COS320: Compiling Techniques

Zak & Arden Kincaid

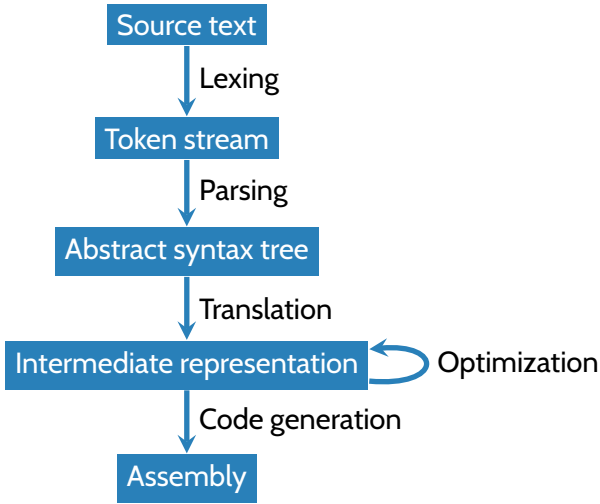
April 23, 2026

Logistics

- HW5 is due on **Dean's date (May 5), 5pm**.
- In-person office hours continue up until final exam.
  - Or schedule an appointment via email for virtual / in-person meeting.
- Final exam
  - Saturday May 9th, 8:30–11:30am in Friend Center 004
  - Same policy as midterm: open book/notes.
  - *Mostly* material since the midterm. Topics:
    - LL/LR parsing (construct LR(0) state machine, identify conflicts)
    - Type systems (reading inference rules, writing proof trees)
    - Data flow analysis (translate a global specification into local constraints)
    - Register allocation (construct, color, coalesce interference graphs)
    - Control flow analysis (identify dominators, loops, perform SSA conversion)
    - High-level languages (understand semantics of functional & object-oriented languages, garbage collection)


Revi
ew!

Compiler phases (simplified)

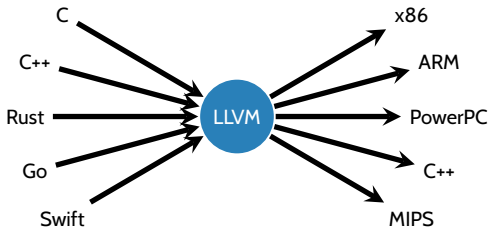


Software engineering

- 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, …
- Compilers are large software projects
  - Decompose the problem into lots of small phases, each of which accomplishes one thing
  - E.g., the optimization phase is also a large piece of software – it too is composed of lots of small individual phases

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 front-end/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

- 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

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 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 algorithms

Dataflow analysis

- Dataflow analysis is an approach to program analysis that unifies the presentation and implementation of many different algorithms
  - Define a system of inequations $\{X_i \sqsupseteq R_i\}_{i \in I}$, where "unknowns" X_i are values in some partially ordered set, and right-hand-sides are monotone expressions over unknowns

Dataflow analysis

- Dataflow analysis is an approach to program analysis that unifies the presentation and implementation of many different algorithms
  - Define a system of inequations $\{X_i \sqsupseteq R_i\}_{i \in I}$, where "unknowns" X_i are values in some partially ordered 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

Dataflow analysis

- Dataflow analysis is an approach to program analysis that unifies the presentation and implementation of many different algorithms
  - Define a system of inequations $\{X_i \sqsupseteq R_i\}_{i \in I}$, where "unknowns" X_i are values in some partially ordered 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

