Using Grammatical Aspects in Language Engineering

Andrey Breslav
abreslav@gmail.com

ITMO St. Petersburg / University of Tartu

Outline

- Motivation
  - What is Language Engineering
  - What problems we address
- Approach
  - *Grammatic* tool and its capabilities
- Applications
  - What *Grammatic* is used for

Language Engineering

- **Compilers**
  - Lexical analysis
  - Syntactical analysis
  - “Static semantics”

Language Engineering

- **Compilers**
  - Lexical analysis
  - Syntactical analysis
  - “Static semantics”
- **Program analysis tools**
  - Style enforcement
  - Bug detecion
- **Pretty-printers**

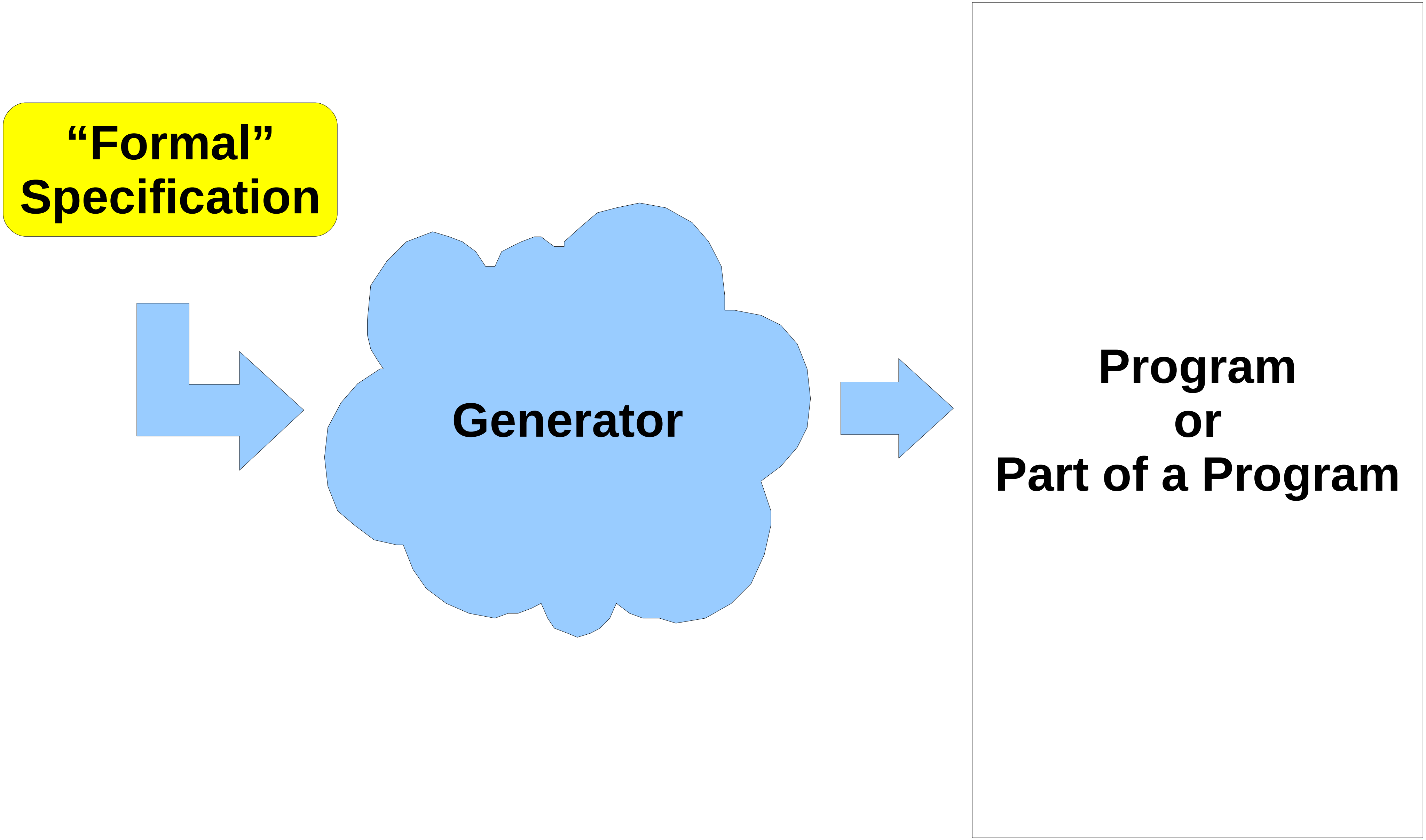
- **IDEs**
  - Syntax highlighting
  - Code navigation
  - Code completion
  - Outline
  - Code folding
  - Refactorings
  - ...

