Lecture 10

Semantic Analyzer

Part 2: Type checking

Hyosu Kim

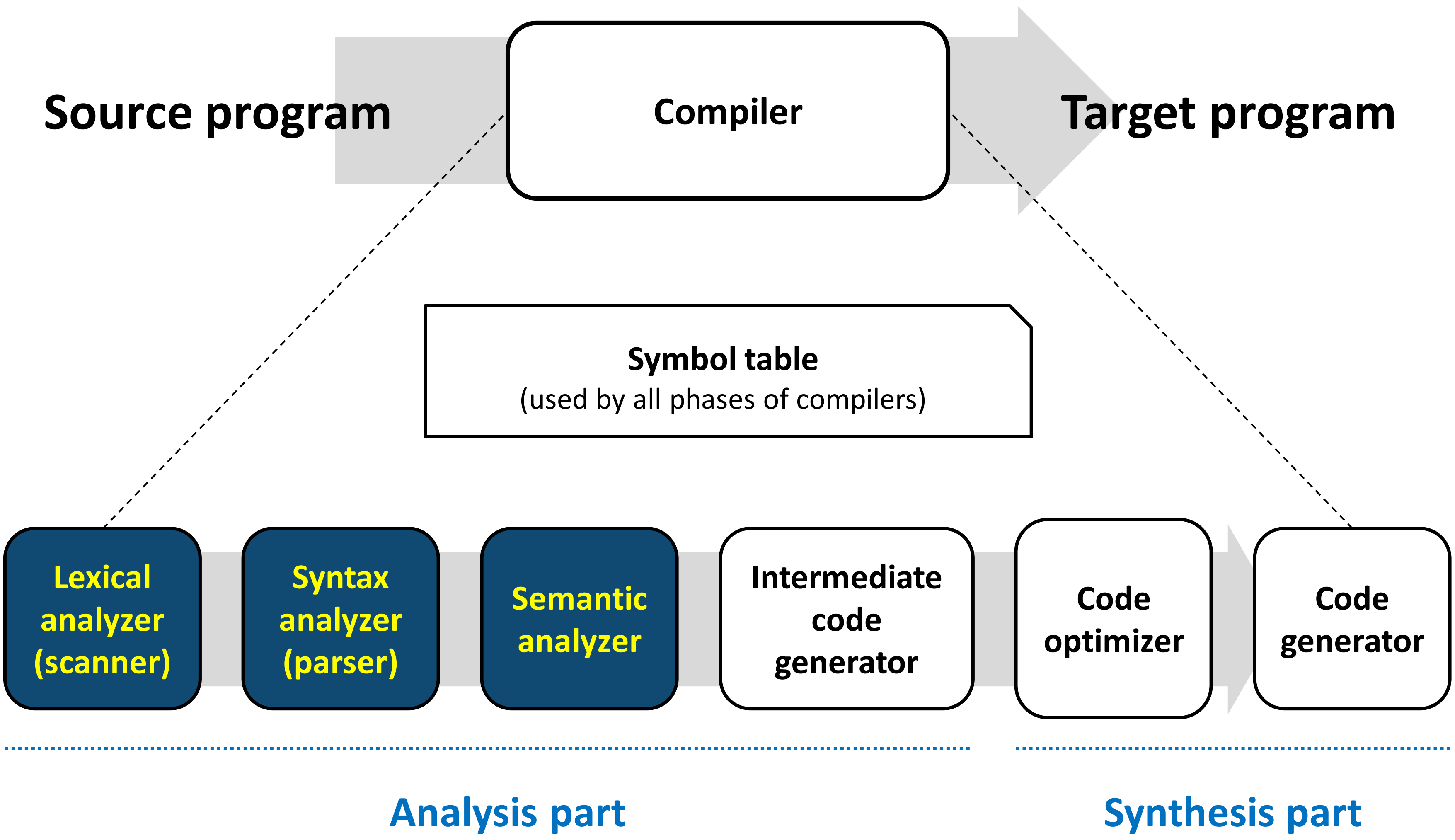
School of Computer Science and Engineering

Chung-Ang University, Seoul, Korea

https://sites.google.com/view/hyosukim

hskimhello@cau.ac.kr, hskim.hello@gmail.com

Overview


Source program
Compiler
Target program
Symbol table
(used by all phases of compilers)
Lexical analyzer (scanner)
Syntax analyzer (parser)
Semantic analyzer
Intermediate code generator
Code optimizer
Code generator
Analysis part
Synthesis part

Semantic analyzer

Checks many kinds of semantic grammars

Semantic grammars can be different depending on the programming language

Common semantic grammars

1. All variables must be declared before their use (globally or locally)
2. All variables must be declared only once (locally)
3. All functions must be declared only once (globally)
4. All variables must be used with the right type of constant or variables
5. All functions must be used with the right number and type of arguments

How to check them?????

Semantic analyzer

Checks many kinds of semantic grammars

Semantic grammars can be different depe

Through scope checking!!

Common semantic grammars

1. All variables must be declared before their us (globally or locally)
2. All variables must be declared only once (locally)
3. All functions must be declared only once (globally)
4. All variables must be used with the right type of constant or variables
5. All functions must be used with the right number and type of arguments

How to check them?????

Semantic analyzer

Checks many kinds of semantic grammars

Semantic grammars can be different depending on the programming language

Common semantic grammars

1. All variables must be declared before their use (globally or locally)
2. All variables must be declared only once (locally)
3. All functions must be declared only once (globally)
4. All variables must be used with the right type of constant or variables
5. All functions must be used with the right number and type of arguments

How to check them?????

Through type checking!!

Type, type system, and type checking

What is a type (data type)?

An attribute of data which tells compilers how programmers intend to use the data

Examples of type (in C)

- int a;
- char a;
- int a[32];
- char* a;
- bool a;

Type, type system, and type checking

What is a type (data type)?

An attribute of data which tells compilers how programmers intend to use the data

What is a type system?

A set of rules that assign specific types to the various constructs of a computer program

Examples of rules in a type system (e.g., in C)

- a && b

  Both a and b are expected to be booleans and the result of a && b is also of type boolean

- char a = b;

  b is expected to be a character

- a = 3.0f;

  a is expected to be of type float or double

Type, type system, and type checking

What is a type (data type)?

An attribute of data which tells compilers how programmers intend to use the data

What is a type system?

A set of rules that assign specific types to the various constructs of a computer program

What is a type checking?

Ensuring that the types of the operands match the type expected by the operator

Type, type system, and type checking

Why do we need type checking?

A lot of type errors can occur in our source code

```
while (foo(x + 5) <= 10) {
    if (1.0 + 2.0) {

    } else if (1 == null) {

    }
}
```

Type, type system, and type checking

Why do we need type checking?

Let's suppose the following assembly language: **add $r1, $r2, $r3**

What are types of $r1, $r2, and $r3????

- If all of them are of type int, this operation is correct

  (e.g., int a = 0; int b = 1; int c = a + b;)

- If $r2 is a pointer but $r3 is a floating point number, this operation is not allowed

  (e.g., int* a = malloc(…); float b = 3.5; int c = a + b;)

But, both operations have the same assembly language implementation!!!!

- We can not distinguish them in the assembly language level
- **We should do the type checking with higher-level representations (e.g., AST)**

Two kinds of languages

- **Statically-typed languages**
  - The type of a variable is determined / known at compile time
  - Type checking is performed during compile-time (before run-time)
    - e.g., C, C++, java, Go, …: int a = 3;   a = “ok”;
  - **Advantage: better run-time performance + easy-to-understand/easy-to-find type-related bugs (due to strict rules)**

- **Dynamically-typed languages**
  - The type of a variable is associated with run-time values
  - Type checking is performed on the fly, during execution
  - e.g., python, javascript, …: a = 3; a = “ok”;
  - **Advantage: higher flexibility + rapid prototyping support**

How types are used/checked in practice?

In statically-typed languages,

Programmers declare types for all identifiers statically

- Types are associated with specific keywords (e.g., int, float, bool, char, …)

Compilers

1) Infer the type of each expression from the types of its components
2) Confirm that the types of expressions matches what is expected

How??