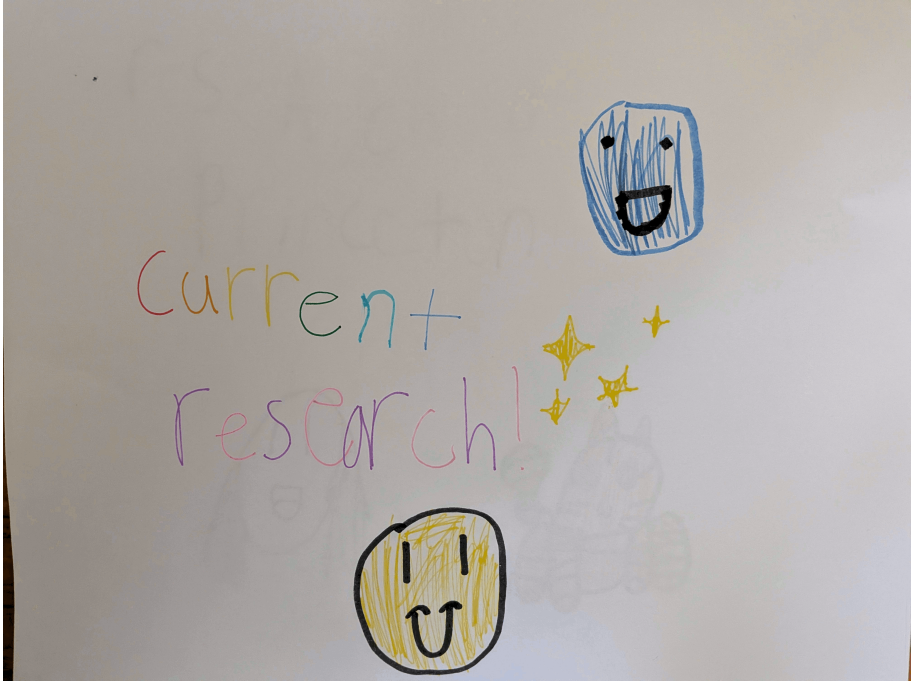
- LL(1) parser can be constructed from *nullable*, *first*, and *follow*, which have the following **global specifications**
  - Fix a grammar $G = (N, \Sigma, R, S)$
  - For any word $\gamma \in (N \cup \Sigma)^*$, define **first**$(\gamma) = \{a \in \Sigma : \gamma \Rightarrow^* aw\}$
  - For any word $\gamma \in (N \cup \Sigma)^*$, 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_n)

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_n)
- **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_n)
- **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)

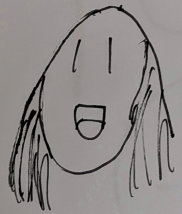
Recent themes

- Compiler correctness
  - Compiler correctness is *critical* – trustworthiness of every component built in a compiled language depends on trustworthiness of the compiler
  - *CSmith*[1]: randomized differential testing of C compilers
  - *CompCert*:[2]: Verified C compiler in Rocq
- Compilers for AI & Specialized hardware
  - Moore's law: processor advances double speed every 18 months
  - Moore's law ended in 2006 for single-threaded applications – performance gains are now through specialized hardware (GPUs, TPUs, NPUs)
  - *XLA* (Google) – global optimization of computation graphs + code generation for CPU/GPU/TPU.

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

[2] Leroy et al. CompCert - A Formally Verified Optimizing Compiler,

research of Princeton!



- David August's parallelization project
  - Need new compiler technology to take advantage of multi-core – automatically find and exploit opportunities for parallel execution
- Mae Milano's work on fearless concurrency
  - Use type system to guarantee absence of data races – integrated in Swift
- Aarti Gupta's Synthesis, Learning, and Verification project
  - Idea: learn program invariants, termination arguments, etc from data

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

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, divide-by-zero, ….
  - Shape analyses – determine whether a data structure in the heap is a list, a tree, a graph, … Can be used to verify memory safety.
  - Termination analyses – e.g., find a sufficient precondition under which a procedure is sure to terminate.

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, divide-by-zero, ...
  - Shape analyses – determine whether a data structure in the heap is a list, a tree, a graph, ... Can be used to verify memory safety.
  - Termination analyses – e.g., find a sufficient precondition under which a procedure is sure to terminate.
- Industrial program analysis
  - Static Driver Verifier (Microsoft): finds bugs in device driver code
  - Infer (Meta): 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, ...

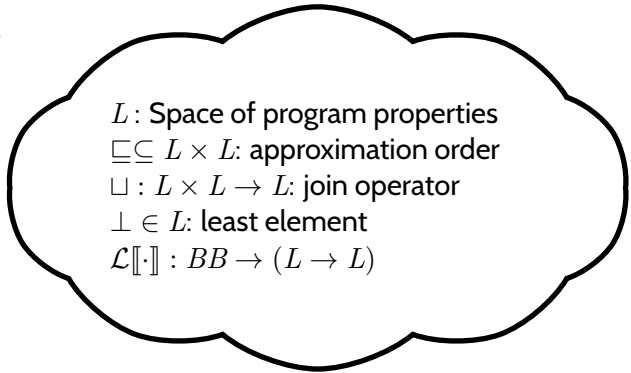
My work

- *Compositional* program analysis
  - Program analyses typically work by propagating information forwards through a program
  - Compositional analysis: Analyze the program by breaking it into parts, analyzing each part, and then combining the results
- *Robust* program analysis
  - Program analyzers rely on *heuristics* for predicting program behavior. Can be brittle and have unexpected behavior.
  - Software developers should be able to predict how a change in their program affects the results of an analysis

