NTNU – Trondheim
Norwegian University of
Science and Technology

Type checking

Where we left off

- We have introduced inference rules
  - And connected them to syntax tree traversal
- We have talked about instantiating inference rules for a simple ternary expression
  - And how it relates to type checking
- We'll continue now with
  - Rules for type checking some different types of statements
  - Connection to syntax tree traversal
  - Static vs. dynamic type checking

Axioms

- Some statements don't need any premises in order to determine their type

$$\frac{}{\text{env} \mid\!- \text{true} : \text{bool}}$$

reads that "true" is a boolean value in any environment,

similiarly,

$$\frac{}{\text{env} \mid\!- 42 : \text{int}}$$

doesn't depend on the environment either

Declarations

- These affect the environment, that's what they're for

$$\frac{\text{env} \mid\!- E : T \qquad\qquad \text{env}\,[id : T] \mid\!- (S2\ ;\ S3\ ;\ \ldots\ ;\ Sn) : T'}{\text{env} \mid\!- id : T = E\ ;\ (S2\ ;\ S3\ ;\ \ldots\ ;\ Sn) : T'}$$

Assignments

- Identifiers

$$\frac{env\ [id : T]\ |\!-\ E : T}{env\ [id : T]\ |\!-\ id = E : T}$$

- Arrays

$$\frac{env\ |\!-\ E1 : array(T) \qquad env\ |\!-\ E2 : int \qquad env\ |\!-\ E3 : T}{env\ |\!-\ E1[E2] = E3 : T}$$

An abbreviation

- There is, implicitly, always an environment containing the context of the statement
- We don't always need to refer to any part of it, so

$$\frac{env \mid\!- E1 : array(T) \qquad env \mid\!- E2 : int \qquad env \mid\!- E3 : T}{env \mid\!- E1[E2] = E3 : T}$$

might as well be written

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

without loss of information.

- When there *is* something to say about the env. contents,

$$\frac{env\ [id : T] \mid\!- E : T}{env\ [id : T] \mid\!- id = E : T}$$

might as well just highlight the part we need, *i.e.*

$$\frac{id : T \mid\!- E : T}{id : T \mid\!- id = E : T}$$

Expressions

- We looked a little bit at these already;

$$\frac{E1 : int \qquad E2 : int}{E1 + E2 : int}$$

specifies that a sum of ints is an int,

$$\frac{E1 : int \qquad E2 : long}{E1 + E2 : long}$$

suggests that adding promotes int to long

(or we could write

$$\frac{E1 : T1 \qquad E2 : T2}{E1 + E2 : lub(T1,T2)}$$

← ("lub" = "least upper bound")

and specify a partial order of types...)

Whiles and sequences

$$\frac{E : bool \qquad S : T}{while(E)\ S : void}$$

$$\frac{S1 : T1 \qquad S2;\ S3;\ S4;\ \ldots;\ Sn : T'}{S1;\ S2;\ S3;\ S4;\ \ldots;\ Sn : T'}$$

Function calls

- The type of a function can be written as the (Cartesian) product of its argument types, and its return type:

  $$T1 \times T2 \times T3 \times \ldots \times Tn \rightarrow Tr$$

- Syntax-wise, calls are a case of expressions

$$\frac{E : T1 \times T2 \times T3 \times \ldots \times Tn \rightarrow Tr \qquad E1{:}T1 \qquad E2{:}T2 \quad \ldots}{E\ (\ E1,\ E2,\ E3,\ \ldots,\ En\ )\ :\ Tr}$$

NTNU – Trondheim
Norwegian University of Science and Technology

Function declarations

- Suppose a declaration consists of a return type and a name,

  Tr id

  a list of parameters,

  (T1 p1, T2 p2, …, Tn pn)

  and a body which evaluates to something,

  { E; }

  for a grand total of

  Tr id (T1 p1, T2 p2, …, Tn pn) { E; }
- What we want is to check E in an environment where all the parameters have their declared types, so put them in there, and expect E to check out as the return type

Function declarations

$$\frac{p1{:}T1,\ p2{:}T2,\ \ldots,\ pn{:}Tn\ \vdash E : Tr}{\vdash Tr\ id\ (\ T1\ p1,\ T2\ p2,\ \ldots,\ Tn\ pn\ )\ \{\ E;\ \}\ :\ void}$$

- Somewhere inside E, a return statement must resolve to the return type Tr
  - How to check it? Return values don't appear in the local environment of the function...

