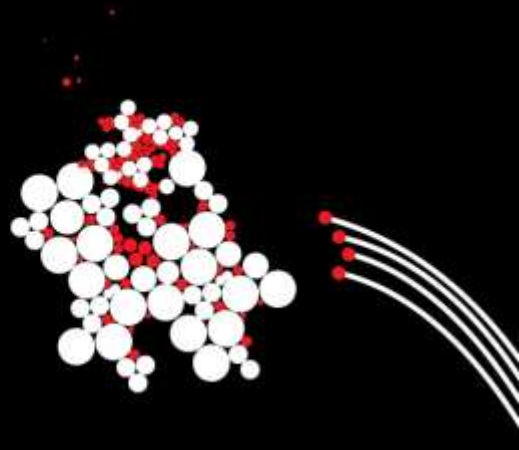
UNIVERSITY OF TWENTE.

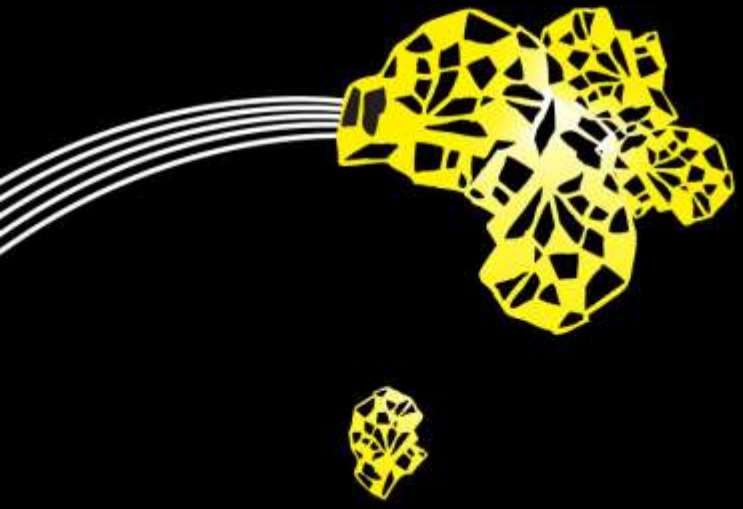


COMPILER CONSTRUCTION:

CC 4-1: GRAPH-BASED IR

15 MAY 2019

MODULE 8: PROGRAMMING PARADIGMS



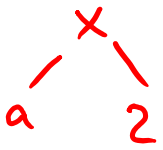
LOOKING BACK AND LOOKING (1 TOKEN) AHEAD

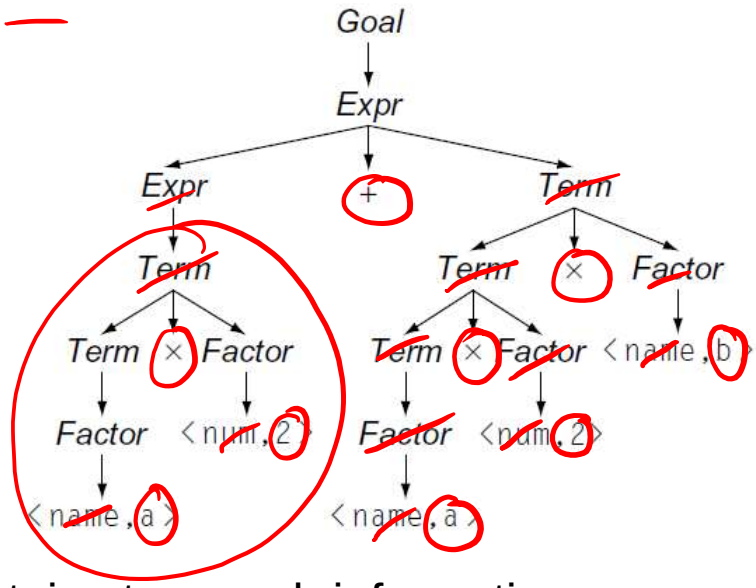
- Block 1: Scanning phase (EC Chapter 2)
  - Regular Expressions and Deterministic Finite Automata
  - Greediness
  - ANTLR lexer grammars
- Block 2: Parsing phase (EC Chapter 3)
  - Context-Free Grammars and parse trees
  - Top-down LL(1)-parsing
  - ANTLR (parser) grammars
- Block 3: Elaboration phase (EC Chapter 4)
  - Type checking and inference
  - Attribute grammars and symbol tables
  - ANTLR actions and tree visitors
- Block 4: Intermediate representations (EC Chapter 5)
  - Graphical: Control Flow Graphs, Data Flow Graphs, Call Graphs
  - Linear: ILOC

