Semantic Analysis, Symbol Tables

CS 4100
Gordon Stewart
Ohio University

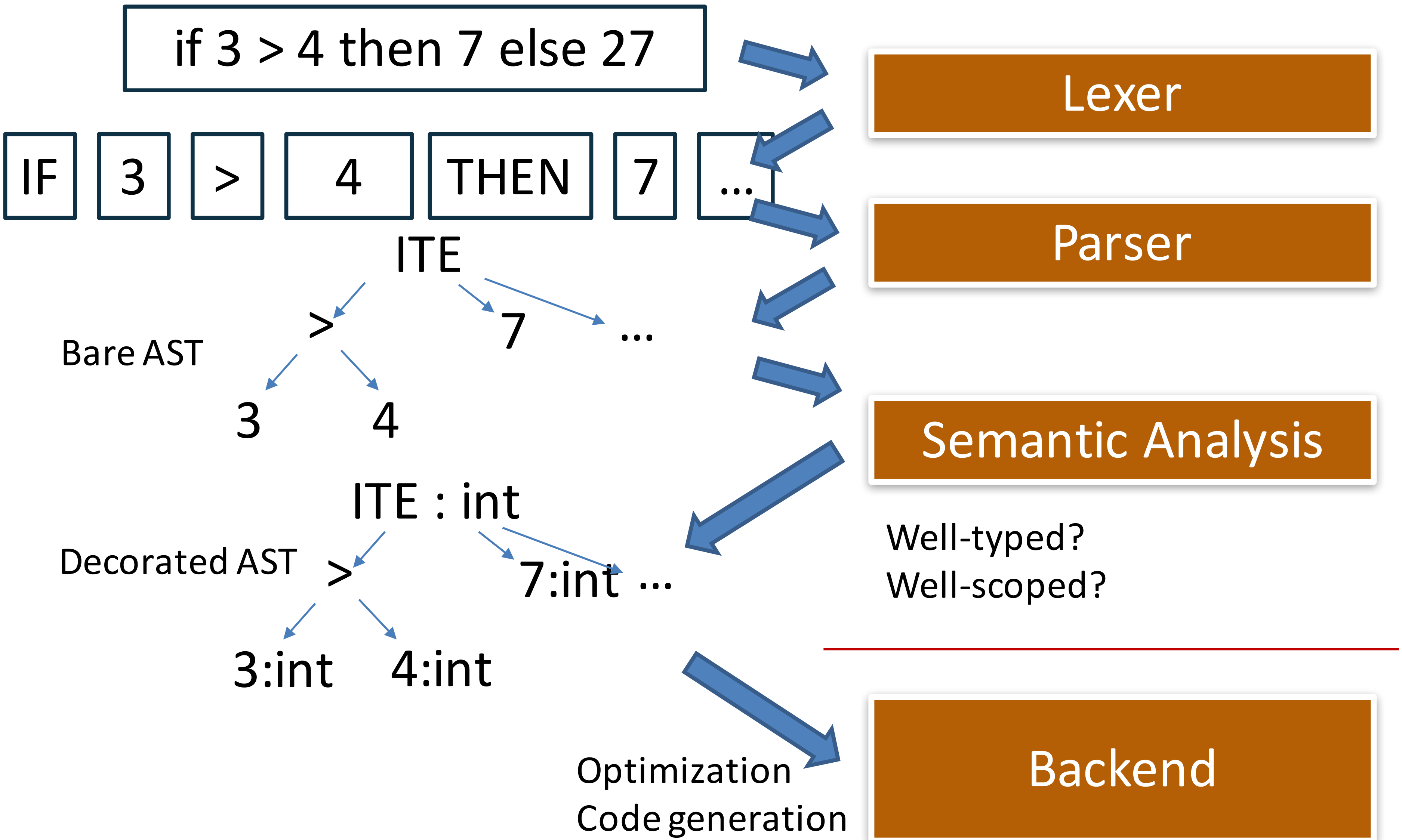
Semantic Analysis

- Many programs that are ***syntactically valid*** as determined by the context-free grammar associated with the language
  - Accepted by ***lexer.mll***
  - Accepted by ***parser.mly***