Return statements

- Use a placeholder in the environment
- If we introduce a "magic" variable ret with the return type

$$\frac{p1:T1,\ p2:T2,\ \ldots,\ pn:Tn,\ ret : Tr \ \vdash E : Tr}{\vdash Tr\ id\ (\ T1\ p1,\ T2\ p2,\ \ldots,\ Tn\ pn\ )\ \{\ E;\ \} : void}$$

return statements can be checked as

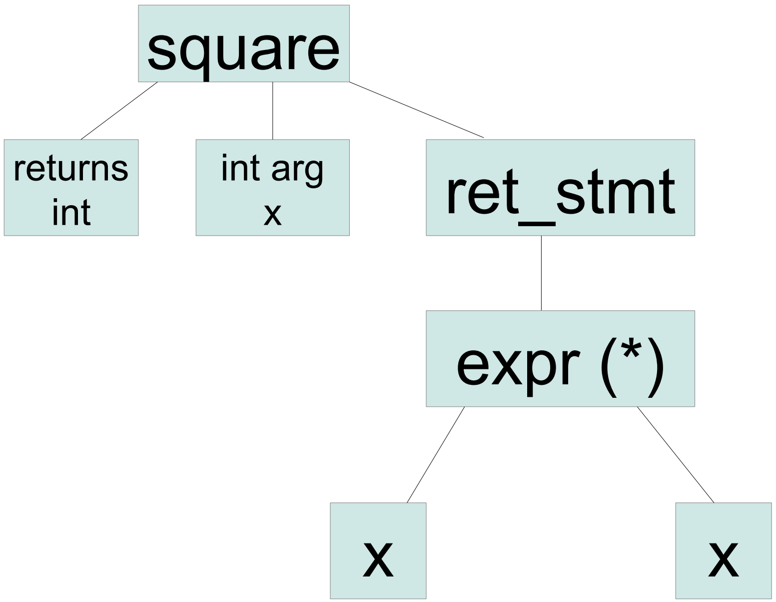
$$\frac{ret : T \ \vdash E : T}{ret : T \ \vdash return\ E : void}$$

NTNU – Trondheim
Norwegian University of Science and Technology

What a type-check must do

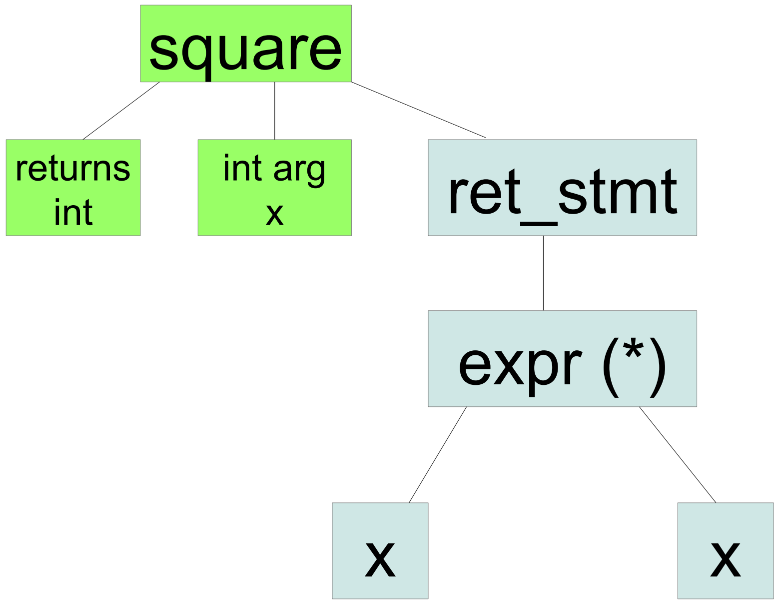
Let's define a function:

int square (int x) { return (x*x); }



What a type-check must do

Enter the function in a global symbol table

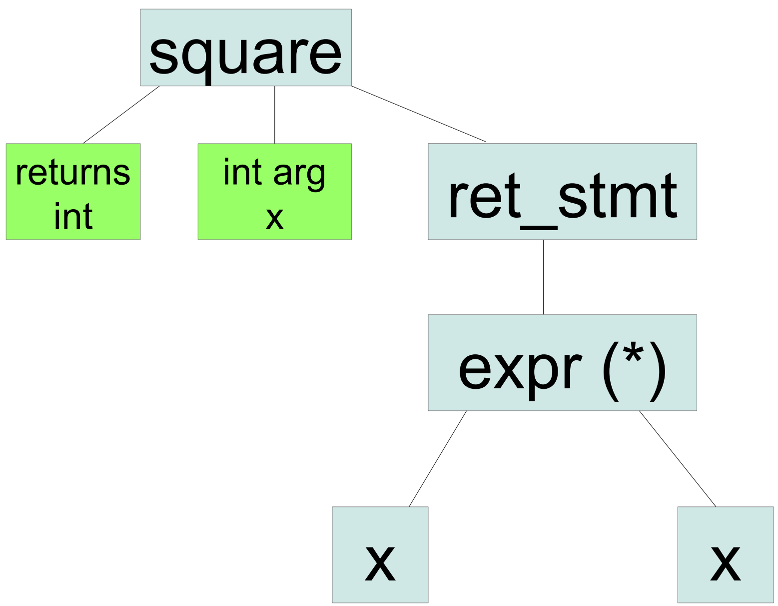


Global symbols

Name	Type	...
Square	function, int → int	

What a type-check must do

Create a local context (either in the global table, or make another)



Global symbols

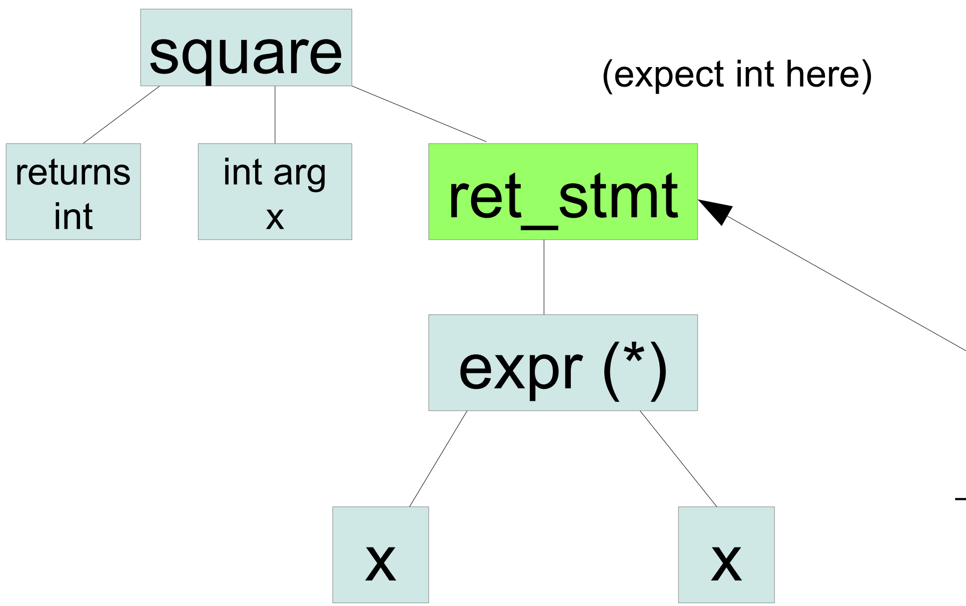
Name	Type	...
Square	function, int → int	

