KU | Fall 2018 | Drew Davidson

EECS 665 COMPILER CONSTRUCTION

21 – Typechecking

Administrivia

Current
Assignments:
H6
P3

- Happy Fall Break's Eve!
- Please fill out the mid-semester evaluation when it comes out tonight
  - Your feedback helps me a <u>LOT</u>
  - Your voice does make a difference

Lecture Outline

- Last time
  - Pseudo-formalized types (type systems)
  - Discussed typing options
- This time
  - Type Checking
  - A little SDT review

What is Type Checking?

- Type analysis
  - Assigning types to expressions
- Type checking
  - Ensure that the type of a construct matches what's expected by its context

Reasons for Type Checking

- Generate appropriate code for operations

  A + B

    - String concatenation? Integer addition? Floating-point addition?

- Catch runtime errors / security issues
  - Make sure type assignments are sensible
  - Augment type system with additional refinements (liquid types)

Research on Types

- A huge topic in and of itself
  - Some CS Departments have a “PLT” focus: “Programming Languages and Types”

Liquid Types *

Patrick M. Rondon   Ming Kawaguchi   Ranjit Jhala
University of California, San Diego
{prondon,mwookawa,jhala}@cs.ucsd.edu

Abstract

We present *Logically Qualified Data Types*, abbreviated to *Liquid Types*, a system that combines *Hindley-Milner* type inference with *Predicate Abstraction* to automatically infer dependent types precise enough to prove a variety of safety properties. Liquid types allow programmers to reap many of the benefits of dependent types, namely static verification of critical properties and the elimination of expensive run-time checks, without the heavy price of manual annotation. We have implemented liquid type inference in DSOLVE, which takes as input an OCAML program and a set of logical qualifiers and infers dependent types for the expressions in the OCAML program. To demonstrate the utility of our approach, we describe experiments using DSOLVE to statically verify the safety of array accesses on a set of OCAML benchmarks that were previously annotated with dependent types as part of the DML project. We show that when used in conjunction with a fixed set of array bounds checking qualifiers, DSOLVE reduces the amount of manual annotation required for proving safety from 31% of program text to under 1%.

Categories and Subject Descriptors D.2.4 [*Software Engineering*]: Software/Program Verification; F.3.1 [*Logics and Meanings of Programs*]: Specifying and Verifying and Reasoning about Programs

General Terms Languages, Reliability, Verification

Keywords Dependent Types, Hindley-Milner, Predicate Abstraction, Type Inference

1. Introduction

Modern functional programming languages, like ML and Haskell, have many features that dramatically improve programmer productivity and software reliability. Two of the most significant are strong static typing, which detects a host of errors at compile-time, and type inference, which (almost) eliminates the burden of annotating the program with type information, thus delivering the benefits of strong static typing for free.

The utility of these type systems stems from their ability to predict, at compile-time, invariants about the run-time values computed by the program. Unfortunately, classical type systems only capture relatively coarse invariants. For example, the system can express the fact that a variable `i` is of the type `int`, meaning that it is always an integer, but not that it is always an integer within a certain range, say between 1 and 99. Thus, the type system is unable to statically ensure the safety of critical operations, such as a division by `i`, or the accessing of an array `a` of size 100 at an index `i`. Instead, the language can only provide a weaker dynamic safety guarantee at the additional cost of high performance overhead.

In an exciting development, several authors have proposed the use of *dependent types* [20] as a mechanism for enhancing the expressivity of type systems [14, 27, 2, 22, 10]. Such a system can express the fact

$$\mathtt{i} :: \{\nu{:}\mathtt{int} \mid 1 \leq \nu \wedge \nu \leq 99\}$$

which is the usual type `int` together with a *refinement* stating that the run-time value of `i` is an always an integer between 1 and 99. Pfenning and Xi devised DML, a practical way to integrate such types into ML, and demonstrated that they could be used to recover static guarantees about the safety of array accesses, while simultaneously making the program significantly faster by eliminating run-time checking overhead [27]. However, these benefits came at the price of automatic inference. In the DML benchmarks, about 31% of the code (or 17% by number of lines) is manual annotations that the typechecker needs to prove safety. We believe that this non-trivial annotation burden has hampered the adoption of dependent types despite their safety and performance benefits.

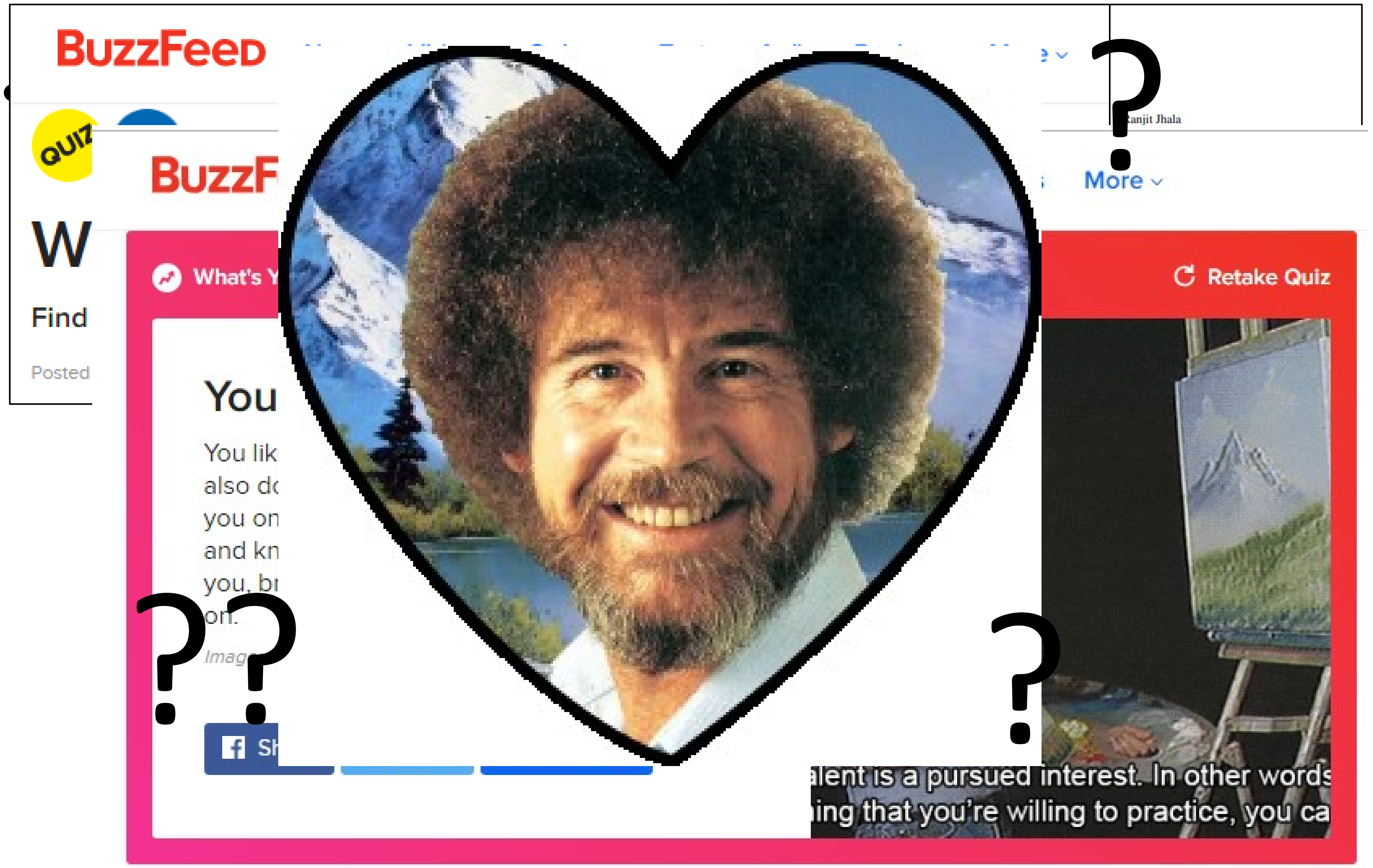
We present *Logically Qualified Data Types*, abbreviated to *Liquid Types*, a system for automatically inferring dependent types precise enough to prove a variety of safety properties, thereby allowing programmers to reap many of the benefits of dependent types without paying the heavy price of manual annotation. The heart of our inference algorithm is a technique for blending Hindley-Milner type inference with *predicate abstraction*, a technique for synthesizing loop invariants for imperative programs that forms the algorithmic core of several software model checkers [3, 16, 4, 29, 17]. Our system takes as input a program and a set of *logical qualifiers* which are simple boolean predicates over the program variables, a special *value variable* ν, and a special placeholder variable $\star$ that can be instantiated with program variables. The system then infers *liquid types*, which are dependent types where the refinement predicates are *conjunctions* of the logical qualifiers.

