On Scope Graphs and Reference Attribute Grammars

Answering Eelco's Questions

Luke Bessant bessa028@umn.edu
Eric Van Wyk evw@umn.edu

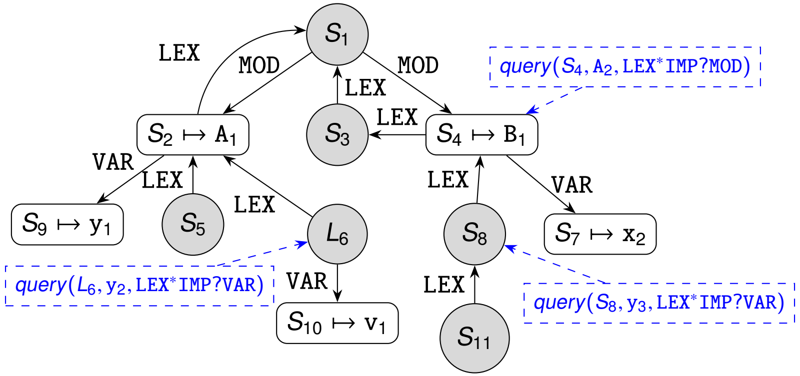
Department of Computer Science & Engineering
University of Minnesota

WG 2.11
Stellenbosch

- Eelco discussed scope graphs here several times
- I talked to him about this in Salem in 2019
- but didn't then understand the issue and was preoccupied with other things
- this work is pulling a thread I should have pulled on earlier
- I spoke about these in Delft and Eelco's symposium but didn't really know what I was talking about then
- this is work w/ Luke and he's clarified many aspect of this.

Scope Graphs

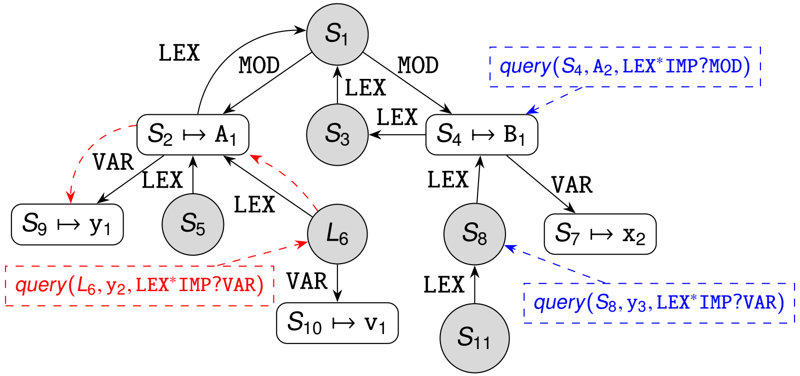
```
module A1 {
 def y1 =
  let v1 = 2 in
    y2 + v2
}

module B1 {
 import A2
 def x2 = y2
}
```



- Example in LM, a language with modules, imports and variable definitions - defined intuitively
- Indices are not in LM syntax, just there to refer to specific instances of names
- This scope graph encodes sequential imports
- L_6 is the scope of the body of the let
- Mention that resolutions can follow any number of LEX edges, followed by an optional IMP edge, followed by a MOD for modules or VAR for variables

Scope Graphs

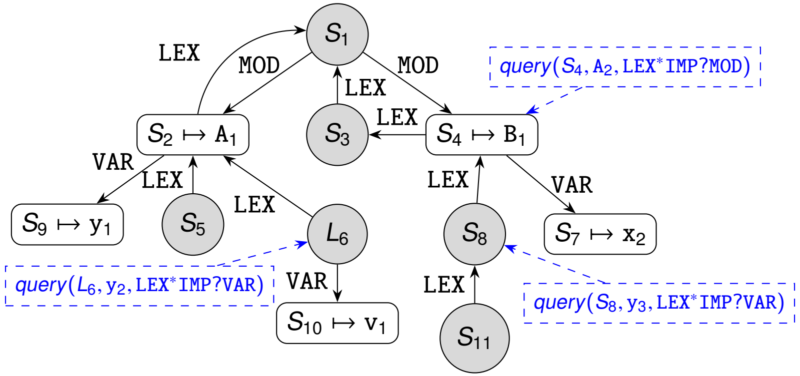
```
module A₁ {
  def y₁ =
    let v₁ = 2 in
      y₂ + v₂
}

module B₁ {
  import A₂
  def x₂ = y₂
}
```



- Example in LM, a language with modules, imports and variable definitions - defined intuitively
- Indices are not in LM syntax, just there to refer to specific instances of names
- This scope graph encodes sequential imports
- L_6 is the scope of the body of the let
- Mention that resolutions can follow any number of LEX edges, followed by an optional IMP edge, followed by a MOD for modules or VAR for variables

Scope Graphs

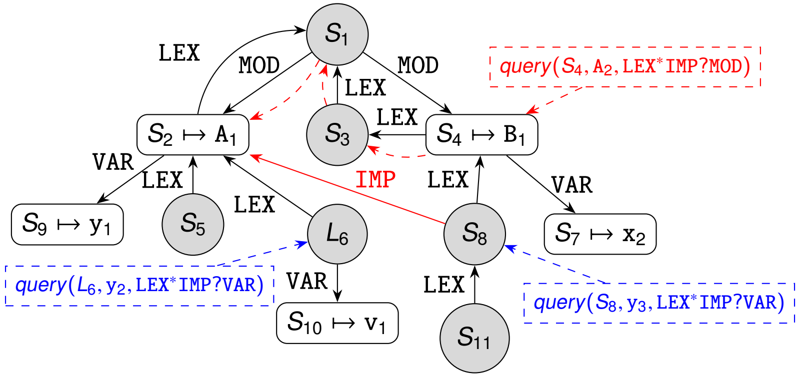
```
module A1 {
  def y1 =
    let v1 = 2 in
      y2 + v2
}

module B1 {
  import A2
  def x2 = y2
}
```



- Example in LM, a language with modules, imports and variable definitions - defined intuitively
- Indices are not in LM syntax, just there to refer to specific instances of names
- This scope graph encodes sequential imports
- L_6 is the scope of the body of the let
- Mention that resolutions can follow any number of LEX edges, followed by an optional IMP edge, followed by a MOD for modules or VAR for variables

Scope Graphs

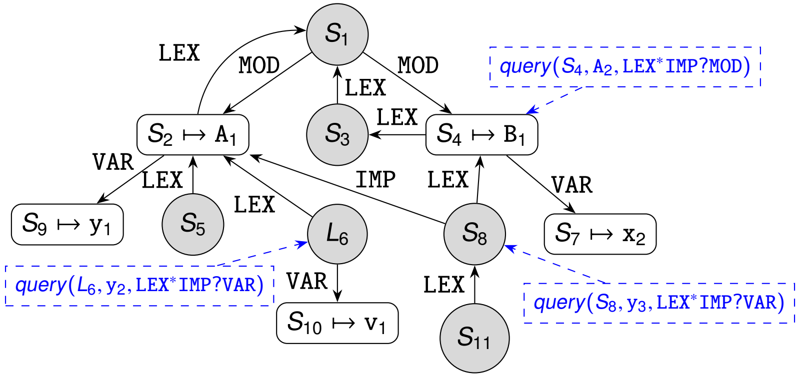
```
module A1 {
 def y1 =
  let v1 = 2 in
    y2 + v2
}

module B1 {
 import A2
 def x2 = y2
}
```



- Example in LM, a language with modules, imports and variable definitions - defined intuitively
- Indices are not in LM syntax, just there to refer to specific instances of names
- This scope graph encodes sequential imports
- L_6 is the scope of the body of the let
- Mention that resolutions can follow any number of LEX edges, followed by an optional IMP edge, followed by a MOD for modules or VAR for variables

Scope Graphs

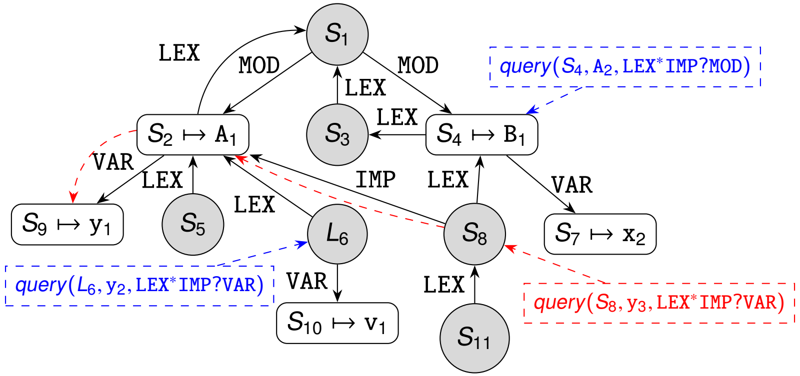
```
module A1 {
 def y1 =
  let v1 = 2 in
    y2 + v2
}

module B1 {
 import A2
 def x2 = y2
}
```



- Example in LM, a language with modules, imports and variable definitions - defined intuitively
- Indices are not in LM syntax, just there to refer to specific instances of names
- This scope graph encodes sequential imports
- L_6 is the scope of the body of the let
- Mention that resolutions can follow any number of LEX edges, followed by an optional IMP edge, followed by a MOD for modules or VAR for variables

Scope Graphs

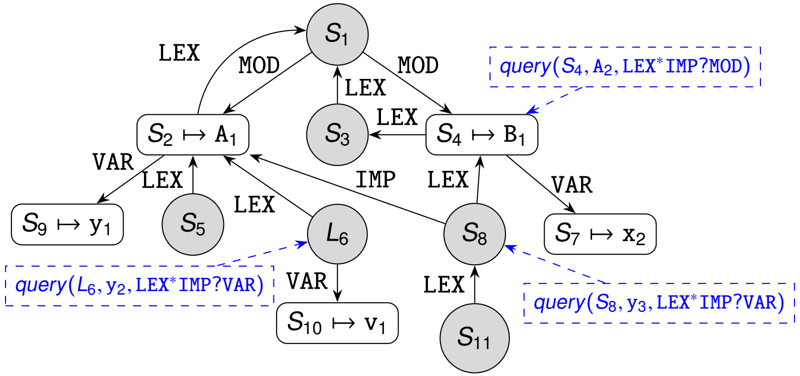
```
module A1 {
  def y1 =
    let v1 = 2 in
      y2 + v2
}

module B1 {
  import A2
  def x2 = y2
}
```