Square's symbols

Name	Type	...
x	int	
ret	int	

What a type-check must do

Check statements in the function body



Global symbols

Name	Type	...
Square	function, int → int	

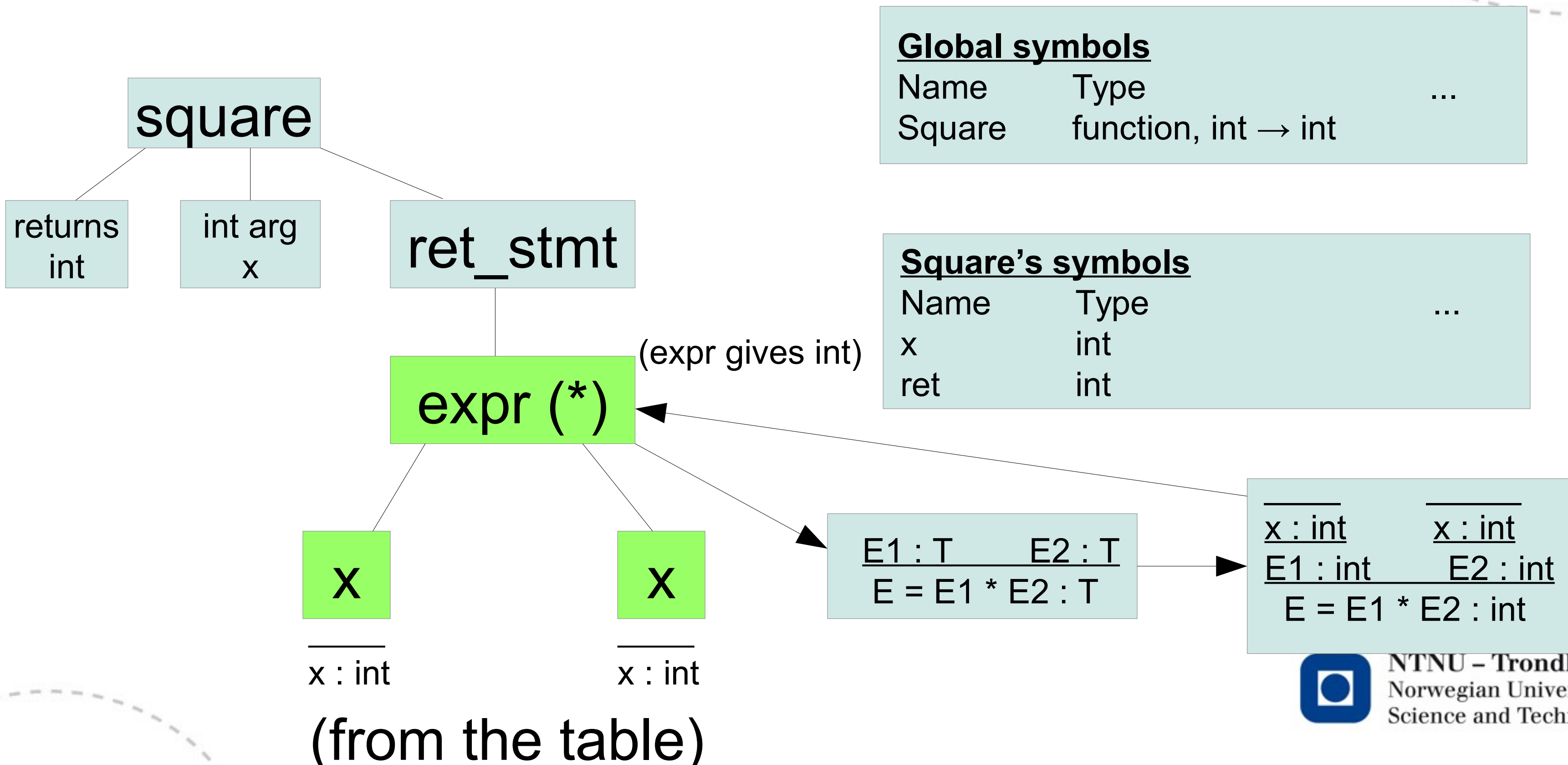
Square's symbols

Name	Type	...
x	int	
ret	int	

$$\frac{ret : int \qquad E : T}{ret : int \mid\!- \; return\ E : void}$$

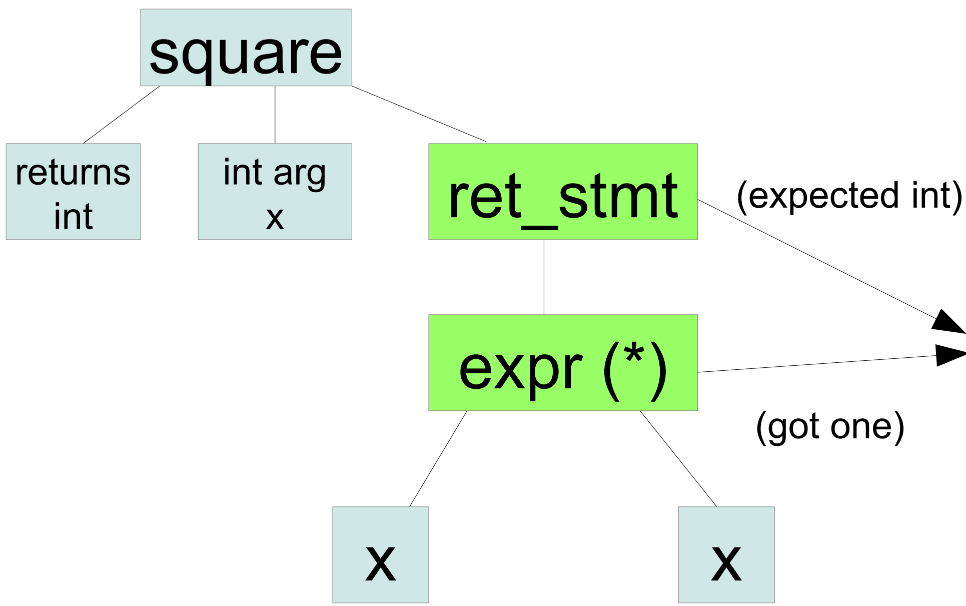
What a type-check must do

Check each part of each statement



What a type-check must do

Check each part of each statement



Global symbols

Name	Type	...
Square	function, int → int	

Square's symbols

Name	Type	...
x	int	
ret	int	

(proof on prev slide)

