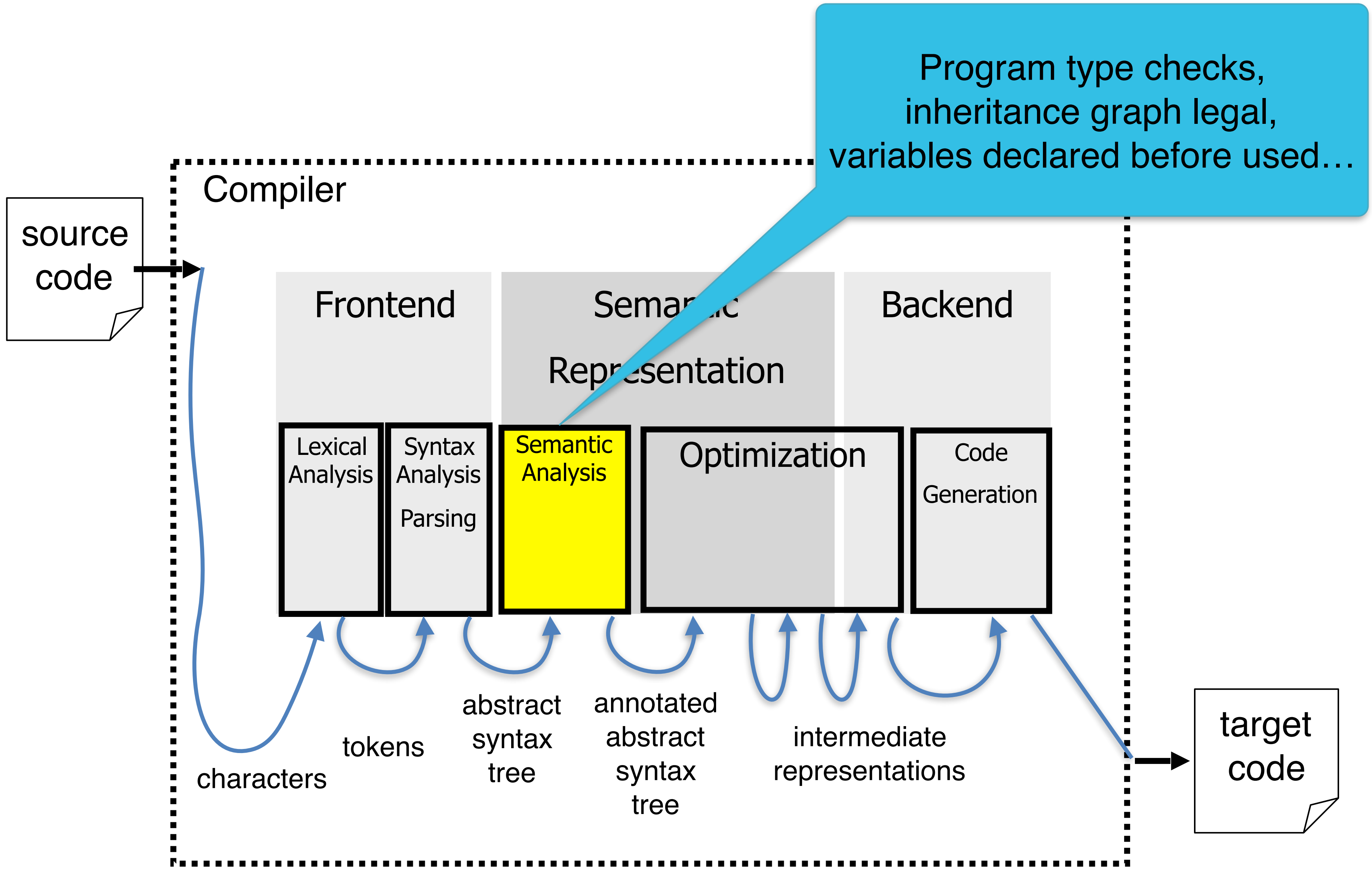
Semantic Analysis

Greta Yorsh


Program type checks, inheritance graph legal, variables declared before used…
Compiler
source code
Frontend
Representation
Backend
Lexical Analysis
Syntax Analysis
Parsing
Semantic Analysis
Optimization
Code Generation
characters
tokens
abstract syntax tree
annotated abstract syntax tree
intermediate representations
target code

What formalism should we use?

- Expressivity
- Computational complexity

finite languages | regular languages | context-free languages | context-sensitive languages | …

finite state automata | pushdown automata | ?

Semantic analysis: formalism?

