Course 142A Compilers & Interpreters

Syntactic Analysis Continued

Lecture Week 3

Prof. Dr. Luc Bläser

Last Lecture - Quiz

```
Expression    = Expression [ ( "+" | "-" ) Term ].
Term          = Number | "(" Expression ")".
```

Can we parse this grammar with a top down parser?

Left Recursion

- Top down parser is unable to parse

```
Expression    = Expression [ ( "+" | "-" ) Term ].
Term          = Number | "(" Expression ")".
```

rewrite

```
Expression    = Term { ( "+" | "-" ) Term }.
Term          = Number | "(" Expression ")".
```

- But bottom-up parser can deal with it

Today’s Topics

- Bottom-Up Parser

Learning Goals

- Understand how a bottom-up parser works
- Know how to generate the LR parsing table

Top-Down Parser (LL)

Input: 1 + (2 - 3)

Derivation:

Expression
Term + Term
1 + Term
1 + (Expression)
1 + (Term - Term)
1 + (2 - Term)
1 + (2 - 3)

top-down

left-most expansion

Bottom-Up Parser (LR)

Input: 1 + (2 - 3)

Derivation:

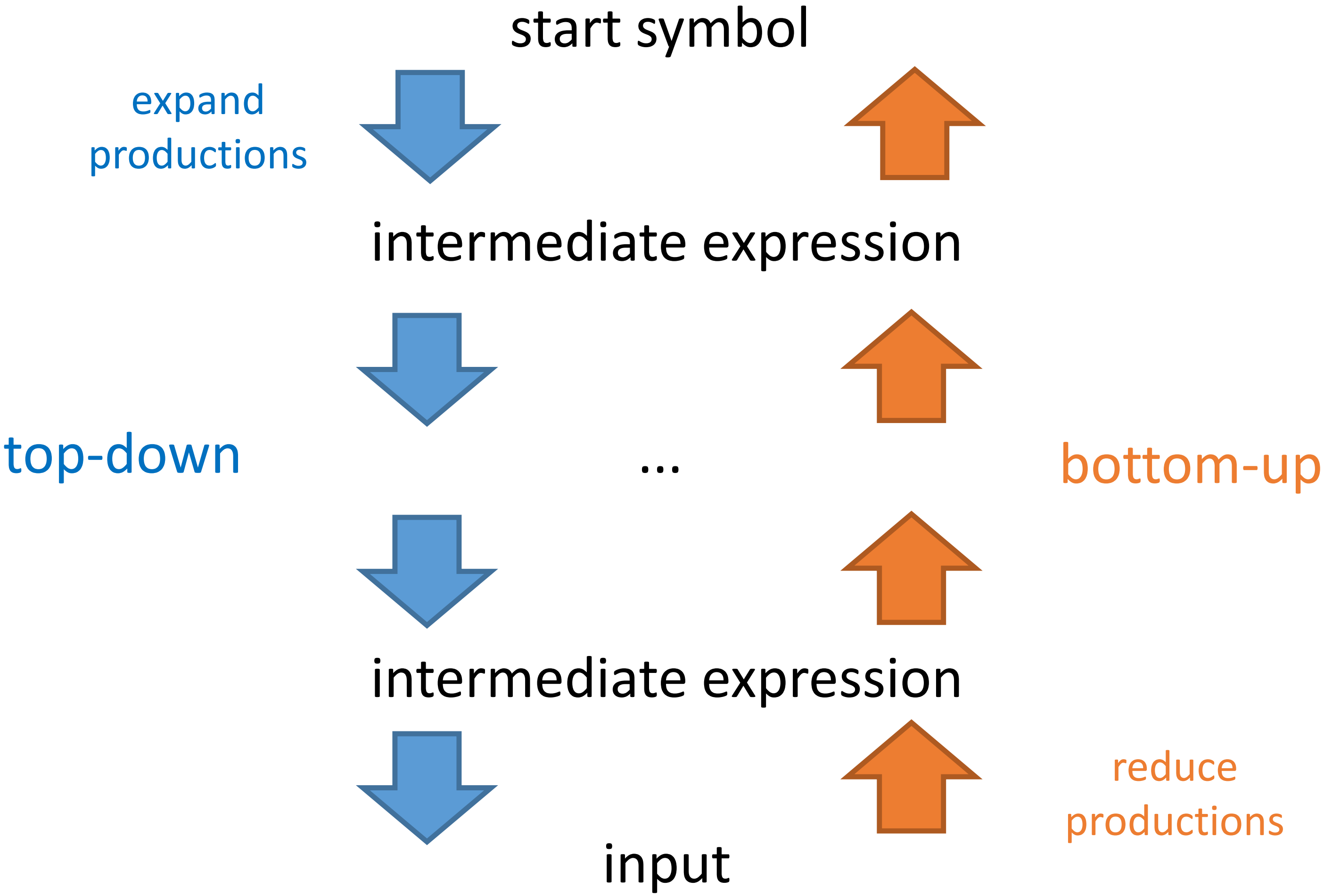
Expression
Term + Term
Term + (Expression)
Term + (Term - Term)
Term + (Term - 3)
Term + (2 - 3)
1 + (2 - 3)

bottom-up

right-most reduction

Our focus

Top-Down vs. Bottom-Up



start symbol

expand productions

intermediate expression

top-down

…

bottom-up

intermediate expression

reduce productions

input

Bottom-Up Approach

- Read symbol in text without fix goal
- Check after each step, whether read sequence corresponds to a production
  - If yes => reduce to syntax construct (REDUCE)
  - If no => read next symbol in input (SHIFT)
- The start symbol remains at the end
  - Otherwise syntax error

Example Run-Through

Step	Detected constructs	Remaining input
		1 + (2 - 3)
SHIFT	1	+ (2 - 3)
REDUCE	Term	+ (2 - 3)
SHIFT	Term +	(2 - 3)
SHIFT	Term + (	2 - 3)
SHIFT	Term + (2	- 3)
REDUCE	Term + (Term	- 3)
SHIFT	Term + (Term -	- 3)
SHIFT	Term + (Term - 3	)
REDUCE	Term + (Term - Term	)
REDUCE	Term + (Expression	)
SHIFT	Term + (Expression)	
REDUCE	Term + Term	
REDUCE	Expression	

Simplified Parsing Table

Detected construct	Rule
... Number	REDUCE Term
... Term + Term	REDUCE Expression
... Term - Term	REDUCE Expression
... "(" Expression ")"	REDUCE Term
Otherwise	SHIFT

Suffix of detected constructs is decisive (stack principle)

Complete Parsing Table

S: SHIFT
R: REDUCE
A: ACCEPT
otherwise ERROR

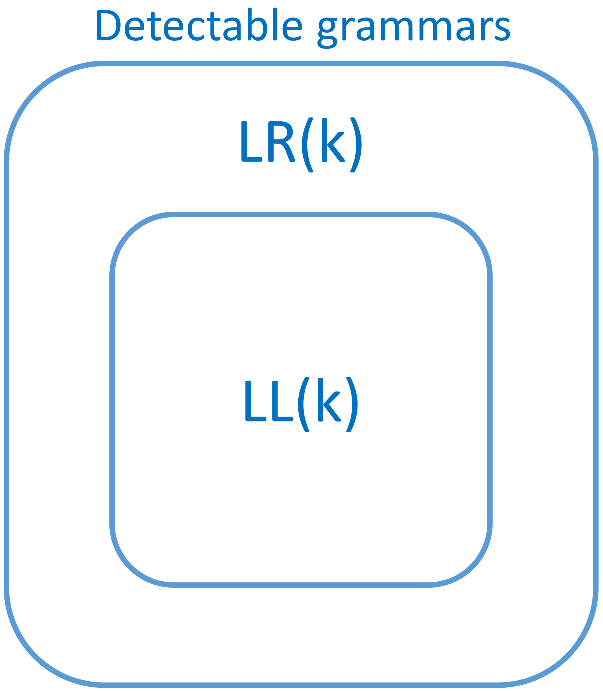
	N	+	-	(	)	$
I_0	S: I_3			S: I_4		
I_1		S: I_5	S: I_6			A
I_2		R: E = T	R: E = T		R: E = T	R: E = T
I_3		R: T = N	R: T = N		R: T = N	R: T = N
I_4	S: I_3			S: I_4		
I_5	S: I_3			S: I_4		
I_6	S: I_3			S: I_4		
I_7		S: I_5	S: I_6		S: I_{10}	
I_8		R: E =E+T	R: E =E+T		R: E =E+T	R: E =E+T
I_9		R: E = E-T	R: E = E-T		R: E = E-T	R: E = E-T
I_{10}		R: T = (E)	R: T = (E)		R: T = (E)	R: T = (E)

Parsing Table

- Construction is complicated
  - LR-parser generator
- Decision conflicts are possible
  - SHIFT-REDUCE conflicts
  - REDUCE-REDUCE conflicts
  - Resolution by programmer
  - Or modification of grammar
  - Or larger lookaheads