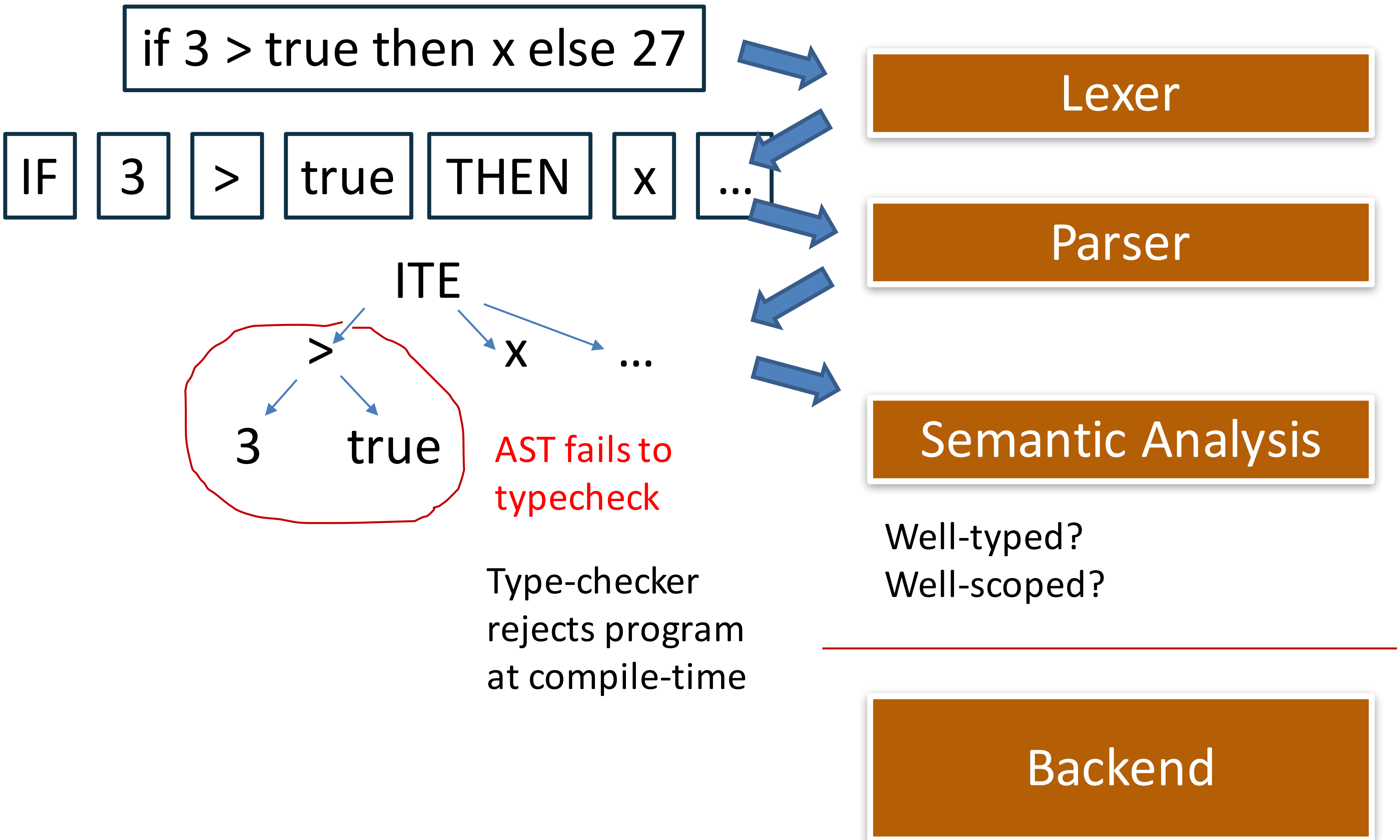
  may yet be **invalid programs** upon further analysis.