Rules of inference

Inference rules usually have the form of "if-then" statements

If hypothesis is true, then conclusion is true

"If you are a student at CAU, then you are smart"

Rules of inference

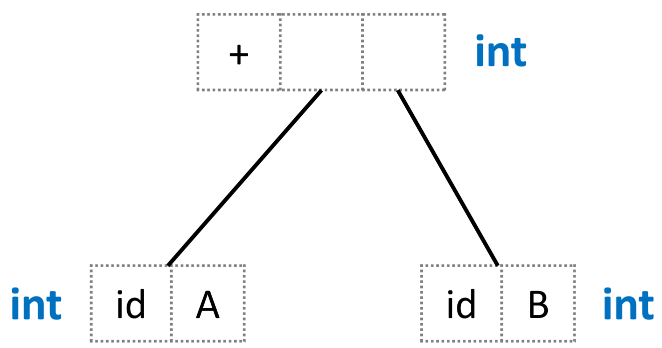
Inference rules usually have the form of “if-then” statements

If hypothesis is true, then conclusion is true

Type checking computes via reasoning

- e.g., for an expression A + B,

  if A has type int and B has type int, **then** A + B has type int



Rules of inference

Inference rules usually have the form of "if-then" statements

If hypothesis is true, then conclusion is true

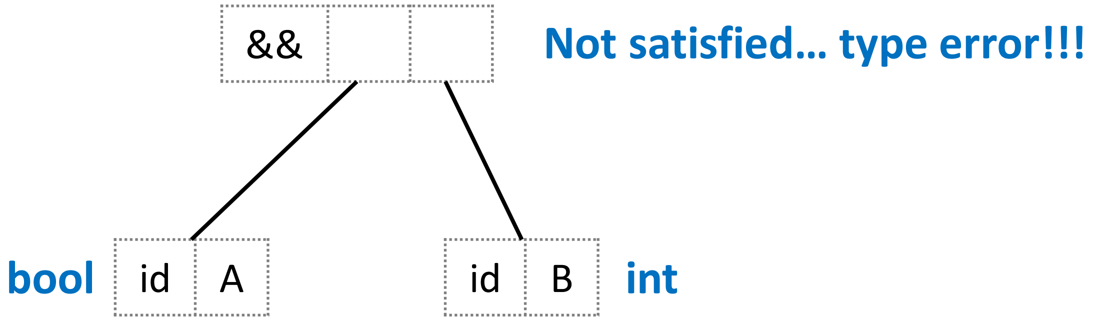
Type checking computes via reasoning

- e.g., for an expression A + B,

  if A has type int and B has type int, **then** A + B has type int

- e.g., for an expression A && B,

  if A has type bool and B has type bool, **then** A && B has type bool



Rules of inference

Notations for rules of inferences

- $x: T$ = x has type T
- $\frac{Hypothesises}{Conclusions}$ = if-then statement "if hypotheses are true, conclusions are true"
- ⊢ = "we can infer"

Examples

For an expression A && B

- if we can infer that A has type bool and B has type bool, then we can infer that A && B has type bool
- $\frac{We\ can\ infer\ that\ A\ has\ type\ bool\ and\ B\ has\ type\ bool}{We\ can\ infer\ that\ A\ \&\&\ B\ has\ type\ bool}$
- $\frac{\vdash A\ has\ type\ bool \quad \vdash B\ has\ type\ bool}{\vdash A\ \&\&\ B\ has\ type\ bool}$
- $\frac{\vdash A: bool \quad \vdash B: bool}{\vdash A\ \&\&\ B: bool}$

Rules of inference

More examples

- Axioms: $\dfrac{}{\vdash true{:}bool}$ $\quad\dfrac{}{\vdash false{:}bool}$ $\quad\dfrac{}{\vdash 1{:}int}$ $\quad\dfrac{}{\vdash 1000{:}int}$

- Simple inference rules:

$$\frac{i \textit{ is an integer constant}}{\vdash i{:}int} \qquad \frac{s \textit{ is a string constant}}{\vdash s{:}string} \qquad \frac{d \textit{ is a double constant}}{\vdash d{:}double}$$

- More complex rules:

$$\frac{\vdash e_1{:}\,int \quad \vdash e_2{:}\,int}{\vdash e_1{+}e_2{:}int} \qquad \frac{\vdash e_1{:}\,double \quad \vdash e_2{:}\,double}{\vdash e_1{+}e_2{:}\,double}$$

$$\frac{\vdash e_1{:}T \qquad \vdash e_2{:}T \qquad T \textit{ is short,int,long,float or double}}{\vdash e_1{+}e_2{:}\,T}$$

Rules of inference

More examples

- More complex rules:

$$\frac{\vdash e_1 : T \qquad \vdash e_2 : T \qquad T \textit{ is a primitive type } (e.g., byte, short, int, long, float, double, char)}{\vdash e_1 == e_2 : bool}$$

$$\frac{\vdash e_1 : T \qquad \vdash e_2 : T \qquad T \textit{ is a primitive type } (e.g., byte, short, int, long, float, double, char)}{\vdash e_1 \leq e_2 : bool \qquad \vdash e_1 \geq e_2 : bool \qquad \vdash e_1 != e_2 : bool}$$

Problem #1: free variables

Free variables?

A variable is free in an expression if its type is not defined/declared within the expression

Examples of free variables

- int y = **x** + 3;

- **y** = **x** + 3;

- if (**a** == **b**)

Problem #1: free variables

When $x + 3$, we can use the following rule

$$\frac{\vdash e_1 : T \qquad \vdash e_2 : T \qquad T \; is \; short, int, long, float \; or \; double}{\vdash e_1 + e_2 : \; T}$$

But, there is no enough information to decide the type of a free variable x

$$\frac{\boldsymbol{x \; is \; a \; variable}}{\vdash \boldsymbol{x} :??}$$

Solution: adding scope information

Scoping information can give types for free variables

$$S \vdash e : T$$

We can infer that an expression $\boldsymbol{e}$ has type $\boldsymbol{T}$ in scope $\boldsymbol{S}$

- Types are now proven relative to the scope the expressions are in

Examples

$$\frac{}{S \vdash true : bool} \quad \frac{}{S \vdash 1 : int} \quad \frac{i \text{ is an integer constant}}{S \vdash i : int} \quad \frac{d \text{ is a double constant}}{S \vdash d : double}$$

$$\frac{S \vdash e_1 : int \quad S \vdash e_2 : int}{S \vdash e_1 + e_2 : int} \quad \frac{S \vdash e_1 : double \quad S \vdash e_2 : double}{S \vdash e_1 + e_2 : double}$$

Solution: adding scope information

More examples

$$\frac{x\ is\ a\ variable \quad x\ is\ declared\ in\ scope\ S\ with\ type\ int}{S \vdash x : int}$$

$$\frac{\begin{array}{c} f\ is\ a\ non-member\ function\ in\ scope\ S \\ f\ is\ globally\ declared\ with\ type\ (T_1, \ldots T_n) \rightarrow U \\ S \vdash e_i : T_i \quad for\ 1 \leq i \leq n \end{array}}{S \vdash f(e_1, \ldots, e_n) : U}$$

$$\frac{S \vdash e_1 : T[\,] \quad S \vdash e_2 : int}{S \vdash e_1[e_2] : T}$$

Problem #2: statements

So far, we've defined inference rules for expressions

e.g., A + B, A&&B, …

Q. How to check whether statements are semantically well-formed or not?

Examples of semantically well-formed statements

- int a = 3;
- bool well_formed = true;
- string s = "this is well-formed";
- while (1 > 0) { well-formed statement1; well-formed statement2; well-formed statement3; …}
- if (10 == 10) { well-formed statements; … } else {well-formed statements; … }

Solution: defining well-formedness rules

Extend our proof system (rules of inferences) to statements

$$S \vdash WF(stmt)$$

We can infer that a statement $stmt$ is semantically well-formed in scope S

Examples

- For assignment statements

$$\frac{S \vdash e_1 \text{ is a variable} \quad S \vdash e_2 \text{ is a variable or constant} \quad S \vdash e_1 : T \quad S \vdash e_2 : T}{S \vdash WF(e_1 = e_2;)}$$

- For return statements

$$\frac{S \text{ is in a function returning type } T \quad S \vdash e : T}{S \vdash WF(return\ e;)} \qquad \frac{S \text{ is in a function returning type void}}{S \vdash WF(return;)}$$

Solution: defining well-formedness rules

More examples

- For a set of statements

$$\frac{S \vdash WF(stmt_1) \quad S \vdash WF(stmt_2) \quad \dots \quad S \vdash WF(stmt_n)}{S \vdash WF(stmt_1 \quad stmt_2 \quad \dots \quad stmt_n)}$$

- For while loop statements

$$\frac{S \vdash e{:}bool \quad S' \textit{ is the scope inside the while loop} \quad S' \vdash WF(stmt_1 \quad stmt_2 \quad \dots \quad stmt_n)}{S \vdash WF(while(e)\{stmt_1 \quad stmt_2 \quad \dots \quad stmt_n\})}$$

- **Q. for if-else statements**

How types are used/checked in practice?

