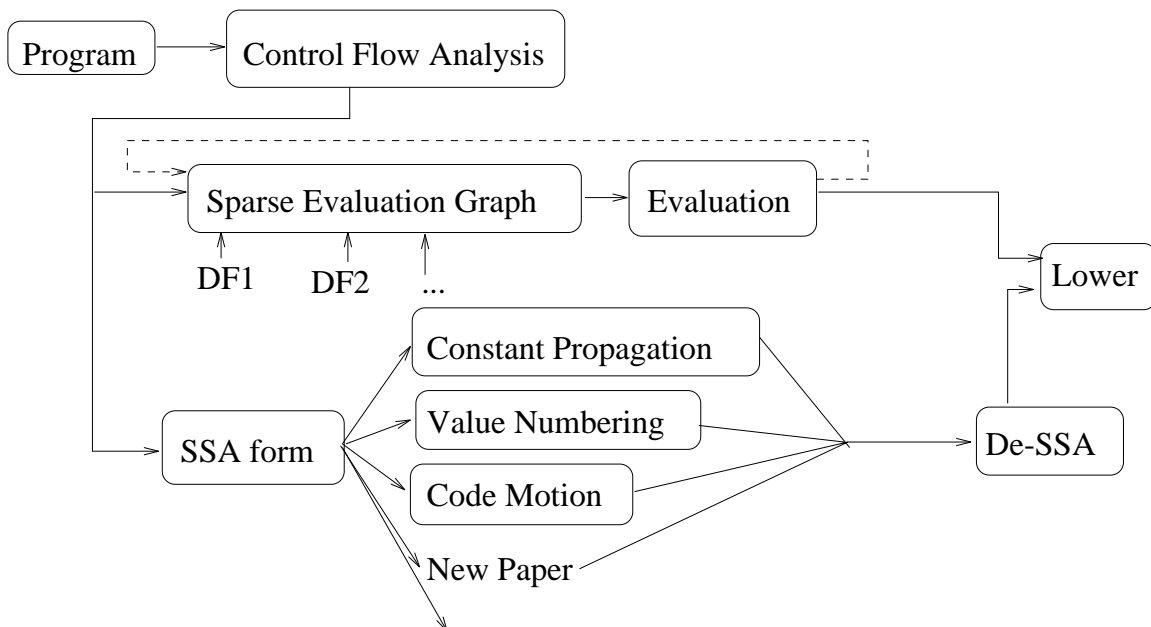
$$(\forall f \in \mathcal{F})(\forall x, y \in L) \\ x \preceq y \rightarrow f(x) \preceq f(y)$$

In other words, a node cannot produce a "better" solution when given "worse" input. Given a two-level lattice, evaluation of the data flow graph shown to the right oscillates between solutions and never reaches a fixed point.



$$f_Y(IN) = \begin{cases} \top & \text{if } IN = \perp \\ \perp & \text{if } IN = \top \end{cases}$$