$$\frac{ret : int \qquad E : int}{ret : int \vdash return\ E : void}$$

Hooray, 'square' is correctly typed

Three views on checking

- Implementation-wise, we traverse the syntax tree and enforce the rules of the type system
- If the rules allow us to do that simultaneously with discovering the syntax tree, it fits a syntax-directed translation scheme *a la* Dragon
    - *i.e. graft checking into the semantic actions of the parser*
- Written as inference rules, it is a construction of a proof tree which resolves a bunch of type judgments

- All the same thing, more or less

What we've looked at is *static*

- All information about types and values comes straight from the source code
  - That's why we can do it by examining the syntax tree
  - When the compiler is finished, so is the type checking
- It's a process of *binding*
  - Explicitly, as with "double z = 2.71828" (declaration says it)
  - Implicitly, as with "z = 3.141593" (value gives it away)

  and *checking*
  - If z is consistently used as a double in the scope of this binding, the program is *type-safe*
- *Type-safety* is the lack of type errors when the program runs

NTNU – Trondheim
Norwegian University of
Science and Technology

How safe is static checking?

- That depends on how it's implemented.
- C lets you lie to the type checker, under the assumption that you have control
- That includes creating type errors at run time

```
% cat square.c
double square ( double x ) { return x*x; }
% cat main.c
#include <stdio.h>
int square ( int );
int main() { printf ( "%d\n", square(64) ); }
% cc -o test main.c square.c
% ./test
4195622
%
```

How safe is static checking?

- Java won't have such shenanigans, and enforces more safety
- Both check statically, but according to different rules

```
% cat Square.java
public class Square { public static double square(double x) { return x*x; } }
% cat Main.java
public class Main {
    public static void main (String args[]) {
        System.out.println ( Square.square(64) );
    }
}
% javac Main.java Square.java
% java Main
4096.0
%
```

Dynamic types

- Other languages permit type information to appear at run time, and check it then
  - Scheme, Ruby, Python
- These are interpreted, but nothing prevents a compiler from inserting dynamic type checks into the program it generates
- Some even give you static types when you declare variables, and dynamic when you don't
  - *Dylan* pioneered this in 1995
  - C# does it today

The strength of a type system

- Strongly typed languages guarantee that programs are type-safe if they pass checking
- Weakly typed languages admit programs that contain type errors
- A *sound* type system statically ensures that all programs are type-safe

  (Sound as in *soundness*, it doesn't make any noise)

Strength is a design trade-off

- A program may be safe for reasons a compiler cannot detect:

```
% cat unsafe.c
#include <stdio.h>
#include <stdint.h>
int main () {
    double hello = 1.81630607015975e-310;
    puts ( (char *)&hello );
}
% make unsafe
cc     unsafe.c   -o unsafe
% ./unsafe
Hello!
%
```

- This won't fail, but it doesn't type-check without forced casting either

NTNU – Trondheim
Norwegian University of
Science and Technology

These words are not absolutes

- We saw that static checks in Java are less permissive than those in C
  - Taken as a whole, Java types also have a dynamic twist to them
  - Objects remember what type they are at run time, that's why you can get ClassCastExceptions instead of wrong answers
- Python does all its checking dynamically, and is pretty firm about consistency (stronger)

```
>>> a = 42
>>> b = “42”
>>> print a == b                  # No number is a string
False
```

- PHP also works dynamically, but has a more liberal philosophy (weaker)

```
php > $a = 42;
php > $b = “42”;
php > var_dump ( $a == $b );      # Sure, why not?
bool(true)
```

Pros and cons of static types

(+) *Speeeeeeeed…*

Dynamic checking runs whenever the program does, and takes time

(+) Evergreen analysis

- Generated result does the same thing every time it runs
- Dynamic types admit dynamic type errors

(-) Has to be conservative

- Can't defer check until values are known, must assume they can be anything
- Stronger checking translates into accepting fewer programs

Next up

More elaborate, derived types

- Arrays
- Records
- Objects