Ch 13 Semantic Analysis III

(Static Semantics)

Yongjun Park

Hanyang University

Static Semantics

- **Can describe the types used in a program**
- **How to describe type checking**
- **Static semantics: Formal description for the programming language**
- **Is to type checking:**
  - As grammar is to syntax analysis
  - As regular expression is to lexical analysis
- **Static semantics defines types for legal ASTs in the language**

Type Judgments or Relations

- **Static semantics = formal notation which describes type judgments:**
  - E : T
  - means “E is a well-typed expression of type T”
  - E is typable if there is some type T such that E : T
- **Type judgment examples:**
  - 2 : int
  - true : bool
  - 2 * (3 + 4) : int
  - “Hello” : string

Type Judgments for Statements

- **Statements may be expressions (i.e., represent values)**
- **Use type judgments for statements:**
  - if (b) 2 else 3 : int
  - x == 10 : bool
  - b = true, y = 2 : int   (result of comma operator is the value of the rightmost expression)
- **For statements which are not expressions: use a special unit type (void or empty type)**
  - S : unit
  - means “S is a well-typed statement with no result type”

Class Problem

Whats the type of the following statements?

Assume i* are int variables, f* are float variables

f1 [3]

i = i1 [i2]

while (i < 10) do S1

(i ? 0) 4.0 : 1.0

Deriving a Judgment

- **Consider the judgment**
  - if (b) 2 else 3 : int
- **What do we need to decide that this is a well-typed expression of type int?**
  - b must be a bool (b : bool)
  - 2 must be an int (2 : int)
  - 3 must be an int (3 : int)

Type Judgements

- **Type judgment notation: $A \vdash E : T$**
  - Means "In the context A, the expression E is a well-typed expression with type T"
- **Type context is a set of type bindings: id : T**
  - (i.e. type context = symbol table)

  - b: bool, x: int $\vdash$ b: bool
  - b: bool, x: int $\vdash$ if (b) 2 else x : int
  - $\vdash$ 2 + 2 : int

Deriving a Judgment

- **To show**
  - b: bool, x: int $\vdash$ if (b) 2 else x : int
- **Need to show**
  - b: bool, x: int $\vdash$ b : bool
  - b: bool, x: int $\vdash$ 2 : int
  - b: bool, x: int $\vdash$ x : int

General Rule

- **For any environment A, expression E, statements S1 and S2, the judgement:**
  - $A \vdash$ if (E) S1 else S2 : T
- **Is true if:**
  - $A \vdash E : bool$
  - $A \vdash S1 : T$
  - $A \vdash S2 : T$

Inference Rules

<u>if-rule</u>

premises

$$\frac{A \vdash E : \text{bool} \qquad A \vdash S1 : T \qquad A \vdash S2 : T}{A \vdash \text{if } (E)\ S1 \text{ else } S2 : T}$$

conclusion

- *Read as, "if we have established the statements in the premises listed above the line, then we may derive the conclusion below the line"*
- Holds for any choice of E, S1, S2, T

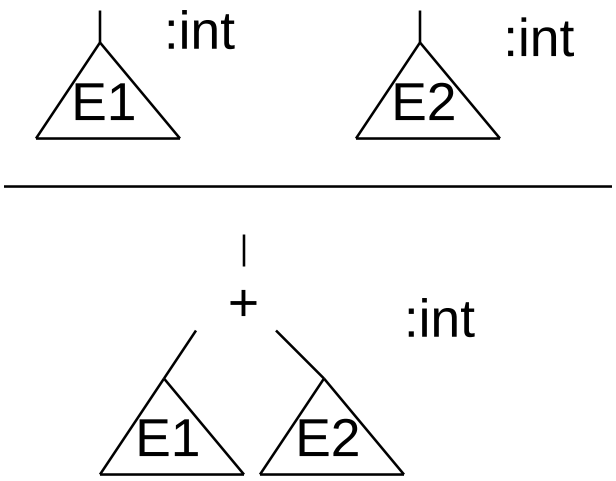
Why Inference Rules?