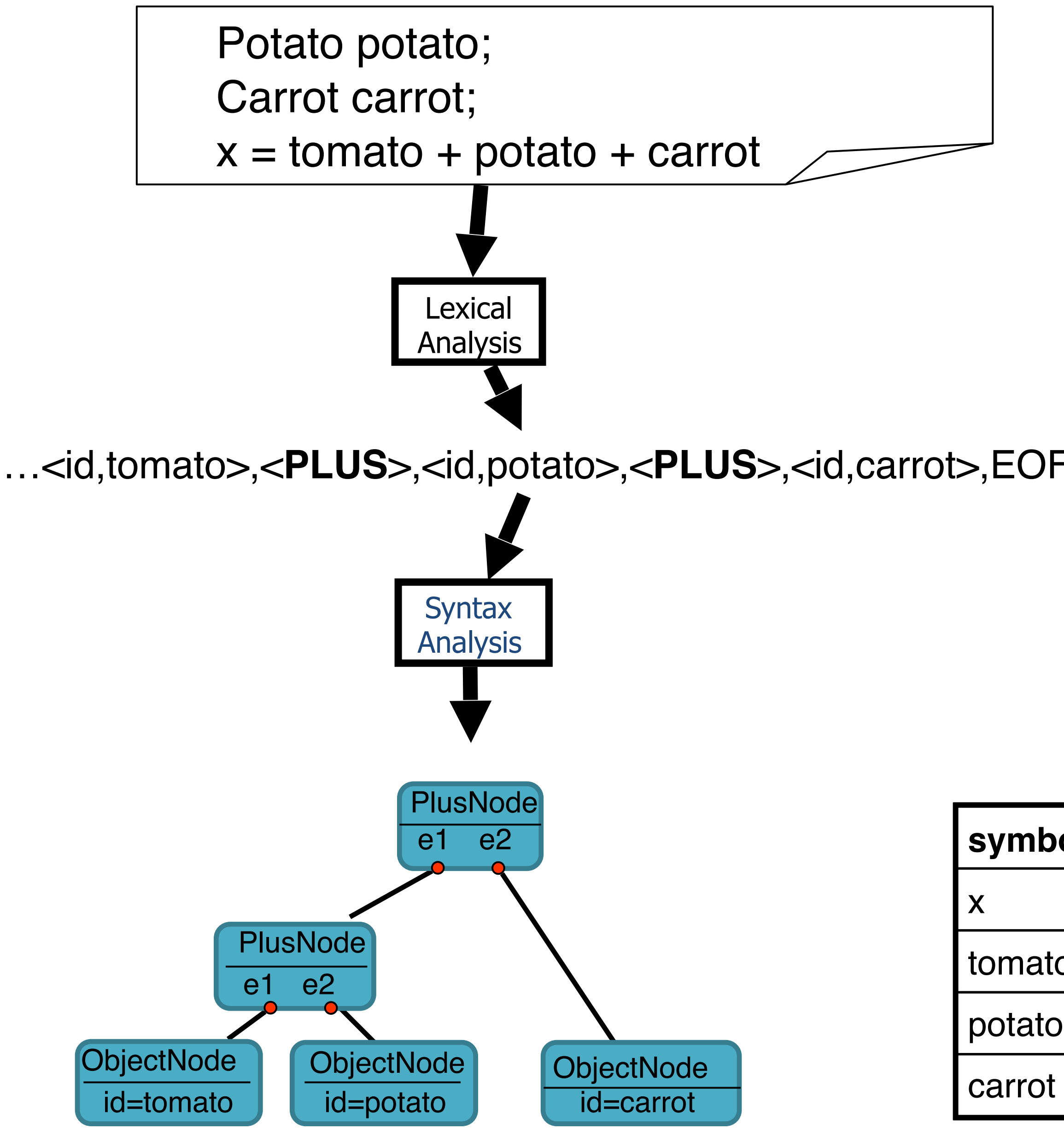
- We're on our own
- No standard tools to check more expressive languages
- No standard data structures
- We have to do it ourselves
  - custom algorithms to check
  - custom data structures to capture meaning

Semantic analysis

- Checking for “correct meaning”
- Warn about dubious meaning

- Long-distance and deep relations
- Lexer and parser are only short-distance

- Implemented **using AST traversals**



symbol	kind	type	properties
x	var	?	
tomato	var	?	
potato	var	Potato	
carrot	var	Carrot	

‘tomato’ is undefined

‘potato’ used before initialized

Cannot add Potato and Carrot

Semantic analysis: context

- Properties that **cannot be formulated** via context free grammars
  - type checking
  - declare before use: identifying the same word “w” re-appearing in input "wbw"
  - initialization
- Properties that are **hard to formulate** via context free grammar
  - “break” only appears inside a loop
- Processing of the AST

Semantic analysis: context

- **Identification**
  - gather information about each **named item** in the program
  - example: what is the declaration for each usage
- **Checking**
  - type compatibility
  - example: condition in an if-statement is a Boolean

Goals

- Reject programs that cannot be guaranteed to run correctly
- Compute information for next phases of compilation

Identification

```
month : integer RANGE [1..12];
month := 1;
while (month <= 12) {
  print(month_name[month]);
  month : = month + 1;
}
```

- Languages that allow use before declaration?
- Languages that don't require declarations?