In our system, type checking and inference are decidable for three reasons (Section 3). First, we use a conservative but decidable notion of subtyping, where we reduce the subtyping of arbitrary dependent types to a set of implication checks over base types, each of which is deemed to hold if and only if an *embedding* of the implication into a decidable logic yields a valid formula in the logic. Second, an expression has a valid liquid type derivation only if it has a valid ML type derivation, and the dependent type

* This work was supported by NSF CAREER grant CCF-0644361, NSF PDOS grant CNS-0720802, NSF Collaborative grant CCF-0702603, and a gift from Microsoft Research.

159

Research on Types



Refinement Types

- A type enhanced with a predicate which must hold for any element of the type

$$f : \mathbb{N} \rightarrow \{ n : \mathbb{N} | \, n\%2 = 0 \}$$

- Could imagine enhancing a type system with annotations for all kinds of properties
  - Single-use variable
  - High security/low security (non-interference)

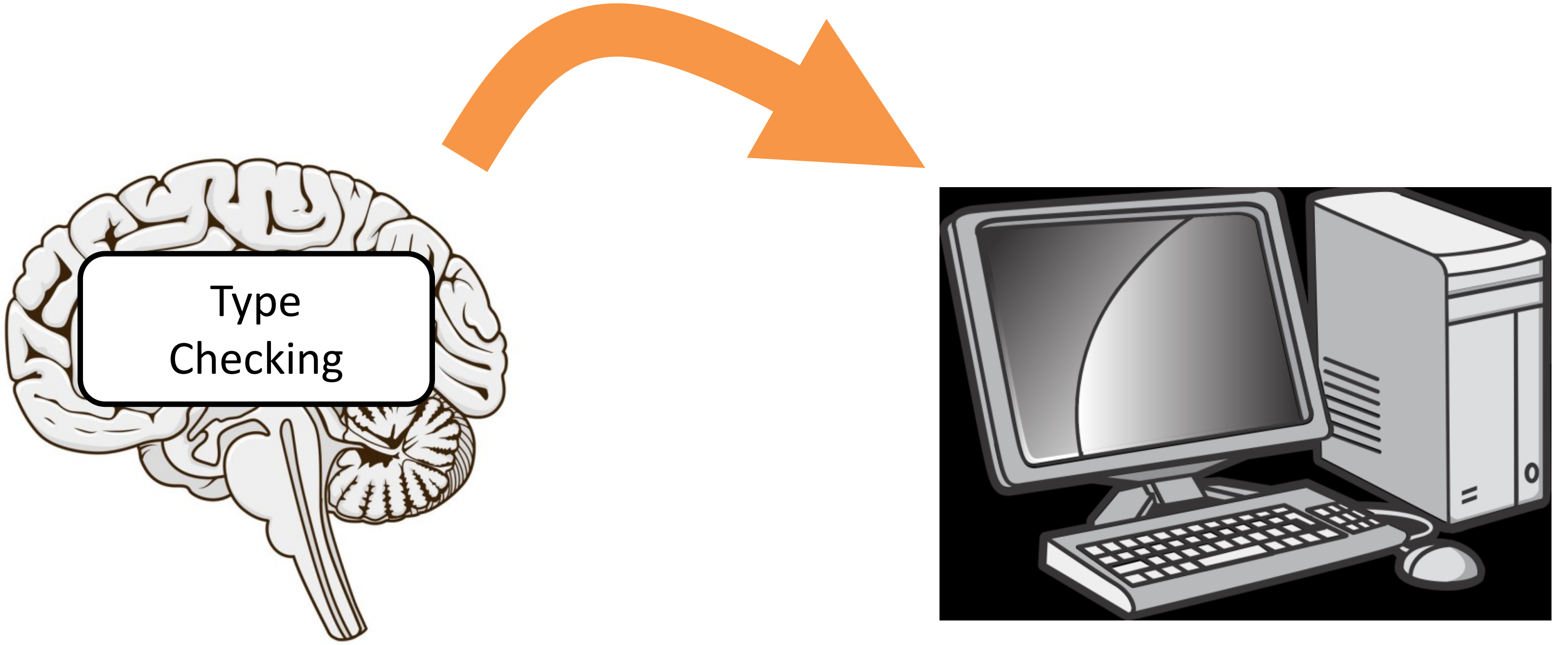
Examples of Refinement Types

- Could imagine enhancing a type system with annotations for all kinds of properties
  - Single-use variable
  - High security/low security (non-interference)

