05 - Notation

Rick Adrion

How to write it down?

- **natural language**
- structured natural language
- pictorial notation
  - Charts, Diagrams, Box-and-Arrow Charts
  - Graphs
    - Flowgraphs
    - Parse Trees
    - Call graphs
    - Dataflow graphs
- data models
- formal language(s)
  - state-oriented
  - function-oriented
  - object-oriented

Which of these are best adapted to providing which types of answers to which types of stakeholders?

Optative vs. indicative mood

- Indicative: describes how things in the world are regardless of the behavior of the system
  - "Each seat is located in one and only one theater."
- Optative: describes what you want the system to achieve
  - "Better seats should be allocated before worse seats at the same price."
- Principle of uniform mood
  - Indicative and optative properties should be entirely separated in a document
  - Reduces confusion of both the authors and the readers
  - Increases chances of finding problems
  - If the software works right, both sets of properties will hold as facts

Mood mixing: example

- The lift never goes from the n^{th} to the $n+2^{nd}$ floor without passing the $n+1^{st}$ floor.
- The lift never passes a floor for which the floor selection light inside the lift is illuminated without stoping at that floor.
- If the motor polarity is set to up and the motor switch setting is changed from off to on, the lift starts to rise within 250 msecs.
- If the upwards arrow indicator at a floor is not illuminated when the lift stops at the floor, it will not leave in the upwards direction. **
- The doors are never open at a floor unless the lift is stationary at that floor. ***
- When the lift arrives at a floor, the lift-present sensor at the floor is set to on.
- If an up call button at a floor is pressed when the corresponding light is off, the light comes on and remains on until the call is serviced by the lift stopping at that floor and leaving in the upwards direction.

Natural Language

- Advantages
  - Easy to train users
  - Clarity is possible (but may be difficult)
  - Completeness is possible (but by no mean assured)
  - Easily modified
  - It is the "least common denominator"
- Disadvantages
  - Determining consistency between natural language artifacts and anything else is hard/subjective
  - Ambiguity in natural language is easy and often intentional
  - Clear natural language expression is very difficult
  - The longer the text, the more information, the more the risk of inconsistency, the harder it is to determine
  - No way of knowing when a specification is "complete"

Natural Language Summary

- Cannot reason definitively about natural language
- Cannot be sure that natural language artifacts are consistent with other artifacts
- Assurances to stakeholders are shaky

UNIVERSITY OF MASSACHUSETTS AMHERST • DEPARTMENT OF COMPUTER SCIENCE • CMPSCI 520/620 FALL 2003

How to write it down?

- natural language
- **structured natural language**
- pictorial notation
  - Charts, Diagrams, Box-and-Arrow Charts
  - Graphs
    - Flowgraphs
    - Parse Trees
    - Call graphs
    - Dataflow graphs
- formal language(s)
  - state-oriented
  - function-oriented
  - object-oriented

Structured "Natural" Language

- Disciplined Use of Natural Language
- Response to natural language problems of:
  - Imprecision
  - Ambiguity
  - Consistency (especially when due to size)
  - Inability to reason effectively and definitively
- Familiar approaches:
  - Restricted use of reserved terms
  - Structuring (paragraph numbering, outline form, templates, etc.)
- Other, earlier examples of disciplined use of natural language:
  - Legal documents
  - Recipes
  - Help systems

Declarative vs. Imperative

- Declarative specification
  - Pre and postcondition pairs, where
    - a precondition is a condition on the input and system state at the start of executing the function and the postcondition is a condition on the output and the system state after the execution of the function.
  - Implementation independent, but under specifies
- Imperative specification
  - describe the activities to be performed to get from the input and initial system state to the output and resulting system state.
  - Leads to executable specification, but over specifies by giving an implementation

Declarative

Event flow input: start heating
Data flow input:batch ID
Data store input:ALLOCATION OF BATCH TO COOKING TANK
HEATER OF COOKING TANK
RECIPE OF BATCH
Event flow output: start controlling
Data store output:TEMPERATURE RAMP DATA

Precondition 1:
batch ID occurs exactly once in ALLOCATION OF BATCH TO COOKING TANK
and allocation of batch is cooking tank ID
and recipe for batch ID occurs in RECIPE OF BATCH with end time and end temperature

Postcondition 1:
new(ramp ID) + batch ID + cooking tank ID + heater ID + end time + end temperature
exists in TEMPERATURE RAMP DATA

Precondition 2:
batch ID does not occur exactly once in ALLOCATION OF BATCH TO COOKING TANK
Postcondition 2: error

Precondition 3:
recipe for batch ID does not occur in RECIPE OF BATCH temperature
Postcondition 3: error

But what about state of other data?

Figure 17. A declarative specification.

ACM Computing Surveys, Vol. 30, No. 4, December 1998

COMPUTER SCIENCE

Imperative

Implementation specific?

```
Event flow input: start heating
Data flow input:batch ID
Data store input:ALLOCATION OF BATCH TO COOKING TANK
HEATER OF COOKING TANK
RECIPE OF BATCH
Event flow output: start controlling
Data store output:TEMPERATURE RAMP DATA
If batch ID occurs exactly once in ALLOCATION OF BATCH TO COOKING TANK
then get cooking tank id from ALLOCATION OF BATCH TO COOKING TANK;
     get heater ID from HEATER OF COOKING TANK;
     if recipe for batch ID occurs in RECIPE OF BATCH
     then get end time and end temperature from RECIPE OF BATCH;
          create ramp ID;
          update TEMPERATURE RAMP DATA
          with ramp ID + batch ID + cooking tank ID + heater ID + end time + end temperature
     else error;
else error
```

Figure 18. An imperative specification.

UNIVERSITY OF MASSACHUSETTS AMHERST • DEPARTMENT OF COMPUTER SCIENCE • CMPSCI 520/620 FALL 2003

COMPUTER SCIENCE

PSL (Problem Statement Language)

DESCRIPTION:
this process performs those actions needed to interpret time records to produce a pay statement for each hourly employee;
KEYWORDS: independent;
ATTRIBUTES ARE:
complexity-level
high;
GENERATES pay-statement, error-listing;
RECEIVES: time-card;
SUBPARTS ARE: hourly-paycheck-validation, hourly-emp-update, h-report-entry-generates, hourly-paycheck-production;
PART OF: payroll-processing;
DERIVES: pay-statement;
USING: time-card, hourly-employee-record;
DERIVES: hourly-employee-report;
USING: time-card, hourly-employee-record;
DERIVES: error-listing;
USING: time-card, hourly-employee-record;
PROCEDURE: read record, add up hours, multiply by pay rate…..
HAPPENS: number-of-payments TIMES-PER pay-period;
TRIGGERED BY: hourly-emp-processing-event;
TERMINATION-CAUSES: new-employee-processing-event;
SECURITY IS: company-only;

