Automatically introducing templates in Domain-Specific Languages

Andrey Breslav

University of Tartu / ITMO (St. Petersburg)

February 06, 2010

Outline

- **Basics**
  - What is a template?
  - How is a language described?
  - How to plug templates in?
  - Intermediate summary
- **Some notes on correctness**
  - Issues with names
  - Constraints for name resolution
  - Open issues
- **Conclusion**

MyToy Language

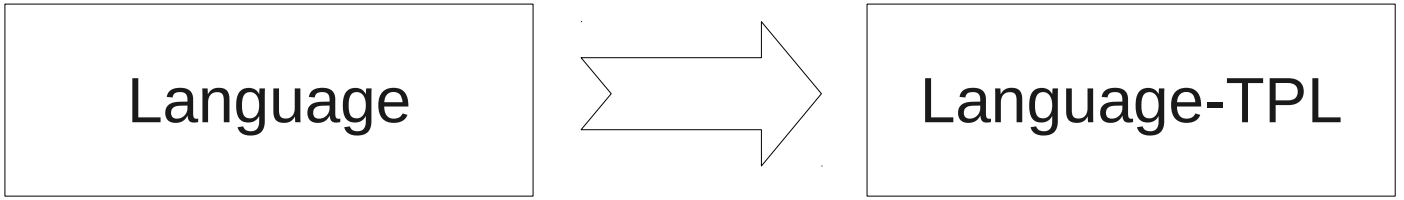
```
VAR x = 2; VAR y = 5;
VAR xy = 0;
VAR trigger = 0;
COMEFROM trigger = 1;
   xy = xy + y;
   x = x - 1;
   trigger = 1;
COMEFROM x = 0;
```

```
VAR x = 2; VAR y = 5;
VAR xy = 0;
$UNTIL <x = 0> <
   xy = xy + y;
   x = x - 1;
>;
```

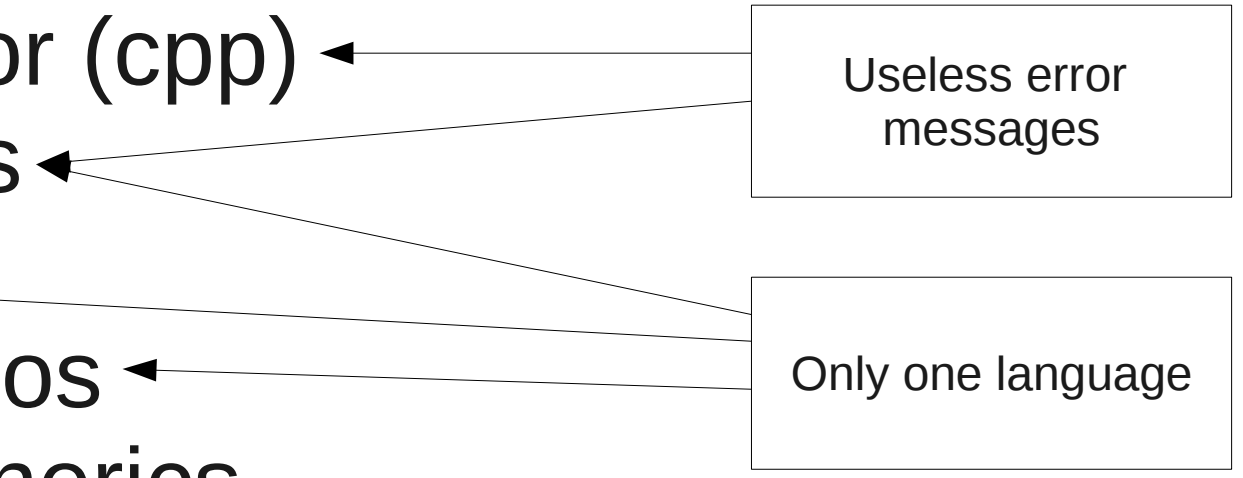
MyToy-TPL Language

```
template $UNTIL <condition> <body> {
    VAR trigger = 0;
    comefrom trigger = 1;
      <body>
      trigger = 1;
    comefrom <condition>;
}
```

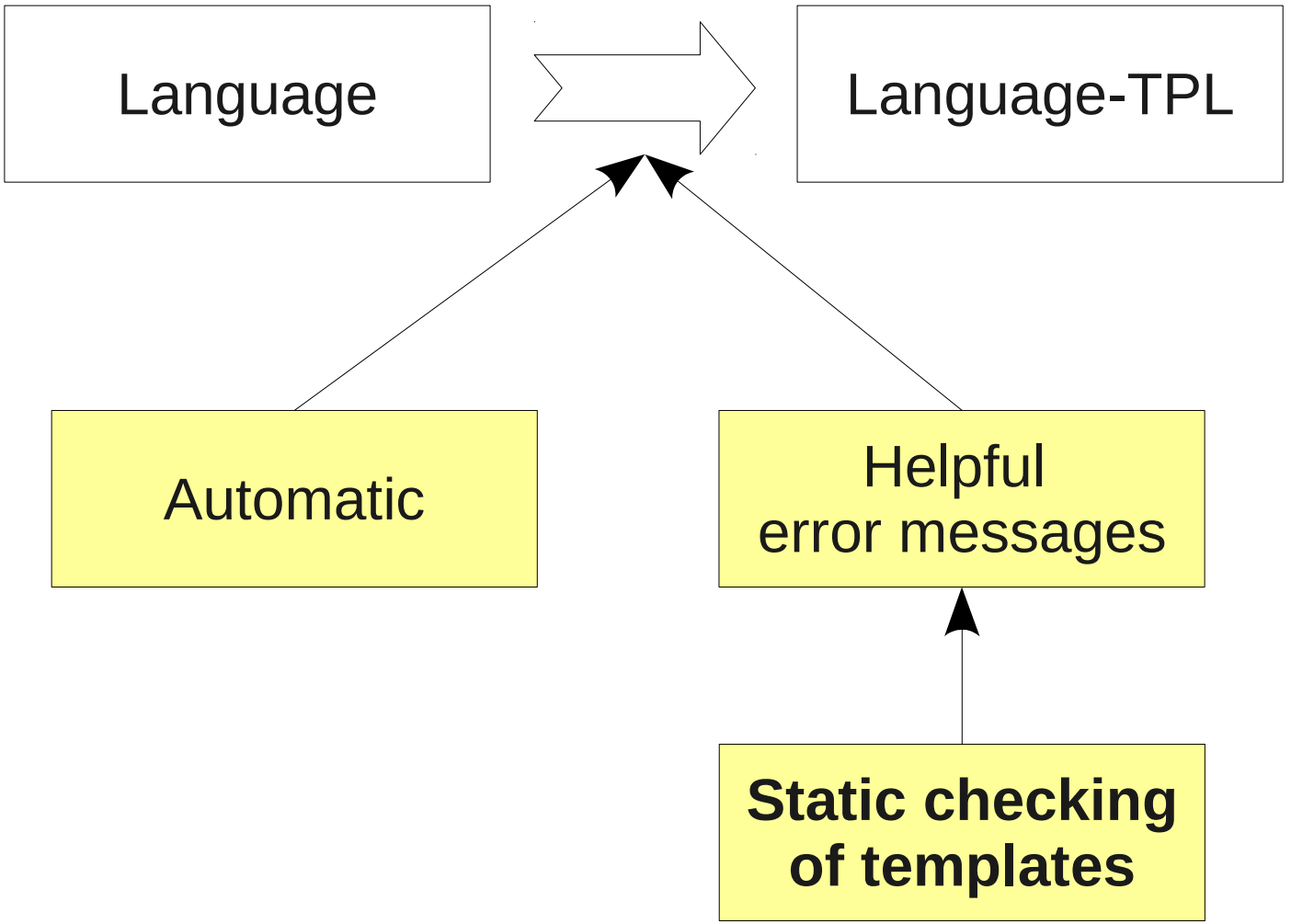
Our Goal



- C preprocessor (cpp)
- C++ templates
- Lisp
- Nemerle macros
- **NOT** Java Generics