Piggybacking on Type Checking

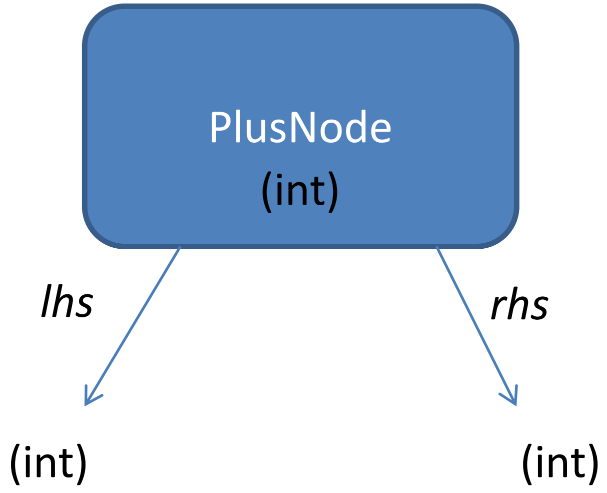
- Type checking is a good place to add some logic because:
  - Programmers are familiar with typing logic
  - The analysis is already well-formulated

Implementing Type Checking



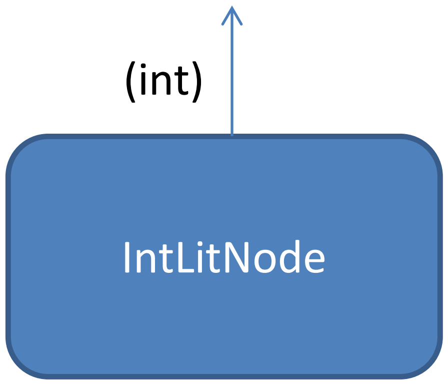
Type Checking: Binary Operator

- Get the type of the LHS
- Get the type of the RHS
- Check that the types are compatible for the operator
- Set the *kind* of the node be a value
- Set the *type* of the node to be the type of the operation's result



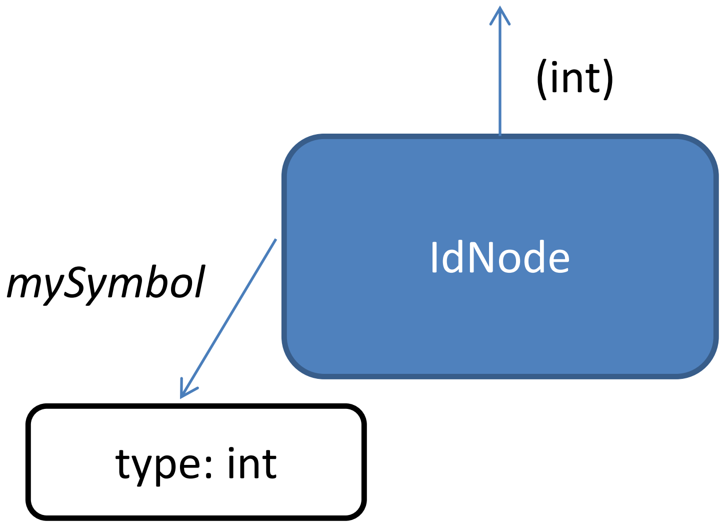
Type "Checking" Literals

- Cannot be wrong
  - Just pass the type of the literal up the tree



Type Checking: IdNode

- Look up the type of the declaration
  - There should be a symbol “linked” to the node
- Pass symbol type up the tree



Type Checking: Others

- Other node types follow these same principles
  - Function calls
    - Get type of each actual argument
      - Match against the formal argument (check symbol)
    - Send the return type up the tree
  - Statement
    - No type

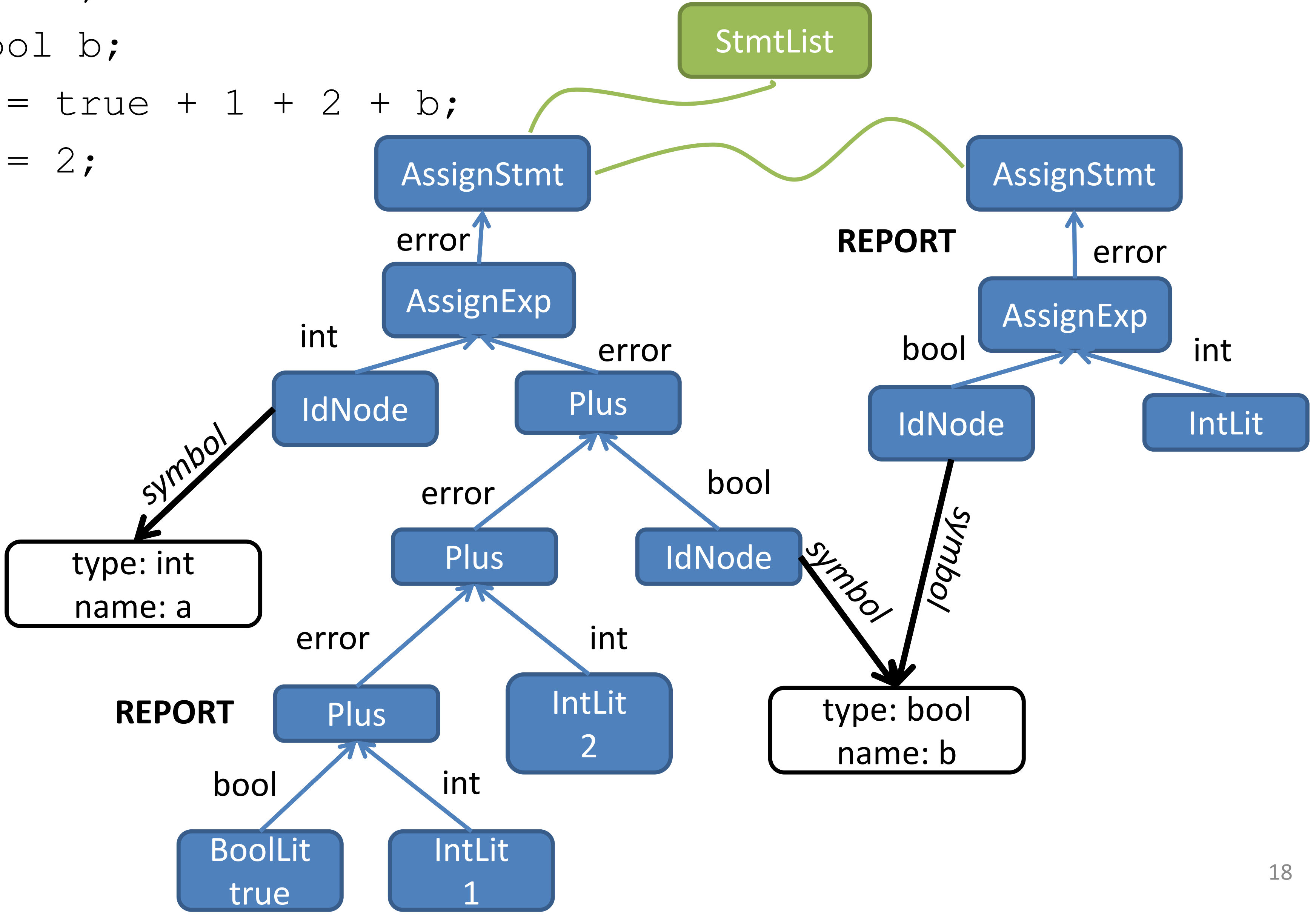
Type Checking: Error Reports

- We'd like all *distinct* errors at the same time
  - Don't give up at the first error
  - Don't report the same error multiple times
- Introduce an internal **error** type
  - When type incompatibility is discovered
    - Report the error
    - Pass **error** up the tree
  - When you get error as an operand
    - Don't (re)report an error
    - Again, pass **error** up the tree

