Semantic Analysis 1

Symbol Tables and Attribute Grammars

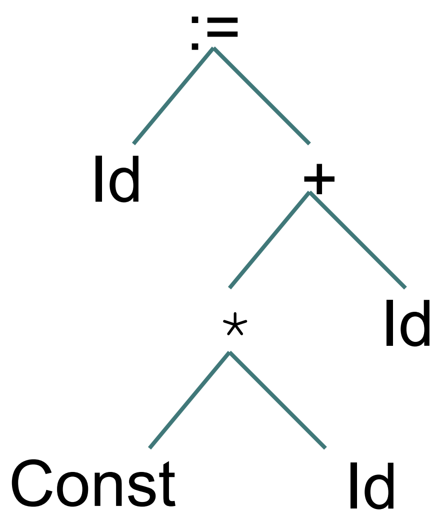
Dr. William Harrison
harrisonwl@missouri.edu
CS 4430 Compilers I

Today: new phase

- Semantic Analysis means
  - analyses (& transformations) based on the meaning of the particular source language your compiling
    - i.e., on its “semantics”
- Some of the new concepts we’ll encounter are:
  - attribute grammars
  - symbol tables
  - intermediate representations
  - syntax directed compilation
    - one pass
    - multiple passes

Abstract Syntax Trees (ASTs)

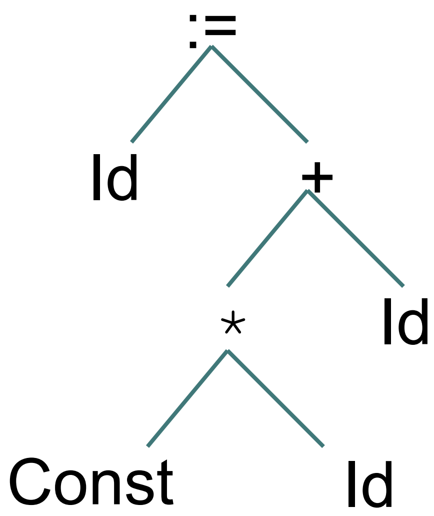
For "Y := 3 * X + I"

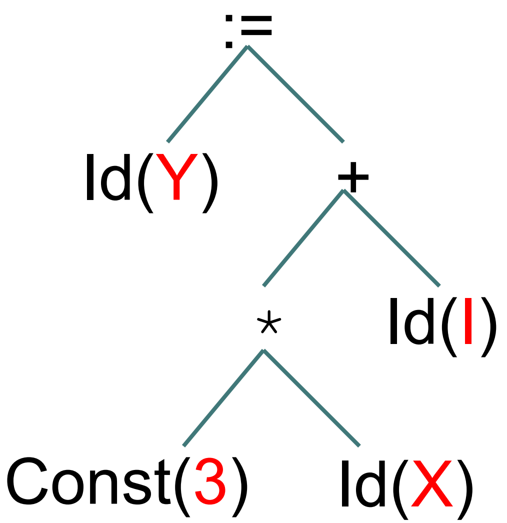


** such a tree could be produced by a compiler's front end*

ASTs with "attributes"

Attribute grammars are CFGs with extra information (a.k.a., "attributes") stored at the nodes





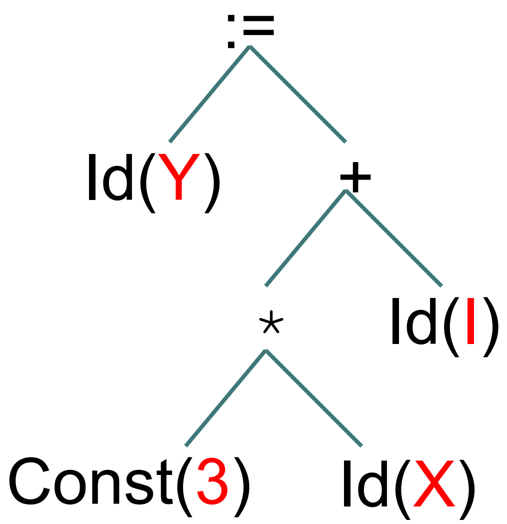
** red data are "initial attributes" in the lingo.*

Static vs. Dynamic program properties

- **Static** properties
  - any property that may be determined through analysis of <u>program text</u>
    - e.g., for some languages, the type of a program may be determined entirely through analysis of program source
      - e.g., ML, Java, & Pascal have “static type inference”
- **Dynamic** properties
  - any property that may only be discovered through execution of the program
    - e.g., “the final result of program p is 42” – can’t be discovered in general without some form of execution
- Compilation involves many forms of “static analysis”
  - e.g., type checking, the definition and use of variables, information of data and control flow, …