In statically-typed languages,

Programmers declare types for all identifiers

- Types are associated with specific keywords (e.g., int, float, bool, char, …)

Compilers

1) Infer the type of each expression from the types of its components
2) Confirm that the types of expressions matches what is expected

Especially, by using the rules of inference + scope information + well-formedness rules

How to use the inference rules for type checking???

How types are used/checked in practice?

How to use the inference rules for type checking???

- First, before doing type checking, do scope checking.
- For each statement:
  - Do type checking for any subexpressions it contains
  - Do type checking for child statements
  - Check the overall well-formedness

Example: for a statement “if (A && B) { var = 3; }”

- Do type checking for a subexpression “A&&B”
- Do type checking for a child statement “var = 3;”
  - Do type checking for a subexpression “var” and “3”
  - Check the well-formedness of “var = 3;”
- Check the well-formedness of “if (A&&B) { var = 3; }”

Implementation of type checking

Recursively walk abstract syntax trees

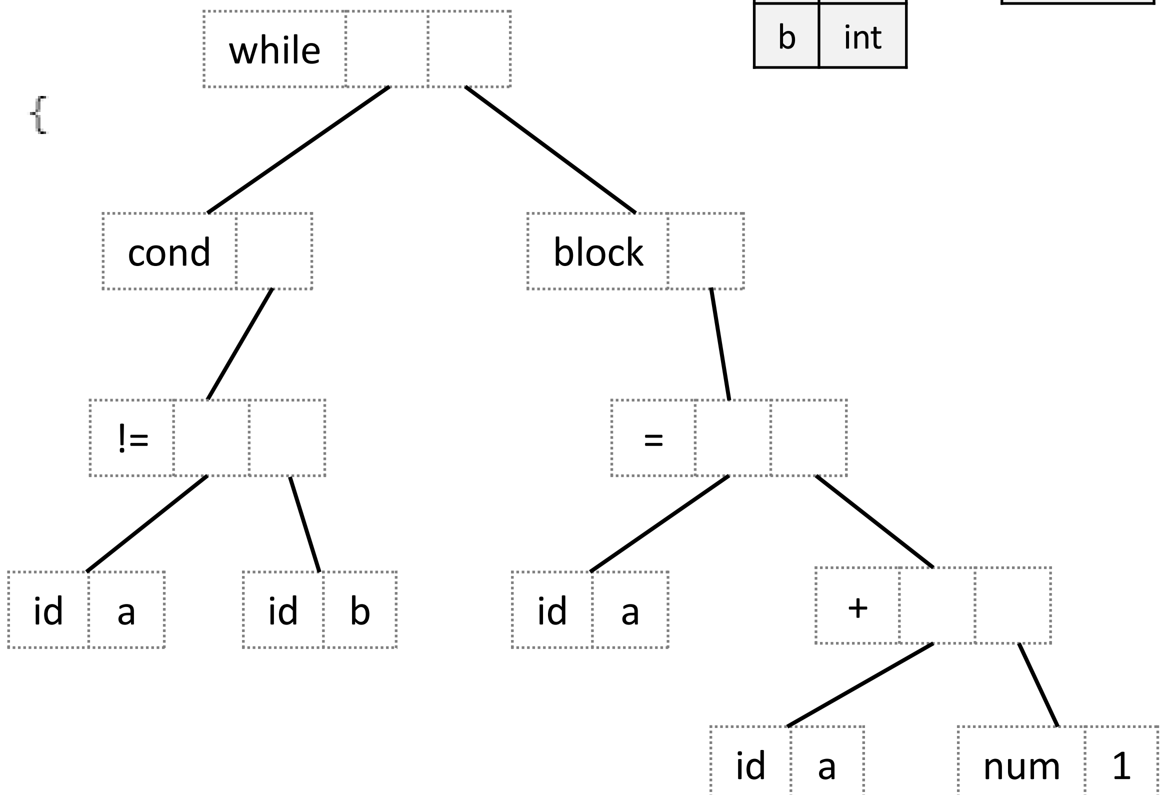
1. Do scope checking

```
...

while (a != b) {
    a = a + 1;
}
...
```

Symbol table after scope checking

a	int
b	int

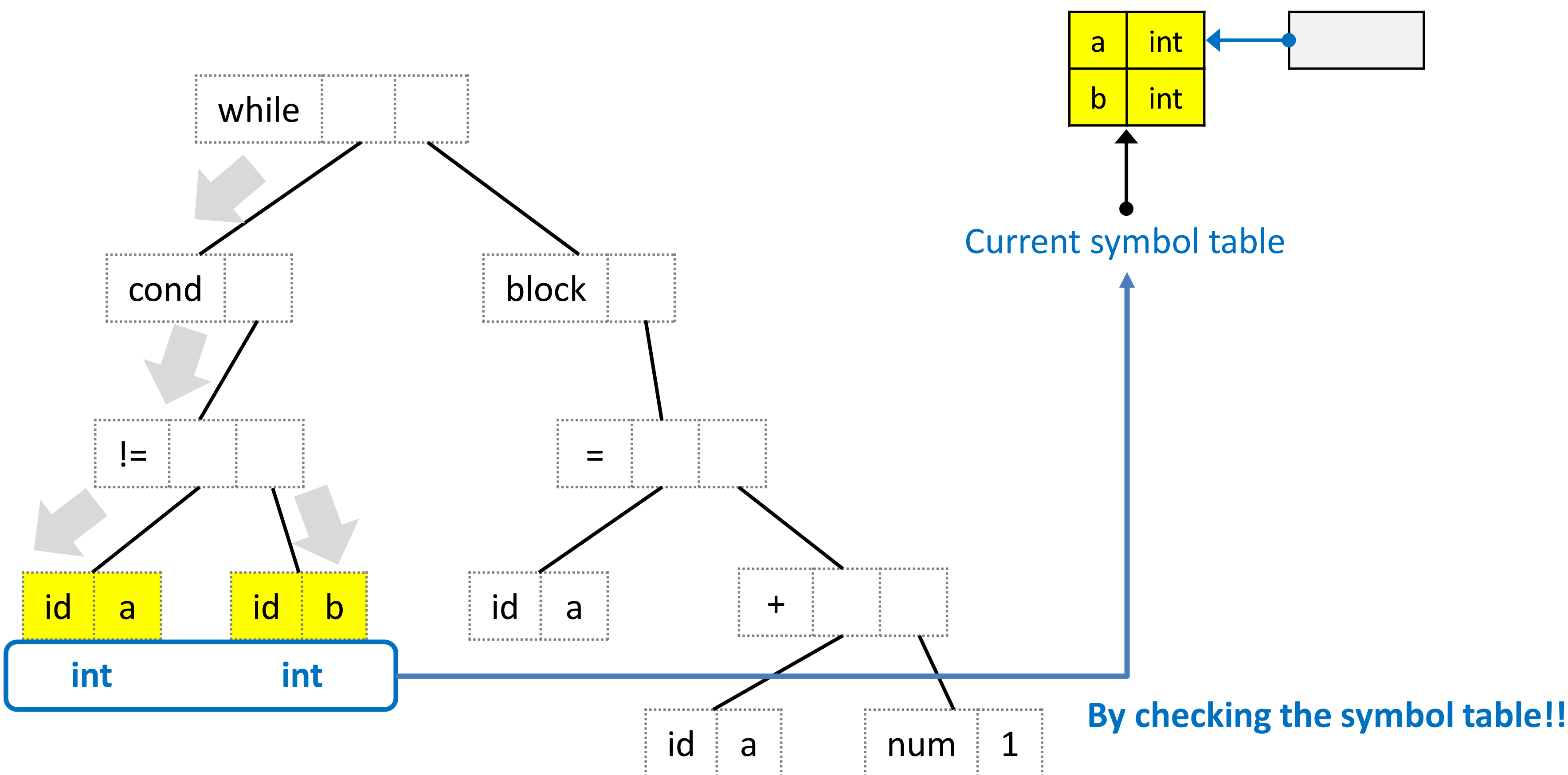


Implementation of type checking

Recursively walk abstract syntax trees

2. Do type checking for a subexpression **a != b**

Symbol table after scope checking

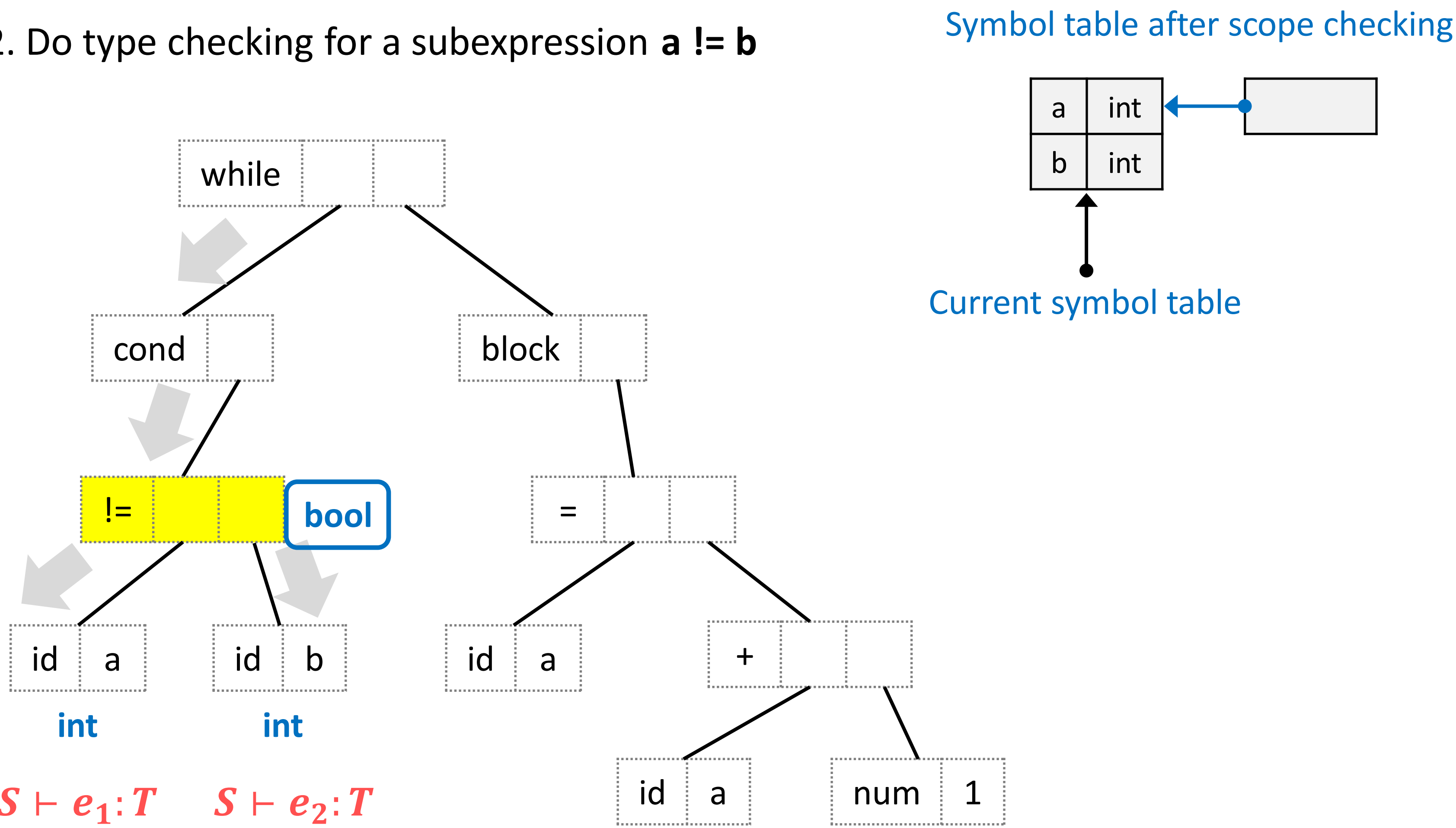


$$\frac{S \vdash e \textit{ is a variable} \quad S \vdash e \textit{ is declared with type } T}{S \vdash e : T}$$

Implementation of type checking

Recursively walk abstract syntax trees

2. Do type checking for a subexpression **a != b**

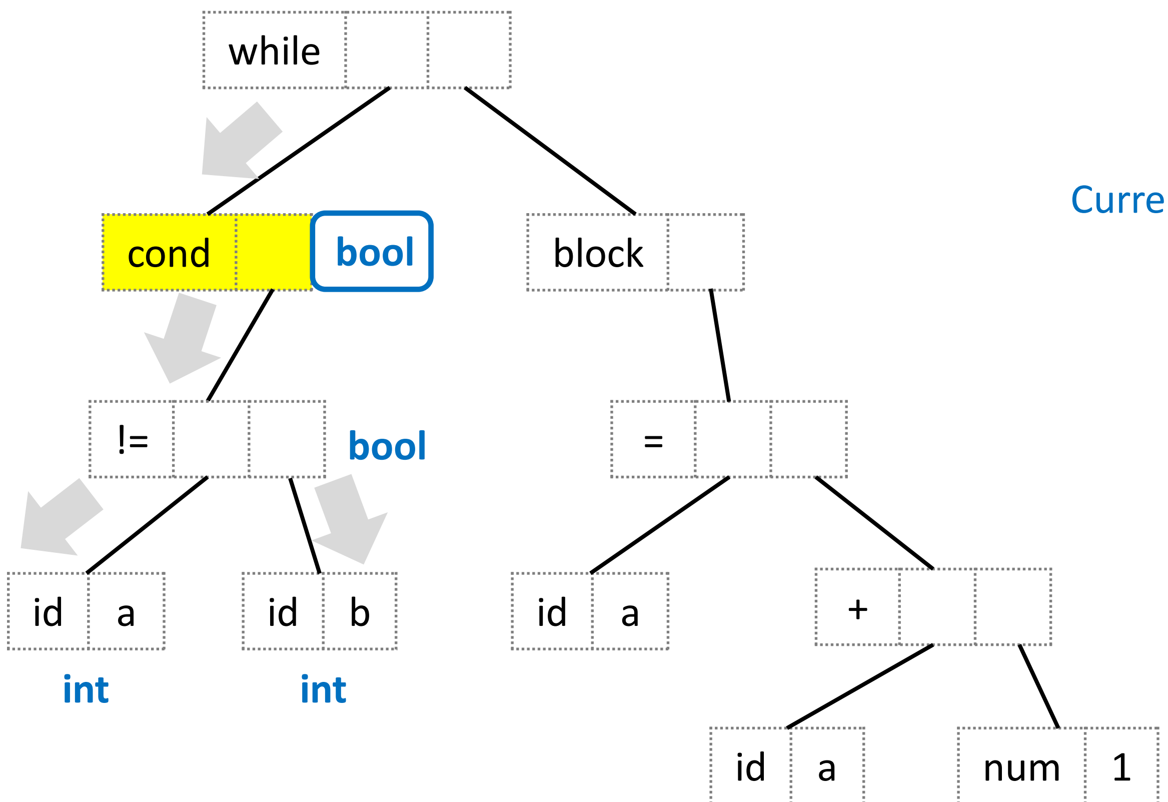


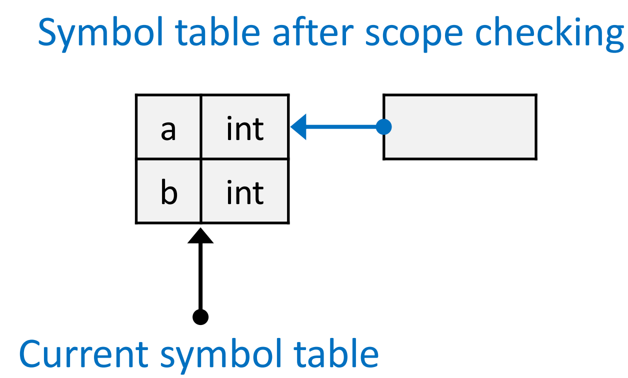
$$\frac{S \vdash e_1 : T \qquad S \vdash e_2 : T}{S \vdash e_1 != e_2 : T}$$

Implementation of type checking

Recursively walk abstract syntax trees

2. Do type checking for a subexpression **a != b**

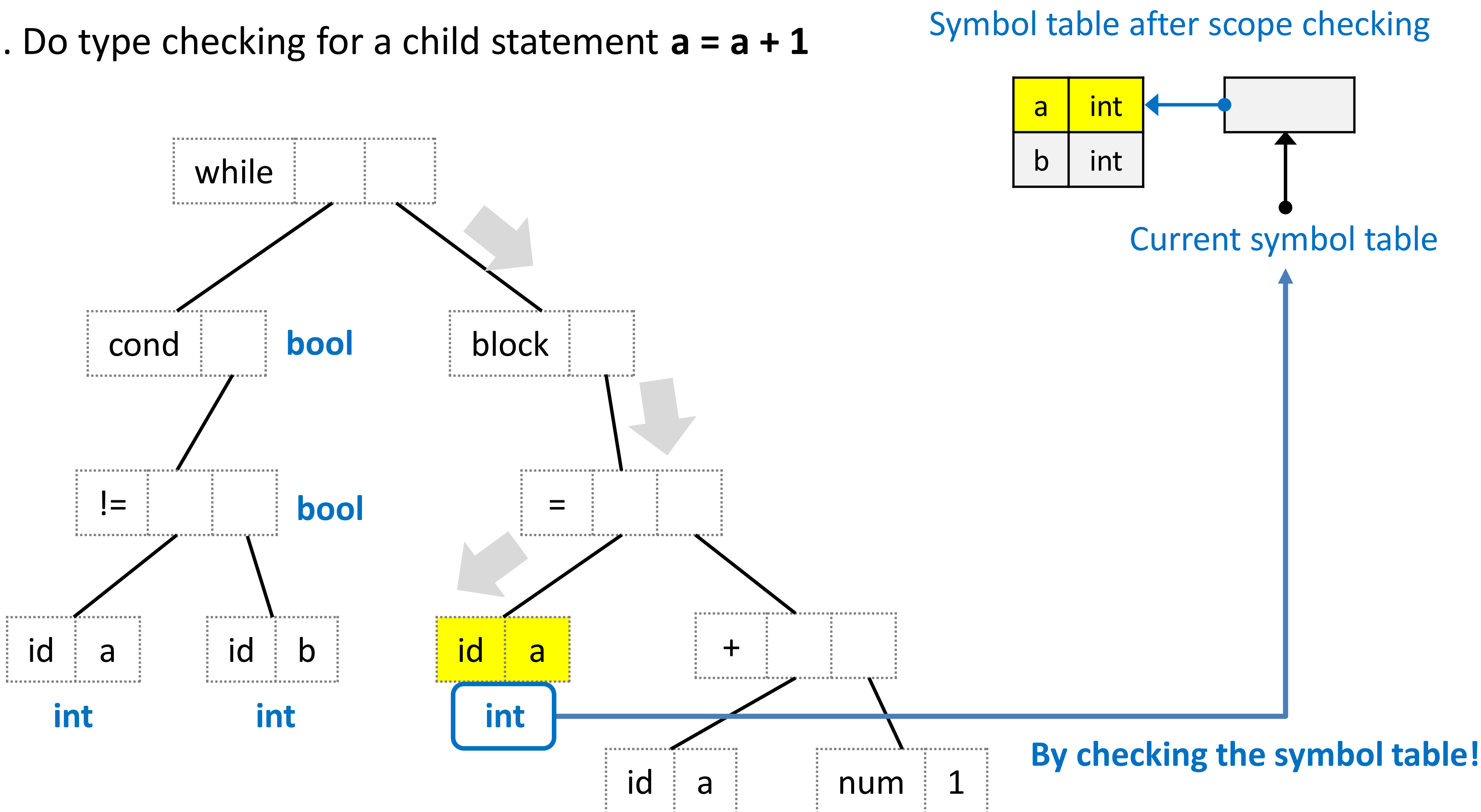




Implementation of type checking

Recursively walk abstract syntax trees

3. Do type checking for a child statement **a = a + 1**

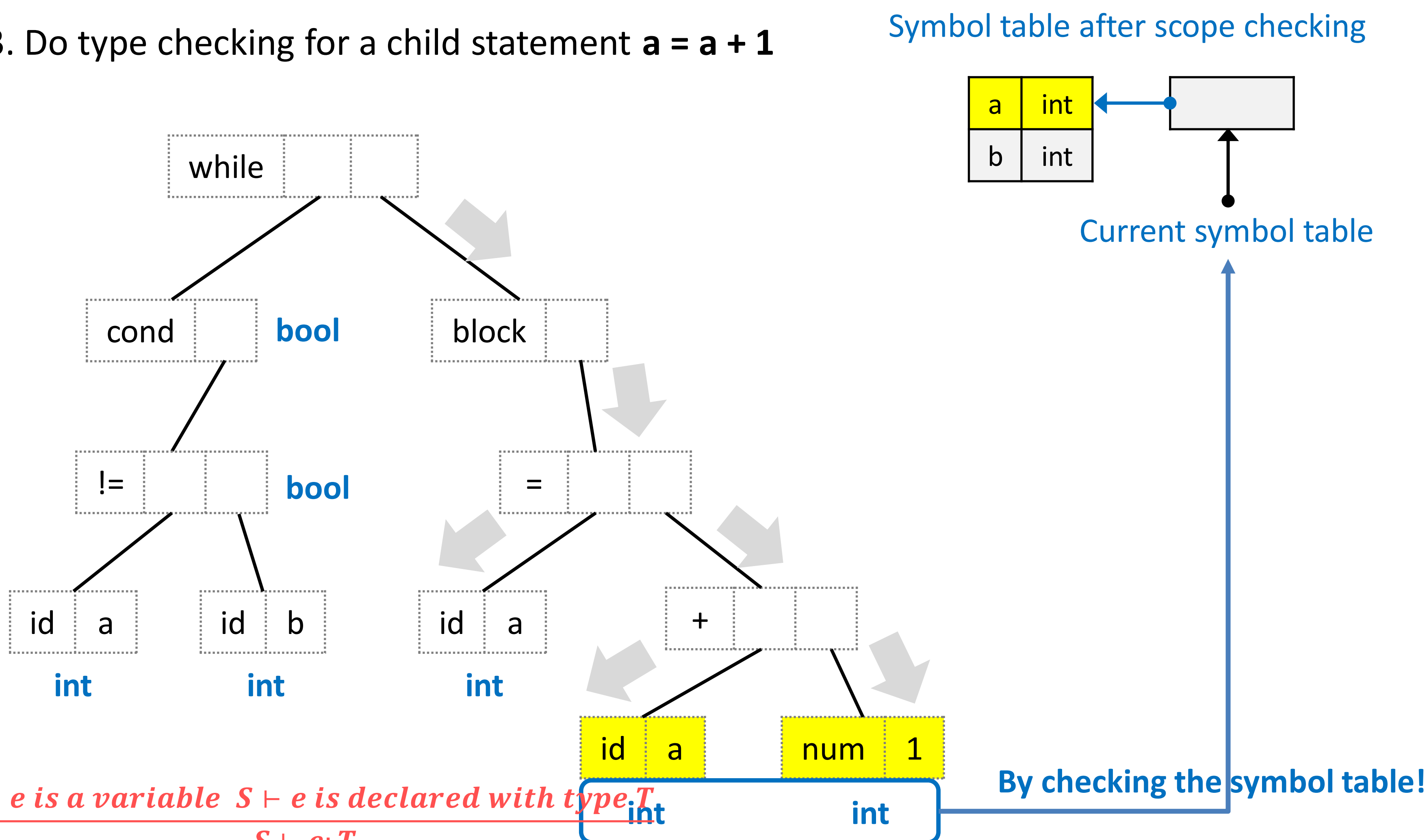


$$\frac{S \vdash e \text{ is a variable} \quad S \vdash e \text{ is declared with type } T}{S \vdash e : T}$$