ABSTRACT SYNTAX TREES

- Given a grammar, every (parsable) expression gives rise to a parse tree
  - E.g., for $a \times 2 + a \times 2 \times b$:

Goal	→	*Expr*
Expr	→	*Expr* + *Term*
	|	*Expr* - *Term*
	|	*Term*
Term	→	*Term* × *Factor*
	|	*Term* ÷ *Factor*
	|	*Factor*
Factor	→	(*Expr*)
	|	num
	|	name





- For most purposes, this contains too much information
  - Spurious terminals (separators, delimiters)
  - Spurious nonterminals (auxiliary levels)

ABSTRACT SYNTAX TREES

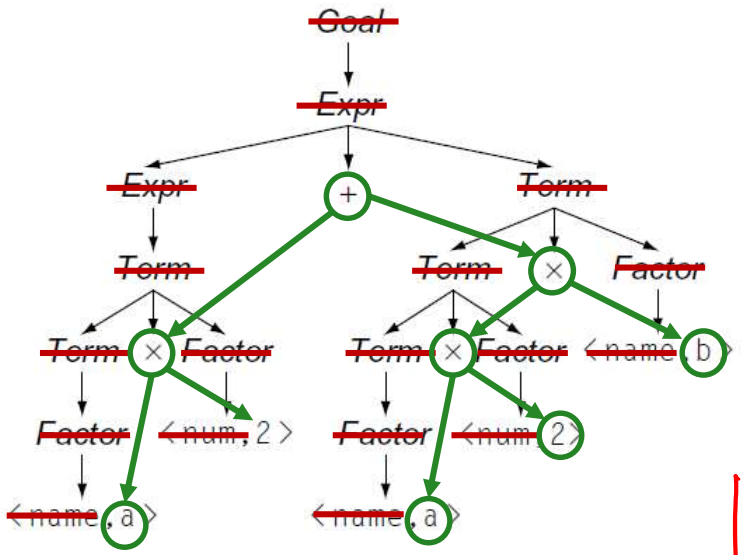
- Given a grammar, every (parsable) expression gives rise to a parse tree
  - E.g., for $a \times 2 + a \times 2 \times b$:

Goal → *Expr*

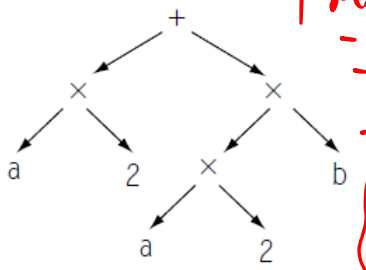
Expr → *Expr* + *Term*
| *Expr* - *Term*
| *Term*

Term → *Term* × *Factor*
| *Term* ÷ *Factor*
| *Factor*

Factor → (*Expr*)
| num
| name

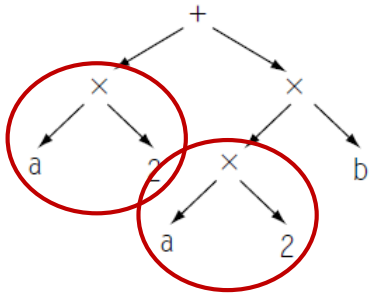


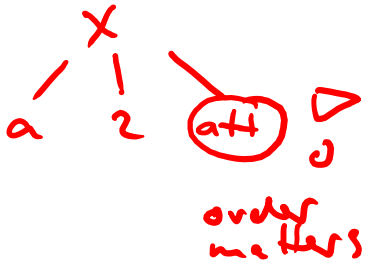
- For most purposes, this contains too much information
  - Spurious terminals (separators, delimiters)
  - Spurious nonterminals (auxiliary levels)
- *Abstract syntax tree*: cleaned-up parse tree
  - Different variants exist
  - Depends on what you want to do