UNIVERSITY OF MASSACHUSETTS AMHERST • DEPARTMENT OF COMPUTER SCIENCE • CMPSCI 520/620 FALL 2003

PSL (Problem Statement Language)

PROCEDURE:

1. compute gross pay from time card data
2. compute tax from gross pay
3. subtract tax from gross pay to obtain net pay
4. update hourly employee record
5. update department record accordingly
6. generate paycheck

Note: if status code indicates that employee did not work this pay period, no processing will be done for this employee

Discipline Mechanisms in PSL

- Use of keywords (defined elsewhere in specification)
  - fosters precision, clarity
  - helps support consistency determination: some
  - keyword fields have defined relations to others (eg. Input-to and output-from)
- Use of templates
  - facilitates determination of completeness
  - fosters clarity
  - facilitates consistency checking
- Use of structure:

  HIERARCHY:
  - standard practice for dealing with size, complexity
  - exploits innate human capacity for abstraction

  DATA FLOW:

  CONTROL FLOW:

Structured Natural Language

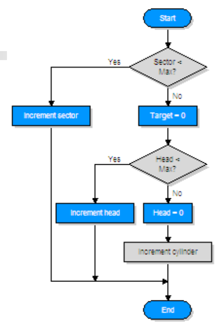
- big step in the right direction
  - improvement over unstructured natural language
- possible to determine some kinds of consistency thru:
  - mechanisms for reducing ambiguity
  - mechanisms for fostering completeness
  - structuring mechanisms for dealing with complexity
- **but**
  - stilted form reduces clarity: less suitable for some key stakeholder groups
  - some residual reliance on natural language means ambiguity remains
  - size is still a problem: PSL specs (for example) can be huge: consistency determination is long/error prone

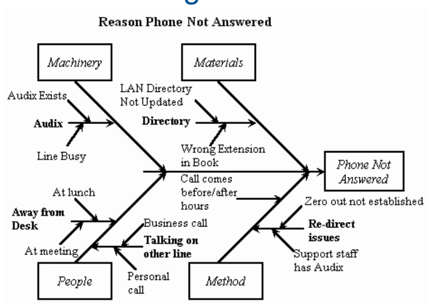
How to write it down?

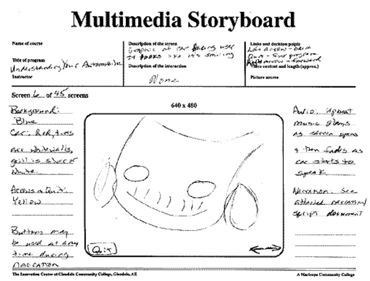
- natural language
- structured natural language
- **pictorial notation**
  - Charts, Diagrams, Box-and-Arrow Charts
  - Graphs
    - Flowgraphs
    - Parse Trees
    - Call graphs
    - Dataflow graphs
- formal language(s)
  - state-oriented
  - function-oriented
  - object-oriented

Various charts

- Flowcharts
- Storyboards
- Cause and Effect Diagram
- Pareto Chart
- Histogram



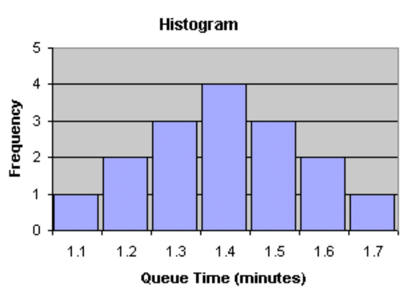


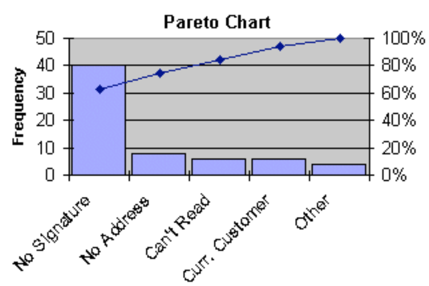


Various charts

- Flowcharts
- Storyboards
- Cause and Effect Diagram
- Pareto Chart
- Histogram … and more

80/20 Rule
- 80% of process defects arise from 20% of the process issues.
- 80% of delays in schedule arise from 20% of the possible causes of the delays.
- 80% of customer complaints arise from 20% of your products or services.





Pictorial and Diagrammatic Approaches

- Diagrams composed of visual elements
  - rigorously defined (definable?) semantics
  - used as modeling devices
  - depict key structural aspects of system
- Benefits
  - greatly improve clarity
  - greatly improve clarity consistency
  - facilitate completeness of notation
  - reduce ambiguity
- but
  - reduce modifiability, perhaps significantly
  - restrictions in semantics impede completeness
  - more on these issues later.....

How to write it down?

- natural language
- structured natural language
- pictorial notation
  - Charts, Diagrams
  - **Graphs**
    - Flowgraphs
    - Parse Trees
    - Call graphs
    - Dataflow graphs
- formal language(s)
  - state-oriented
  - function-oriented
  - object-oriented

Graphs

- A graph, $G = (N, E)$, is an ordered pair consisting of a node set, N, and an edge set, $E = \{(n_i, n_j)\}$
  - If the pairs in E are ordered, then G is called a directed graph, and is depicted with arrowheads on its edges
  - If not, the graph is called an undirected graph
- Graphs provide a mathematical basis for reasoning about s/w
- Graphs are suggestive devices that **help in the visualization of relations**. The set of edges in the graph are visual representations of the ordered pairs that compose relations

Relations:

A ***relation***, R, over a set, $S = \{s_t\}$ is a set of ordered n-tuples
$R = \{r_i\}$, where $r_i = (s_{i,1}, s_{i,2}, \ldots, s_{i,n})$

A ***binary relation*** is a relation where all the tuples are 2-tuples

If (s_i, s_j) is an element of R, then we often write $s_i R s_j$

Another view of relations:

The relation, R, over the set S can be defined as:
$R = \{ (s_i, .., s_j) \mid PRED(s_i, \ldots, s_j) = True$, for some predicate, PRED$\}$

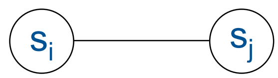
If the tuples are ordered, the relation is called an ***ordered relation***

If the tuples, $<t_{i,1}, t_{i,2}, \ldots\ t_{i,n}>$ are unordered, the relation is an ***unordered relation***

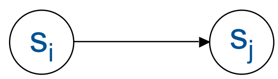
Relations & graphs

- Binary relations ($s_i R\ s_j$)can be represented as a graph
  - unordered

$s_i R\ s_j$



  - ordered



- General relations can be represented as multigraphs, hypergraphs

Some Examples

Let I = {all integers},

Define Q= { (x,y,z) | x, y, z are integers and y = x**2 , z = x**3 }

Let S = {all states of the U.S., S_i},

Define B = { (S_i, S_j) | S_i and S_j share a border}

Let L = {all statements L_i in a program, P},

Define ImmFol = {(L_i, L_j) | the execution of L_j may immediately follow the execution of L_i for some execution of P}