Why it is important

- We have many programming languages
  - And even more tools accompanying them
- We create Domain-Specific Languages
  - Designed to express notions of a single domain
  - Created **on demand**

- => **Must be cheap to create and maintain**
  - **Must still be reliable**

Cheap ~~but~~ **and** Reliable

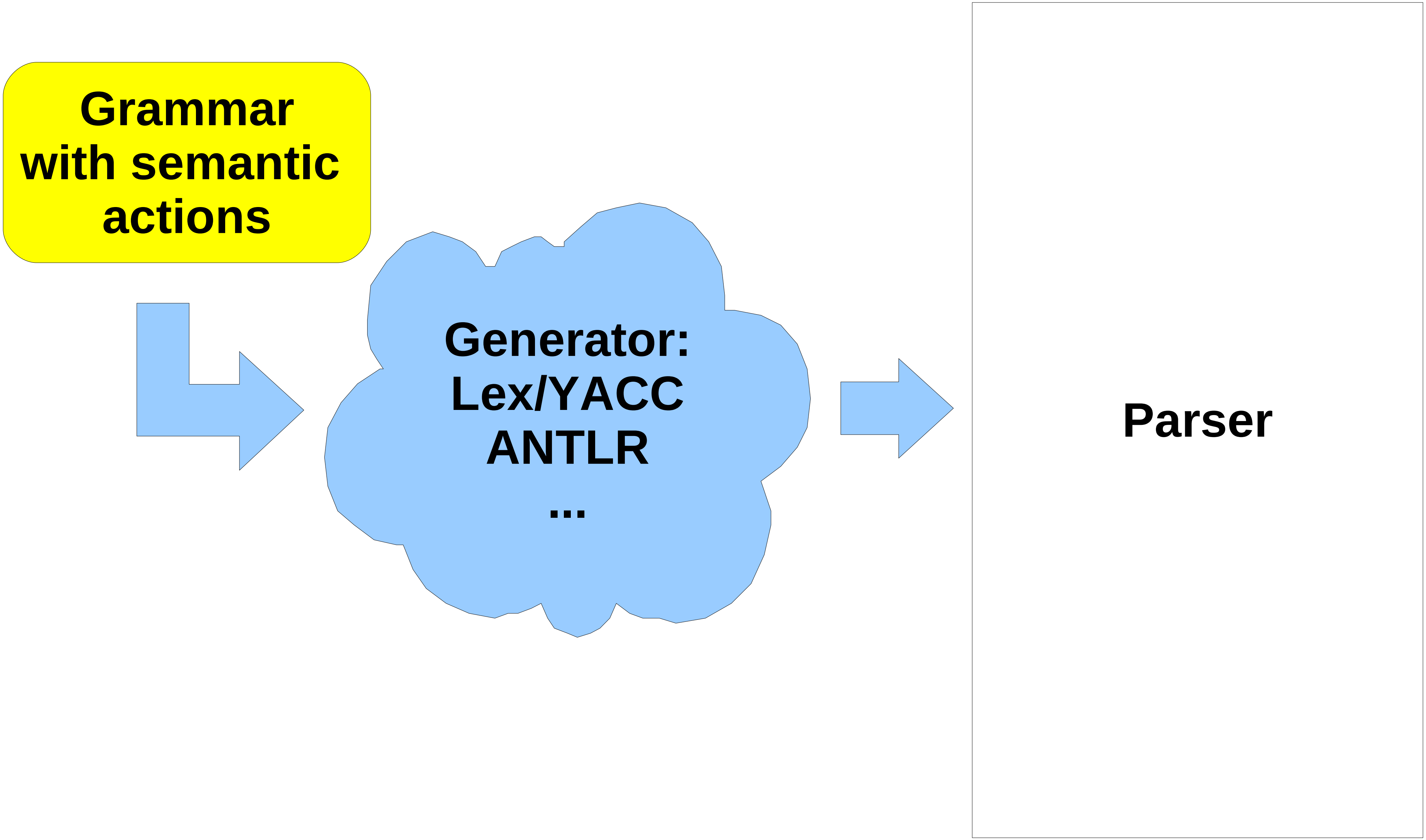