- Inference rules: compact, precise language for specifying static semantics
- Inference rules correspond directly to recursive AST traversal that implements them
- Type checking is the attempt to prove type judgments $A \vdash E : T$ true by walking backward through the rules

Meaning of Inference Rule

- **Inference rule says:**
  - Given the premises are true (with some substitutions for A, E1, E2)
  - Then, the conclusion is true (with consistent substitution)

$$\frac{A \vdash E1 : \text{int} \qquad A \vdash E2 : \text{int}}{A \vdash E1 + E2 : \text{int}}\ (+)$$



Proof Tree

- **Expression is well-typed if there exists a type derivation for a type judgment**
- **Type derivation is a proof tree**
- **Example: if A1 = b : bool, x : int, then:**

$$\dfrac{\dfrac{A1 \vdash b : bool}{A1 \vdash !b : bool} \qquad \dfrac{A1 \vdash 2 : int \qquad A1 \vdash 3 : int}{A1 \vdash 2 + 3 : int} \qquad A1 \vdash x : int}{b : bool, x : int \vdash \text{if } (!b)\ 2 + 3 \text{ else } x : int}$$

More About Inference Rules

- **No premises = axiom**

$$\frac{}{A \vdash true : bool}$$

- **A goal judgment may be proved in more than one way**

$$\frac{A \vdash E1 : float \qquad A \vdash E2 : float}{A \vdash E1 + E2 : float} \qquad \frac{A \vdash E1 : float \qquad A \vdash E2 : int}{A \vdash E1 + E2 : float}$$

- **No need to search for rules to apply – they correspond to nodes in the AST**

Class Problem

Given the following syntax for arithmetic expressions:

t ::=

- true
- false
- if t then t else t
- 0
- succ t
- pred t
- iszero t

And the following typing rules for the language:

true : bool

false : bool

$$\frac{\text{t1: bool} \quad \text{t2: T} \quad \text{t3 : T}}{\text{if t1 then t2 else t3 : T}}$$

$$\frac{\text{t1 : int}}{\text{succ t1 : int}}$$

$$\frac{\text{t1 : int}}{\text{pred t1 : int}}$$

$$\frac{\text{t1 : int}}{\text{iszero t1 : bool}}$$

Construct a type derivations to show

(1) if iszero 0 then 0 else pred 0 : int

(2) pred(succ(iszero(succ(pred(0)))) : int

Assignment Statements

$$\frac{id : T \in A \qquad A \vdash E : T}{A \vdash id = E : T}$$

(variable-assign)

$$\frac{A \vdash E3 : T \qquad A \vdash E2 : int \qquad A \vdash E1 : array[T]}{A \vdash E1[E2] = E3 : T}$$

(array-assign)

If Statements

- If statement as an expression: its value is the value of the clause that is executed

$$\frac{A \vdash E : \text{bool} \qquad A \vdash S1 : T \quad A \vdash S2 : T}{A \vdash \text{if } (E)\ S1 \text{ else } S2 : T} \quad \text{(if-then-else)}$$

- If with no else clause, no value, why??

$$\frac{A \vdash E : \text{bool} \qquad A \vdash S : T}{A \vdash \text{if } (E)\ S : \text{unit}} \quad \text{(if-then)}$$

Class Problem

1. **Show the inference rule for a while statement, while (E) S**

2. **Show the inference rule for a variable declaration with initializer, Type id = E**

3. **Show the inference rule for a question mark/colon operator, E1 ? S1 : S2**

Sequence Statements

- **Rule: A sequence of statements is well-typed if the first statement is well-typed, and the remaining are well-typed as well:**

$$\frac{A \vdash S1 : T1 \qquad A \vdash (S2; \ldots ; Sn) : Tn}{A \vdash (S1; S2; \ldots ; Sn) : Tn} \quad \text{(sequence)}$$

