EECS 598-008 & EECS 498-008: Intelligent Programming Systems

Lecture 2

Announcements

- A0 due **midnight Monday, September 6**
- A0 tutorial released on Canvas (see zoom recordings)
- Friday discussion on September 3 converted to remote office hour

Today’s Agenda

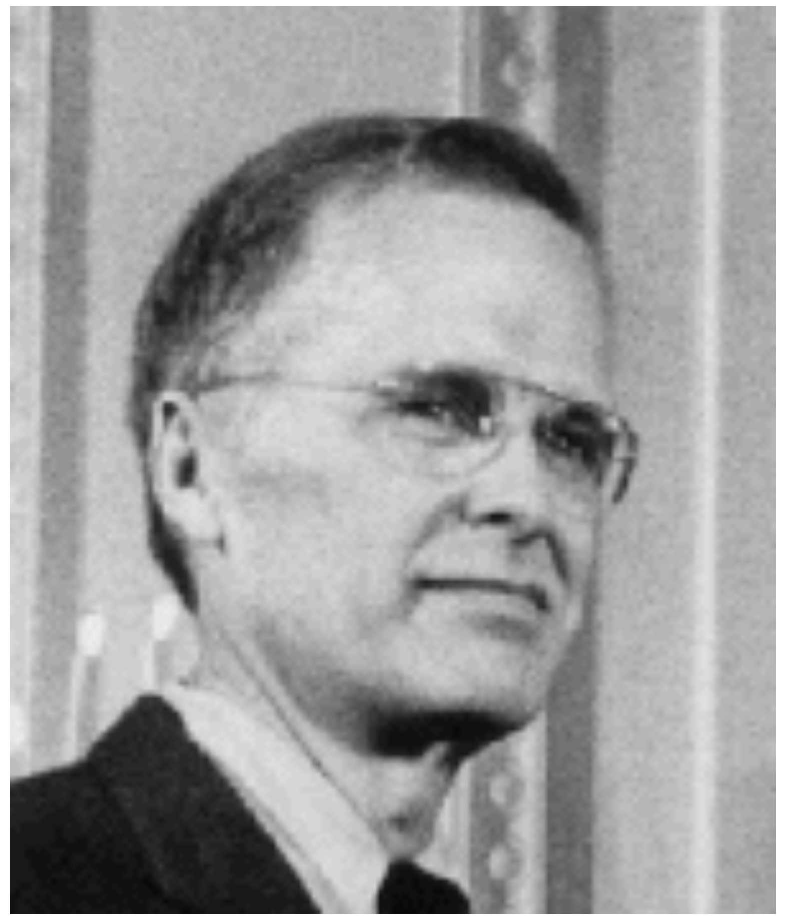
- **History of Program Synthesis**
- Syntax-Guided Synthesis (SYGUS)
- Context-Free Grammars (CFGs)

What is program synthesis?

1950’s - 1990’s

1950's: Fortran

John Backus



Much of my work has come from being lazy. I didn't like writing programs, and so, when I was working on the IBM 701, writing programs for computing missile trajectories, I started work on a programming system to make it easier to write programs.

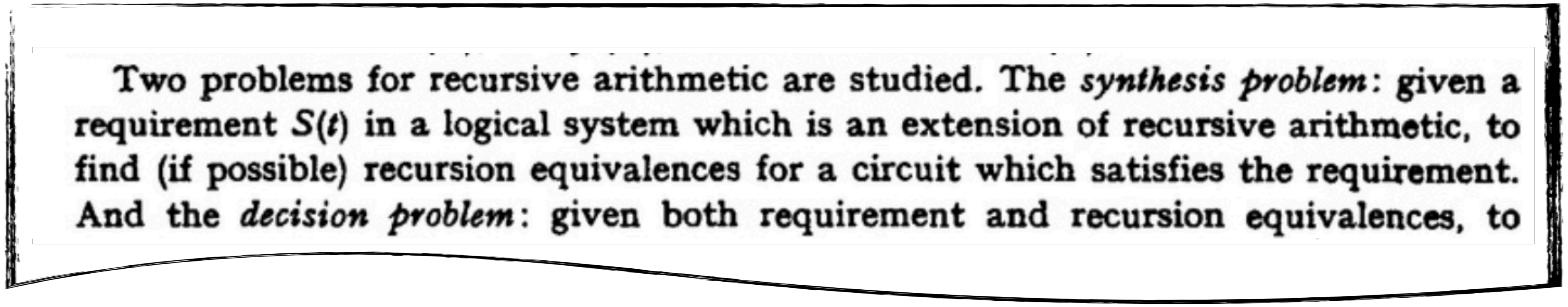
Essentially, compilation!

Backus et al., *The FORTRAN Automatic Coding System*, 1957

1950's, 1960's: Church's Synthesis Problem

Ongoing/Reactive Programs

Alonzo Church

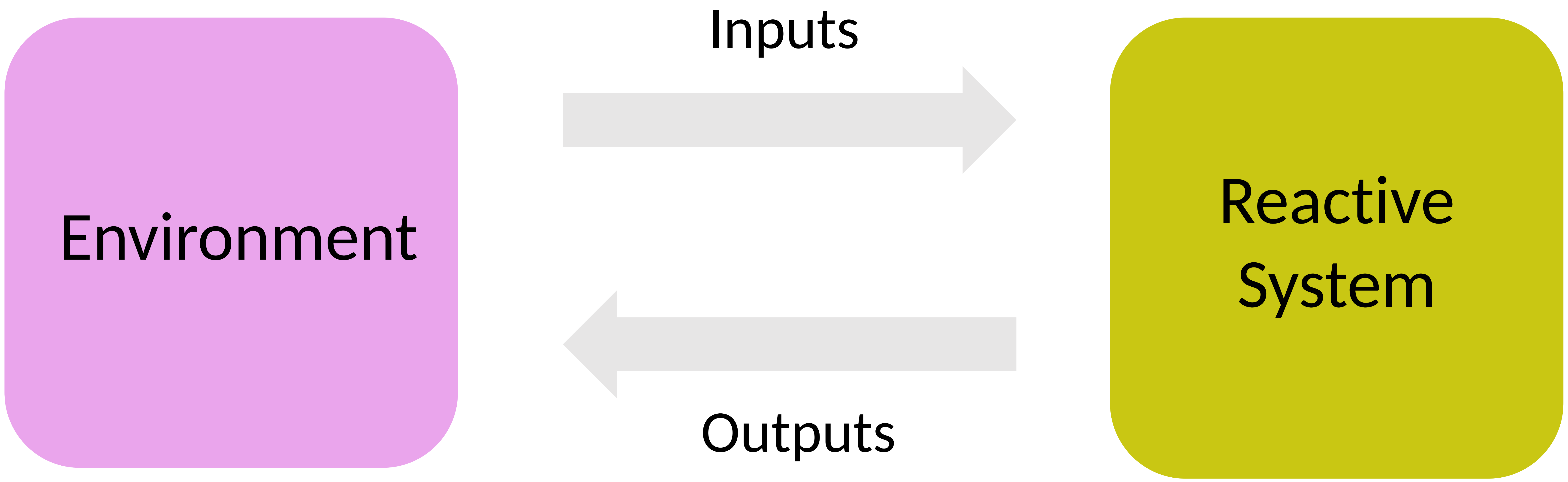

Two problems for recursive arithmetic are studied. The *synthesis problem*: given a requirement $S(t)$ in a logical system which is an extension of recursive arithmetic, to find (if possible) recursion equivalences for a circuit which satisfies the requirement. And the *decision problem*: given both requirement and recursion equivalences, to

Programs represented as circuits/finite automata

Church, *Application of Recursive Arithmetic to the Problem of Circuit Synthesis*, 1957

Church, Logic, Arithmetic and Automata, 1962

The goal of reactive synthesis is to generate a reactive system whose behavior satisfies a temporal specification, **in the presence of continuous interaction with an environment**