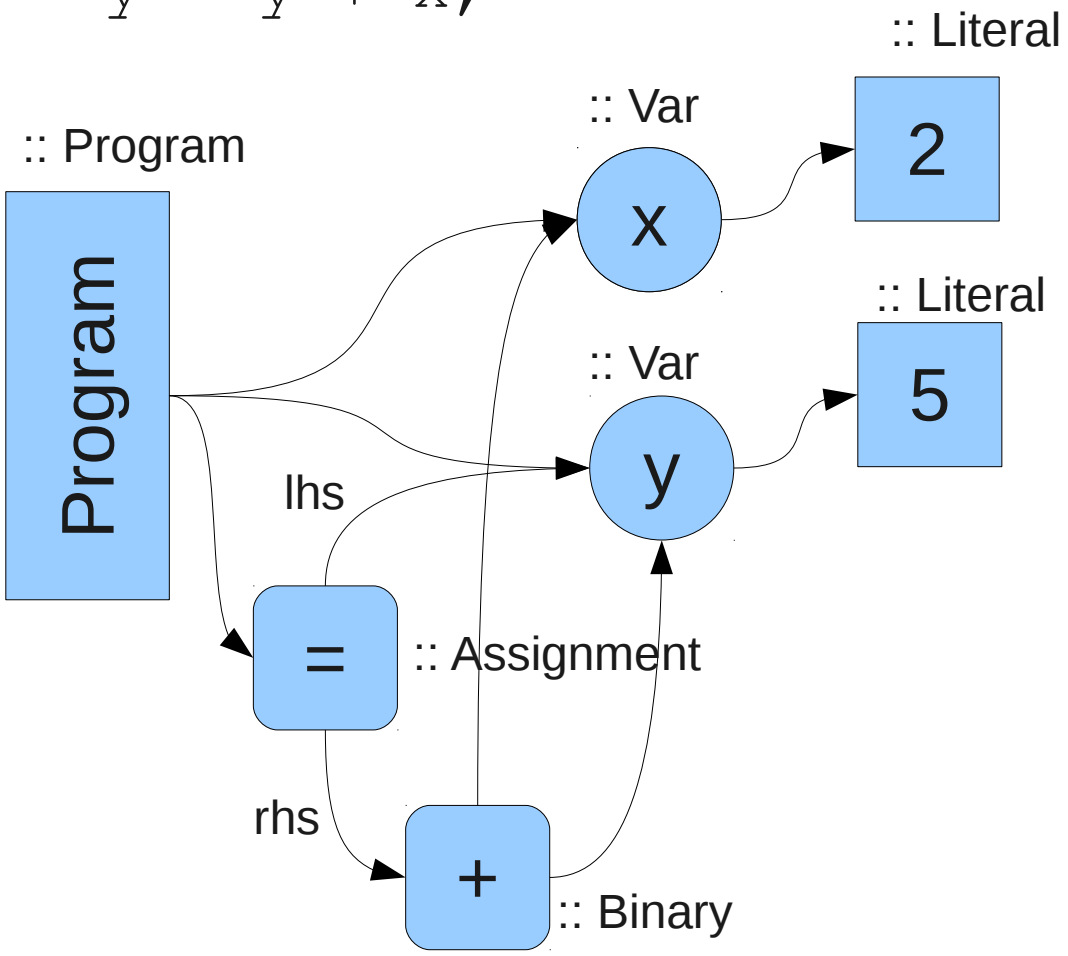


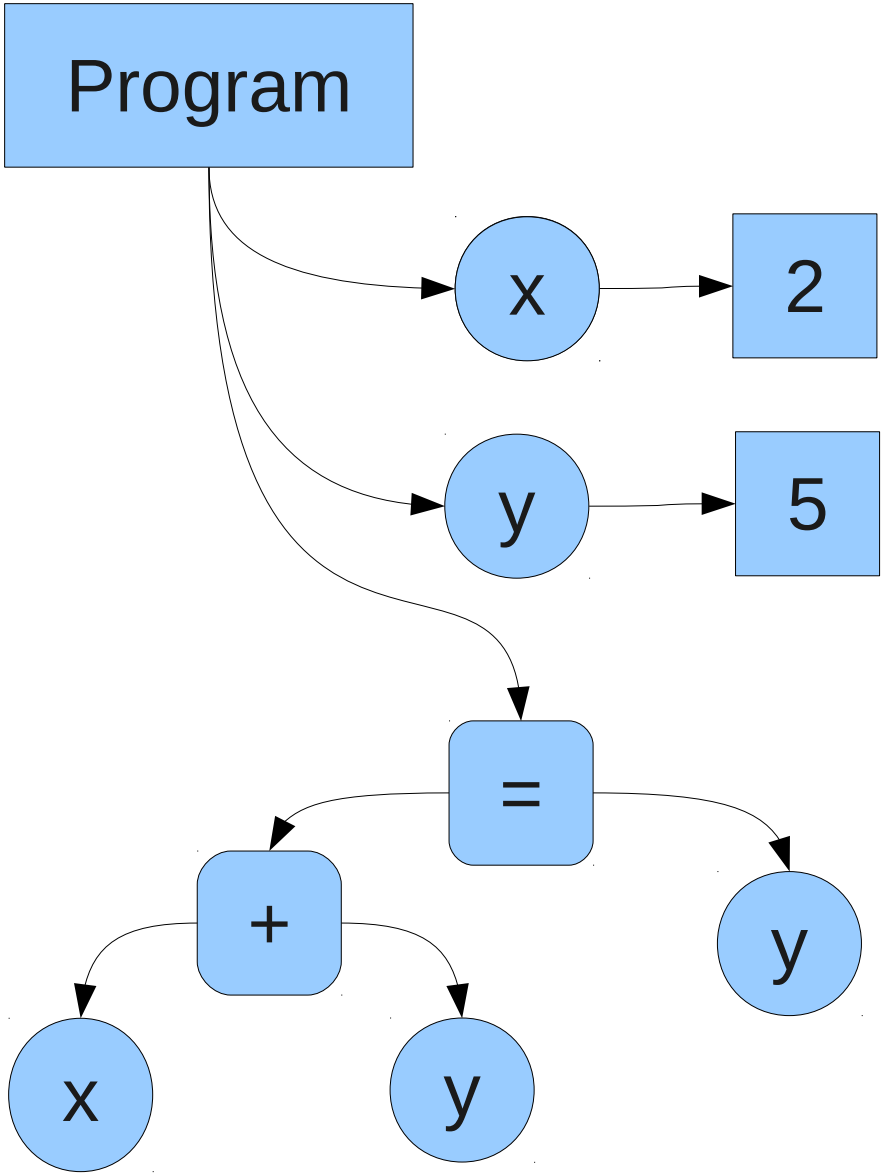
Our Goal


Language
Language-TPL
Automatic
Helpful
error messages
Static checking
of templates