Declarations

= unit if no E

$$\frac{A \vdash id : T\ [= E\ ] : T1 \qquad A, id : T \vdash (S2; \ldots ; Sn) : Tn}{A \vdash (id : T\ [= E\ ]; S2; \ldots ; Sn) : Tn} \quad \text{(declaration)}$$

Declarations add entries to the environment (e.g., the symbol table)

Function Calls

- **If expression E is a function value, it has a type $T1 \times T2 \times \ldots \times Tn \rightarrow Tr$**
- **Ti are argument types; Tr is the return type**
- **How to type-check a function call?**
  - E(E1, ..., En)

$$\frac{A \vdash E : T1 \times T2 \times \ldots Tn \rightarrow Tr \qquad A \vdash Ei : Ti \quad (i \in 1 \ldots n)}{A \vdash E(E1, \ldots, En) : Tr} \text{(function-call)}$$

Function Declarations

- **Consider a function declaration of the form:**
  - Tr fun (T1 a1, ... , Tn an) = E
  - Equivalent to:
    - Tr fun (T1 a1, ..., Tn an) {return E;}
- **Type of function body S must match declared return type of function, i.e., E : Tr**
- **But, in what type context?**

Add Arguments to Environment

- **Let A be the context surrounding the function declaration.**
  - The function declaration:
    - Tr fun (T1 a1, ... , Tn an) = E
  - Is well-formed if
    - A, a1 : T1 , ... , an : Tn E : Tr
- **What about recursion?**
  - Need: fun: T1 x T2 x ... x Tn → Tr $\in$ A

Class Problem

Recursive function – factorial

int fact(int x) = if (x == 0) 1 else x * fact(x-1);

Is this well-formed?, if so construct the type derivation

Mutual Recursion

- **Example**
  - int f(int x) = g(x) + 1;
  - int g(int x) = f(x) – 1;
- **Need environment containing at least**
  - f: int → int, g: int → int
  - when checking both f and g
- **Two-pass approach:**
  - Scan top level of AST picking up all function signatures and creating an environment binding all global identifiers
  - Type-check each function individually using this global environment

How to Check Return?

$$\frac{A \vdash E : T}{A \vdash \text{return } E : \text{unit}} \quad \text{(return)}$$

- **A return statement produces no value for its containing context to use**
- **Does not return control to containing context**
- **Suppose we use type unit ...**
  - Then how to make sure the return type of the current function is T??

Put Return in the Symbol Table

- **Add a special entry {return_fun : T} when we start checking the function "fun", look up this entry when we hit a return statement**
- **To check Tr fun (T1 a1, ... , Tn an) { S } in environment A, need to check:**

$$A, a1 : T1, ..., an : Tn, return\_fun : Tr \qquad A : Tr$$

$$\frac{A \vdash E : T \qquad return\_fun : T \in A}{A \vdash return\ E\ : unit} \quad \text{(return)}$$

Static Semantics Summary

- Static semantics = formal specification of type-checking rules
- Concise form of static semantics: typing rules expressed as inference rules
- Expression and statements are well-formed (or well-typed) if a typing derivation (proof tree) can be constructed using the inference rules