Remember lexing/parsing?

- How did we know to always map an identifier to the same token?

Symbol table

```
month : integer RANGE [1..12];
…
month := 1;
while (month <= 12) {
  print(month_name[month]);
  month : = month + 1;
}
```

name	**type**	**…**
month	RANGE[1..12]	
month_name	…	
…		

- A table containing information about identifiers in the program
- Single entry for each named item

Not so fast…

```
struct one_int {
  int i;
} i;

main() {
 i.i = 42;
 int t = i.i;
 printf("%d",t);
}
```

A struct field named i

A struct variable named i

Assignment to the "i" field of struct "i"

Reading the "i" field of struct "i"

Not so fast…

```
struct one_int {
   int i;
} i;

main() {
  i.i = 42;
  int t = i.i;
  printf("%d",t);
  {
   int i = 73;
   printf("%d",i);
  }
}
```

A struct field named i

A struct variable named i

Assignment to the “i” field of struct “i”

Reading the “i” field of struct “i”

int variable named “i”

Scope of an identifier

- The part of the program in which the identifier is **accessible** or **visible**
- An identifier may have restricted scope
- Same identifier may refer to different things in different parts of the program
- Different scopes for same name **don't overlap**
- Not all kinds of identifiers follow **most-closely nested rule**

Example: scopes

```
class Foo {
  value : Int ← 39;
  test() : Int {
    let b:Int ← 3 in
       value + b
  };
  setValue(c:Int):Int {{
    value ← c;
    let d:Int ← c in {
       c ← c + d;
       value ← c;
    };
  }};
};
```

scope of **b**

scope of **d**

scope of **c**

scope of value

```
public class Bar {
  value:Int ← 42;
  setValue(int c):Int {
    value ← c;
  }
}
```

scope of **c**

scope of value

Scope rules

- Match identifier declarations with uses
- Why ?
  - for type checking…
  - example: Let y : String ← “abc” in y + 3
- Static scope: depends only on the program text, not runtime behavior

Scope: static vs dynamic

- Static: at compile time
  - name resolution based on source code location of identifier
  - examples: C,C++,Java, Ocaml, JavaScript
- Dynamic: at runtime
  - name resolution based on calling context
  - examples: Perl, Common LISP.

Example: static vs dynamic scope

```
int x = 37;
int y = 42;
void Function1() {
    Print(x + y);
}
void Function2() {
    int x = 0;
    Function1();
}
void Function3() {
    int y = 0;
    Function2();
}
Function1();
Function2();
Function3();
```

static scoping
output:
79
79
79

dynamic scoping
output:
79
42
0

Formalizing semantic analysis

- Scope rules
  - identifiers are declared
  - no multiple declarations of same identifier
  - local variables are declared before use
  - …
- Type rules
  - which types can be combined with certain operator
  - assignment of expression to variable
  - formal and actual parameters of a method call
  - …

Plan

- Scope rules
- Symbol tables
- Inheritance graph

- Next week: type rules

COOL SCOPE RULES

Where do identifiers come from ?

- How do we introduce identifier bindings in Cool ?

- Let expressions
- Formal parameters
- Attribute definitions
- Case expressions

(object ids)

- Class declarations (class names)
- Method declarations (method names)

Let scope: “most-closely nested” rule

```
let x : Int ← 0 in {
    x;
    let x : Int ← 2 in
    x;
    x;
}
```

Scope of class definitions

- Cannot be nested
- Globally visible
- Class name can be used before it is defined

```
Class Foo {
   … let y : Bar in …
}
Class Bar {
…
         }
```

Scope of attributes

- Global within the class in which they are defined
- Can be used before defined in the class

```
Class Foo {
   f() : Int { a };
   a : Int ← 0;
}
```

Scope of methods

- Method need not be defined in the class in which it is used, but in some parent class

```
class A {
    foo():Int { …};
};
class B inherits A { };
class C {
   b : B ← new B;
   bar():Int  { b.foo() };
};
```

Scope of methods

- Overriding: methods may be redefined

```
class A {
   foo():Int { …};
};
class B inherits A {
   foo():Int {…};
};
class C {
    b : B ← new B;
    bar():Int{ b.foo() };
};
```

Some cool scope rules

- Local variable declared before use
- Attributes need not be declared before use
- Variables cannot be defined multiple times in same scope, but can be redefined in nested scopes
- It is allowed to shadow method parameters

```
class A {
   x : String ← “a”;
   foo(x : Int, x : String): SELF_TYPE   { x };
};
```

SYMBOL TABLES

What is symbol table ?