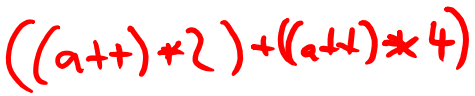


DIRECTED ACYCLIC GRAPHS (AKA TERM GRAPHS)

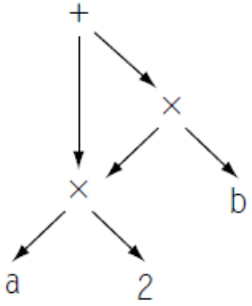
- What strikes you about
  - Two of its subtrees are “the same”
  - More precisely: They are isomorphic







- Isomorphic subtrees can be shared
  - This does not change the meaning of the tree
  - We avoid double work when walking it, e.g. for type checking
  - In absence of side effects, need to execute only once



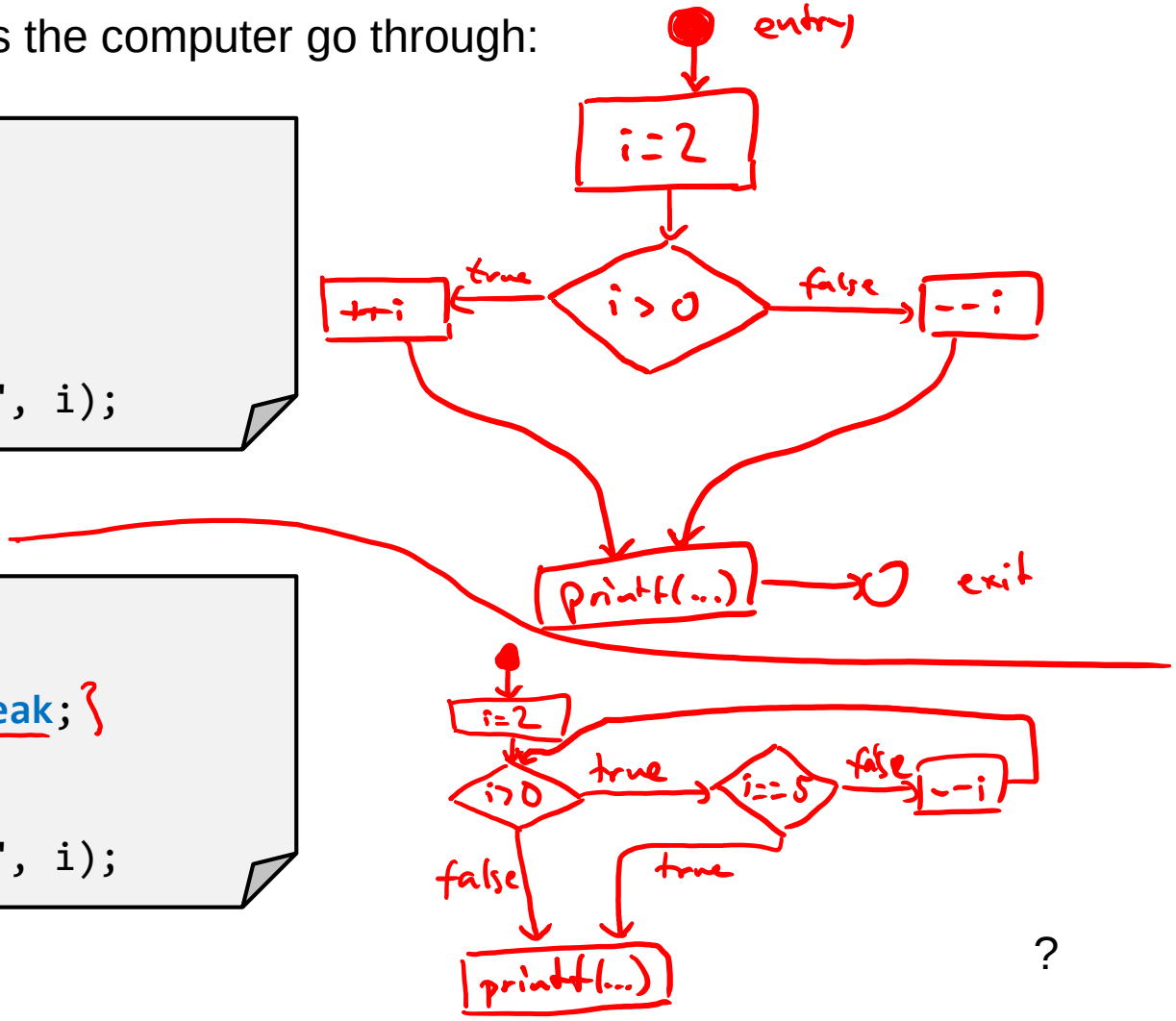
- *Term graph*
  - Directed acyclic (abstract) graph representation of a term, where common subtrees are shared

CONTROL FLOW GRAPHS (AKA FLOW GRAPHS)

- In what order does the computer go through:

```
1 (...) i=2;
2 if(i > 0)
3   ++i;
4 else
5   --i;
6 printf("i: %d%n", i);
```

```
1 (...) i=2;
2 while(i > 0) {
3   if(i == 5) break;
4   --i;
5 }
6 printf("i: %d%n", i);
```



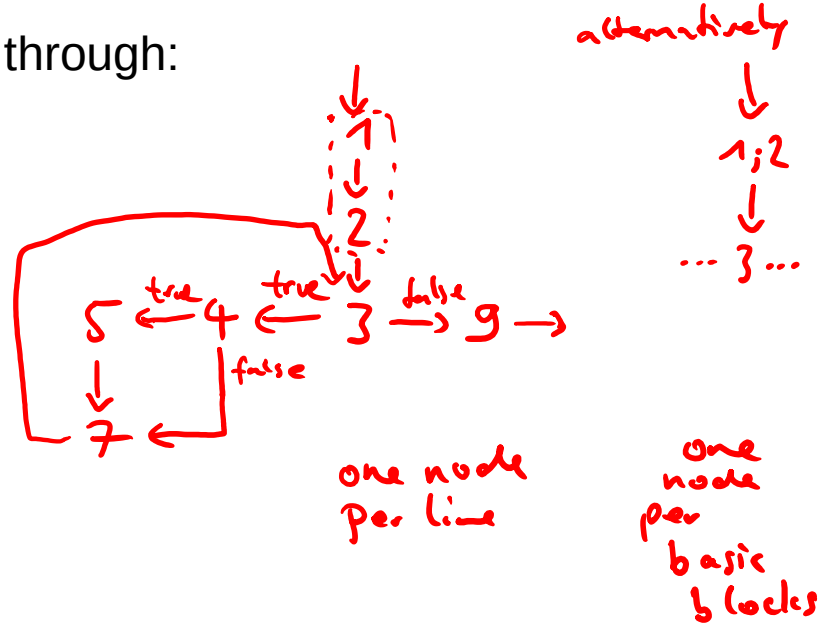
?

CONTROL FLOW GRAPHS (AKA FLOW GRAPHS)

- In what order does the computer go through:

```
1 int max = 0;
2 int i = 0;
3 while (i < a.length) {
4     if (a[i] > max) {
5         max = a[i];
6     }
7     i = i + 1;
8 }
9 printf("Max: %d%n", max);
```

?



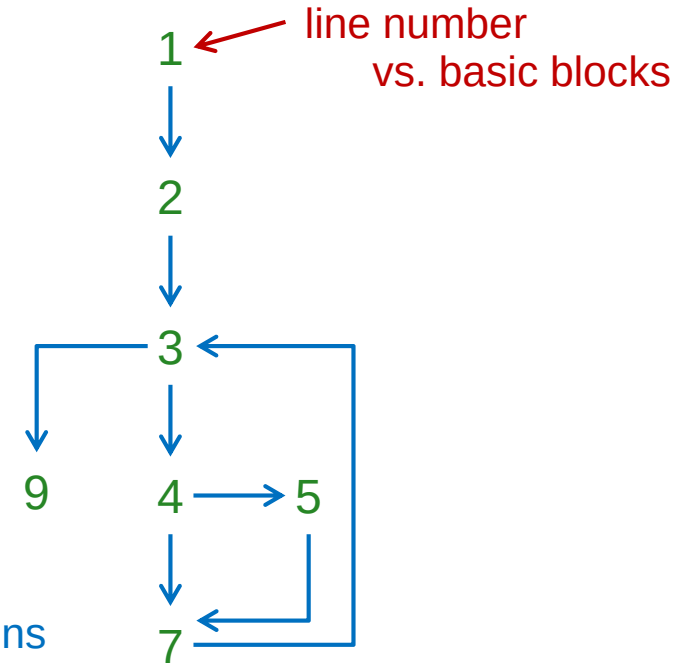
- Representable as a graph
  - Nodes represent statements
  - Edges stand for possibilities that the control goes from source to target
- This is called a (Control) Flow Graph
  - Forms the basis of many types of optimizations
  - E.g., *dead code analysis*

CONTROL FLOW GRAPHS (AKA FLOW GRAPHS)

- In what order does the computer go through:

```
1 int max = 0;
2 int i = 0;
3 while (i < a.length) {
4     if (a[i] > max) {
5         max = a[i];
6     }
7     i = i + 1;
8 }
9 printf("Max: %d%n", max);
```

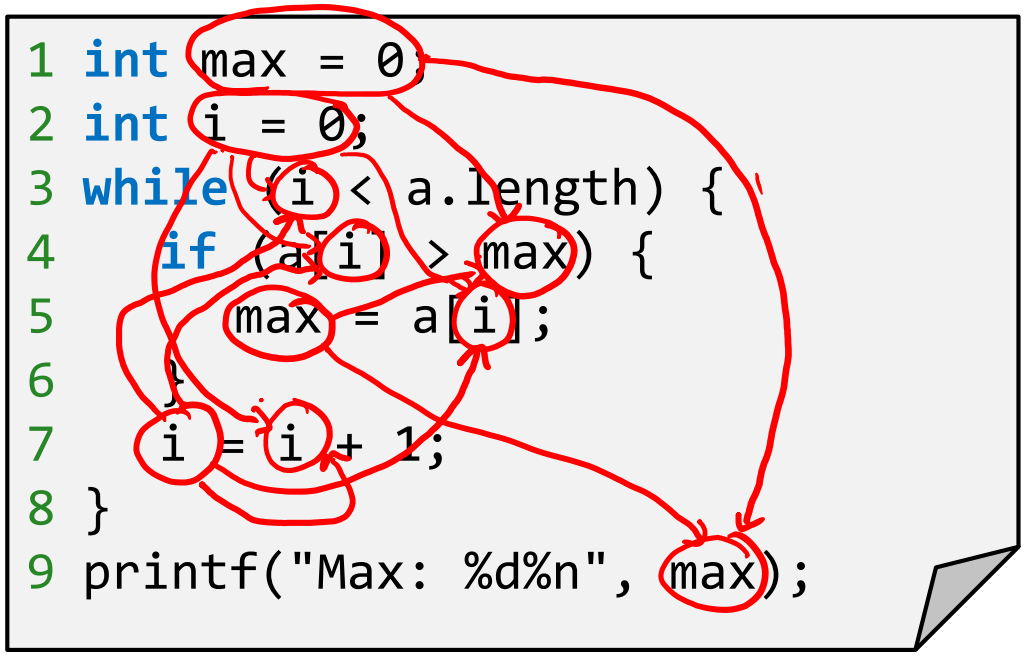
?



- Representable as a graph
  - Nodes represent statements
  - Edges stand for possibilities that the control goes from source to target
- This is called a (Control) Flow Graph
  - Forms the basis of many types of optimizations
  - E.g., *dead code analysis*

DATA DEPENDENCE GRAPHS (AKA DATA FLOW GRAPHS)

- What dependencies between variables exist in

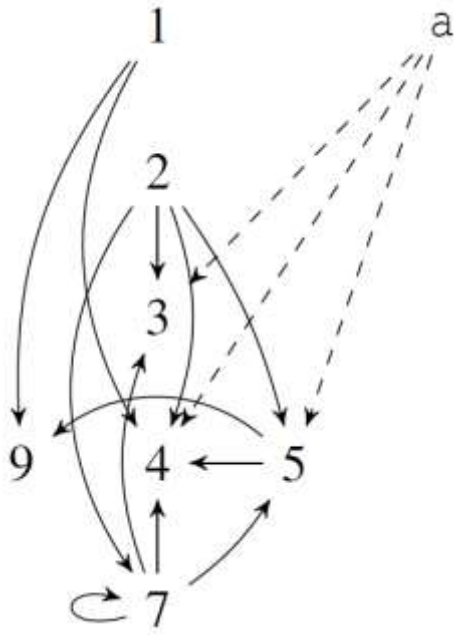


?