LR-Parser

- More powerful than LL-parser
  - E.g. can deal with left recursion


Detectable grammars

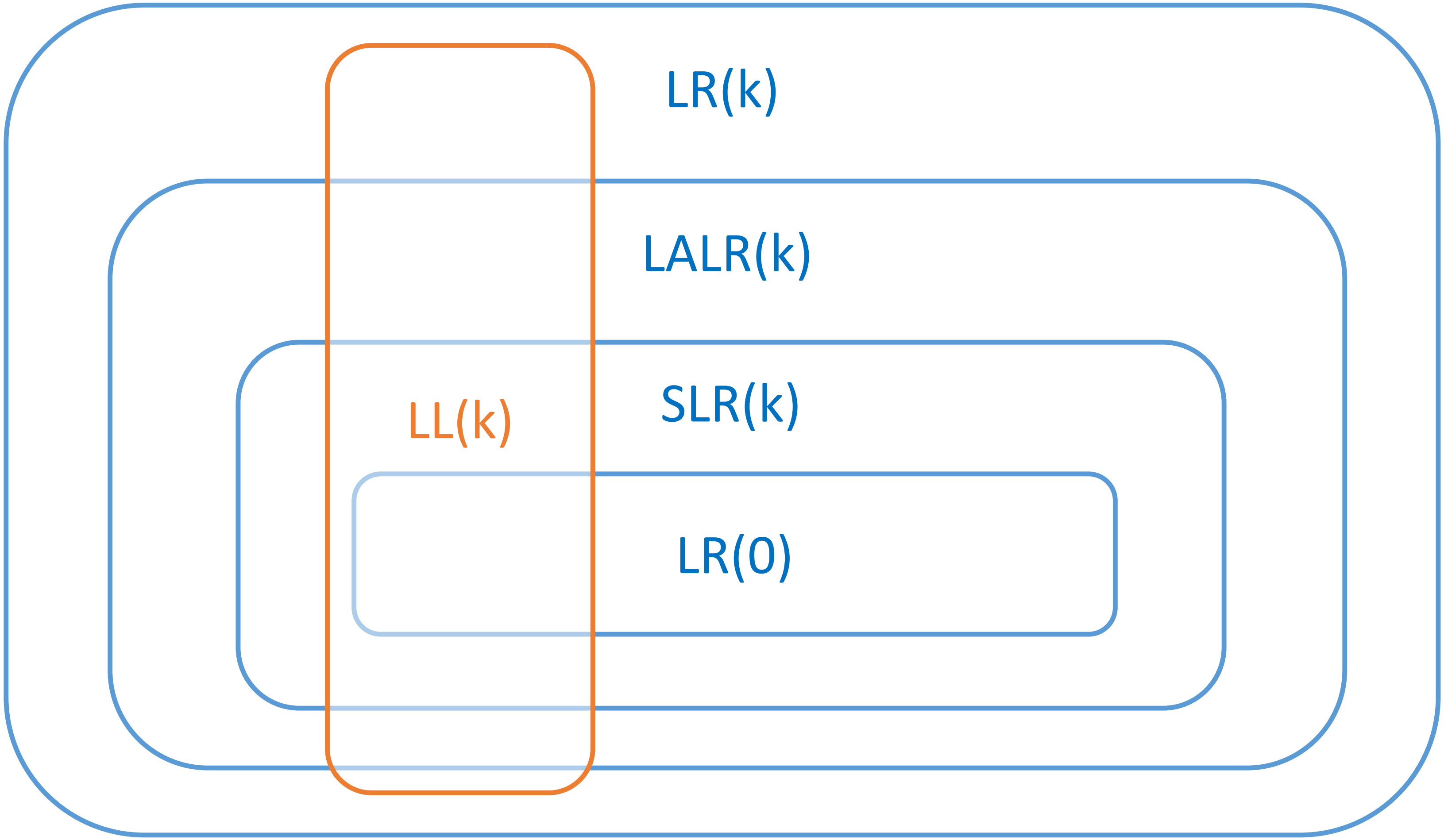
LR(k)

LL(k)

LR-Parser Types

- LR(0)
  - Computing parsing table without lookahead
  - State is sufficient to decide
- SLR(k) (Simple LR)
  - Lookahead on REDUCE to resolve certain conflicts
  - No additional states
- LALR(k) (Look-Ahead LR)
  - Analyzes grammar for LR(0) conflicts
  - Uses lookaheads at conflict places with new states
- LR(k)
  - A state per grammar step + lookahead
  - Unpractical, too many states

Powerfulness



LR-Parser Details

- 4 possible steps
  - SHIFT
  - REDUCE
  - ACCEPT
  - ERROR
- Parser ingredients
  - Parsing table (SHIFT, REDUCE etc.)
  - State machine
  - Derivation stack (detected symbols & latest states)
  - Lookahead (remaining input symbols)

LR Parser Construction

1. Adjust grammar
   - Augmented grammar
2. Compute state machine
   - Item, Handle, Closure, Goto
3. Construct parsing table
   - FOLLOW-Set

Adjust Grammar (1)

```
Expression = Term { ("+" | "-") Term }.
Term = Number | "(" Expression ")".
```

Introduce dedicated start symbol
(augmented grammar)

```
Start = Expression.
Expression = Term { ("+" | "-") Term }.
Term = Number | "(" Expression ")".
```

Replace EBNF repetitions
by recursion

```
Start = Expression.
Expression = Term  | Expression ("+" | "-") Term.
Term = Number | "(" Expression ")".
```

Adjust Grammar (2)

```
Start = Expression.
Expression = Term  | Expression ("+" | "-") Term.
Term = Number | "(" Expression ")".
```

Structure EBNF-alternatives and options into multiple productions

```
Start = Expression.
Expression = Term.
Expression = Expression ("+" | "-") Term.
Term = Number.
Term = "(" Expression ")".
```

Item

- Item = Production with point ● at right hand side
  - Point denotes how far the parser has analyzed

Example: `Expression = Expression "+" Term.`

Possible items:

```
[ Expression = ● Expression "+" Term ]
[ Expression = Expression ● "+" Term ]
[ Expression = Expression "+" ● Term ]
[ Expression = Expression "+" Term ● ]
```

Item with ● at end is called handle =>
here we can reduce the production

Closure

- Transitive closure over sets of items

closure { [A = α ● B β.] } includes [B = ● γ.] to the set, if ● precedes non-terminal symbol B.
Repeatedly perform this for all items in set.
(α, β, γ are terminal or non-terminal symbols.).

Example:

closure { [Start = ● Expression] } =
{ [Start = ● Expression] ,
[Expression = ● Term],
[Expression = ● Expression "+" Term],
[Expression = ● Expression "-" Term],
[Term = ● Number],
[Term = ● "(" Expression ")"] }.

Goto

- Goto for item set I and symbol X
  - X is a terminal or non-terminal symbol

Goto(I, X) = closure of all items [$A = \alpha X \bullet \beta$], if [$A = \alpha \bullet X \beta$] is part of I.

Parser proceeds with symbol X

- Serves to compute state machine

Compute Gotos (1)

Start state
I_0 = { [Start = ● Expression],
[Expression = ● Term],
[Expression = ● Expression "+" Term],
[Expression = ● Expression "-" Term],
[Term = ● Number],
[Term = ● "(" Expression ")"] }

Goto(I_0, Expression) =
{ [Start = Expression ●],
[Expression = Expression ● "+" Term],
[Expression = Expression ● "-" Term] } =: I_1

Goto(I_0, Term) =
{ [Expression = Term ●] } =: I_2

Goto(I_0, Number) =
{ [Term = Number ●] } =: I_3

Compute All Gotos (2)

Goto(I_0, "(") =
{ [Term = "(" ● Expression ")"],
[Expression = ● Term],
[Expression = ● Expression "+" Term],
[Expression = ● Expression "-" Term],
[Term = ● Number],
[Term = ● "(" Expression ")"] } =: I_4

Goto(I_1, "+") =
{ [Expression = Expression "+" ● Term],
[Term = ● Number],
[Term = ● "(" Expression ")"] } =: I_5

Goto(I_1, "-") =
{[Expression = Expression "-" ● Term],
[Term = ● Number],
[Term = ● "(" Expression ")"] } =: I_6

Compute All Gotos (3)

Goto(I_4, Expression) =
{ [Term = "(" Expression ● ")"],
[Expression = Expression ● "+" Term],
[Expression = Expression ● "-" Term] } =: I_7

Goto(I_4, Term) = { [Expression = Term ●] } = I_2
Goto(I_4, Number) = { [Term = Number ●] } = I_3
Goto(I_4, "(") = I_4

Goto(I_5, Term) =
{ [Expression = Expression "+" Term ●] } =: I_8
Goto(I_5, Number) = I_3
Goto(I_5, "(") = I_4

Goto(I_6, Term) =
{ [Expression = Expression "-" Term ●] } =: I_9
Goto(I_6, Number) = I_3
Goto(I_6, "(") = I_4

Compute All Gotos (4)

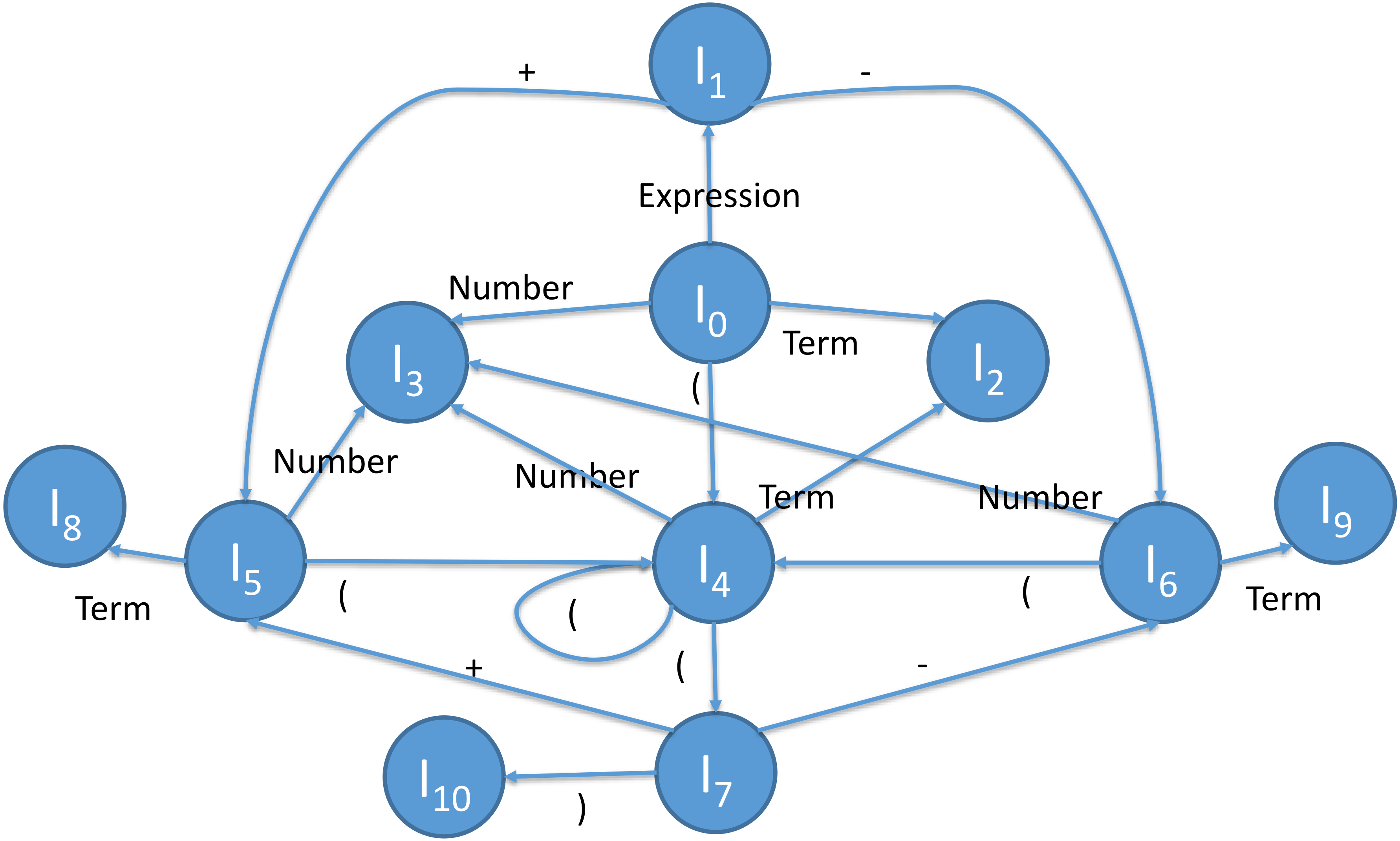
Goto(I_7, ")") =

{ [Term = "(" Expression ")" ●] } =: I_{10}

Goto(I_7, "+") = I_5

Goto(I_7, "-") = I_6

State Machine



FOLLOW-Set

- FOLLOW(X) = All terminal symbols that can follow after non-terminal symbol X.

```
FOLLOW(Expression) = { "+", "-", ")", $ }
FOLLOW(Term)       = { "+", "-", ")", $ }
```

$ denotes end of input

Construct Parsing Table

- If [Start = X ●] in I
  (X is original start symbol)
  => ACTION(I, $): ACCEPT
- If [A = α ●] in I
  => ACTION(I, a): REDUCE for each a in FOLLOW(A)
- If [A = α ● a β] in I, Goto(I, a) = J
  => ACTION(I, a): SHIFT, go to J

S: SHIFT
R: REDUCE
A: ACCEPT
otherwise ERROR

Parsing Table

E: Expression
T: Term
N: Number

	N	**+**	**-**	**(**	**)**	**$**
I_0	S: I_3			S: I_4		
I_1		S: I_5	S: I_6			A
I_2		R: E = T	R: E = T		R: E = T	R: E = T
I_3		R: T = N	R: T = N		R: T = N	R: T = N
I_4	S: I_3			S: I_4		
I_5	S: I_3			S: I_4		
I_6	S: I_3			S: I_4		
I_7		S: I_5	S: I_6		S: I_{10}	
I_8		R: E =E+T	R: E =E+T		R: E =E+T	R: E =E+T
I_9		R: E = E-T	R: E = E-T		R: E = E-T	R: E = E-T
I_{10}		R: T = (E)	R: T = (E)		R: T = (E)	R: T = (E)

Parser Stack

- Stack of currently detected symbols including state

Stack	Input
$\underbrace{\$\ I_0}_{\text{Initial stack empty with state } I_0}$	1 + 2
	⇓ SHIFT I_3
$\$\ I_0\ \underbrace{1\ I_3}_{\text{Top of stack symbol 1 with state } I_3}$	+ 2

Parsing Operations

- SHIFT symbol a in state I
  - push (a, Goto(I, a))
- REDUCE X = ... with n symbols on right hand side
  - n times pop()
  - Look at current state I on stack
  - push (X, Goto(I, X))

$\$\ I_0\ 1\ I_3$ REDUCE Term = Number.

$\$\ I_0\ \mathrm{Term}$ Goto(I_0, Term) = I_2

$\$\ I_0\ \mathrm{Term}\ I_2$

Parsing 1 + (2 - 3)

Op	Stack	Input rest
	$\$\ I_0$	`1+(2-3)$`
S	$\$\ I_0\ 1\ I_3$	`+(2-3)$`
R	$\$\ I_0\ \mathtt{Term}\ I_2$	`+(2-3)$`
R	$\$\ I_0\ \mathtt{Expr}\ I_1$	`+(2-3)$`
S	$\$\ I_0\ \mathtt{Expr}\ I_1\ +\ I_5$	`(2-3)$`
S	$\$\ I_0\ \mathtt{Expr}\ I_1\ +\ I_5\ (\ I_4$	`2-3)$`
S	$\$\ I_0\ \mathtt{Expr}\ I_1\ +\ I_5\ (\ I_4\ 2\ I_3$	`-3)$`
R	$\$\ I_0\ \mathtt{Expr}\ I_1\ +\ I_5\ (\ I_4\ \mathtt{Term}\ I_2$	`-3)$`
R	$\$\ I_0\ \mathtt{Expr}\ I_1\ +\ I_5\ (\ I_4\ \mathtt{Expr}\ I_7$	`-3)$`
S	$\$\ I_0\ \mathtt{Expr}\ I_1\ +\ I_5\ (\ I_4\ \mathtt{Expr}\ I_7\ -\ I_6$	`3)$`
S	$\$\ I_0\ \mathtt{Expr}\ I_1\ +\ I_5\ (\ I_4\ \mathtt{Expr}\ I_7\ -\ I_6\ 3\ I_3$	`)$`
R	$\$\ I_0\ \mathtt{Expr}\ I_1\ +\ I_5\ (\ I_4\ \mathtt{Expr}\ I_7\ -\ I_6\ \mathtt{Term}\ I_9$	`)$`
R	$\$\ I_0\ \mathtt{Expr}\ I_1\ +\ I_5\ (\ I_4\ \mathtt{Expr}\ I_7$	`)$`
S	$\$\ I_0\ \mathtt{Expr}\ I_1\ +\ I_5\ (\ I_4\ \mathtt{Expr}\ I_7\ )\ I_{10}$	`$`
R	$\$\ I_0\ \mathtt{Expr}\ I_1\ +\ I_5\ \mathtt{Term}\ I_8$	`$`
R	$\$\ I_0\ \mathtt{Expr}\ I_1$	`$`

ACCEPT

Discussion

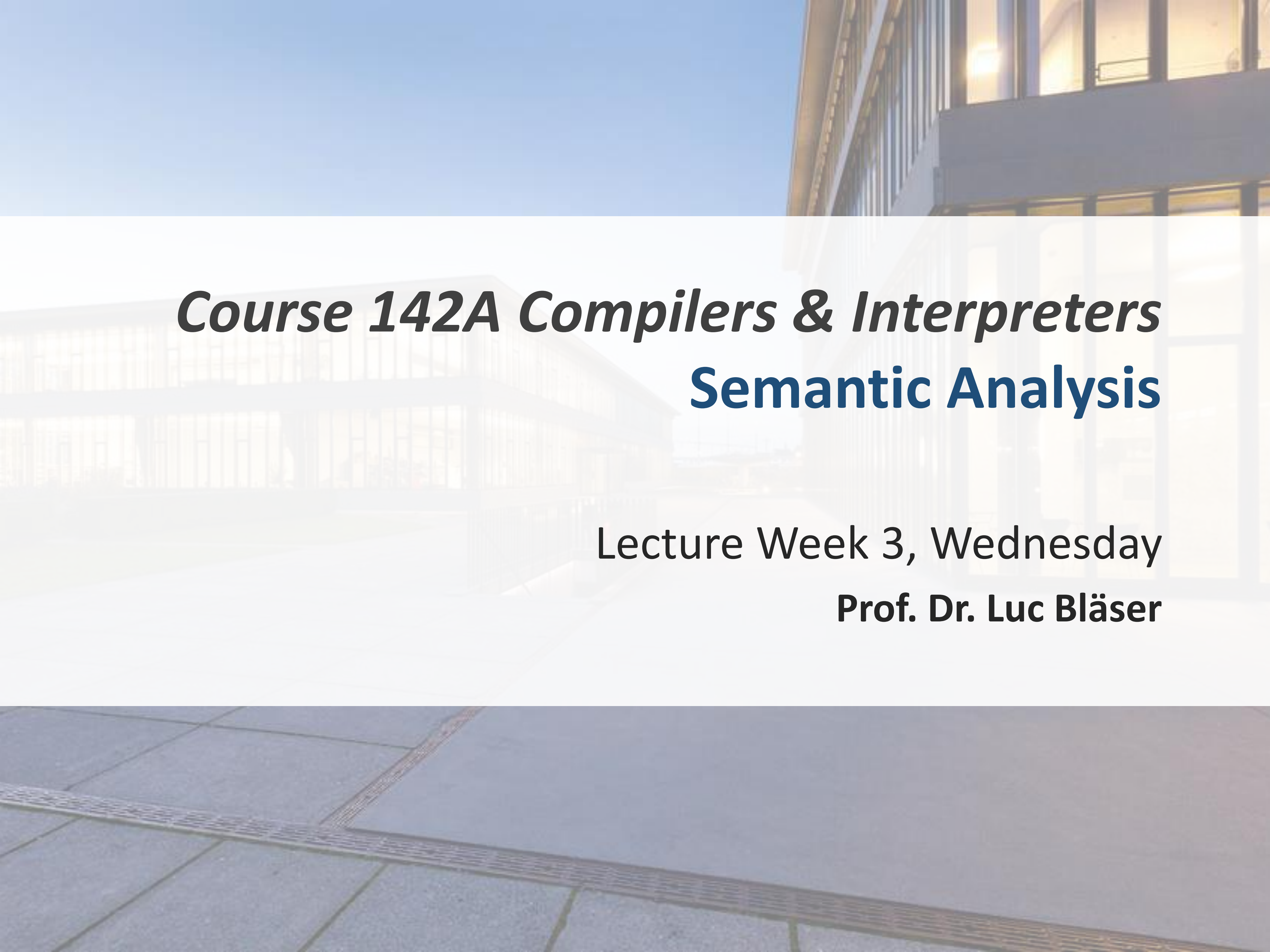
- LL(k) parser is often sufficient in practice
  - Grammar can usually be adjusted to it
- C++, Java and C# have hand-crafted LL-parser
  - Although grammar is not designed for LL
  - Need to rewrite grammar at some places
  - Or require larger lookahead
- LALR(k) is common in parser generators
  - yacc, bison
- But also LL(k) is regaining importance
  - AntLR, Coco/R

Review: Learning Goals

- ✓ Understand how a bottom-up parser works
- ✓ Know how to generate the LR parsing table

Further Reading

- Dragon Book, Chapter 4 (Syntax Analysis)
  - Sections 4.5 – 4.6 (Bottom-Up Parser, SLR)
- Optional, if interested
  - Sections 4.7 – 4.8 (LR and LALR)
  - Section 4.9 (Yacc Generator)



Course 142A Compilers & Interpreters

Semantic Analysis

Lecture Week 3, Wednesday

Prof. Dr. Luc Bläser

Last Lecture - Quiz

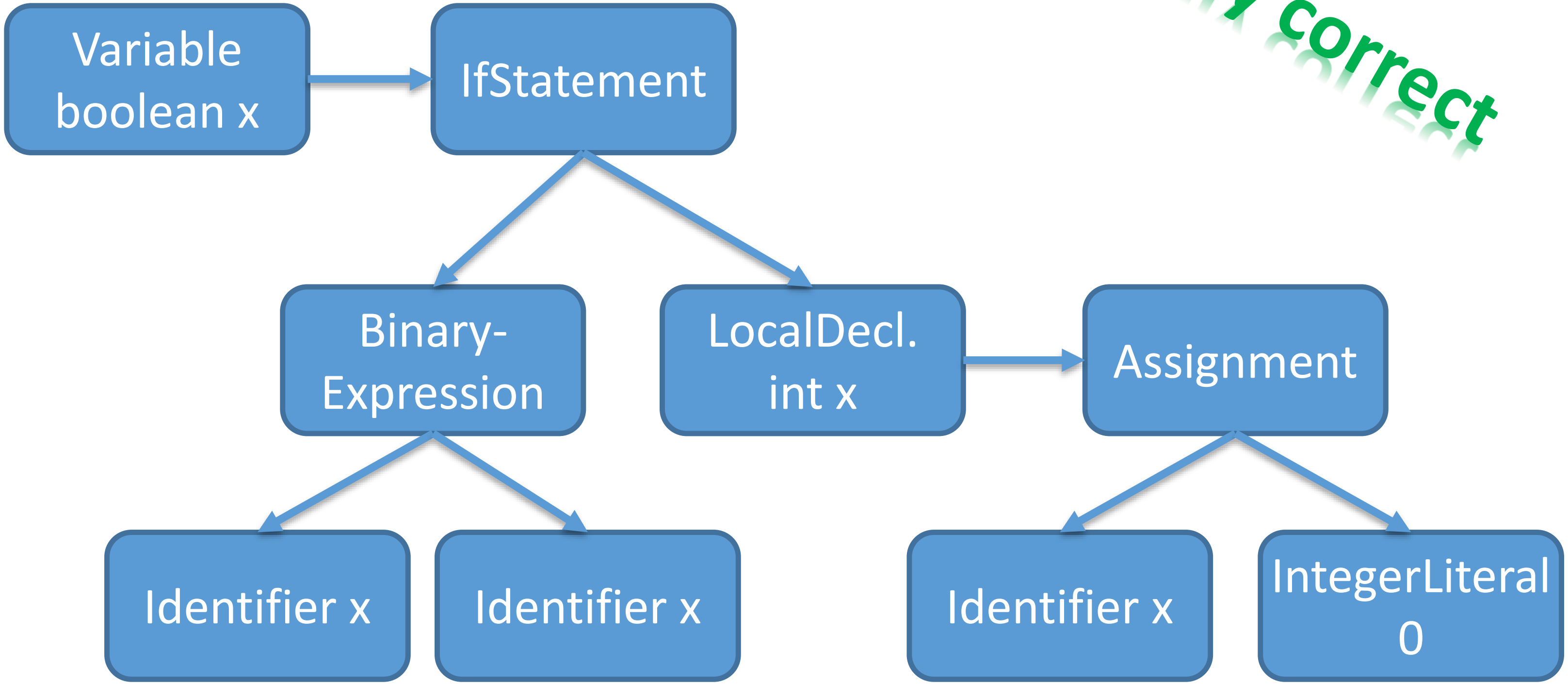
```
boolean x;
if (x + x) {
  int x;
  x = 0;
}
```

Is the program syntactically correct?
What does the parser return?

Syntactic Analysis

- Context-free grammar
- Parser returns syntax tree

Syntactically correct



Semantic Analysis

- Context-sensitive rules
- Types, declarations etc.

Semantically wrong

```
boolean x;
if (x + x) {
  int x;
  x = 0;
}
```

boolean cannot be added

Multiple declaration

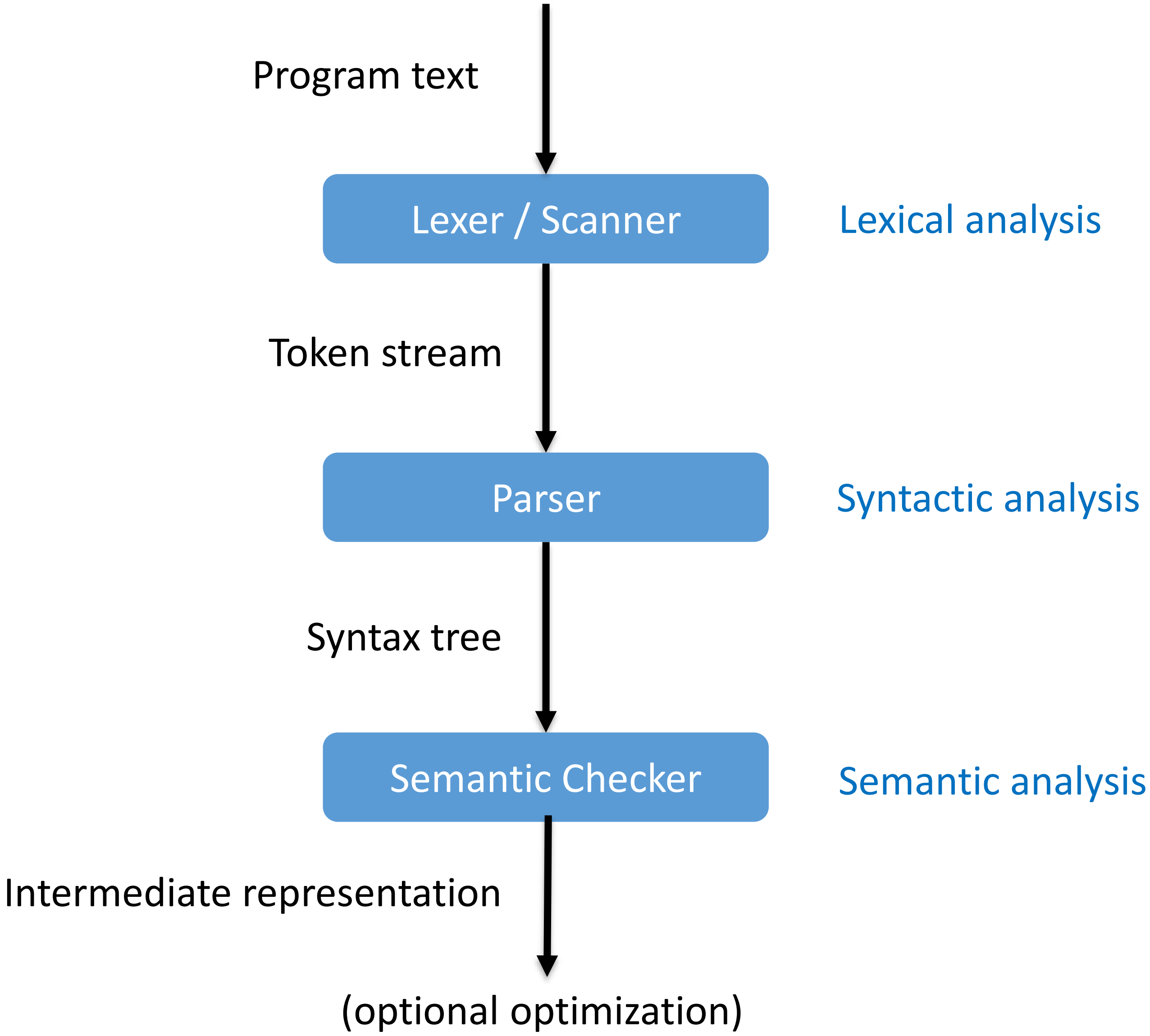
Today's Content

- Semantic checker
- Symbol table
- Name resolution
- Type checks

Learning Goals

- Understand the purpose and functionality of the semantic analysis
- Understand the design and construction of a symbol table
- Know how to implement type resolution and type checks

Compiler Frontend



Program text

Lexer / Scanner

Lexical analysis

Token stream

Parser

Syntactic analysis

Syntax tree

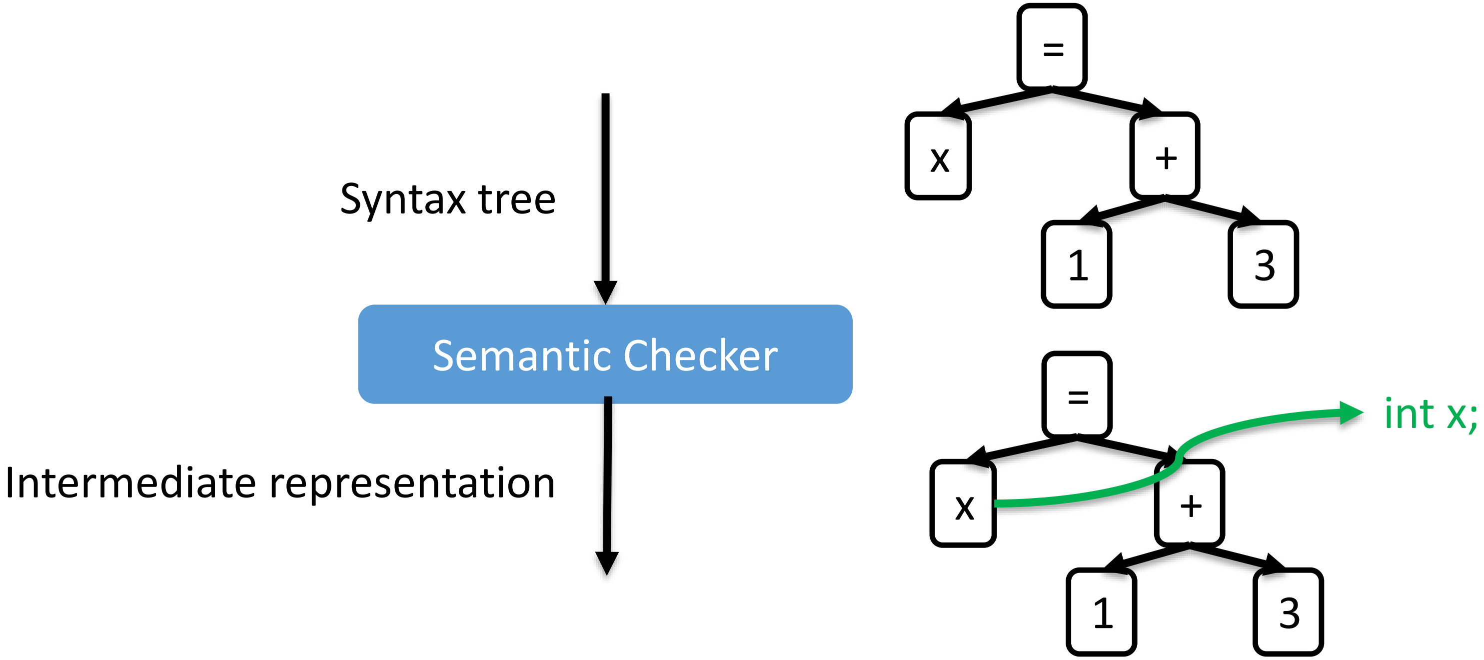
Semantic Checker

Semantic analysis

Intermediate representation

(optional optimization)

Our Focus: Semantic Checker



Syntax tree

Semantic Checker

Intermediate representation

Semantic Checker

- Cares about the semantic analysis
- Input: Syntax tree
    - Concrete or abstract
- Output: Intermediate representation
    - Abstract syntax tree + symbol table

Tasks of a Semantic Checker

- Check whether the program conforms with the semantic language rules
- Transform the program into a form that can be easily processed by code generation

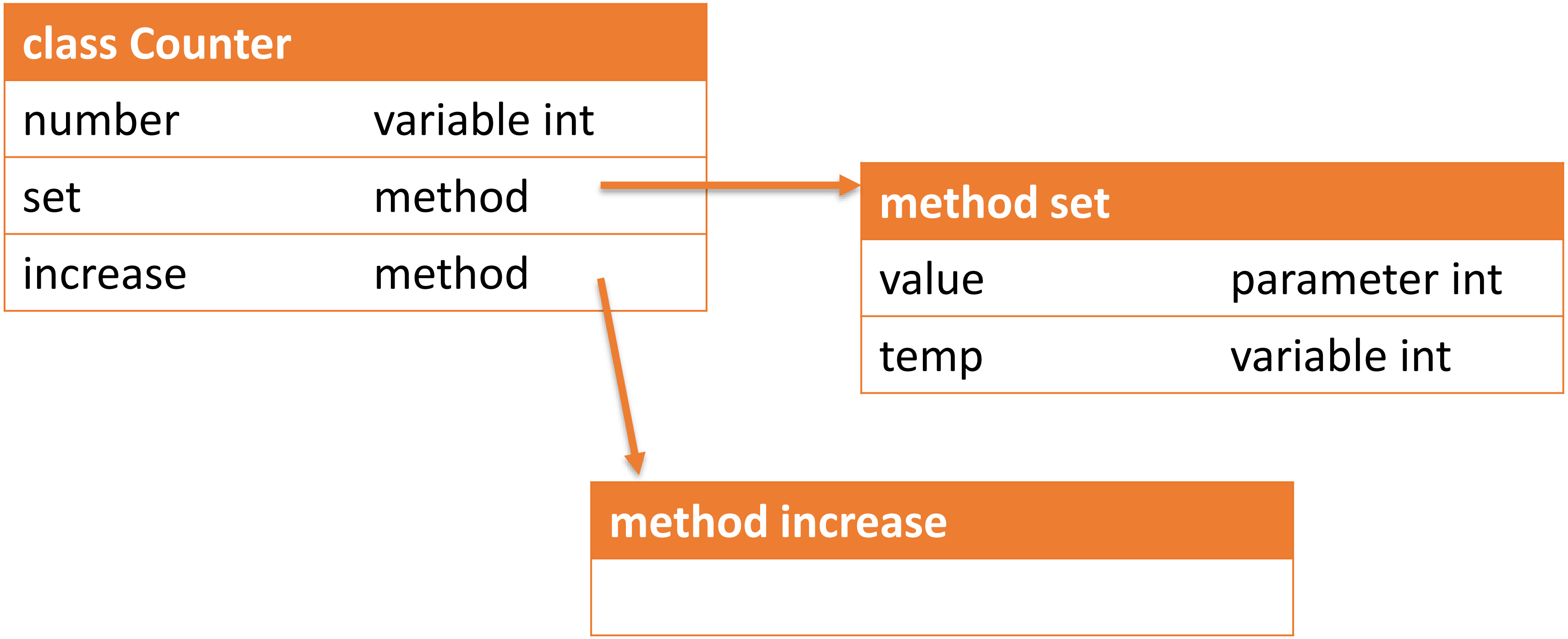
Declarations

```
class Counter {
   int number;

   void set(int value) {
     int temp;
     temp = number;
     number = value;
     writeInt(temp);
   }

   void increase() {
      number = number + 1;
   }
}
```

Declarations appear in hierarchical scopes

Symbol Table

- Data structure for managing declarations
- Reflects hierarchical program scopes



class Counter	
number	variable int
set	method
increase	method

method set	
value	parameter int
temp	variable int

method increase

Global Scope

- Multiple classes in program

```
class X {
  Y ref;
}
```

```
class Y {
  int x;
}
```

```
class Z {
  Z sub;
}
```

Global Scope

class X	
ref	var Y

class Y	
x	var int

class Z	
sub	var Z

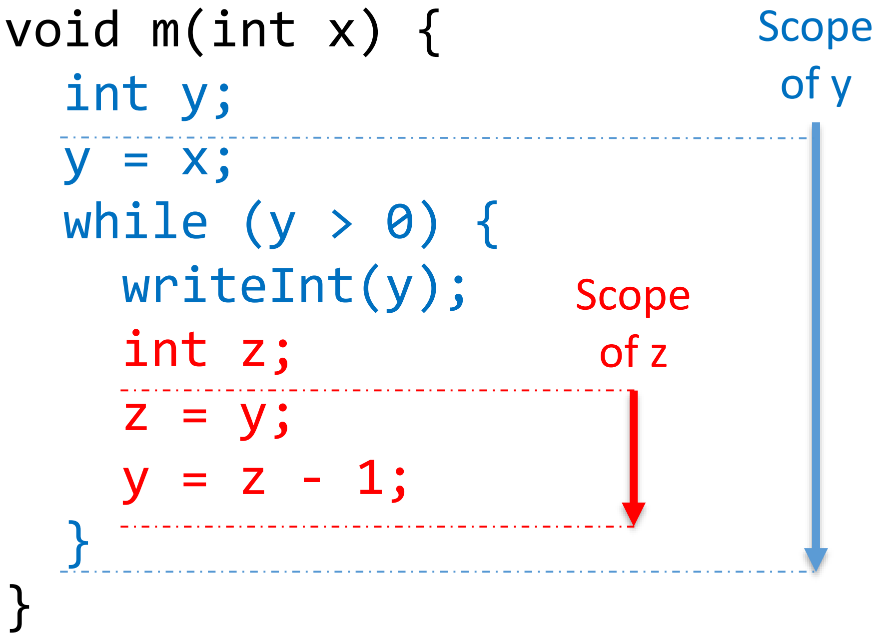
Shadowing

- Declaration in inner scopes shadow equally named declaration in other scopes

```
class C {
  int x;

  void m1() {
    int x;
  }

  void m2() {
    boolean x;
  }
}
```

Shadow outer x

Scopes of Local Variables



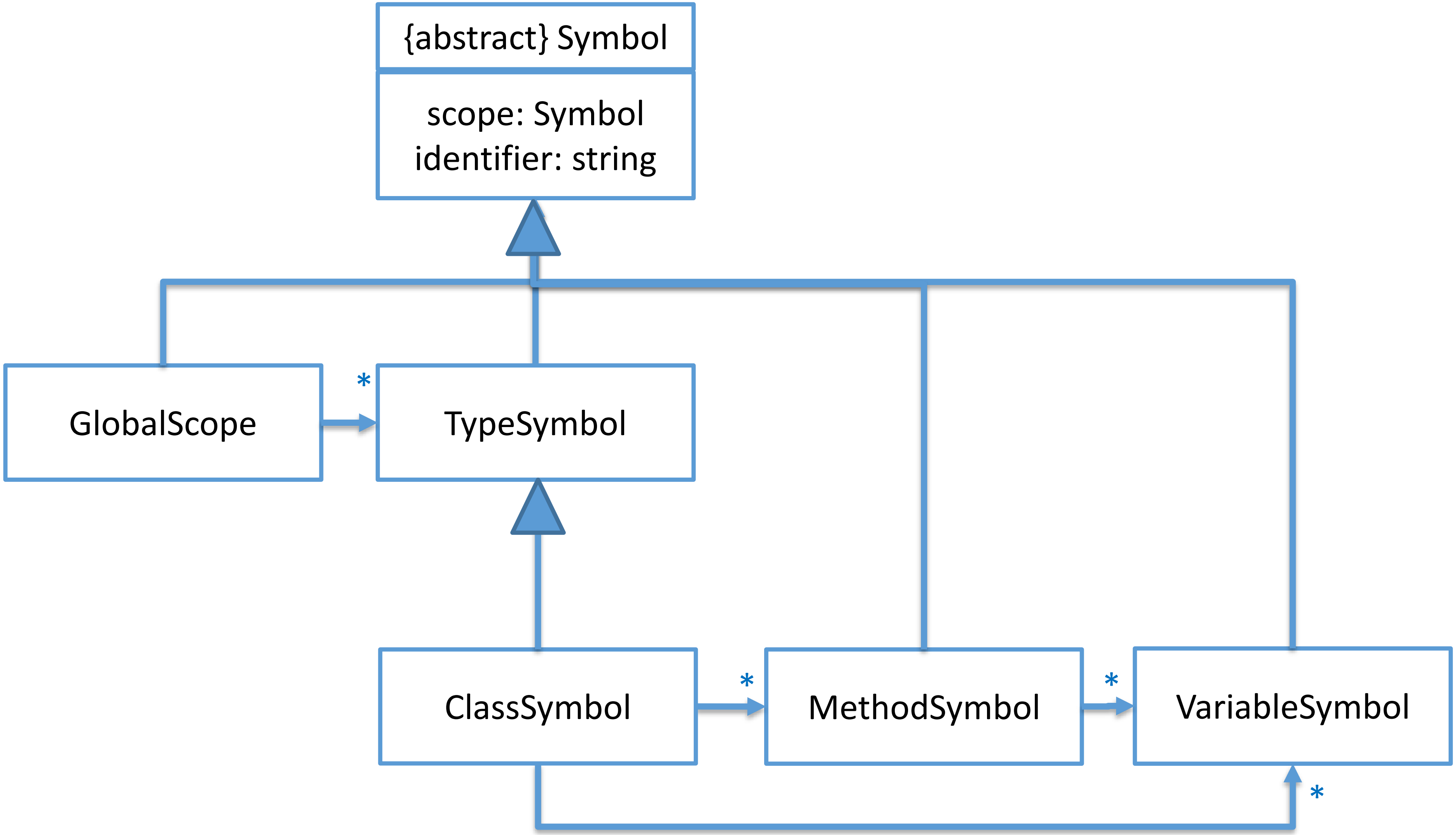
```
void m(int x) {
   int y;
   y = x;
   while (y > 0) {
      writeInt(y);
      int z;
      z = y;
      y = z - 1;
   }
}
```

Scope of y

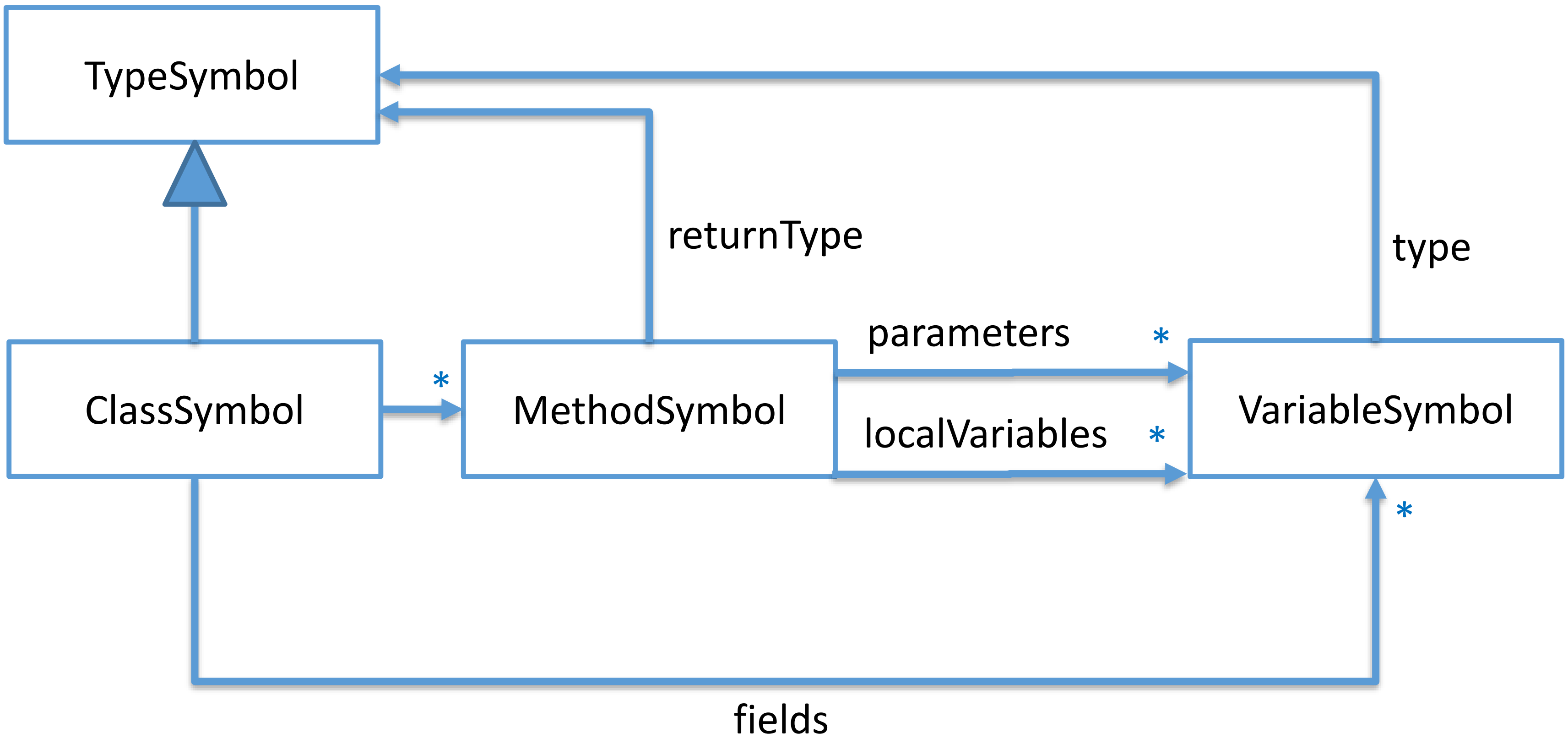
Scope of z

Shadowing within same-method-scopes are usually forbidden

Symbol Table Design



More Detailed Relations

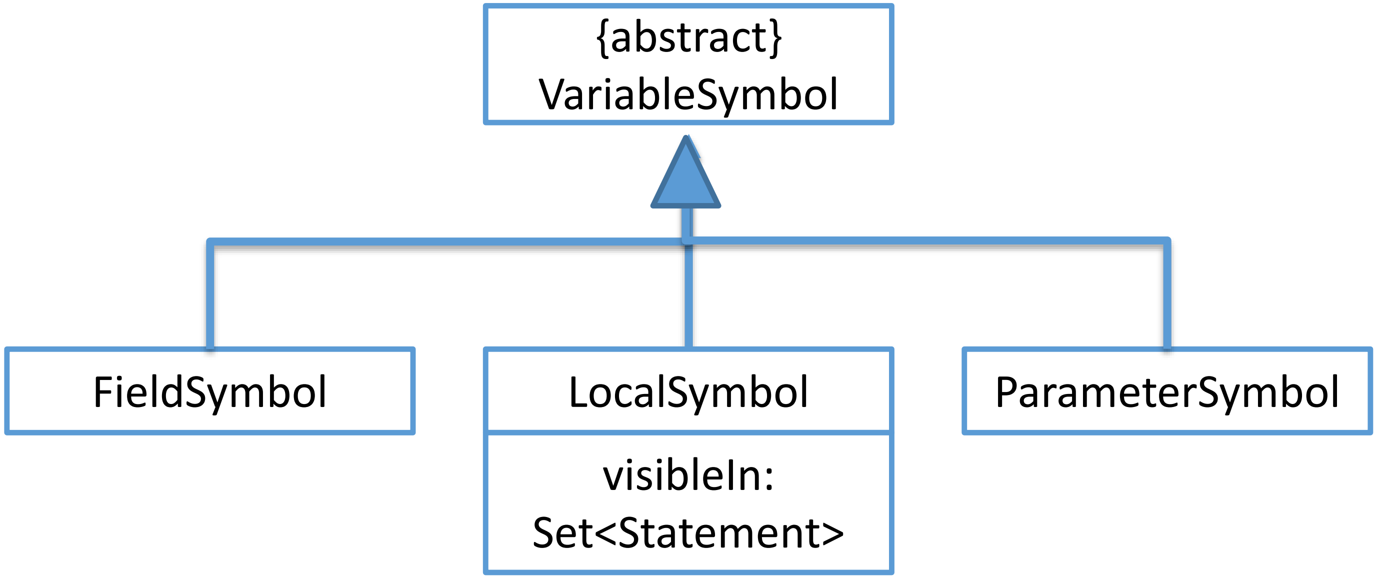


Design Aspects (1)

- Type information for variable symbol
  - Initially unresolved (identifier)
- Additional information
  - Class: Base class

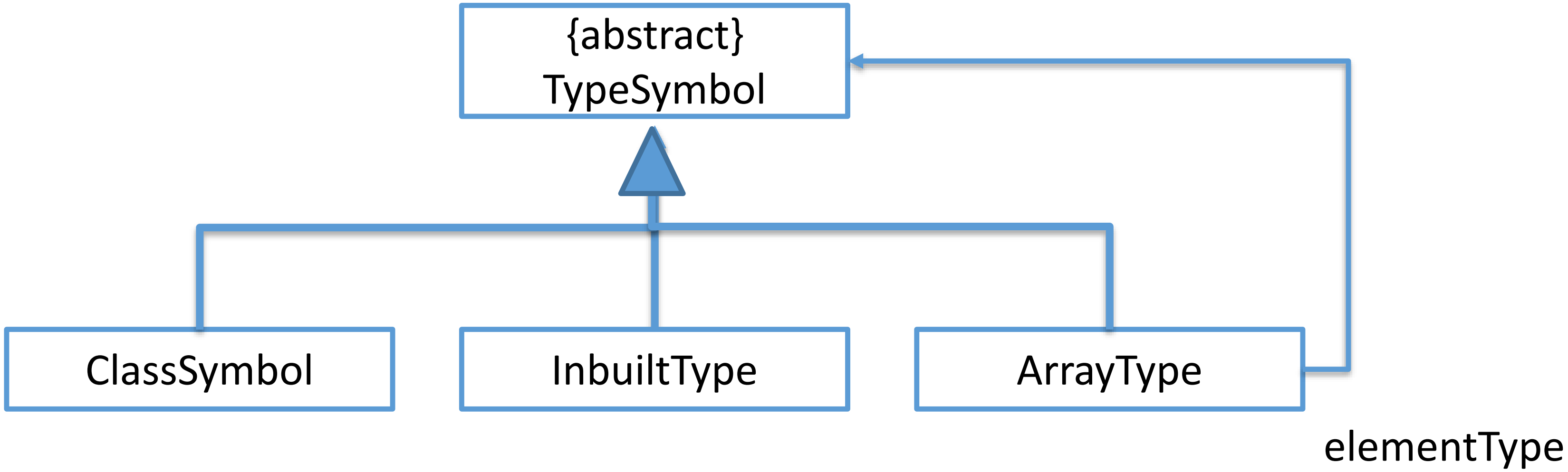
Design Aspects (2)

- Local variables
  - Remember declaration range (statements)
- Extended variable design:



Design Aspects (3)

- Extended type design:
  - Classes
  - Inbuilt types (int, boolean, string)
  - Arrays



Special Cases

- Predefined types: int, boolean, string
  - Insert as inbuilt types to global scope
- Predefined constants: true, false, null, this
  - true, false, null as constants in global scope
  - null is poly-type (compatible to all reference types)
  - “this” needs special handling in the analysis
- Predefined methods: writeString etc.
- Predefined variables: length
  - Only for array types
  - Read-only

Approach

- Construct symbol table
- Resolve types in table
- Resolve declarations in AST
- Resolve types in AST

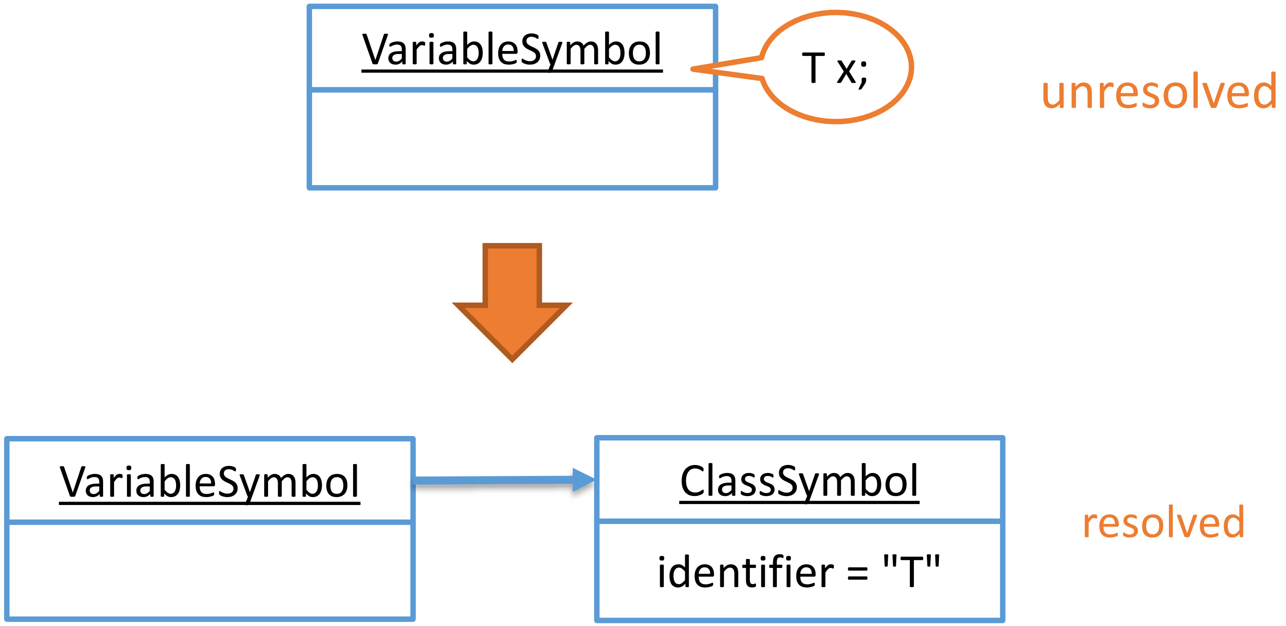
1. Construct Symbol Table

- Traverse AST
  - Start in global scope
  - For each class, method, parameter, variable
    - Insert symbol in surrounding scope
  - Explicit traversal or with visitor

Forward references => do not yet resolve type names or designators

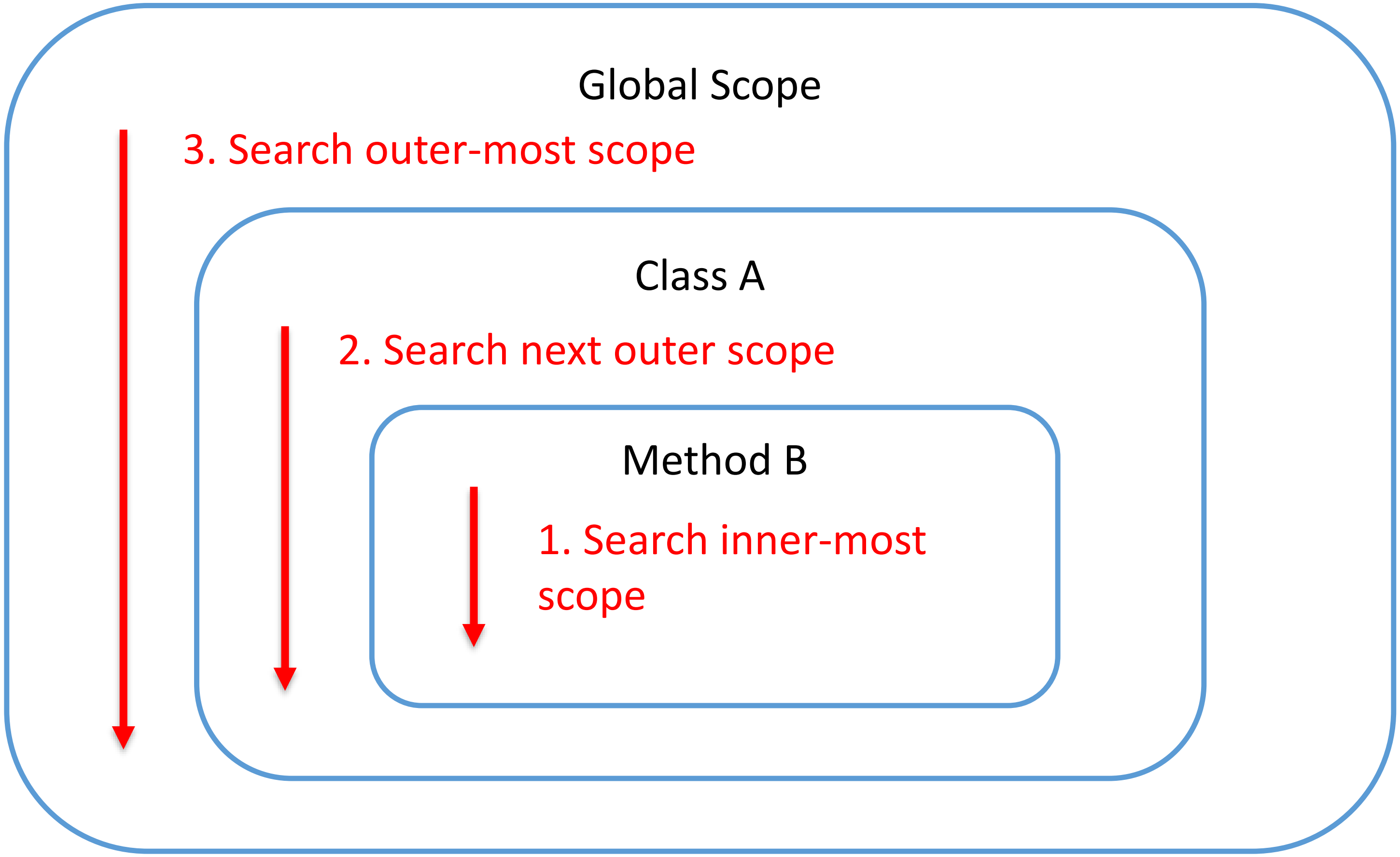
2. Resolve Types in Table

- For variables, parameters, return type etc.
- Search by identifier in symbol table



Name Resolution

- Question: "Which symbol declares identifier "id"?