Implementation of type checking

Recursively walk abstract syntax trees

3. Do type checking for a child statement **a = a + 1**

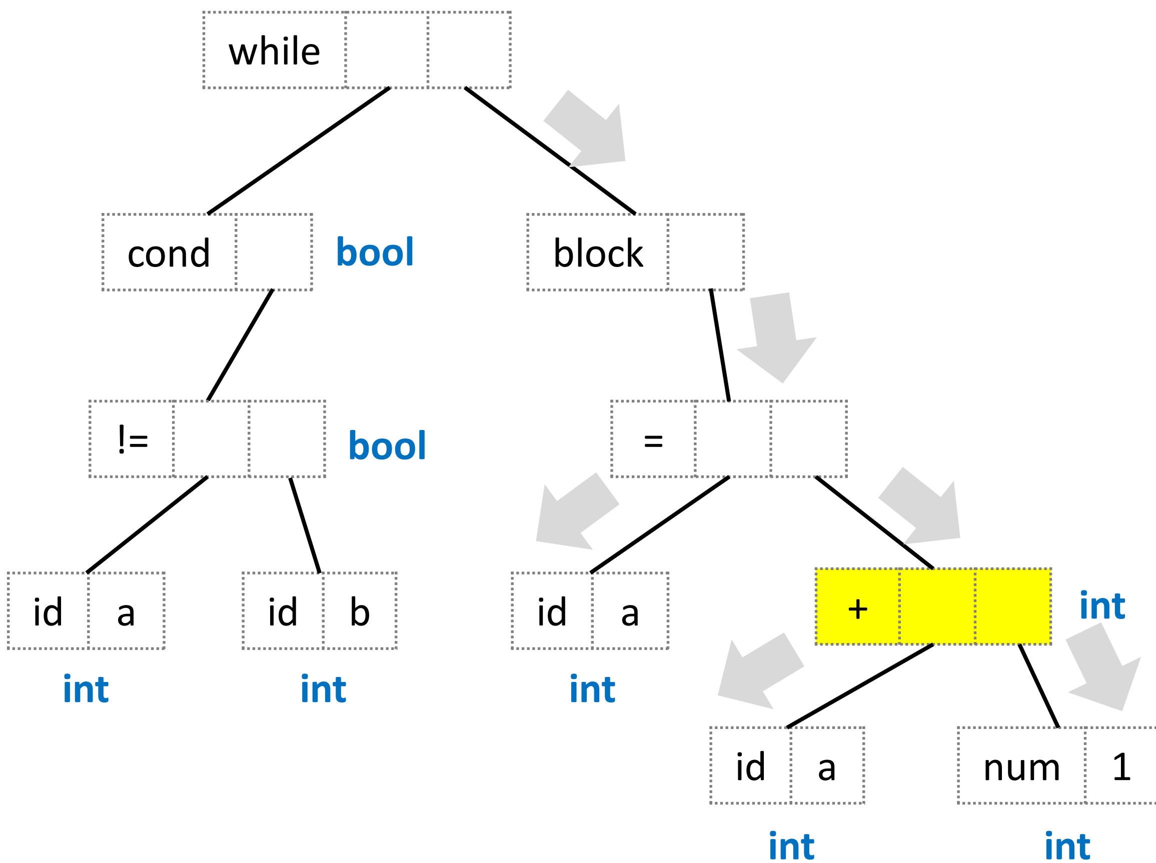


$$\frac{S \vdash e \text{ is a variable} \quad S \vdash e \text{ is declared with type } T}{S \vdash e : T}$$

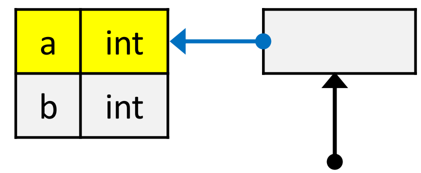
Implementation of type checking

Recursively walk abstract syntax trees

3. Do type checking for a child statement **a = a + 1**



Symbol table after scope checking



Current symbol table

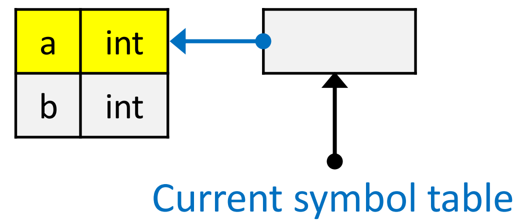
$$\frac{S \vdash e_1 : T \qquad S \vdash e_2 : T}{S \vdash e_1 + e_2 : T}$$