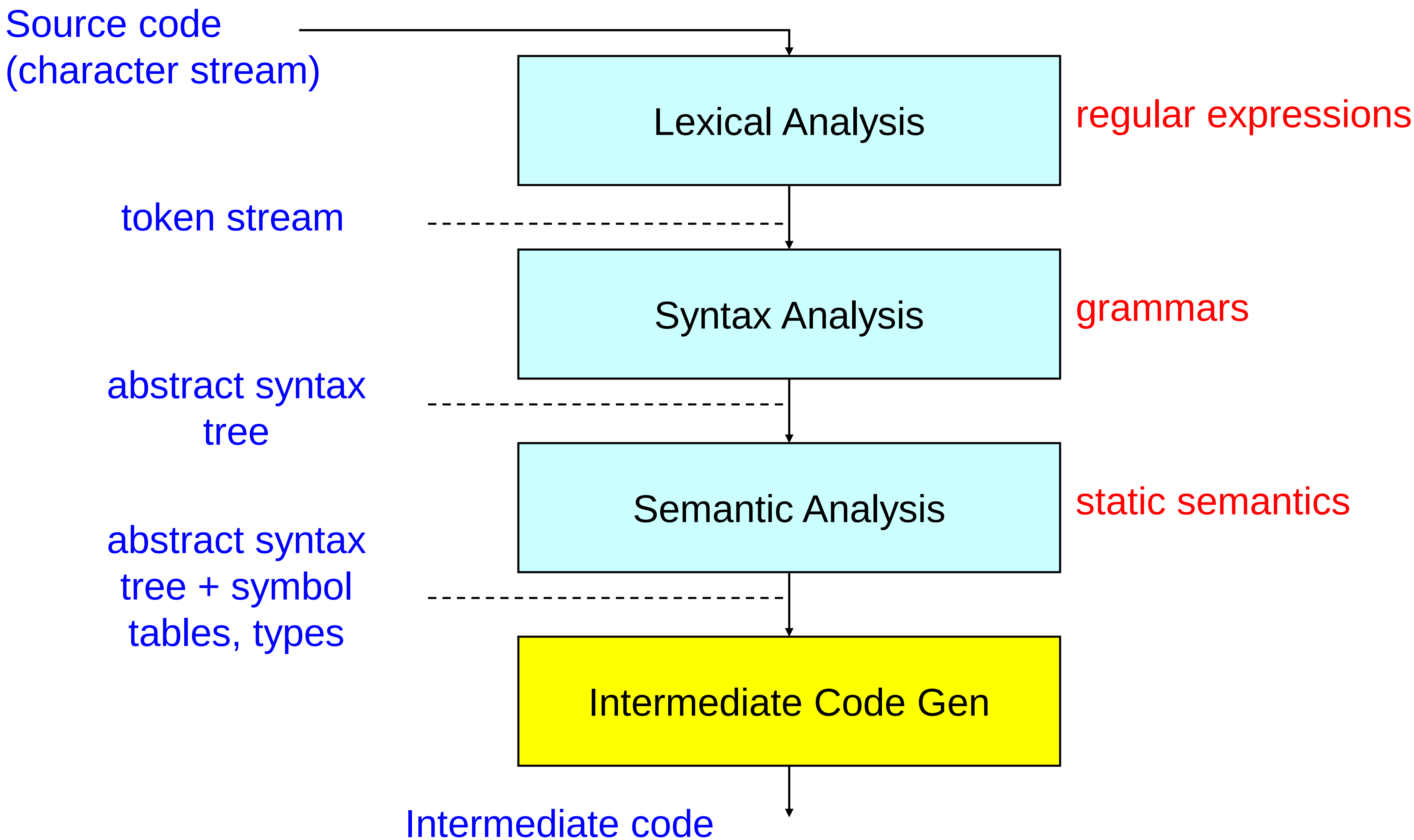
Review of Semantic Analysis

- **Check errors not detected by lexical or syntax analysis**
- **Scope errors**
  - Variables not defined
  - Multiple declarations
- **Type errors**
  - Assignment of values of different types
  - Invocation of functions with different number of parameters or parameters of incorrect type
  - Incorrect use of return statements

Other Forms of Semantic Analysis

- **One more category that we have not discussed**
- **Control flow errors**
  - Must verify that a break or continue statements are always encosed by a while (or for) stmt
  - Java: must verify that a break X statement is enclosed by a for loop with label X
  - Goto labels exist in the proper function
  - Can easily check control-flow errors by recursively traversing the AST

Where We Are...


Source code
(character stream)
Lexical Analysis
regular expressions
token stream
Syntax Analysis
grammars
abstract syntax tree
Semantic Analysis
static semantics
abstract syntax tree + symbol tables, types
Intermediate Code Gen
Intermediate code