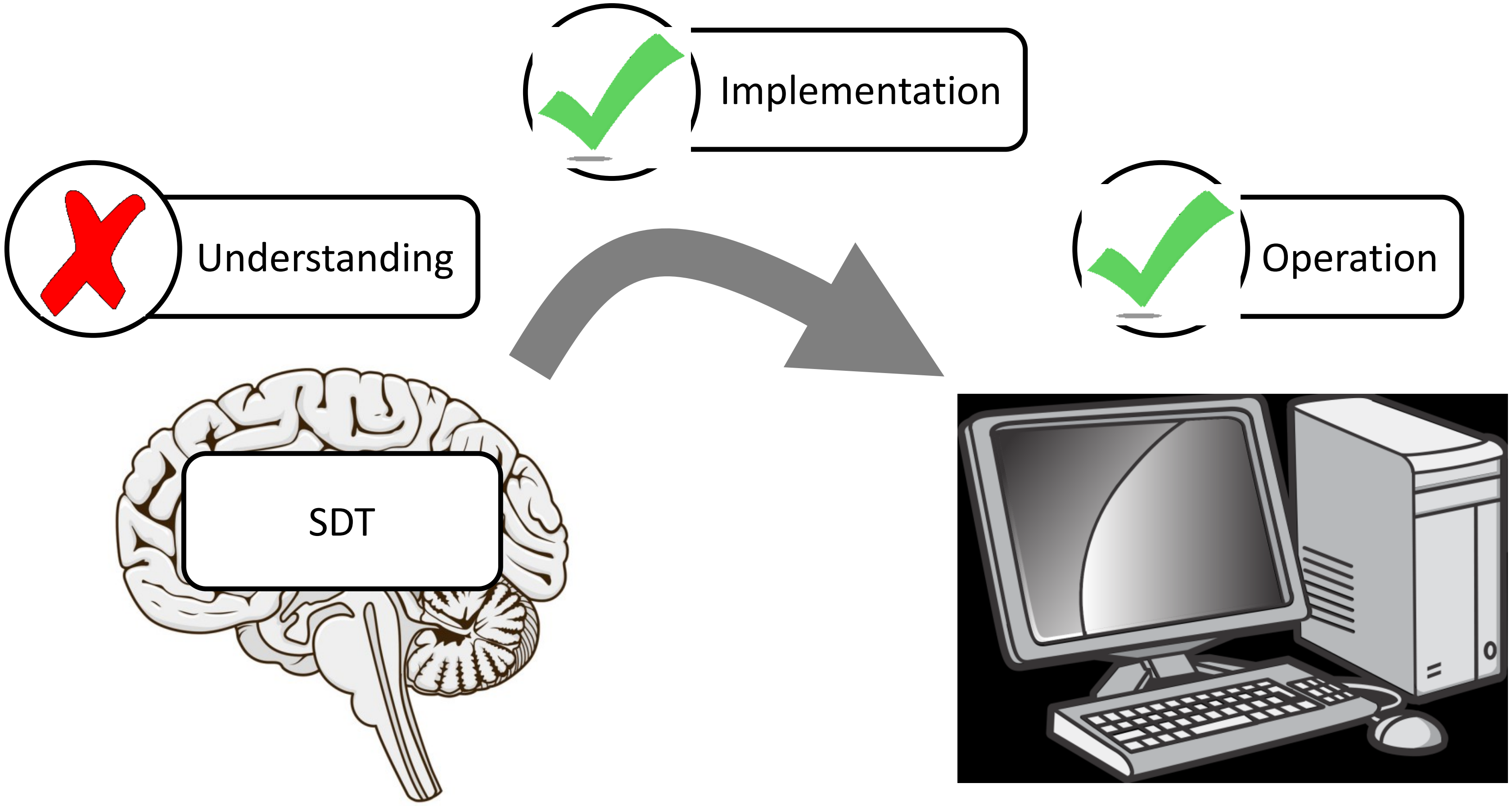
```
A problem has been detected and windows has been shut down to prevent damage
to your computer.

DRIVER_IRQL_NOT_LESS_OR_EQUAL

If this is the first time you've seen this Stop error screen,
restart your computer, If this screen appears again, follow
these steps:

Check to make sure any new hardware or software is properly installed.
If this is a new installation, ask your hardware or software manufacturer
for any windows updates you might need.

If problems continue, disable or remove any newly installed hardware
or software. Disable BIOS memory options such as caching or shadowing.
If you need to use Safe Mode to remove or disable components, restart
your computer, press F8 to select Advanced Startup Options, and then
select Safe Mode.

Technical information:

*** STOP: 0x000000D1 (0x0000000C,0x00000002,0x00000000,0xF86B5A89)


***      gv3.sys - Address F86B5A89 base at F86B5000, DateStamp 3dd991eb

Beginning dump of physical memory
Physical memory dump complete.
Contact your system administrator or technical support group for further
assistance.
```

Type Error Example

```
int a;
bool b;
a = true + 1 + 2 + b;
b = 2;
```



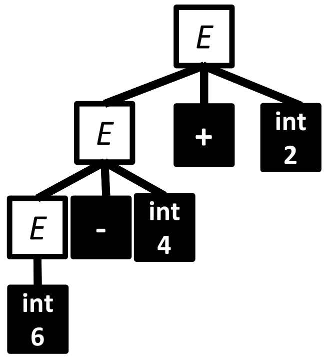
Understanding SDT



WTF is SDT?