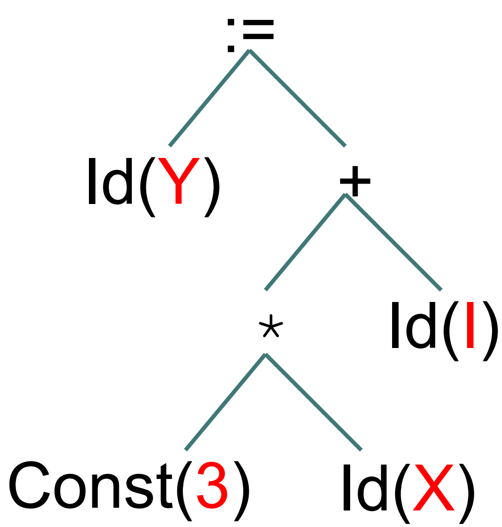
Attribute grammars and static checking

Assume: we know Y, I, and X are variables of type float
Question: is the following a legal program?



Attribute grammars and static checking

Assume: we know Y, I, and X are variables of type float
Question: is the following a legal program?

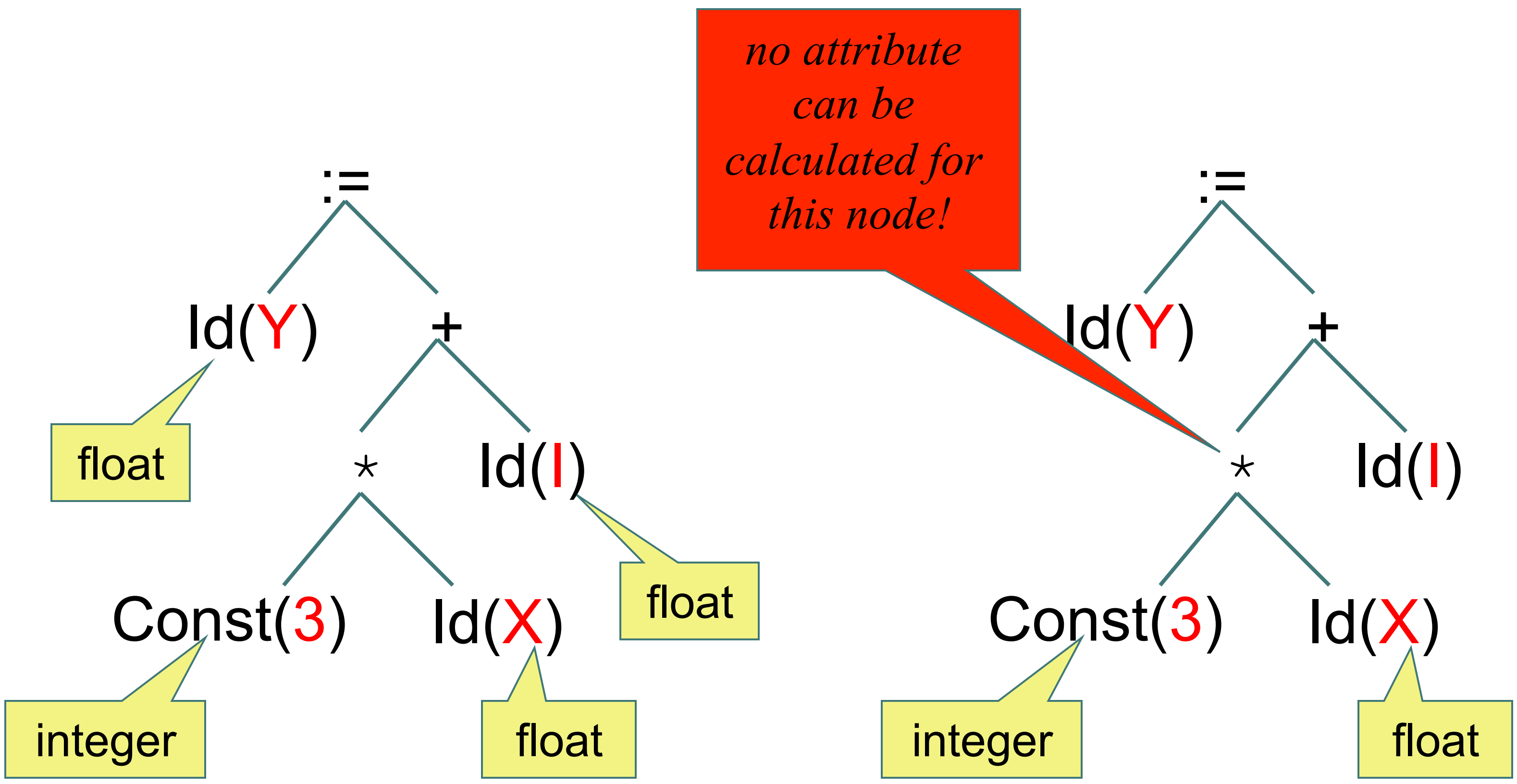


Answer: it depends on the language definition

- ML, Java, etc: no implicit coercion
- C, Basic, Scheme would allow

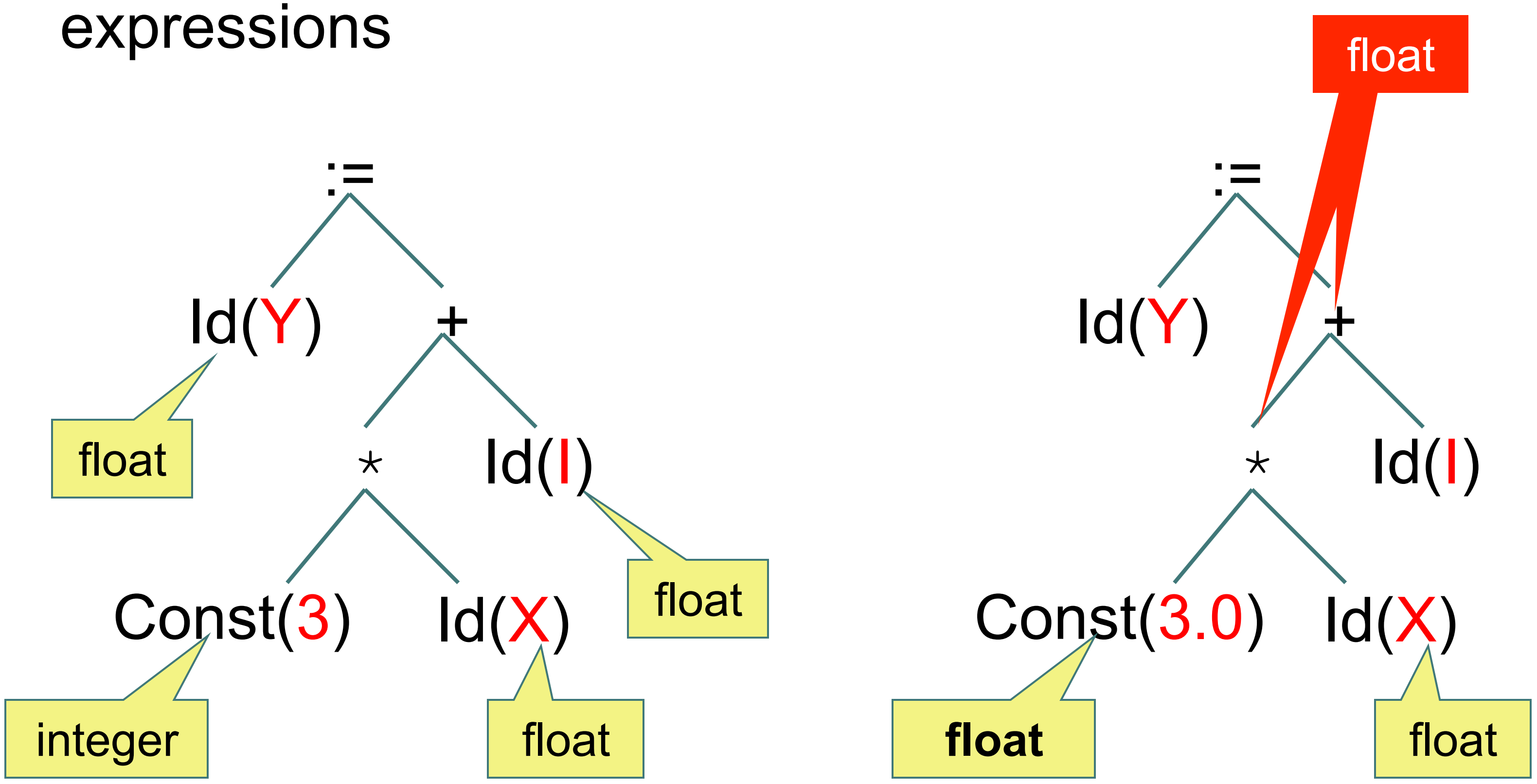