- The ***semantic analysis*** phase of a compiler is all about detecting such invalid programs at ***compile-time***, before generating code and running the programs on an actual machine. Major classes of errors:
  - Type errors: Detected by type-checking the program
  - Scope errors: Also typically detected during type-checking

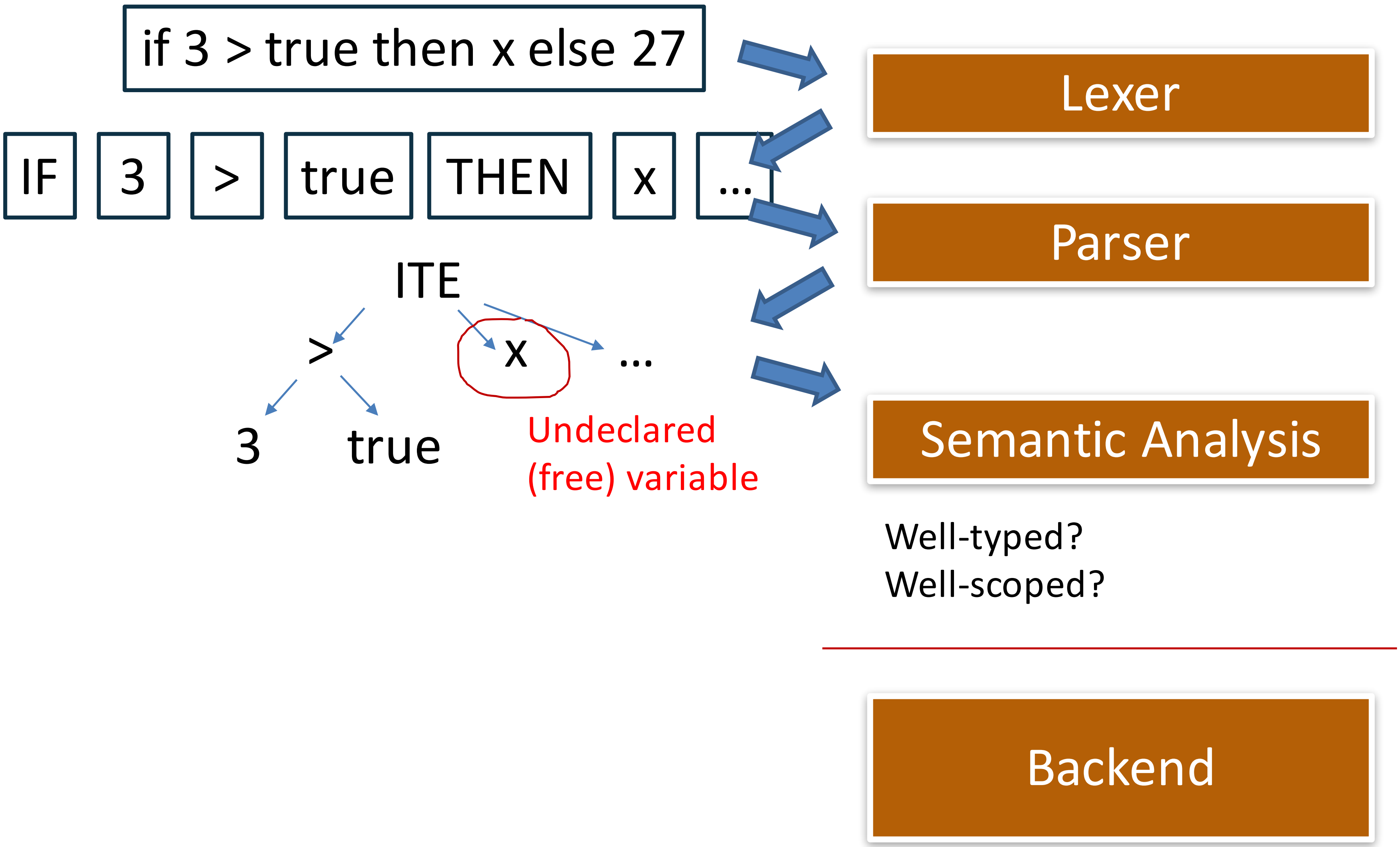
Where have we been? Where are we going?