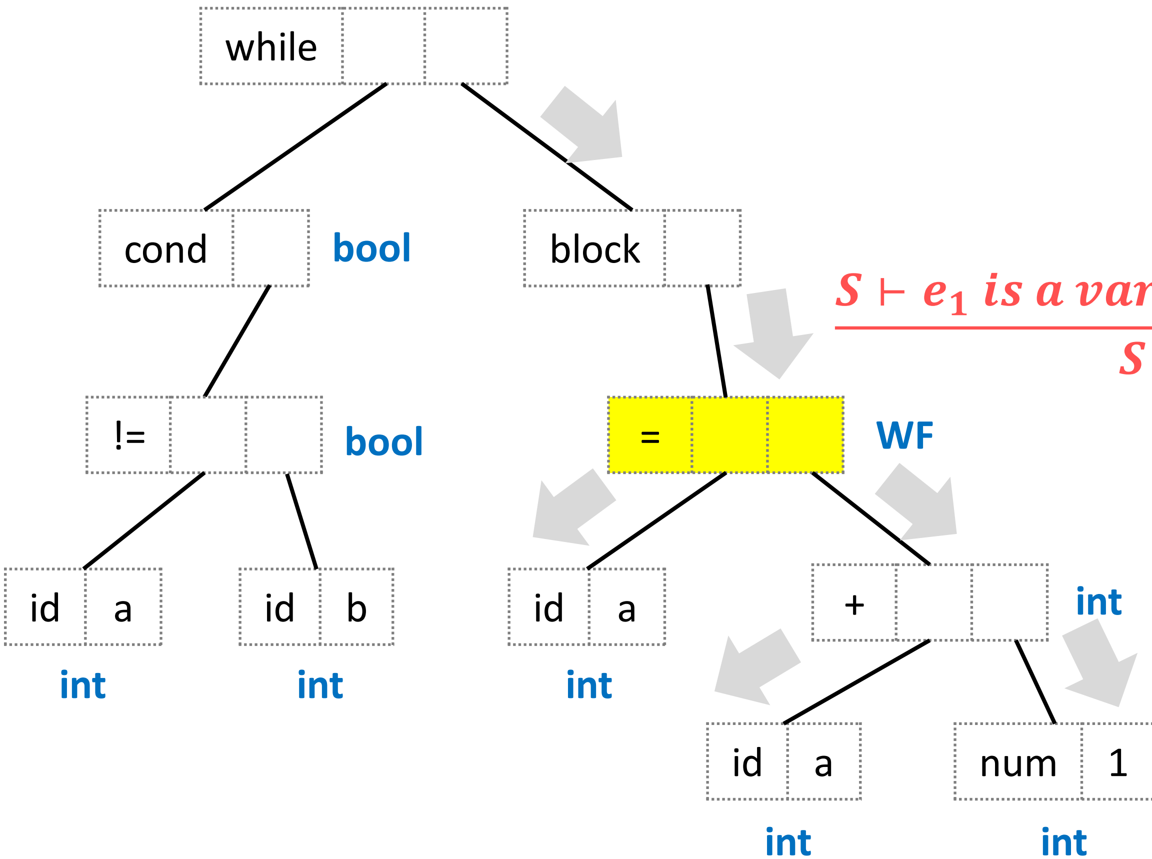
Implementation of type checking

Recursively walk abstract syntax trees

3. Do type checking for a child statement **a = a + 1**

Symbol table after scope checking



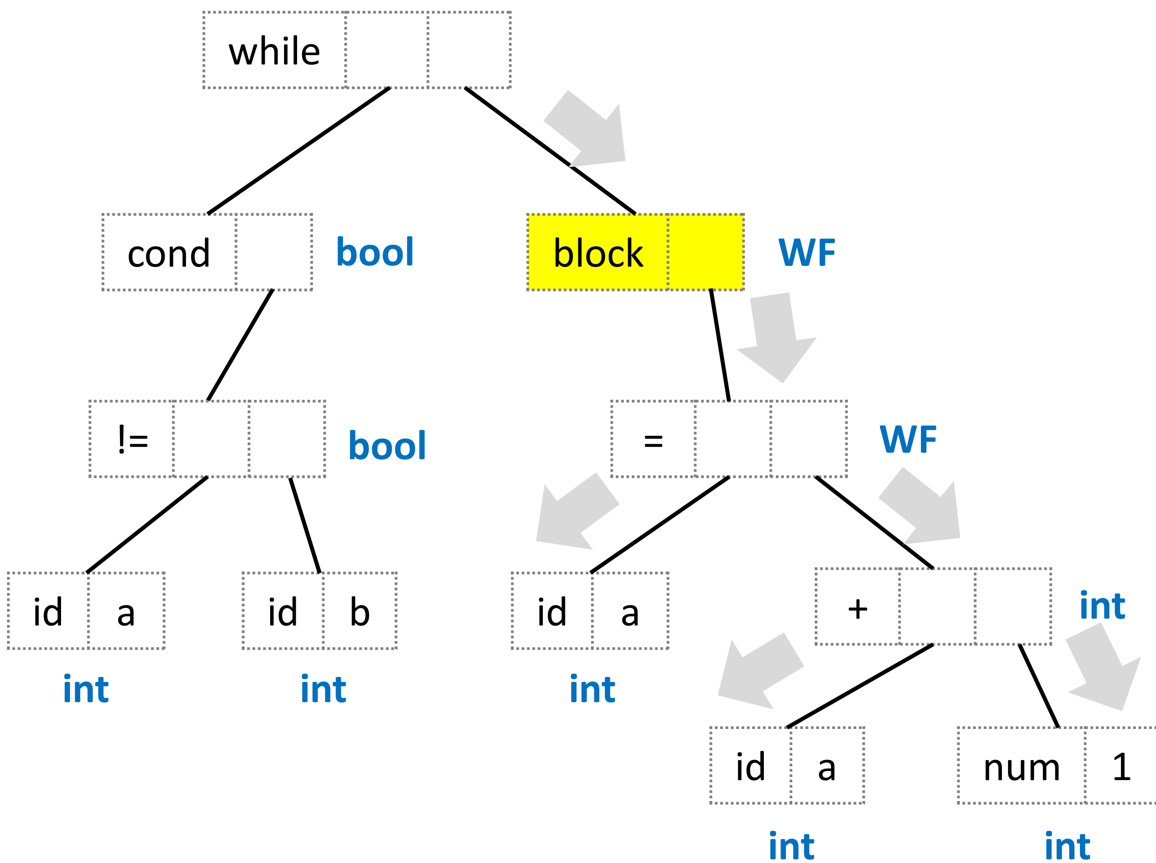


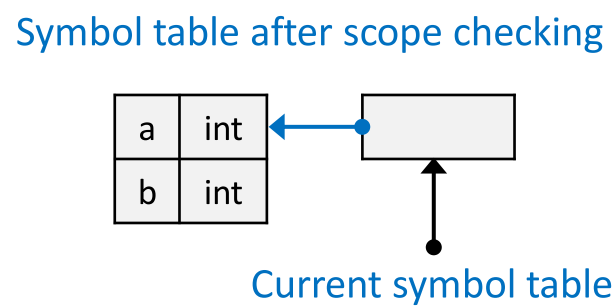
$$\frac{S \vdash e_1\ is\ a\ variable \qquad S \vdash e_1 : T \qquad S \vdash e_2 : T}{S \vdash WF(e_1 = e_2)}$$