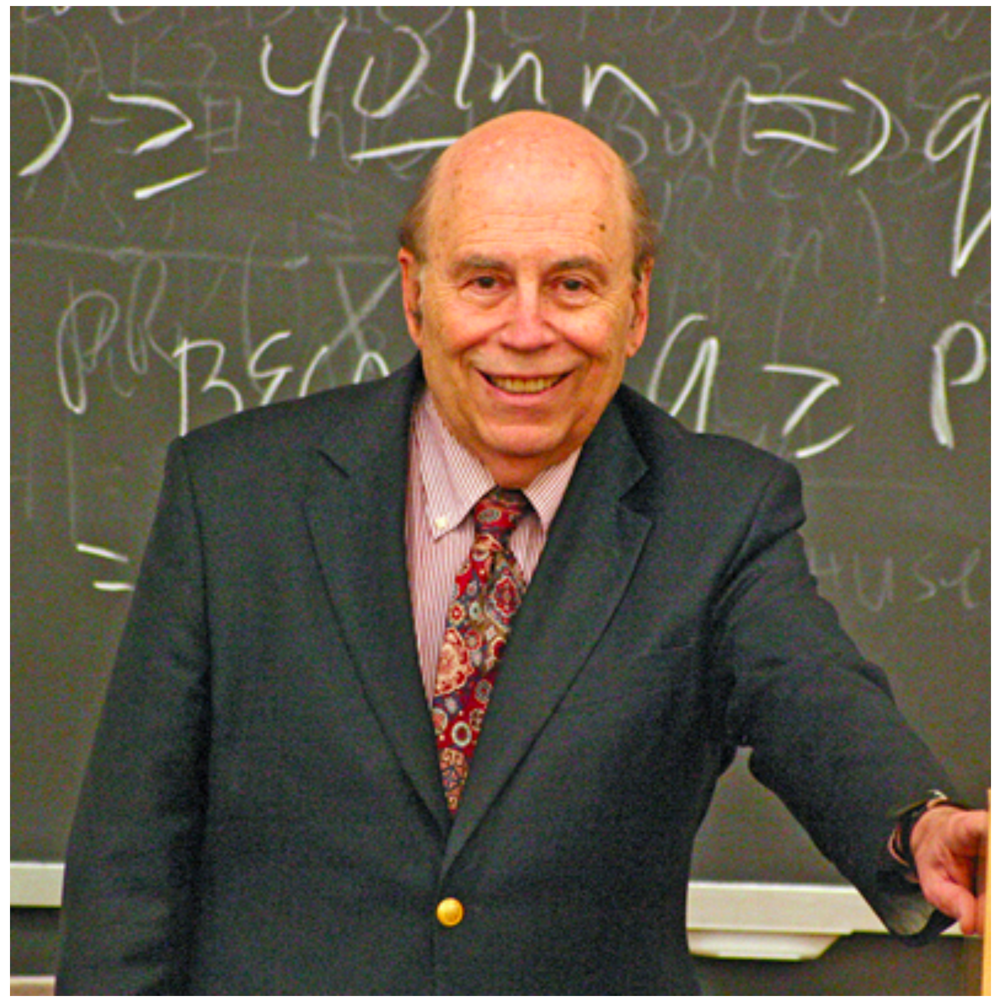
Slide from Roopsha Samanta

1960’s, 1970’s: Church’s Synthesis Problem Solved!

Julius Richard Buchi

Lawrence Landwebber

Michael O. Rabin



Extract program from finite-state winning strategy of an infinite two-player game

Buchi and Landwebber, *Solving Sequential Conditions by Finite-State Strategies*, 1967

