Dataflow analysis

https://www.youtube.com/playlist?list=PLF3-CvSRq2SaApl3Lnu6Tu_ecsBr94543

- The field of software analysis is highly diverse. There are many approaches with different strengths and limitations, in aspects such as
  - soundness,
  - completeness,
  - applicability,
  - and scalability.
- Dataflow analysis is one of the dominant approaches to software analysis.

What is dataflow analysis?

- Static analysis reasoning about flow of data in program
- Different kinds of data constants, variables, expressions
- User by bug-finding tools and compilers

The WHILE language

- an example program written in this language to compute 5!

```
x = 5;
y = 1;
while (x != 1){
      y = x * y;
      x = x - 1;
}
```

- a formal grammar that precisely describes the syntax of programs written in the while language:

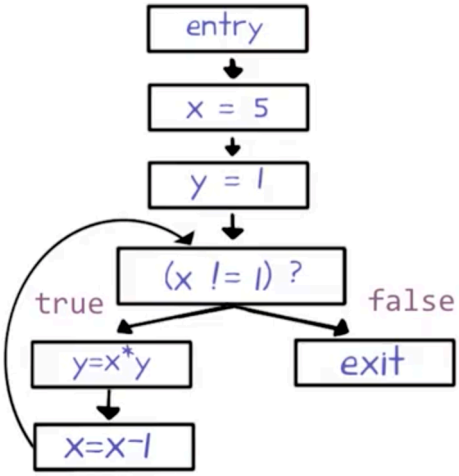
```
(statement) S :: x = a | S1 ; S2 |
            if (b) { S1 } else { S2 } |
            while (b) { S1 }
(arithmetic expression) a ::= x | n | a1 * a2 | a1 - a2
(boolean expression) b ::= true | !b | b1 && b2 | a1 != a2
(integer variable) x
(integer constant) n
```

- Notice that this definition of statements, arithmetic expressions is recursive.
- The syntax of these expressions can be extended to include other operators.
- The WHILE language does not have any fancy constructs such as functions, pointers or threads that are provided in commonly used programming languages such C and Java. This is because the presence of loops already makes the WHILE language expressive enough that interesting properties of programs written in this language are

undecidable, and yet simple enough to allow us to study the fundamentals of data flow analysis.

- Read more about Backus-Naur Form here: https://en.wikipedia.org/wiki/Backus-Naur_Form

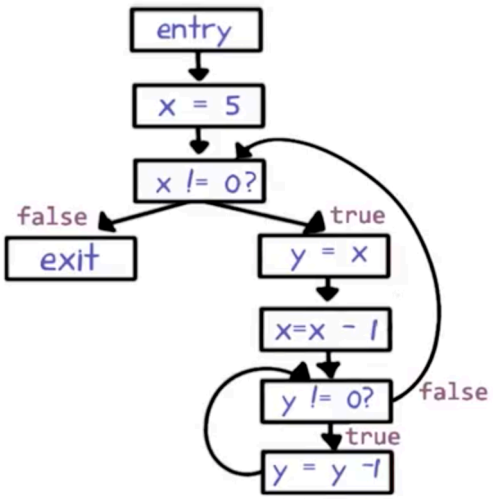
Control-flow graphs




```
x = 5;
y = 1;
while (x != 1){
  y = x * y;
  x = x - 1
}
```

- An example converting a control-flow graph into the program it came from in the syntax of the while language:

```
S ::== V = A | S ; S | if( B ) { S } else { S } | while( B ) { S }
A ::== V | C | A * A | A - A
B ::== true | ! B | B && B | A != A
V ::== (any variable name)
C ::== (any integer constant)
```




```
x = 5;
while (x != 0) {
  y = x;
  x = x - 1;
  while (y != 0) {
    y = y - 1
  }
}
```

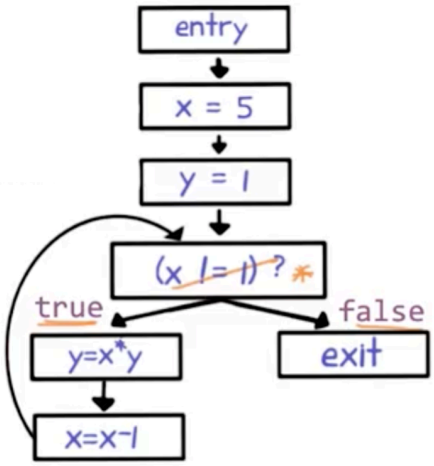
Soundness, completeness, termination

- Impossible for analysis to achieve all three together
- **Dataflow analysis sacrifices completeness**