“Formal” Specification

Generator

Program
or
Part of a Program

Familiar Example: Parser generator

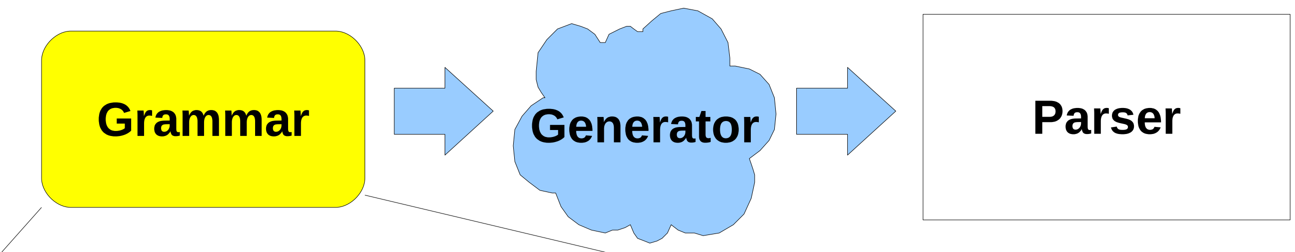


Grammar with semantic actions

Generator: Lex/YACC ANTLR ...

Parser

A Concise Specification



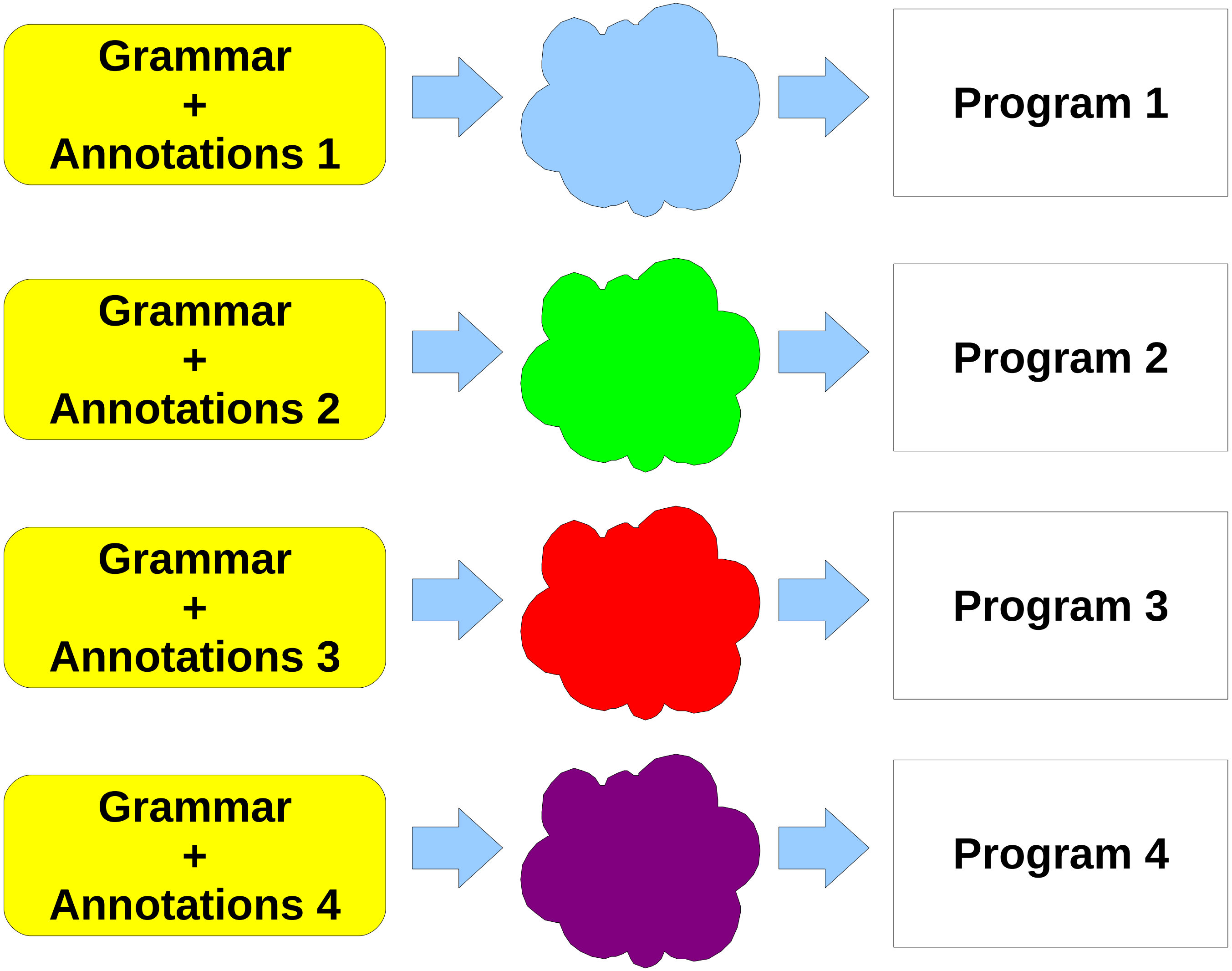
```
expr   : term ((PLUS | MINUS) term)* ;
term   : factor ((MULT | DIV) factor)* ;
factor : NUMBER ;
```