- Data-structure for “look-up”
  - key – identifier
  - value – type of identifier, other semantic properties

- Scopes implemented using symbol tables

Symbol table: 1st attempt

```
class Test {
  a: Int ← 39;
  test(): Int {
    let b:Int ← 3 in
          a + b
  };
};
```

Symbol	Kind	Type	Properties
a	var	Int	…
b	var	Int	…
test	method	-> Int	…

Symbol table: 1st attempt

```
class Test {
  a: Int ← 39;
  test(): Int {
    let b:Int ← 3, a: String ← "hello" in
            a + b
  };
};
```

Symbol	Kind	Type	Properties
a	**var**	**Int**	**…**
b	var	Int	…
test	method	-> Int	…
a	**var**	**String**	**…**

Implementing scopes

- Let x : Int ← 0 in e
- before processing e
  - add definition of x to current definitions
  - override any other definition of x
- after processing e
  - remove definition of x
  - restore old definition of x

Symbol table: 2nd attempt

```
class Foo {
  value : Int ← 39;
  test(b:Int) : Int {
    value + b
  };
  setValue(c:Int):Int {{
    value ← c;
    let d:Int ← c in {
       c ← c + d;
       value ← c;
    };
  }};
};
```

Symbol table: 2nd attempt

(Foo)

Symbol	Kind	Type	Properties
value	var	Int	…
test	method	Int -> Int	
setValue	method	Int -> Int	

(test)

Symbol	Kind	Type	Properties
b	var	Int	…

(setValue)

Symbol	Kind	Type	Properties
c	var	Int	…

(setValue-let)

Symbol	Kind	Type	Properties
d	var	Int	…

Symbol table lookup

⋮

(Foo)

Symbol	Kind	Type	Properties
value	var	Int	…
test	method	Int -> Int	
setValue	method	Int -> Int	

(test)

Symbol	Kind	Type	Properties
b	var	Int	…

(setValue)

Symbol	Kind	Type	Properties
c	var	Int	…

(setValue-let)

Symbol	Kind	Type	Properties
d	var	Int	…

```
setValue(c:Int):Int {{
  value ← c;
  let d:Int ← c in {
    c ← c + d;
    value ← c;
  };
}};
```

Lookup(value)

Symbol table lookup

⋮

Error!

(Foo)

Symbol	Kind	Type	Properties
value	var	Int	…
test	method	Int -> Int	
setValue	method	Int -> Int	

(test)

Symbol	Kind	Type	Properties
b	var	Int	…

(setValue)

Symbol	Kind	Type	Properties
c	var	Int	…

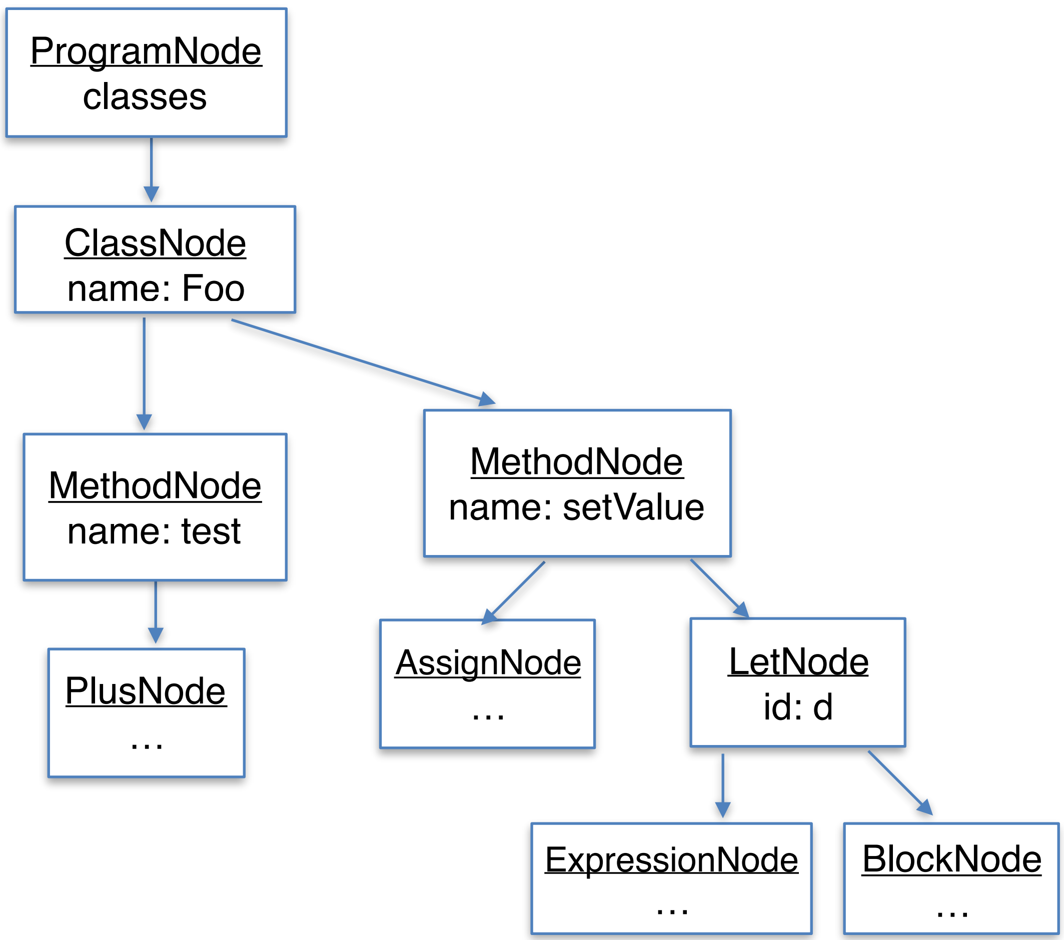
(setValue-let)

