

COS320: Compiling Techniques

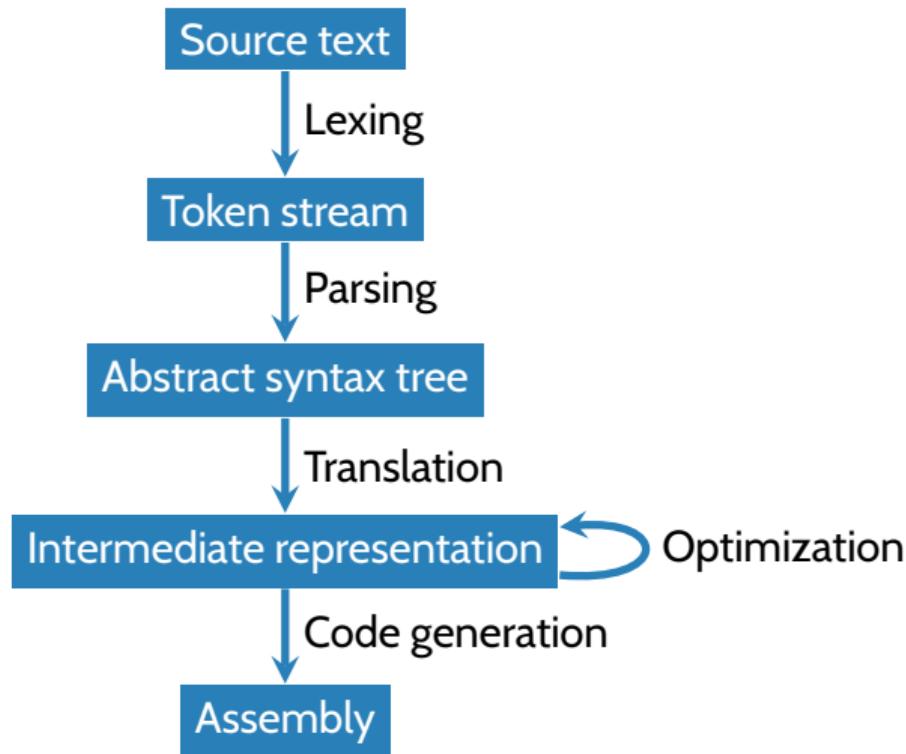
Zak Kincaid

January 29, 2026

Logistics

- Midterm scores released – please submit regrade requests by Friday 3/22
- HW3 due next Monday

Compiler phases (simplified)



Semantic Analysis

Semantic analysis

- The *semantic analysis phase* is responsible for:
 - Connecting symbol *occurrences* to their definitions (i.e., implement scoping rules)
 - Checking that the AST is well-typed
 - Various other well-formedness checks not captured by the grammar (e.g., break must appear inside a for, while, or switch)

Semantic analysis

- The *semantic analysis phase* is responsible for:
 - Connecting symbol *occurrences* to their definitions (i.e., implement scoping rules)
 - Checking that the AST is well-typed
 - Various other well-formedness checks not captured by the grammar (e.g., break must appear inside a for, while, or switch)
- Semantic analysis phase can report *warnings* (potential problems) or *errors* (severe problems that must be resolved in order to compile)
 - ex.c:4:5: warning: assignment makes integer from pointer without a cast
 - ex.c:3:11: error: ‘i’ undeclared (first use in this function)

Semantic analysis

- The *semantic analysis phase* is responsible for:
 - Connecting symbol *occurrences* to their definitions (i.e., implement scoping rules)
 - Checking that the AST is well-typed
 - Various other well-formedness checks not captured by the grammar (e.g., break must appear inside a for, while, or switch)
- Semantic analysis phase can report *warnings* (potential problems) or *errors* (severe problems that must be resolved in order to compile)
 - ex.c:4:5: warning: assignment makes integer from pointer without a cast
 - ex.c:3:11: error: ‘i’ undeclared (first use in this function)
- Main data structure manipulated by semantic analysis: *symbol table*
 - Mapping from symbols to information about those symbols (type, location in source text, ...)
 - Symbol table is used to help translation into IR
 - Semantic analysis may also *decorate* AST (e.g., attach type information to expressions, or replace symbols with references to their symbol table entry)

Types

- Type checking catches errors at *compile time*, eliminating a class of mistakes that would otherwise lead to run-time errors
- Type information is sometimes necessary for code generation
 - Floating-point + is not the same instruction as integer + is not the same as pointer/integer +
 - pointer/integer compiled differently depending on pointer type
 - Assignment x = y compiled differently if y is an int or a struct

What is a type?