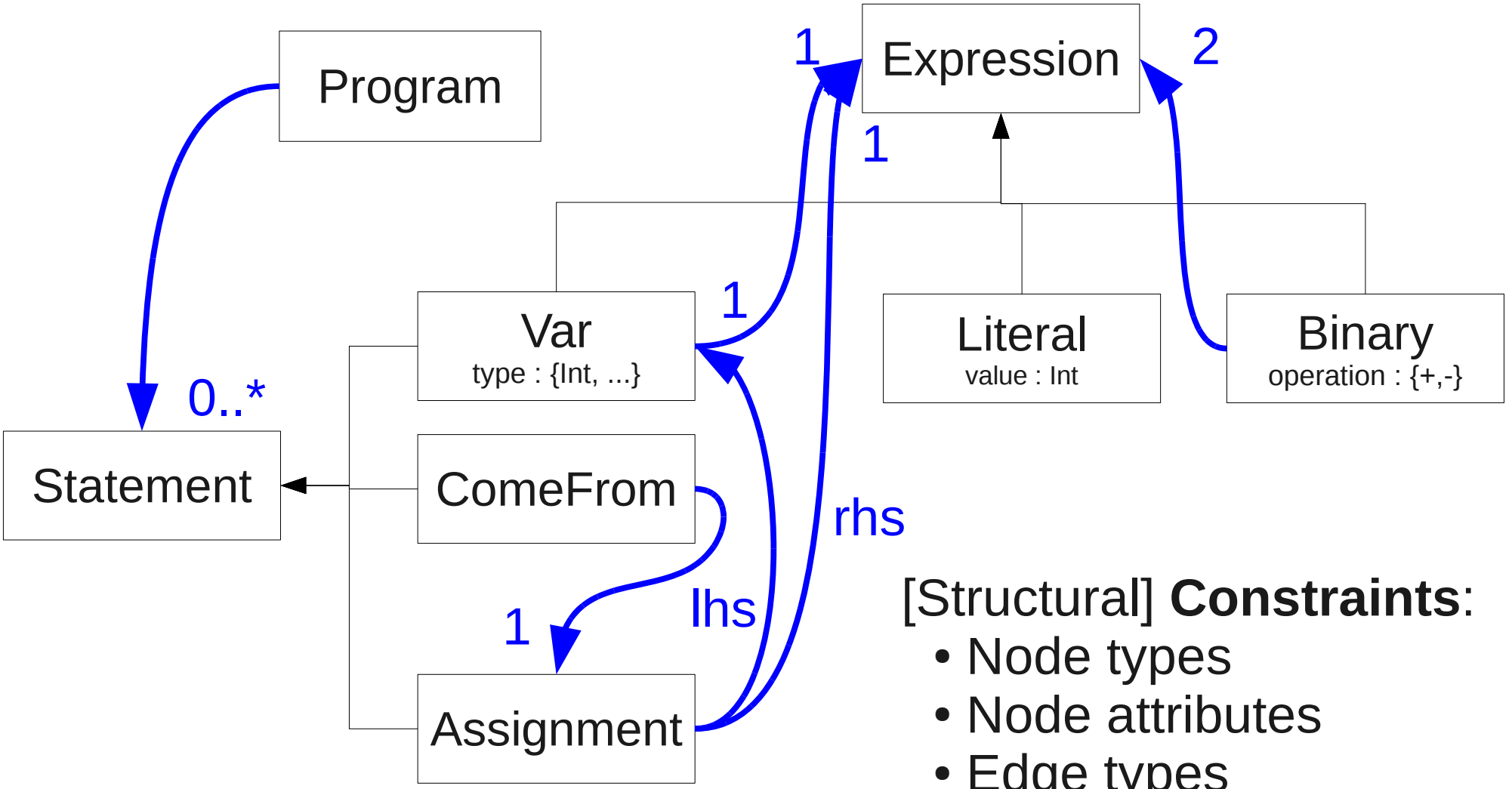
Alternative Views of a Program

```
VAR x = 2;
VAR y = 5;
y = y + x;
```





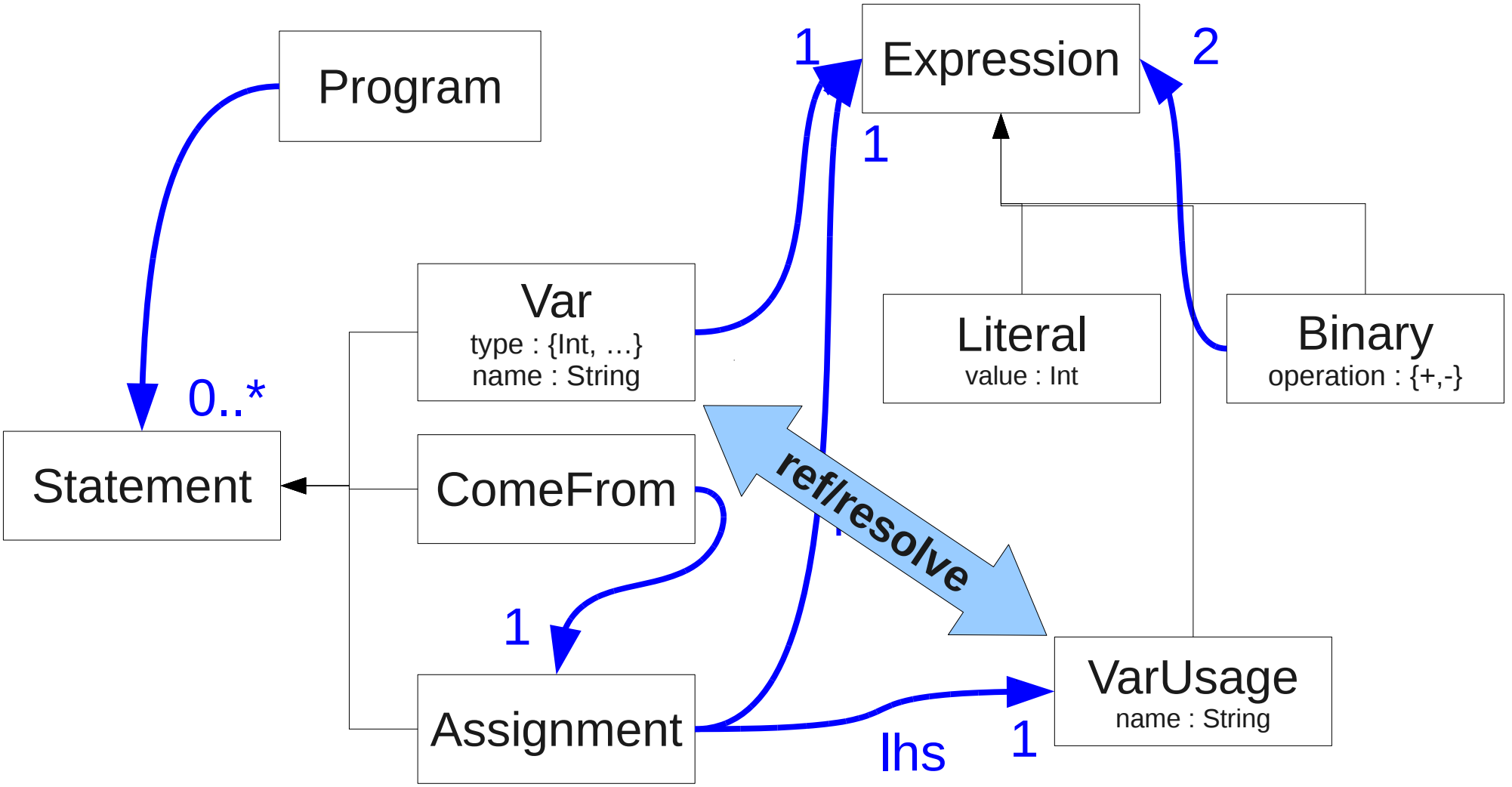
Describing graph structure



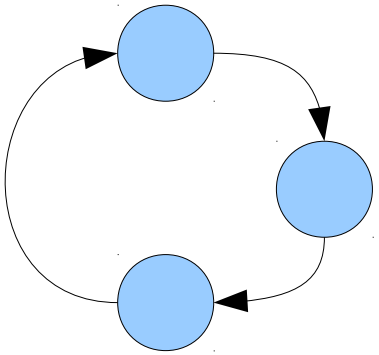
[Structural] **Constraints**:

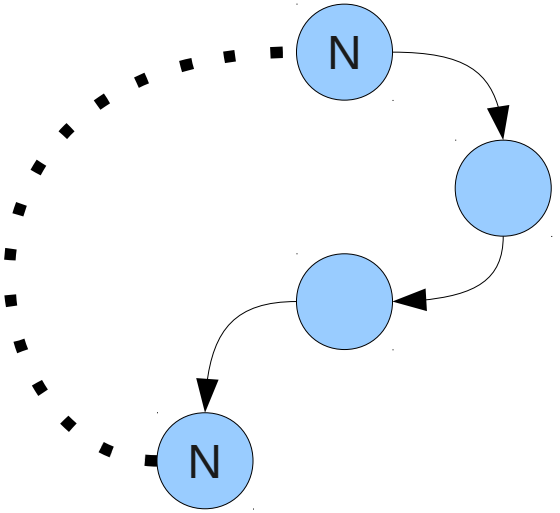
- Node types
- Node attributes
- Edge types
- Edge multiplicities
- Typing of edge ends