Symbol	Kind	Type	Properties
d	var	Int	…

```
setValue(c:Int):Int {{
  value ← c;
  let d:Int ← c in {
    c ← c + d;
    myvalue ← c;
  };
}};
```

Lookup(myvalue)

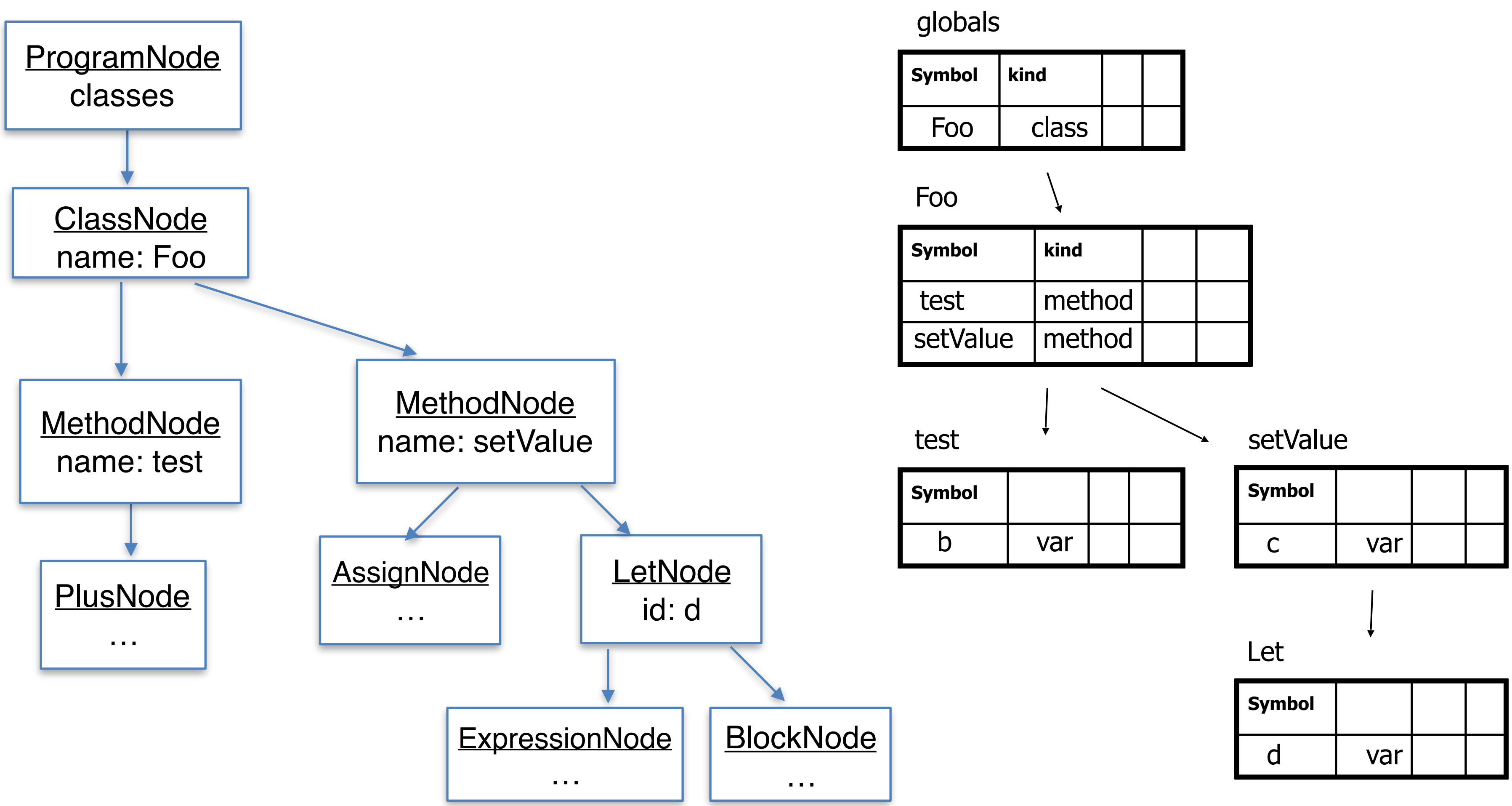
Symbol table construction




```
class Foo {
  value : Int ← 39;
  test(b:Int) : Int {
    value + b
  };
  setValue(c:Int):Int {{
    value ← c;
    let d:Int ← c in {
      c ← c + d;
      value ← c;
    };
  }};
};
```

