COS320: Compiling Techniques

Zak Kincaid

May 2, 2019

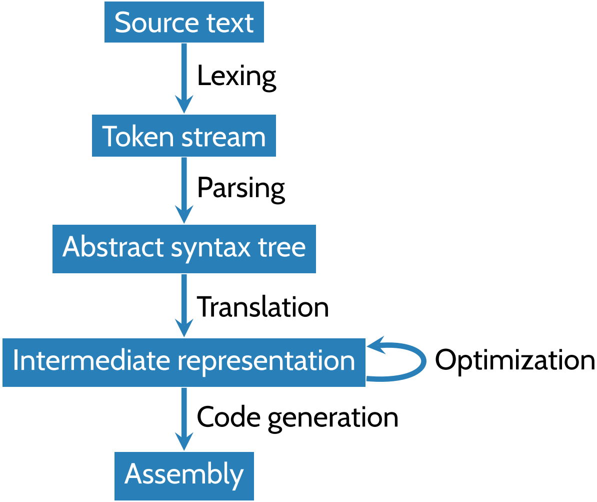
- HW6 is due on **Dean’s date, 5pm**.
- Final exam: Sunday May 19th 1pm in CS 104

Final Exam

- *Mostly* material since the midterm (LR parsing and up). Topics:
    - LR Parsing
    - Type systems (be comfortable reading inference rules, writing proof trees)
    - Data flow analysis (translate a global specification into local constraints)
    - Register allocation (graph coloring, coalescing)
    - Control flow analysis (dominators, loops, SSA conversion)
- Format similar to midterm
- Past COS320 exams @ Princeton & CIS341 exams @ UPenn are online

Review

Compiler phases (simplified)



Source text

Lexing

Token stream

Parsing

Abstract syntax tree

Translation

Intermediate representation

Optimization

Code generation

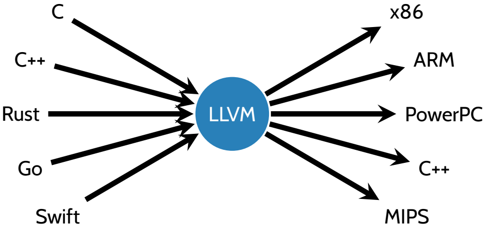
Assembly

Software engineering

- Compilers are large software projects
  - Decompose the problem into lots of small phases, each of which accomplishes
  - E.g., the optimization phase is also a large piece of software – it too is composed of lots of small individual phases
- Many problems do not have a “right” answer: pick a *convention*, document it well, and adhere to it.
  - E.g., calling conventions, pass environment as first argument to a closure, store pointer to dipatch vector in object, …

Intermediate representations

- An IR breaks code generation up into two phases. Simpler & easier to implement
- IRs (such as SSA) can drastically simplify optimization
- Makes compiler back-end re-usable



Lexing and parsing

- The lexing phase of a compiler breaks a stream of characters (source text) into a stream of *tokens*
- The parsing phase of a compiler takes in a stream of tokens (produced by a lexer), and builds an abstract syntax tree (AST).
- Lexing and parsing are based on *automata*
  - Lexing: finite automata (DFAs, NFAs)
  - Parsing: (deterministic) pushdown automata
- Useful tool to have in your toolbox!
  - Parsing useful for programming languages, domain specific languages, custom data formats, ...
  - Lexer generators: lex, flex, **ocamllex**, jflex
  - Parser generators: Yacc, Bison, ANTLR, **menhir**

Type Systems

- Specified by *inference rules*

$$\frac{J_1 \qquad J_2 \qquad \cdots \qquad J_n}{J}\ \textsc{Side-condition}$$

  - Succinct way to communicate a precise specification
  - Pervasive in formal logic and programming language theory. Can be used to specify
    - the semantics of programming languages
    - logics for reasoning about programs
    - program analyses
    - ...
- Type theory is a large subject and an active area of research
  - Close ties to logic (Curry-Howard correspondence: formulas are types, programs are proofs)
  - More in COS 510

Dataflow analysis

- Dataflow analysis is an approach to program analysis that unifies the presentation and implementation of many different analyses
  - Define a system of inequations $\{X_i \sqsupseteq R_i\}_{i \in I}$, where "unknowns" X_i are values in some partially orderd set, and right-hand-sides are monotone expressions over unknowns
  - Solve the system by repeatedly:
    1. Choosing a constraint $X_j \sqsupseteq R_j$ that is not satisfied
    2. Increasing X_j so that the constraint *is* satisfied

    until all constraints are satified
- Idea: can sometimes transform a global specification into a system of local constraints, which can be solved iteratively