- A method to translate a parse tree into… something

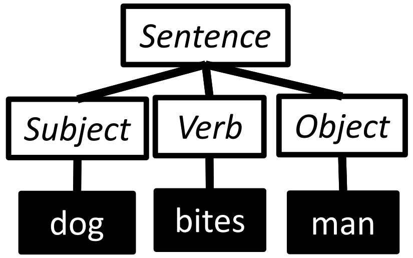
A mathematical expression's parse tree



…to its evaluation: 4

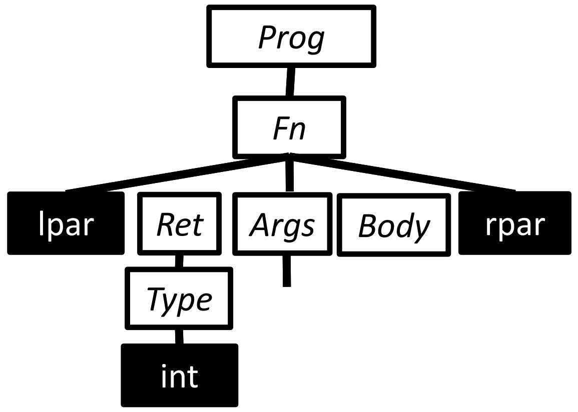
…to its set of operators: { **-** , **+** }

A simple sentence's parse tree



…to its subject: "dog"

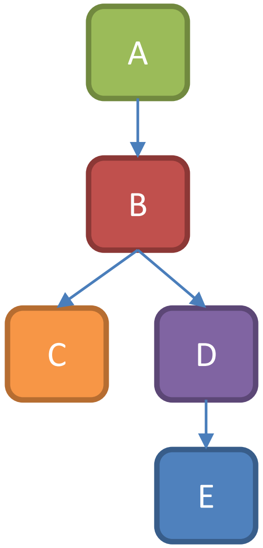
A programming language's parse tree into its AST