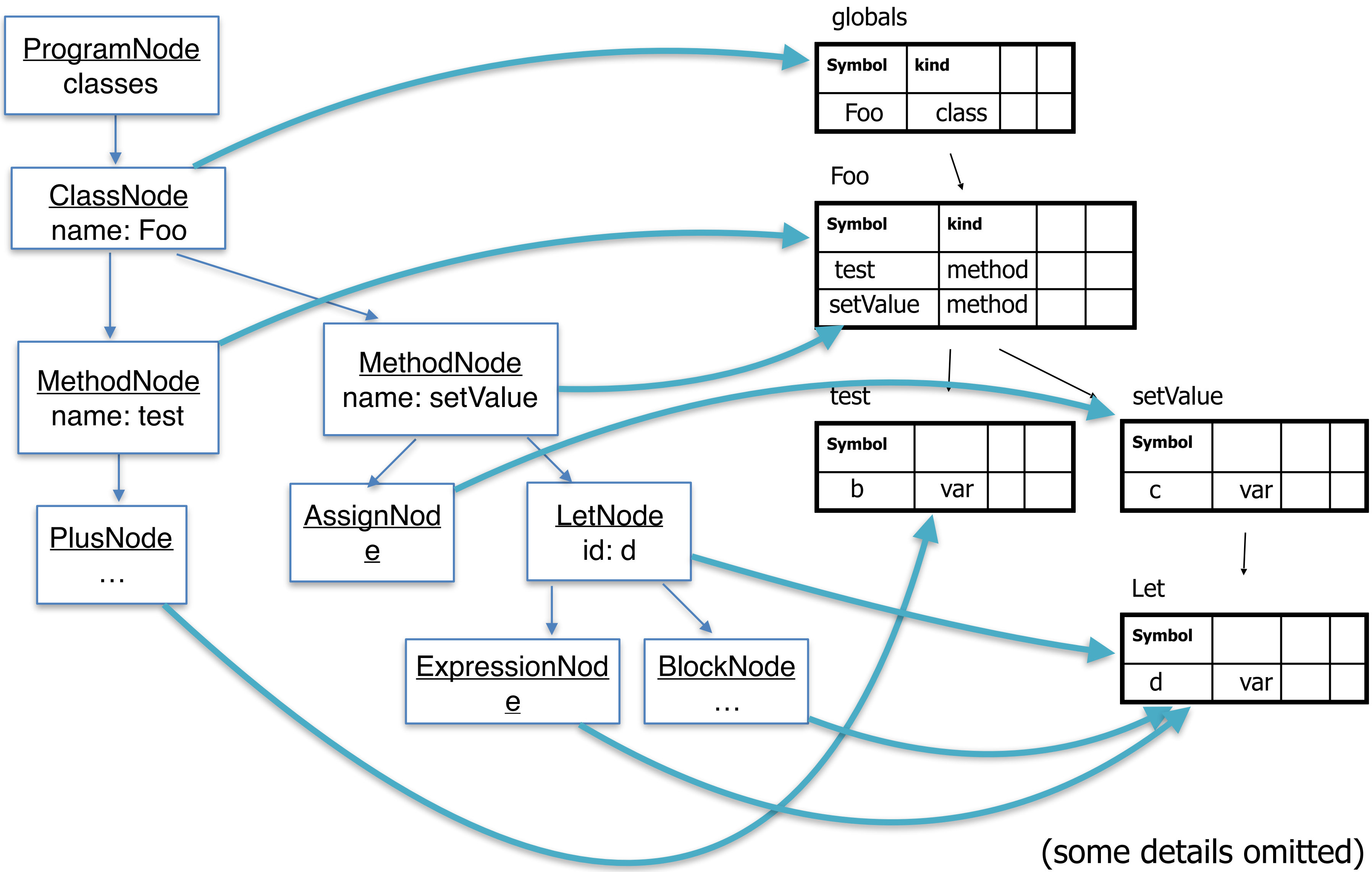
(some details omitted)

