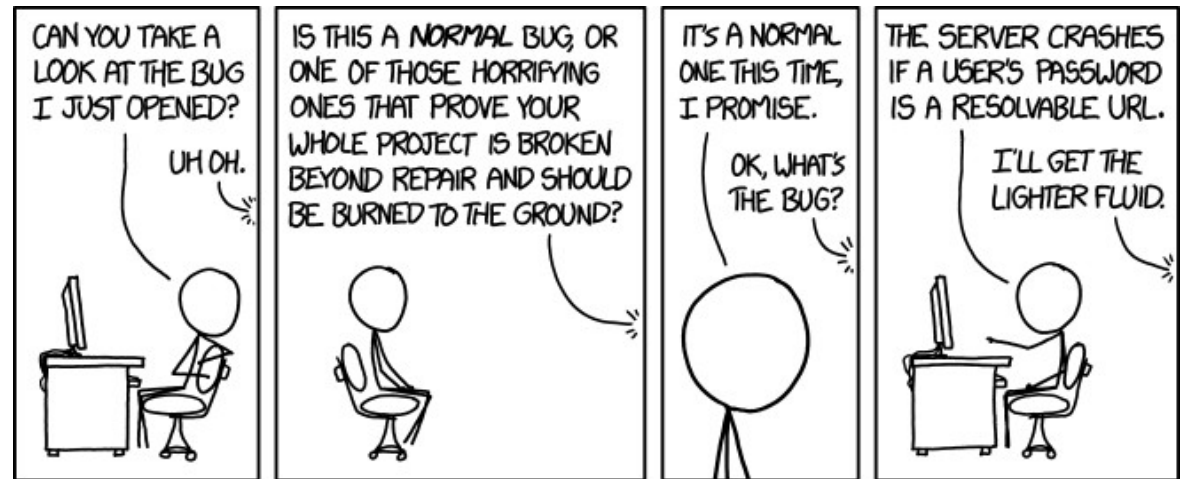


CS 432

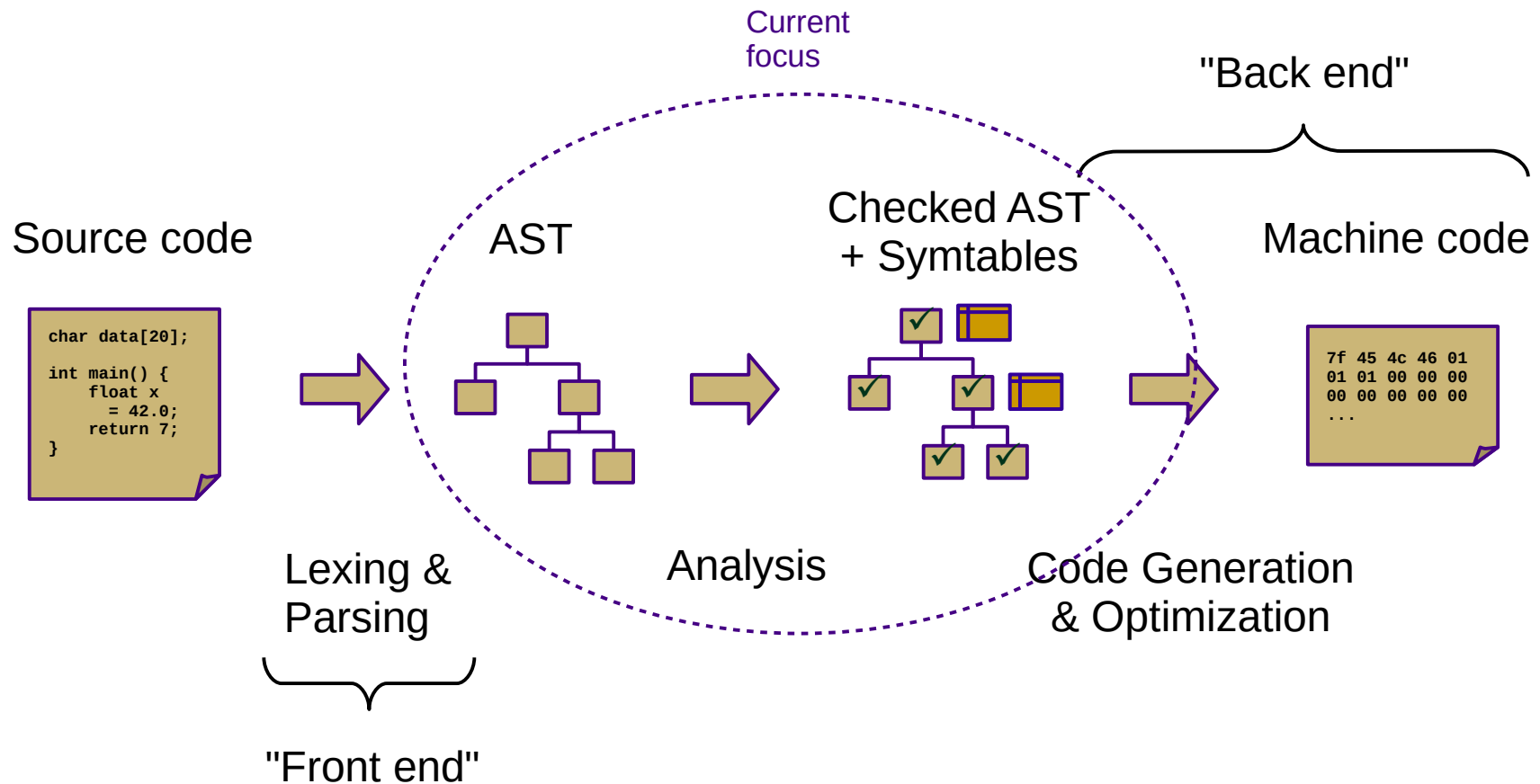
Fall 2023

Mike Lam, Professor



Static Analysis

Compilation



Analysis goal: reject as many incorrect programs as possible at the AST level before attempting code generation

Overview

- **Syntax**: *form* of a program
 - Described using regular expressions and context-free grammars
- **Semantics**: *meaning* of a program
 - Much more difficult to describe clearly
 - Described using type systems and language reference specifications

Valid character strings (identified by I/O system)

Valid sequences of Decaf tokens (identified by lexer)

Syntactically-valid Decaf programs (identified by parser)

Semantically-valid Decaf programs (identified by analysis)

Correct Decaf programs (identified by ???)

Static Analysis

- Goal: reject incorrect programs
- Problem: checking semantics is hard!
 - In general, we won't be able to check for full correctness
 - However, some aspects of semantics can be robustly encoded using **types** and **type systems**
 - We will also implement other rudimentary correctness checks
 - E.g., Decaf programs must have a “main” function

Static Analysis

- **Sound** vs. **complete** static analysis
 - A “sound” system has no false positives
 - All errors reported are true errors
 - A “complete” system has no false negatives
 - All true errors are reported
- Most static analysis is **sound** but not **complete**
 - A lack of type errors does not mean the program is correct
 - However, the presence of a type error generally does mean that the program is NOT correct

Types

- A **type** is an abstract category characterizing a range of data values
 - Base types: integer, character, boolean, floating-point
 - Enumerated types (finite list of constants)
 - Pointer types (“address of X”)
 - Array or list types (“list of X”)
 - Compound/record types (named collections of other types)
 - Function types: (type1, type2, type3) → type4

Static Analysis

- **Type inference** is the process of assigning types to expressions
 - This information must be “inferred” if it is not explicit
 - For Decaf, every expression has an unambiguous inferred type!
 - Conclusions of the type rules – assume the premises are true
- **Type checking** is the process of ensuring that a program has no type-related errors
 - Ensure that operations are supported by a variable's type
 - Ensure that operands are of *compatible* types
 - This could happen at compile time (**static**) or at run time (**dynamic**)
 - A type error is usually considered a bug
 - For Decaf, almost every ASTNode type will have some kind of check