Describing tree structure



Templates for Graph





Graph

Serialize

Deserialize

AST

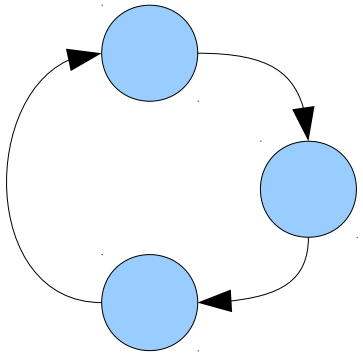
Templates

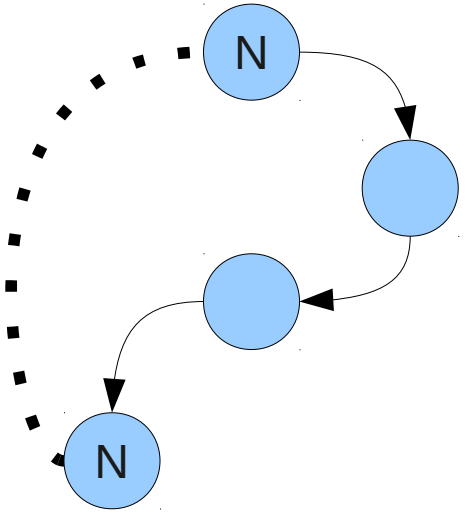
Serialize

Deserialize

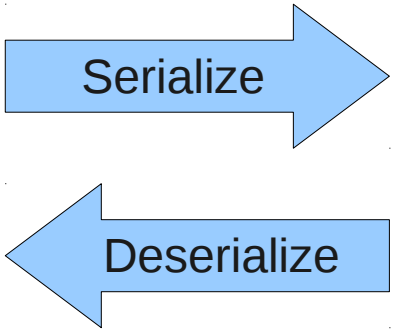
Templates AST

Templates for AST

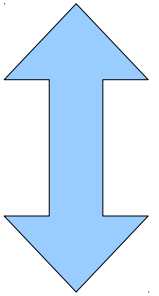




Graph

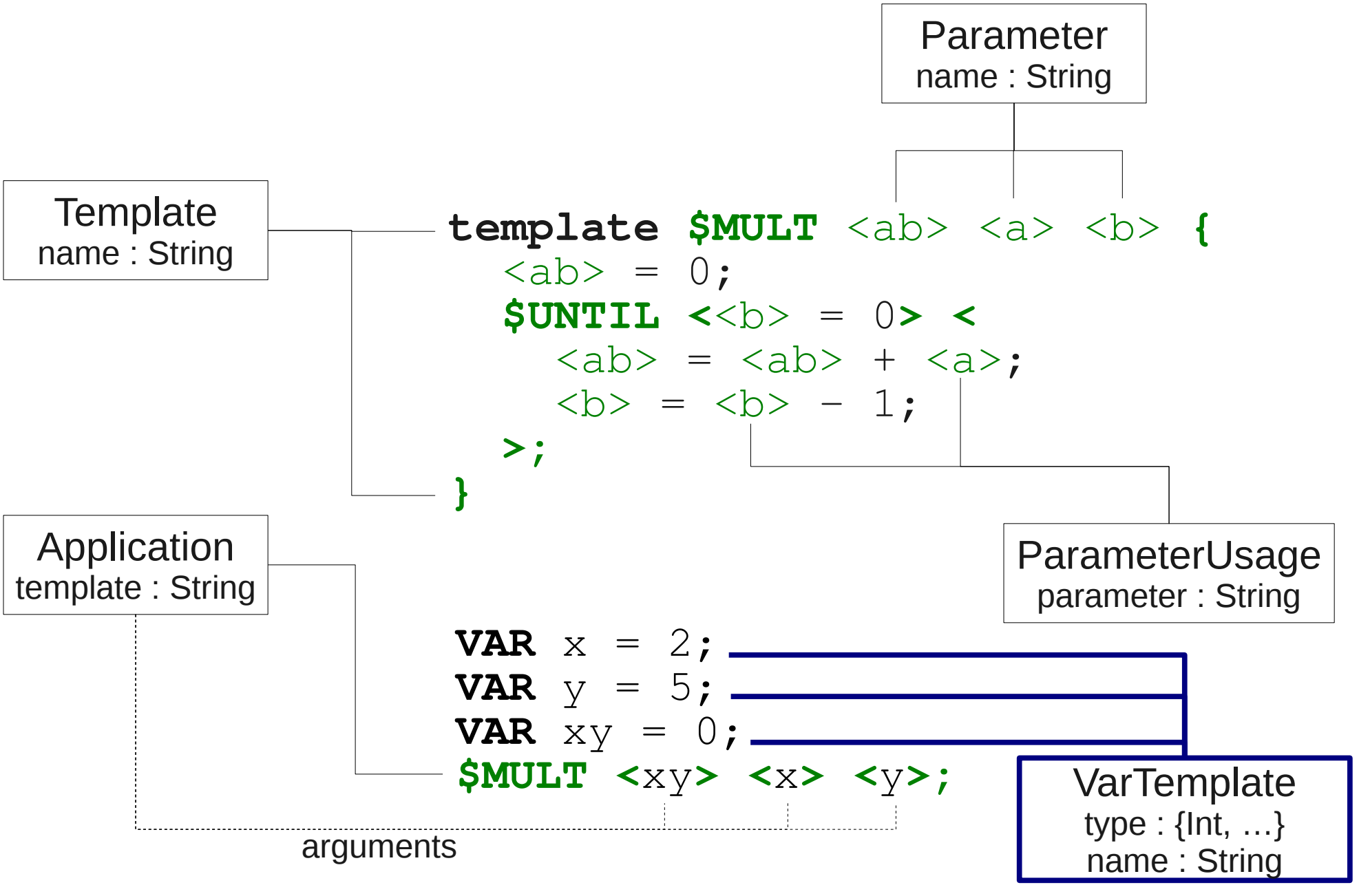


AST

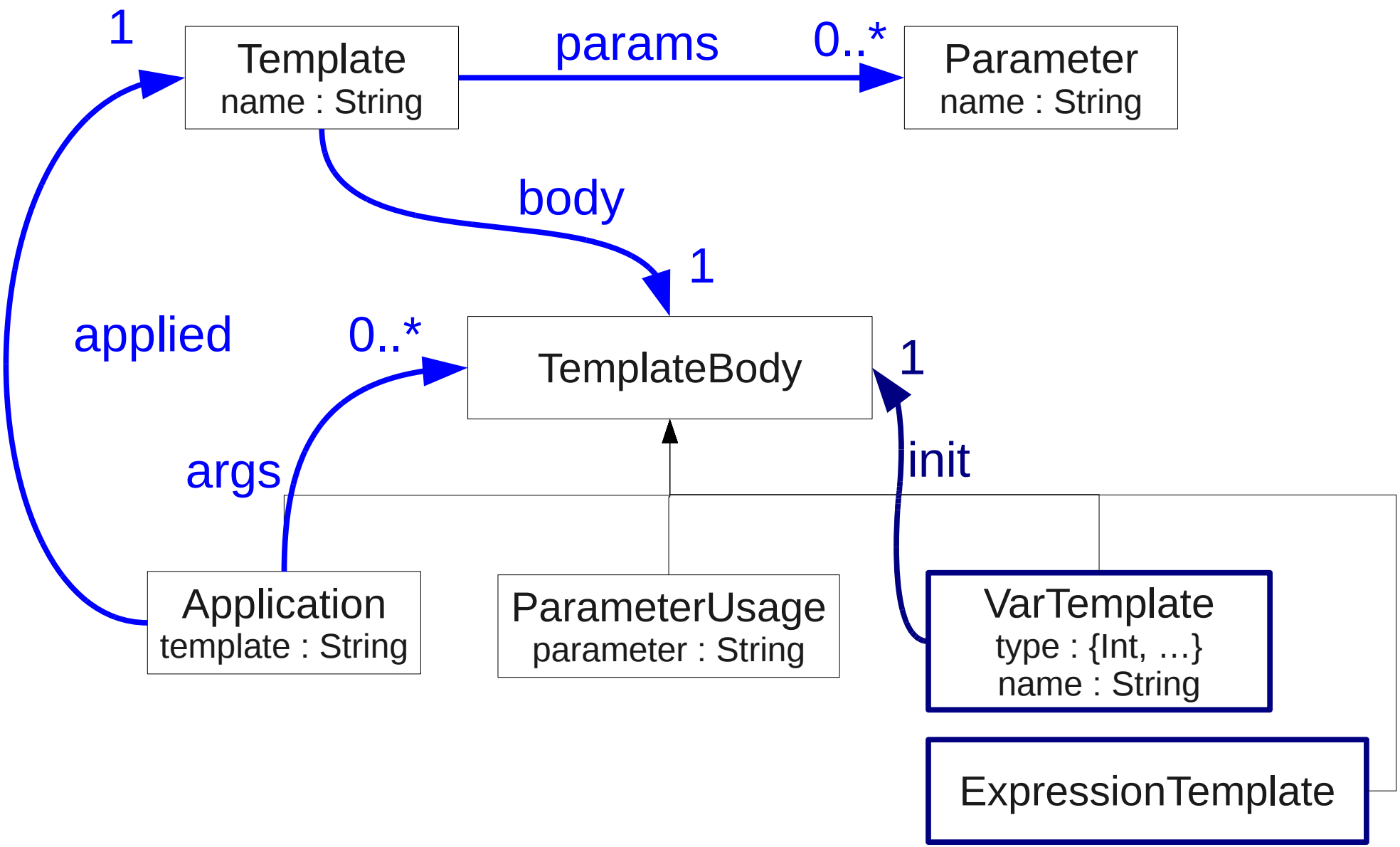


Templates AST

Describing templates

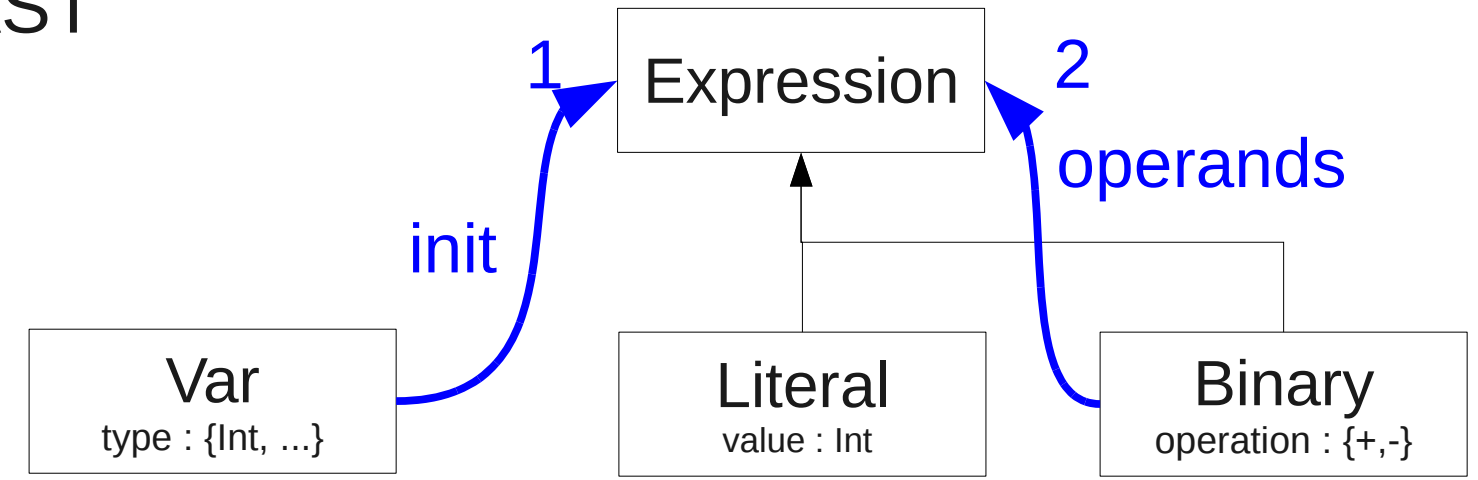