Symbol table construction



(some details omitted)

Symbol table construction



(some details omitted)

Symbol tables

- Used for computing scopes and checking scope rules
- Typically **stack structured**

- **Scope entry**: push new empty scope element
- **Scope exit**: pop scope element and discard its content
- **Identifier declaration**:
  identifier created inside (current) top scope
- **Identifier lookup**:
  search for identifier top-down in scope stack

Cool implementation

- SymbolTable.java
  - symbol table is a stack of scopes
  - scope is a hash table from key to value
  - key is ast.Symbol
  - value is generic

Our implementation support

enterScope()	start a new nested scope
lookup(x)	finds current x (or null)
addId(x)	adds a symbol x to the table
probe(x)	true if x defined in current scope
exitScope()	exit current scope

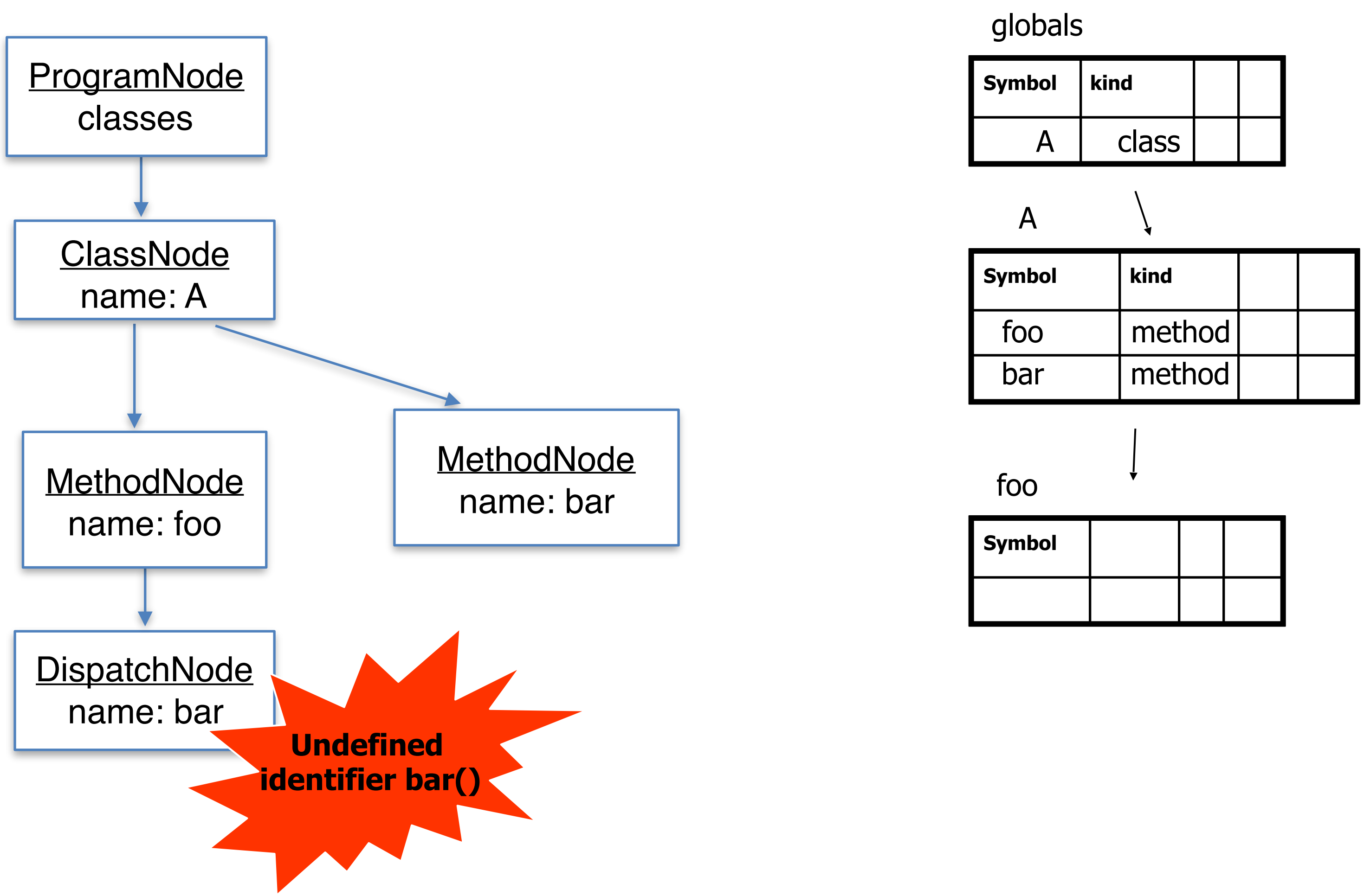
Your implementation should …

- Symbol table key should combine **id and kind**
  - separating table in advance according to kinds
  - method, attribute/local variable bindings, classes