Rabin, *Automata on infinite objects and Church’s Problem*, 1972

1960's, 1970's: Deductive Synthesis

Transformational/Functional Programs

Cordell Green

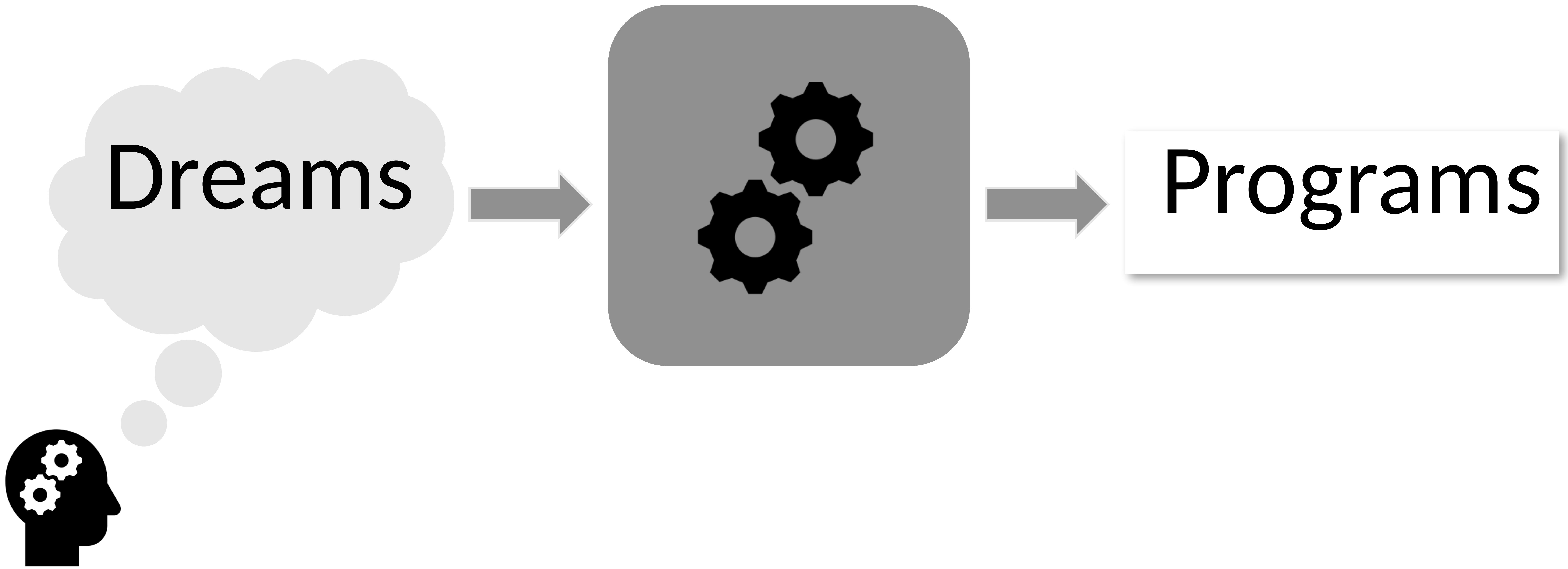
Zohar Manna

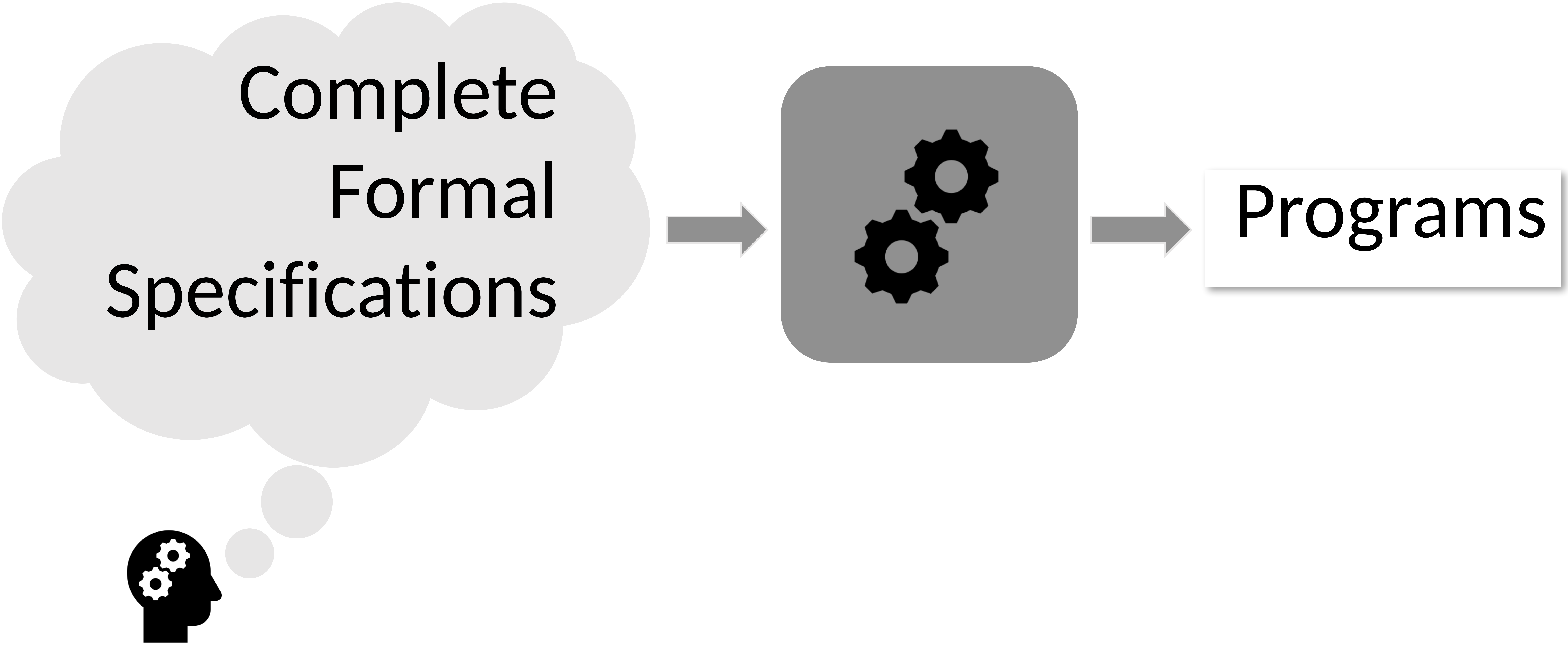
Richard Waldinger

Extract LISP-y program from proof of satisfiability of formal specification

Green, *Application of Theorem Proving to Problem Solving*, 1963

Manna and Waldinger, *Synthesis: Dreams ⇒ Programs*, 1979


Dreams
Programs


Complete
Formal
Specifications
Programs

$$\forall x,y,z\,.\; x \le \max(x,y,z) \;\wedge\; y \le \max(x,y,z) \;\wedge\; z \le \max(x,y,z) \;\wedge\; (\max(x,y,z)=x \;\vee\; \max(x,y,z)=y \;\vee\; \max(x,y,z)=z)$$

Easier?

Easier?

```
int max (int x,int y,int z)
 int m = z;
 if (z <= y) m = y;
 if (m <  x) m = x;
 return m;
```

Easier!

$(0,\ 10,\ 2) \mapsto 10$

$(-1,\ 10,\ 20) \mapsto 20$

$(-1,\ -2,\ -3) \mapsto -1$

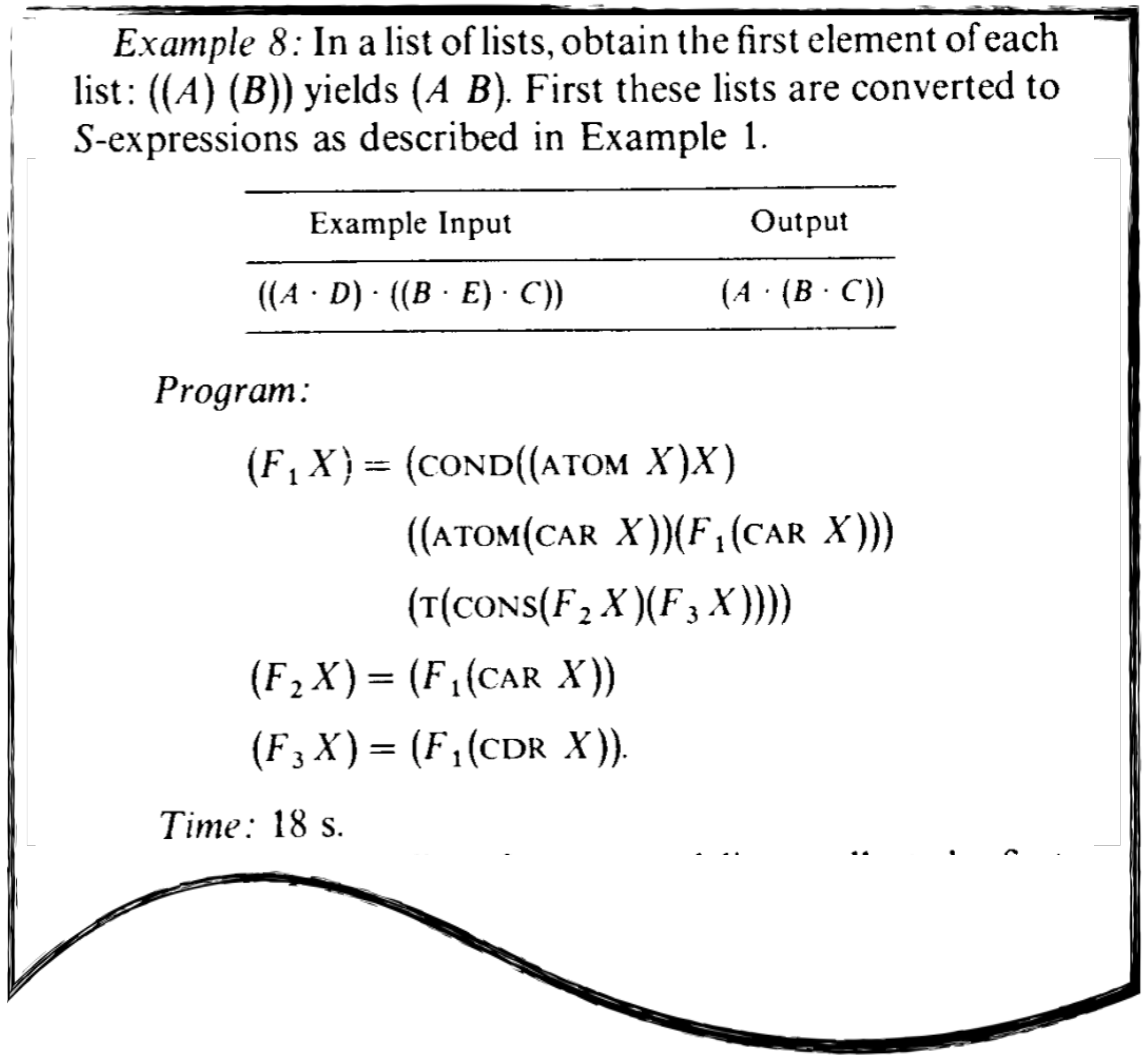
```
int max (int x,int y,int z)
 int m = z;
 if (z <= y) m = y;
 if (m <  x) m = x;
 return m;
```

1970’s: Inductive Programming

Transformational Programs

Summers, *A Methodology for LISP Program Construction from Examples*, 1977

Biermann, *Inference of Regular LISP Programs from Examples*, 1978



Example 8: In a list of lists, obtain the first element of each list: $((A)\ (B))$ yields $(A\ B)$. First these lists are converted to S-expressions as described in Example 1.

Example Input	Output
$((A \cdot D) \cdot ((B \cdot E) \cdot C))$	$(A \cdot (B \cdot C))$

Program:

$$(F_1\ X) = (\text{COND}((\text{ATOM}\ X)X)\ ((\text{ATOM}(\text{CAR}\ X))(F_1(\text{CAR}\ X)))\ (\text{T}(\text{CONS}(F_2\ X)(F_3\ X))))$$

$$(F_2\ X) = (F_1(\text{CAR}\ X))$$

$$(F_3\ X) = (F_1(\text{CDR}\ X)).$$

Time: 18 s.

1980's: Synthesis of Reactive Programs

Clarke

Emerson

Extract program (model) from algorithmically-constructed witness to satisfiability of formal specification.

Clarke & Emerson, *Design and Synthesis of Synchronization Skeletons using Branching-Time Temporal Logic, 1981*