UNIVERSITY OF MASSACHUSETTS AMHERST • DEPARTMENT OF COMPUTER SCIENCE • CMPSCI 520/620 FALL 2003

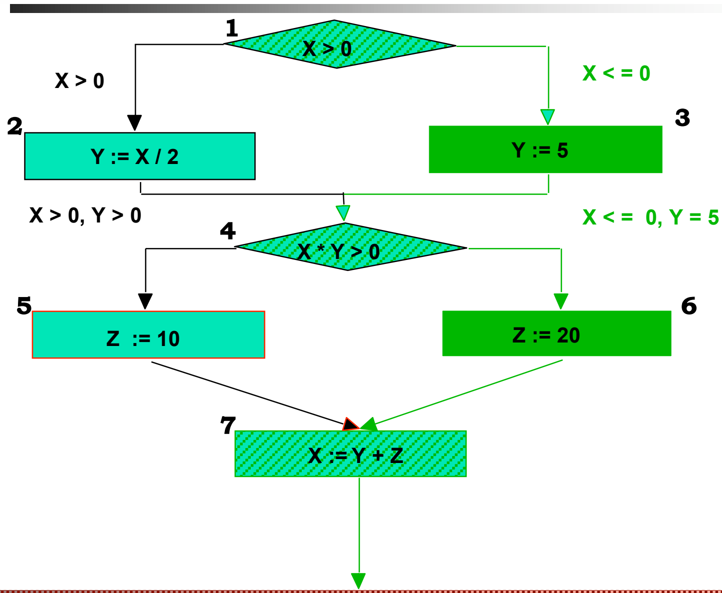
Flowgraphs

Let S = {all statements s_i in a program, P}; and let ImmFol = { (s_i, s_j) | The execution of s_j immediately follows the execution of s_i for some execution of P }

Then FG = (S, ImmFol) is called the **flowgraph** of P

- FG is an ordered graph
- Every execution sequence (ie. the sequence in which the statements of P are executed for a given execution of P) corresponds to a path in FG.
- However, the converse is not true. A path through FG may not correspond to an execution sequence for P
- A loop in P appears as a cycle in FG

Example with an infeasible path



Some Properties of Relations

- Some familiar properties of ordered binary relations, R, over the set $S=\{s_k\}$:
  - Symmetry: s_i R s_j ==> s_j R s_i for all pairs, s_i and s_j in S
  - Reflexivity: s R s, for all s in S
  - Transitivity: s_i R s_j and s_j R s_k ==> s_i R s_k, for all s_i, s_j and s_k in S
  - A relation that is symmetric, reflexive and transitive is called an equivalence relation
  - If R = $\{(s_i, s_j)\}$ is transitive, then C=$\{(s_a, s_b)|$ there exists a sequence, i1, i2, ..., in, such that $s_a=s_{i1}$ R s_{i2}, s_{i2} R s_{i3}, ...
  s_{in-1} R s_{in} = s_b } is called the transitive closure of R
  - Antisymmetry: s_i R s_j ==> ~(s_j R s_i) for all pairs, s_i and s_j in S
  - Irreflexivity: s ~R s for all s in S

University of Massachusetts Amherst · Department of Computer Science · CMPSCI 520/620 Fall 2003

Examples

If S={all subroutines written in Fortran} s_1 R s_2 if and only if s_1 calls s_2, then R is an irreflexive relation

Let PS ={c_e, all the statements in a program that consists of
a set of modules, M={m_t} },
INMOD = { (c_e, c_f) | c_e and c_f appear in
the same module m_t }
INMOD is an equivalence relation

The relation ImmFol (earlier slide) is not transitive

Change ImmFol to Fol, by defining Fol = { (L1, L2) | the execution of L2 may follow the execution of L1 for some execution of P} Fol is still not transitive

University of Massachusetts Amherst · Department of Computer Science · CMPSCI 520/620 Fall 2003

Paths

A ***path,*** P, through an ***ordered graph*** G=(N ,E) is a sequence of edges, (<$n_{i,1}$, $n_{j,1}$ >, <$n_{i,2}$, $n_{j,2}$ >, ... , <$n_{i,t}$, $n_{j,t}$ >) such that $n_{j,k-1} = n_{i,k}$ for all $2 \leq k \leq n$

A **path**, UP, through an ***unordered graph*** UG=(N,U) is a sequence of edges, (<$n_{i,1}$, $n_{j,1}$ >, <$n_{i,2}$, $n_{j,2}$ >, ... , <$n_{i,t}$, $n_{j,t}$ >) such that all of the <$n_{i,z}$, $n_{j,z}$> can be ordered to assure that $n_{j,z-1} = n_{i,z}$ for all $2 \leq k \leq n$

In either case, $n_{i,1}$ is called the start node and $n_{j,t}$ is called the end node.

The length of a path is the number of edges in the path

A **graph** G is **connected** if and only if, for every pair of nodes, n_1, n_2, there is path from one of them to the other with G considered to be an unordered graph.

Paths

- a path, P, through a directed graph G = (N, E) is a sequence of edges, (($n_{i,1}$, $n_{j,1}$), ($n_{i,2}$, $n_{j,2}$), ... , ($n_{i,t}$, $n_{j,t}$)) such that $n_{j,k-1} = n_{i,k}$ for all $2 \leq k \leq t$
  - $n_{i,1}$ is called the start node and $n_{j,t}$ is called the end node
  - the length of a path is the number of edges in the path
  - paths are also frequently represented by a sequence of nodes ($n_{i,1}$, $n_{i,2}$, $n_{i,3}$, …, $n_{i,t}$)

UNIVERSITY OF MASSACHUSETTS AMHERST • DEPARTMENT OF COMPUTER SCIENCE • CMPSCI 520/620 FALL 2003

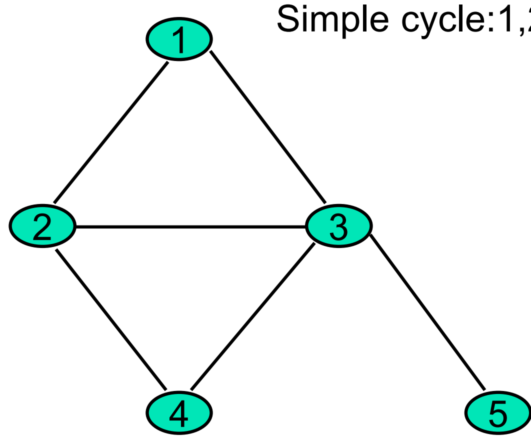
Cycles

- a cycle in a graph G is a path whose start node and end node are the same
- a simple cycle in a graph G is a cycle such that all of its nodes are different (except for the start and end nodes)
- if a graph G has no path through it that is a cycle, then the graph is called acyclic