- Sound: will report all facts that could occur in actual runs
- Incomplete: may report additional facts that can't occur in actual runs

Abstracting control-flow conditions

- Abstracts away control-flow conditions with non-deterministic choice denoted by ***** **symbol**
- Non-deterministic choice -> assumes condition can evaluate to true or false
- Considers all paths possible in actual runs (**sound**) and maybe paths that are never possible (**incomplete**)

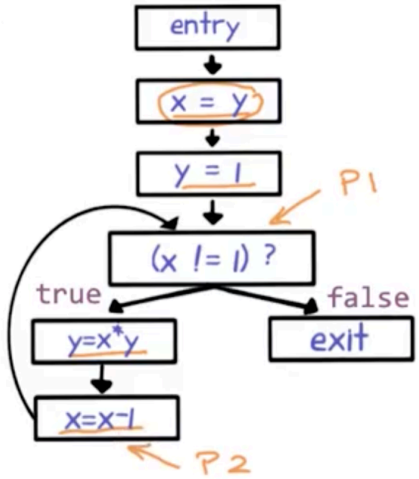


Applications of dataflow analysis

Reaching definitions analysis	**Very busy expressions analysis**
-> Find usage of uninitialized variables	-> Reduce code size
Available expressions analysis	**Live variables analysis**
-> Avoid recomputing expressions	-> Allocate registers efficiently

Reaching definitions analysis

- **Goal:** determine, for each program point, which assignments have been made and not overwritten, when execution reaches that point along some path



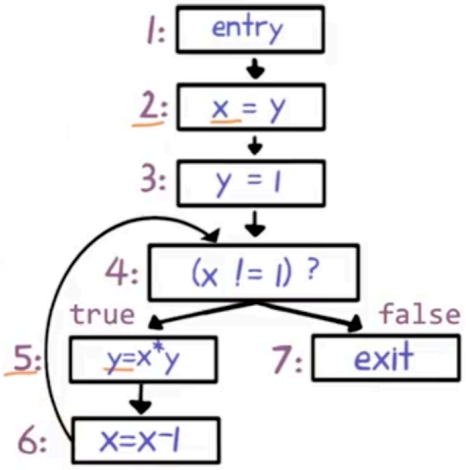
- Consider “**assignment**” == “**definition**” and 2 program points P1 and P2 for the above control-flow graph.
- Statements 1 and 3 in the following list are true:
  - The assignment y = 1 reaches P1
  - The assignment y = 1 reaches P2
  - The assignment y = x * y reaches P1

Result of dataflow analysis (informally)

- Set of facts at each program point
- For reaching definitions analysis, fact is a pair of the form:

  <defined variable name, defining node label>
- Example

<x,2>, <y,5>

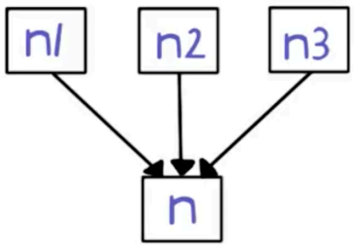


Result of dataflow analysis (formally)

- Give distinct label n to each node
- IN(n) = set of facts at entry of node n
- OUT(n) = set of facts at exit of node n
- Dataflow analysis computes IN(n) and OUT(n) for each node
- Repeat 2 operations until IN(n) and OUT(n) stop changing
  - Called “**saturated**” or “**fixed point**”

Reaching definitions analysis: operation #1

$$IN[n] = \bigcup_{n' \in predecessors(n)} OUT[n']$$

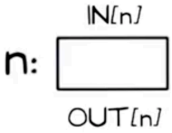


$$IN[n] = OUT[n1] \cup OUT[n2] \cup OUT[n3]$$

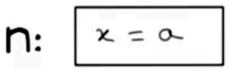
Reaching definitions analysis: operation #2

- GEN(n) = the set of definitions that are generated by node n
- KILL(n) = the set of definitions that are killed/modified/overwritten by node n, i.e. the set of definitions except the one at node n
- b? = a condition statement

$$OUT[n] = (IN[n] - KILL[n]) \cup GEN[n]$$



$GEN[n] = \emptyset$
$KILL[n] = \emptyset$



$GEN[n] = \{ \langle x, n \rangle \}$
$KILL[n] = \{ \langle x, m \rangle : m \neq n \}$