Template Description

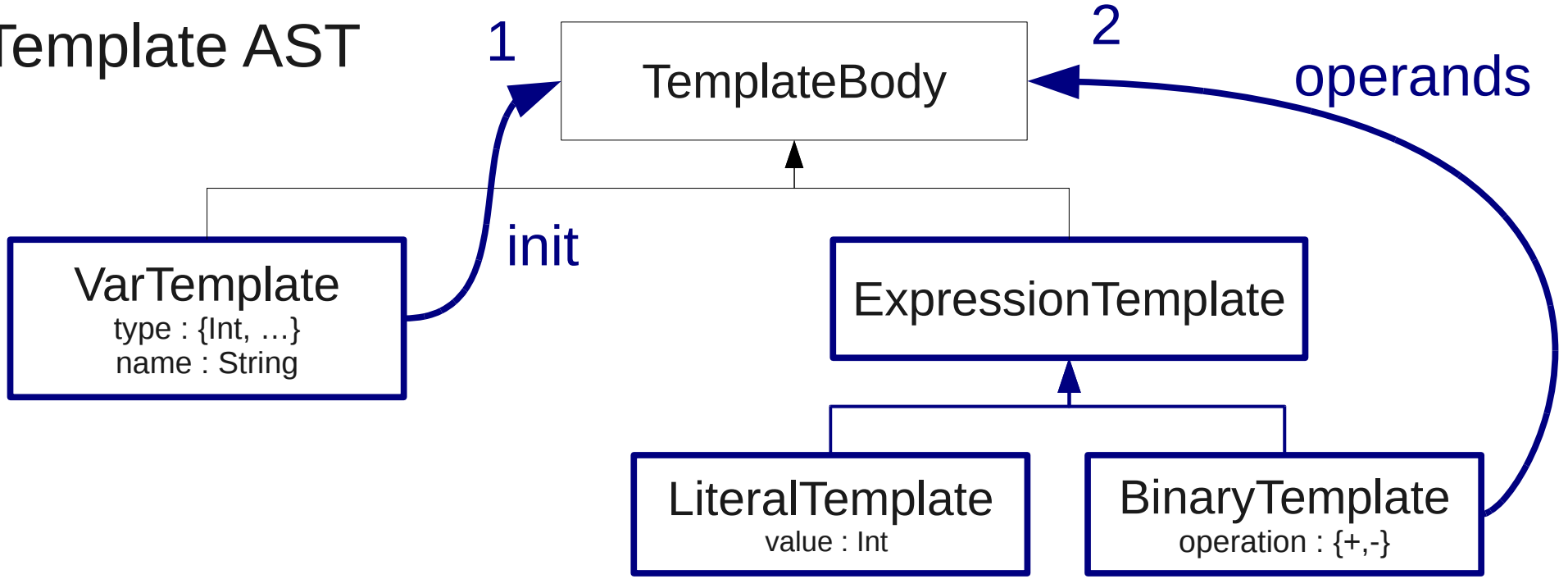


Language-Specific Templates

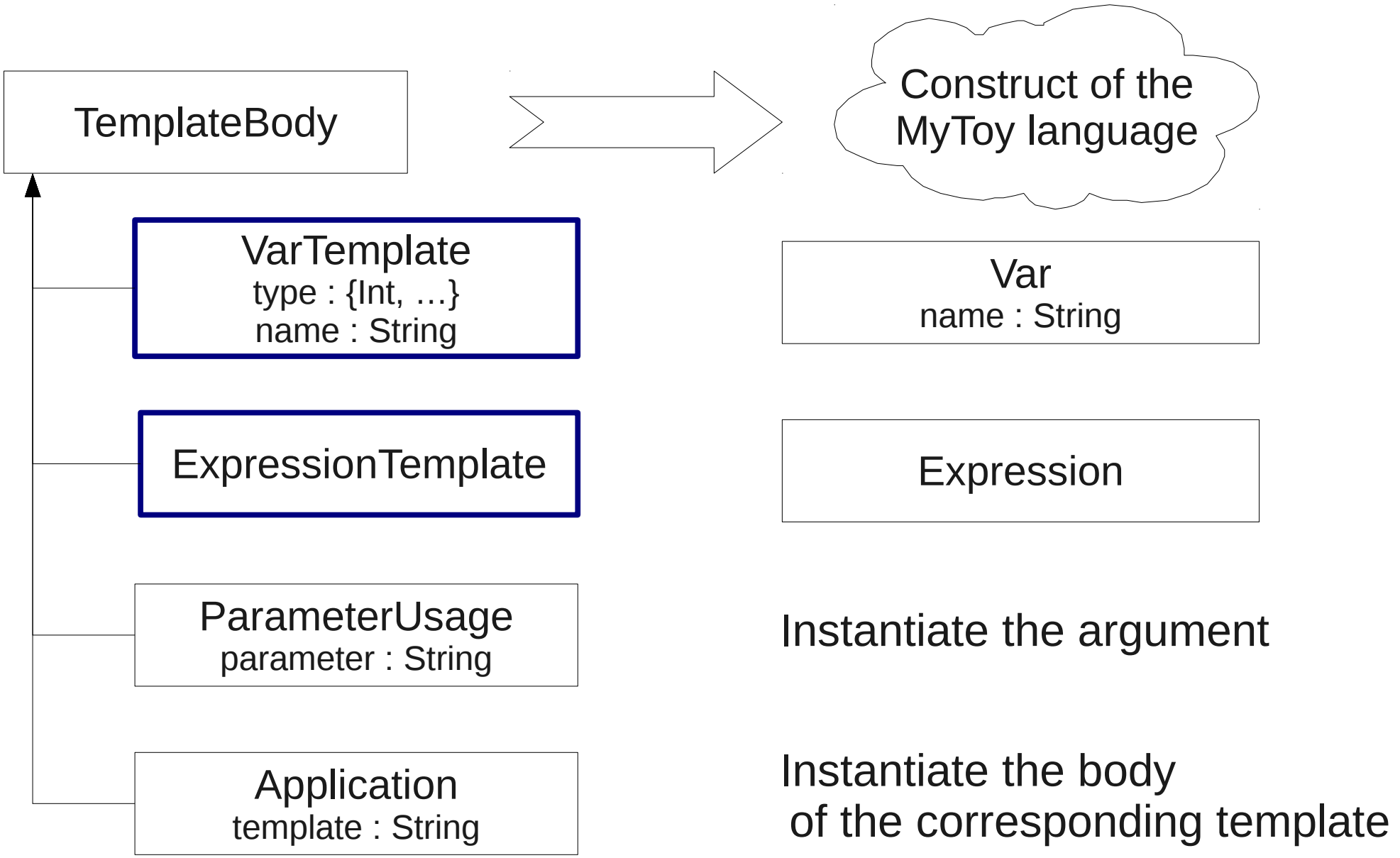
Initial AST



Template AST



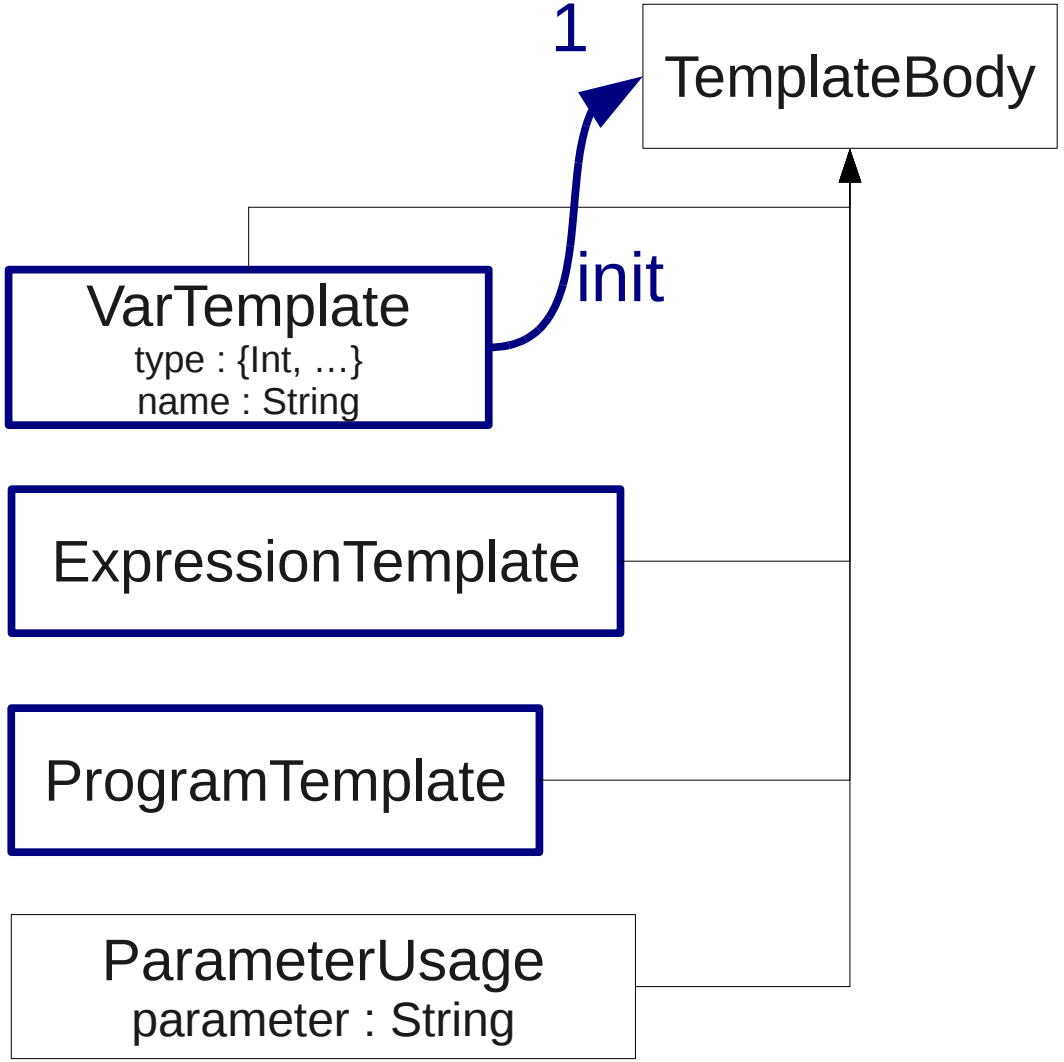
Instantiation



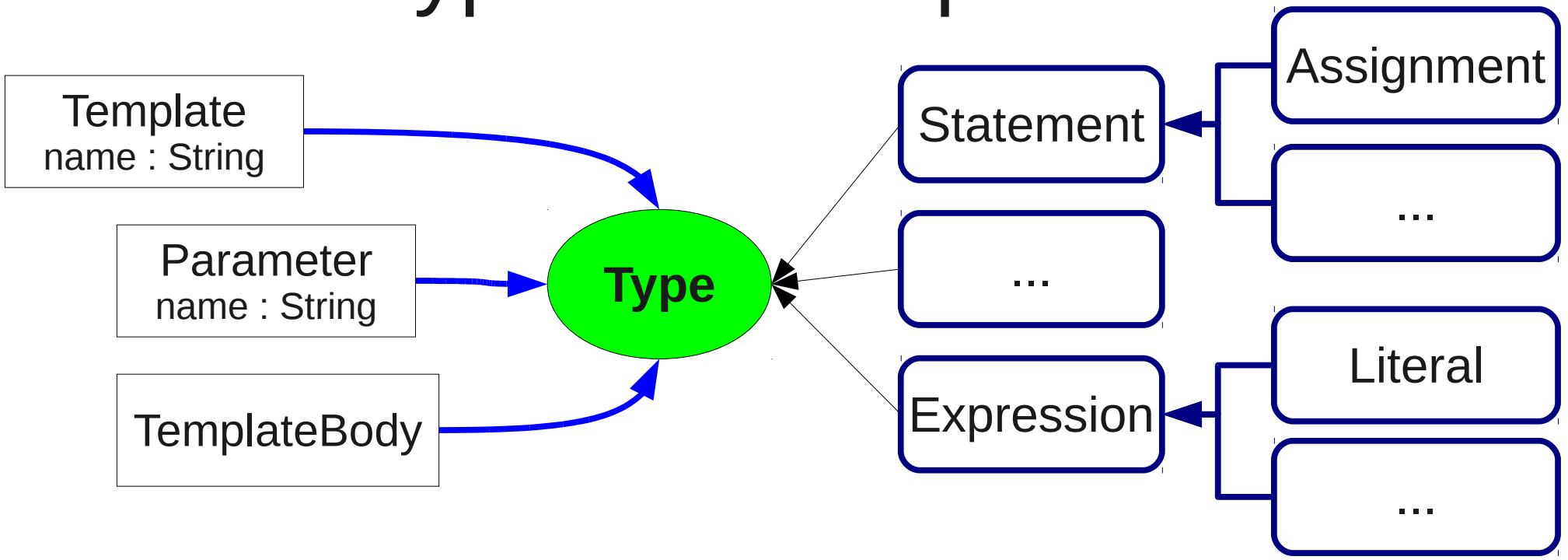
Structural Correctness

Constraints:

- Node types
- Node attributes
- Edge types
- Edge multiplicities
- **Typing of edge ends**



Types for Templates




```
template $UNTIL :: Statement[1..*]
     <condition :: Assignment[1]>