- Example in LM, a language with modules, imports and variable definitions - defined intuitively
- Indices are not in LM syntax, just there to refer to specific instances of names
- This scope graph encodes sequential imports
- L_6 is the scope of the body of the let
- Mention that resolutions can follow any number of LEX edges, followed by an optional IMP edge, followed by a MOD for modules or VAR for variables

Scope Graphs

```
module A1 {
 def y1 =
  let v1 = 2 in
    y2 + v2
}

module B1 {
 import A2
 def x2 = y2
}
```



- Example in LM, a language with modules, imports and variable definitions - defined intuitively
- Indices are not in LM syntax, just there to refer to specific instances of names
- This scope graph encodes sequential imports
- L_6 is the scope of the body of the let
- Mention that resolutions can follow any number of LEX edges, followed by an optional IMP edge, followed by a MOD for modules or VAR for variables

Statix: A constraint-solving implementation for scope graphs

- Conjunction of scope graph and type checking constraints
- Syntax predicates resemble nonterminal and production declarations in a context free grammar
- Scope graphs built gradually by solving scope and edge assertions
- Query constraints return resolution paths in the *current* scope graph

```
dcls(s, sm, ds) :- ds match
{ cons(d, ds) -> {sn}
    new sn,
    sn -[LEX]-> s,
    dcl(s, sn, sm, d), dcls(sn, sm, ds)
| nil() -> true
}.

dcl(s, sn, sm, d) :- d match
{ mod(id, ds) ->  {sm'}
    new sm' -> id,
    sm -[MOD]-> sm',
    sm' -[LEX]-> s,
    dcls(sm', sm', ds)
| imp(name) {rs, rs', r, sm}
    query(s, LEX*IMP?MOD, mod-is(name), rs),
    min(rs, LEX > IMP > VAR = MOD, rs'),
    single(rs', r), tgt(r, sm),
    sn -[IMP]-> sm
}.
```

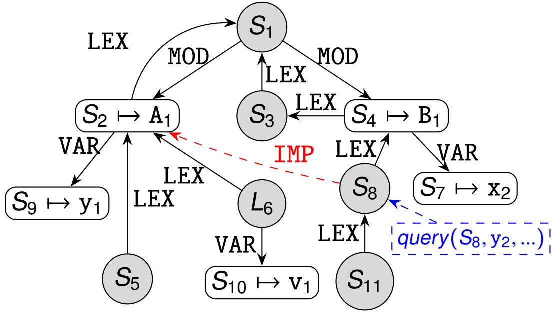
Statix: Ensuring Soundness of Name Resolution

- Queries are blocked if edges they can follow are asserted but unsolved
- These edges *may* lead to new answers for the query if added to the scope graph
- Found by analysis of the constraint set
- Referred to as weakly critical edges

Statix constraint set:

```
{ ...,
  s8 -[IMP]-> s2 ,
  query(s8 , LEX* IMP? VAR, "y2"),
  ... }
```

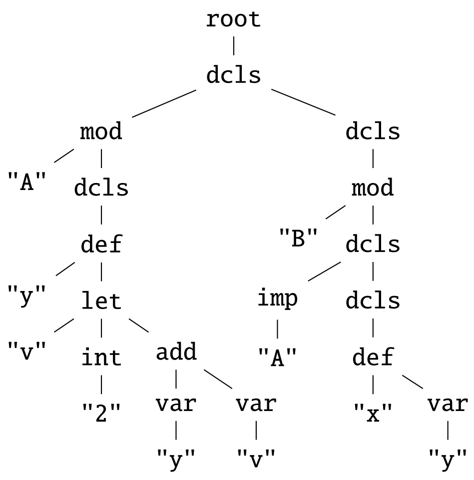
Corresponding graph state:



- I don't think Eelco liked these weakly critical edges.
- He had questions about their relationship to AGs and attribute evaluation.
- These imposed some restrictions he was trying to address.

Scope Graphs and Reference Attribute Grammars

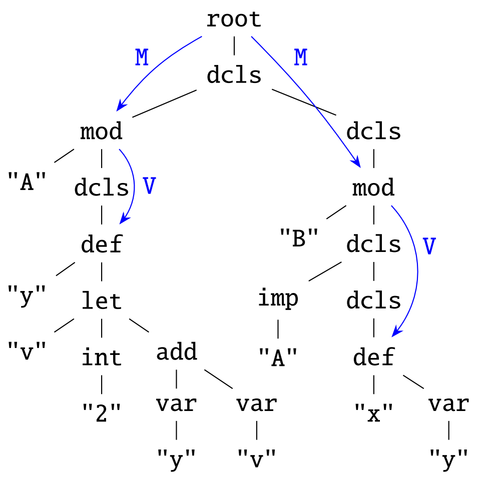
- Context free grammars define syntax.
- Attribute Grammars (AGs) decorate a syntax tree with semantic values called attributes.
- Equations (on productions) specify their values.
- Reference attributes are pointers to remote nodes in the tree.
- RAGs can draw scope graphs over the tree.



- make the code the same as in the "marriage" slide
- remove demand edges
- show edges for all VAR edges - missing the one to "v" defined in the let. But if this looks messy leave it out.
- add resolution edges from reference to (at least) "y" that used the import and one of "y" and "v" that do not.
- Eelco's questions ... what is the correspondence?

Scope Graphs and Reference Attribute Grammars

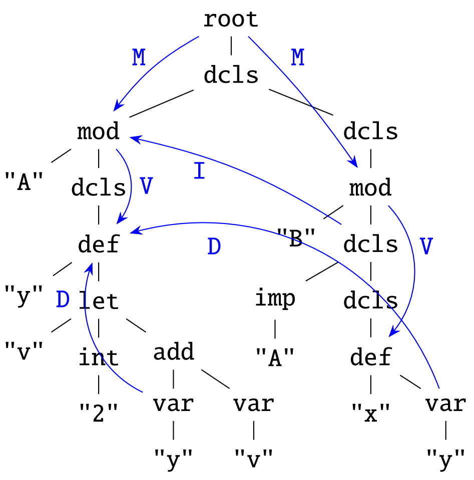
- Context free grammars define syntax.
- Attribute Grammars (AGs) decorate a syntax tree with semantic values called attributes.
- Equations (on productions) specify their values.
- Reference attributes are pointers to remote nodes in the tree.
- RAGs can draw scope graphs over the tree.



- make the code the same as in the "marriage" slide
- remove demand edges
- show edges for all VAR edges - missing the one to "v" defined in the let. But if this looks messy leave it out.
- add resolution edges from reference to (at least) "y" that used the import and one of "y" and "v" that do not.
- Eelco's questions ... what is the correspondence?

Scope Graphs and Reference Attribute Grammars

- Context free grammars define syntax.
- Attribute Grammars (AGs) decorate a syntax tree with semantic values called attributes.
- Equations (on productions) specify their values.
- Reference attributes are pointers to remote nodes in the tree.
- RAGs can draw scope graphs over the tree.

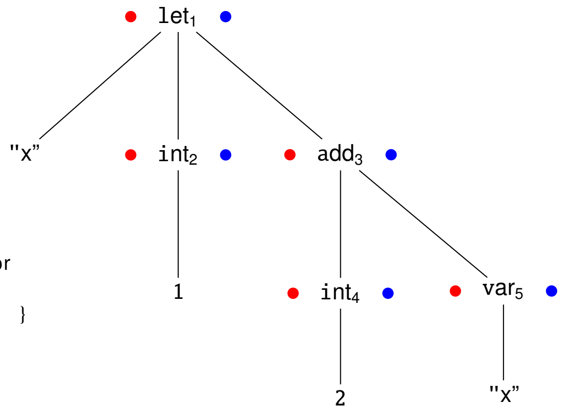


- make the code the same as in the "marriage" slide
- remove demand edges
- show edges for all VAR edges - missing the one to "v" defined in the let. But if this looks messy leave it out.
- add resolution edges from reference to (at least) "y" that used the import and one of "y" and "v" that do not.
- Eelco's questions ... what is the correspondence?