Overall algorithm: chaotic iteration

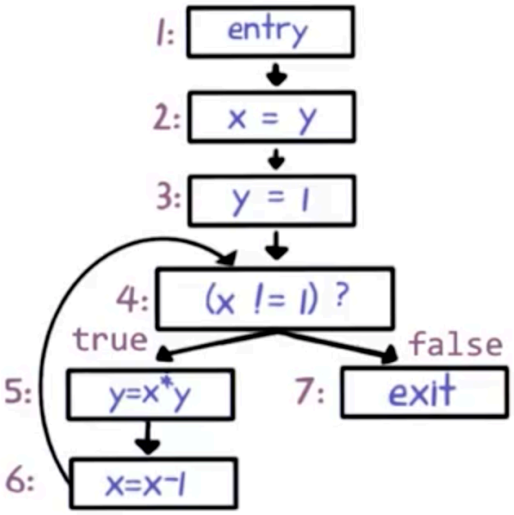
For (each node n):
$IN[n] = OUT[n] = \emptyset$
$OUT[entry] = \{ \langle v, ? \rangle : v \text{ is a program variable} \}$
Repeat:
For (each node n):
$IN[n] = \bigcup_{n' \in predecessors(n)} OUT[n']$
$OUT[n] = (IN[n] - KILL[n]) \cup GEN[n]$
until $IN[n]$ and $OUT[n]$ stop changing for all n

- OUT set s initialized to contain a hypothetical definition for each variable v in the program denoted by <v,?>.

It captures the fact that each variable is undefined or uninitialized at the start of the program.

Reaching definitions analysis example

n	IN[n]	OUT[n]
1	–	{ <x,?>, <y,?> }
2	{ <x,?>, <y,?> }	{ <x,2>, <y,?> }
3	{ <x,2>, <y,?> }	{ <x,2>, <y,3> }
4	{<x,2>, <y,3>, <y,5>, <x,6>}	{<x,2>, <y,3>, <y,5>, <x,6>}
5	{<x,2>, <y,3>, <y,5>, <x,6>}	{<x,2>, <y,3>, <y,5>, <x,6>}
6	{<x,2>, <y,5>, <x,6>}	{<x,2>, <y,5>, <x,6>}
7	{<x,2>, <y,3>, <y,5>, <x,6>}	–

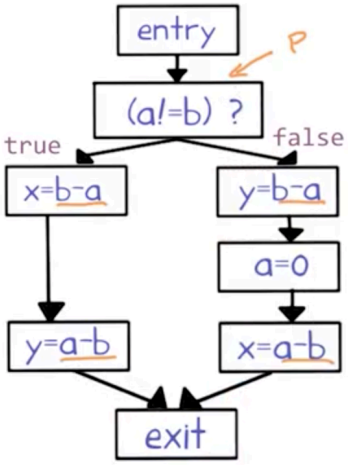


Does it always terminate?

- Chaotic iteration algorithm always terminate
  - The 2 operations of reaching definitions analysis are **monotonic**
    - **IN and OUT sets never shrink, only grow**
  - Largest they can be is set of all definitions in program, which is finite
    - **IN and OUT cannot grow forever**
  - **=> IN and OUT will stop changing after some iteration**

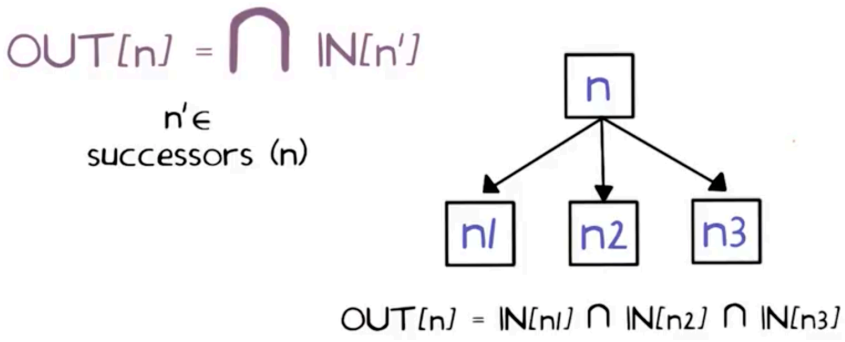
Very busy expressions analysis

- **Goal:** Determine very busy expressions at the exit from the point
- An **expression** is **very busy** if, no matter what path is taken, the expression is used before any of the variables occuring in it are redefined
- Let's consider the 2 expressions in the following program, a-b and b-a.