- *Intrinsic view* (Church-style): a type is syntactically part of a term.

- A term that cannot be typed is not a term at all
- Types do not have inherent meaning – they are just used to encode syntax

- *Extrinsic view* (Curry-style): a type is a *property* of a term.

- For any term and every type, either the term has that type or it does not
- A term may have multiple types
- A term may have no types



Alonzo Church



Haskell Curry

What is a type system?

A type system consists of a system of judgements and inference rules

- (Extrinsic view) A **judgement** is a *claim*, which may or may not be valid
 - $\vdash 3 : \text{int}$ – “3 has type integer”
 - $\vdash (1 + 2) : \text{bool}$ – “(1+2) has type boolean”
 - A type system might involve many different kinds of judgement (well-typed expressions, well-formed types, well-formed statements, ...)
- **Inference rules** are used to derive *valid* judgements from other valid judgements.

$$\text{ADD} \quad \frac{\vdash e_1 : \text{int} \quad \vdash e_2 : \text{int}}{\vdash e_1 + e_2 : \text{int}}$$

Read: “If e_1 and e_2 have type int, so does $e_1 + e_2$ ”

Inference rules, generally

An *inference rule* consists of a list of **premises** J_1, \dots, J_n and one **conclusion** J (and optionally a side-condition), typically written as:

$$\frac{J_1 \quad J_2 \quad \dots \quad J_n}{J} \text{ SIDE-CONDITION}$$

- Side-condition: additional premise, but not a judgement
- Read *top-down*: If J_1 and J_2 and ... and J_n are valid (and the side condition holds) then J is valid.
- Read *bottom-up*: To prove J is valid, sufficient to prove J_1, J_2, \dots, J_n are valid (+ side condition)

A simple expression language

- Syntax of expressions

$$\begin{aligned} \langle \text{Exp} \rangle ::= & \langle \text{Var} \rangle \mid \langle \text{Int} \rangle \\ & \mid \langle \text{Exp} \rangle + \langle \text{Exp} \rangle \mid \langle \text{Exp} \rangle * \langle \text{Exp} \rangle \\ & \mid \langle \text{Exp} \rangle \wedge \langle \text{Exp} \rangle \mid \langle \text{Exp} \rangle \vee \langle \text{Exp} \rangle \\ & \mid \langle \text{Exp} \rangle \leq \langle \text{Exp} \rangle \mid \langle \text{Exp} \rangle = \langle \text{Exp} \rangle \\ & \mid \text{if } \langle \text{Exp} \rangle \text{ then } \langle \text{Exp} \rangle \text{ else } \langle \text{Exp} \rangle \end{aligned}$$

- $3 + (2 \wedge 0)$ is syntactically well-formed, but not well-typed
- Is $x + 1$ well-typed?

Type judgements

- A *type environment* is a symbol table mapping symbols to types.
 - E.g., $[x \mapsto \text{int}, y \mapsto \text{bool}, z \mapsto \text{int}]$: x and z are ints, y is a bool
 - Notation: type environment denoted by Γ
 - Notation: $\Gamma\{x \mapsto t\}$ is a functional update

$$\Gamma\{x \mapsto t\}(y) = \begin{cases} t & \text{if } x = y \\ \Gamma(y) & \text{otherwise} \end{cases}$$

- E.g., $[x \mapsto \text{int}, y \mapsto \text{int}]\{x \mapsto \text{bool}\} = [x \mapsto \text{bool}, y \mapsto \text{int}]$

Type judgements

- A *type environment* is a symbol table mapping symbols to types.
 - E.g., $[x \mapsto \text{int}, y \mapsto \text{bool}, z \mapsto \text{int}]$: x and z are ints, y is a bool
 - Notation: type environment denoted by Γ
 - Notation: $\Gamma\{x \mapsto t\}$ is a functional update

$$\Gamma\{x \mapsto t\}(y) = \begin{cases} t & \text{if } x = y \\ \Gamma(y) & \text{otherwise} \end{cases}$$