Problem 1: A Real Specification

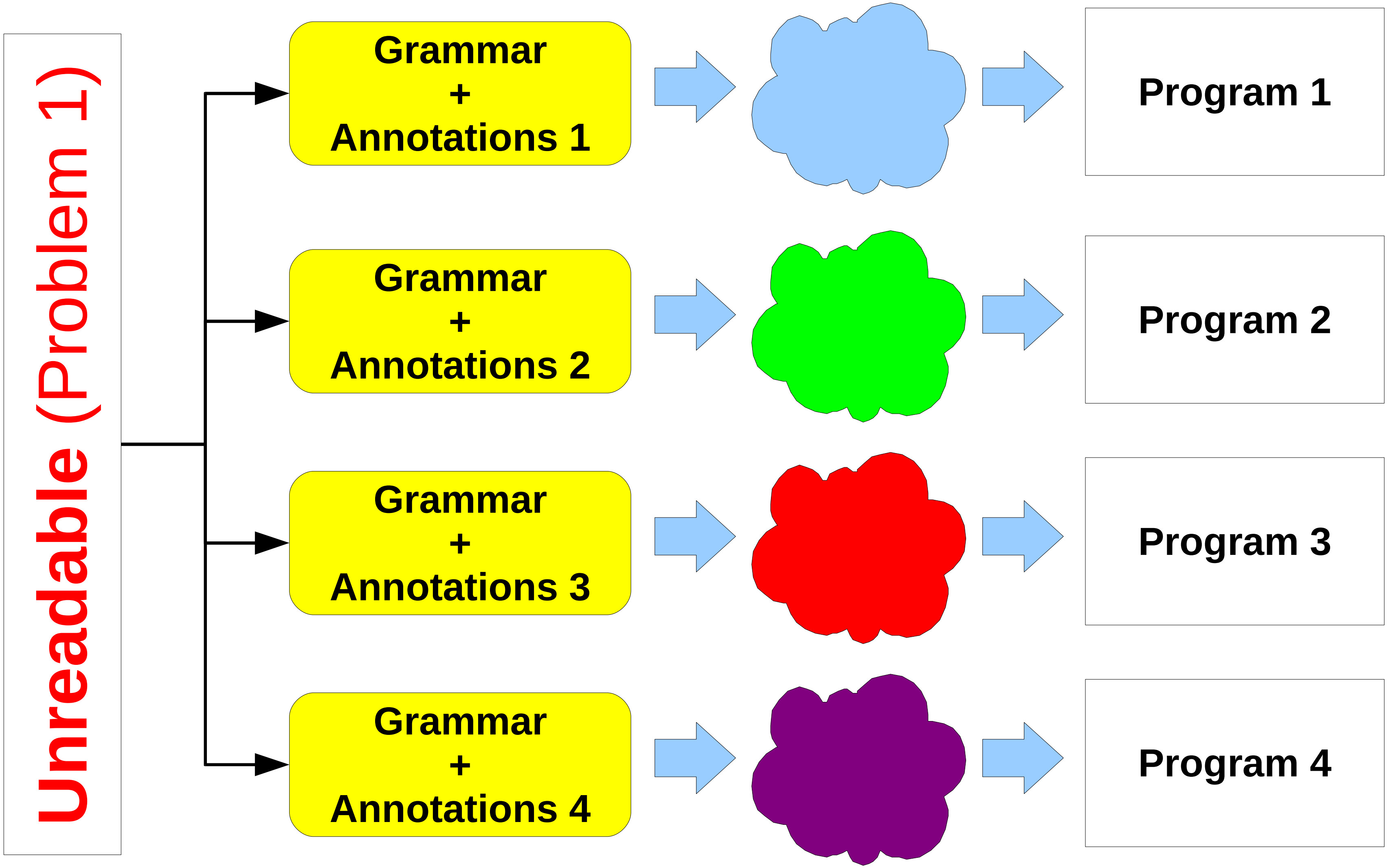
Grammar with **semantic actions** (ANTLR v3)

```
expr returns [int result] :
    t=term {result = t;}
    ({int sign = 1;} (PLUS | MINUS {sign = -1;}) t=term {result += sign * t;})*;

term returns [int result] :
    f=factor {result = f;}
    ({boolean div = false;} (MULT | DIV {div = true;}) f=factor {
        if (div)
            result /= f;
        else
            result *= f;
     })*;

factor returns [int result] :
    n=NUMBER {result = Integer.parseInt(n);};
```