Algebraic program analysis

Consists of:

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



Algebraic program analysis

Consists of:

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

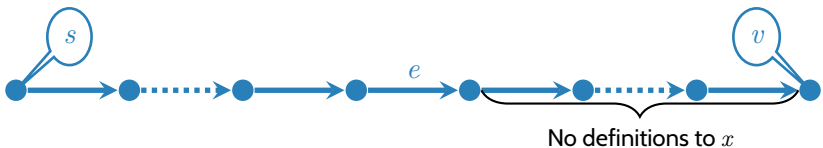
Analyze a program by evaluating its syntax in a semantic algebra

$$\mathcal{D}[\![S_1; S_2]\!] = \mathcal{D}[\![S_1]\!] \cdot \mathcal{D}[\![S_2]\!]$$
$$\mathcal{D}[\![\textbf{if}(*)\{S_1\}\textbf{else}\{S_2\}]\!] = \mathcal{D}[\![S_1]\!] + \mathcal{D}[\![S_2]\!]$$
$$\mathcal{D}[\![\textbf{while}(*)\{S\}]\!] = (\mathcal{D}[\![P]\!])^*$$

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:

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

Iterative reaching definitions:

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

Algebraic reaching definitions :

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

Iterative reaching definitions:

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

Algebraic reaching definitions :

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

Iterative reaching definitions:

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

Algebraic reaching definitions :

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

Iterative reaching definitions:

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

Algebraic reaching definitions :

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

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

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

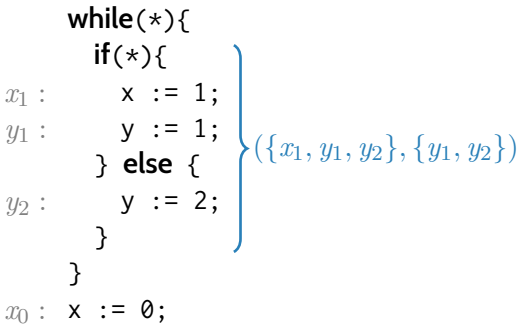
```
     while(*){
       if(*){
x1 :     x := 1;
y1 :     y := 1;
       } else {
y2 :     y := 2;
       }
     }
x0 : x := 0;
```

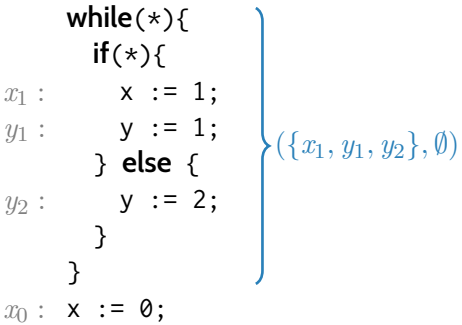
$(\{x_1, y_1\}, \{x_1, x_0, y_1, y_2\})$

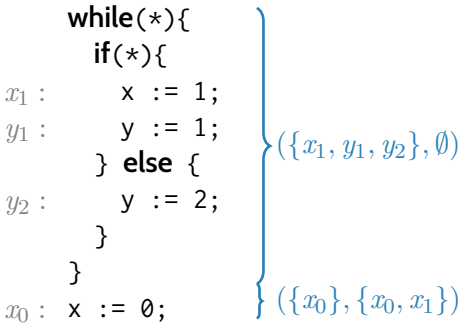
```
    while(*){
      if(*){
x1 :    x := 1;
y1 :    y := 1;
      } else {
y2 :    y := 2; }
      }
    }
x0 : x := 0;
```

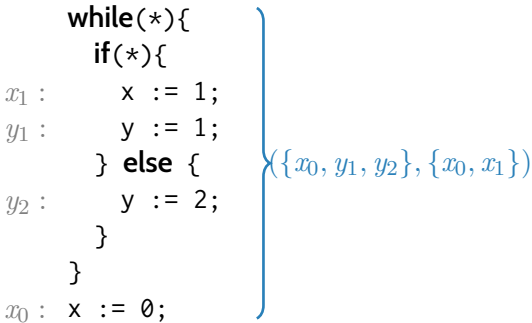
$(\{x_1, y_1\}, \{x_1, x_0, y_1, y_2\})$

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


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


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


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

Path expressions [Tarjan '81]

- Can generalize from structured programming languages to control flow graphs using path expressions
- 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.