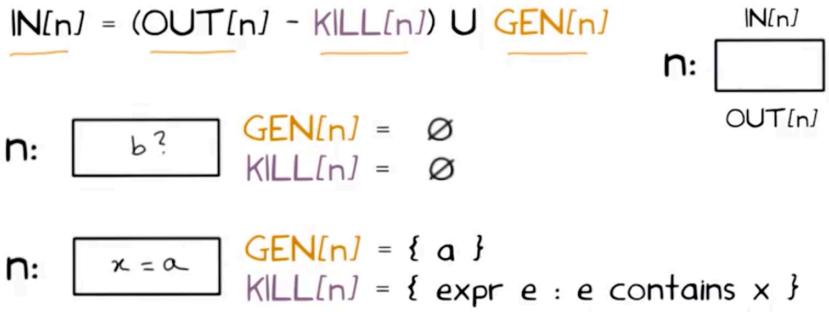


- The expression b-a is very busy at the program point P, but a-b it is not.

Very busy expressions analysis: operation #1



Very busy expressions analysis: operation #2



Overall algorithm: chaotic iteration

- The overall algorithm for very busy expressions analysis is nearly identical to that for reaching definitions analysis, but it has 3 notable differences.
  - Notice that the roles of the IN and OUT sets in these 2 operations are switched, reflecting the fact that very busy expressions analysis propagates information backwards in a control-flow graph, in contrast to reaching definitions analysis, which propagates information forward.
  - we take the intersection of sets as opposed to their union.

For (each node n):

IN[n] = OUT[n] = set of all exprs in program

IN[exit] = ∅

Repeat:

For (each node n):

$$OUT[n] = \bigcap_{n' \in successors(n)} IN[n']$$

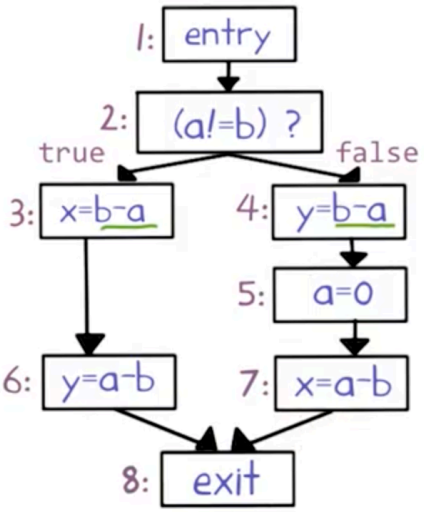
$$IN[n] = (OUT[n] - KILL[n]) \cup GEN[n]$$

until IN[n] and OUT[n] stop changing for all n

Very busy expressions analysis example

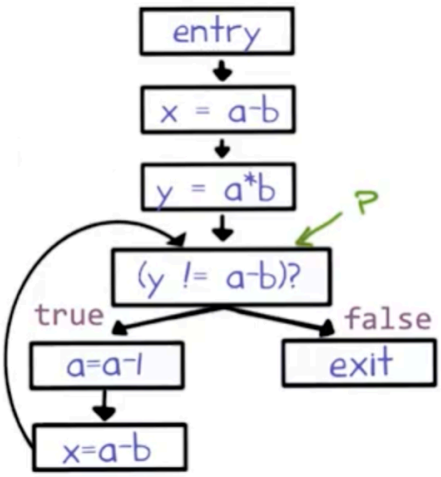
- The expression a-b is very busy.

n	IN[n]	OUT[n]
1:	—	{b-a}
2:	{b-a}	{b-a}
3:	{a-b, b-a}	{a-b}
4:	{b-a}	∅
5:	∅	{a-b}
6:	{a-b}	∅
7:	{a-b}	∅
8:	∅	—



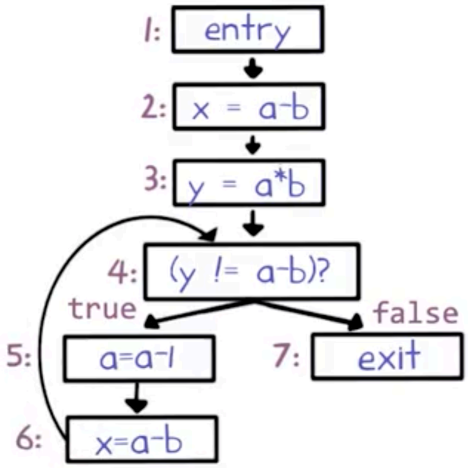
Available expressions analysis

- **Goal**: Determine, for each program point, which expressions must already have been computed, and not later modified, on all paths to the program point.