Attribute grammars and static checking

first case: it's illegal

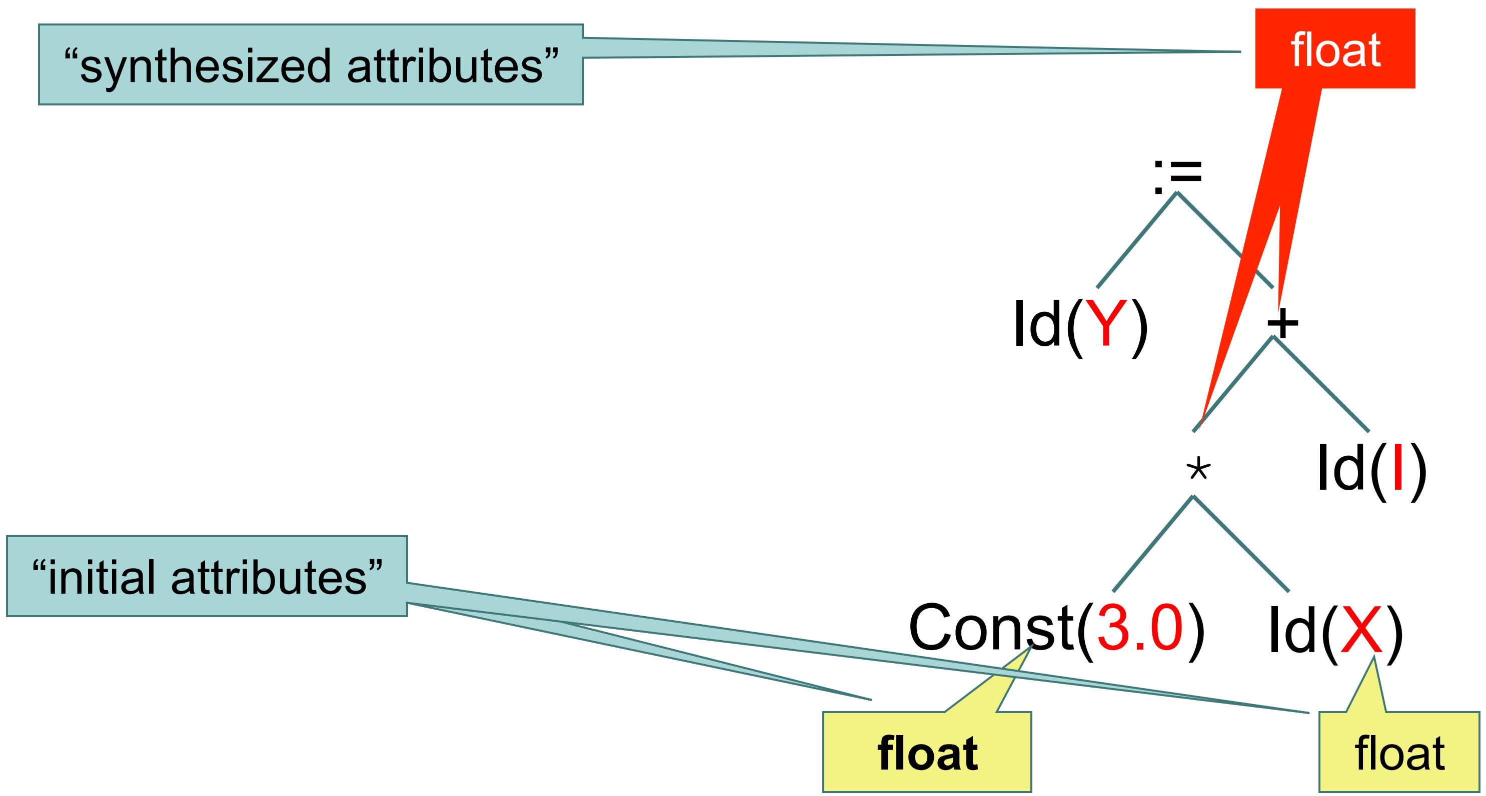


Attribute grammars and static checking

second case: implicitly coerce the constant so that it makes sense; calculate the types of the intermediate expressions



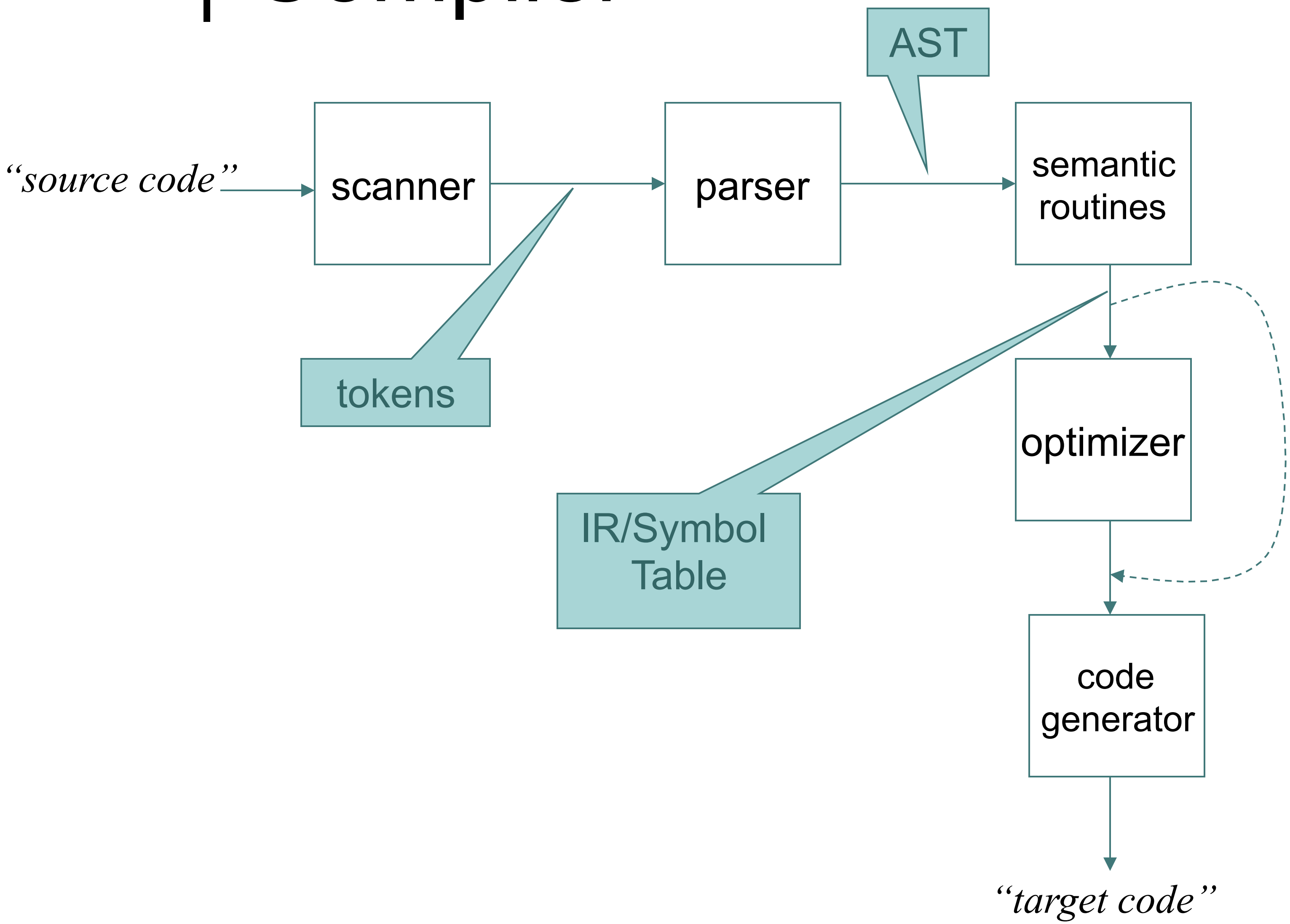
Attribute grammars and static checking



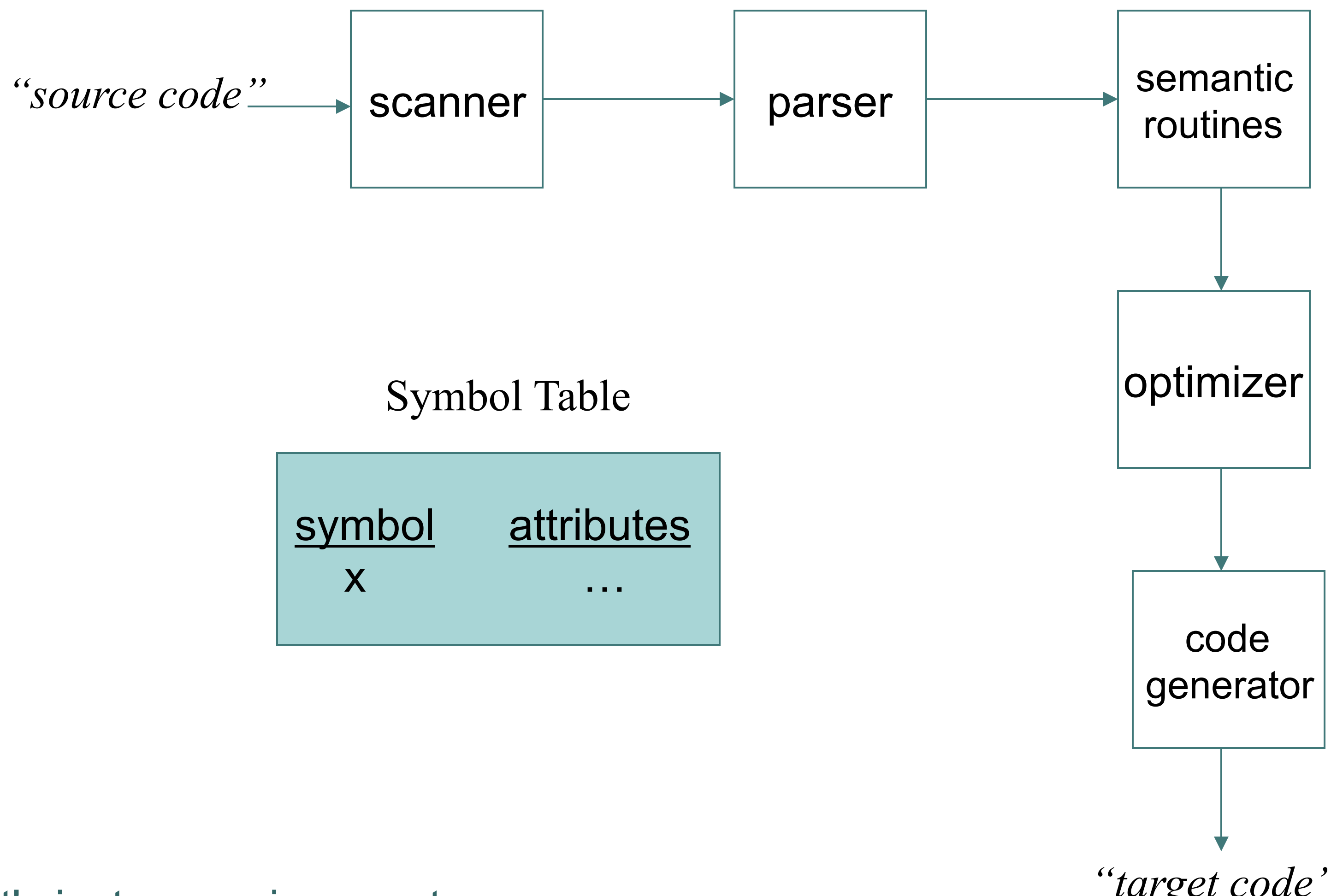
Syntax-directed Compilation

- All modern compilers are **syntax-directed**
  - meaning that, based on a representation of source code, they:
    - perform analyses
      - allowing for desirable performance characteristics
        - e.g., speed, code size
    - generate target code

Structure of Syntax-directed Compiler

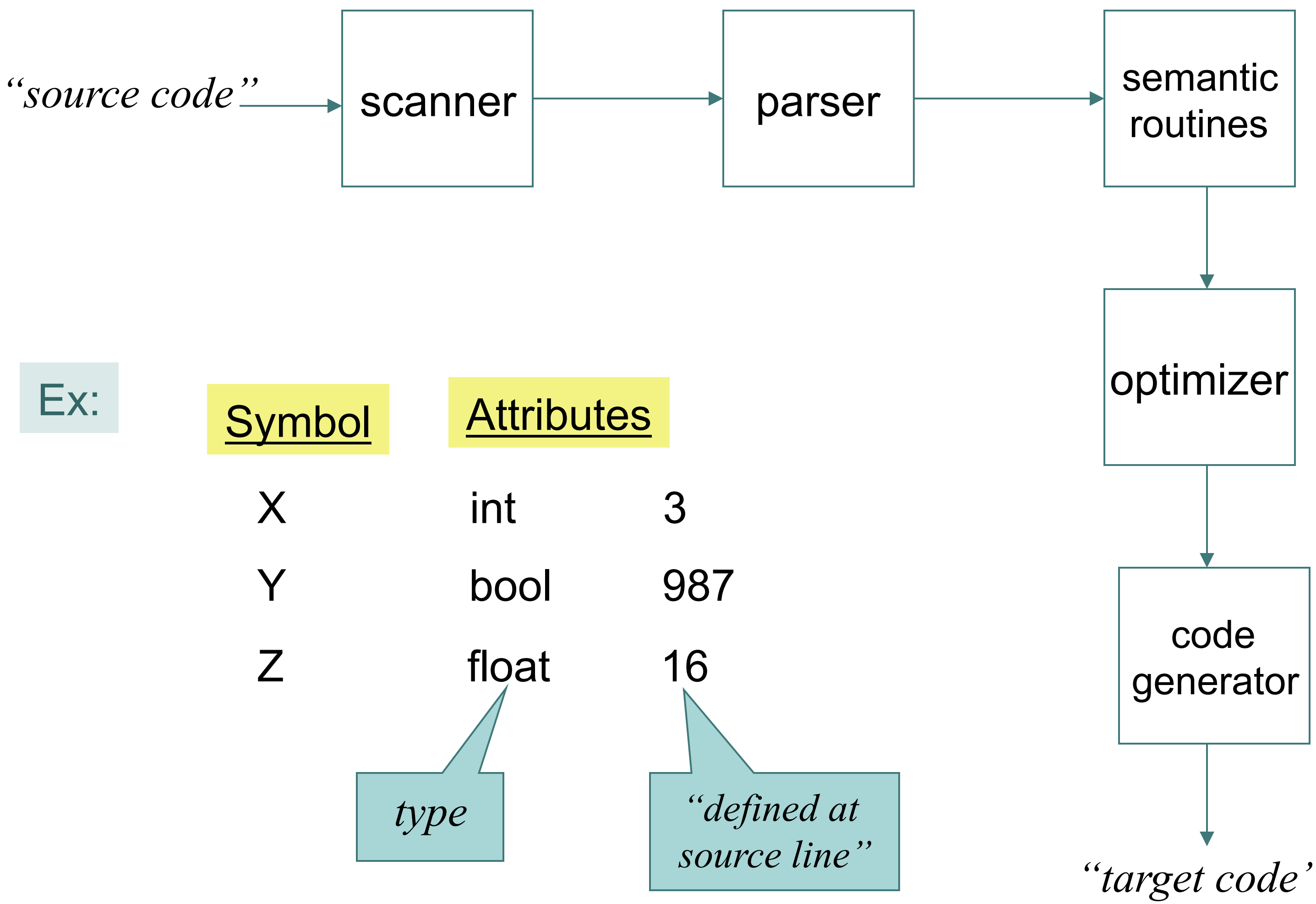


Symbol Table



It's just an environment.

Symbol Table: attributes are many and varied



Ex:

<u>Symbol</u>	<u>Attributes</u>	
X	int	3
Y	bool	987
Z	float	16

type

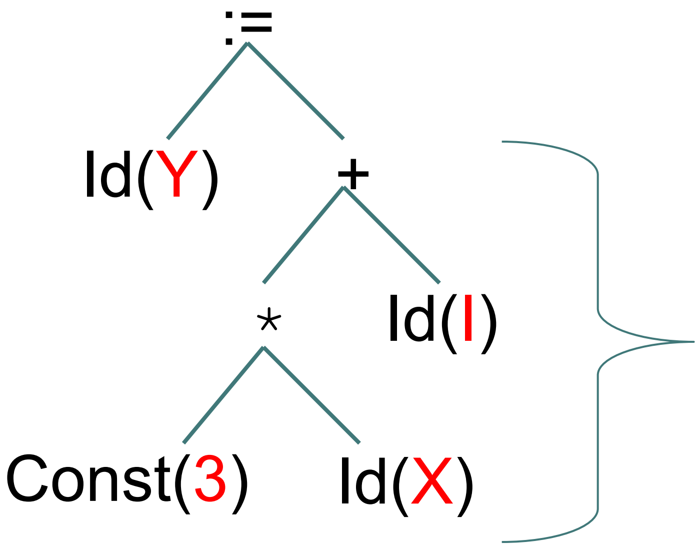
"defined at source line"

Intermediate Representations

- A.k.a., “IR” or “Intermediate Code”
- Varieties of IR
  - abstract syntax trees
    - written in a particular style to resemble target code
  - three-address code
    - a.k.a. register transfer language (RTL)
  - “Enriched” forms: IR annotated with useful information
    - def-use, use-def chains: connects definition and use of variables
    - Single Static Assignment form (SSA)
- Many compilers use multiple forms
  - Ex: GCC uses two – ASTs and RTL.

ASTs as IR

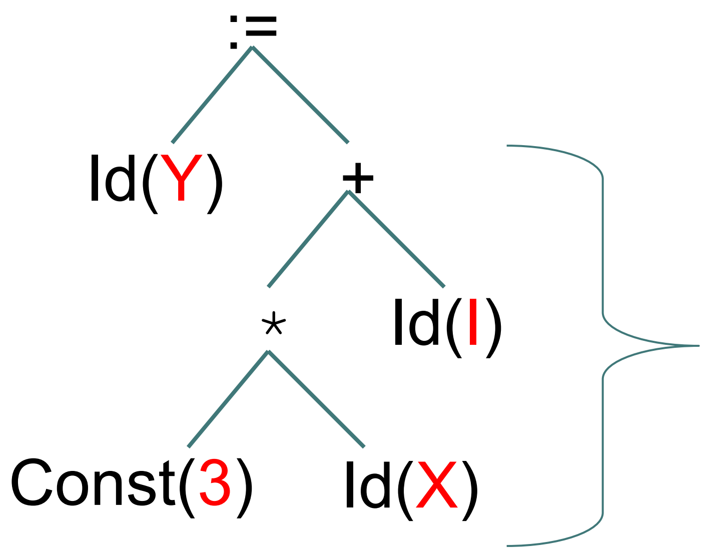
Idea: just continue using tree form produced by parser



why? machine languages don't have complicated expressions

ASTs as IR

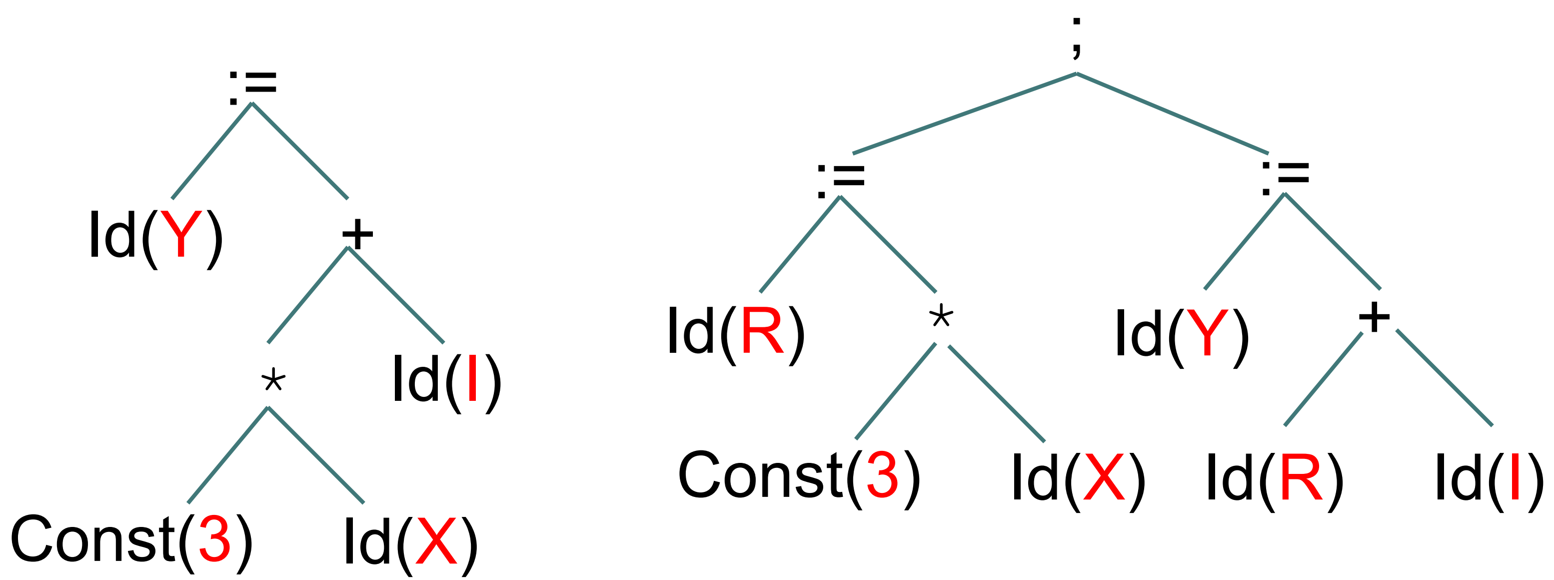
Problem: may not be in “machine language form”



why? machine languages don’t have complicated expressions “(3 * X) + Y”

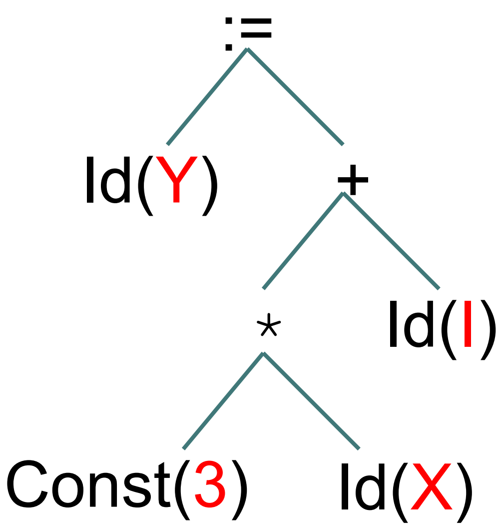
ASTs as IR

Problem: may not be in “machine language form”
Idea: rewrite as several assignments in sequence



* *May involve introduction of new temporaries like* R

three-address code/RTL



3-address/RTL representation

$R \leftarrow 3 * X;$
$Y \leftarrow R + X;$

Advantage: RTL enforces simplicity of expressions
Disadvantage: not as flexible

DU, UD chains

- DU chain = “definition use” chain
  - directed arc(s) from each variable definition to the use(s) of that variable
- UD chain = “use definition”
  - directed arc(s) from a variable use to the instruction defining that variable
- Both are implemented as graphs

Example: DU, UD chains

def-use

x ← …

…

z ← x + y

…

z’ ← x

no def’s of x

use-def

x ← …

…

z ← x + y

…

z’ ← x

conditional statements may mean more than one “def” per “use”

Static Single-Assignment (SSA)

Invariant on IR

- Every virtual register has **one** (static) definition site
- **Never** re-assign a virtual register.

This is straightforward for straight-line code.

```
a ← x * y
b ← a - 1
a ← y * b
b ← x * 4
a ← a + b
```

$$
\begin{aligned}
a_1 &\leftarrow x * y \\
b_1 &\leftarrow a_1 - 1 \\
a_2 &\leftarrow y * b_1 \\
b_2 &\leftarrow x * 4 \\
a_3 &\leftarrow a_2 + b_2
\end{aligned}
$$

Next Class

- Continue “Semantic Processing”