Lets talk more about trees

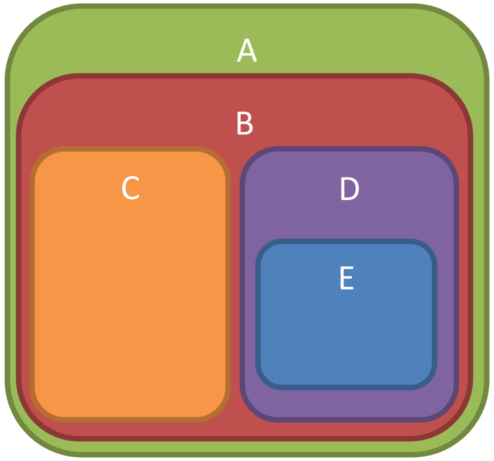
Two ways of thinking about Trees

As a type of graph

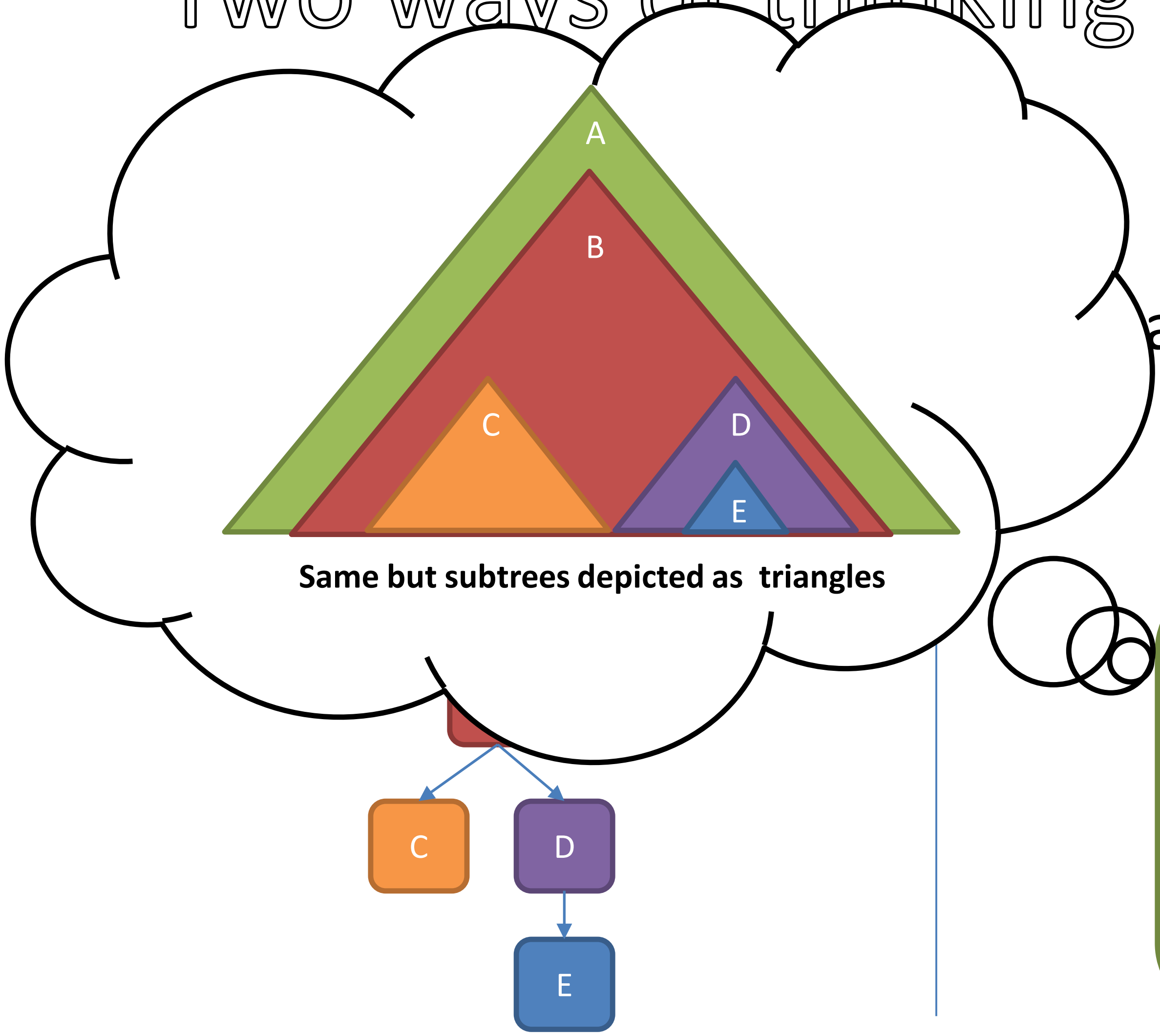


Compilers use insights from both views

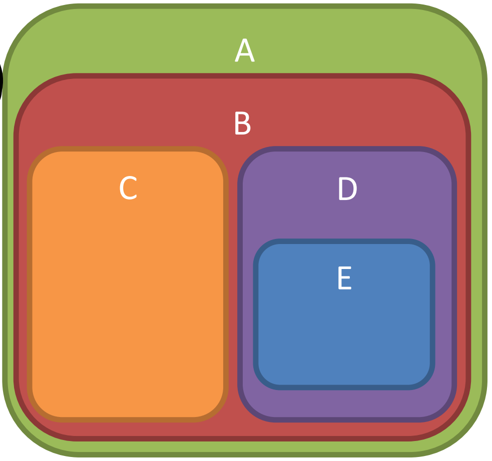
As a type of nesting



Two ways of thinking about Trees



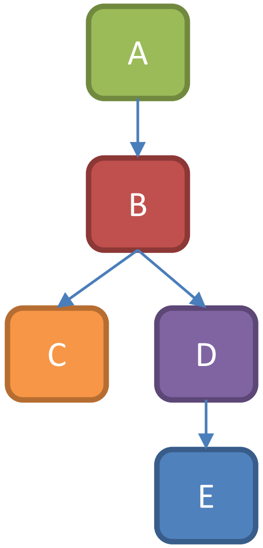
a type of nesting



Two ways of thinking about Trees

As a type of graph

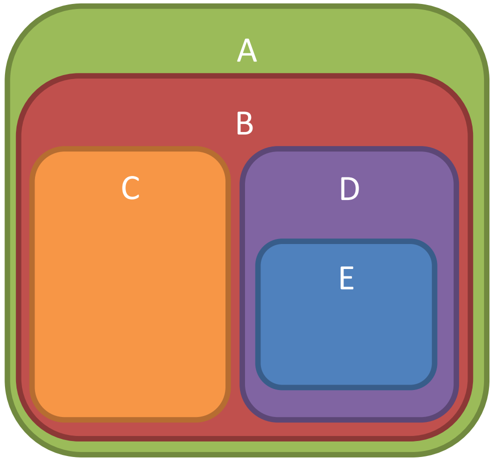
Root is a node in the tree with successors



Compilers use insights from both views

As a type of nesting

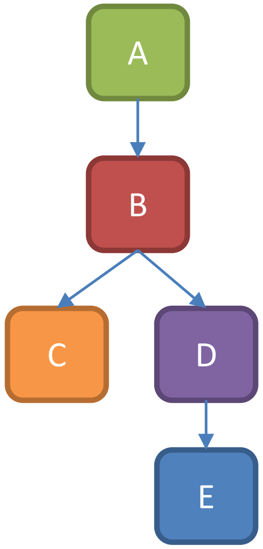
Root is the whole tree and contains subtrees



Two ways of thinking about Trees

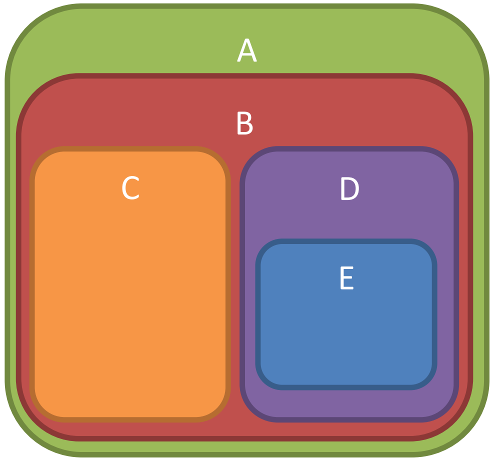
As a type of graph

Work
“top to bottom” / “preorder”
Or
“bottom to top” / “postorder”



As a type of nesting

Work
“outside in”
Or
“inside out”