Demand-driven Attribute Evaluation

```
syn attr val:Int; inh attr env:Env;
nt Expr with val, env;

production int:
top:Expr ::= i:Int
{ top.val = i; }

production add:
top:Expr ::= l:Expr r:Expr
{ top.val = l.val + r.val;
  l.env = top.env; r.env =  top.env; }

production let:
top:Expr ::= id:String bnd:Expr bod:Expr
{ top.val = bod.val;
  bod.env = addEnv(id, bnd.val, top.env); }

production var:
top:Expr ::= id:String
{ top.val = lookup(top.env, id); }
```

- Abstract tree for ‘`let x = 1 in 2 + x`’
- Arrows illustrate attribute demands

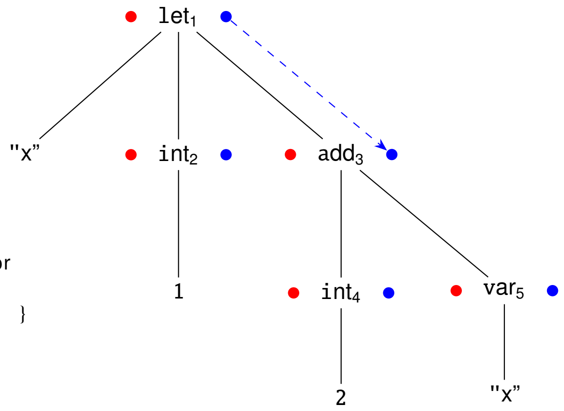


- Animate the tree:
  1. Demand let val
  2. Demand add val
  3. Demand int val
  4. Demand var val
  5. Demand var env
  6. Demand add env
  7. Demand int val
  8. ...
- Show values next to dots when attributes completed

Demand-driven Attribute Evaluation

```
syn attr val:Int; inh attr env:Env;
nt Expr with val, env;

production int:
top:Expr ::= i:Int
{ top.val = i; }

production add:
top:Expr ::= l:Expr r:Expr
{ top.val = l.val + r.val;
  l.env = top.env; r.env =  top.env; }

production let:
top:Expr ::= id:String bnd:Expr bod:Expr
{ top.val = bod.val;
  bod.env = addEnv(id, bnd.val, top.env); }

production var:
top:Expr ::= id:String
{ top.val = lookup(top.env, id); }
```

- Abstract tree for ‘`let x = 1 in 2 + x`’
- Arrows illustrate attribute demands

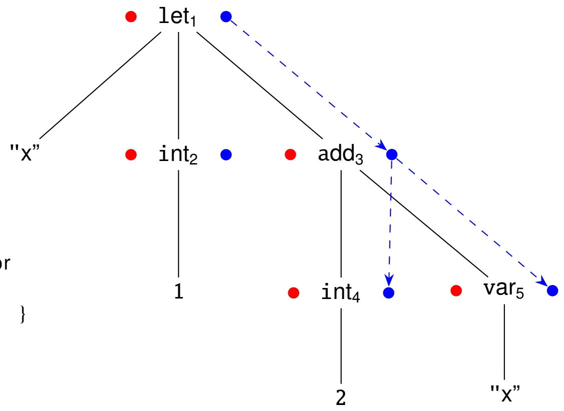


- Animate the tree:
  1. Demand let val
  2. Demand add val
  3. Demand int val
  4. Demand var val
  5. Demand var env
  6. Demand add env
  7. Demand int val
  8. ...
- Show values next to dots when attributes completed

Demand-driven Attribute Evaluation

```
syn attr val:Int; inh attr env:Env;
nt Expr with val, env;

production int:
top:Expr ::= i:Int
{ top.val = i; }

production add:
top:Expr ::= l:Expr r:Expr
{ top.val = l.val + r.val;
  l.env = top.env; r.env =  top.env; }

production let:
top:Expr ::= id:String bnd:Expr bod:Expr
{ top.val = bod.val;
  bod.env = addEnv(id, bnd.val, top.env); }

production var:
top:Expr ::= id:String
{ top.val = lookup(top.env, id); }
```

- Abstract tree for '`let x = 1 in 2 + x`'
- Arrows illustrate attribute demands

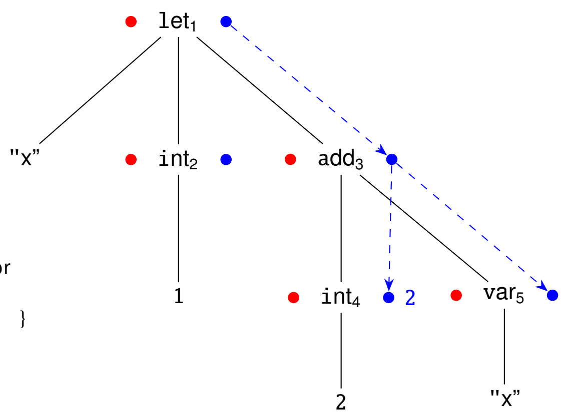


- Animate the tree:
  1. Demand let val
  2. Demand add val
  3. Demand int val
  4. Demand var val
  5. Demand var env
  6. Demand add env
  7. Demand int val
  8. ...
- Show values next to dots when attributes completed

Demand-driven Attribute Evaluation

```
syn attr val:Int; inh attr env:Env;
nt Expr with val, env;

production int:
top:Expr ::= i:Int
{ top.val = i; }

production add:
top:Expr ::= l:Expr r:Expr
{ top.val = l.val + r.val;
  l.env = top.env; r.env =  top.env; }

production let:
top:Expr ::= id:String bnd:Expr bod:Expr
{ top.val = bod.val;
  bod.env = addEnv(id, bnd.val, top.env); }

production var:
top:Expr ::= id:String
{ top.val = lookup(top.env, id); }
```

- Abstract tree for '`let x = 1 in 2 + x`'
- Arrows illustrate attribute demands

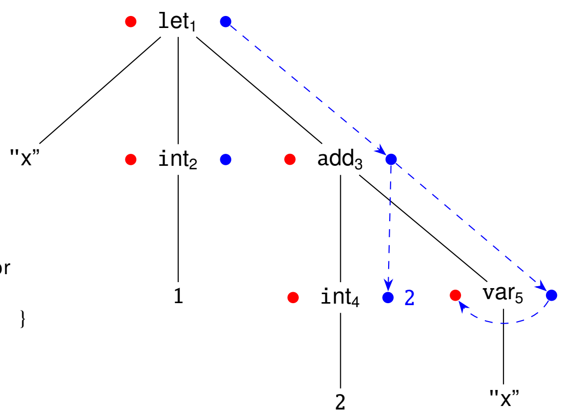


- Animate the tree:
    1. Demand let val
    2. Demand add val
    3. Demand int val
    4. Demand var val
    5. Demand var env
    6. Demand add env
    7. Demand int val
    8. ...
- Show values next to dots when attributes completed

Demand-driven Attribute Evaluation

```
syn attr val:Int; inh attr env:Env;
nt Expr with val, env;

production int:
top:Expr ::= i:Int
{ top.val = i; }

production add:
top:Expr ::= l:Expr r:Expr
{ top.val = l.val + r.val;
  l.env = top.env; r.env =  top.env; }

production let:
top:Expr ::= id:String bnd:Expr bod:Expr
{ top.val = bod.val;
  bod.env = addEnv(id, bnd.val, top.env); }

production var:
top:Expr ::= id:String
{ top.val = lookup(top.env, id); }
```

- Abstract tree for 'let x = 1 in 2 + x'
- Arrows illustrate attribute demands

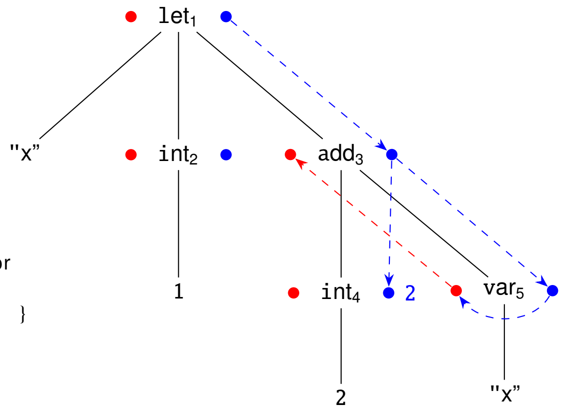


- Animate the tree:
  1. Demand let val
  2. Demand add val
  3. Demand int val
  4. Demand var val
  5. Demand var env
  6. Demand add env
  7. Demand int val
  8. ...
- Show values next to dots when attributes completed

Demand-driven Attribute Evaluation

```
syn attr val:Int; inh attr env:Env;
nt Expr with val, env;

production int:
top:Expr ::= i:Int
{ top.val = i; }

production add:
top:Expr ::= l:Expr r:Expr
{ top.val = l.val + r.val;
  l.env = top.env; r.env =  top.env; }

production let:
top:Expr ::= id:String bnd:Expr bod:Expr
{ top.val = bod.val;
  bod.env = addEnv(id, bnd.val, top.env); }

production var:
top:Expr ::= id:String
{ top.val = lookup(top.env, id); }
```

- Abstract tree for '`let x = 1 in 2 + x`'
- Arrows illustrate attribute demands

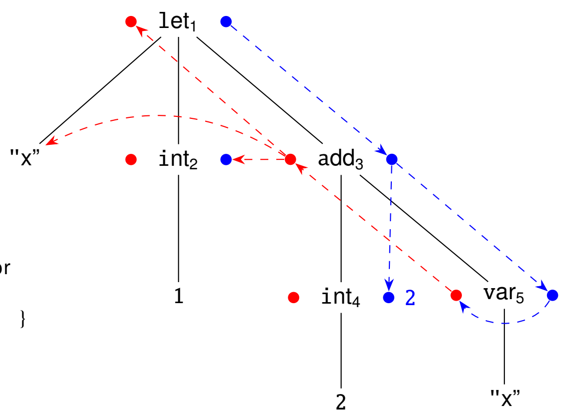


- Animate the tree:
  1. Demand let val
  2. Demand add val
  3. Demand int val
  4. Demand var val
  5. Demand var env
  6. Demand add env
  7. Demand int val
  8. ...
- Show values next to dots when attributes completed

Demand-driven Attribute Evaluation

```
syn attr val:Int; inh attr env:Env;
nt Expr with val, env;

production int:
top:Expr ::= i:Int
{ top.val = i; }

production add:
top:Expr ::= l:Expr r:Expr
{ top.val = l.val + r.val;
  l.env = top.env; r.env = top.env; }

production let:
top:Expr ::= id:String bnd:Expr bod:Expr
{ top.val = bod.val;
  bod.env = addEnv(id, bnd.val, top.env); }

production var:
top:Expr ::= id:String
{ top.val = lookup(top.env, id); }
```