Semantic Analysis: Type-Checking



Semantic Analysis: Scope Checking



Incorrect Programs

```
let x = 27
in
  let y = 28
  in
    if x then y else x
```

Not well-typed

Incorrect Programs

```
let x = 27
in
  let y = 28
  in
    if x then y else x
```

Not well-typed

Not well-scoped

```
let x = 27
in
  let y = 28
  in
    x + z
```

Incorrect Programs

```
let x = 27
in
 let y = 28
 in
  if x then y else x
```

Not well-typed

Not well-scoped

```
let x = 27
in
 let y = 28
 in
  x + z
```

Not well-scoped

```
let x = 27
in
 let y = y
 in
  x + x
```

Incorrect Programs

```
let x = 27
in
  let y = 28
  in
    if x then y else x
```

Not well-typed

Not well-scoped

```
let x = 27
in
  let y = 28
  in
    x + z
```

Not well-typed

```
let x = 27
in
  let y = x
  in
    y + 3.0
```

Not well-scoped

```
let x = 27
in
  let y = y
  in
    x + x
```

Incorrect Programs

```
let x = 27
in
  let y = 28
  in
    if x then y else x
```

Not well-typed

Not well-scoped

```
let x = 27
in
  let y = 28
  in
    x + z
```

Not well-typed

```
let x = 27
in
  let y = x
  in
    y + 3.0
```

Not well-scoped

```
let x = 27
in
  let y = y
  in
    x + x
```

```
let x = 27
in
  let y = x + 28
  in
    y * y
```

Incorrect Programs

```
let x = 27
in
  let y = 28
  in
    if x then y else x
```

Not well-typed

Not well-scoped

```
let x = 27
in
  let y = 28
  in
    x + z
```

Not well-typed

```
let x = 27
in
  let y = x
  in
    y + 3.0
```

Not well-scoped

```
let x = 27
in
  let y = y
  in
    x + x
```

```
let x = 27
in
  let y = x + 28
  in
    y * y
```

Actually…I think this one's OK.

Type-Checking

- A few lectures ago, we saw a bunch of ***inference rules****, which* formed the type system for a small language, **Grumpy1**.

```
\Gamma(x) = t
-------------------------------- [T-Var]
\Gamma |- x : t

\Gamma |- e1 : bool     \Gamma |- e2 : t   \Gamma |- e3 : t
--------------------------------------------------------------------------- [T-IfThenElse']
\Gamma |- if e1 then e2 else e3 : t
```

- These rules carved out a subset of programs, the ***well-typed programs***: those with valid derivations according to the rules. Such programs ***did not go wrong at runtime (type safety).***

- ***Type-checker:*** just a program that enforces these rules!

  Throw away programs that are ill-typed; send well-typed programs to the backend of the compiler, for optimization and code generation.

Type-Checking, Practically Speaking

- Makes sure we **never add**, **subtract**, **multiply**, etc. values of ***incorrect or incompatible type***:
  - 7 + 4.0 (*int plus float*)
  - 28 * true (*int times bool*)

- Makes sure we never ***update a reference with a value of the wrong type***:
  - let p = ref 23 in p := 23.0

  Why is this important? Allocated storage may be of the wrong size, or perhaps is misaligned for the updated value.

- Makes sure we call functions with the correct number of arguments, of the expected types:
  - let f (x : int) (y : float) : int = x + 7;; f(false)

Scope

- ***Scoping Rules:*** which identifiers/names are visible at which points in a program? which ids/names refer to the same/different storage?
- The ***(lexical) scope*** of an identifier / name:

  The parts of the program in which it's valid to use that name.