Type Compatibility

- Rules about **type compatibility** define types that can be used together in expressions, assignments, etc.
 - Sometimes this may require a type *conversion*
- Two types are **name-equivalent** if their names are identical
- Two types are **structurally-equivalent** if
 - They are the same basic type or
 - They are recursively structurally-equivalent

C example:

```
typedef unsigned char byte_t;
unsigned char a;           // types of a and b are structurally-
byte_t b;                 // equivalent but not name equivalent
```


Type Conversions

- Implicit vs. explicit
 - **Implicit** conversions are performed automatically by the compiler (“coercions”)
 - E.g., `double x = 2;`
 - **Explicit** conversions are specified by the programmer (“casts”)
 - E.g., `int x = (int)1.5;`
- Narrowing vs. widening
 - **Widening** conversions preserve information
 - E.g., `int → long`
 - **Narrowing** conversions may lose information
 - E.g., `float → int`

Advanced Type Inference

- **Polymorphism**: literally “taking many forms”
 - A *polymorphic* construct supports multiple types
 - **Subtype polymorphism**: object inheritance
 - **Function polymorphism**: overloading
 - **Parametric polymorphism**: generic type identifiers
 - E.g., templates in C++ or generics in Java
 - During type inference, create type variables and unify type variables with concrete types
 - Some type variables might remain unbound
 - E.g., `len : ([a]) → int`
 - E.g., `map : ((a → b), [a]) → [b]`

In Haskell:

```
len l = case l of
  []      → 0
  (x:xs) → 1 + (len xs)
```

```
map f l = case l of
  []      → []
  (x:xs) → (f x):(map f xs)
```

Problem

- Inferring the type of an ASTLiteral is easy

$$\text{TDec} \frac{}{\vdash \text{DEC} : \text{int}} \quad \text{THex} \frac{}{\vdash \text{HEX} : \text{int}} \quad \text{TStr} \frac{}{\vdash \text{STR} : \text{str}}$$

$$\text{TTrue} \frac{}{\vdash \text{true} : \text{bool}} \quad \text{TFalse} \frac{}{\vdash \text{false} : \text{bool}}$$

- How do we infer the type of an ASTLocation?

$$\text{TLoc} \frac{\text{ID} : \tau \in \Gamma}{\Gamma \vdash \text{ID} : \tau}$$

- Need information about Γ (type environment)
- Systems core theme: **Information = Bits + Context**