DATA DEPENDENCE GRAPHS (AKA DATA FLOW GRAPHS)

- What dependencies between variables exist in

```
1 int max = 0;
2 int i = 0;
3 while (i < a.length) {
4     if (a[i] > max) {
5         max = a[i];
6     }
7     i = i + 1;
8 }
9 printf("Max: %d%n", max);
```



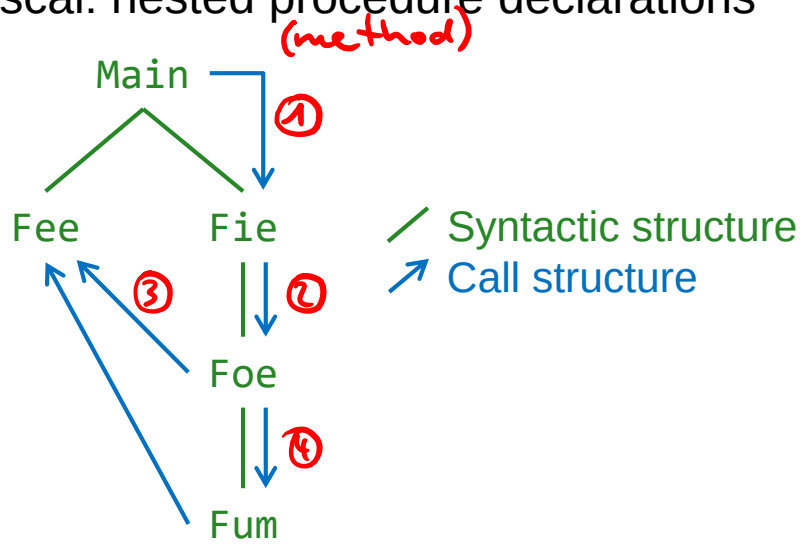
- Representable as graph
  - Edges stand for use (in target) of the value provided by source
- Nodes typically stand for single operations
  - load, store – in linear IR
- Used in optimisations
  - e.g. statement reordering

Here, nodes stand for entire lines of code

CALL GRAPHS

```
program Main(...);
  procedure Fee;
    begin { Fee }
      ...
    end;
  procedure Fie;
    procedure Foe;
      procedure Fum;
        begin { Fum }
          Fee
        end;
      begin { Foe }
        Fee;
        Fum
      end;
    begin { Fie }
      Foe
    end;
  begin { Main }
    Fie
end.
```

- Pascal: nested procedure declarations

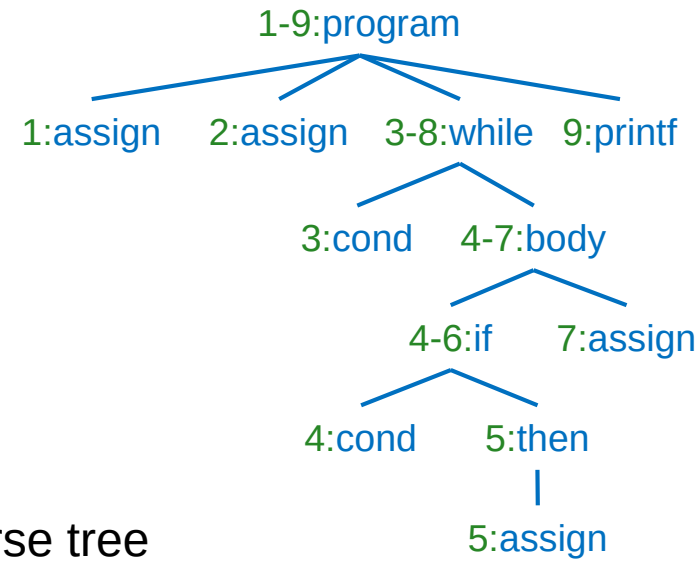


- Procedures call one another
  - Calls can be represented graphically
  - Edges stand for (potential) calls
- Further complication in OO programming
  - Target of call determined at runtime by type of receiver (“dynamic dispatch”)

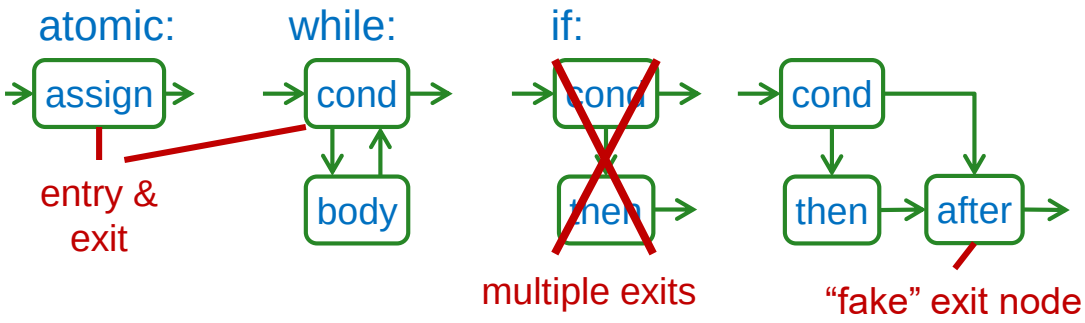
CONSTRUCTING CONTROL FLOW GRAPHS

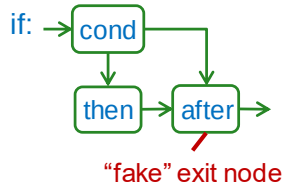
- Construction can be done as attribute computation

```
1 int max = 0;
2 int i = 0;
3 while (i < a.length) {
4     if (a[i] > max) {
5         max = a[i];
6     }
7     i = i + 1;
8 }
9 printf("Max: %d%n", max);
```

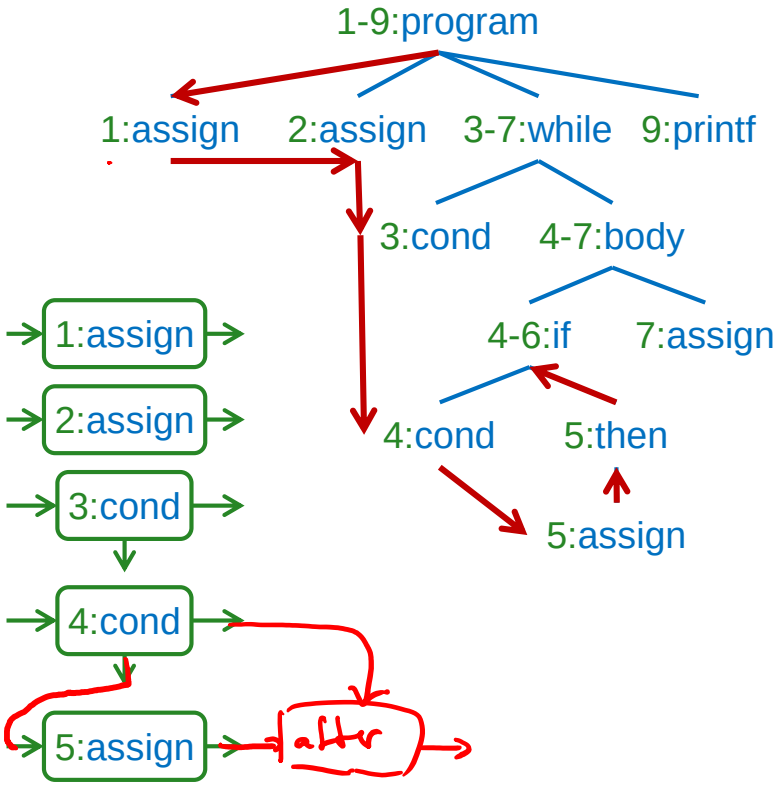


- Build flow graph for every node in parse tree
  - Every subtree has distinct single entry and exit node

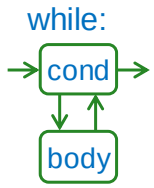


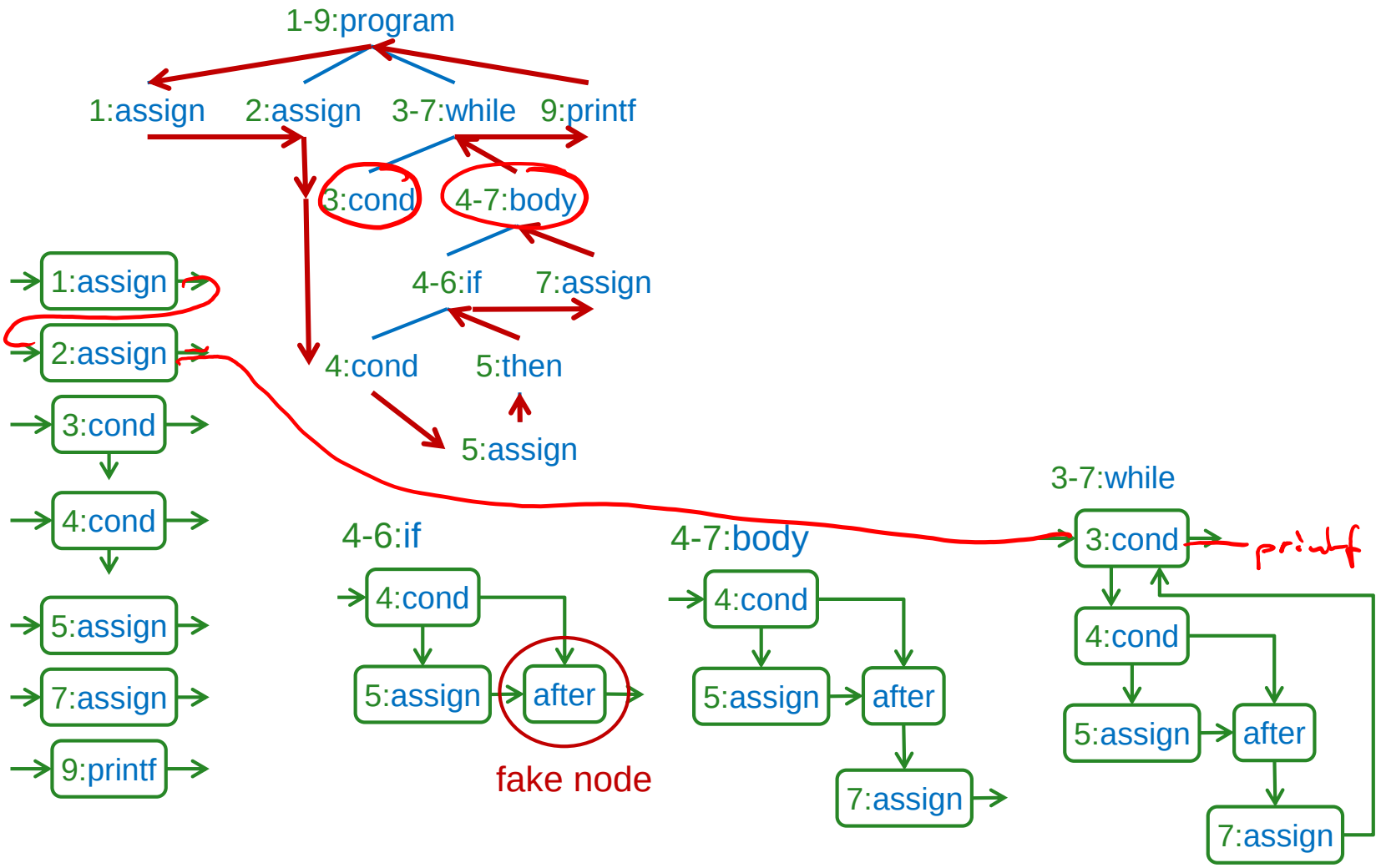


EXAMPLE CFG CONSTRUCTION (SYNTHESIZED)



EXAMPLE CFG CONSTRUCTION (SYNTHESIZED)





EXAMPLE CFG CONSTRUCTION (SYNTHESIZED)