Implementation of type checking

Recursively walk abstract syntax trees

3. Do type checking for a child statement **a = a + 1**

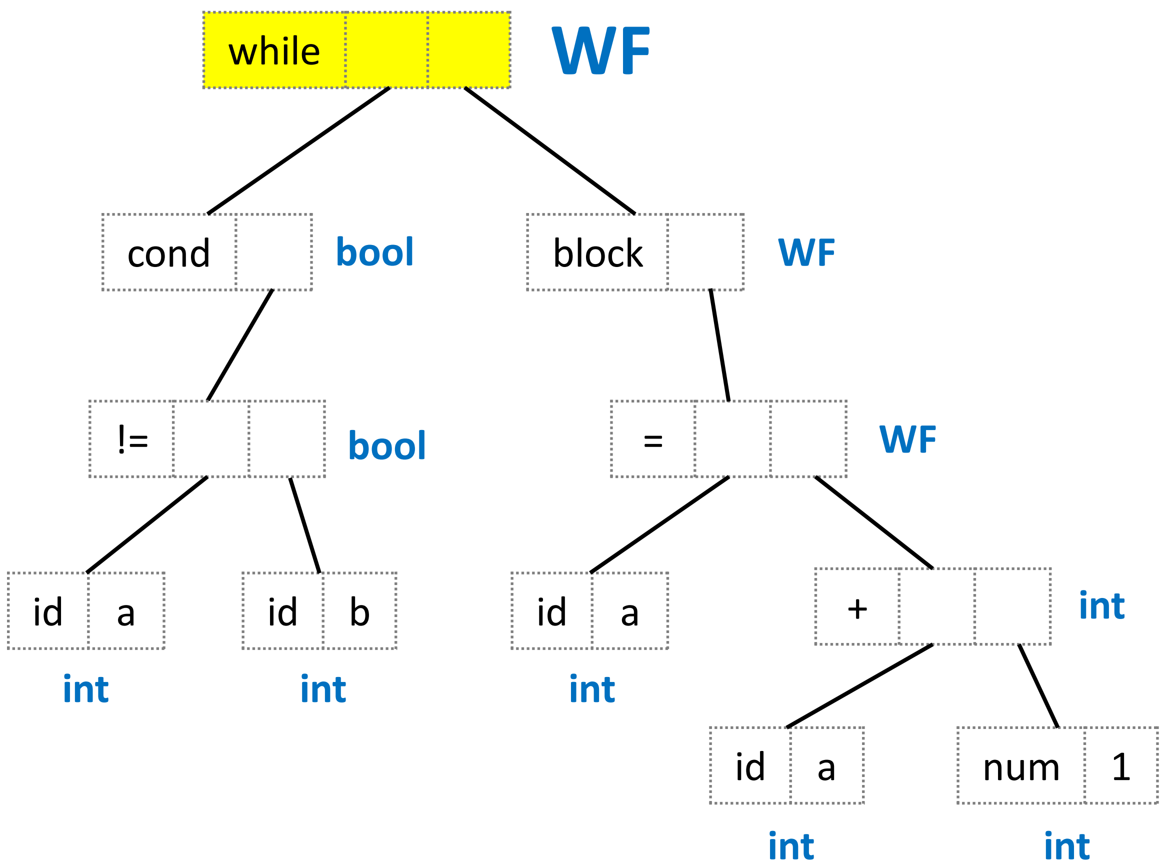




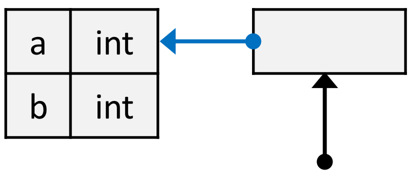
Implementation of type checking

Recursively walk abstract syntax trees

4. Check the overall well-formedness



Symbol table after scope checking



Current symbol table

Implementation of type checking

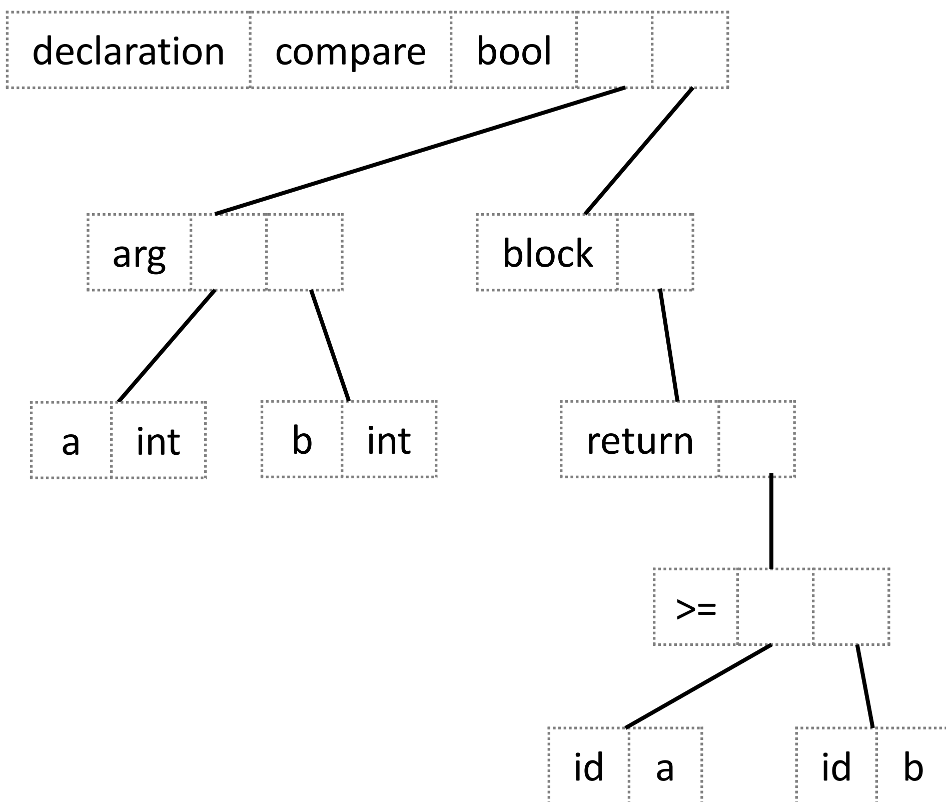
Another example: Let's check whether a return statement is well-formed or not

bool compare (int a, int b) {

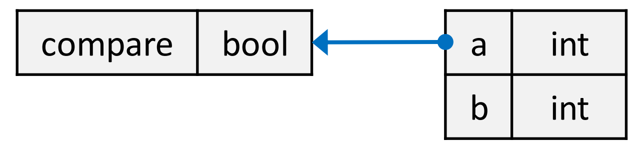
  return a >= b;

}

AST



Symbol table



Summary: type checking

Ensuring that the types of the operands match the type expected by the operator

In statically-typed languages,

Programmers declare types for all identifiers statically

- Types are associated with specific keywords (e.g., int, float, bool, char, …)

Compilers

1) Infer the type of each expression from the types of its components
2) Confirm that the types of expressions matches what is expected

Especially, by using the rules of inference + scope information + well-formedness rules

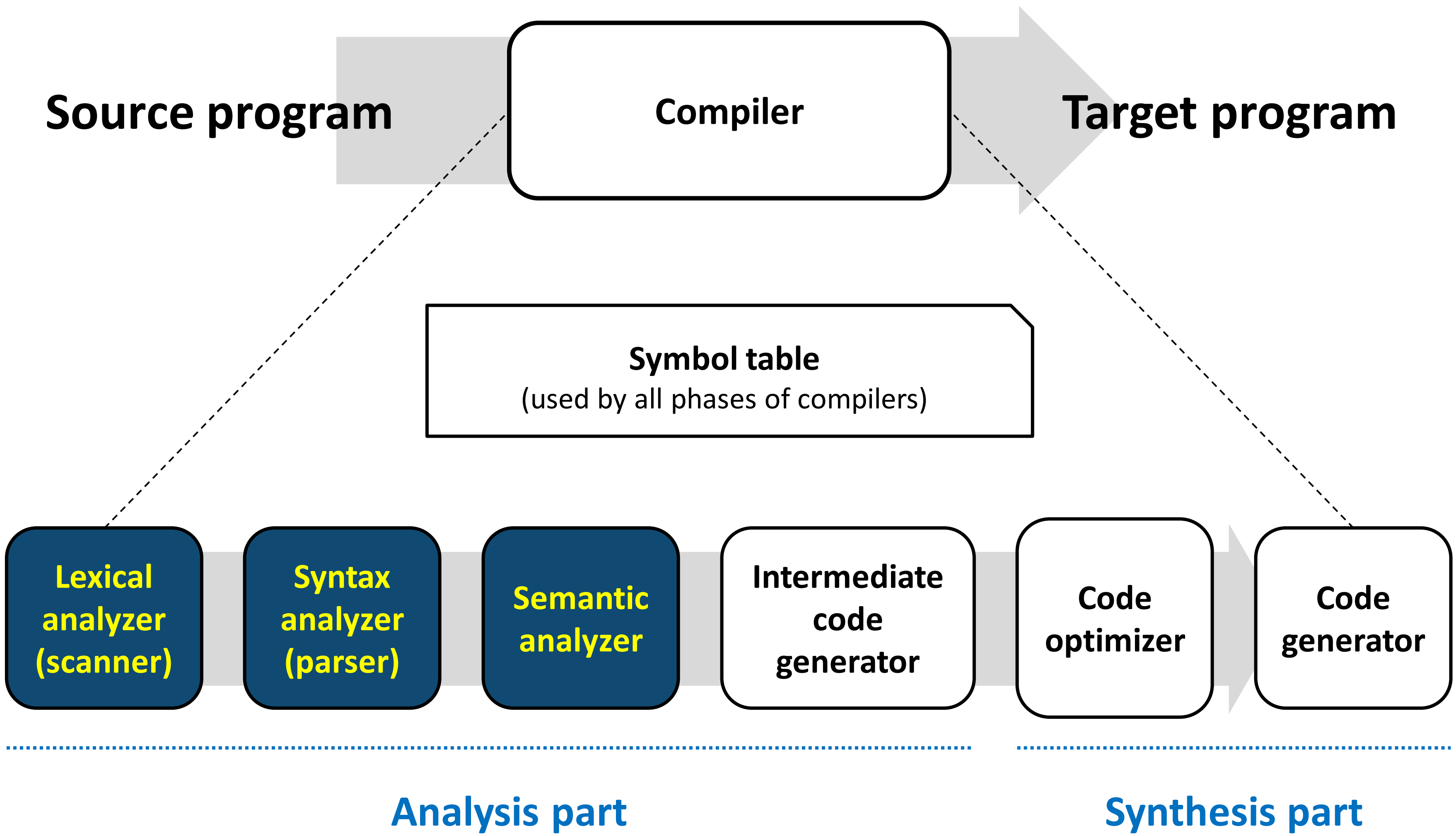
Summary: type checking

How to use the inference rules for type checking???

- First, before doing type checking, do scope checking.

- For each statement:
  - Do type checking for any subexpressions it contains
  - Do type checking for child statements
  - Check the overall well-formedness

So far...


Source program
Compiler
Target program
Symbol table
(used by all phases of compilers)
Lexical analyzer (scanner)
Syntax analyzer (parser)
Semantic analyzer
Intermediate code generator
Code optimizer
Code generator
Analysis part
Synthesis part