Examples

Cycle:1,3,2,4,3,1

Simple cycle:1,2,3,1



Trees

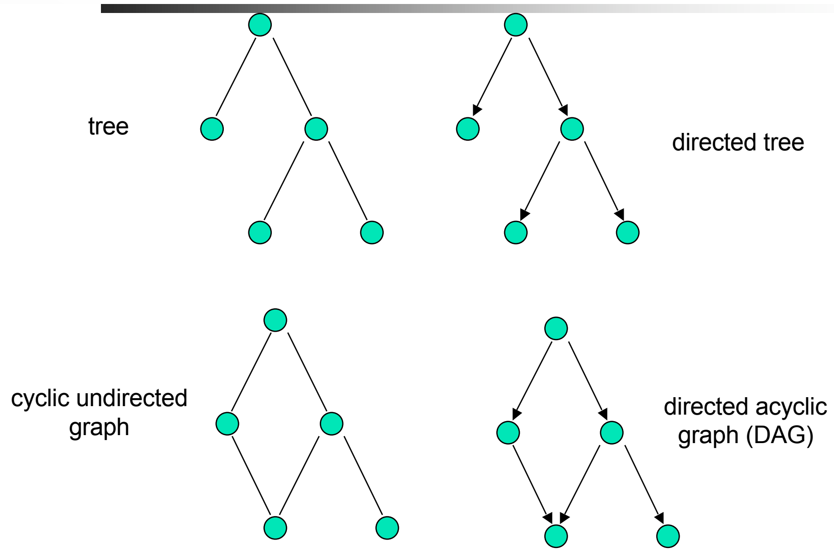
- A **cycle** in a graph G is a path whose start node and end node are the same
- A **simple cycle** in a graph G is a cycle such that all of its nodes are different (except for the start and end nodes)
- If a graph G is connected and has no path through it that is a cycle, then the graph is called **acyclic**.
- An acyclic unordered graph is called a **tree**
  - If the unordered version of an ordered graph is acyclic, the graph is called a **directed tree**
  - A collection of trees is called a **forest**
- If the unordered version of an ordered graph has cycles, but the ordered graph itself has no cycles, then the graph is called a **Directed Acyclic Graph (DAG)**

Trees

- an acyclic, undirected graph is called a tree
- if the undirected version of a directed graph is acyclic, then the graph is called a directed tree
- if the undirected version of a directed graph has cycles, but the directed graph itself has no cycles, then the graph is called a Directed Acyclic Graph (DAG)

COMPUTER SCIENCE

Examples



COMPUTER SCIENCE

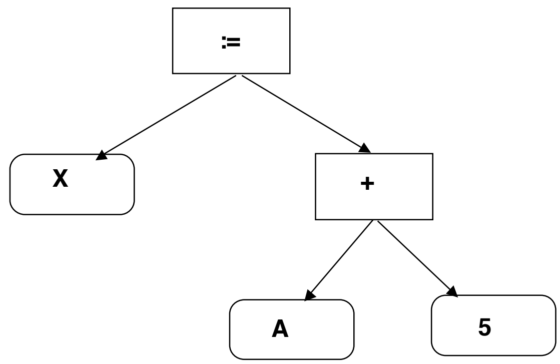
Abstract Syntax Tree (AST)

- a common form for representing expressions
  - executable statements are expressions
  - programs are expressions, where the operator is execute and the operands are the statements
- 2 kinds of nodes: operator and operands
  - operator applied to N operands
- An abstract syntax graph G = (N1, N2, E) where N1 are nodes that represent operators in the language, N2 are nodes that represent identifiers or literals , and E represents is "applied to"

UNIVERSITY OF MASSACHUSETTS AMHERST · DEPARTMENT OF COMPUTER SCIENCE · CMPSCI 520/620 FALL 2003

Abstract Syntax Tree

X:= A + 5;



Abstract Syntax Trees

- have many advantages
  - provide a visual display of the body of an object
    - body of an assignment, addition, while, etc.
  - supports incremental modification
    - incremental syntactic or semantic analysis
  - basis for structural editing
    - user is provided with a template and fills in the slots
    - can assure syntactic consistency
    - need to control granularity of consistency checking
      - e.g., keystroke, semi-colon, user-request
  - used to create other graph models

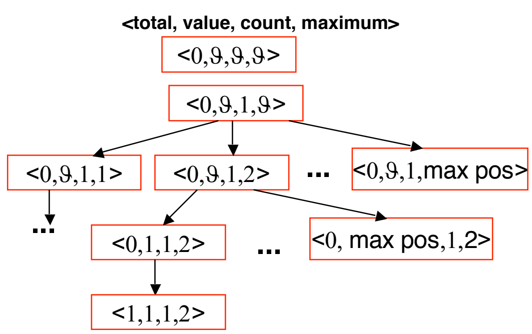
UNIVERSITY OF MASSACHUSETTS AMHERST · DEPARTMENT OF COMPUTER SCIENCE · CMPSCI 520/620 FALL 2003

computation tree

- models all the possible executions of a system
- at each node, shows the state (value) of each variable
- effectively infinite number of paths
- some paths may be effectively infinite

example computation tree

```
total, value, count, maximum : pos int;
total := 0;
count := 1;
read maximum;
while (count <= maximum) do
        read value;
        total := total + value;
        count: = count + 1;
endwhile;
print total;
```



UNIVERSITY OF MASSACHUSETTS AMHERST • DEPARTMENT OF COMPUTER SCIENCE • CMPSCI 520/620 FALL 2003

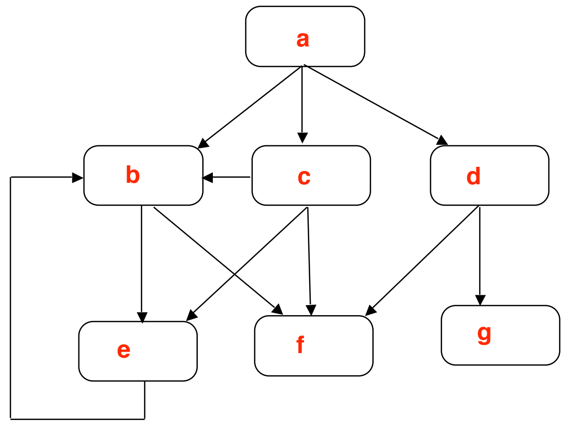
Computation Trees

- have advantages
  - represent the space that we want to reason about
  - for anything interesting they are too large to create or reason about
  - other models of executable behavior are providing **abstractions** of the computation tree model
    - abstract values
    - abstract flow of control
    - specialize abstraction depending on focus of analysis