- Abstract tree for ‘`let x = 1 in 2 + x`’
- Arrows illustrate attribute demands

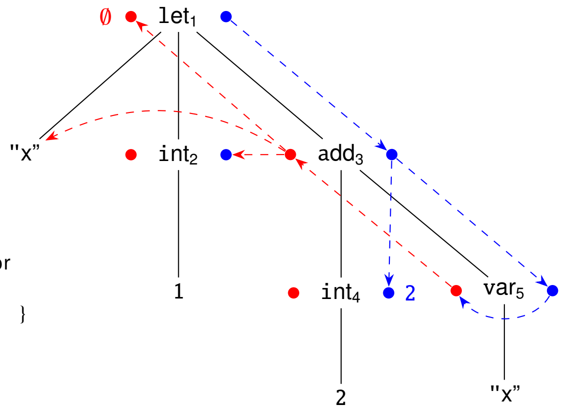


- Animate the tree:
  1. Demand let val
  2. Demand add val
  3. Demand int val
  4. Demand var val
  5. Demand var env
  6. Demand add env
  7. Demand int val
  8. ...
- Show values next to dots when attributes completed

Demand-driven Attribute Evaluation

```
syn attr val:Int; inh attr env:Env;
nt Expr with val, env;

production int:
top:Expr ::= i:Int
{ top.val = i; }

production add:
top:Expr ::= l:Expr r:Expr
{ top.val = l.val + r.val;
  l.env = top.env; r.env =  top.env; }

production let:
top:Expr ::= id:String bnd:Expr bod:Expr
{ top.val = bod.val;
  bod.env = addEnv(id, bnd.val, top.env); }

production var:
top:Expr ::= id:String
{ top.val = lookup(top.env, id); }
```

- Abstract tree for '`let x = 1 in 2 + x`'
- Arrows illustrate attribute demands

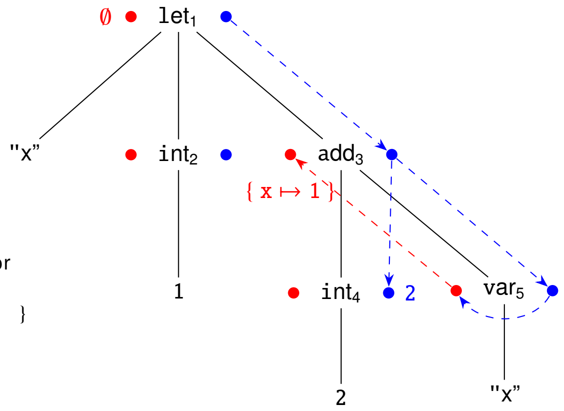


- Animate the tree:
  1. Demand let val
  2. Demand add val
  3. Demand int val
  4. Demand var val
  5. Demand var env
  6. Demand add env
  7. Demand int val
  8. ...
- Show values next to dots when attributes completed

Demand-driven Attribute Evaluation

```
syn attr val:Int; inh attr env:Env;
nt Expr with val, env;

production int:
top:Expr ::= i:Int
{ top.val = i; }

production add:
top:Expr ::= l:Expr r:Expr
{ top.val = l.val + r.val;
  l.env = top.env; r.env =  top.env; }

production let:
top:Expr ::= id:String bnd:Expr bod:Expr
{ top.val = bod.val;
  bod.env = addEnv(id, bnd.val, top.env); }

production var:
top:Expr ::= id:String
{ top.val = lookup(top.env, id); }
```

- Abstract tree for '`let x = 1 in 2 + x`'
- Arrows illustrate attribute demands

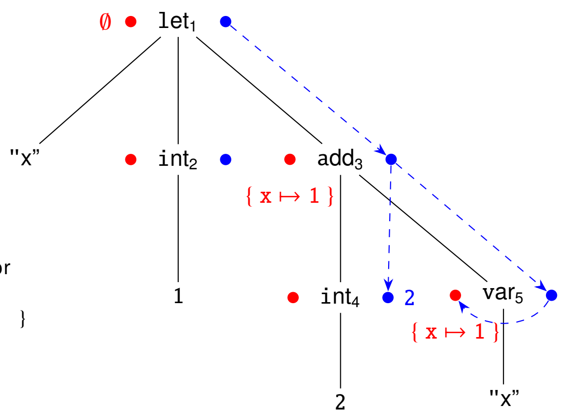


- Animate the tree:
  1. Demand let val
  2. Demand add val
  3. Demand int val
  4. Demand var val
  5. Demand var env
  6. Demand add env
  7. Demand int val
  8. ...
- Show values next to dots when attributes completed

Demand-driven Attribute Evaluation

```
syn attr val:Int; inh attr env:Env;
nt Expr with val, env;

production int:
top:Expr ::= i:Int
{ top.val = i; }

production add:
top:Expr ::= l:Expr r:Expr
{ top.val = l.val + r.val;
  l.env = top.env; r.env =  top.env; }

production let:
top:Expr ::= id:String bnd:Expr bod:Expr
{ top.val = bod.val;
  bod.env = addEnv(id, bnd.val, top.env); }

production var:
top:Expr ::= id:String
{ top.val = lookup(top.env, id); }
```

- Abstract tree for ‘`let x = 1 in 2 + x`’
- Arrows illustrate attribute demands

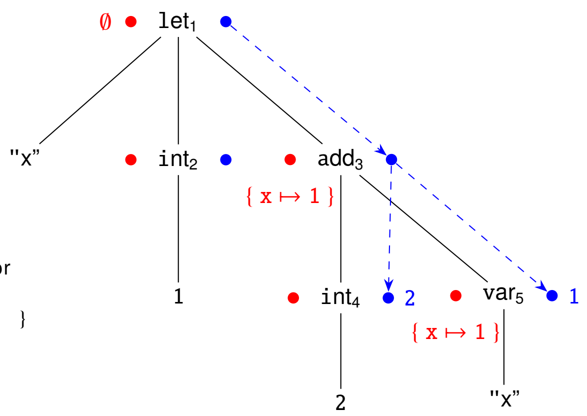


- Animate the tree:
  1. Demand let val
  2. Demand add val
  3. Demand int val
  4. Demand var val
  5. Demand var env
  6. Demand add env
  7. Demand int val
  8. ...
- Show values next to dots when attributes completed

Demand-driven Attribute Evaluation

```
syn attr val:Int; inh attr env:Env;
nt Expr with val, env;

production int:
top:Expr ::= i:Int
{ top.val = i; }

production add:
top:Expr ::= l:Expr r:Expr
{ top.val = l.val + r.val;
  l.env = top.env; r.env =  top.env; }

production let:
top:Expr ::= id:String bnd:Expr bod:Expr
{ top.val = bod.val;
  bod.env = addEnv(id, bnd.val, top.env); }

production var:
top:Expr ::= id:String
{ top.val = lookup(top.env, id); }
```

- Abstract tree for ‘`let x = 1 in 2 + x`’
- Arrows illustrate attribute demands

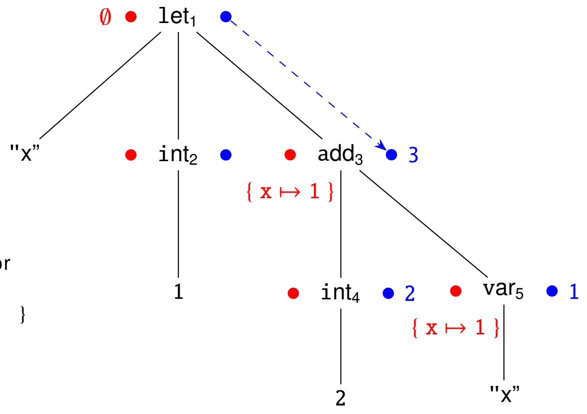


- Animate the tree:
  1. Demand let val
  2. Demand add val
  3. Demand int val
  4. Demand var val
  5. Demand var env
  6. Demand add env
  7. Demand int val
  8. ...
- Show values next to dots when attributes completed

Demand-driven Attribute Evaluation

```
syn attr val:Int; inh attr env:Env;
nt Expr with val, env;

production int:
top:Expr ::= i:Int
{ top.val = i; }

production add:
top:Expr ::= l:Expr r:Expr
{ top.val = l.val + r.val;
  l.env = top.env; r.env =  top.env; }

production let:
top:Expr ::= id:String bnd:Expr bod:Expr
{ top.val = bod.val;
  bod.env = addEnv(id, bnd.val, top.env); }

production var:
top:Expr ::= id:String
{ top.val = lookup(top.env, id); }
```

- Abstract tree for ‘`let x = 1 in 2 + x`’
- Arrows illustrate attribute demands

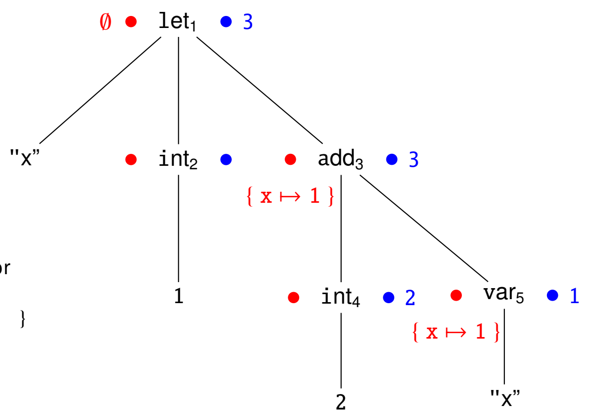


- Animate the tree:
  1. Demand let val
  2. Demand add val
  3. Demand int val
  4. Demand var val
  5. Demand var env
  6. Demand add env
  7. Demand int val
  8. ...
- Show values next to dots when attributes completed

Demand-driven Attribute Evaluation

```
syn attr val:Int; inh attr env:Env;
nt Expr with val, env;

production int:
top:Expr ::= i:Int
{ top.val = i; }

production add:
top:Expr ::= l:Expr r:Expr
{ top.val = l.val + r.val;
  l.env = top.env; r.env = top.env; }

production let:
top:Expr ::= id:String bnd:Expr bod:Expr
{ top.val = bod.val;
  bod.env = addEnv(id, bnd.val, top.env); }

production var:
top:Expr ::= id:String
{ top.val = lookup(top.env, id); }
```