Global Scope

3. Search outer-most scope

Class A

2. Search next outer scope

Method B

1. Search inner-most scope

Search Function

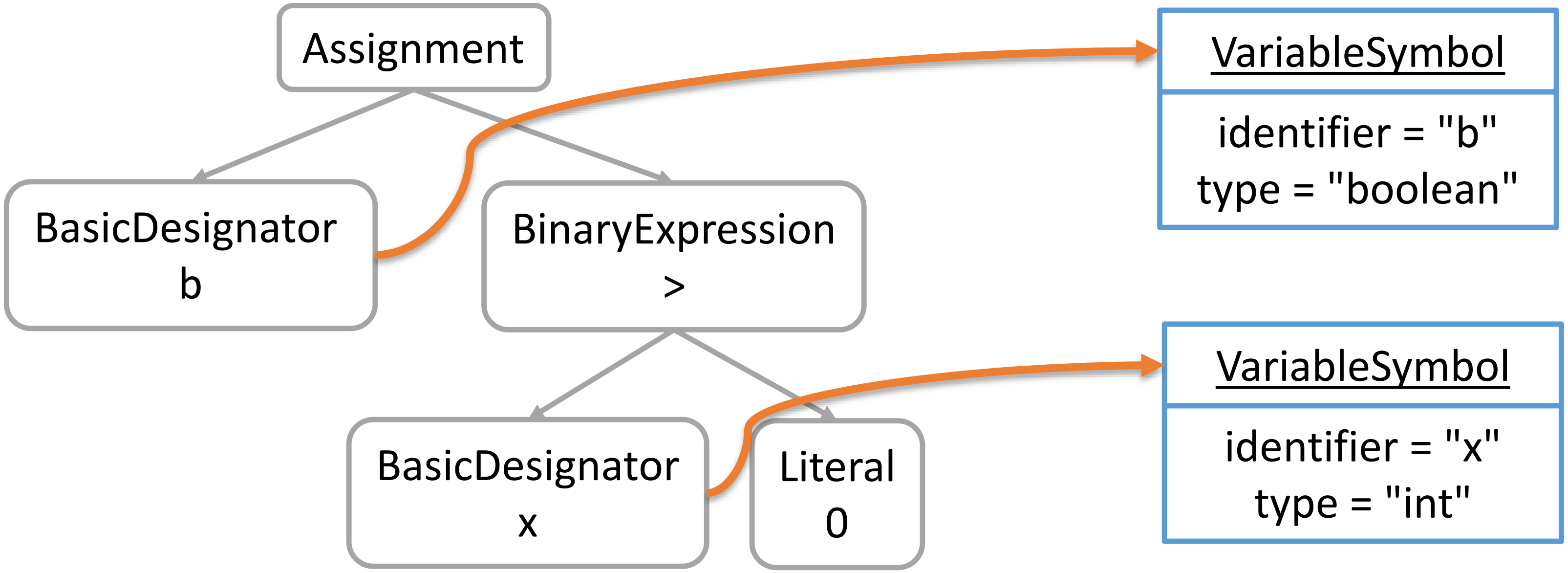
```
Symbol find(Symbol scope, String identifier) {
  if (scope == null) {
    return null;
  }
  for (Symbol declaration : scope.allDeclarations()) {
    if (declaration.getIdentifier().equals(identifier)) {
      return declaration;
    }
  }
  return find(scope.getScope(), identifier);
}
```

E.g. variables & methods inside class

Recursive search in next outer scope

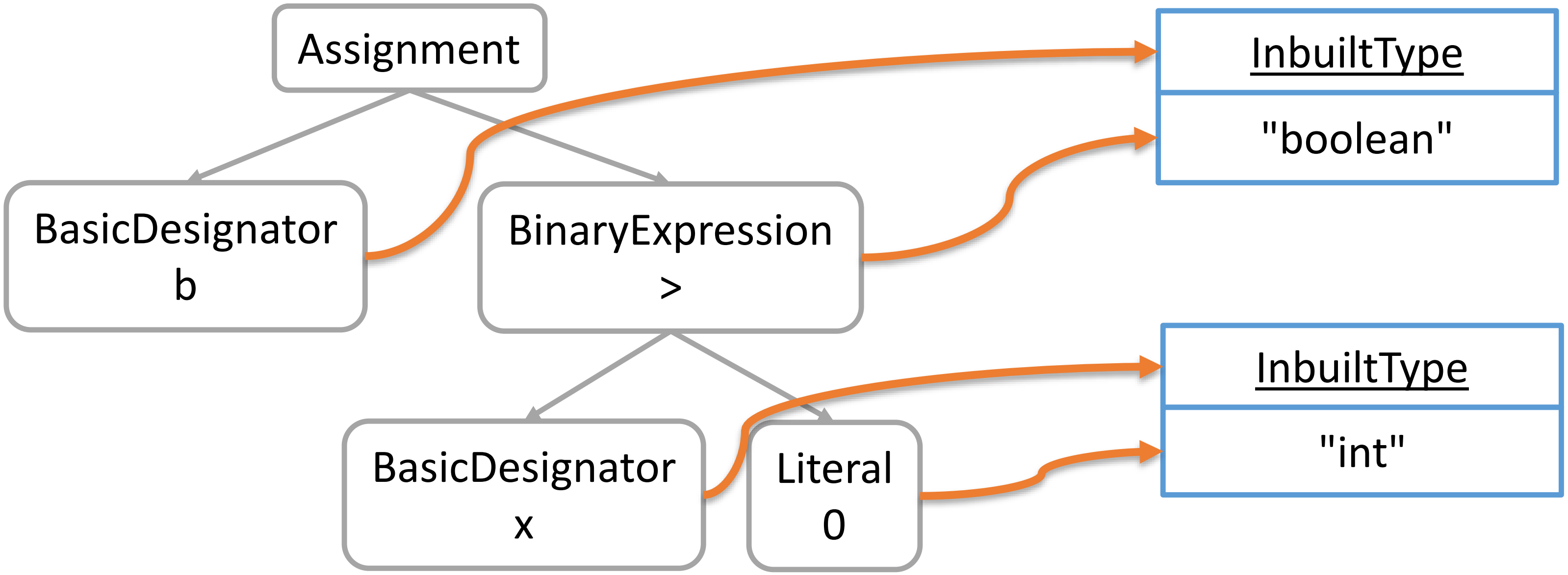
3. Resolve Declarations in AST

- Traverse execution code in AST
  - Method body
- Resolve each designator
  - Associate declaration



4. Resolve Types in AST

- Associate type for each expression
  - Literal: Predefined type
  - Designator: Type of declaration
  - Unary/BinaryExpression: Result of operator



Resolution Procedure

- Post order traversal
  - First resolve types in lower nodes
- We can leave AST unmodified
  - Use maps in symbol table
    - DesignatorNode -> Symbol (declaration)
    - ExpressionNode -> TypeSymbol (type)

Type Resolution per Visitor

```
@Override
public void visit(BinaryExpressionNode node) {
  Visitor.super.visit(node); // post-order traversal
  …
  if (node.getOperator() == Operator.PLUS) {
    symbolTable.fixType(node, GlobalScope.INT_TYPE);
  }
  …
}
```

All Resolved

Which checks do we have to perform now?

Semantic Checks

- All designators refer to variables/methods
- Types match on operators
- Compatible types on assignments
- Arguments matches parameters
- Conditions in if, while are boolean
- Return expression matches
- No multiple same declarations
- No identifier is a reserved keyword
- Exactly one main-method
- Array length is read-only
- More according language report…

Type rules

Types Match on Operators

`1 + true;` ⚡

`!3` ⚡

`1 && 2` ⚡

`array == null` ✓

Visitor Extension

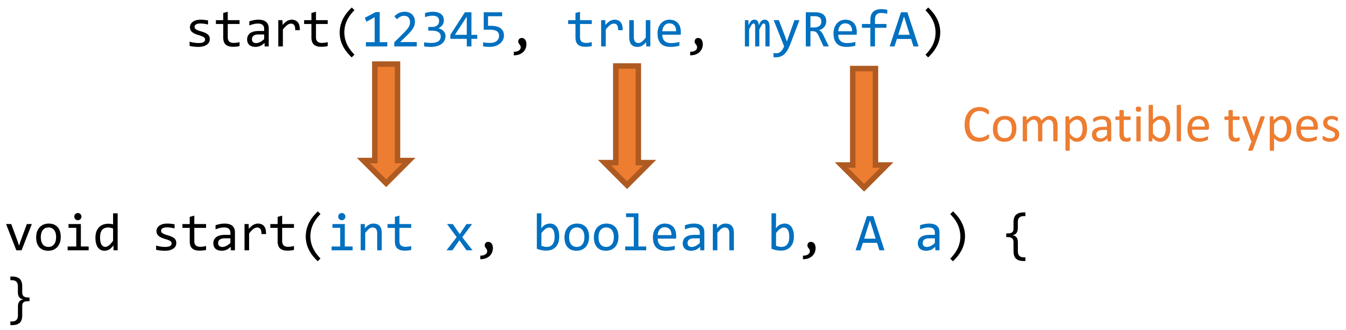
```
@Override
public void visit(BinaryExpressionNode node) {
  Visitor.super.visit(node); // post-order traversal
  var leftType = symbolTable.findType(node.getLeft());
  var rightType = symbolTable.findType(node.getRight());

  if (node.Operator == Operator.PLUS) {
    if (leftType != GlobalScope.INT_TYPE ||
        rightType != GlobalScope.INT_TYPE) {
       error();
    }
    symbolTable.fixType(node, GlobalScope.INT_TYPE);
  }
  …
}
```

Type-Compatibility on Assignments

- Same type left and right
  - int x; x = 12;
- Null assignment
  - int[] x; x = null;
  - For all reference types (strings, arrays, classes)
- OO type polymorphism
  - Vehicle v; v = new Car();
  - If Car is a sub-class of Vehicle

Argument List Matching Parameter List

- Uniquely defined static target method
- #arguments = #parameters
- nth argument is type-compatible to nth parameter



```
      start(12345, true, myRefA)

void start(int x, boolean b, A a) {
}
```

Compatible types

How much would overloading complicate this?

No Multiple Same Declarations

- In same scope

```
int x;
boolean x;
```

```
int f;
void f() { }
```

In Java allowed
but not in C++ and C#

```
void f() {}
void f(int x) {}
```

Allowed with overloading

Reserved Keywords

- Identifiers cannot be named like keywords
  - If not yet prevented by lexer

```
int int; ⚡

boolean true; ⚡

class void { ⚡
}
```

Additional Checks

- No exit without return (except void)
- Reading uninitialized variables
- Null dereferencing
- Invalid array index
- Division by zero
- Out of memory on new()

Not decidable in general
=> Static analysis
=> Runtime checks

Intermediate Representation

- Symbol table with resolved AST yields intermediate representation for the compiler backend
  - Code generation and optimization

Topic of next week

Review: Learning Goals

- ✓ Understand the purpose and functionality of the semantic analysis
- ✓ Understand the design and construction of a symbol table
- ✓ Know how to implement type resolution and type checks

Further Reading

- Dragon Book, selected sections
  - Section 2.7 (Symbol Tables)
  - Section 6.3 (Types and Declarations)
  - Section 6.5 (Type Checking)