- E.g., $[x \mapsto \text{int}, y \mapsto \text{int}]\{x \mapsto \text{bool}\} = [\text{bool}, y \mapsto \text{int}]$
- A *type judgement* takes the form $\Gamma \vdash e : t$
 - Read “Under the type environment Γ , the expression e has type t ”

Inference rules

INT

$$\frac{}{\Gamma \vdash n : \text{int}} n \in \{..., -1, 0, 1, ...\}$$

VAR

$$\frac{}{\Gamma \vdash x : t} \Gamma(x) = t$$

ADD

$$\frac{\Gamma \vdash e_1 : \text{int} \quad \Gamma \vdash e_2 : \text{int}}{\Gamma \vdash e_1 + e_2 : \text{int}}$$

AND

$$\frac{\Gamma \vdash e_1 : \text{bool} \quad \Gamma \vdash e_2 : \text{bool}}{\Gamma \vdash e_1 \wedge e_2 : \text{bool}}$$

LEQ

$$\frac{\Gamma \vdash e_1 : \text{int} \quad \Gamma \vdash e_2 : \text{int}}{\Gamma \vdash e_1 \leq e_2 : \text{bool}}$$

IF

$$\frac{\Gamma \vdash e_1 : \text{bool} \quad \Gamma \vdash e_2 : t \quad \Gamma \vdash e_3 : t}{\Gamma \vdash \text{if } e_1 \text{ then } e_2 \text{ else } e_3 : t}$$

Derivations

- A **derivation** or *proof tree* is a tree where each node is labelled by a judgement, and edges connect premises to a conclusion according to some inference rule.
- Leaves of the tree are *axioms* (inference rules w/o premises)

Derivation of $x : \text{int} \vdash 2 + x \leq 10 : \text{bool}$:

$$\text{LEQ} \frac{\text{ADD} \frac{\text{INT} \frac{}{x : \text{int} \vdash 2 : \text{int}} \text{ VAR} \frac{}{x : \text{int} \vdash x : \text{int}}}{x : \text{int} \vdash 2 + x : \text{int}} \text{ INT} \frac{}{x : \text{int} \vdash 10 : \text{int}}}{x : \text{int} \vdash 2 + x \leq 10 : \text{bool}}$$

Derivation for $x: \text{int} \vdash \text{if } x \leq 0 \text{ then } x \text{ else } -1 * x: \text{int}$:

$$\text{If } \frac{\text{LEQ } \frac{\text{VAR } \frac{}{x: \text{int} \vdash x: \text{int}} \quad \text{INT } \frac{}{x: \text{int} \vdash -1: \text{int}}}{x: \text{int} \vdash x \leq 0: \text{bool}} \quad \text{VAR } \frac{}{x: \text{int} \vdash x: \text{int}} \quad \text{INT } \frac{}{x: \text{int} \vdash -1: \text{int}} \quad \text{VAR } \frac{}{x: \text{int} \vdash x: \text{int}}}{x: \text{int} \vdash \text{if } x \leq 0 \text{ then } x \text{ else } -1 * x: \text{int}}$$
$$\text{MUL } \frac{}{x: \text{int} \vdash -1 * x: \text{int}}$$

Type checking

- Goal of a type checker: given a context Γ , expression e , and type t , determine whether a derivation of the judgement $\Gamma \vdash e : t$ exists.
- Method: recurse on the structure of the AST, applying inference rules “bottom-up”

Binders & functions: scope logic

LET

$$\frac{\Gamma \vdash e_1 : t_1 \quad \Gamma\{x \mapsto t_1\} \vdash e_2 : t}{\Gamma \vdash \mathbf{let} \, x = e_1 \, \mathbf{in} \, e_2 : t}$$

FUN

$$\frac{\Gamma\{x \mapsto t_1\} \vdash e : t_2}{\Gamma \vdash \mathbf{fun} \, (x : t_1) \rightarrow e : t_1 \rightarrow t_2}$$

APP

$$\frac{\Gamma \vdash e_1 : t_1 \rightarrow t_2 \quad \Gamma \vdash e_2 : t_1}{\Gamma \vdash e_1 \, e_2 : t_2}$$