     <body :: Statement[0..*]> {
     VAR trigger = 0 :: Literal[1];
     comefrom trigger = 1;
       <body>
       Trigger = 1; :: Assignment[1]
     comefrom <condition>;
}
```

Types reflect multiplicities

Intermediate Summary

- A language is described as (Graph <-> AST)
- We extend a language to support templates
- We generate a type checker for templates
  - (no type annotations required)
  - This facilitates error reporting
- We generate an instantiation procedure
- Everything else is reused form the original language

- That's all great, but we capture only **some** errors

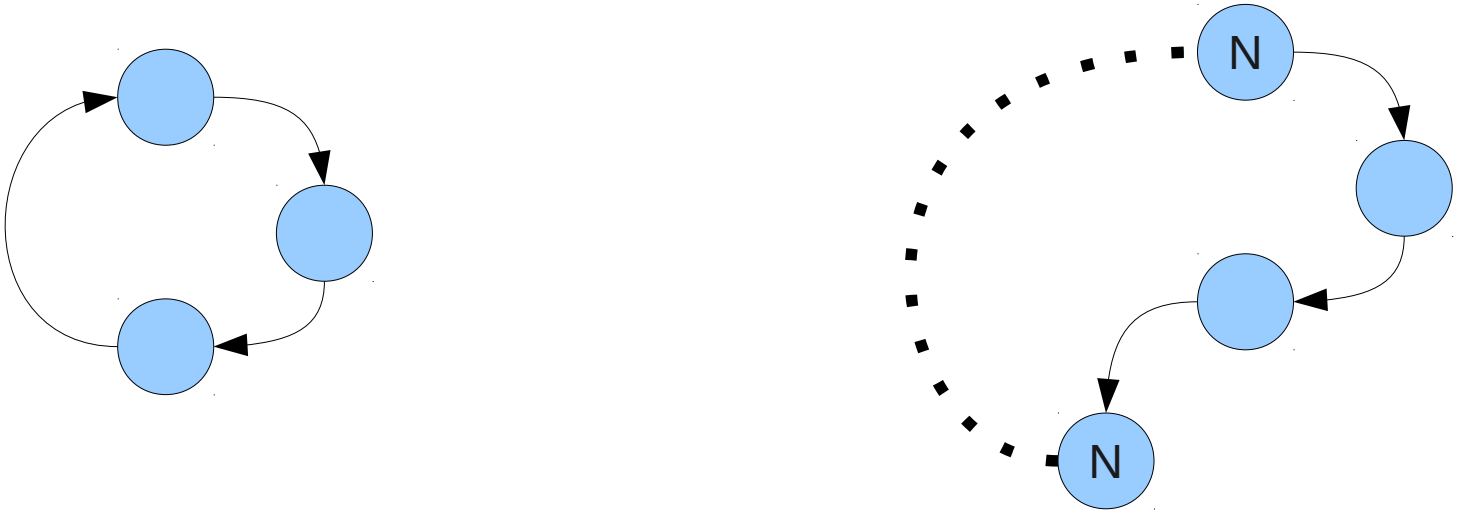
One more type of errors

```
VAR x = 2; VAR y = 5;
VAR xy = 0;
$UNTIL <x = 0> <
   xy = xy + y;
   x = x - 1;
>;
$UNTIL <y = 0> <
   y = y - 1;
>;
```

Error: variable 'trigger' is already defined!

```
template $UNTIL <condition> <body> {
     VAR trigger = 0;
     comefrom trigger = 1;
       <body>
       trigger = 1;
     comefrom <condition>;
}
```

Who said that was an error?!