Callgraphs

- Let PROC = {procedures S_i comprising a program P} and CALLS = {(S_i, S_j) | S_j is directly invoked from S_i during some execution of P}, then CG = (PROC, CALLS) is called the **Call Graph** of P
- CG is
  - a directed graph
  - does not represent the order entities are invoked
  - does not represent the number of times an entity is invoked
  - a cycle in g indicates that the nodes along the cycle syntactically participate in a recursive calling chain
  - if P is written in a language that does not allow recursion, then CG will be acyclic
  - provides a framework for inter-component analysis

Call Graph Example



Control Flow Graph (CFG)

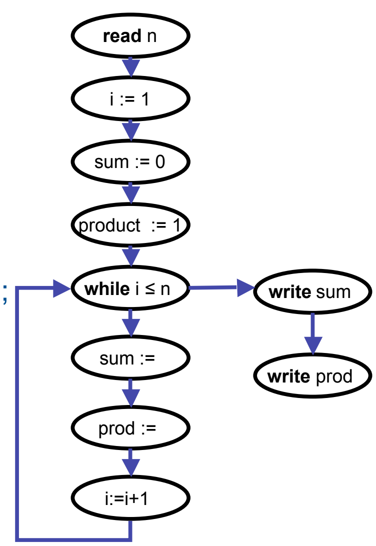
- represents the flow of executable behavior
- G = (N, E, S, T) where
  - the nodes N represent executable instructions (statement or statement fragments);
  - the edges E represent the potential transfer of control;
  - S is a designated start node;
  - T is a designated final node
  - E = { (ni, nj) | syntactically, the execution of nj follows the execution of ni}

Control Flow Graph (CFG)

- nodes may correspond to single statements, parts of statements, or several statements
- execution of a node means that the instructions associated with a node are executed in order from the first instruction to the last
- nodes are 1-in, 1-out

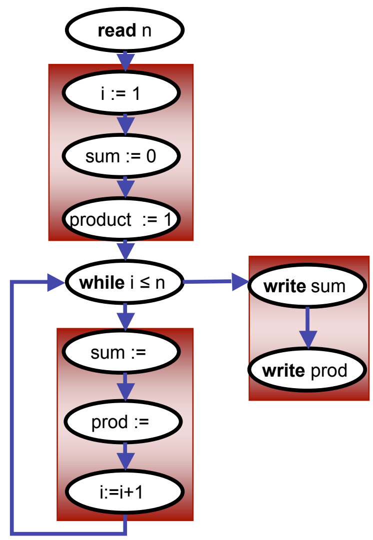
Control Flow Graph Model

```
read n;
i := 1;
sum := 0;
product  := 1;
while i ≤ n do
    sum := sum +1;
    product := product * i;
    i:= i+1;
endwhile;
write sum;
write product;
```



Reducing the CFG

- basic blocks are nodes that contain sequential execution
- Can reduce the number of nodes in the CFG, but may add more complications to the analysis



Benefits of CFG

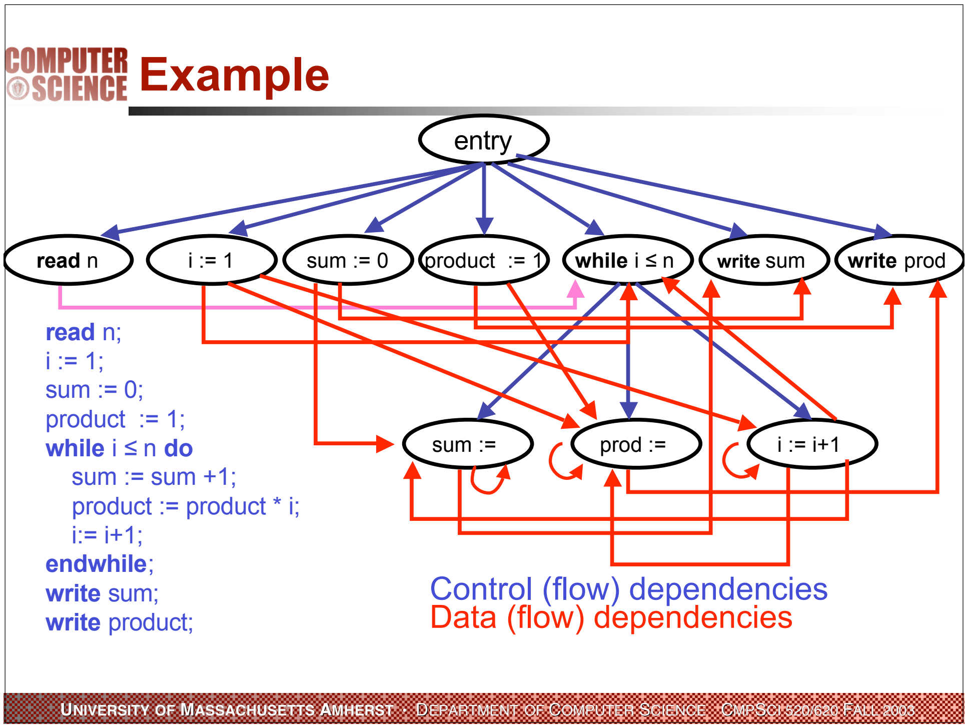
- probably the most commonly used representation
  - numerous variants
- basis for inter-component analysis
  - collections of CFGs
- basis for various transformations
  - compiler optimizations
  - S/W analysis
- basis for automated analysis
  - graphical representations of interesting programs are too complex for direct human understanding

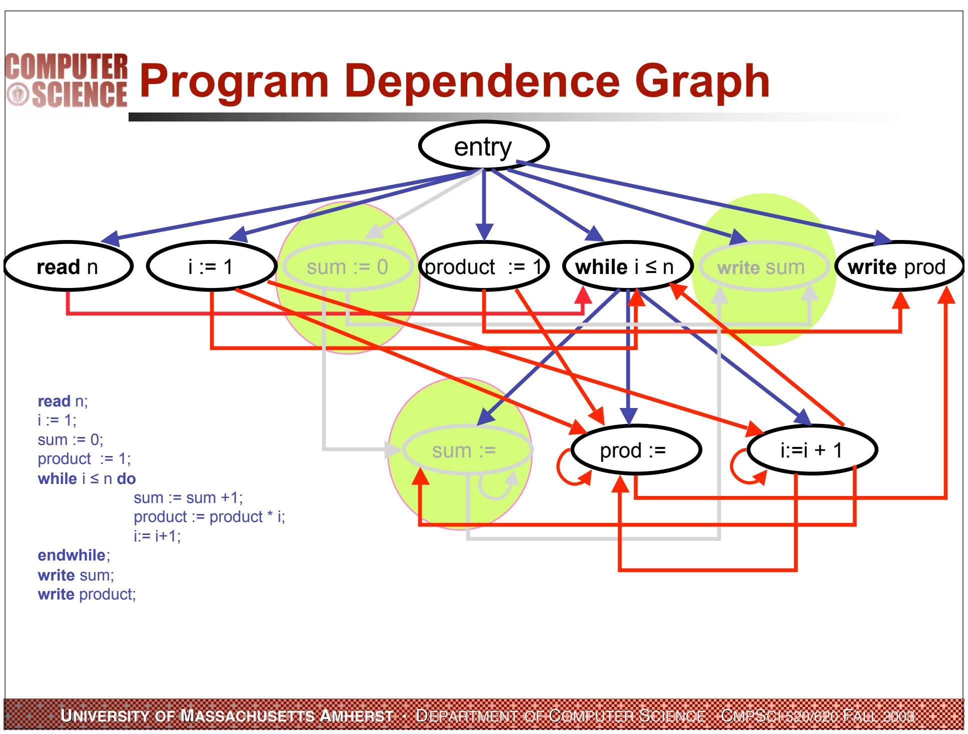
Some dataflow relations

- DataFlow(i, j) if node i creates data that node j uses
- Input(n) if n is a node that supplies initial input data
- Output(n) if n is a node that transmits data to end users
- EdgeAnnotation(e, text) if the string text describes the data that flows along edge e
- NodeAnnotation(n, text) if the string text describes the functioning of node n
- Questions this helps answer:
  - Why create this data? Who uses this data? What results does the end user see? What does the end user have to input?
  - Questions this can't answer: What is the exact sequence of events? How does a node do its job?

Program Dependence Graphs

- G is a directed graph, G = (V, E)
  - edges in E are of several types, representing control and data dependencies
  - vertices in V represent assignment statements and predicates and other special nodes
- Program Slice - Concept introduced by Mark Weiser in 1979
  - Argued it was a mental abstraction that programmers used when debugging
  - Program slice S is a reduced, executable program obtained from P by removing statements from P, such that S replicates part of the behavior of P
  - A slice includes all statements and predicates that might affect V at point p.
- How can we use the Program Dependency Graph to create slices?
  - A slice corresponds to all nodes that are reachable from a selected node (forward slice)


COMPUTER SCIENCE
Example
entry
read n
i := 1
sum := 0
product := 1
while i ≤ n
write sum
write prod
sum :=
prod :=
i := i+1
read n;
i := 1;
sum := 0;
product := 1;
while i ≤ n do
sum := sum +1;
product := product * i;
i:= i+1;
endwhile;
write sum;
write product;
Control (flow) dependencies
Data (flow) dependencies
UNIVERSITY OF MASSACHUSETTS AMHERST • DEPARTMENT OF COMPUTER SCIENCE • CMPSCI 520/620 FALL 2003


COMPUTER SCIENCE
Program Dependence Graph
entry
read n
i := 1
sum := 0
product := 1
while i ≤ n
write sum
write prod
sum :=
prod :=
i:=i + 1
read n;
i := 1;
sum := 0;
product := 1;
while i ≤ n do
sum := sum +1;
product := product * i;
i:= i+1;
endwhile;
write sum;
write product;
UNIVERSITY OF MASSACHUSETTS AMHERST • DEPARTMENT OF COMPUTER SCIENCE • CMPSCI 520/620 FALL 2003

Dataflow graphs & slices

- Uses
  - Data flow coverage criteria for selecting test cases
    - coverage criteria exercise subsets of control and data dependencies in the hope of exposing faults
  - debugging:
    - which statements could have caused an observed failure?
  - maintenance:
    - which statements will be affected by a change?
    - which statements could affect this statement?
  - dependence analysis
    - program dependencies provide a theory for restricting/focusing attention
- Problems
  - in practice, a program slice is often too big to be useful
  - infeasible paths lead to imprecision
  - complex data structures lead to imprecision

Other Types of Graphs

- A **Multigraph** MG is an ordered pair MG = (N, C) where N is a set of nodes $\{n_i\}$ and C is a collection of pairs of nodes (edges) with repetitions allowed (ie. C can be a multiset)
- A **Hypergraph** HG is an ordered pair HG = (N, T) where N is a set of nodes $\{n_i\}$ and T is a set of t-tuples of nodes, where t > 2.
- A **Hypermultigraph** is a hypergraph where the set of t-tuples can be a **multiset**
- A **bipartite graph** BG is an ordered pair, BG = {BN, E} where BN is a node set that is the union of two disjoint subsets, N_1 U N_2, and no edge in E has both nodes in either N_1 or N_2
  - A bipartite graph is often called a 2-colorable graph
  - An k-colorable graph is defined analogously, with BN being the disjoint union of k subsets

UNIVERSITY OF MASSACHUSETTS AMHERST • DEPARTMENT OF COMPUTER SCIENCE • CMPSCI 520/620 FALL 2003

Types of graphs

- Differences in graphs result from different choices for nodes & relations
- Hierarchy:
  - Models "consists of" or "is a part of"
  - Key to divide-and -conquer approaches to understanding
- Data Flow:
  - Nodes represent set of sites where data is generated/used
  - Each edge is a (data generated, data used) node pair
- Control Flow:
  - Nodes represent units of functionality
  - (n1, n2) is an edge in this graph if and only if unit n2 can execute immediately after n1 executes

Types of graphs

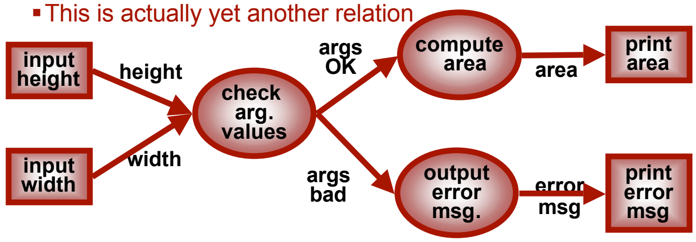
- Finite State Machines
  - Nodes represent all possible different "execution states"
  - (s1, s2) is an edge if and only if it is possible for state s2 to immediately succeed s1. Called a transition from s1 to s2
  - Edges annotated with transition condition
    - Annotations are relations too
  - Juxtaposition of annotation atop what it is annotating
- Petri Nets
  - Multiple node and edge types in the same diagram

- We will come back to this …

COMPUTER SCIENCE

DATA FLOW DIAGRAMS

- Capture system functionality : What does system do? How?
- Basic components of a data flow diagram:
  - Nodes, represented by circles (boxes), are functional units
  - Edges, represented by arrows, are data flows between units
  - Both augmented by separate annotation relations
- Boxes (sometimes circles), represent I/O data
  - This is actually yet another relation



- There is ambiguity and misuse of notation here:
  - one circle is a test, others are functions
  - are multiple arrows in and out “and” or “or”?