- Abstract tree for ‘`let x = 1 in 2 + x`’
- Arrows illustrate attribute demands

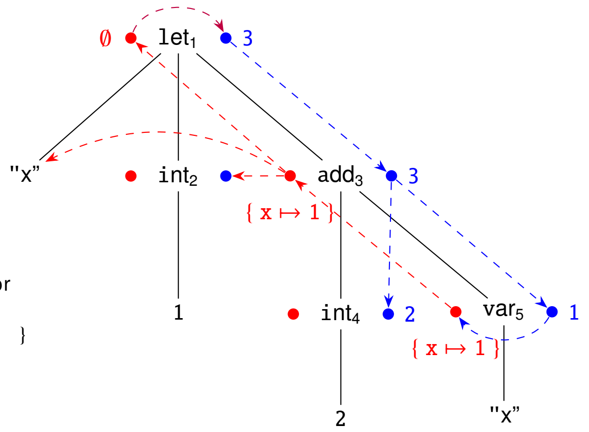


- Animate the tree:
  1. Demand let val
  2. Demand add val
  3. Demand int val
  4. Demand var val
  5. Demand var env
  6. Demand add env
  7. Demand int val
  8. ...
- Show values next to dots when attributes completed

Demand-driven Attribute Evaluation - with a cycle

```
syn attr val:Int; inh attr env:Env;
nt Expr with val, env;

production int:
top:Expr ::= i:Int
{ top.val = i; }

production add:
top:Expr ::= l:Expr r:Expr
{ top.val = l.val + r.val;
  l.env = top.env; r.env = top.env; }

production let:
top:Expr ::= id:String bnd:Expr bod:Expr
{ top.val = bod.val;
  bod.env = addEnv(id, bnd.val, top.env); }

production var:
top:Expr ::= id:String
{ top.val = lookup(top.env, id); }
```

- Abstract tree for 'let x = 1 in 2 + x'
- Arrows illustrate attribute demands



- Animate the tree:
  1. Demand let val
  2. Demand add val
  3. Demand int val
  4. Demand var val
  5. Demand var env
  6. Demand add env
  7. Demand int val
  8. ...
- Show values next to dots when attributes completed

Statix to Attribute Grammars: Translation and Correspondence

- We translate Statix to attribute grammars to show faithfulnes of our approach
- Statix constraints translated to equations
- AG trace yields a Statix solving order
- Corresponding Statix and the AG specifications give the same results

RAG	Statix
`root.ok = true`	All constraints solved
`root.ok = false`	Constraints unsatisfiable
Cycle on an attribute	Constraint solving “stuck”

- See Luke’s SLE 2025 paper.

Statix syntax predicate:

```
@syntax dcl(@inh s: scope, @inh sn: scope,
            @inh sm: scope, d: dcl) :- d match
{ mod(id: string, ds: dcls) -> {sm': scope}
    new sm' -> id,
    sm -[MOD]-> sm', sm' -[LEX]-> s,
    dcls(sm', sm', ds) }
```

Corresponding RAG definitions:

```
inh attr s:Scope, sn:Scope, sm:Scope;
syn attr MOD_sm:[Scope], ...;
nt Dcl with s, sn, sm, MOD_sm, ...;
production mod:
top:Dcl ::= id:String ds:Dcls | sm':Scope {
  sm' = mkScopeDcl(id);
  sm'.LEX = [top.s]; sm'.VAR = ds.VAR_s;
  sm'.MOD = ds.MOD_s; sm'.IMP = ds.IMP_s;
  top.MOD_sm <- [sm'];
  ds.s = sm'; ds.sm = sm';
  top.ok <- ds.ok; }
```

Choose your adventure

1. Circular attributes to get "unstuck"
2. Integrating RAGs and Scope Graphs in one framework

a.k.a. Theres other stuff to ask about if you run out of legitimate questions.

"Self-influencing" imports

- First, without self-influencing imports.
  - Rust program:

```
1  pub mod foo {
2    pub static x:u8 = 1;
3    pub mod bar {
4      pub static x:u8 = 1;
5    }
6  }
7  pub mod test {
8    use super::*;
9    use foo::*;
10   use bar::*;
11   pub static y:u8 = x;
12 }
```

  - Name resolution results:

1. foo ↦ foo on line 1
2. bar ↦ bar on line 3
3. x ↦ x on line 2, x on line 4

- Discuss here what a "standard" import resolution looks like. i.e. we need to resolve foo before bar, and bar before x.
- This feels like a "natural" way of resolving imports. Intuitive. The resolution of each name can be run to completion, and there is a clear order in which we should resolve, such that resolutions for all names are found.
- If possible, draw arrows showing resolutions.

“Self-influencing” imports

- An example with self-influencing import edges
  - Rust program:

```
1  pub mod foo {
2    pub static x:u8 = 1;
3    pub mod foo {
4      pub static x:u8 = 1;
5    }
6  }
7  pub mod test {
8    use super::*;
9    use foo::*;
10   pub static y:u8 = x;
11
12 }
```

- Name resolution results:

1. `foo` ↦ `foo` on line 1, `foo` on line 2
2. `x` ↦ none, `foo` is ambiguous

- Resolution to the outer `foo` is used to resolve to the inner one

- Rust gets more “interesting”.
- Now introduce the self-influencing import behavior. This is an ambiguous program in Rust because the import of foo can be used to resolve itself, thereby finding the two foo module declarations.
- In scope graphs we’ve been running queries to a result once, but this seems to need multiple iterations of a query.
- If possible, draw arrows showing resolutions.

LM, A Simple Toy language

- We use a toy language LM, a sandbox for different name resolution semantics
  - *e.g.* sequential, parallel, unordered, recursive, Rust-like

- Rust program:

```
pub mod foo {
  pub statix x:u8 = 2;
  pub mod foo {
    pub static x:u8 = 1;
  }
}
pub mod test {
  use super::*;
  use foo::*;
  pub static y:u8 = x;
}
```

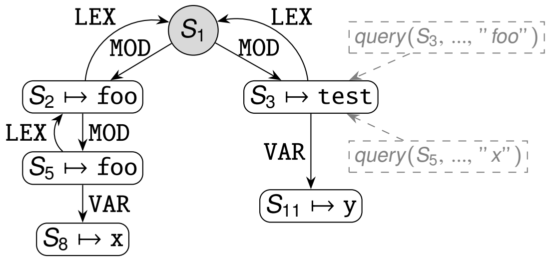
- LM translation:

```
module foo {
  def x = 1;
  module foo {
    def x = 1;
  }
}
module test {
  import foo;
  def y = x;

}
```

Recap: Self-influencing Imports

- Self-influencing name resolution: one resolution of a name may influence further resolutions of the same name
- Name resolution is circularly defined

```
module foo {
  module foo {
    def x = 1;
  }
}

module test {
  import foo;
  def y = x;
}
```

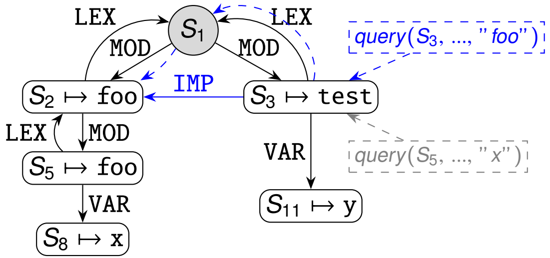


- Discuss again the example which exhibits the interesting cyclic behavior. etc.
- animate resolution of foo to S_2 and the resulting available resolution to the inner.
- Import foo resolves to the outer foo module, then also the inner
- We shift our focus to a *recursive* import resolution semantics which gives the inner module as a resolution for the import foo

Recap: Self-influencing Imports

- Self-influencing name resolution: one resolution of a name may influence further resolutions of the same name
- Name resolution is circularly defined

```
module foo {
  module foo {
    def x = 1;
  }
}

module test {
  import foo;
  def y = x;
}
```

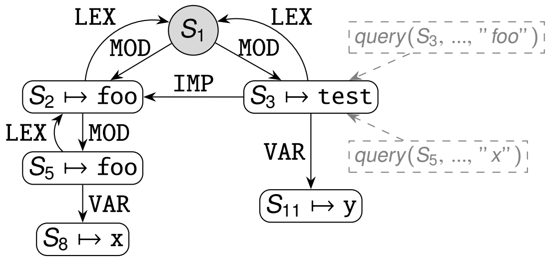


- Discuss again the example which exhibits the interesting cyclic behavior. etc.
- animate resolution of foo to S_2 and the resulting available resolution to the inner.
- Import foo resolves to the outer foo module, then also the inner
- We shift our focus to a *recursive* import resolution semantics which gives the inner module as a resolution for the import foo

Recap: Self-influencing Imports

- Self-influencing name resolution: one resolution of a name may influence further resolutions of the same name
- Name resolution is circularly defined

```
module foo {
  module foo {
    def x = 1;
  }
}

module test {
  import foo;
  def y = x;
}
```

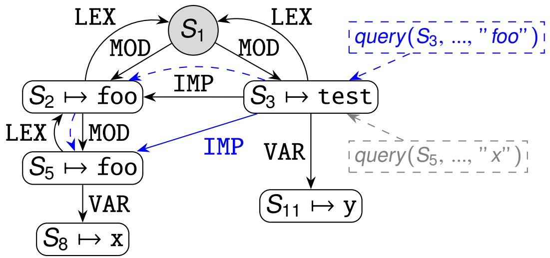


- Discuss again the example which exhibits the interesting cyclic behavior. etc.
- animate resolution of foo to S_2 and the resulting available resolution to the inner.
- Import foo resolves to the outer foo module, then also the inner
- We shift our focus to a *recursive* import resolution semantics which gives the inner module as a resolution for the import foo

Recap: Self-influencing Imports

- Self-influencing name resolution: one resolution of a name may influence further resolutions of the same name
- Name resolution is circularly defined

```
module foo {
  module foo {
    def x = 1;
  }
}

module test {
  import foo;
  def y = x;
}
```