LL parsing revisited

- LL(1) parser can be constructed from *nullable*, *first*, and *follow*, which have the following **global specifications**
  - Fix a grammar $G = (N, \Sigma, R, S)$
  - For any word $\gamma \in (N \cup \Sigma)^*$, define first$(\gamma) = \{a \in \Sigma : \gamma \Rightarrow^* aw\}$
  - For any word $\gamma \in (N \cup \Sigma)^*$, say that γ is nullable if $\gamma \Rightarrow^* \epsilon$
  - For any non-terminal A, define follow$(A) = \{a \in \Sigma : \exists \gamma, \gamma'. S \Rightarrow \gamma A a \gamma'\}$

LL parsing revisited

- LL(1) parser can be constructed from *nullable*, *first*, and *follow*, which have the following **global specifications**
    - Fix a grammar $G = (N, \Sigma, R, S)$
    - For any word $\gamma \in (N \cup \Sigma)^*$, define first$(\gamma) = \{a \in \Sigma : \gamma \Rightarrow^* aw\}$
    - For any word $\gamma \in (N \cup \Sigma)^*$, say that γ is nullable if $\gamma \Rightarrow^* \epsilon$
    - For any non-terminal A, define follow$(A) = \{a \in \Sigma : \exists \gamma, \gamma'. S \Rightarrow \gamma A a \gamma'\}$
- nullable : $N \to \{$*true*, *false*$\}$ (w/ *false* $\sqsubseteq$ *true*) is the *least function* such that
    - For each rule $A ::= \gamma_1 ... \gamma_n$, **nullable**$(A) \sqsupseteq$ **nullable**$(\gamma_1) \wedge \cdots \wedge$ **nullable**(γ_1)

LL parsing revisited

- LL(1) parser can be constructed from *nullable*, *first*, and *follow*, which have the following **global specifications**
  - Fix a grammar $G = (N, \Sigma, R, S)$
  - For any word $\gamma \in (N \cup \Sigma)^*$, define first$(\gamma) = \{a \in \Sigma : \gamma \Rightarrow^* aw\}$
  - For any word $\gamma \in (N \cup \Sigma)^*$, say that γ is nullable if $\gamma \Rightarrow^* \epsilon$
  - For any non-terminal A, define follow$(A) = \{a \in \Sigma : \exists \gamma, \gamma'. S \Rightarrow \gamma A a \gamma'\}$
- nullable : $N \to \{$*true*, *false*$\}$ (w/ *false* $\sqsubseteq$ *true*) is the *least function* such that
  - For each rule $A ::= \gamma_1 ... \gamma_n$, **nullable**$(A) \sqsupseteq$ **nullable**$(\gamma_1) \wedge \cdots \wedge$ **nullable**(γ_1)
- first is the *smallest function* such that
  - For each $a \in \Sigma$, **first**$(a) = \{a\}$
  - For each $A ::= \gamma_1 ... \gamma_i ... \gamma_n \in R$, with $\gamma_1, ..., \gamma_{i-1}$ nullable, **first**$(A) \supseteq$ **first**(γ_i)

LL parsing revisited

- LL(1) parser can be constructed from *nullable*, *first*, and *follow*, which have the following **global specifications**
  - Fix a grammar $G = (N, \Sigma, R, S)$
  - For any word $\gamma \in (N \cup \Sigma)^*$, define first$(\gamma) = \{a \in \Sigma : \gamma \Rightarrow^* aw\}$
  - For any word $\gamma \in (N \cup \Sigma)^*$, say that γ is nullable if $\gamma \Rightarrow^* \epsilon$
  - For any non-terminal A, define follow$(A) = \{a \in \Sigma : \exists \gamma, \gamma'. S \Rightarrow \gamma A a \gamma'\}$
- nullable : $N \to \{$*true*, *false*$\}$ (w/ *false* $\sqsubseteq$ *true*) is the *least function* such that
  - For each rule $A ::= \gamma_1 ... \gamma_n$, **nullable**$(A) \sqsupseteq$ **nullable**$(\gamma_1) \wedge \cdots \wedge$ **nullable**(γ_1)
- first is the *smallest function* such that
  - For each $a \in \Sigma$, **first**$(a) = \{a\}$
  - For each $A ::= \gamma_1 ... \gamma_i ... \gamma_n \in R$, with $\gamma_1, ..., \gamma_{i-1}$ nullable, **first**$(A) \supseteq$ **first**(γ_i)
- follow is the *smallest function* such that
  - For each $A ::= \gamma_1 ... \gamma_i ... \gamma_n \in R$, with $\gamma_{i+1}, ..., \gamma_n$ nullable, **follow**$(\gamma_i) \supseteq$ **follow**(A)
  - For each $A ::= \gamma_1 ... \gamma_i ... \gamma_j ... \gamma_n \in R$, with $\gamma_{i+1}, ..., \gamma_{j-1}$ nullable, **follow**$(\gamma_i) \supseteq$ **first**(A)

Current research

Conferences

- Programming Language Design and Implementation (PLDI)
- Principles of Programming Languages (POPL)
- Object Oriented Programming Systems, Languages & Applications (OOPSLA)
- Principles and Practice of Parallel Programming (PPoPP)
- Code Generation and Optimization (CGO)
- Compiler Construction (CC)
- International Conference on Functional Programming (ICFP)
- European Symposium on Programming (ESOP)
- Architectural Support for Programming Languages and Operating Systems (ASPLOS)

The job of a compiler is to translate from the syntax of one language to another, but preserve the semantics.

- Compiler correctness is *critical*
  - Trustworthiness of every component built in a compiled language depends on trustworthiness of the compiler
- Compilers tend to be well-engineered and well-tested, but that does not mean bug-free

Bug-finding in compilers

- CSmith[1]: randomized differential testing of C compilers
  - Randomly generate a C program *without undefined behavior*
    - Integrates program analysis to find interesting test cases
  - Compile with several different compilers
  - Compare the results
- Over 3 years found several real bugs
  - 79 bugs in GCC (25 maximum-priority/release-blocking)
  - 202 bugs in LLVM

---

[1] Yang et al. Finding and Understanding Bugs in C Compilers, PLDI 2011

Verified compilation

- *CompCert*: (Xavier Leroy, primary developer of OCaml)
  - Optimizing C compiler, implemented and proved correct in the Coq proof assistant
  - Coq proof assistant an (essentially) implementation of a sophisticated type system (CoIC)

> *The striking thing about our CompCert results is that the middle-end bugs we found in all other compilers are absent*
>
> *– Yang et al. Finding and Understanding Bugs in C Compilers, 2011*

Verified compilation

- *CompCert*: (Xavier Leroy, primary developer of OCaml)
  - Optimizing C compiler, implemented and proved correct in the Coq proof assistant
  - Coq proof assistant an (essentially) implementation of a sophisticated type system (CoIC)

> *The striking thing about our CompCert results is that the middle-end bugs we found in all other compilers are absent*
>
> *– Yang et al. Finding and Understanding Bugs in C Compilers, 2011*

- At Princeton: *CertiCoq* (Andrew Appel)
  - CompCert is implemented the proof assistant Coq… but why should we trust the Coq compiler?
  - CertiCoq is an optimizing compiler for Coq, implemented and verified in Coq.

Automatic parallelization

- Moore's law: processor advances double speed every 18 months
- (Proebsting's law: compiler advances double speed every 18 *years*)

Automatic parallelization

- Moore's law: processor advances double speed every 18 months
- (Proebsting's law: compiler advances double speed every 18 *years*)
- Moore's law ended in 2006 for single-threaded applications
    - Started to hit fundamental limits in how small transistors can be
- Processor manufacturers shifted to *multi-core* processors

Automatic parallelization

- Moore's law: processor advances double speed every 18 months
- (Proebsting's law: compiler advances double speed every 18 *years*)
- Moore's law ended in 2006 for single-threaded applications
    - Started to hit fundamental limits in how small transistors can be
- Processor manufacturers shifted to *multi-core* processors
- Need new compiler technology to take advantage of multi-core – automatically find and exploit opportunities for parallel execution
- At Princeton: David August's parallelization project

Program synthesis

- *Verification*: Given a program and a specification, prove that the program satisfies the specification
- *Synthesis*: Given a specification, find a program that satisfies the specification
- Superoptimization: find the least costly sequence of instructions that is equivalent to a given sequence
  - Specification is a program, but used as a black box
  - Solved by exhaustive search
  - Symbolic search (SAT,SMT), stochastic search (Markov-Chain Monte Carlo sampling)
- At Princeton: Synthesizing Lenses (David Walker), synthesis via logical games (Zak Kincaid)

Program analysis

- The goal of a program analysis is to answer questions about the run-time behavior of software
  - In compilers: data flow analysis, control flow analysis
  - Typical goal: determine whether an optimization is safe
- Research in program analysis has shifted to more sophisticated properties
  - Numerical analyses – e.g., find geometric regions that contain reachable values for integer variables. Can be used to verify absence of buffer overflows.
  - Shape analyses – determine whether a data structure in the heap is a list, a tree, a graph, … Can be used to verify memory safety.
  - Resource analyses – e.g., find a conservative upper bound on the run-time complexity of a loop. Can be used to find timing side-channel attacks.

Program analysis

- The goal of a program analysis is to answer questions about the run-time behavior of software
  - In compilers: data flow analysis, control flow analysis
  - Typical goal: determine whether an optimization is safe
- Research in program analysis has shifted to more sophisticated properties
  - Numerical analyses – e.g., find geometric regions that contain reachable values for integer variables. Can be used to verify absence of buffer overflows.
  - Shape analyses – determine whether a data structure in the heap is a list, a tree, a graph, … Can be used to verify memory safety.
  - Resource analyses – e.g., find a conservative upper bound on the run-time complexity of a loop. Can be used to find timing side-channel attacks.
- Industrial program analysis
  - Static Driver Verifier (Microsoft): finds bugs in device driver code
  - Infer (Facebook): proves memory safety & finds race conditions
  - Astrée (AbsInt): static analyzer for safety-critcal embedded code (e.g., automotive & aerospace applications)
  - Several commerical static analyzers: Codesonar, Coverity, PVS-Studio, Fortify, …

Program analysis at Princeton

- Synthesis, Learning, and Verification project (Aarti Gupta)
  - Idea: learn program invariants, termination arguments, etc from data
- My work on *algebraic program analysis*
  - Program analyses typically work by propagating information forwards through a program
    - Requires that we know the program's entry procedure
    - Analysis complexity is polynomial (or exponential, or worse) in program size
    - Changing one part of a codebase may change everything down-stream
  - We want analyses to be *compositional*
    - Analye the program by breaking it into parts, analyzing each part, and then combining the results

Algebraic program analysis

Consists of:

① **Semantic algebra** $\mathcal{D} = \langle D, \otimes, \oplus, *, 0, 1 \rangle$
  - D: Space of program properties
  - $\otimes : D \times D \to D$: sequencing operator
  - $\oplus : D \times D \to D$: choice operator
  - $* : D \to D$: iteration operator
  - $0, 1 \in D$: unit of $\oplus$, $\otimes$ respectively

② **Semantic function** $\mathcal{D}[\![\cdot]\!] : \mathit{Edge} \to D$

Algebraic program analysis

Consists of:

1. **Semantic algebra** $\mathcal{D} = \langle D, \otimes, \oplus, *, 0, 1 \rangle$
    - D: Space of program properties
    - $\otimes : D \times D \to D$: sequencing operator
    - $\oplus : D \times D \to D$: choice operator
    - $* : D \to D$: iteration operator
    - $0, 1 \in D$: unit of $\oplus$, $\otimes$ respectively
2. **Semantic function** $\to D$

L : Space of program properties
$\sqsubseteq \subseteq L \times L$: approximation order
$\sqcup : L \times L \to L$: join operator
$\bot \in L$: least element
$\mathcal{L}[\![\cdot]\!] :$ *Edge* $\to (L \to L)$

Algebraic program analysis

Consists of:

1. **Semantic algebra** $\mathcal{D} = \langle D, \otimes, \oplus, *, 0, 1 \rangle$
   - D: Space of program properties
   - $\otimes : D \times D \to D$: sequencing operator
   - $\oplus : D \times D \to D$: choice operator
   - $* : D \to D$: iteration operator
   - $0, 1 \in D$: unit of $\oplus$, $\otimes$ respectively
2. **Semantic function** $\mathcal{D}[\![\cdot]\!] : \textit{Edge} \to D$

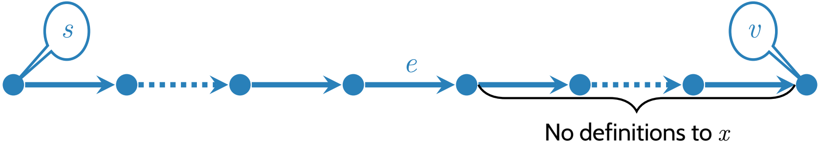
Analyze a program by evaluating its syntax in a semantic algebra

$$\begin{aligned} \mathcal{D}[\![S_1; S_2]\!] &= \mathcal{D}[\![S_1]\!] \otimes \mathcal{D}[\![S_2]\!] \\ \mathcal{D}[\![\textbf{if}(*)\{S_1\}\textbf{else}\{S_2\}]\!] &= \mathcal{D}[\![S_1]\!] \oplus \mathcal{D}[\![S_2]\!] \\ \mathcal{D}[\![\textbf{while}(*)\{S\}]\!] &= (\mathcal{D}[\![P]\!])^* \end{aligned}$$

Reaching definitions analysis

If a control flow edge e is an assignment `x := t`*, then we say that e is a **definition** that **defines** x.*

A definition e of a variable `x` reaches a vertex v if there exists a path from the root to v of the form:



Iterative reaching definitions:

- $L \triangleq 2^{\mathit{Def}}$
- $\mathcal{L}[\![e : x := t]\!](R) \triangleq (R \setminus \{e' : e' \text{ defines x}\}) \cup \{e\}$
- $R_1 \sqsubseteq R_2 \iff R_1 \subseteq R_2$
- $R_1 \sqcup R_2 \triangleq R_1 \cup R_2$
- $\bot \triangleq \emptyset$

Iterative reaching definitions:

- $L \triangleq 2^{\textit{Def}}$
- $\mathcal{L}[\![e : x := t]\!](R) \triangleq (R \setminus \{e' : e' \text{ defines x}\}) \cup \{e\}$
- $R_1 \sqsubseteq R_2 \iff R_1 \subseteq R_2$
- $R_1 \sqcup R_2 \triangleq R_1 \cup R_2$
- $\bot \triangleq \emptyset$

Algebraic reaching definitions :

- $D = (2^{\textit{Def}}) \times (2^{\textit{Def}})$
- $\mathcal{D}[\![e : x := t]\!] \triangleq (\{e\}, \{e' : e' \text{ defines x}\})$

Iterative reaching definitions:

- $L \triangleq 2^{\mathit{Def}}$
- $\mathcal{L}[\![e : x := t]\!](R) \triangleq (R \setminus \{e' : e' \text{ defines x}\}) \cup \{e\}$
- $R_1 \sqsubseteq R_2 \iff R_1 \subseteq R_2$
- $R_1 \sqcup R_2 \triangleq R_1 \cup R_2$
- $\bot \triangleq \emptyset$

Algebraic reaching definitions :

- $D = (2^{\mathit{Def}}) \times (2^{\mathit{Def}})$
- $\mathcal{D}[\![e : x := t]\!] \triangleq (\{e\}, \{e' : e' \text{ defines x}\})$
- $(G_1, K_1) \otimes (G_2, K_2) \triangleq ((G_1 \setminus K_2) \cup G_2, (K_1 \setminus G_2) \cup K_2)$

Iterative reaching definitions:

- $L \triangleq 2^{\mathit{Def}}$
- $\mathcal{L}[\![e : x := t]\!](R) \triangleq (R \setminus \{e' : e' \text{ defines x}\}) \cup \{e\}$
- $R_1 \sqsubseteq R_2 \iff R_1 \subseteq R_2$
- $R_1 \sqcup R_2 \triangleq R_1 \cup R_2$
- $\bot \triangleq \emptyset$

Algebraic reaching definitions :

- $D = (2^{\mathit{Def}}) \times (2^{\mathit{Def}})$
- $\mathcal{D}[\![e : x := t]\!] \triangleq (\{e\}, \{e' : e' \text{ defines x}\})$
- $(G_1, K_1) \otimes (G_2, K_2) \triangleq ((G_1 \setminus K_2) \cup G_2, (K_1 \setminus G_2) \cup K_2)$
- $(G_1, K_1) \oplus (G_2, K_2) \triangleq (G_1 \cup G_2, K_1 \cap K_2)$

Iterative reaching definitions:

- $L \triangleq 2^{\mathit{Def}}$
- $\mathcal{L}[\![e : x := t]\!](R) \triangleq (R \setminus \{e' : e' \text{ defines x}\}) \cup \{e\}$
- $R_1 \sqsubseteq R_2 \iff R_1 \subseteq R_2$
- $R_1 \sqcup R_2 \triangleq R_1 \cup R_2$
- $\bot \triangleq \emptyset$

Algebraic reaching definitions :

- $D = (2^{\mathit{Def}}) \times (2^{\mathit{Def}})$
- $\mathcal{D}[\![e : x := t]\!] \triangleq (\{e\}, \{e' : e' \text{ defines x}\})$
- $(G_1, K_1) \otimes (G_2, K_2) \triangleq ((G_1 \setminus K_2) \cup G_2, (K_1 \setminus G_2) \cup K_2)$
- $(G_1, K_1) \oplus (G_2, K_2) \triangleq (G_1 \cup G_2, K_1 \cap K_2)$
- $(G, K)^* \triangleq (G, \emptyset)$

```
        while(*){
          if(*){
$x_1$ :       x := 1;
$y_1$ :       y := 1;
          } else {
$y_2$ :       y := 2;
          }
        }
$x_0$ : x := 0;
```

```
        while(*){
          if(*){
$x_1$ :       x := 1; } $(\{x_1\}, \{x_1, x_0\})$
$y_1$ :       y := 1; } $(\{y_1\}, \{y_1, y_2\})$
          } else {
$y_2$ :       y := 2;
          }
        }
$x_0$ : x := 0;
```

```
        while(*){
          if(*){
$x_1$ :       x := 1;  }
$y_1$ :       y := 1;  } $(\{x_1, y_1\}, \{x_1, x_0, y_1, y_2\})$
          } else {
$y_2$ :       y := 2;
          }
        }
$x_0$ : x := 0;
```

```
          while(*){
            if(*){
$x_1$ :         x := 1;  } $(\{x_1, y_1\}, \{x_1, x_0, y_1, y_2\})$
$y_1$ :         y := 1;  }
            } else {
$y_2$ :         y := 2;  } $(\{y_2\}, \{y_1, y_2\})$
            }
          }
$x_0$ :   x := 0;
```

```
      while(*){
        if(*){
x1 :      x := 1;
y1 :      y := 1;
        } else {
y2 :      y := 2;
        }
      }
x0 :  x := 0;
```

$(\{x_1, y_1, y_2\}, \{y_1, y_2\})$

```
        while(*){
          if(*){
$x_1$ :       x := 1;
$y_1$ :       y := 1;
          } else {
$y_2$ :       y := 2;
          }
        }
$x_0$ : x := 0;
```

$(\{x_1, y_1, y_2\}, \emptyset)$

```
        while(*){
          if(*){
x1 :        x := 1;
y1 :        y := 1;
          } else {
y2 :        y := 2;
          }
        }
x0 :    x := 0;
```

$$\left.\begin{array}{l}\texttt{while(*)\{ \dots \}}\end{array}\right\}(\{x_1, y_1, y_2\}, \emptyset)$$

$$\left.\begin{array}{l}\texttt{x := 0;}\end{array}\right\}(\{x_0\}, \{x_0, x_1\})$$

```
        while(*){
          if(*){
x1 :        x := 1;
y1 :        y := 1;
          } else {
y2 :        y := 2;
          }
        }
x0 :    x := 0;
```

$(\{x_0, y_1, y_2\}, \{x_0, x_1\})$

Path expressions [Tarjan '81]

Let $G = \langle \textit{Loc}, \textit{Edge}, s\rangle$ be a control flow graph.
A *path expression* of G is a regular expression E over the alphabet *Edge* such that each word recognized by E corresponds to a path in G.

$$E, F \in \mathsf{RegExp}(G) ::= e \in \textit{Edge} \mid E + F \mid EF \mid E^* \mid 0 \mid 1$$

Path expressions [Tarjan '81]

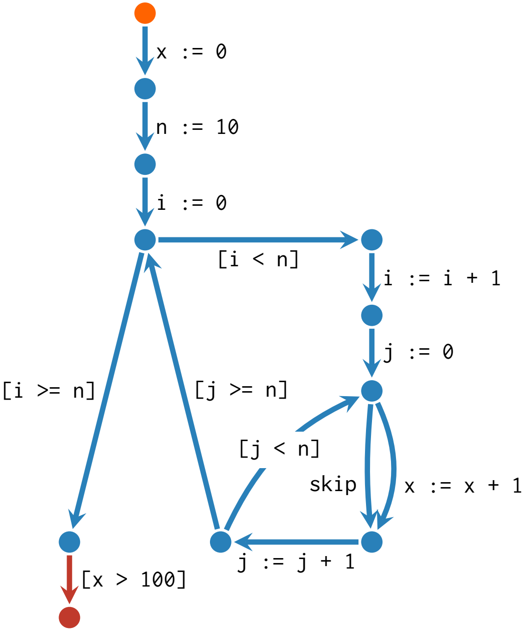
Let $G = \langle \textit{Loc}, \textit{Edge}, s\rangle$ be a control flow graph.
A *path expression* of G is a regular expression E over the alphabet *Edge* such that each word recognized by E corresponds to a path in G.

$$E, F \in \text{RegExp}(G) ::= e \in \textit{Edge} \mid E + F \mid EF \mid E^* \mid 0 \mid 1$$

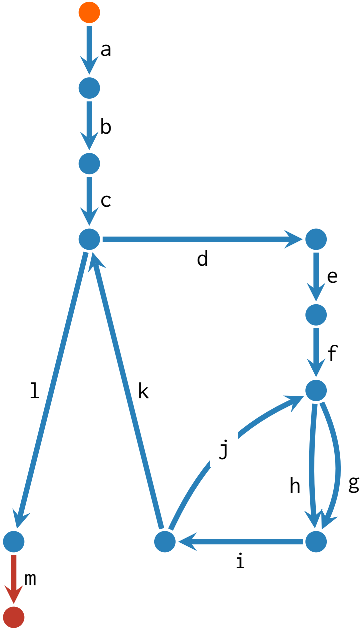
If $u, v \in \textit{Loc}$ are control locations, a *path expression from u to v* is a path expression that recognizes the set of all paths from u to v in G.

```
        x := 0
        n := 10
        i := 0
outer:  if(i >= n):
           goto end
        i := i + 1
inner:  j := 0
        if(*):
           x := x + 1
        j := j + 1
        if(j < n):
           goto inner
        goto outer
end:    assert(x <= 100)
```

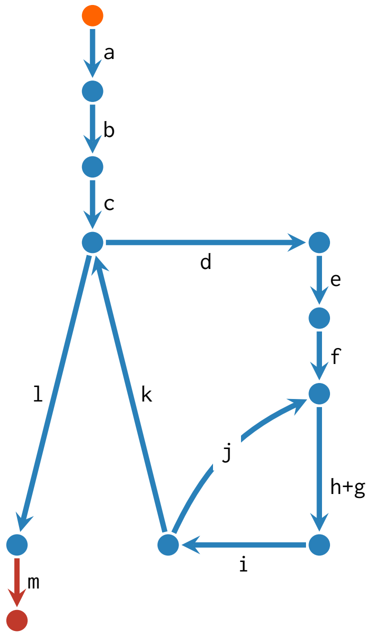
```
outer:  if(i >= n):
            goto end
        i := i + 1
inner:  j := 0
        if(*):
            x := x + 1
        j := j + 1
        if(j < n):
            goto inner
        goto outer
end:    assert(x <= 100)
```



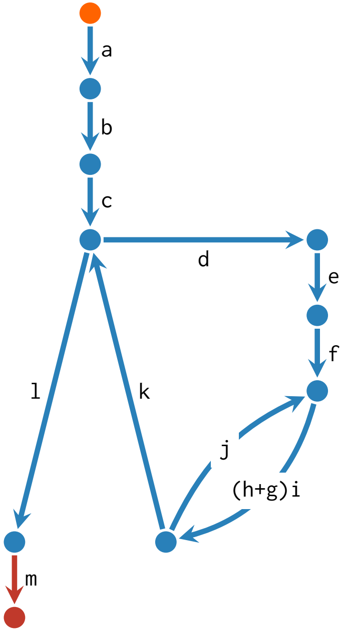
```
        x := 0
        n := 10
        i := 0
outer:  if(i >= n):
            goto end
        i := i + 1
inner:  j := 0
        if(*):
            x := x + 1
        j := j + 1
        if(j < n):
            goto inner
        goto outer
end:    assert(x <= 100)
```



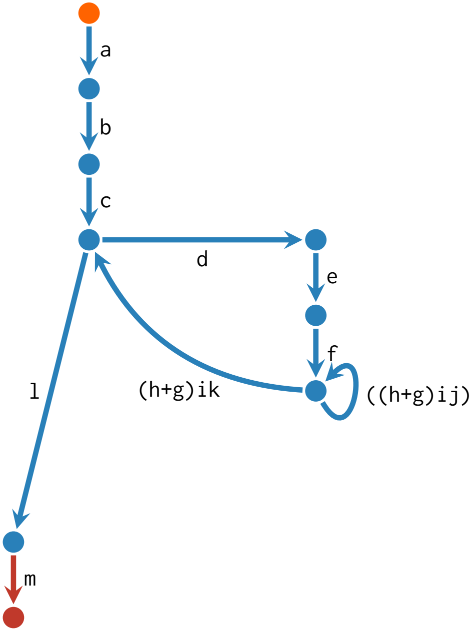
```
        x := 0
        n := 10
        i := 0
outer:  if(i >= n):
          goto end
        i := i + 1
inner:  j := 0
        if(*):
          x := x + 1
        j := j + 1
        if(j < n):
          goto inner
        goto outer
end:    assert(x <= 100)
```



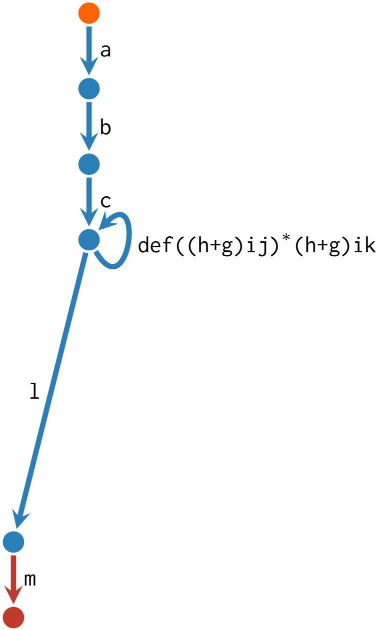
```
        x := 0
        n := 10
        i := 0
outer:  if(i >= n):
            goto end
        i := i + 1
inner:  j := 0
        if(*):
            x := x + 1
        j := j + 1
        if(j < n):
            goto inner
        goto outer
end:    assert(x <= 100)
```



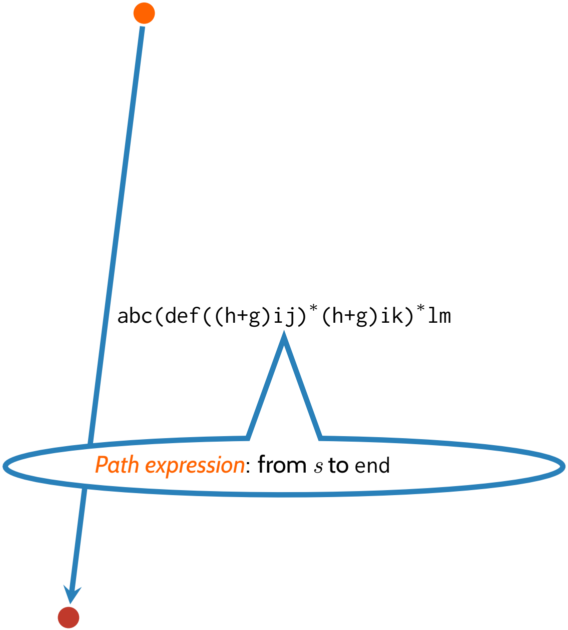
```
        x := 0
        n := 10
        i := 0
outer:  if(i >= n):
          goto end
        i := i + 1
inner:  j := 0
        if(*):
          x := x + 1
        j := j + 1
        if(j < n):
          goto inner
        goto outer
end:    assert(x <= 100)
```



```
        x := 0
        n := 10
        i := 0
outer:  if(i >= n):
            goto end
        i := i + 1
inner:  j := 0
        if(*):
            x := x + 1
        j := j + 1
        if(j < n):
            goto inner
        goto outer
end:    assert(x <= 100)
```



```
        x := 0
        n := 10
        i := 0
outer:  if(i >= n):
           goto end
        i := i + 1
inner:  j := 0
        if(*):
           x := x + 1
        j := j + 1
        if(j < n):
           goto inner
        goto outer
end:    assert(x <= 100)
```



$abc(def((h+g)ij)^*(h+g)ik)^*lm$

Path expression: from s to end

Running an algebraic program analysis

① Compute a *path expression* from the program entry to each vertex

② Evaluate the path expressions in the *semantic algebra* defining the analysis.

$$\begin{aligned}\mathcal{D}[\![S_1 S_2]\!] &= \mathcal{D}[\![S_1]\!] \otimes \mathcal{D}[\![S_2]\!]\\ \mathcal{D}[\![S_1 + S_2]\!] &= \mathcal{D}[\![S_1]\!] \oplus \mathcal{D}[\![S_2]\!]\\ \mathcal{D}[\![S^*]\!] &= (\mathcal{D}[\![P]\!])^*\end{aligned}$$

Running an algebraic program analysis

① Compute a *path expression* from the program entry to each vertex

② Evaluate the path expressions in the *semantic algebra* defining the analysis.

$$\begin{aligned} \mathcal{D}[\![S_1 S_2]\!] &= \mathcal{D}[\![S_1]\!] \otimes \mathcal{D}[\![S_2]\!] \\ \mathcal{D}[\![S_1 + S_2]\!] &= \mathcal{D}[\![S_1]\!] \oplus \mathcal{D}[\![S_2]\!] \\ \mathcal{D}[\![S^*]\!] &= (\mathcal{D}[\![P]\!])^* \end{aligned}$$

Tarjan's algorithm [Tarjan '81]: do both steps & avoid repeated work

What next?

- COS 375: Computer Architecture and Organization
- COS 326: Functional Programming
- COS 510: Programming Languages
- COS 516: Automated Reasoning about Software

Thanks!