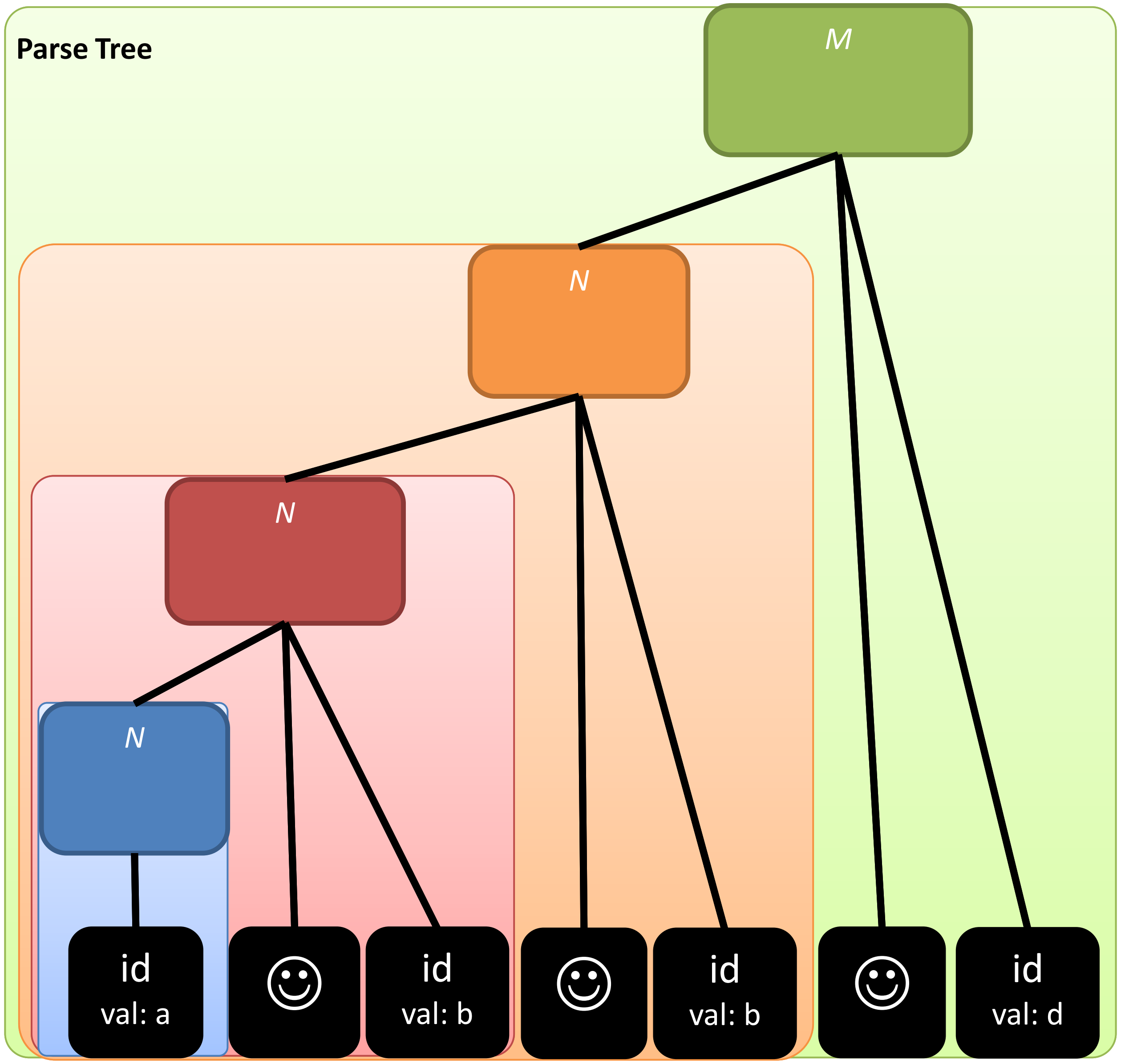
Production

$M \rightarrow N$ ☺ **id**

$\quad | \ \varepsilon$

$N \rightarrow N$ ☺ **id**

$\quad |$ **id**

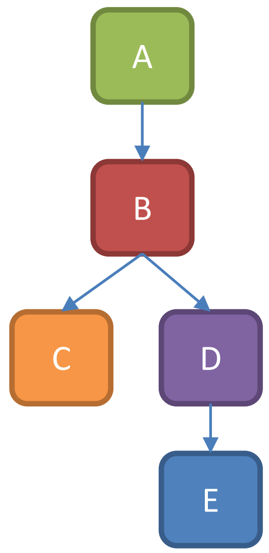


a :-) b :-) b :-) d

Thinking about SDT Actions

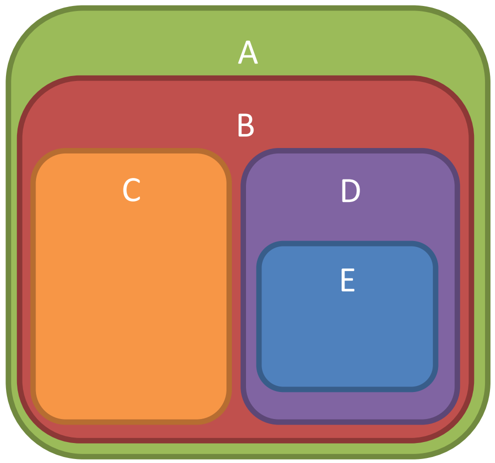
As a type of graph

SDT Actions
Backpropagate information from nodes
(preorder traversal)



As a type of nesting

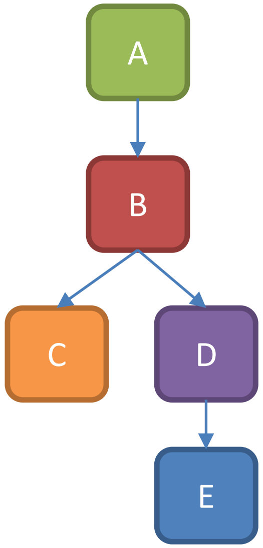
SDT Result
Expand the definition of the tree
(recursion)



Thinking about SDT Results

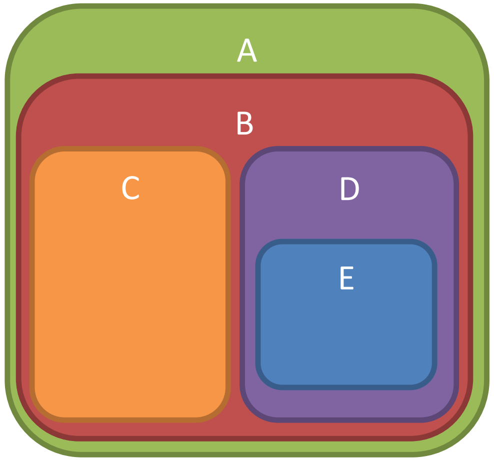
As a type of graph

SDT Result
Translation field of the root node



As a type of nesting

SDT Result
The translation of the whole tree
(the outermost tree IS the whole tree)



SDT Goals

- SDT is with respect to a goal
  - Derive some meaning or property from the tree
    - The SDT actions will be in service of that goal
    - The translation types will also be in service of that goal
  - Rules expressed in pseudocode
    - We'll let the fields be dynamically typed
    - Nonterminal .trans used to denote the translation OF THAT SUBTREE
    - Terminals have no translation (they aren't subtrees!)

SDT goal: Set of identifiers in the expression

Production	**SDT Rule**
$M \rightarrow N$ ☺ **id**	R_1
$\quad \mid \; \varepsilon$	R_2
$N \rightarrow N$ ☺ **id**	R_3
$\quad \mid$ **id**	R_4

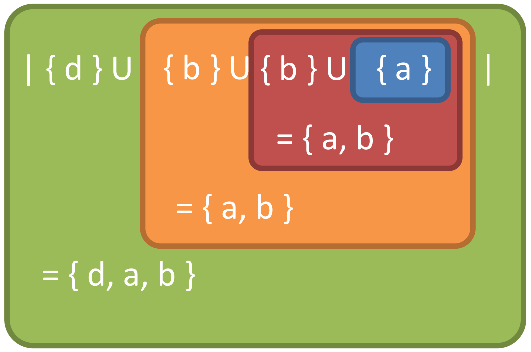
SDT rule definitions:

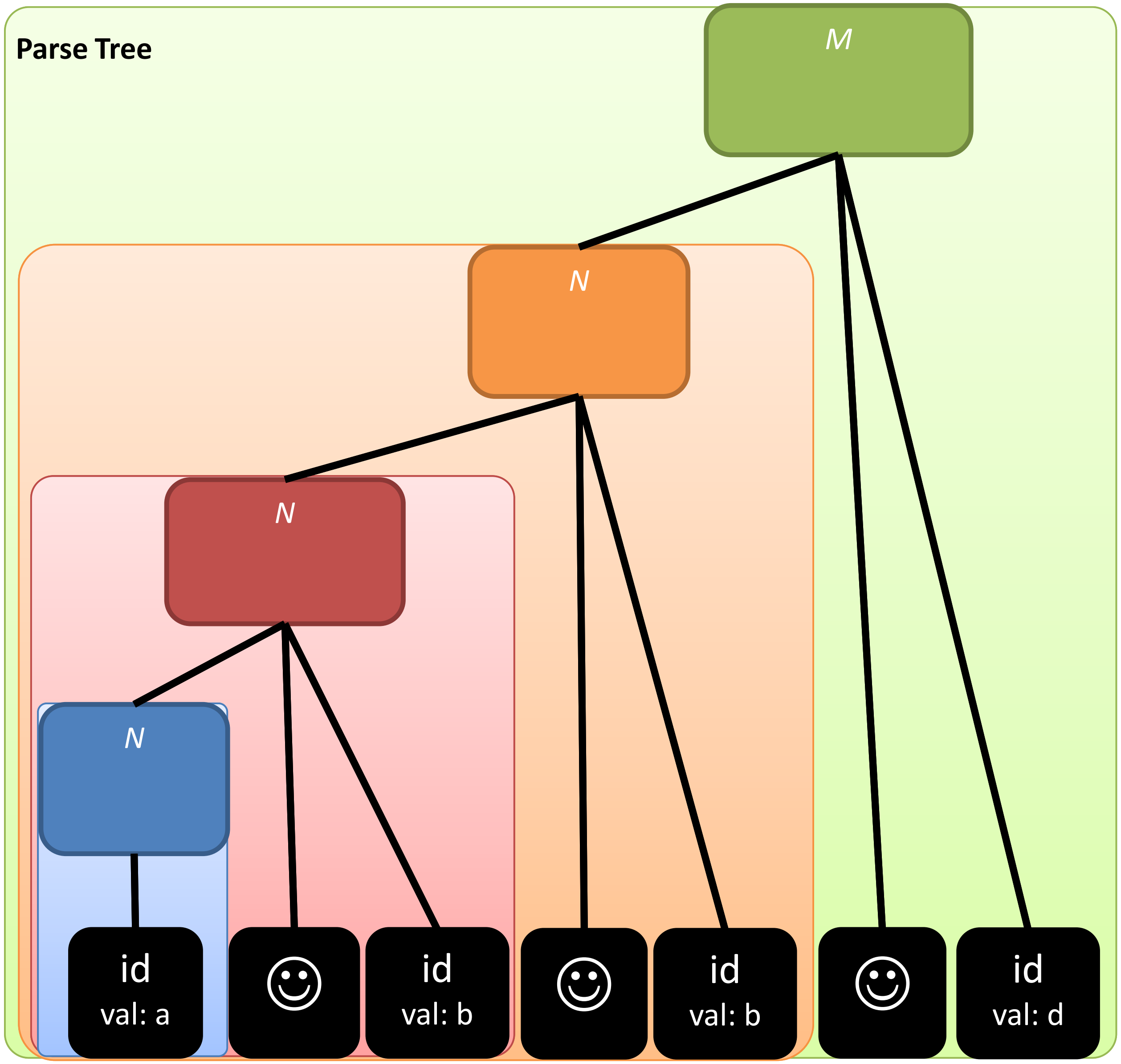
R_1: *M*.trans = { **id**.value } U *N*.trans

