Compiler Project

Semantic Analysis

Florent De Geeter & Pascal Fontaine

University of Liège

March 19, 2024

Outline

The Assignment

Semantics of VSOP

Practical Advice

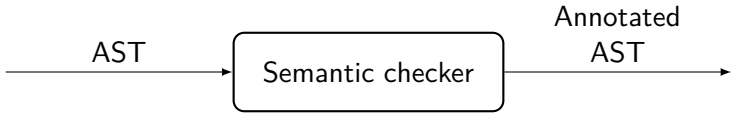
Outline

The Assignment

Semantics of VSOP

Practical Advice

Semantic Analysis



Converts bare AST into an **annotated (typed) AST**

Detects **semantically invalid** source code

`vsopc -c example.vsop` should:

- If the source file is semantically valid, return 0 and print on `stdout` the annoted abstract syntax tree following the format given in the statement
- Otherwise, return a non-zero value and print on `stderr` (at least) one semantic, syntax or lexical error

Assignment

Due **April 17, 2024**

Automated tests worth 5% of your grade

No external tool should be used for this part

Support for **custom tests** in `tests` subfolder

Three modes: `-c`, `-p` and `-l`

Output Format

Similar to the syntax analysis one, but the AST will be **annotated with types** for all expressions

For example:

```
[Class(C, Object,
  [Field(a_string, string, "C" : string)],
  [Method(a_fun, [i : int32], unit,
    [Call(self : C, printInt32, [i : int32]) : Object,
     () : unit] : unit)]),
 Class(Main, Object, [],
       [Method(main, [], int32, [0 : int32] : int32)])]
```

You can (and should) **keep other information** in your *annotated* AST, but only types will be printed

Error Management

Error messages on `stderr`, fail with code $\neq 0$

```
input_file.vsop:4:12: semantic error: description
```

Exact error positions are up to you (examples coming)

During semantic analysis, it is easy to give **insightful** error messages

You can do **error recovery** using a *dummy* type

Lexical and syntax errors must of course still be supported

Outline

The Assignment

Semantics of VSOP

Practical Advice

Semantics of VSOP is Given in the Manual

Scoping rules.

- Where is a given variable, field or method accessible ?
- Where does an identifier points to ?

Typing rules.

- Are expressions of the right types for the operations ?
- Are method calls consistent with their declarations ?

Evaluation rules.

- What do VSOP expressions mean ?
- How is VSOP code executed ?

VSOP is Statically Typed

The compiler checks types at **compile time**: no type error can occur at runtime

Every expression is assigned a type, according to developer annotations and VSOP typing rules

Contrast this to dynamic languages like python, where

```
def f():
    return 40 + "two"
```

would crash at runtime, when (and only if) `f()` is actually called:

```
TypeError: unsupported operand types for +:
    'int' and 'str'
```

VSOP is Object-Oriented, with (Single) Inheritance

When

```
class Child extends Parent { ... }
```

`Child` **inherits** all fields and methods of `Parent` in addition to its own

The type of `Child` **conforms to** the type of `Parent`:

- i.e. a `Child` *instance* can be used everywhere a `Parent` one can
- A `Child` **is a** `Parent`.
- `Child` **is a subtype of** `Parent`.

In VSOP, it is illegal to **redefine** parent's fields, but you can **override** methods (with same *signature*)

Method Dispatch is Checked with Declared Type of Object

```
class C extends P { onlyInC() : int32 { ... } }
                    // New method not in P

let o : P <-  // Declared/Static type is P
    if inputInt32() = 0 // Take int32 typed by user
        then new C  // `new C` valid here as C <: P
                    // Actual/Dynamic type is C
        else new P  // Actual/Dynamic type is P
in o.onlyInC(); // Type error! Static type is P
    // Would be valid if the user typed 0, but we
    // cannot tell at compile time

let o : P <- new C
in o.onlyInC(); // Type error: declared type still P
    // Always valid, but forbidden nonetheless
```

VSOP Allows Dynamic Dispatch

```
class P { name() : string { "P" } }
class C extends P { name() : string { "C" } }

let o : P <- new C
in {
    o.name() // Will return "C", not "P"
}
```

Manipulating an identifier with the type of the parent class allows to only use the methods declared in the parent class (**statically typed**)

But if the object pointed by this identifier has the type of the child class, the method that will be called is the child's one (**dynamic dispatch**)

The Unit Type

The type `unit`, of which only inhabitant is `()`, is an **alternative** to `void` in other languages (it is **similar to** `NoneType` in Python)

It is not specific to VSOP, but appears in **many functional languages**, which are often **expression-based** (e.g. Haskell, OCaml, SML, ... but also Kotlin, Rust, Swift, etc.)

As VSOP is **statically typed**, you don't need to actually represent `()` (for code generation): if something has type `unit`, then it can only have value `()`

Why is it called `unit`? See *algebraic data types* (ADT) if curious

Conditionals

In a conditional of the form

`if <expr_cond> then <expr_then> else <expr_else>`

the condition `<expr_cond>` must be of `bool` type

The types of both branches must **agree**:

- If **both** branches have **class type**, the types **agree** and the resulting type is the class of the **first common ancestor**
- If **both** branches have the **same primitive type**, the types **agree**
- If **a** branch has **type** `unit`, the types **agree** and the resulting type of the conditional is `unit`
- Else, it is a **typing error**

Outline

The Assignment

Semantics of VSOP

Practical Advice

Error Reporting

Semantic errors can usually be reported at **different locations**

There exist **two** common approaches for error reporting:

- **Bottom-up** error reporting: first **check** the *children* expressions and **obtain** their type, then check if their type **agree** with their expected type (if there is one)
- **Top-down** error reporting: first **infer** the expected type of the *children* expressions (if possible), then **check** the *children* expressions by **giving** them their expected type

Bottom-up error reporting example

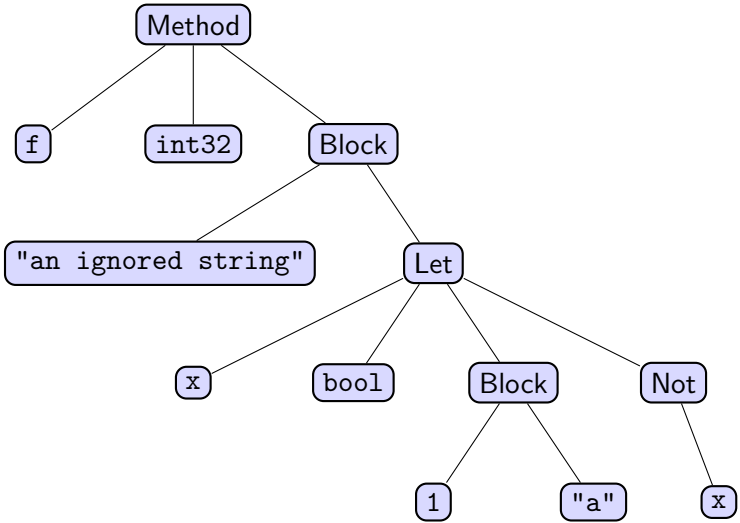
```
class C {
    f() : int32 {
        "an ignored string";
        let x : bool <- { 1; "a" }
        in not x // Last block expression
    }
}
```

```
input.vsop:4:25: semantic error:
  expected type bool, but found type string.
input.vsop:2:17: semantic error:
  expected type int32, but found type bool.
```

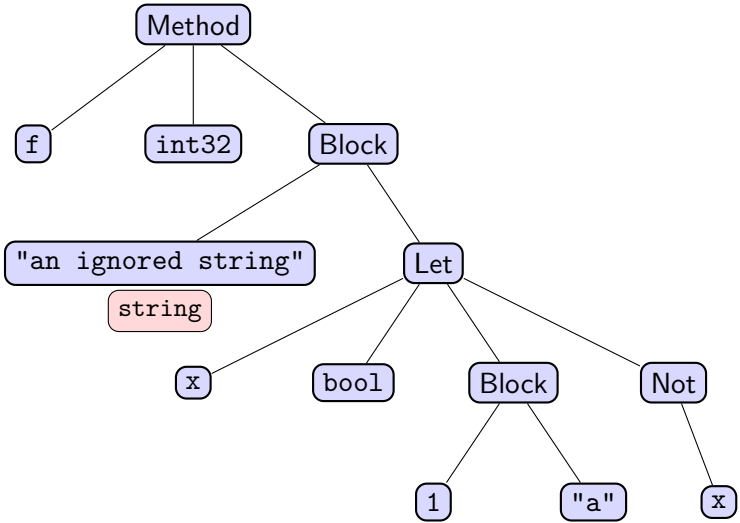
First message complains that the type of `{ 1; "a"}` is not `bool`

Second message complains that the whole method body should be of type `int32`

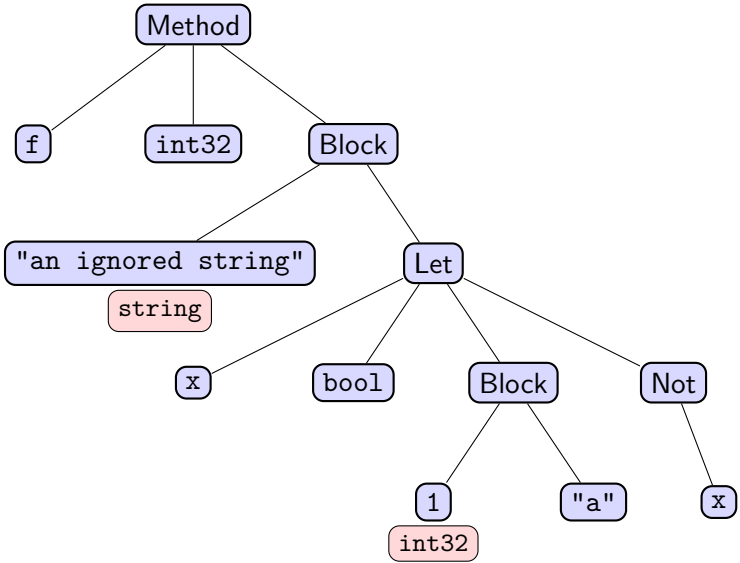
Bottom-up error reporting example



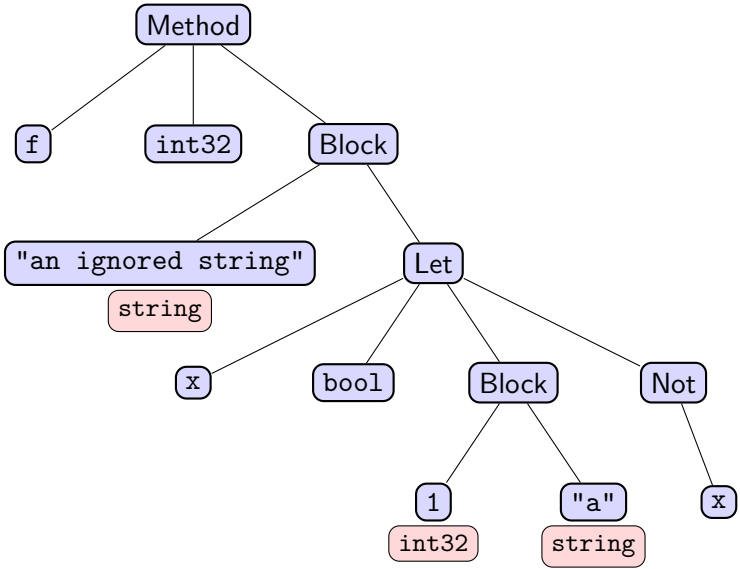
Bottom-up error reporting example



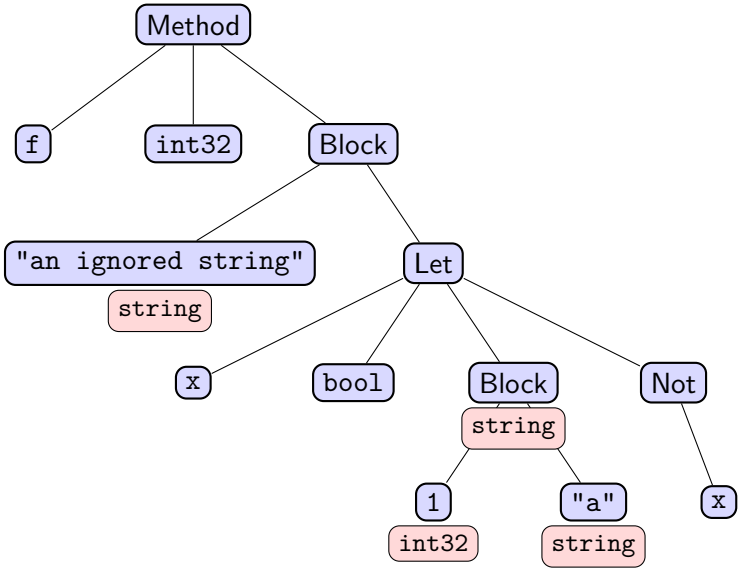
Bottom-up error reporting example



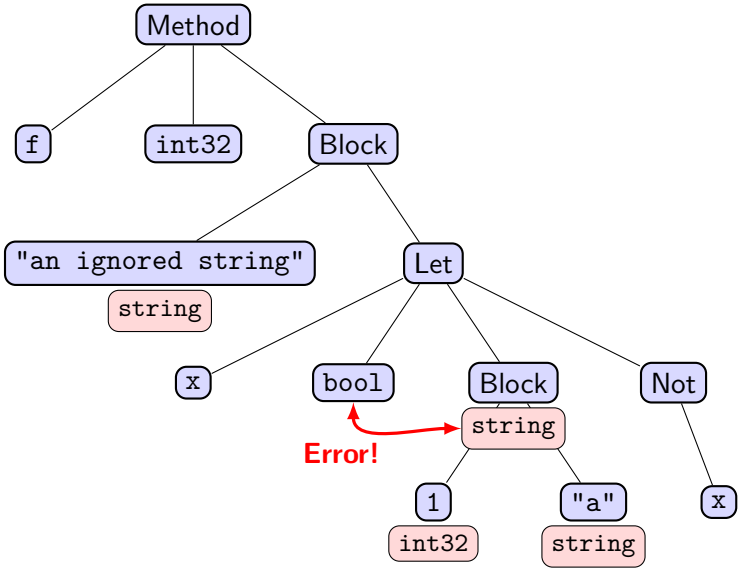
Bottom-up error reporting example



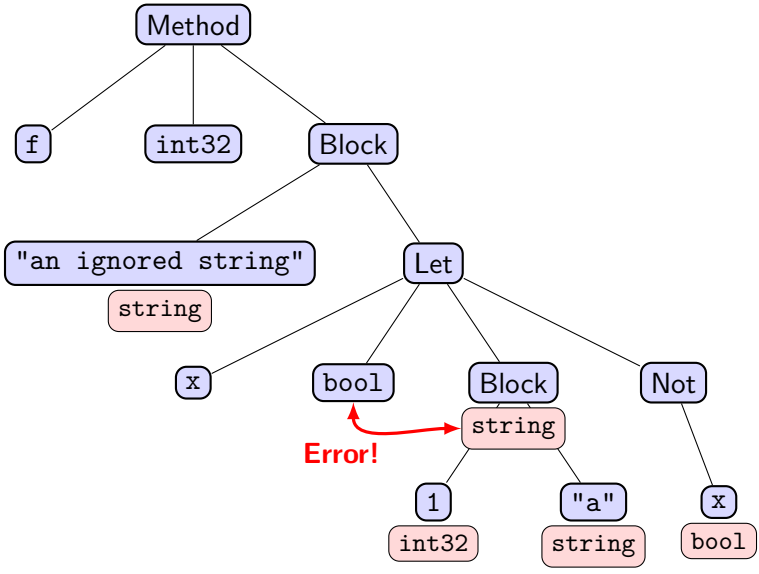
Bottom-up error reporting example



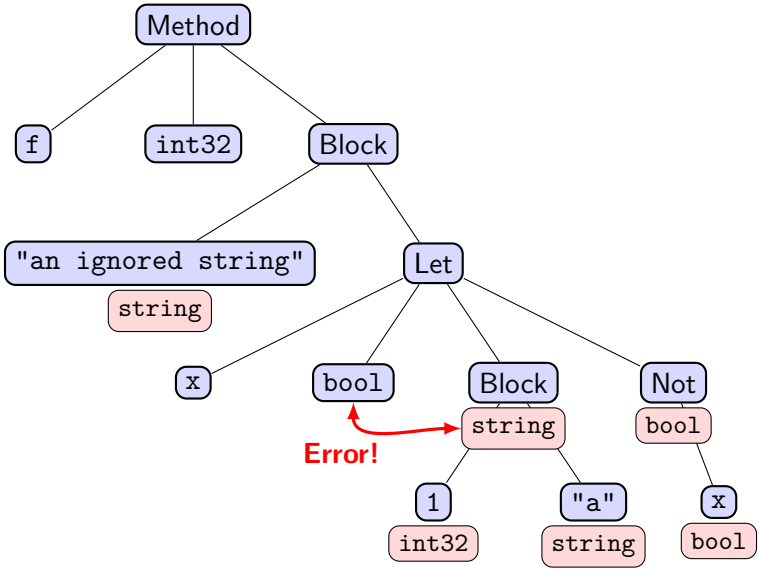
Bottom-up error reporting example



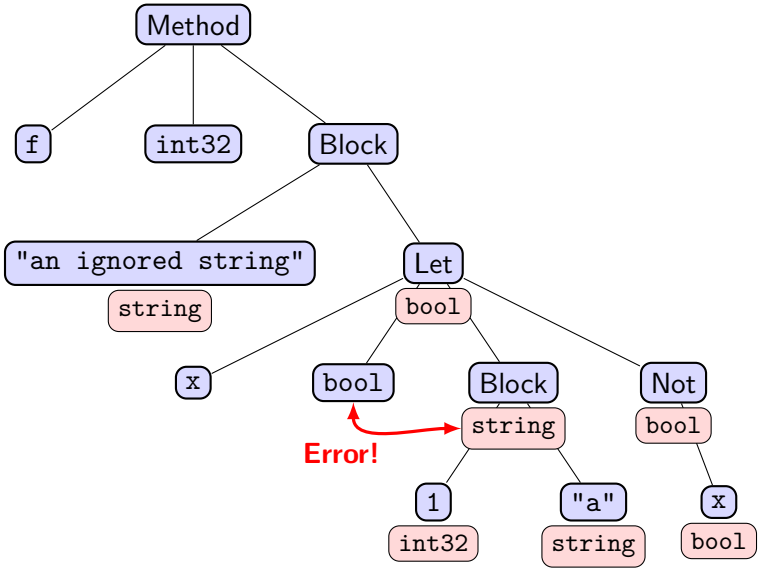
Bottom-up error reporting example



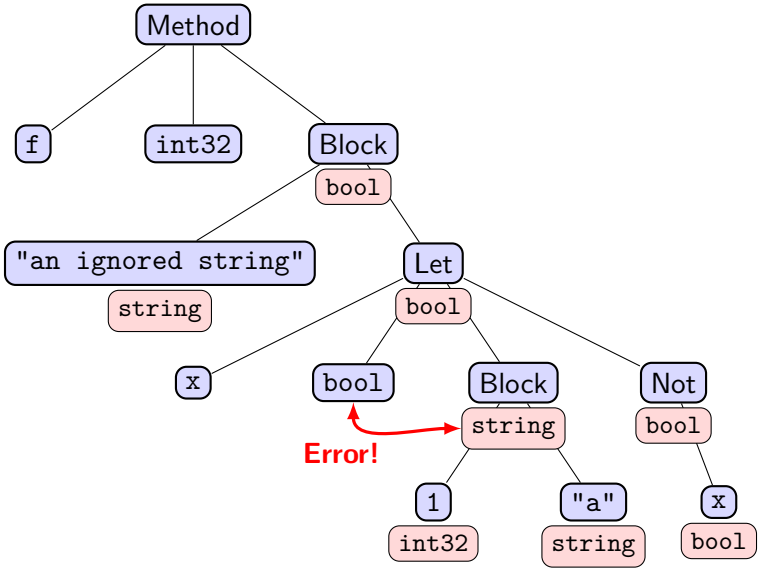
Bottom-up error reporting example



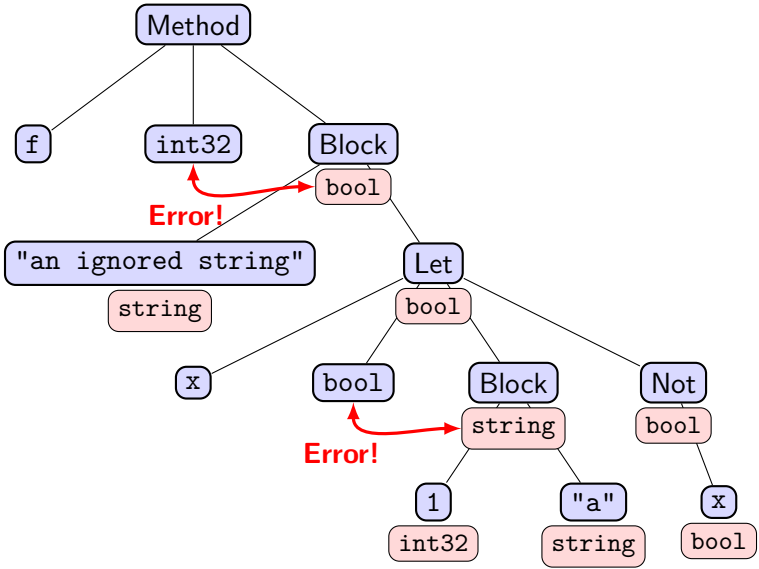
Bottom-up error reporting example



Bottom-up error reporting example



Bottom-up error reporting example



Top-down error reporting example

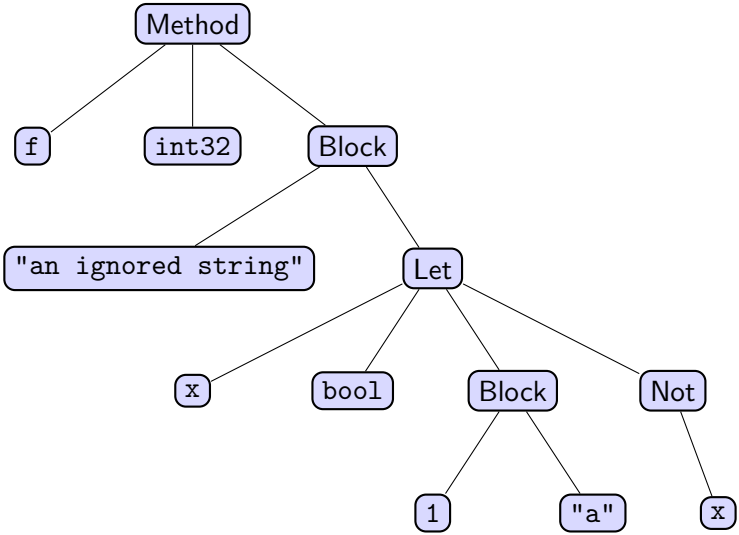
```
class C {
    f() : int32 {
        "an ignored string";
        let x : bool <- { 1; "a" }
        in not x // Last block expression
    }
}
```

```
input.vsop:4:30: semantic error:
  expected type bool, but found type string.
input.vsop:5:12: semantic error:
  expected type int32, but found type bool.
```

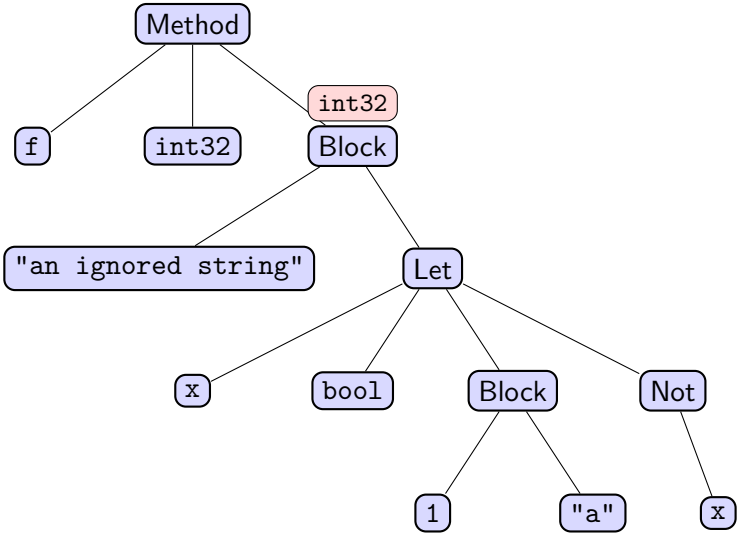
First message reports the position of `"a"`

Second message reports the position of `not x`

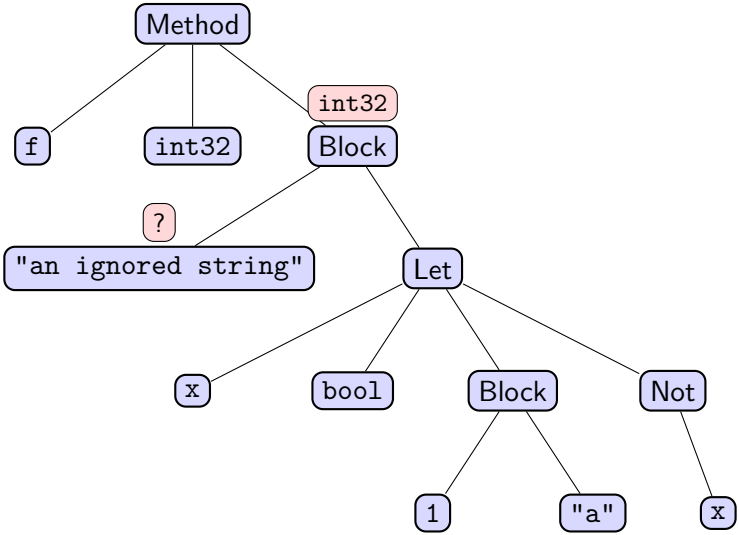
Top-down error reporting example



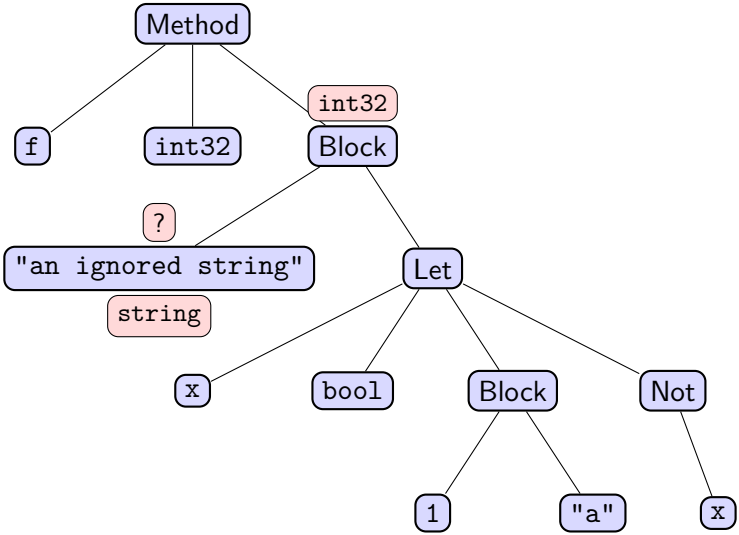
Top-down error reporting example



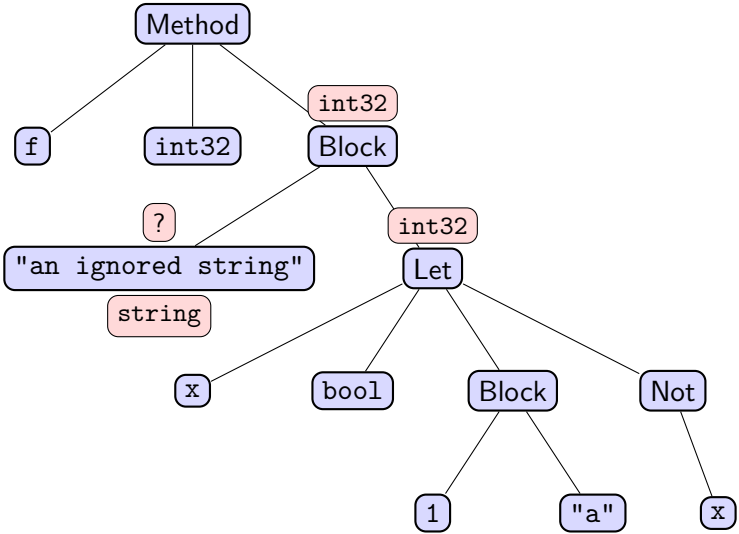
Top-down error reporting example



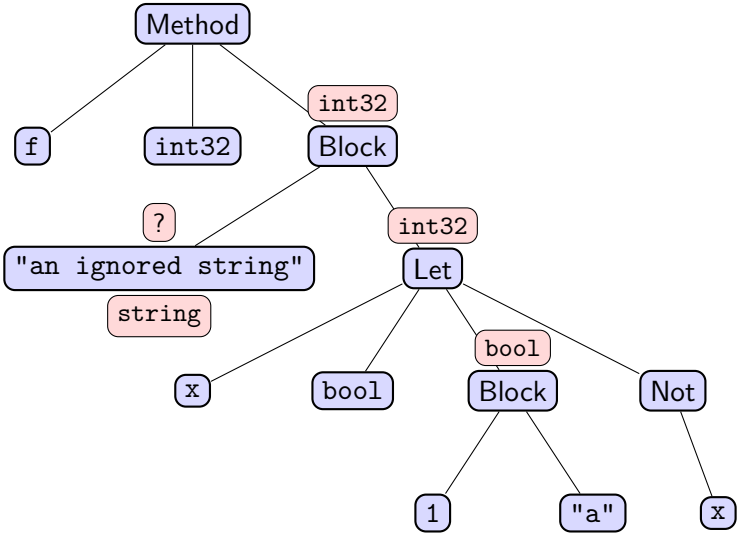
Top-down error reporting example



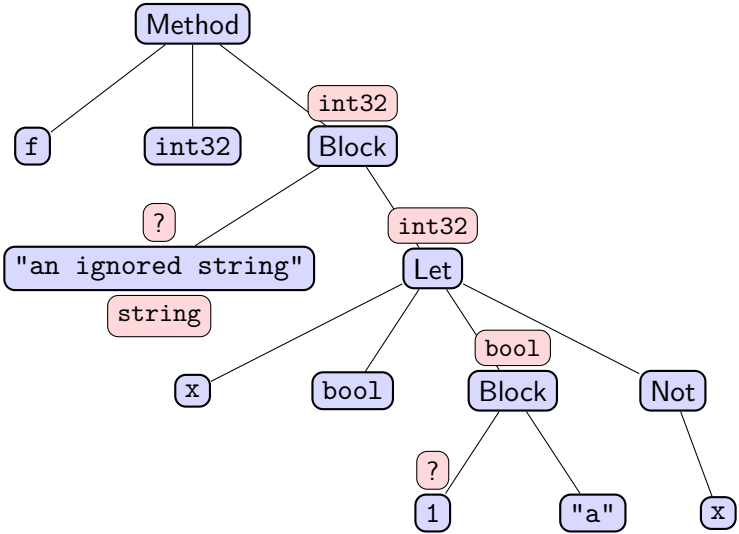
Top-down error reporting example



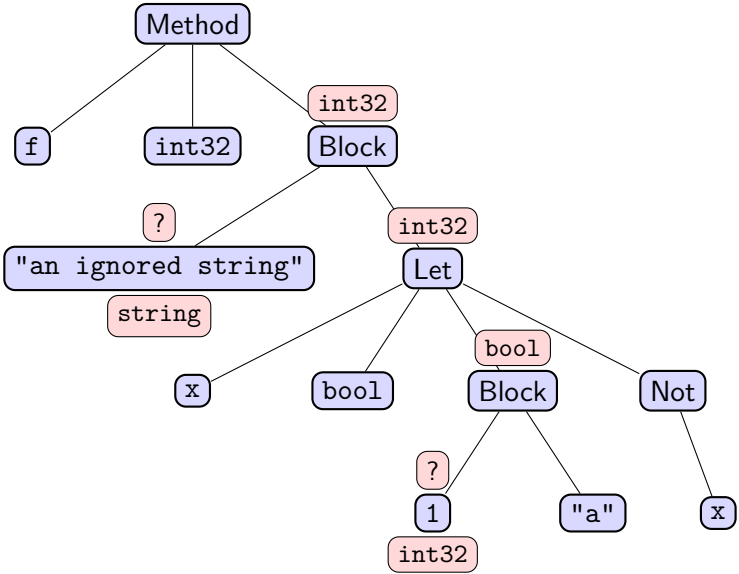
Top-down error reporting example



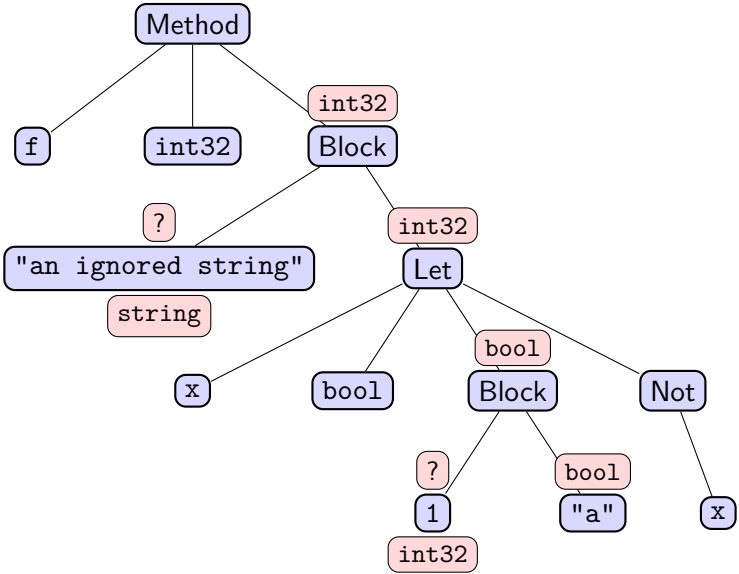
Top-down error reporting example



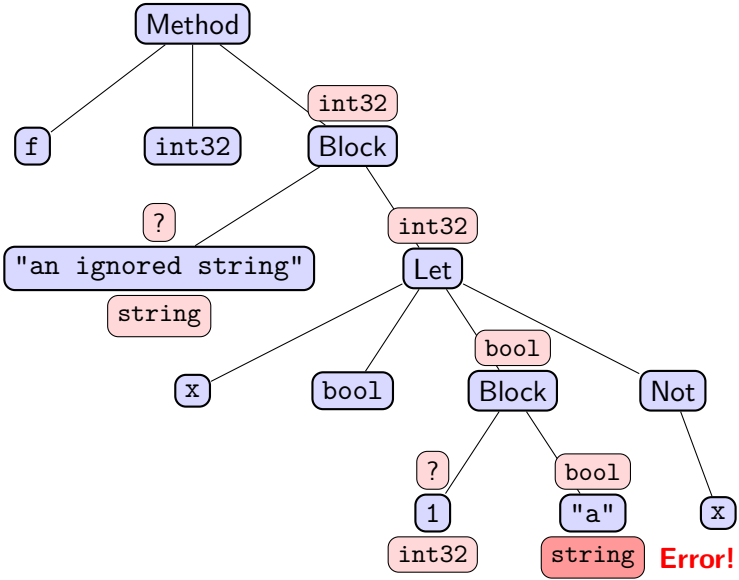
Top-down error reporting example



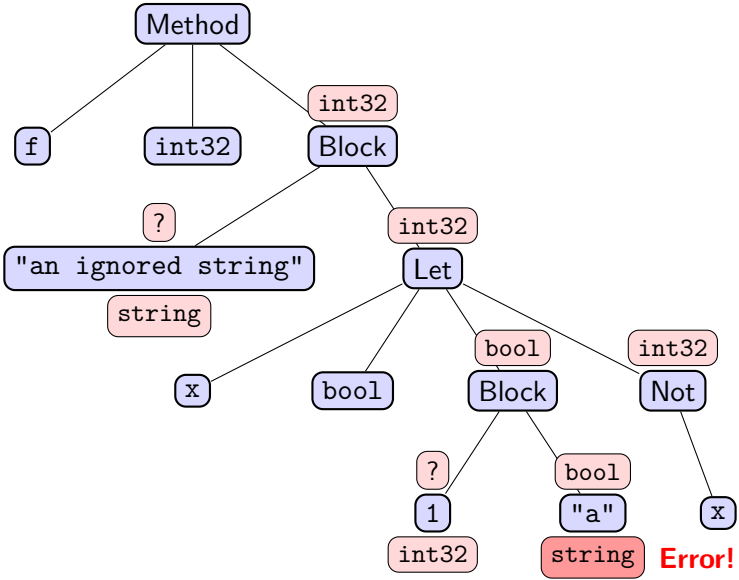
Top-down error reporting example



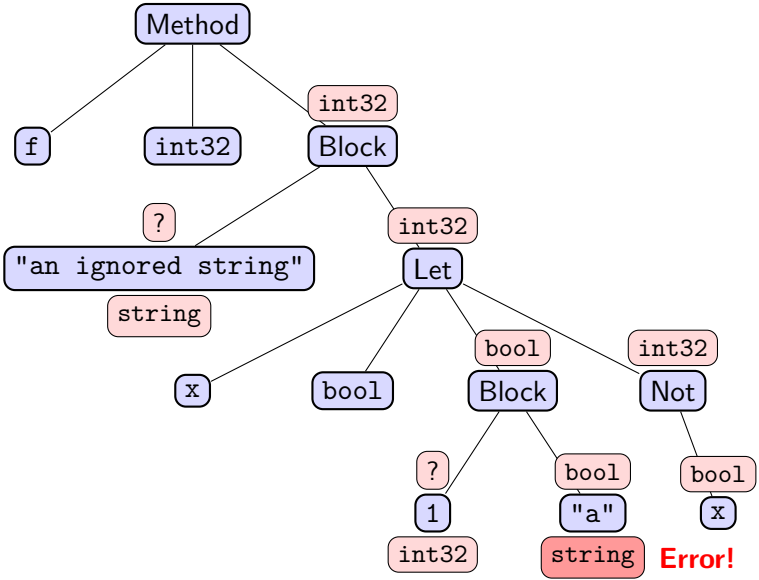
Top-down error reporting example



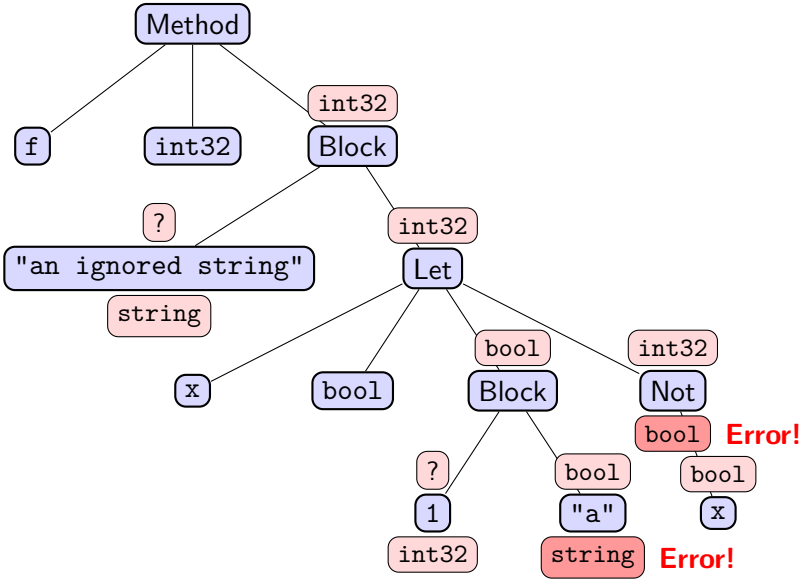
Top-down error reporting example



Top-down error reporting example



Top-down error reporting example



Error reporting for VSOP

With VSOP, **top-down** error reporting usually yields **better error messages**, but it cannot be used all the time

E.g., when a VSOP block contains several instructions, it is not possible to infer an expected type for the instructions, except the last one

Nonetheless, you are **free to choose** the error reporting approach you want, as long as you give coherent error messages

Make multiple passes

Order of declarations does not matter in VSOP, and it is hard to analyze a VSOP program **in one pass**:

```
class C extends P { ... }
// many other classes ...
class P { ... }
```

On line 1, is `P` defined? Is it a child of class `C` ? You need information from the whole file

Similar issue with method and fields declarations

You could try to be smart (e.g. queue checks to be done), but it is not worth it: keep it simple and do **as many passes as needed**

Questions and (Possibly) Answers