COMPUTER SCIENCE

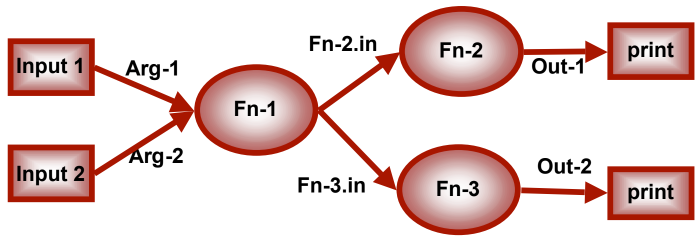
Constraints

- Supply Additional Semantics
  - Node cannot begin until data arrives along all in-edges
    - Or any arrow(?)
  - When node terminates, data passes along all out-edges
    - (at most) one (?)
  - There must be exactly one Output node
    - Clearly an unusual type of DFD
- These constraints support additional types of reasoning, e.g. about parallelism
- Constraints often specified using first order logic

UNIVERSITY OF MASSACHUSETTS AMHERST • DEPARTMENT OF COMPUTER SCIENCE • CMPSCI 520/620 FALL 2003

The power of annotation:

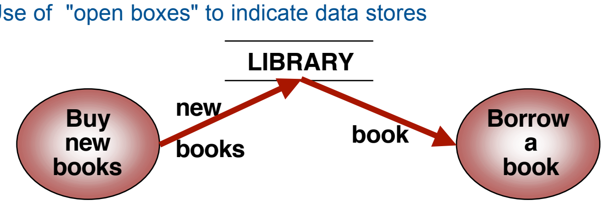
- Improper use of notation is not saved by annotation here:



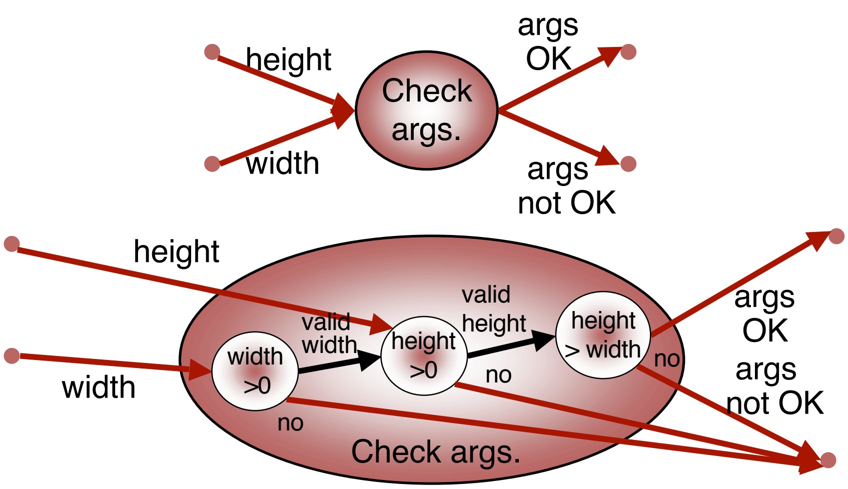
- "Data Flow" is a family of types of diagrams/relation sets
- Need more formal definitions of DFD's
- Need more modeling power
  - Large systems (diagrams)?
  - Multiple input and output streams?
  - What about data stores?
  - Say more about data?

More Elaborate DFD's

- Hierarchy enables representation of large, complex systems
  - bubbles can be "opened up"
  - details of a bubble represented by a whole sub-DFD
  - constraints on consistency: all arrows ending on/starting from parent bubble are shown as inputs and outputs on sub-DFD--and that the sub-DFD has no additional I/O
- Use of logical connectives to add semantics to multiple inputs and outputs to functions
  - sometimes **all** inputs are needed, sometimes **any**
  - sometimes **all** outputs are generated, sometimes **some**
  - annotations on edges, indicating logical conditions, can be useful also
- Use of "open boxes" to indicate data stores



Hierarchical DFD



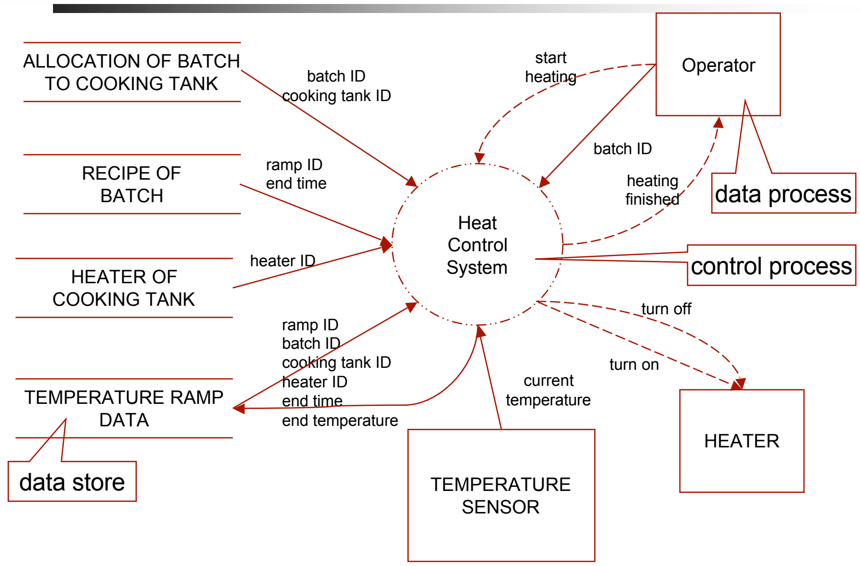
Augmenting DFD's

- DFD's focus on functionality, using data as a vehicle
  - Data shown as unstructured atomic units--usually unrealistic
  - Complex functions cannot be adequately defined without delving into the details of how they handle structured data
  - Sub-DFD's can show how the high level data that high level DFD's deal with is decomposed
  - But this is implicit data definition
    - Can be hard to read/inconsistent
- Data specification is worth doing explicitly, carefully
- Usually using Disciplined Natural Language--eg. templates
  - Hierarchical relations
  - Function(s) creating and using data
  - Possible other attributes:
    - persistent? where? encoding? ...

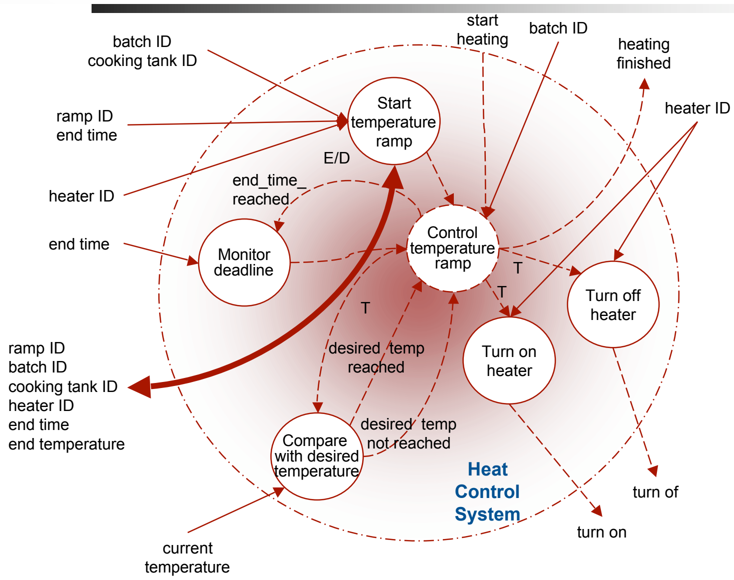
Juice Plant control system

- Configuration
  - two kinds of TANKs
    - STORAGE TANKs and COOKING TANKs
  - each COOKING TANK is connected to one HEATER
  - each HEATER belongs to one COOKING TANK
  - BATCH of juice is allocated to one COOKING TANK and belongs to exactly one RECIPE.
  - each RECIPE is related to a JUICE SPECIFICATION
- Why is this a useful example
  - Relatively complex
  - Ambiguous
  - Can compare with other notations

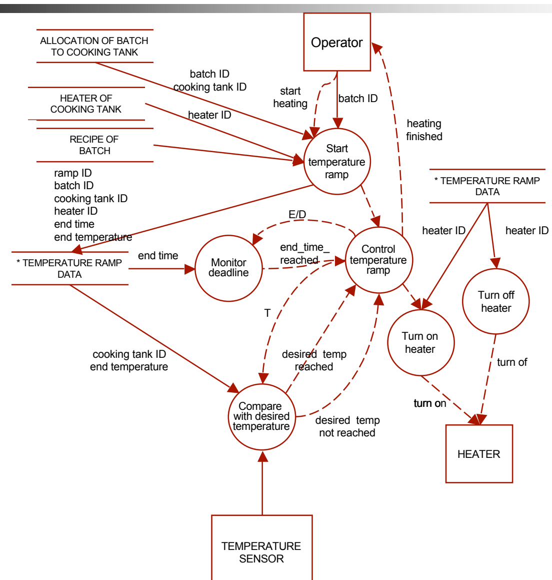
Juice Plant control system



Elaboration of heat controller



Juice Plant control system



UNIVERSITY OF MASSACHUSETTS AMHERST • DEPARTMENT OF COMPUTER SCIENCE • CMPSCI 520/620 FALL 2003

Stakeholder Concerns

- Buyer (Juice Company)
  - Before development: What should it do?
    - How to improve productivity? Quality?
  - During development: What will it do?
    - What heat control algorithms to use?
    - How to plan for expansion? More cookers, more storage, more recipes. …
    - Is the project on time?
  - After development, before delivery: What does it do?
    - Does it do what it was intended to do?
- Software developer: How should it be developed?
    - What is the system architecture?
    - What sensors, algorithms to use?
- User (plant workers): Does it improve job performance, maintain job security?
- Safety Inspector: Is it safe?

DFD

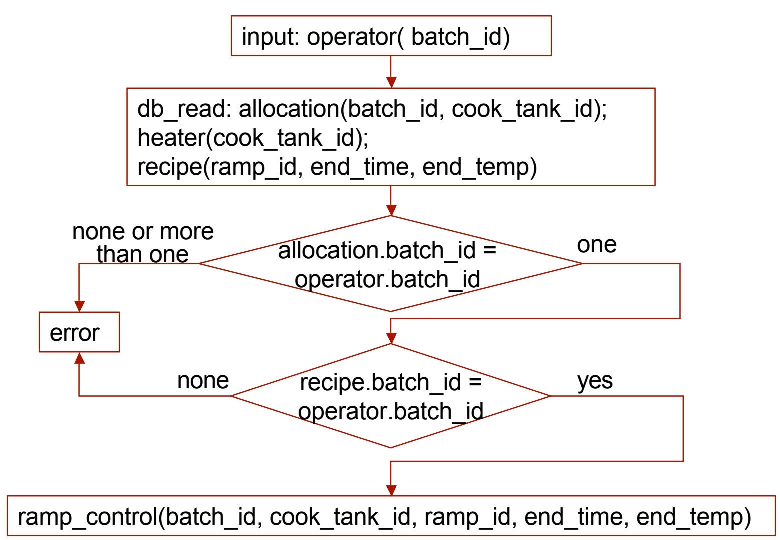
- Advantages
  - See overall system structure
  - Reason about what outputs the system will produce
  - Powerful aid to intuition and efficiency of communication with a clear advantages over natural language
- Disadvantages
  - Very primitive type of model, as noted it is actually more a family of model types
  - The actual relation(s) are rarely made clear and precise
    - How will functionality be achieved
    - How fast will this run
    - Database locking/consistency management
- Questions DFD's are adept at answering:
  - What results are produced? (What does this do?)
  - How might the answers be evolved?

UNIVERSITY OF MASSACHUSETTS AMHERST • DEPARTMENT OF COMPUTER SCIENCE • CMPSCI 520/620 FALL 2003

Control Flow Diagrams (redux)

- Similar to DFD's except edges represent flow of control, rather than flow of data
- Usual enhancements:
  - annotate edges with predicates
  - special symbols for branching, concurrency control....
- Control flow graphs also address questions like
  - “what does this do” and
  - “how does this do it”

CFG for juice plant



COMPUTER SCIENCE

CFG

- Advantages
  - Sense of what algorithms to use
  - Constraints on data appearing
    - that the allocation is maintained in some (e.g. sorted) order
- Can reason about functionality
  - Possible to assign batch to more than one heating tank
  - Possible a batch has no associated recipe
- Drawbacks:
  - What about safety?
  - What about data?

COMPUTER SCIENCE

Control vs. Data Flow Views

- Both shed light on similar questions
- One focuses on data evolution, the other on functional development
- Both are useful, neither removes the need for the other
- Control flow graphs map closely to implementation code written in procedural languages, e.g., imperative
  - Good basis for determining consistency of code with ideas expressed as data flow
- Data flow graphs focus more on the product itself, seem better at helping understand if and how it gets evolved, e.g., declarative
  - Seem better adapted to studying earlier formulations of the problem to be solved, and ways of solving it