Serialize

Graph

AST

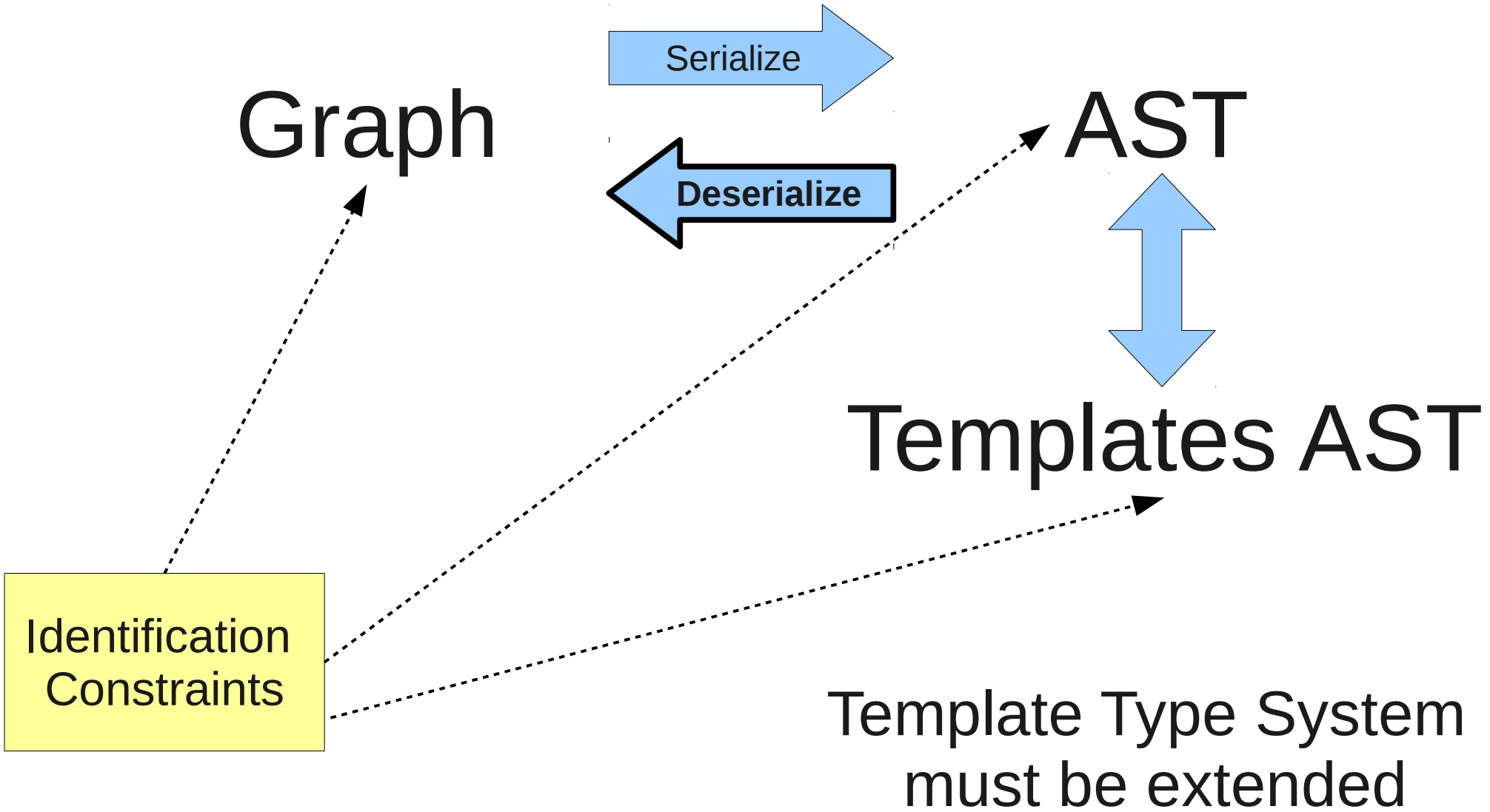
Deserialize

Structural Constraints

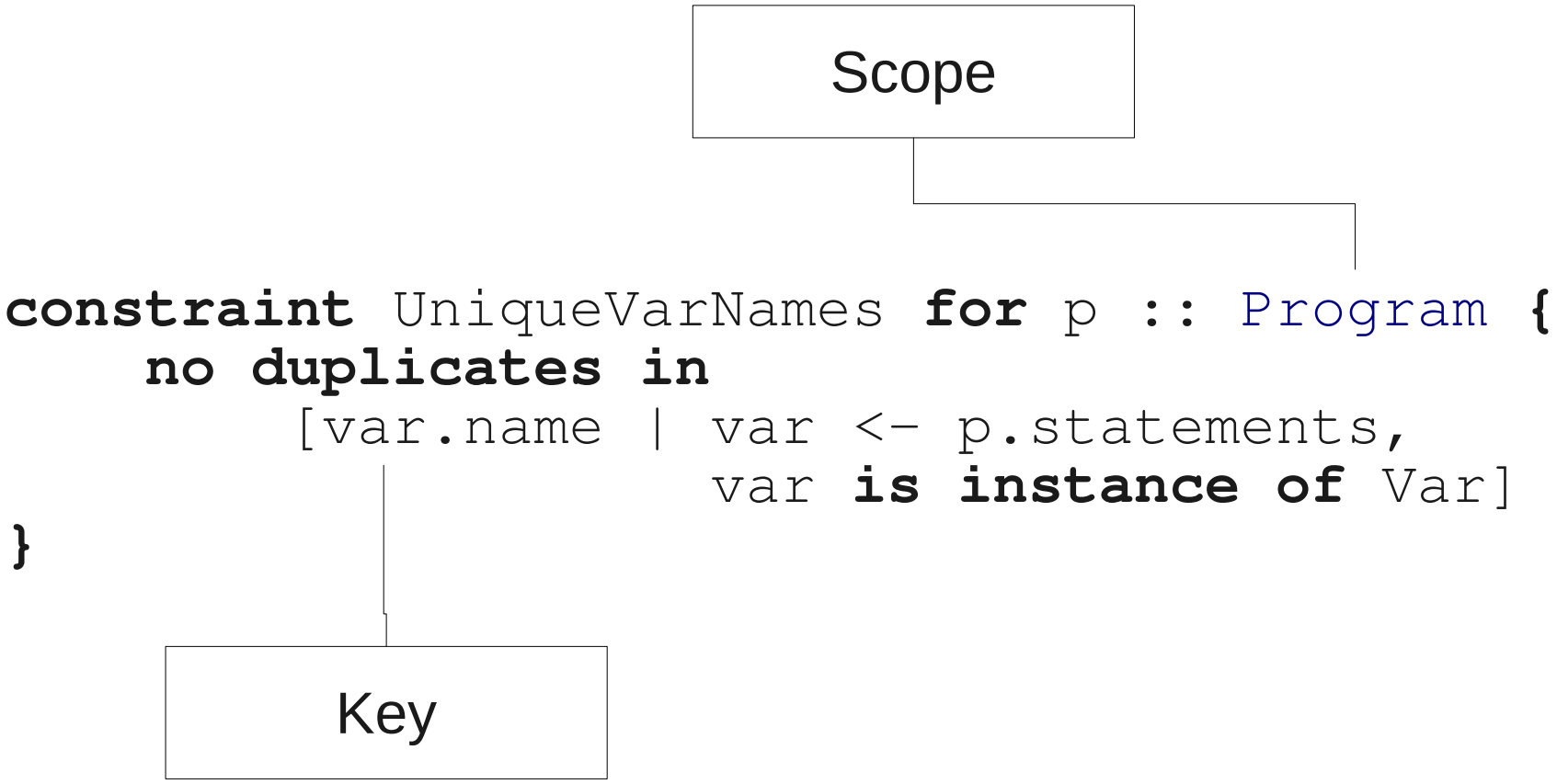
Identification Constraints

Templates AST

Our plan

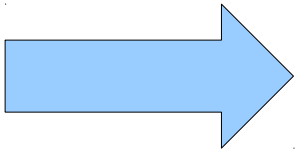

Serialize
Deserialize
Graph
AST
Templates AST
Identification
Constraints
Template Type System
must be extended

Identification Constraints



Name Checking for MyToy

```
VAR x = 2; VAR y = 5;
VAR xy = 0;
...
VAR trigger = 0;
...
VAR trigger = 0;
...
```



```
{x, y, xy,
 trigger,
 trigger}
```

- for every scope
  - compute the collection of keys
  - check it for duplicates

Is it sufficient?

- **Yes!**
  - MOF/Ecore
  - EBNF
  - DDL
  - DTD
  - Other language which describe static structures
- **No**
  - Programming languages
  - ...

Types for Name checking

Forget about names of templates and their parameters!

```
template $UNTIL :: defines('trigger', bodyDef)
 <condition> <body :: defines(bodyDef)> {
    VAR trigger = 0; :: defines('trigger')
    comefrom trigger = 1;
      <body> :: defines(bodyDef)
      trigger = 1;
    comefrom <condition>;
}
```

Name Checking Example

```
VAR x = 2; :: defines('x')
VAR y = 5; :: defines('y')
VAR xy = 0; :: defines('xy')
$UNTIL <x = 0> <
   xy = xy + y;
   x = x - 1;
>; :: defines('trigger')
$UNTIL <y = 0> <
   VAR one = 1;
   y = y - one;
>; :: defines('trigger', 'one')
```

Error: templates 'UNTIL' and 'UNTIL' define the same variable: trigger!

```
template $UNTIL :: defines('trigger', bodyDef)
```

Naming Errors

- Duplicate definition
  - done
- “Hygienic” templates
  - achievable
- Usage of undefined name (only AST)
  - symmetric
- Illegal context
  - In the graph an object is used while it's name is not visible
- Collision
  - Two distinct objects with the same names are used

Symmary

- If a language is described as (Graph <-> AST)
  - Automatically extend it with template support
  - Structural correctness guarantees
  - Some naming correctness guarantees
- Ecore, EBNF, DTD, etc.
  - Complete support for templates
- More complicated languages
  - Subject to future work...

Thanks for your attention!