Type inference

- Goal of type inference: given a context Γ and expression e , determine a type t for which there is a derivation of the judgement $\Gamma \vdash e : t$.
- Method: (again) recurse on the structure of the AST, applying inference rules “bottom-up”
- This only works because we have a very simple type system
 - OCaml type inference (Hindley-Milner): recurse on the structure of the AST to produce a *constraint system*, then solve the constraints

Type soundness



Well typed programs cannot “go wrong”

Robin Milner

- More formally: if $\vdash e : t$ is derivable, then evaluating e either fails to terminate or yields a value of type t
 - Note: for our language (extension of simply-typed lambda calculus with integers and booleans), we have something stronger: evaluating e always yields a value of type t

Well-formed types

- In languages with type definitions, need additional rules to define well-formed types
- Judgements take the form $H \vdash t$
 - H is set of type names
 - t is a type
 - $H \vdash t$ - “Assuming H names well-formed types, t is a well-formed type”

Well-formed types

- In languages with type definitions, need additional rules to define well-formed types
- Judgements take the form $H \vdash t$
 - H is set of type names
 - t is a type
 - $H \vdash t$ - “Assuming H names well-formed types, t is a well-formed type”

INT	BOOL	ARROW	NAMED
$\frac{}{H \vdash \text{int}}$	$\frac{}{H \vdash \text{bool}}$	$\frac{H \vdash t_1 \quad H \vdash t_2}{H \vdash t_1 \rightarrow t_2}$	$\frac{}{H \vdash s \quad s \in H}$

Well-formed types

- In languages with type definitions, need additional rules to define well-formed types
- Judgements take the form $H \vdash t$
 - H is set of type names
 - t is a type
 - $H \vdash t$ - “Assuming H names well-formed types, t is a well-formed type”

$$\begin{array}{c} \text{INT} & \text{BOOL} & \text{ARROW} & \text{NAMED} \\ \hline \frac{}{H \vdash \text{int}} & \frac{}{H \vdash \text{bool}} & \frac{H \vdash t_1 \quad H \vdash t_2}{H \vdash t_1 \rightarrow t_2} & \frac{}{H \vdash s \quad s \in H} \end{array}$$

- Note: also need to modify the typing rules & judgements. E.g.,

$$\begin{array}{c} \text{FUN} \\ \hline \frac{H \vdash t_1 \quad H, \Gamma \{x \mapsto t_1\} \vdash e : t_2}{H, \Gamma \vdash \mathbf{fun} (x : t_1) \rightarrow e : t_1 \rightarrow t_2} \end{array}$$

Statements

- In languages with statements, need additional rules to define well-formed statements
- E.g., judgements may take the form $\Gamma; rt \vdash s$
 - Γ is a type environment (variables \rightarrow types)
 - rt is a type
 - $\Gamma; rt \vdash s$ – “assuming type environment Γ , s is a well-formed statement within a function that returns a value of type rt ”

Statements

- In languages with statements, need additional rules to define well-formed statements
- E.g., judgements may take the form $\Gamma; rt \vdash s$
 - Γ is a type environment (variables \rightarrow types)
 - rt is a type
 - $\Gamma; rt \vdash s$ – “assuming type environment Γ , s is a well-formed statement within a function that returns a value of type rt ”

ASSIGN

$$\frac{\Gamma \vdash e : \Gamma(x)}{\Gamma; rt \vdash x := e}$$

RETURN

$$\frac{\Gamma \vdash e : rt}{\Gamma; rt \vdash \mathbf{return} \ e}$$

DECL

$$\frac{\Gamma \vdash e : t \quad \Gamma\{x \mapsto t\}; rt \vdash s_2}{\Gamma; rt \vdash \mathbf{var} \ x = e; s_2}$$

Additional aspects

- In OCaml, can have a variable and a type with the same name
 - Multiple namespaces \Rightarrow multiple environments / symbol tables
- Parametric polymorphism
 - E.g., `fun x -> x` in ocaml has type `'a -> 'a`
 - Finite representation of infinitely many typings
- Subtyping (e.g., object-oriented languages) – next time
 - Related: casting, coercion