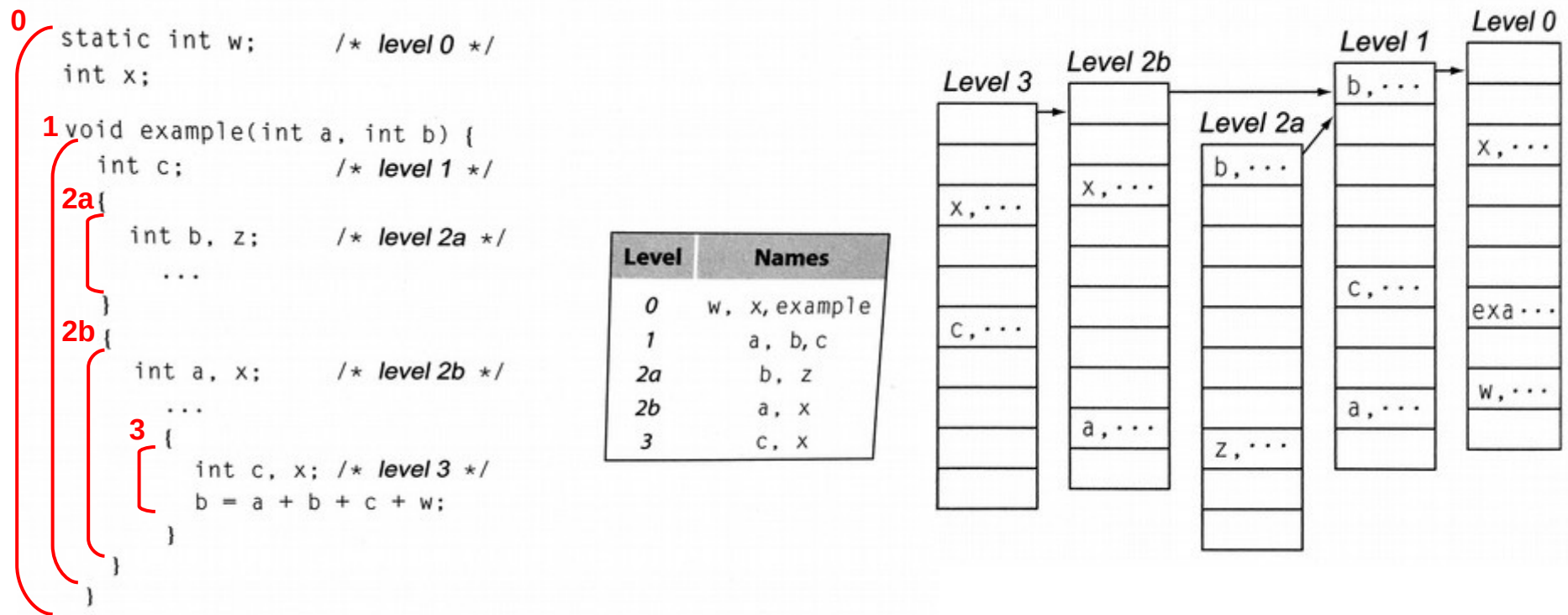
Symbols

- A **symbol** is a single name in a program
 - What kind of value is it: variable or function?
 - If it is a variable:
 - What is its type? How big is it?
 - Where is it stored?
 - How long must its value be preserved?
 - Who is responsible for allocating, initializing, and de-allocating it?
 - If it is a function:
 - What parameters (name and type) does it take?
 - What type does it return?

Symbol Tables

- A **symbol table** stores info about symbols during compilation
 - Aggregates information from (potentially) distant parts of code
 - Maps symbol names to symbol information
 - Often implemented using hash tables
 - Usually one symbol table per scope
 - Each table contains a pointer to its parent (next larger scope)
- Supported operations
 - **Insert** (name, record) – add a new symbol to the current table
 - **LookUp** (name) – retrieve information about a symbol

Symbol Table Example

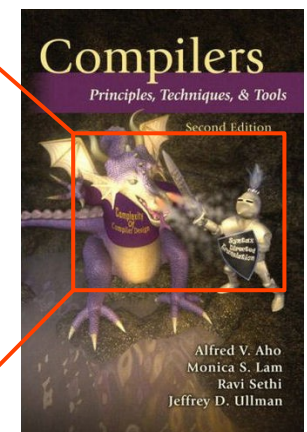
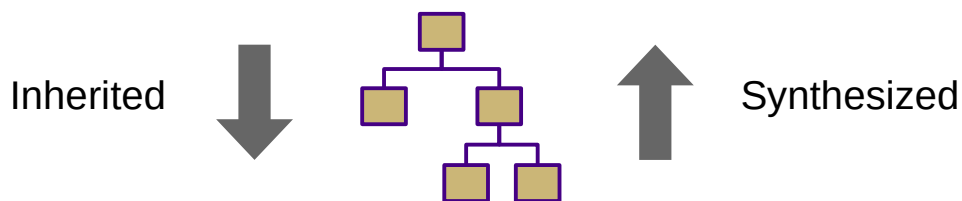


■ FIGURE 5.10 Simple Lexical Scoping Example in C.

NOTE: For Decaf, we will have two scopes for each function, one associated with the FuncDecl (for parameters) and one associated with the body Block (for local variables).

AST Attributes

- An AST **attribute** is an additional piece of information
 - Often used to store data useful to multiple passes
 - Aside: some translations can be done purely using attributes
 - **Syntax-directed translation** (original dragon book!)
 - Modern translation is often too complex for this to be feasible
 - **Inherited** vs. **synthesized** attributes
 - Inherited attributes depend only on parents/siblings
 - Synthesized attributes depend only on children



Attribute Grammars

- Some synthesized attributes can be calculated using post-visit rules in a grammar

Production	Attribution Rules
$Block_0 \rightarrow Block_1 Assign$	$\{ Block_0.cost \leftarrow Block_1.cost + Assign.cost \}$
$Assign$	$\{ Block_0.cost \leftarrow Assign.cost \}$
$Assign \rightarrow name = Expr;$	$\{ Assign.cost \leftarrow Cost(store) + Expr.cost \}$
$Expr_0 \rightarrow Expr_1 + Term$	$\{ Expr_0.cost \leftarrow Expr_1.cost + Cost(add) + Term.cost \}$
$Expr_1 - Term$	$\{ Expr_0.cost \leftarrow Expr_1.cost + Cost(sub) + Term.cost \}$
$Term$	$\{ Expr_0.cost \leftarrow Term.cost \}$
$Term_0 \rightarrow Term_1 \times Factor$	$\{ Term_0.cost \leftarrow Term_1.cost + Cost(mult) + Factor.cost \}$
$Term_1 \div Factor$	$\{ Term_0.cost \leftarrow Term_1.cost + Cost(div) + Factor.cost \}$
$Factor$	$\{ Term_0.cost \leftarrow Factor.cost \}$
$Factor \rightarrow (Expr)$	$\{ Factor.cost \leftarrow Expr.cost \}$
num	$\{ Factor.cost \leftarrow Cost(loadI) \}$
$name$	$\{ Factor.cost \leftarrow Cost(load) \}$

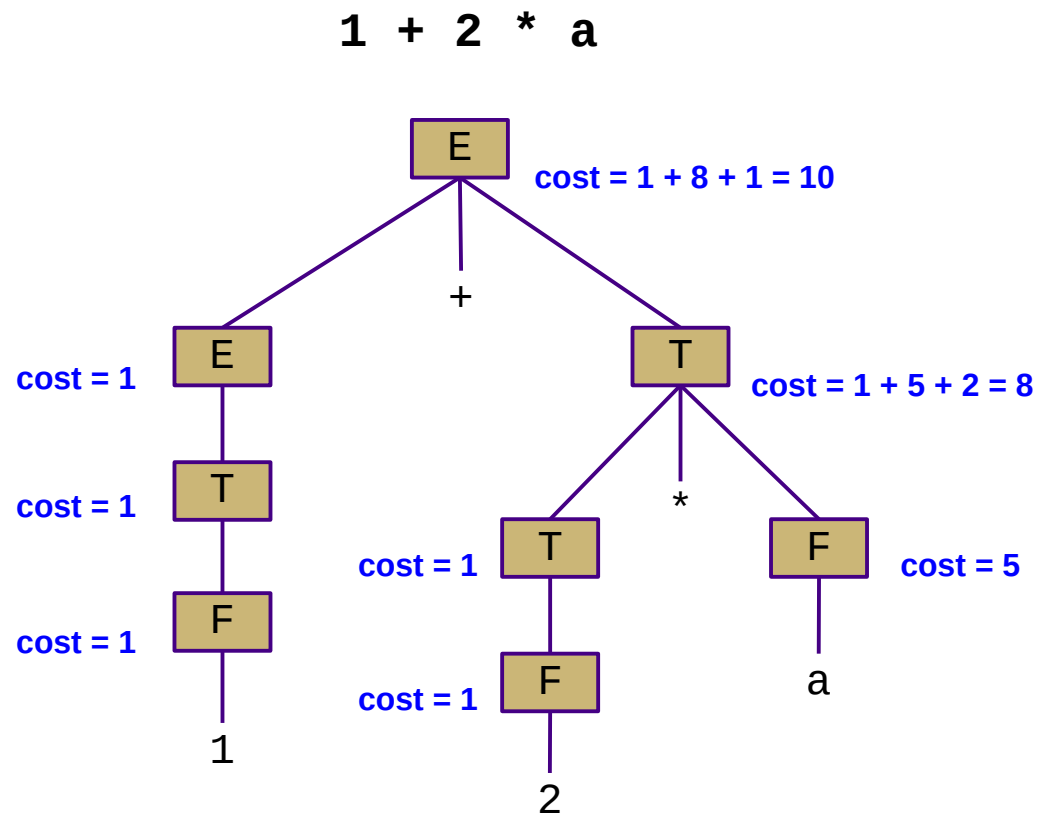
■ FIGURE 4.8 Simple Attribute Grammar to Estimate Execution Time.

Example

$E \rightarrow E_1 + T$ { $E.cost = E_1.cost + T.cost + 1$ }
| T { $E.cost = T.cost$ }

$T \rightarrow T_1 * F$ { $T.cost = T_1.cost + F.cost + 2$ }
| F { $T.cost = F.cost$ }

$F \rightarrow (E)$ { $F.cost = E.cost$ }
| ID { $F.cost = 5$ }
| DEC { $F.cost = 1$ }



Attributes in P3 and P4

Key	Description
parent	Uptree parent ASTNode reference
depth	Tree depth (int)
symbolTable	Symbol table reference (only in program, function, and block nodes)
type	DecafType of node (only in expression nodes)
staticSize	Size (in bytes as int) of global variables (only in program node)
localSize	Size (in bytes as int) of local variables (only in function nodes)
code	ILOC instructions generated from the subtree rooted at this node
reg	Register storing the result of the expression rooted at this node (only in expression nodes)

P3 (circled in red) highlights the **depth** and **symbolTable** attributes.

P4 (circled in red) highlights the **staticSize**, **localSize**, **code**, and **reg** attributes.

Building Symbol Tables (pre-P3)

- Walk the AST, creating linked tables using a stack
 - Create new symbol table for each scope
 - Every Program, FuncDecl, and Block
 - Caveat: every function contains a function-wide block for local vars, so the function level symbol table will ONLY contain the function parameters
 - Store tables as an attribute (“symbolTable”) in AST nodes
 - Add all symbol information
 - Global variables go in Program table (including arrays)
 - Function symbols go in Program table
 - Function parameters go in FuncDecl table
 - Local variables go in Block table

Static Analysis (P3)

- Walk the AST, checking correctness properties
 - Infer the types of all expressions (pre-visits)
 - Use symbol table lookups where necessary
 - Store in “type” attribute
 - Verify all types are correct (post-visits)
 - Refer to type rules (section 6 of language reference)
 - May require checking “type” attribute of children
 - May require symbol table lookups
 - May require maintaining some state (e.g., current function)
 - Verify other properties of correct programs (post-visits)
 - Example: break and continue should only occur in while loops
 - **Re-read Decaf reference carefully for these**

Decaf Example

```
bool g;  
  
def int add(int x, int y)  
{  
    return x + y;  
}  
  
def int main()  
{  
    int a;  
    a = 3;  
    return add(a, 2);  
}
```

Program
add : (int, int) -> int
main : () -> int
g : bool

FuncDecl name="add"
x : int
y : int

Block

FuncDecl name="main"

Block
a : int

Decaf Example

```
def int add(int x, int y)
{
  return x + y;
}
```

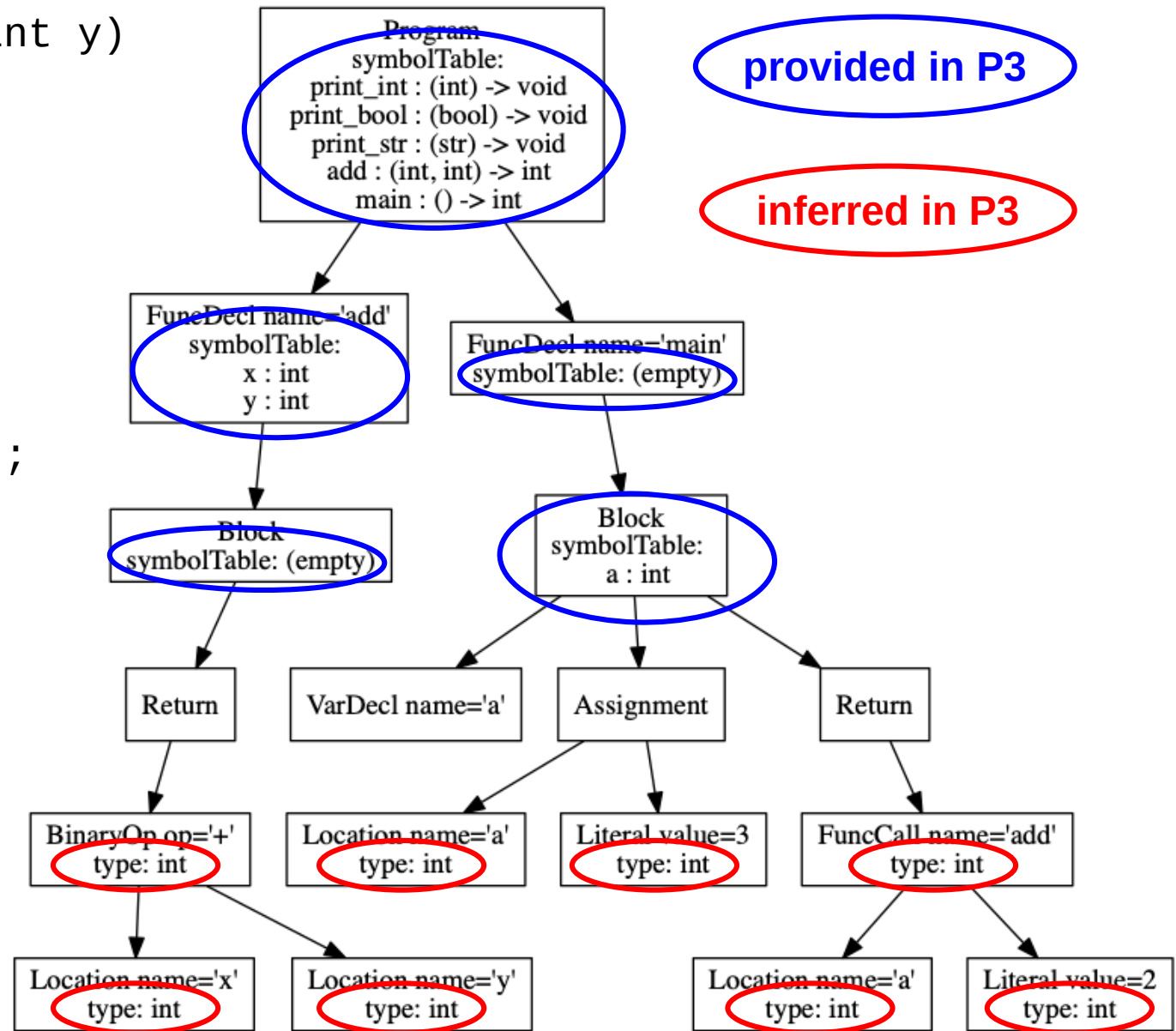
```
def int main()
{
  int a;
  a = 3;
  return add(a, 2);
}
```

$$\text{TDec} \frac{}{\vdash \text{DEC} : \text{int}}$$

$$\text{TLoc} \frac{\text{ID} : \tau \in \Gamma}{\Gamma \vdash \text{ID} : \tau}$$

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

$$\text{TAssign} \frac{\Gamma \vdash \text{ID} : \tau \quad \Gamma \vdash e : \tau}{\Gamma \vdash \text{ID} '=' e ';'}$$

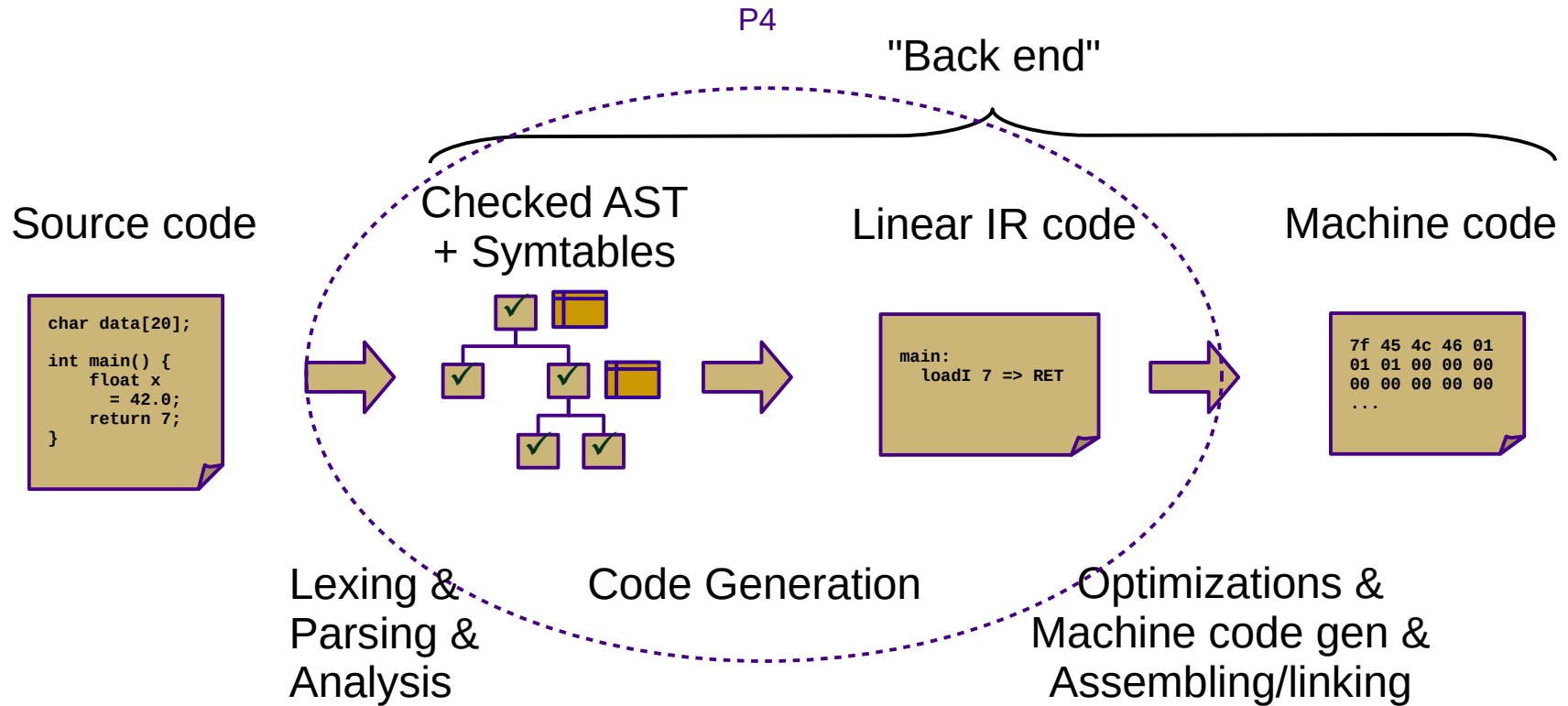


P3 reminder

- Check your implementation against the reference compiler (/cs/students/cs432/f23/decaf)
 - If the reference compiler rejects a program, you should too (and vice versa for correct programs)
 - Use “- -fdump-tables” to print the symbol tables
 - Also, the graphical AST should have the tables now (both in the reference compiler and in your project)

*Optional challenge: write P3 using a “pure” visitor; i.e., the visitor methods perform **no tree traversals**, only symbol lookups and accesses of direct child attributes.*

Preview: P4



Allocating Symbols (pre-P4)

- Walk the AST, allocating memory for symbols
 - Each symbol has a location and offset field
 - This is a form of **static coordinates**
 - `STATIC_VAR` and static offset for global variables
 - `STACK_LOCAL` and BP offset for local variables
 - `STACK_PARAM` and BP offset for function parameters
 - Track allocated memory
 - `localSize` attribute for each `FuncDecl`
 - `staticSize` attribute for the Program

Code Generation (P4)

- Walk the AST, generating code
 - Build ILOC instructions for all nodes
 - Refer to operational semantics (section 7 of language reference)
 - Store in “code” attribute
 - May require copying “code” attribute of children
 - Store expression results in temporary registers
 - Use “reg” attribute
 - Need state information to track the next temporary ID
 - Location loads and stores will require static coordinate info