- Scopes in C:
  - **Function scope:** formal parameters, function-local variables
  - **File or translation-unit scope:** static global variables
  - **Whole-program scope**: extern global variables

- ***Lifetime*** is a bit different:
  - The portion of time (during runs of the program) in which a particular identifier is valid.

Scope in OCaml

Algorithm: To resolve the value of a variable **x**, simply look for its closest enclosing definition.

```
let x = 27
in
  let y = 28
  in
    x + z
```

x’s scope

y’s scope

Scope in OCaml

Algorithm: To resolve the value of a variable **x**, simply look for its closest enclosing definition.

```
let x = 27
in
  let y = 28
  in
   x + z
```

x's scope

y's scope

```
let f(x) = x*x in print_int (f 3);;

let f(x) = x+x;;

let g(x) = let f x = x-x in x + f(x);;

print_int (f 1); print_int (g 3);;
```

First f's scope

Third f's scope

Second f's scope

Dynamic Scoping

- Some -- bad ☺ -- languages implement so-called dynamic scoping (**perl**, **elisp, SNOBOL, APL**)

- ***Main strategy:*** to resolve a variable, look for it in the current function's stack frame, then in the caller's stack frame, and so on until you find a definition.*
- Problems?
  - Much more difficult for programmer (and also compiler) to understand what the heck is going on
  - The value of a variable depends not just on the program, but also on the context in which it is called!

* Good examples: https://courses.cs.washington.edu/courses/cse341/09wi/general-concepts/scoping.html

SYMBOL TABLES

Symbol Tables

- Both type-checking and scope checking rely on properties of ***identifiers*** that appear in the program.
  - Type-checking: When type-checking the body of a **let**, what's the type of the identifier **x** that was bound above?

```
let x = true
in if x && false then 3 else 4
```

In our typing rules, we track this info using Γ!

  - Scope checking: Which identifiers are in scope at a particular program point?

- ***Symbol table:*** a data structure for keeping track of this information during the analysis of a program

Symbol Tables

- One way to think of the symbol table:

Variable	Type
x	int
y	bool

Symbol Tables

- One way to think of the symbol table:

Variable	Type
x	int
y	bool

- ***Another:*** a data structure α ***symtab*** that implements the following three operations:
  - ***Create***: () -> α symtab
  - ***Get:*** id -> α symtab -> α option
  - ***Set:*** id -> α -> α symtab -> α symtab

Symbol Tables

- One way to think of the symbol table:

Variable	Type
x	int
y	bool

- ***Another:*** a data structure α ***symtab*** that implements the following three operations:
  - ***Create***: () -> α symtab
  - ***Get:*** id -> α symtab -> α option
  - ***Set:*** id -> α -> α symtab -> α symtab

- ***Many ways to implement this interface:***
  - ***Imperative:*** Hash Table
  - ***Functional:*** Balanced Binary Tree (e.g., RB or AVL)

Implementing Scope

- One symbol table for each scope!

```
let x = 27
in
  let y = 28
  in
    x + z
```

Variable	Type
y	int

Variable	Type
x	int

- To check whether a variable is in scope
  - Look up in symbol table associated with current scope
  - If not there, go up one level
  - If variable in ***no*** table, then program is ill-scoped

Shadowing

- One symbol table for each scope!

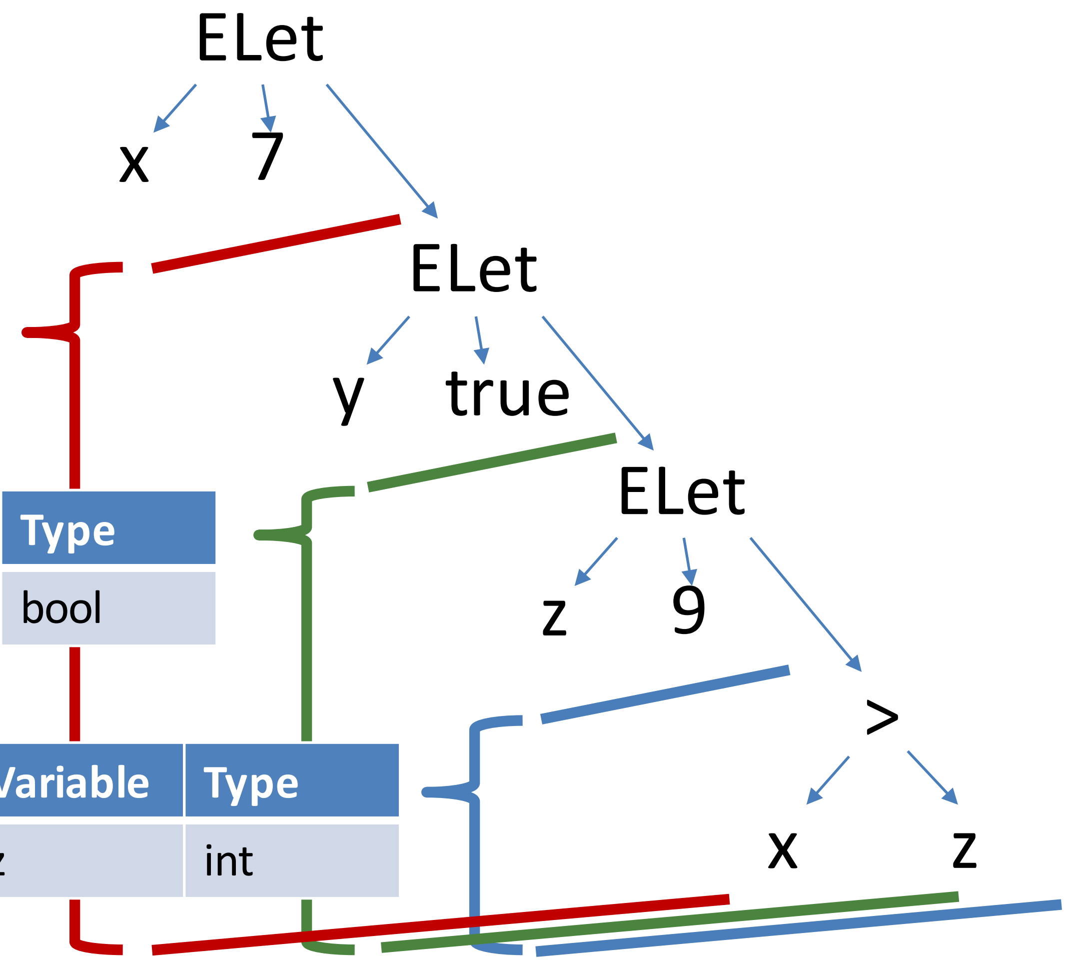
```
let x = false
in
  let x = 28
  in
    x + x
```

Variable	Type
x	int

Variable	Type
x	bool

- To check whether a variable is in scope
  - Look up in symbol table associated with current scope
  - If not there, go up one level
  - If variable in ***no*** table, then program is ill-scoped

AST View of Scope



Variable	Type
x	int

Variable	Type
y	bool

Variable	Type
z	int

Summary

- Many programs that are ***syntactically valid*** according to context-free description of a language may yet be invalid upon further ***semantic analysis***:
  - Not well-typed
    - Apply operators to operands of invalid or incompatible types
    - Call functions with ill-typed (or the wrong number of) args
  - Not well-scoped
    - Refer to identifiers that have not been bound, either by a let declaration or as the parameter of a function
- ***Type-checkers:*** Check that programs are well-typed and well-scoped, according to the inference rules of a type system
- ***Symbol tables:*** An important data structure for maintaining information about identifiers during type-checking; the analog in type inference rules being the type context Γ

TYPE-CHECKING CASE STUDY

https://github.com/gstew5/cs4100-public/tree/master/tyckeck-example