1980's: Synthesis of Reactive Programs

Pnueli

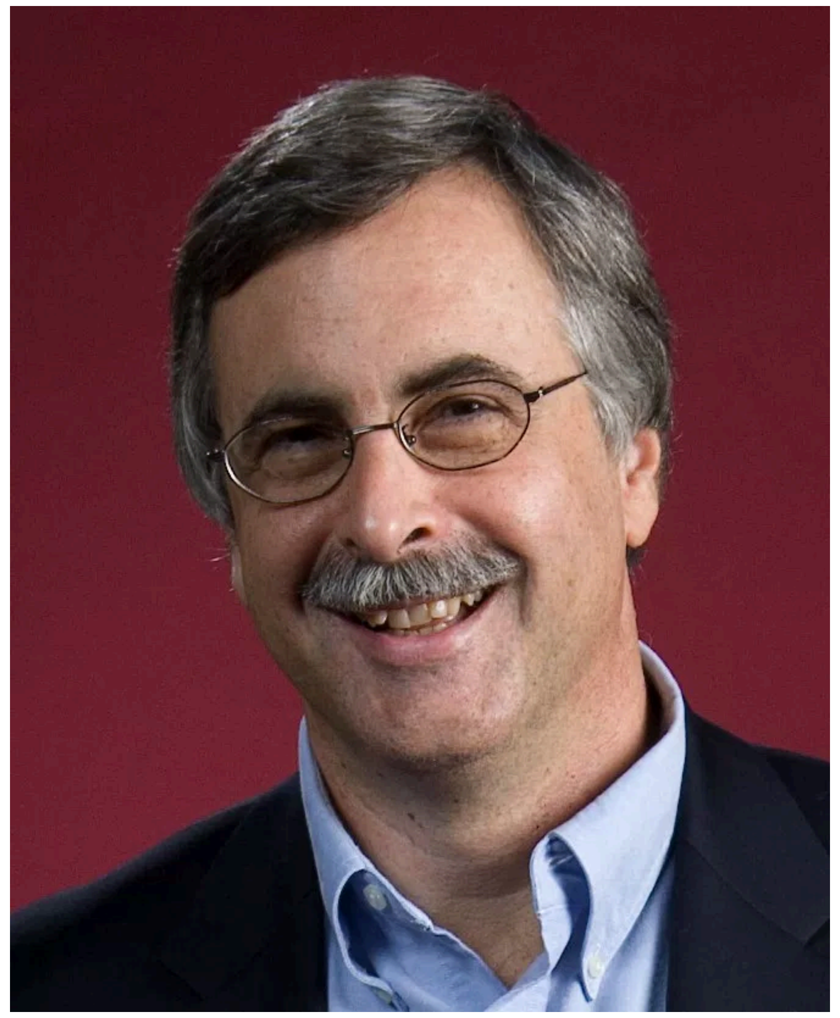
Better algorithms than Buchi, Landwebber, Rabin

Still an active research area!

Pnueli & Rosner, *On the Synthesis of a Reactive Module, 1989*

1980's: Programmer's Apprentice

Charles Rich



Richard C. Waters

Assist, not replace!

- Codify expert knowledge on how to solve programming problems
- User guided synthesis

Rich and Waters, *Programmer's Apprentice*, MIT 1987

1990's: Inductive Learning

Transformational Programs

Tessa Lau

Replaced ad-hoc approaches for PBE/PBD with techniques based on version space generalization and inductive logic programming

Lau and Weld, *Programming by Demonstration: An Inductive Learning Framework*, 1998

Post 2000: Modern Program Synthesis

Transformational program synthesis: A *search* problem

Input-output examples

Logical specifications

Equivalent programs

Natural language

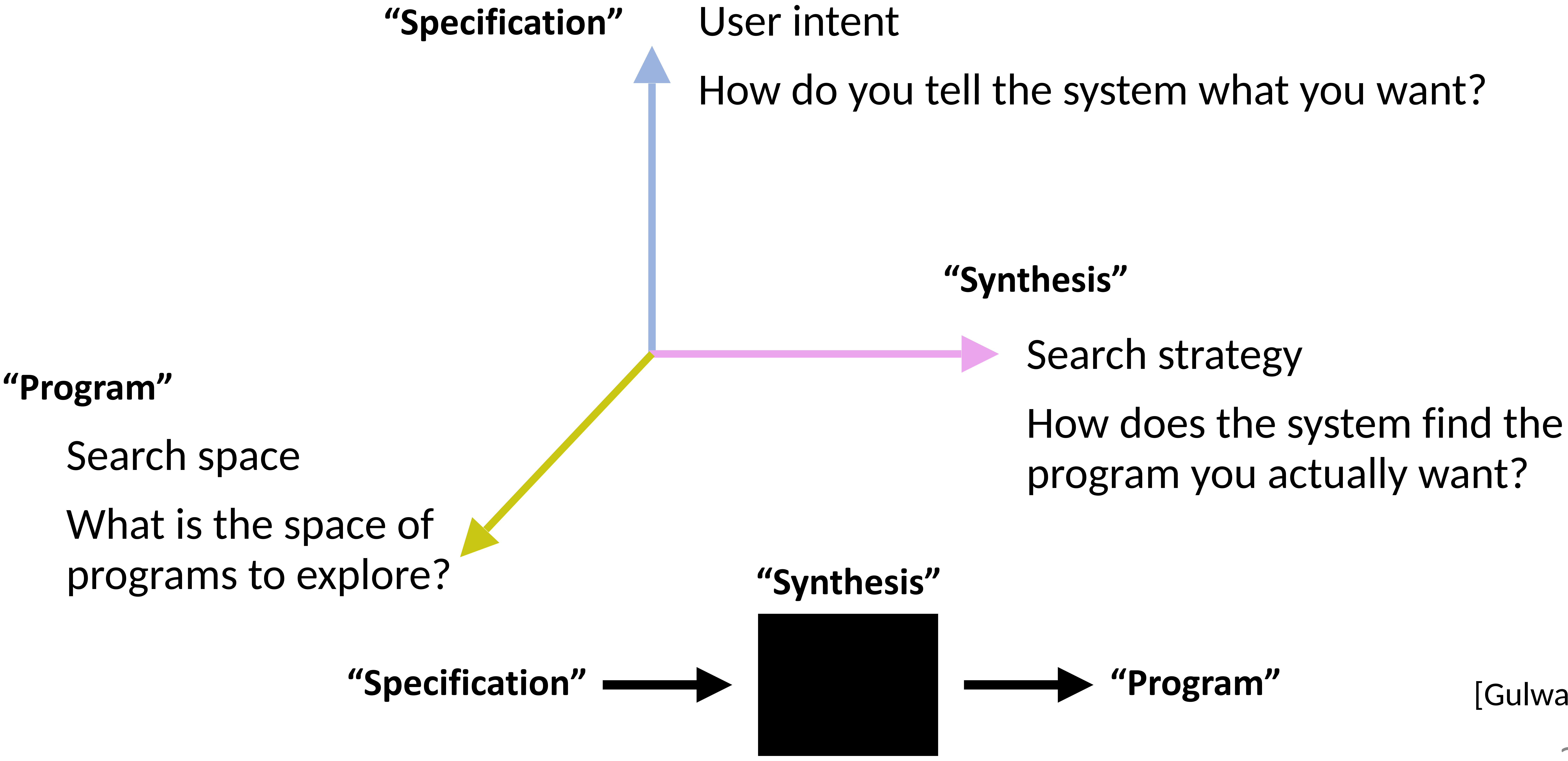
→ [synthesizer] → **Programs**

Search space:

- Grammar
- DSL
- Partial program
- Components

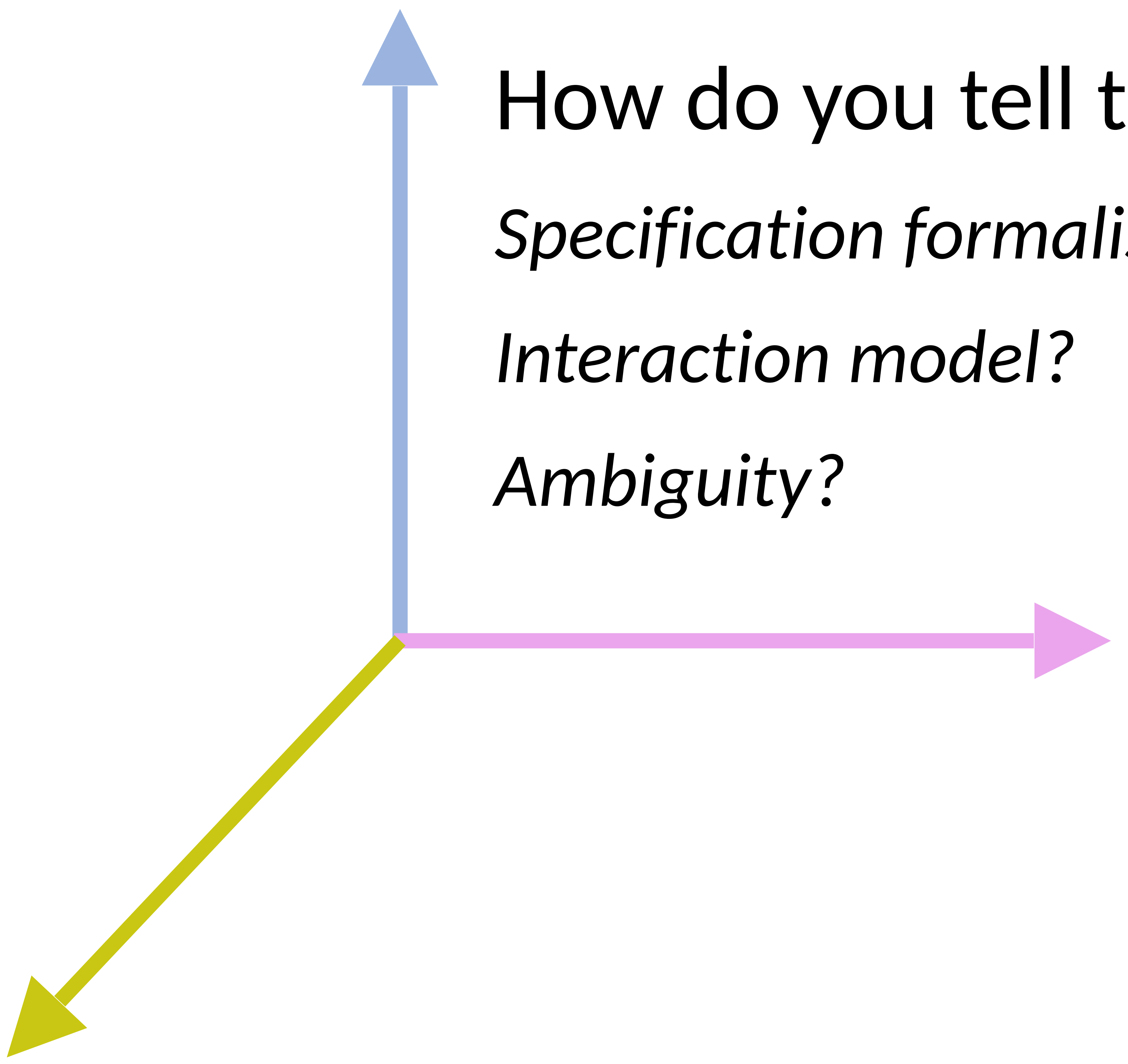
Find a program in search space consistent with specification

Dimensions in modern program synthesis



[Gulwani 2010]

Dimensions in modern program synthesis



User intent

How do you tell the system what you want?

Specification formalism?

Interaction model?

Ambiguity?

Search strategy

How does the system find the program you actually want?

How do you guide the system towards relevant programs?

How does the system exploit the structure of the search space?

Search space

What is the space of programs to explore?

How do you represent domain knowledge?

Built-in or user-defined?

Dimensions in modern program synthesis

User intent

Input-output examples

Logical specifications

Equivalent programs

Natural language

Search strategy

Enumerative search + pruning

Constraint-based search

Representation-based search

Stochastic search

ML-based

…

Search space

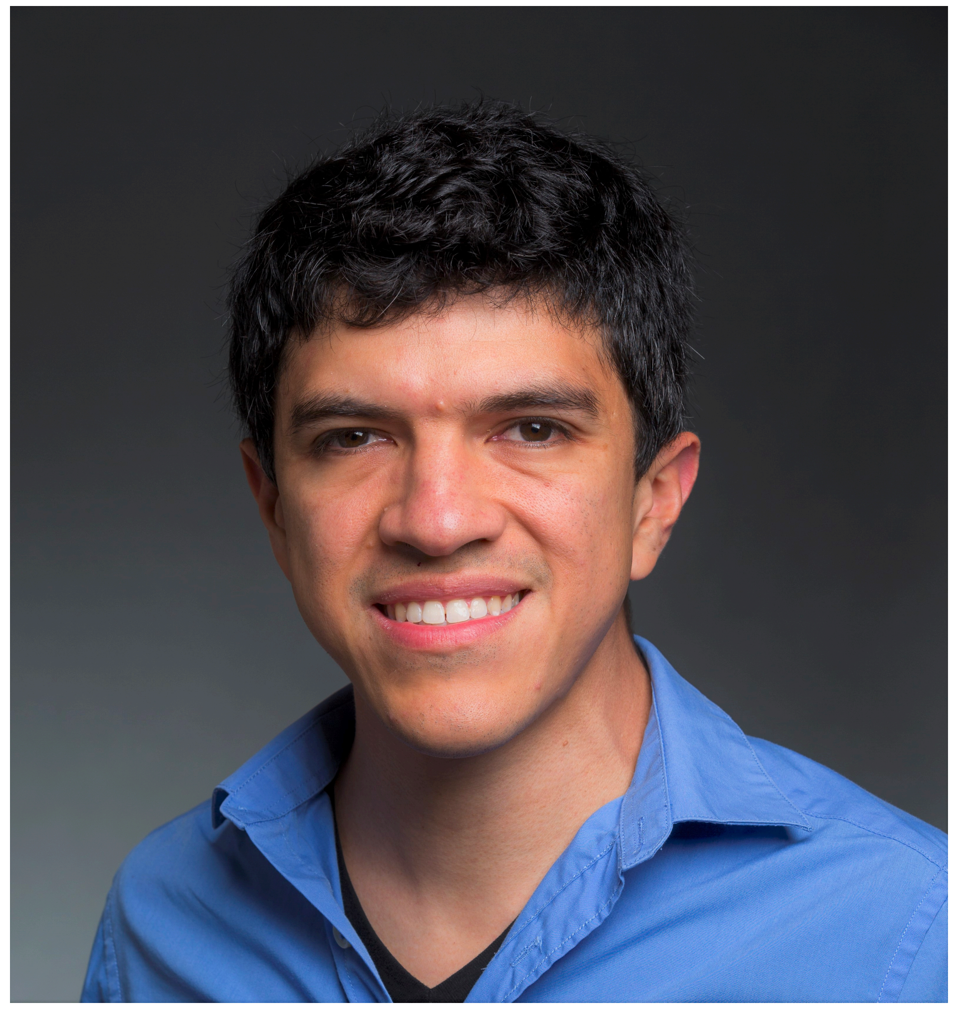
Grammars/DSLs

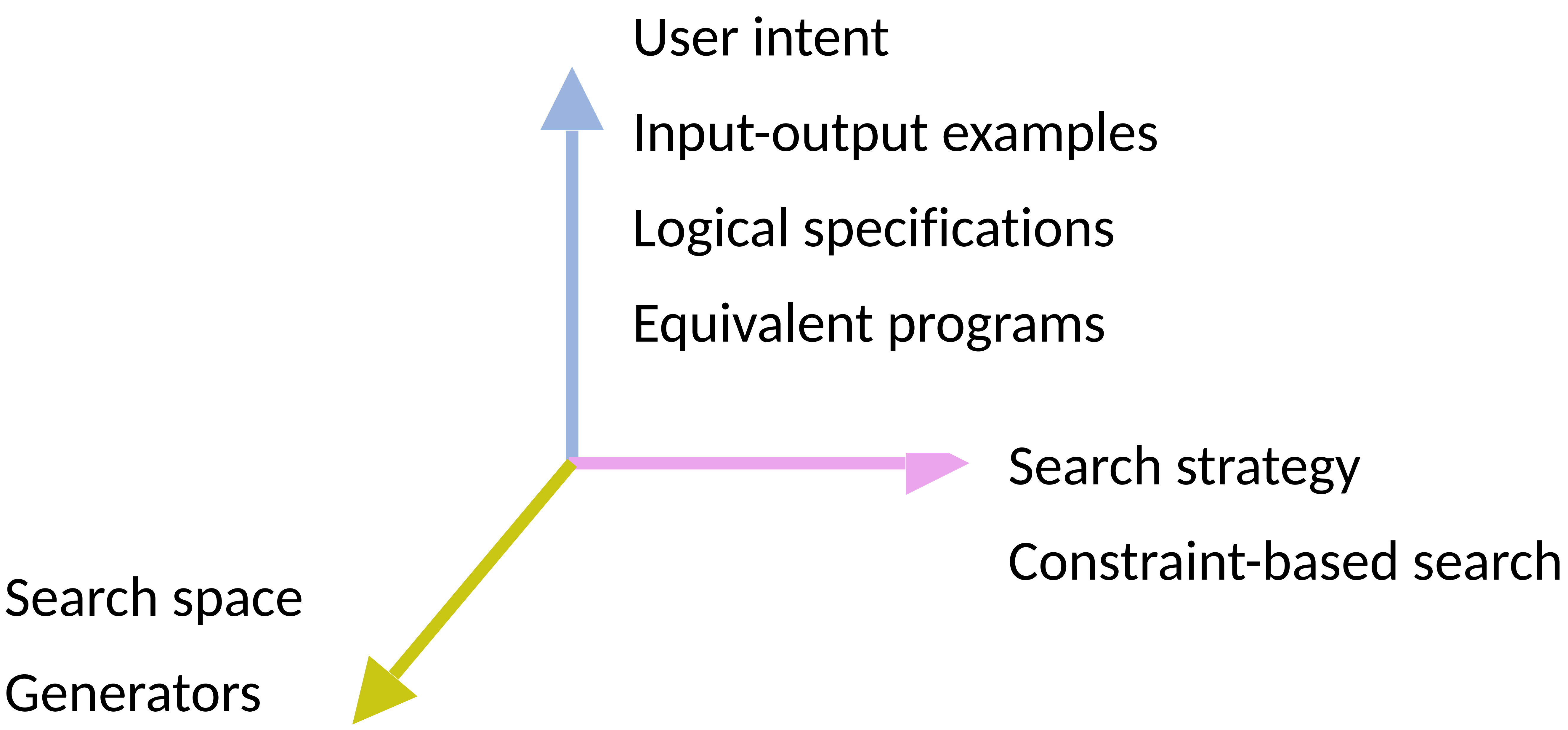
Generators

Components

2006: Sketch

Armando-Solar Lezamma

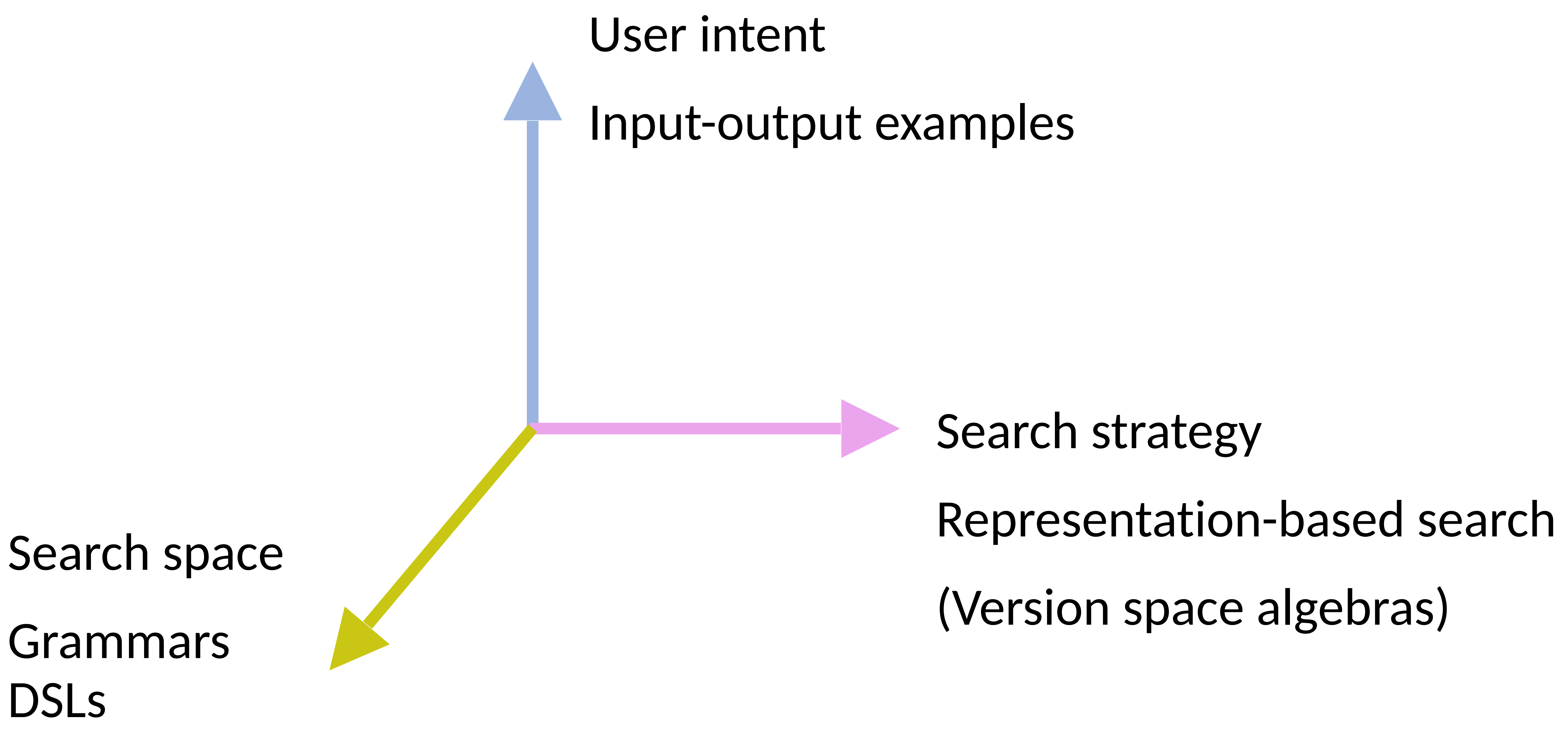




Solar-Lezamma et al., *Combinatorial Sketching for Finite Programs*, 2006

2011: FlashFill

Sumit Gulwani



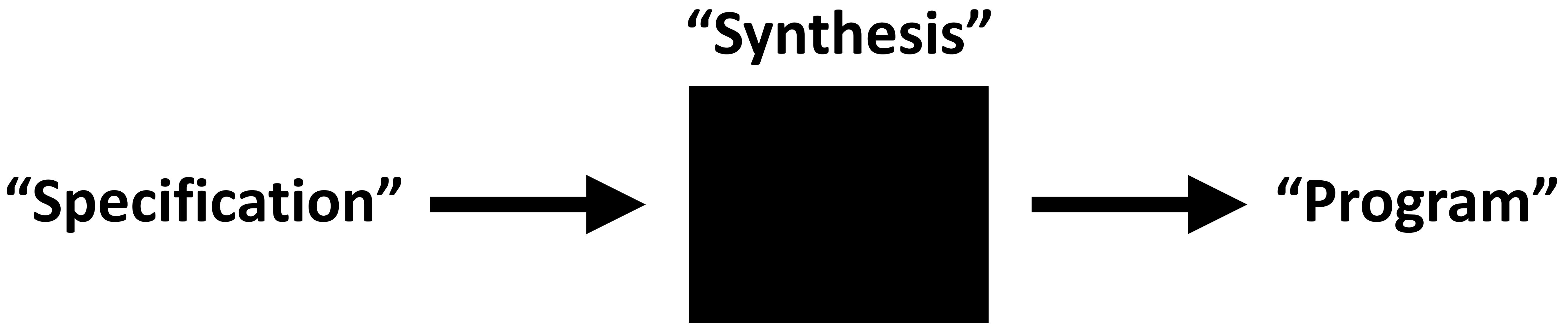
Gulwani, Automatic String Processing in Spreadsheets using Input-Output Examples, 2011

Today’s Agenda

- History of Program Synthesis
- **<u>Syntax-Guided Synthesis (SYGUS)</u>**
- Context-Free Grammars (CFGs)

Syntax-Guided Synthesis (SYGUS)

- SYGUS is an instantiation of our program synthesis definition



Syntax-Guided Synthesis (SYGUS)

- Key idea 1: **Restrict the programming language** in which a program is written
  - E.g., FlashFill uses a domain-specific language for string transformations

$$
\begin{array}{rcl}
\text{String expr } P & := & \texttt{Switch}((\mathrm{b}_1, \mathrm{e}_1), \cdot\cdot, (\mathrm{b}_n, \mathrm{e}_n)) \\
\text{Bool b} & := & \mathrm{d}_1 \vee \cdot\cdot \vee \mathrm{d}_n \\
\text{Conjunct d} & := & \pi_1 \wedge \cdot\cdot \wedge \pi_n \\
\text{Predicate } \pi & := & \texttt{Match}(v_i, \mathrm{r}, k) \mid \neg\,\texttt{Match}(v_i, \mathrm{r}, k) \\
\text{Trace expr e} & := & \texttt{Concatenate}(\mathrm{f}_1, \cdot\cdot, \mathrm{f}_n) \\
\text{Atomic expr f} & := & \texttt{SubStr}(v_i, \mathrm{p}_1, \mathrm{p}_2) \\
 & \mid & \texttt{ConstStr}(s) \\
 & \mid & \texttt{Loop}(\lambda w : \mathrm{e}) \\
\text{Position p} & := & \texttt{CPos}(k) \mid \texttt{Pos}(\mathrm{r}_1, \mathrm{r}_2, \mathrm{c}) \\
\text{Integer expr c} & := & k \mid k_1 w + k_2 \\
\text{Regular Expression r} & := & \texttt{TokenSeq}(\mathrm{T}_1, \cdot\cdot, \mathrm{T}_m) \\
\text{Token T} & := & C+ \quad \mid \quad [\neg C]+ \\
 & \mid & \text{SpecialToken}
\end{array}
$$

Syntax-Guided Synthesis (SYGUS)

- Key idea 1: **Restrict the programming language** in which a program is written
  - E.g., FlashFill uses a domain-specific language for string transformations
- Key idea 2: **Search** in the space of programs constrained by this language
  - This enables aggressive optimizations to accelerate search/synthesis
  - Typically use Context-Free Grammars (CFGs) to represent a space of programs

Syntax-Guided Synthesis (SYGUS)

- Key idea 1: **Restrict the programming language** in which a program is written
  - E.g., FlashFill uses a domain-specific language for string transformations
- Key idea 2: **Search** in the space of programs constrained by this language
  - This enables aggressive optimizations to accelerate search/synthesis
  - Typically use Context-Free Grammars (CFGs) to represent a space of programs
- Key idea 3: **Specifications** are provided as **logical formulas**
  - Sometimes can be relaxed

Syntax-Guided Synthesis (SYGUS)

“Specification”
Logical formulas

“Program”
in a CFG

Search-based technique

Today's Agenda

- History of Program Synthesis
- Syntax-Guided Synthesis (SYGUS)
- **<u>Context-Free Grammars (CFGs)</u>**

Context-Free Grammars (CFGs)

- Formal grammars are used to describe strings in a formal language
  - E.g., regular grammars/languages, tree grammars/languages, CFGs
  - Different from grammars of natural languages such as English
- In this course, use CFGs to describe programs (which are also strings)
  - CFGs define **syntax** of programs (how to write programs)
  - CFGs do **not** define **semantics** of programs (what programs mean)
    - We will talk about semantics in a few lectures

CFG Formalisms

- Consider this CFG example: $S \rightarrow 01 \mid 0S1$

- **Terminals**: Symbols of the alphabet of the language being defined.
  - $\{0,1\}$
- **Variables (non-terminals)**: A finite set of other symbols. Each of these symbols represents a language. A variable can be replaced.
  - S
- **Start symbol**: A special variable whose language is the language being defined.
  - S
- **Production rules (productions, substitution rules, rules)**: These rules define how a variable can get replaced. A production has the form: **variable (head) —> a string of variables and terminals (body)**
  - Left hand side is the variable that is to be replaced. Right hand side is the “content” to replace with.
  - $\{\ S \rightarrow 01,\ S \rightarrow 0S1\ \}$

CFG Formalisms

- Consider this CFG example: $S \to 01 \mid 0S1$

- **Terminals:** $\{0,1\}$
- **Non-terminals:** S
- **Start symbol:** S
- **Productions:** $\{\ S \to 01,\ S \to 0S1\ \}$

- **What's the language that's represented by this CFG?**

CFG Formalisms

- Consider this CFG example: $S \rightarrow 01 \mid 0S1$

- **Terminals:** $\{0,1\}$
- **Non-terminals:** S
- **Start symbol:** S
- **Productions:** $\{\ S \rightarrow 01,\ S \rightarrow 0S1\ \}$

- **What's the language that's represented by this CFG?**
  - $\{\ 0^n1^n \mid n \geq 1\ \}$

CFG Formalisms

- Consider this CFG example: $S \to 01 \mid 0S1$

- **Terminals:** $\{0,1\}$
- **Non-terminals:** S
- **Start symbol:** S
- **Productions:** $\{\ S \to 01,\ S \to 0S1\ \}$

- **What's the language that's represented by this CFG?**
  - $\{\ 0^n 1^n \mid n \geq 1\ \}$
  - Why? How to "generate" these strings from CFG?
    - $S \to 01$ is the "base case" and $S \to 0S1$ is the "recursive case"

Formal CFG Definition

- A Context-Free Grammar is a 4-tuple (N, T, P, S)
  - N is a set of non-terminals (variables)
  - T is a set of terminals that is disjoint from V
  - $P \subseteq N \times \{N \cup T\}*$ is a finite set of production rules
  - $S \subseteq N$ is the start symbol (variable)

Generate Strings from CFG: Derivations

- Given a CFG G, we can derive strings in the language L defined by G
  - Start with the start symbol, and repeatedly replace some variable by the body of one of its productions, until no replacement is possible (only terminals)

Generate Strings from CFG: Derivations

- Given a CFG G, we can derive strings in the language L defined by G
  - Start with the start symbol, and repeatedly replace some variable by the body of one of its productions, until no replacement is possible (only terminals)
- E.g., how to derive 000111 from previous CFG? $S \to 01 \mid 0S1$

Generate Strings from CFG: Derivations

- Given a CFG G, we can derive strings in the language L defined by G
  - Start with the start symbol, and repeatedly replace some variable by the body of one of its productions, until no replacement is possible (only terminals)
- E.g., how to derive 000111 from previous CFG? $S \to 01 \mid 0S1$
  - $S \Rightarrow 0S1 \Rightarrow 00S11 \Rightarrow 000111$
- Can you derive 00011 from previous CFG?

Generate Strings from CFG: Derivations

- Given a CFG G, we can derive strings in the language L defined by G
  - Start with the start symbol, and repeatedly replace some variable by the body of one of its productions, until no replacement is possible (only terminals)
- E.g., how to derive 000111 from previous CFG?
  - $S \Rightarrow 0S1 \Rightarrow 00S11 \Rightarrow 000111$
- Can you derive 00011 from previous CFG?
- Derivations may not be unique. There may be multiple ways to derive the same string

Language Defined by CFG

- Given a CFG G, the language defined by G, denoted $L(G)$, is the set of all strings that can be derived from G, i.e., $L(G) = \{ w \mid S \Rightarrow^* w \}$
- E.g., consider previous CFG with productions $S \to \epsilon, S \to 0S1$
- Both define the same language, though they are different grammars

Another CFG Example

- A (small) subset of R language

$$df ::= x \mid gather(df, k, k, s, s) \mid unite(df, k, k, s)$$

$$k ::= 1 \mid 2 \mid 3 \mid 4$$

$$s ::= tmp1 \mid tmp2 \mid tmp3$$

- Terminals?
- Non-terminals?
- Start symbol?
- Productions?

Abstract Syntax Trees (ASTs)

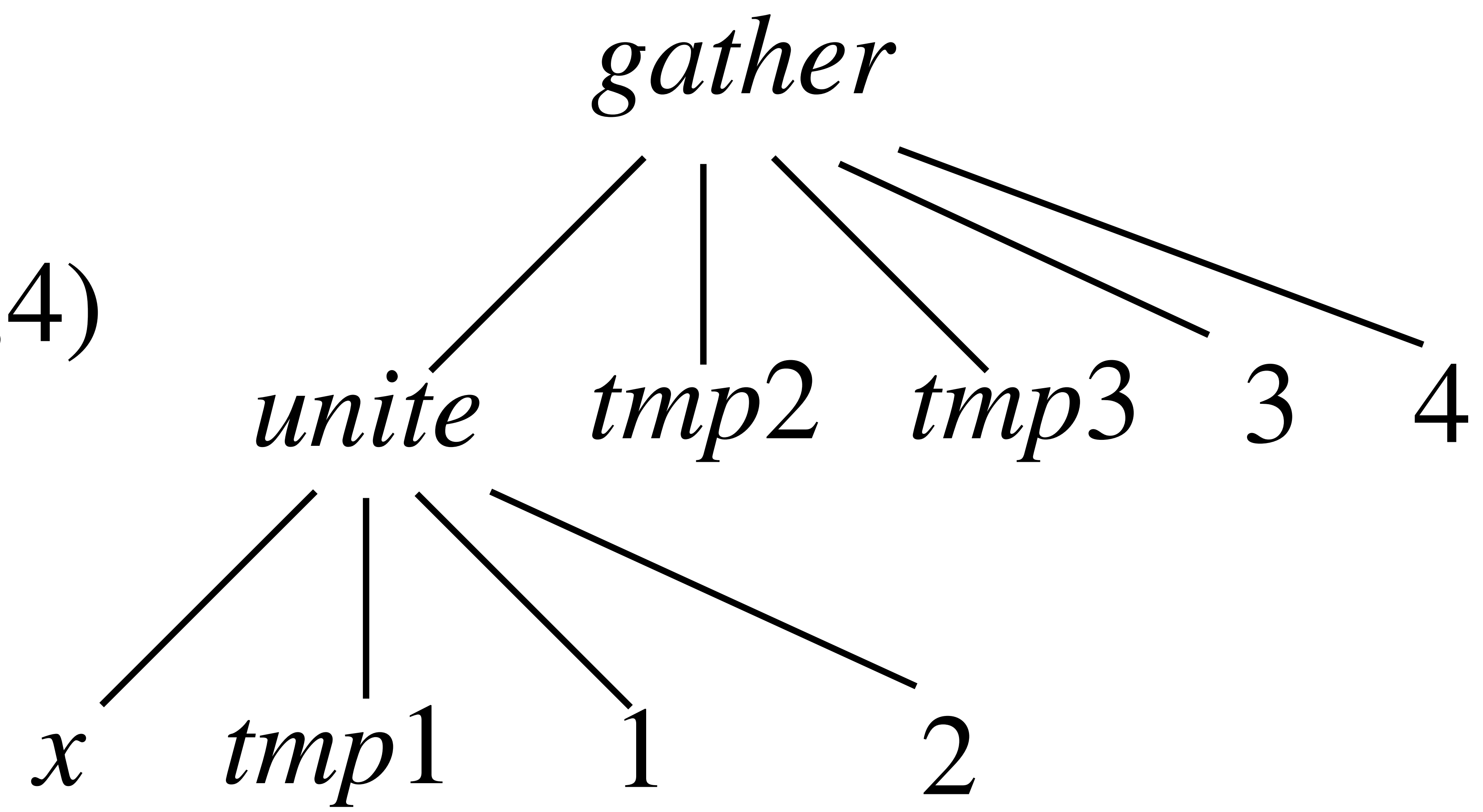
- If CFG describes a language of programs, each string in CFG corresponds to a program
- Use AST as a tree representation of the abstract syntactic structure of a program
  - Each AST node denotes an operator in the program