  - implement using 2-level maps (kind->id->value)
  - implement this using key objects ((kind,id)->value))

Symbol table construction

```
class A {
   foo() {
      bar();
};
   bar() {…}
};
```



Symbol tables: naïve solution

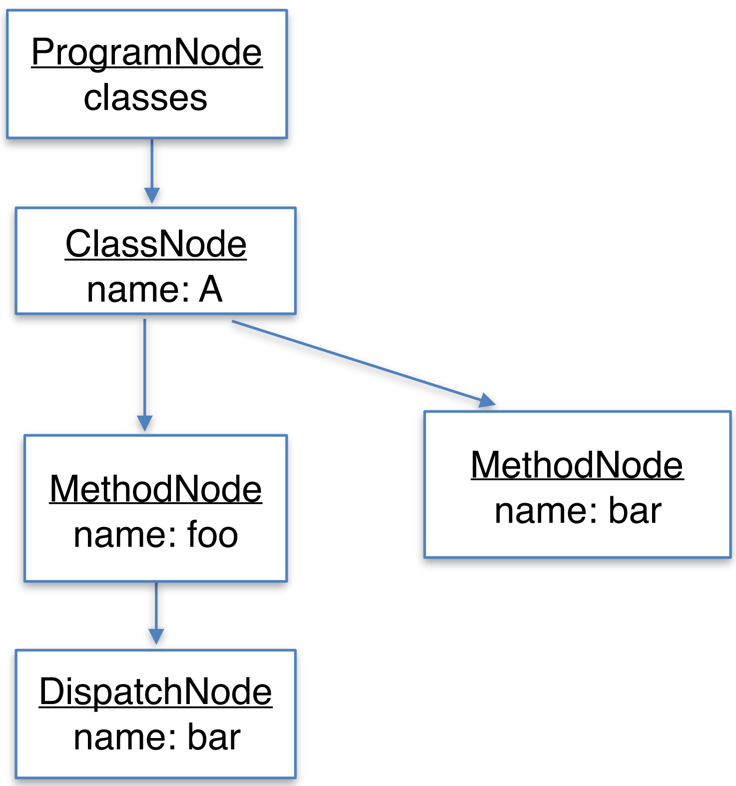
- Building visitor
  - Propagates (at least) a reference to the symbol table of the current scope
  - In some cases have to use type information (inherits)
- Checking visitor
  - On visit to node – perform check using symbol tables
    - resolve identifiers
  - Try to find symbol in table hierarchy
  - In some cases have to use global type table and type information
    - you may postpone these checks

Symbol tables: less naïve solution

- Use forward references
- And/or construct some of the symbol table during parsing

Symbol table construction

```
class A {
  foo() {
    bar();
}
  bar() {…}
}
```



globals

Symbol	kind		
A	class		

A

Symbol	kind	FREF	
foo	method		
bar	method	T	

foo

Symbol			

Forward references

- **Optimistically** assume that symbol will be eventually defined
- Update symbol table when symbol defined
  - Remove forward-reference marker
- But check correctness when exiting scope
  - No forward references should exist at exit

Passes

- Can we check class names using symbol table?
  - No.
- Can we check class names in one pass ?
  - No.

- Semantic analysis requires multiple passes
  - probably more than 2 for Cool

- pass 1: gather all class names
- pass 2: do the checking

INHERITANCE GRAPH

Inheritance graph

- What do we put in it?
    - all predefined classes: Int, String, Bool, Object, IO
    - all user-defined classes
- Why do we put it there?
    - to be able to check that something is a type
    - to be able to compare types (for type checking)

- Implementation
    - mapping class names to nodes in the graph
    - node points to its superclass node

Inheritance graph

- Node in the inheritance graph for each class
- Edges between parent and its children

Construct inheritance graph

- Initially the symbol table mapping class names to inheritance graph nodes is empty
- Add predefined classes
- Add user-defined classes

Check inheritance graph

- Local properties
  - do not require traversing inheritance graph
  - base class is not redefined
  - base class is not inherited from
- Global properties
  - all classes are reachable from root class Object
  - inheritance graphs is a tree (no cycles)

Major semantic tasks

- Inheritance graph construction and checking
- Symbol table construction
  - construct symbol table for features using inheritance tree
  - **assign enclosing scope for each AST node**
- Scope checking using symbol tables
- Check for Main class and main method
- Type checking for all expressions
  - uses inheritance graph and symbol tables
  - **assign type for each AST node**