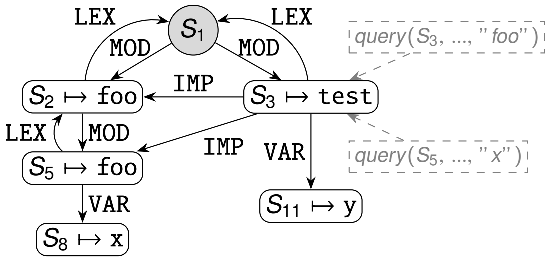


- Discuss again the example which exhibits the interesting cyclic behavior. etc.
- animate resolution of foo to S_2 and the resulting available resolution to the inner.
- Import foo resolves to the outer foo module, then also the inner
- We shift our focus to a *recursive* import resolution semantics which gives the inner module as a resolution for the import foo

Recap: Self-influencing Imports

- Self-influencing name resolution: one resolution of a name may influence further resolutions of the same name
- Name resolution is circularly defined

```
module foo {
  module foo {
    def x = 1;
  }
}

module test {
  import foo;
  def y = x;
}
```

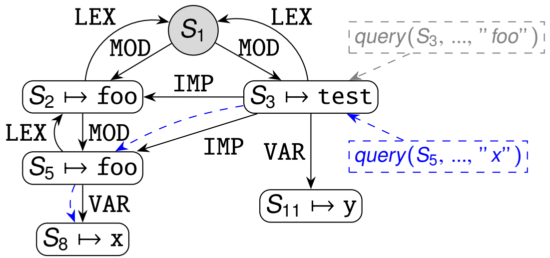


- Discuss again the example which exhibits the interesting cyclic behavior. etc.
- animate resolution of foo to S_2 and the resulting available resolution to the inner.
- Import foo resolves to the outer foo module, then also the inner
- We shift our focus to a *recursive* import resolution semantics which gives the inner module as a resolution for the import foo

Recap: Self-influencing Imports

- Self-influencing name resolution: one resolution of a name may influence further resolutions of the same name
- Name resolution is circularly defined

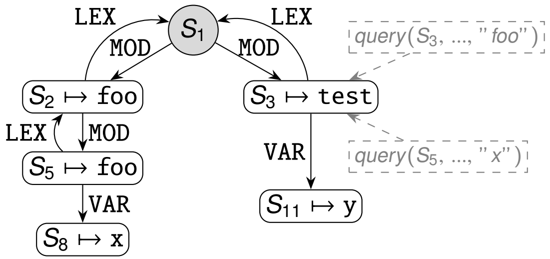
```
module foo {
  module foo {
    def x = 1;
  }
}

module test {
  import foo;
  def y = x;
}
```



- Discuss again the example which exhibits the interesting cyclic behavior. etc.
- animate resolution of foo to S_2 and the resulting available resolution to the inner.
- Import foo resolves to the outer foo module, then also the inner
- We shift our focus to a *recursive* import resolution semantics which gives the inner module as a resolution for the import foo

Fixed-point Computation of Self-influencing Imports

- Self-influencing imports are implemented as a fixed-point computation
- Each iteration uses the `IMP` edges discovered by the previous
- Computation ends when no more module declarations are found
- First iteration yields the magenta edge, the second uses it to yield the teal edge
- Computes *candidate* edges for recursive, unordered and Rust-like imports

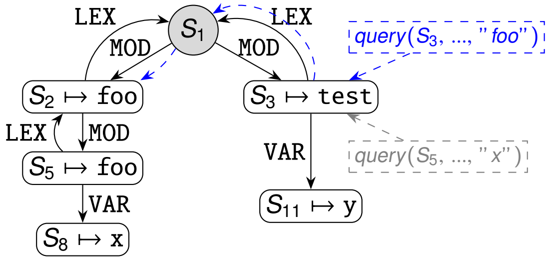


Iter.	Input `IMP` targets	Output `IMP` targets
1		S_2
2	S_2	S_2, S_5
3	S_2, S_5	S_2, S_5

- Animate the fixed-point resolution of foo from S_3. First comes the IMP edge to the outer foo, then to the inner. then do another iteration where no more foo modules are found.

Fixed-point Computation of Self-influencing Imports

- Self-influencing imports are implemented as a fixed-point computation
- Each iteration uses the IMP edges discovered by the previous
- Computation ends when no more module declarations are found
- First iteration yields the magenta edge, the second uses it to yield the teal edge
- Computes *candidate* edges for recursive, unordered and Rust-like imports

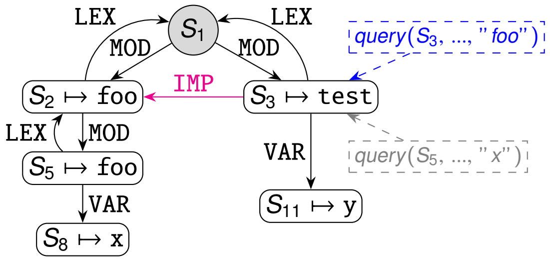


Iter.	Input IMP targets	Output IMP targets
1		S_2
2	S_2	S_2, S_5
3	S_2, S_5	S_2, S_5

- Animate the fixed-point resolution of foo from S_3. First comes the IMP edge to the outer foo, then to the inner. then do another iteration where no more foo modules are found.

Fixed-point Computation of Self-influencing Imports

- Self-influencing imports are implemented as a fixed-point computation
- Each iteration uses the `IMP` edges discovered by the previous
- Computation ends when no more module declarations are found
- First iteration yields the magenta edge, the second uses it to yield the teal edge
- Computes *candidate* edges for recursive, unordered and Rust-like imports

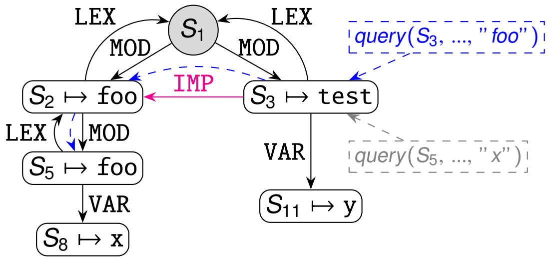


Iter.	Input `IMP` targets	Output `IMP` targets
1		S_2
2	S_2	S_2, S_5
3	S_2, S_5	S_2, S_5

- Animate the fixed-point resolution of foo from S_3. First comes the IMP edge to the outer foo, then to the inner. then do another iteration where no more foo modules are found.

Fixed-point Computation of Self-influencing Imports

- Self-influencing imports are implemented as a fixed-point computation
- Each iteration uses the `IMP` edges discovered by the previous
- Computation ends when no more module declarations are found
- First iteration yields the magenta edge, the second uses it to yield the teal edge
- Computes *candidate* edges for recursive, unordered and Rust-like imports

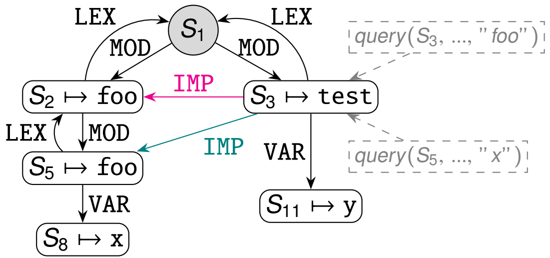


Iter.	Input `IMP` targets	Output `IMP` targets
1		S_2
2	S_2	S_2, S_5
3	S_2, S_5	S_2, S_5

- Animate the fixed-point resolution of foo from S_3. First comes the IMP edge to the outer foo, then to the inner. then do another iteration where no more foo modules are found.

Fixed-point Computation of Self-influencing Imports

- Self-influencing imports are implemented as a fixed-point computation
- Each iteration uses the `IMP` edges discovered by the previous
- Computation ends when no more module declarations are found
- First iteration yields the magenta edge, the second uses it to yield the teal edge
- Computes *candidate* edges for recursive, unordered and Rust-like imports



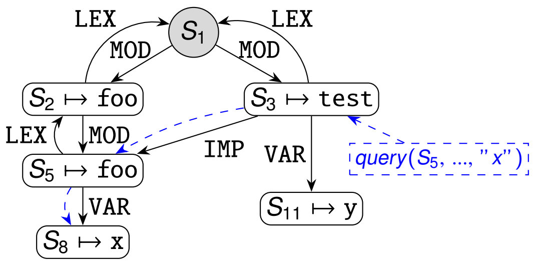
Iter.	Input `IMP` targets	Output `IMP` targets
1		S_2
2	S_2	S_2, S_5
3	S_2, S_5	S_2, S_5

- Animate the fixed-point resolution of foo from S_3. First comes the IMP edge to the outer foo, then to the inner. then do another iteration where no more foo modules are found.

Filtering of collected IMP

- Fixpoint result: Many candidate IMP edges to all found module declarations
- Same fixpoint process for several notions of import semantics:
  - recursive, unordered and Rust-like imports
- To distinguish these semantics: filter set of candidate IMP edges
- Edges remaining after filtering: *persistent* edges

- Resulting persistent IMP edge and resolution of "x" in our example:



- Intended to show the result of the filtering from the previous slide.
- Wasn't sure how to animate the enumerate block in the previous slide disappearing and this scope graph replacing it, so have them as separate slides!

Recall Statix to Attribute Grammars: Translation and Correspondence

- We translate Statix to attribute grammars to show faithfulnes of our approach
- Statix constraints translated to equations
- AG trace yields a Statix solving order
- Corresponding Statix and the AG specifications give the same results

RAG	Statix
`root.ok = true`	All constraints solved
`root.ok = false`	Constraints unsatisfiable
Cycle on an attribute	Constraint solving "stuck"

- See Luke's SLE 2025 paper.

Statix syntax predicate:

```
@syntax dcl(@inh s: scope, @inh sn: scope,
            @inh sm: scope, d: dcl) :- d match
{ mod(id: string, ds: dcls) -> {sm': scope}
    new sm' -> id,
    sm -[MOD]-> sm', sm' -[LEX]-> s,
    dcls(sm', sm', ds) }
```

Corresponding RAG definitions:

```
inh attr s:Scope, sn:Scope, sm:Scope;
syn attr MOD_sm:[Scope], ...;
nt Dcl with s, sn, sm, MOD_sm, ...;
production mod:
top:Dcl ::= id:String ds:Dcls | sm':Scope {
  sm' = mkScopeDcl(id);
  sm'.LEX = [top.s];  sm'.VAR = ds.VAR_s;
  sm'.MOD = ds.MOD_s; sm'.IMP = ds.IMP_s;
  top.MOD_sm <- [sm'];
  ds.s = sm'; ds.sm = sm';
  top.ok <- ds.ok; }
```

Circular Attribute Grammars

- Compute values for circular attribute definitions
- Use fixed-point computation from an initial value
- All attributes in a cycle computed at once
- Each equation involved may be evaluated many times
- Have been implemented in *e.g.* JastAdd, an AG system of Görel Hedin et al.
- We can use circular attributes to compute self-influencing imports.

Circular Attribute Definitions for Scope Graphs

- Import queries demand `IMPc` (candidate) edges in source scope, `IMP` (persistent) elsewhere
- The set of inherited candidate import edges for scopes (`IMPc`) is declared as circular
- Circular attributes IMPc and s_IMPc: Collect/distribute candidate edges on every fixpoint iteration
- Filter function for recursive imports

```
inh circ attr IMPc:[Path] init [];
nt Scope with !⋆name, LEX, VAR, IMP,⋆! IMPc;

inh attr !⋆s:Scope;⋆!
syn attr !⋆s_VAR:[Scope], s_MOD:[Scope], ...;⋆!
syn circ attr s_IMPc:[Path] init [];

nt Stmt with !⋆ok, s, s_MOD, s_VAR,⋆! s_IMPc;

prod mod: top:Stmt ::= x:String ds:Stmt {
  !⋆sm = mkScopeDcl(x); ds.s = sm;⋆!
  !⋆m.LEX = [top.s]; sm.VAR = ds.s_VAR;⋆!
  !⋆sm.MOD = ds.s_MOD;⋆! sm.IMPc = ds.s_IMPc;
  sm.IMP = filter-recursive(sm.IMPc);
  !⋆top.sm_MOD = [sm]; top.s_VAR = [];⋆!
  top.s_IMPc = []; ... }

prod imp: top:Stmt ::= i:String {
  top.s_IMPc = query(top.s, LEX⋆IMP?MOD, i);
  !⋆top.s_MOD = []; top.s_VAR = [];⋆! ...}
```


these attributes collect and distribute edge targets.

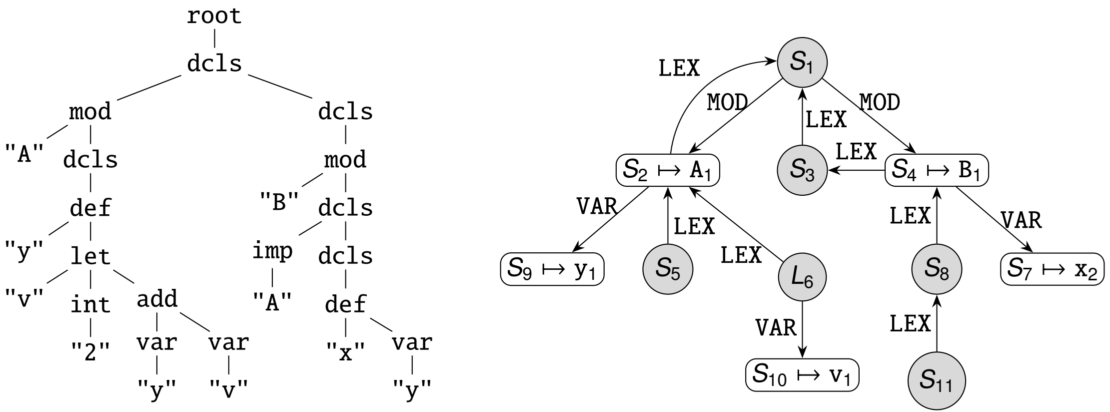
Integrating RAGs and Scope Graphs

- The translation of Statix to RAGs above is monolithic.
  It takes a complete Statix spec and generates an RAG spec.
- Could we have a more fine grained integration?
  Can we write Statix-like specifications next to equations in AG productions?
  Specifically
  - scope assertions
  - edge assertions
  - resolution queries
- Can the AST and the scope graph cross reference one another?

Integrating the data structures

References attributes between the syntax tree and the scope graph.

- Reference attributes associate tree nodes with scope graph nodes

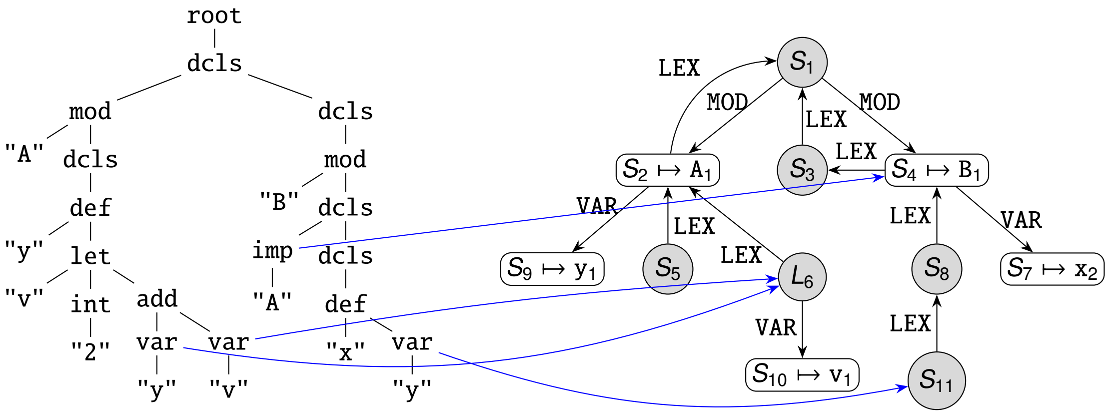


- Blue: reference attributes from var refs to lookup scopes
- Red: reference attributes from SG decls to AST decls
- Orange: resolution path
- Violet: reference edges in tree
- Teal: focus in new IMP edge, turns to black after
- Gray: "out of focus"

Integrating the data structures

References attributes between the syntax tree and the scope graph.

- Some reference attributes identify the scope to resolve a name in

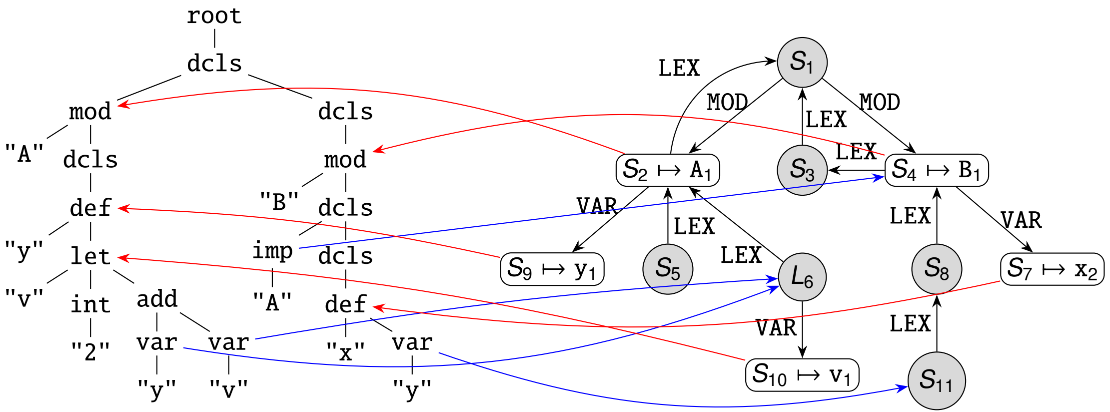


- Blue: reference attributes from var refs to lookup scopes
- Red: reference attributes from SG decls to AST decls
- Orange: resolution path
- Violet: reference edges in tree
- Teal: focus in new IMP edge, turns to black after
- Gray: “out of focus”

Integrating the data structures

References attributes between the syntax tree and the scope graph.

- Others associate a graph declaration with its corresponding tree declaration

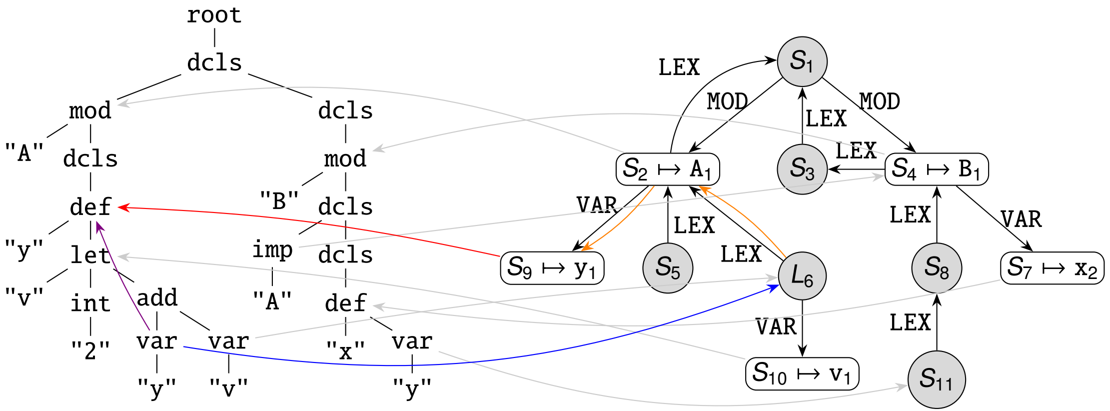


- Blue: reference attributes from var refs to lookup scopes
- Red: reference attributes from SG decls to AST decls
- Orange: resolution path
- Violet: reference edges in tree
- Teal: focus in new IMP edge, turns to black after
- Gray: "out of focus"

Integrating the data structures

References attributes between the syntax tree and the scope graph.

- Resolution of the first reference "y"

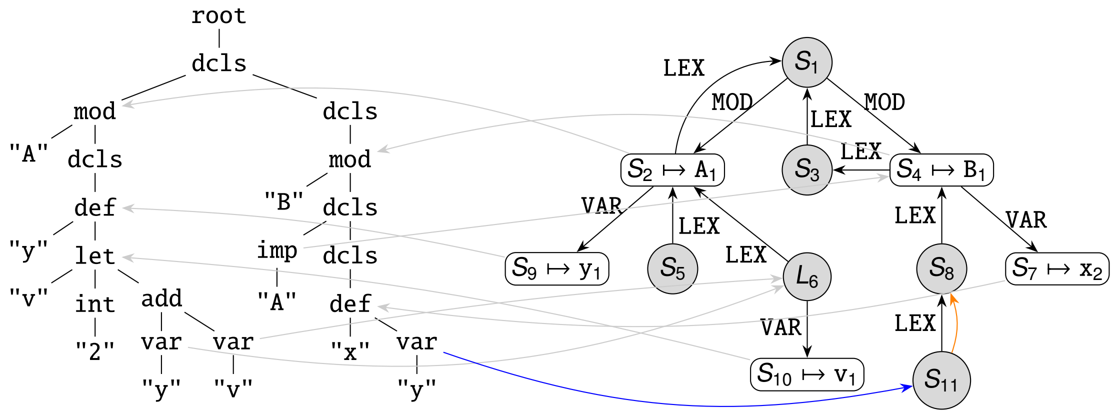


- Blue: reference attributes from var refs to lookup scopes
- Red: reference attributes from SG decls to AST decls
- Orange: resolution path
- Violet: reference edges in tree
- Teal: focus in new IMP edge, turns to black after
- Gray: "out of focus"

Integrating the data structures

References attributes between the syntax tree and the scope graph.

- Beginning resolution of second "y"

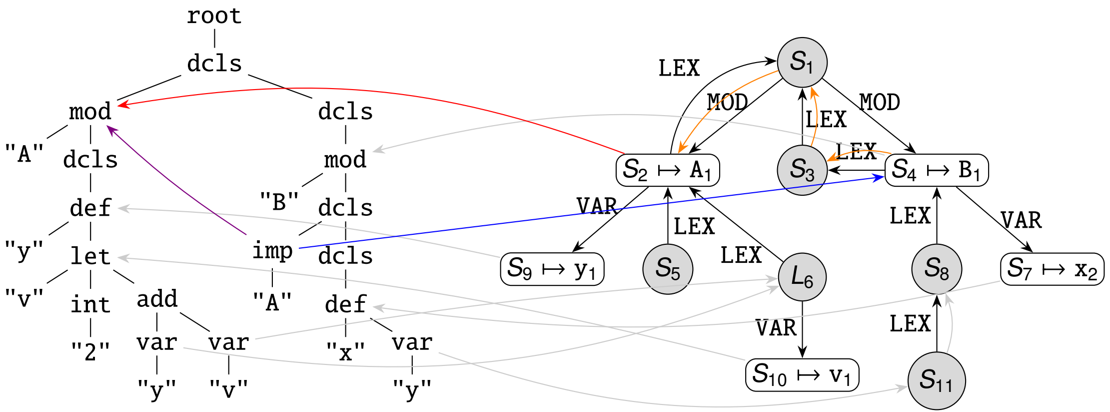


- Blue: reference attributes from var refs to lookup scopes
- Red: reference attributes from SG decls to AST decls
- Orange: resolution path
- Violet: reference edges in tree
- Teal: focus in new IMP edge, turns to black after
- Gray: "out of focus"

Integrating the data structures

References attributes between the syntax tree and the scope graph.

- Resolving import reference "A"

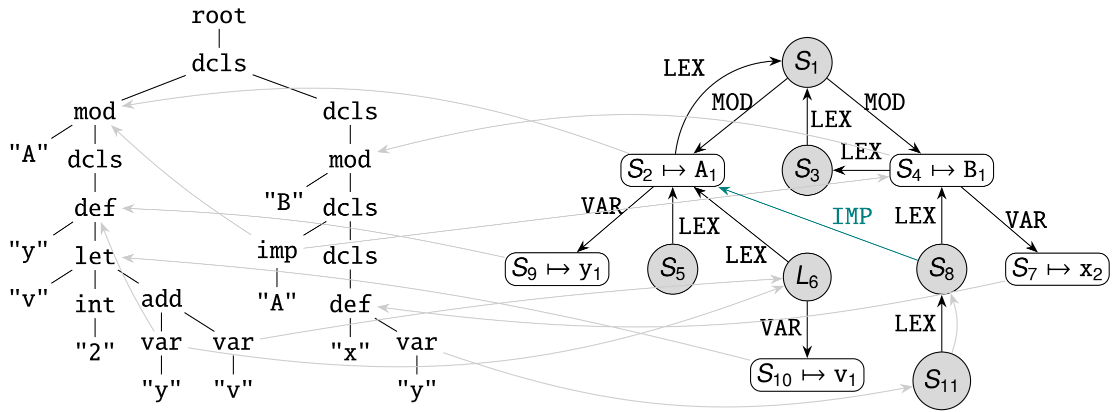


- Blue: reference attributes from var refs to lookup scopes
- Red: reference attributes from SG decls to AST decls
- Orange: resolution path
- Violet: reference edges in tree
- Teal: focus in new IMP edge, turns to black after
- Gray: “out of focus”

Integrating the data structures

References attributes between the syntax tree and the scope graph.

- Resulting `IMP` edge in the scope graph

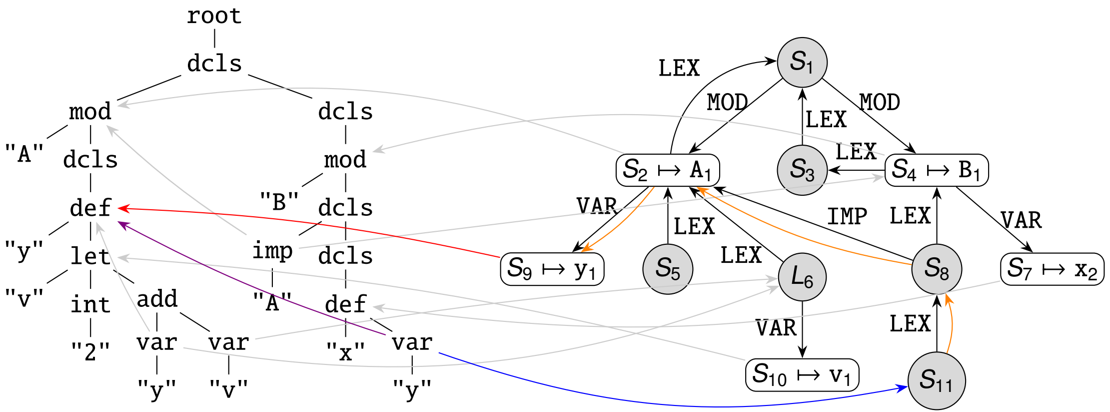


- Blue: reference attributes from var refs to lookup scopes
- Red: reference attributes from SG decls to AST decls
- Orange: resolution path
- Violet: reference edges in tree
- Teal: focus in new IMP edge, turns to black after
- Gray: "out of focus"

Integrating the data structures

References attributes between the syntax tree and the scope graph.

- Continuing resolution of second "y"

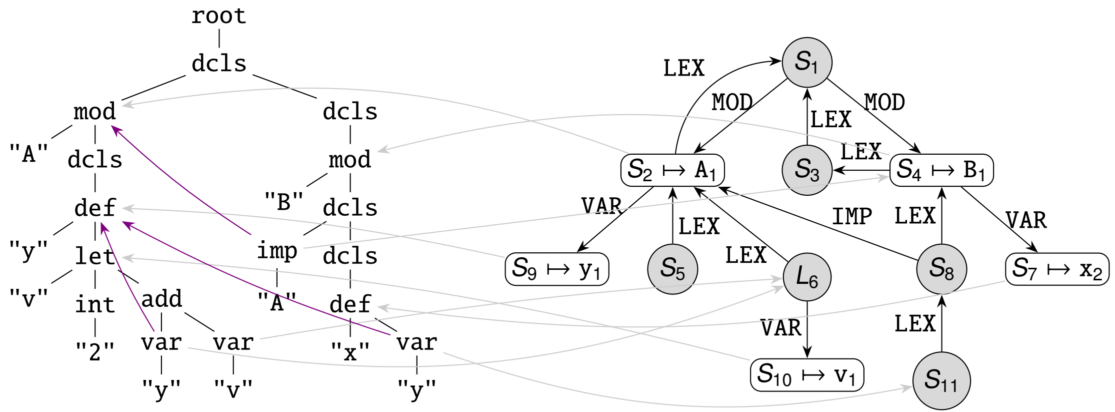


- Blue: reference attributes from var refs to lookup scopes
- Red: reference attributes from SG decls to AST decls
- Orange: resolution path
- Violet: reference edges in tree
- Teal: focus in new IMP edge, turns to black after
- Gray: "out of focus"

Integrating the data structures

References attributes between the syntax tree and the scope graph.

- Reference attribute edges in the AST after these resolutions



- Blue: reference attributes from var refs to lookup scopes
- Red: reference attributes from SG decls to AST decls
- Orange: resolution path
- Violet: reference edges in tree
- Teal: focus in new IMP edge, turns to black after
- Gray: "out of focus"

Integrating the specifications

A look at some sample (speculative) specifications...

Thank you for your attention.

Questions?