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

- Idea: rather than interpret *program* over an algebra, interpret a *path expression*
  - Tarjan's single-source all-destination path expression algorithm can perform the analysis in nearly-linear time

Transition Formula Algebras

- A *transition formula* is a logical formula reprsenting an input/output relation of a program
  - e.g., $x := x + y \rightsquigarrow x' = x + y \wedge y' = y$
- Semantic algebra for transition formulas over variables X:

$$0^{\mathsf{TF}} \triangleq \mathit{false} \qquad \text{Empty relation}$$

$$1^{\mathsf{TF}} \triangleq \bigwedge_{x \in X} x' = x \qquad \text{Identity relation}$$

Transition Formula Algebras

- A *transition formula* is a logical formula reprsenting an input/output relation of a program
  - e.g., $x := x + y \rightsquigarrow x' = x + y \wedge y' = y$
- Semantic algebra for transition formulas over variables X:

$$0^{\mathsf{TF}} \triangleq \mathit{false} \qquad \text{Empty relation}$$

$$1^{\mathsf{TF}} \triangleq \bigwedge_{x \in X} x' = x \qquad \text{Identity relation}$$

$$F +^{\mathsf{TF}} G \triangleq F \vee G \qquad \text{Union}$$

Transition Formula Algebras

- A *transition formula* is a logical formula reprsenting an input/output relation of a program
  - e.g., $x := x + y \rightsquigarrow x' = x + y \wedge y' = y$
- Semantic algebra for transition formulas over variables X:

$$0^{\mathsf{TF}} \triangleq \mathit{false} \qquad \text{Empty relation}$$

$$1^{\mathsf{TF}} \triangleq \bigwedge_{x \in X} x' = x \qquad \text{Identity relation}$$

$$F +^{\mathsf{TF}} G \triangleq F \vee G \qquad \text{Union}$$

$$F \cdot^{\mathsf{TF}} G \triangleq \exists X''. F(X, X'') \wedge G(X'', X') \qquad \text{Relational composition}$$

Transition Formula Algebras

- A *transition formula* is a logical formula reprsenting an input/output relation of a program
  - e.g., $x := x + y \rightsquigarrow x' = x + y \wedge y' = y$
- Semantic algebra for transition formulas over variables X:

$$0^{\mathsf{TF}} \triangleq \mathit{false} \qquad \text{Empty relation}$$

$$1^{\mathsf{TF}} \triangleq \bigwedge_{x \in X} x' = x \qquad \text{Identity relation}$$

$$F +^{\mathsf{TF}} G \triangleq F \vee G \qquad \text{Union}$$

$$F \cdot^{\mathsf{TF}} G \triangleq \exists X''. F(X, X'') \wedge G(X'', X') \qquad \text{Relational composition}$$

$$F^{*\mathsf{TF}} \triangleq \ldots \qquad \text{Approximate transitive closure}$$

Transition Formula Algebras

- A *transition formula* is a logical formula reprsenting an input/output relation of a program
  - e.g., $x := x + y \rightsquigarrow x' = x + y \wedge y' = y$
- Semantic algebra for transition formulas over variables X:

$$0^{\mathsf{TF}} \triangleq \mathit{false} \qquad \text{Empty relation}$$

$$1^{\mathsf{TF}} \triangleq \bigwedge_{x \in X} x' = x \qquad \text{Identity relation}$$

$$F +^{\mathsf{TF}} G \triangleq F \vee G \qquad \text{Union}$$

$$F \cdot^{\mathsf{TF}} G \triangleq \exists X''. F(X, X'') \wedge G(X'', X') \qquad \text{Relational composition}$$

$$F^{*\mathsf{TF}} \triangleq \ldots \qquad \text{Approximate transitive closure}$$

Many different implementations

Iteration

$$(-)^* : \mathbf{TF} \to \mathbf{TF}$$

- Input: transition formula summarizing loop body
  - Regardless of structure of inner loop (nested loops, procedure calls, ...)
- Output: transition formula summarizing loop
  - Output language is the same as input language!

Example: induction variable analysis

- Suppose F is a transition formula for a loop body.
  - $x \in X$ is an induction variable *iff* F implies $x' = x + c$ for some constant c

Example: induction variable analysis

- Suppose F is a transition formula for a loop body.
  - $x \in X$ is an induction variable *iff* F implies $x' = x + c$ for some constant c
  - Satisfiability Modulo Theories (SMT): Find a satisfying interpretation of formula F (if one exists).
    - First, find a satisfying interpretation m of F
    - For each variable x, check if $F \wedge x' = x + m(x)$ is sat. Unsat $\iff$ x an induction var

Example: induction variable analysis

- Suppose F is a transition formula for a loop body.
  - $x \in X$ is an induction variable *iff* F implies $x' = x + c$ for some constant c
  - Satisfiability Modulo Theories (SMT): Find a satisfying interpretation of formula F (if one exists).
    - First, find a satisfying interpretation m of F
    - For each variable x, check if $F \wedge x' = x + m(x)$ is sat. Unsat $\iff$ x an induction var
  - Let Y be the set of induction variables. Define $F^* = \exists k.k \geq 0 \wedge \bigwedge_{y \in Y} y' = y + k \cdot m(y)$

Example: induction variable analysis

- Suppose F is a transition formula for a loop body.
  - $x \in X$ is an induction variable *iff* F implies $x' = x + c$ for some constant c
  - Satisfiability Modulo Theories (SMT): Find a satisfying interpretation of formula F (if one exists).
    - First, find a satisfying interpretation m of F
    - For each variable x, check if $F \wedge x' = x + m(x)$ is sat. Unsat $\iff$ x an induction var
  - Let Y be the set of induction variables. Define $F^* = \exists k.k \geq 0 \wedge \bigwedge_{y \in Y} y' = y + k \cdot m(y)$
  - Generalizing: find & solve *linear recurrence inequations*

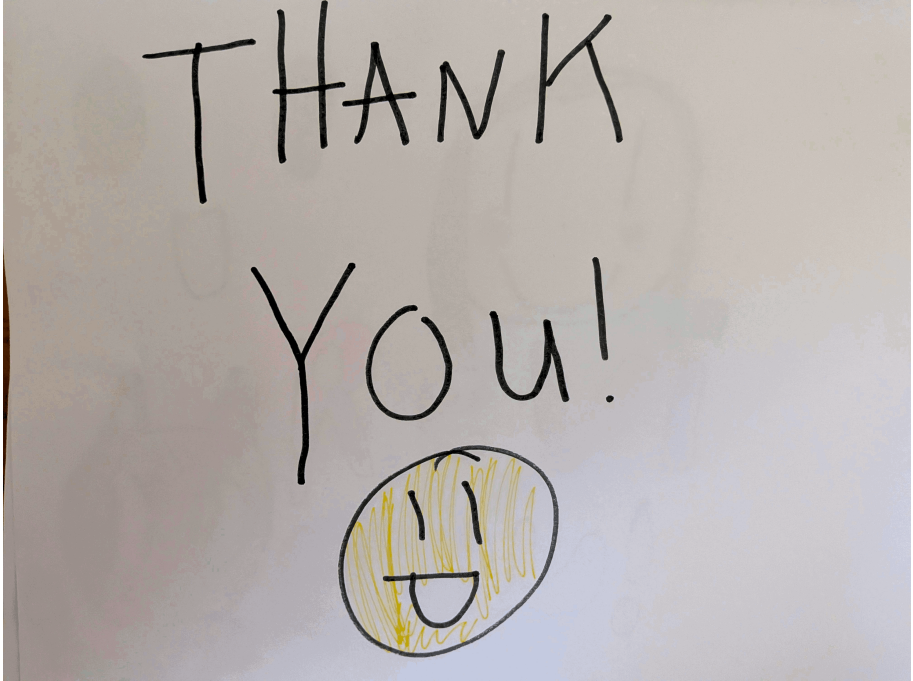
```
while(lo < hi) {
  if(a[hi] < pivot) {
    swap a[hi] and a[lo]
    lo ++;
  } else hi --;
}
```

$\rightsquigarrow$

$$lo < hi \wedge pivot' = pivot$$
$$\wedge \left(\begin{array}{l} \left(\begin{array}{l} a[hi] < pivot \\ \wedge a' = a\{hi \mapsto a[lo]\}\{lo \mapsto a[hi]\} \\ \wedge lo' = lo + 1 \\ \wedge hi' = hi \end{array} \right) \\ \vee \left(\begin{array}{l} \neg(a[hi] < pivot) \\ a' = a \\ \wedge lo' = lo \\ \wedge hi' = hi - 1 \end{array} \right) \end{array} \right) \rightsquigarrow \begin{array}{l} lo \leq lo' \leq lo + 1 \\ hi - 1 \leq hi' \leq hi \\ hi - lo = hi - lo - 1 \\ pivot' = pivot \end{array}$$

What next?

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


THANK
You!