Abstract Syntax Trees (ASTs)

- If CFG describes a language of programs, each string in CFG corresponds to a program
- Use AST as a tree representation of the abstract syntactic structure of a program
  - Each AST node denotes an operator in the program
- Consider CFG:

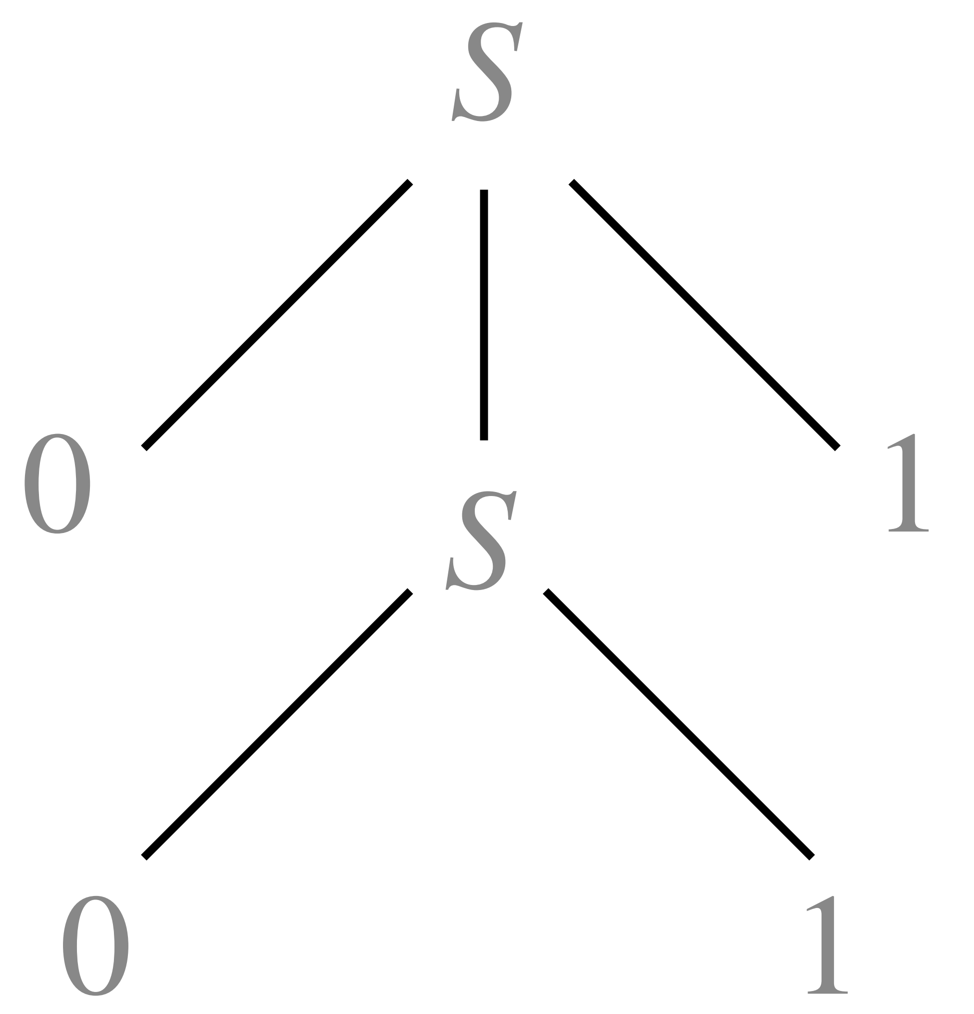
$$df ::= x \mid gather(df, k, k, s, s) \mid unite(df, k, k, s)$$
$$k ::= 1 \mid 2 \mid 3 \mid 4$$
$$s ::= tmp1 \mid tmp2 \mid tmp3$$

- Consider: $gather(unite(x, tmp1, 1, 2), tmp2, tmp3, 3, 4)$



Transform String to Tree: Parsing

- Broadly, transforming strings (unstructured data) into trees (structured data)
- Depending on concrete problems, trees may be defined differently
  - In this course, we mostly use Abstract Syntax Trees (ASTs)
  - E.g., consider $S ::= 01 \mid 0S1$, parse tree for 0011
- Parsing itself is a research area but is not focus of this course



Today’s Agenda

- History of Program Synthesis
- Syntax-Guided Synthesis (SYGUS)
- Context-Free Grammars
- <u>**Revisit SYGUS**</u>

Revisit SYGUS

"Specification"
Logical formulas

"Program"
in a CFG

Search-based technique

Formal Definition of SYGUS

- Given a first-order formula ϕ in a background theory T and a CFG G, the syntax-guided synthesis problem is to find an expression $e \in G$ such that formula $\phi[f/e]$ is valid in theory T.

Formal Definition of SYGUS

- Given a first-order formula ϕ in a background theory T and a CFG G, the syntax-guided synthesis problem is to find an expression $e \in G$ such that formula $\phi[f/e]$ is valid in theory T.

"Specification"

$$f(1) = 2$$

Search-based technique

"Program"

in a CFG

$$e ::= x \mid 1 \mid e + e$$

Formal Definition of SYGUS

- Given a first-order formula ϕ in a background theory T and a CFG G, the syntax-guided synthesis problem is to find an expression $e \in G$ such that formula $\phi[f/e]$ is valid in theory T.

Theory of Integer Linear Arithmetic

"Specification"

$f(1) = 2$

Search-based technique

"Program"

in a CFG

$e ::= x \mid 1 \mid e + e$

Formal Definition of SYGUS

- Given a first-order formula ϕ in a background theory T and a CFG G, the syntax-guided synthesis problem is to find an expression $e \in G$ such that formula $\phi[f/e]$ is valid in theory T.

Theory of Integer Linear Arithmetic

Potential Solutions:

$f(x) = x + 1$

"Specification" → → "Program"

$f(1) = 2$

1+1 is valid in ILA

Search-based technique

in a CFG

$e ::= x \mid 1 \mid e + e$

Formal Definition of SYGUS

- Given a first-order formula ϕ in a background theory T and a CFG G, the syntax-guided synthesis problem is to find an expression $e \in G$ such that formula $\phi[f/e]$ is valid in theory T.

Theory of Integer Linear Arithmetic

Potential Solutions:

$f(x) = x + 1$

$f(x) = x + x$

"Specification"

$f(1) = 2$

1+1 is valid in ILA

Search-based technique

"Program"

in a CFG

$e ::= x \mid 1 \mid e + e$

SYGUS Recap

Potential Solutions:

$$f(x) = x + 1$$

$$f(x) = x + x$$

"Specification" → → "Program"

$$f(1) = 2$$

Semantic Constraint

Search-based technique

in a CFG

$$e ::= x \mid 1 \mid e + e$$

Syntactic Constraint

Summary

- History of Program Synthesis
  - Deductive synthesis —> Inductive synthesis
- Syntax-Guided Synthesis (SYGUS)
  - Key idea: Search within a constrained space of programs defined by grammars
- Context-Free Grammars (CFGs)