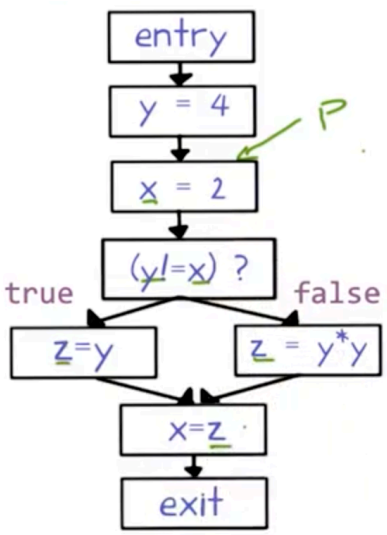
- The expression a - b is available at program point P, but a * b is not.

n	IN[n]	OUT[n]
1	—	∅
2	~~{a-b, a*b, a-1}~~ ∅	{a-b, ~~a*b~~, ~~a-1~~}
3	{a-b, ~~a*b~~, ~~a-1~~}	{a-b, a*b, ~~a-1~~}
4	{a-b, ~~a*b~~, ~~a-1~~}	{a-b, ~~a*b~~, ~~a-1~~}
5	{a-b, ~~a*b~~, ~~a-1~~}	~~{a-b, a*b, a-1}~~ ∅
6	~~{a-b, a*b, a-1}~~ ∅	{a-b, ~~a*b~~, ~~a-1~~}
7	{a-b, ~~a*b~~, ~~a-1~~}	—



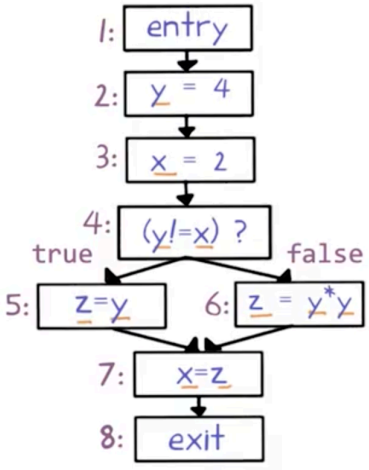
Live variables analysis

- **Goal:** Determine for each program point which variables could be **live** at this point's exit.
- A variable is **live** if there is a path to a use of the variable that doesn't redefine the variable.



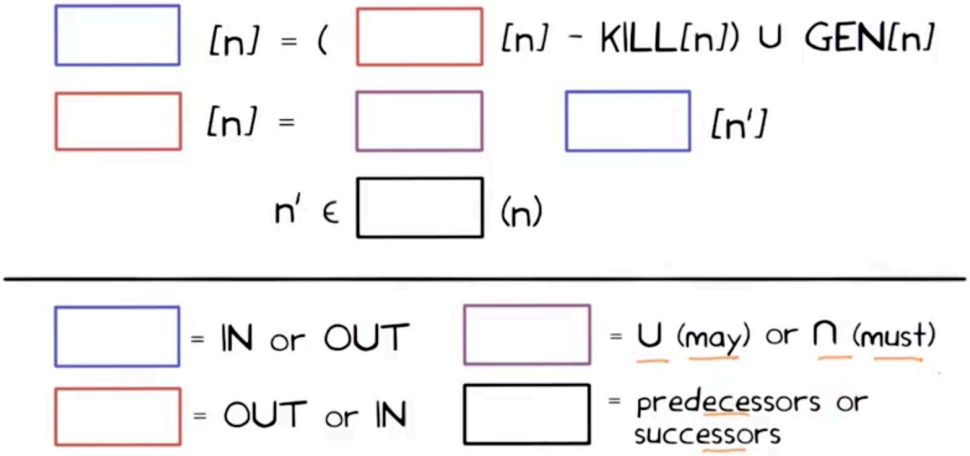
- The variable y is live, but x and z are not at program point P.

n	IN[n]	OUT[n]
1:	—	∅
2:	∅	~~∅~~ {y}
3:	~~∅~~ {y}	~~∅~~ {x, y}
4:	~~∅~~ {x, y}	~~∅~~ {y}
5:	~~∅~~ {y}	~~∅~~ {z}
6:	~~∅~~ {y}	~~∅~~ {z}
7:	~~∅~~ {z}	∅
8:	∅	—

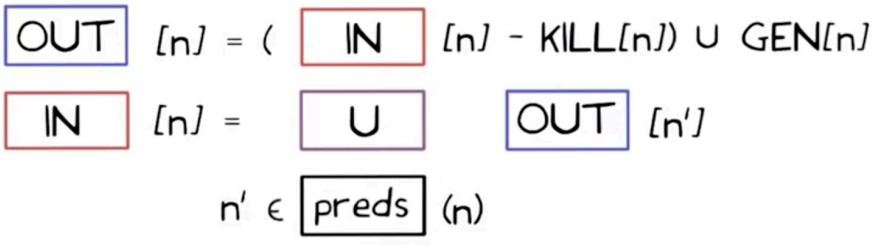


- What's interesting is that even though this program has three variables, x, y, and z, at no point are more than two of these 3 variables simultaneously live. This information can be used to generate assembly code that uses only 2 instead of 3 registers for storing the contents of these variables.

Overall pattern of dataflow analysis

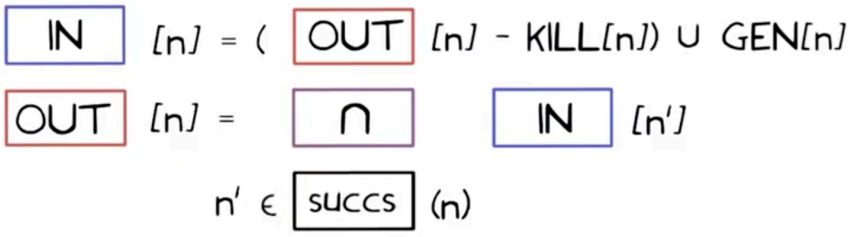


Reaching definitions analysis



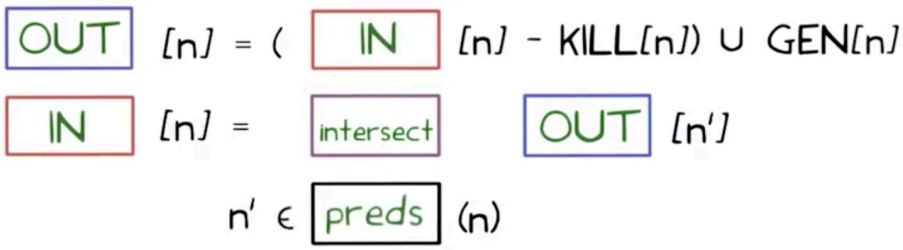
- a forward may analysis

Very busy expression analysis



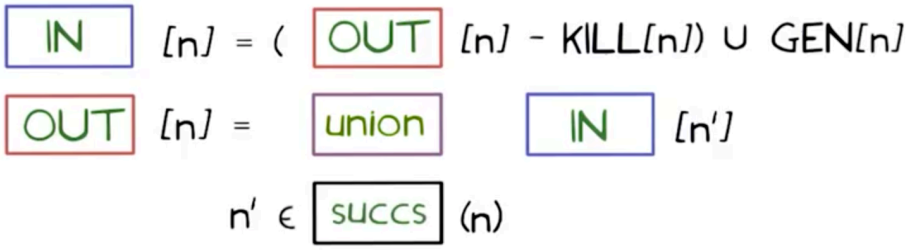
- a backwards must analysis

Available expressions analysis



- a forward must analysis

Live variables analysis



- a backward may analysis

Classifying dataflow analysis

	May	**Must**
Forward	Reaching definitions	Available expressions
Backward	Live variables	Very busy expressions

What have we learned?

- What is dataflow analysis?
- Reasoning about flow of data using control-flow graphs
- Specifying dataflow analyses using local rules
- Chaotic iteration algorithm to compute global properties
- Four classical dataflow analyses

- Classification: forward vs. backward, may vs. must

Suplimentar

- Marius Minea, *Analiză statică. Analiza fluxului de date, note de curs.*

Notă[1]

[1]