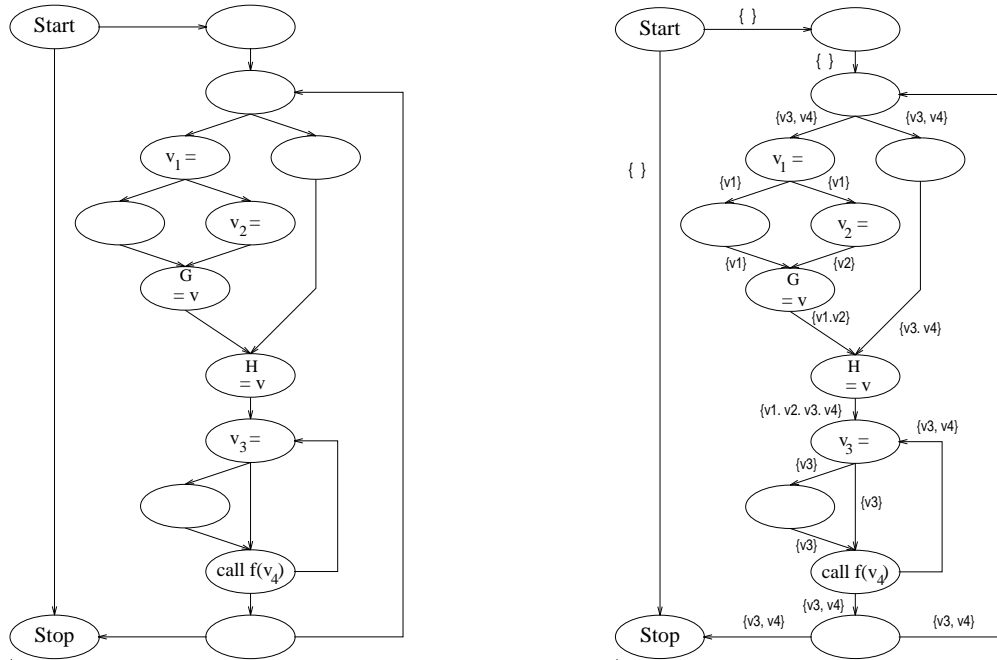
Big picture



We'll now examine some special algorithms for optimization, based on a single assignment representation.

Static Single Assignment (SSA) form

Below are shown a program and its reaching definitions.

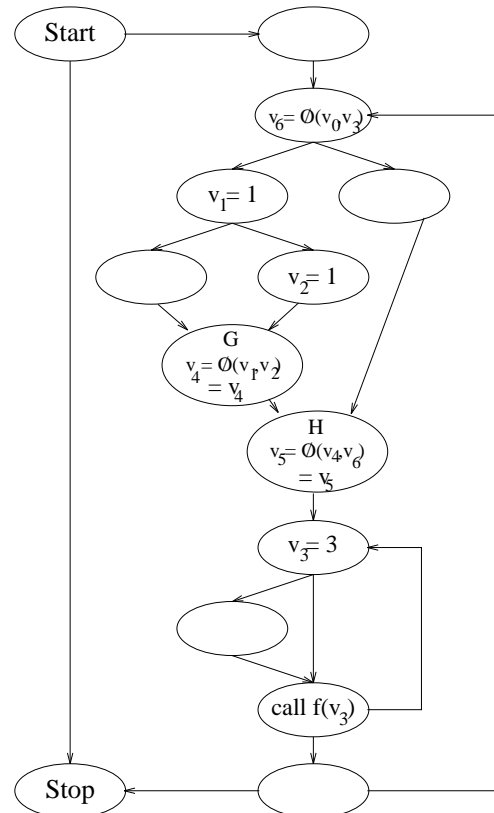


Notice how the use of v at G is reached by two definitions, and the use at H is reached by four definitions. If each use were reached by just a single definition, data flow analysis based on definitions could consult one definition per use.

SSA form (cont'd)

Here we see the SSA form of the program.

- Each definition of v is with respect to a distinct symbol: v_1 is as different from v_2 as x would be from y .
- Where multiple definitions reach a node, a ϕ -function is inserted, with arguments sufficient to receive a different "name" for v on each in-edge.
- Each use is appropriately renamed to the distinct definition that reaches it.
- Although ϕ -functions could have been placed at every node, the program shown has exactly the right number and placement of ϕ -functions to combine multiple defs from the original program.
- Our example assumes that procedure f does not modify v .



SSA form (cont'd)

Each def is now regarded as a “killing” def, even those usually regarded as preserving defs. For example, if v is *potentially* modified by the call site, then the old value for v must be passed into the called procedure, so that its value can be assigned to the name for v that *always* emerges from the procedure.

Procedure $foo(v)$

```
if (c) then
  v ← 7
else
  /* Do nothing */
fi
end
```

Procedure $foo(v_{out}, v_{in})$

```
v0 ← vin
if (c) then
  v1 ← 7
else
  /* Do nothing */
fi
v2 ←  $\phi(v_0, v_1)$ 
vout ← v2
end
```

SSA form can be computed by a data flow framework, in which the transfer function for a node with multiple reaching defs of v generates its own def of v . Uses are then named by the solution in effect at the associated node.

SSA form construction [9]

1. Every preserving def is turned into a killing def, by copying potentially unmodified values (at subscripted defs, call sites, aliased defs, etc.).
2. Each ordinary definition of v defines a new name.
3. At each node in the flow graph where multiple definitions of v meet, a ϕ -function is introduced to represent yet another new name for v .
4. Uses are renamed by their dominating definition (where uses at a ϕ -function are regarded as belonging to the appropriate predecessor node of the ϕ -function).