R_2: *M*.trans = { }

R_3: *N*.trans = { **id**.value } U N_2.trans

R_4: *M*.trans = { **id**.value }





<u>SDT goal</u>: Set of identifiers in the expression

Production	**SDT Rule**
$M \rightarrow N$ ☺ **id**	R_1
$\mid \varepsilon$	R_2
$N \rightarrow N$ ☺ **id**	R_3
$\mid$ **id**	R_4

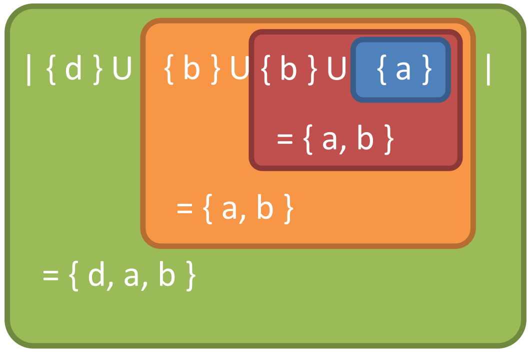
SDT rule definitions:

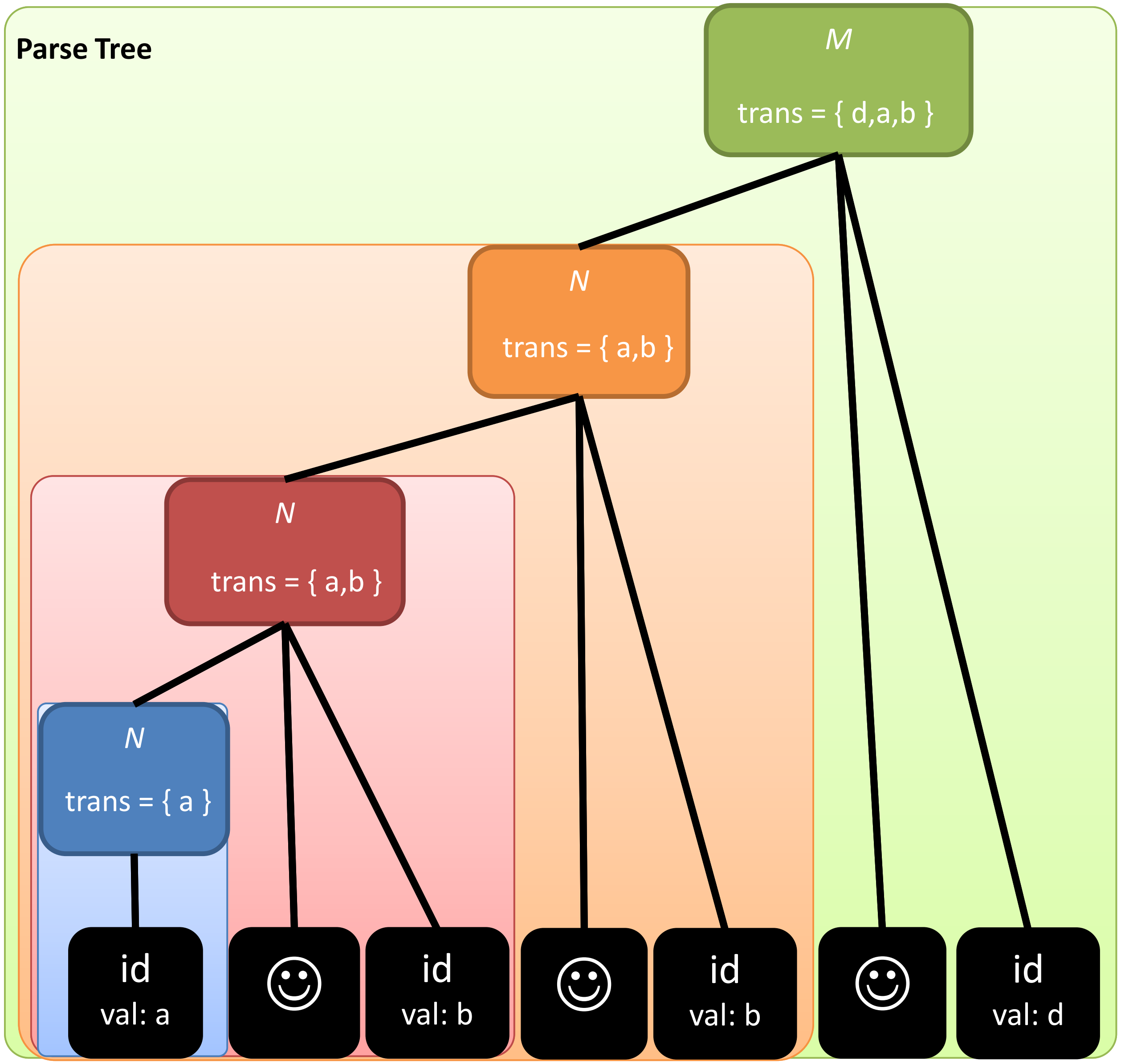
R_1: *M*.trans = { **id**.value } U *N*.trans

R_2: *M*.trans = { }

R_3: *N*.trans = { **id**.value } U N_2.trans

R_4: *M*.trans = { **id**.value }





Production	**SDT Rule**
$M \rightarrow N$ ☺ **id**	R_1
$\quad\mid\ \varepsilon$	R_2
$N \rightarrow N$ ☺ **id**	R_3
$\quad\mid$ **id**	R_4

SDT goal: Number of identifiers in the expression

SDT rule definitions: