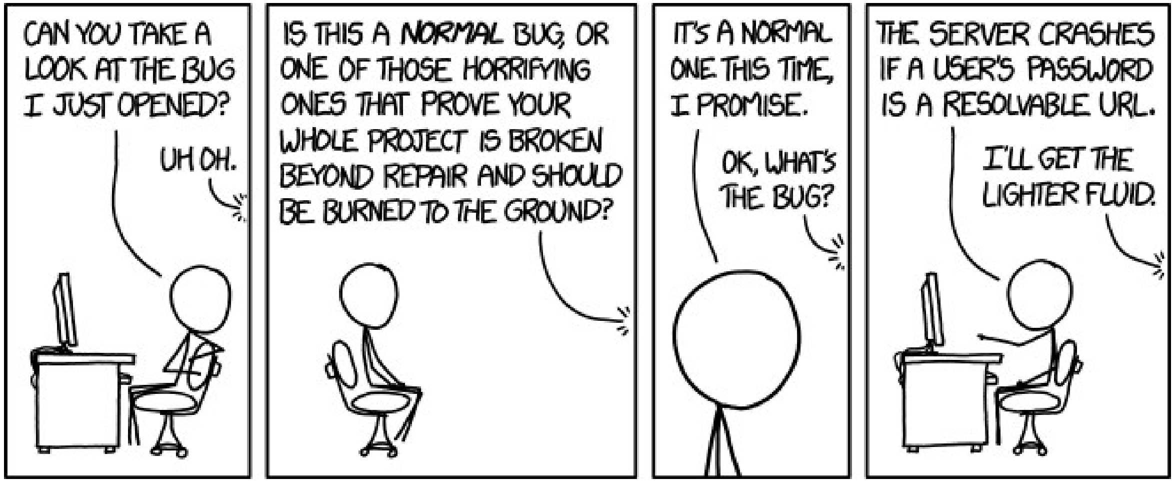
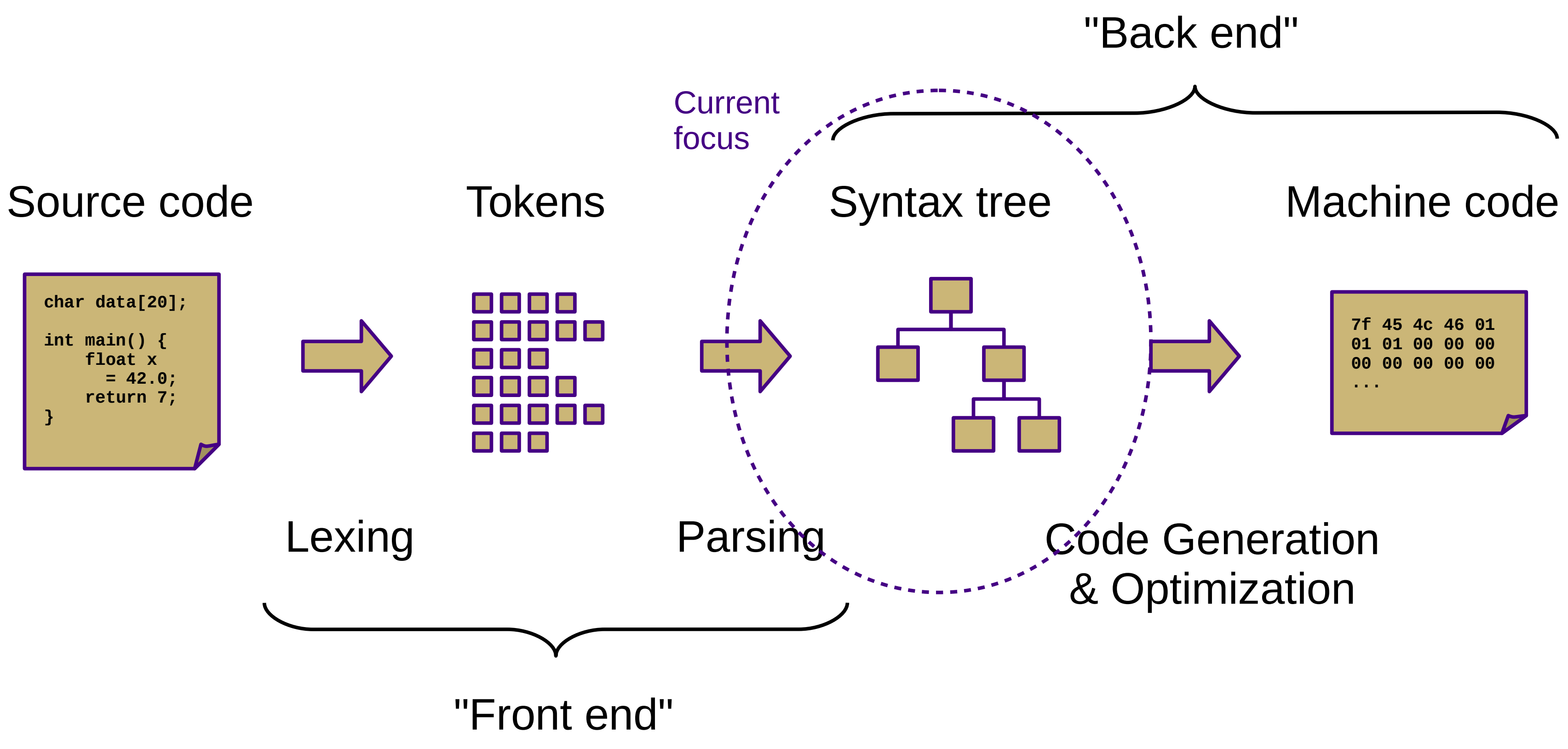
CS 432
Fall 2018

Mike Lam, Professor



Static Analysis

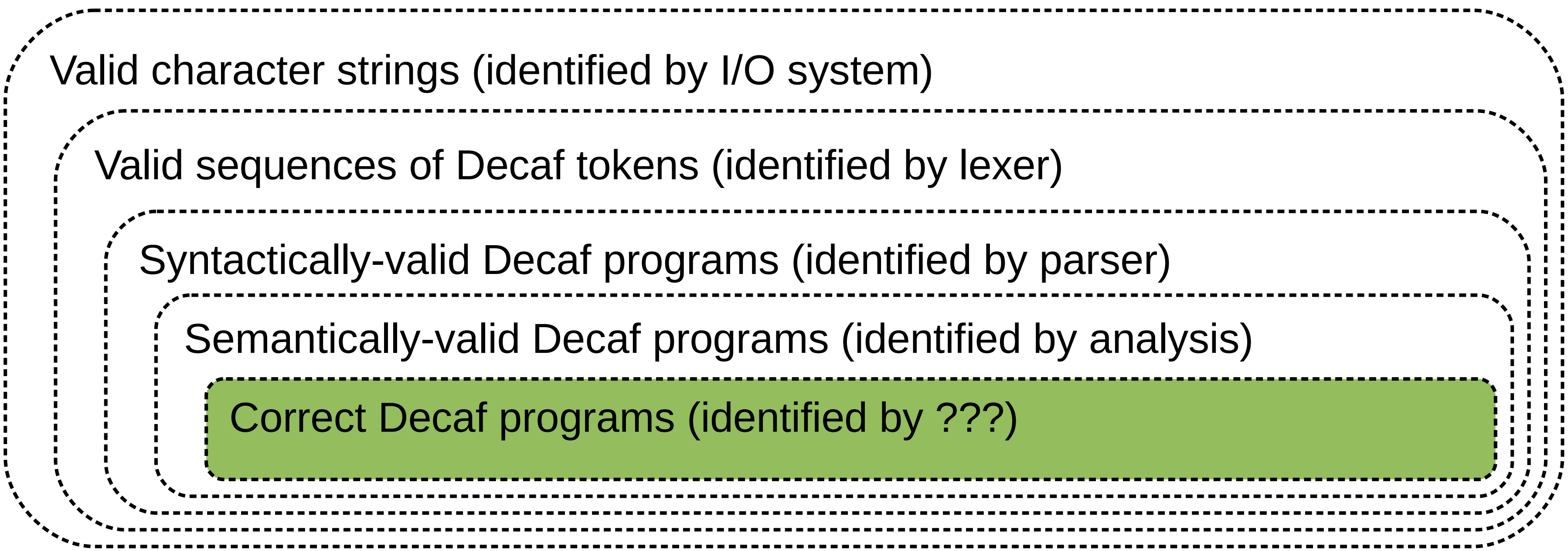
Compilation



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 **informally** using abstract syntax trees



Aside: Semantic approaches

- Three main **formal** approaches:
  - *Operational* semantics
  - *Axiomatic* semantics
  - *Denotational* semantics

Operational Semantics

- Describe a program's effects using a simpler language that is closer to the hardware

```
for (i=0; i<n; i++) {
  m *= i;
}
```

```
       i=0;
loop:  if i>=n goto done
       m *= i
       i++
       goto loop
done:
```

```
for (e1; e2; e3) {
  e4
}
```

```
       e1
loop:  if !e2 goto done
       e4
       e3
       goto loop
done:
```

Axiomatic Semantics

- Express programs as proof trees
  - Loops can be difficult to handle

$$\frac{\{P \wedge e1\}\ e2\ \{Q\} \qquad \{P \wedge \neg e1\}\ e3\ \{Q\}}{\{P\}\ \textbf{if}\ e1\ \textbf{then}\ e2\ \textbf{else}\ e3\ \{Q\}}\ \text{SConditional}$$

$$\frac{\dfrac{\ldots}{\{x=10 \wedge x>5\}\ y:=3\ \{x=10 \wedge y=3\}}\ \text{SAssign}}{\{x=10\}\ \textbf{if}\ x > 5\ \textbf{then}\ y := 3\ \textbf{else}\ y := 7\ \{x=10 \wedge y=3\}}\ \text{SConditional}$$

Denotational Semantics

- Describes a program's results using functions
  - Must also track system state

```
eval :: (Program, State) → (Value, State)

eval(e1 + e2, S) =
    let (v1, S')  = eval(e1, S)  in
    let (v2, S'') = eval(e2, S') in
    (v1 + v2, S'')

eval(while e1 do e2, S) =
    let (v, S') = eval(e1, S) in
    if not v then
        (v, S')
    else let (_, S'') = eval(e2, S')
        eval(while e1 do e2, S'')
```

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

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
- 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

Type Systems

- A type system is a set of type rules
  - Rules: valid types, type compatibility, and how values can be used
  - “Strongly typed” if every expression can be assigned an unambiguous type
  - “Statically typed” if all types can be assigned at compile time
  - “Dynamically typed” if some types can only be discovered at runtime
- Benefits of a robust type system
  - Earlier error detection
  - Better documentation
  - Increased modularization

Type Checking

- Type inference is the process of assigning types to expressions
  - This information must be “inferred” if it is not explicit
  - For Decaf, every ASTExpression has an unambiguous inferred type!
    - Conclusions of the type proofs – 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 (for static type systems) or at run time (for dynamic type systems)
  - A type error is usually considered a bug
  - For Decaf, almost every ASTNode child class will have some kind of check

Type Checking

- Sound vs. complete type checking
  - 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 type checking is **sound** but not **complete**
  - The 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

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]`

Symbols

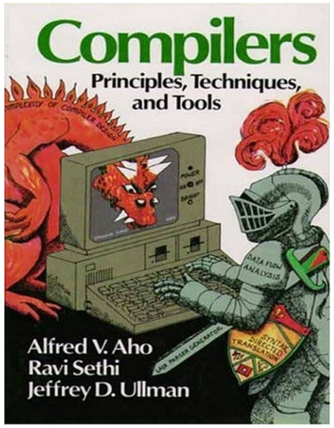
- A symbol is a single name in a program
  - What type of value is it?
  - If it is a variable:
    - How big is it?
    - Where is it stored?
    - How long must its value be preserved?
    - Who is responsible for allocating, initializing, and de-allocating it?
  - If it is a function:
    - What parameters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

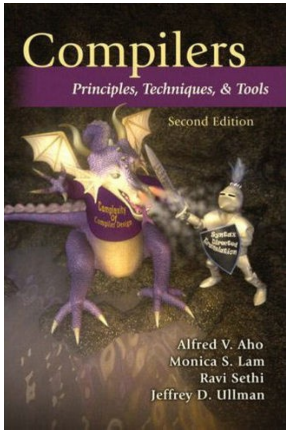
Symbol Tables

- A symbol table stores information 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

AST attributes

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





Attributes in P4 and P5

Fields

Modifier and Type	Field and Description
`Map<String,Object>`	`attributes` Generic key/value store.

Potential attributes

Key	Description
`parent`	Uptree parent `ASTNode` reference
`depth`	Tree depth (`int`)
`source`	`SourceInfo` reference
`symbolTable`	`SymbolTable` reference (only in `ASTProgram`, `ASTFunction`, and `ASTBlock`)
`type`	`ASTNode.DataType` of node (only in `ASTExpression` subclasses)
`staticSize`	Size (in bytes) of global variables (only in `ASTProgram`)
`localSize`	Size (in bytes) of local variables (only in `ASTFunction`)

Building Symbol Tables (P4)

- Walk the AST, creating linked tables using a stack
  - Create new symbol table for each scope
    - Global symbols in ASTProgram
    - Function local symbols in ASTFunction
    - Block-local symbols in ASTBlock
    - 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 ASTProgram table (including arrays)
    - Function symbols go in ASTProgram table
    - Function parameters go in ASTFunction table
    - Local variables go in ASTBlock table

Static Analysis (P4)

- 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
    - 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
    - Full list on the project website

P4 reminder

- Check your implementation against the reference compiler (`decaf-1.0.jar`)
  - 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:" it is possible to write P4 using a "pure" visitor; i.e., the visitor methods perform no tree traversals aside from symbol lookups and accessing child attributes.