Problem 2: Many features for the same language



Problem 2: Many features for the same language



Unreadable (Problem 1)

Grammar
+
Annotations 1

Program 1

Grammar
+
Annotations 2

Program 2

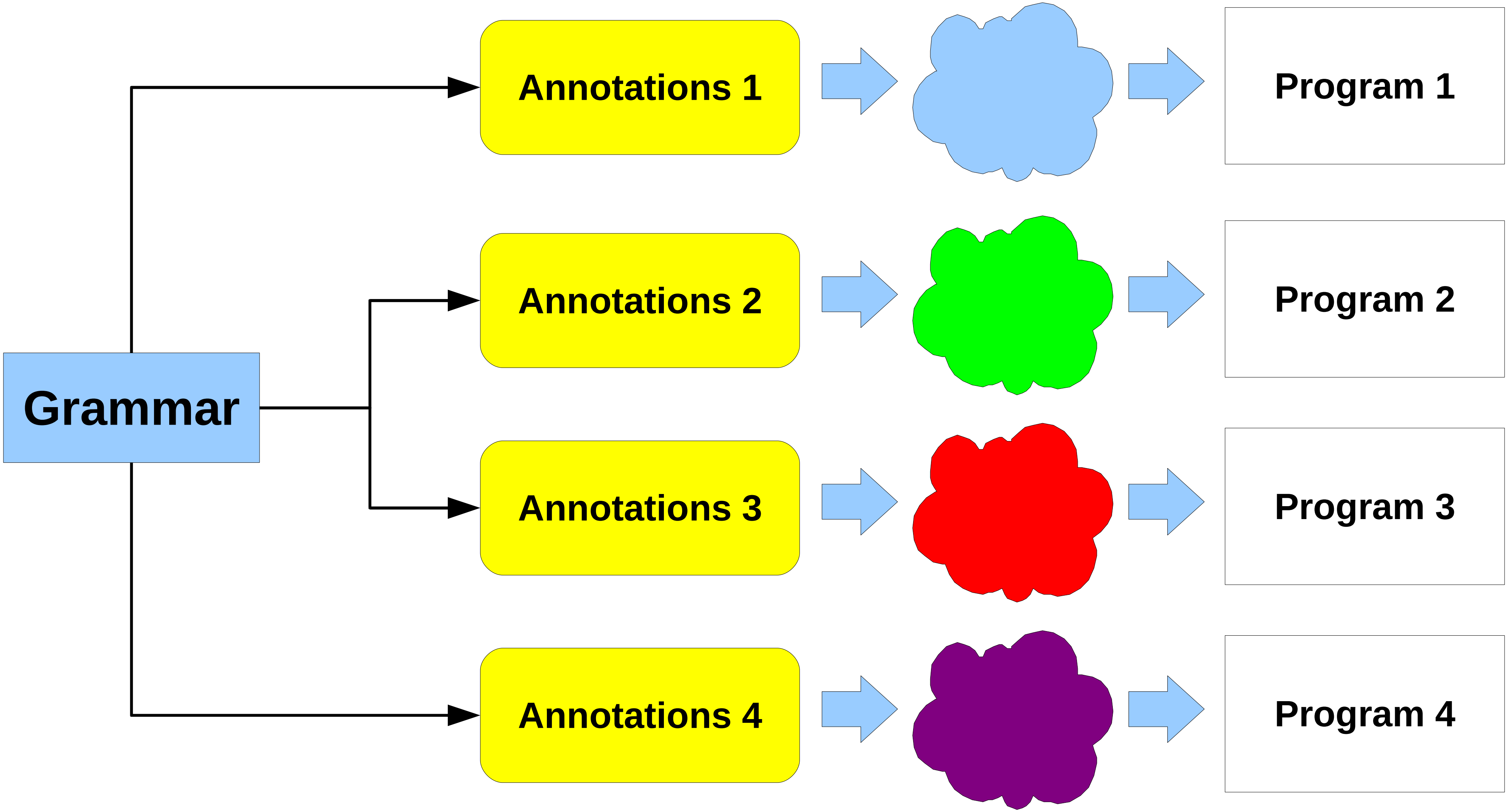
Grammar
+
Annotations 3

Program 3

Grammar
+
Annotations 4

Program 4

Solution (1 and 2): Separation of concerns



Grammar

Annotations 1

Annotations 2

Annotations 3

Annotations 4

Program 1

Program 2

Program 3

Program 4

Separation of concerns for semantic actions

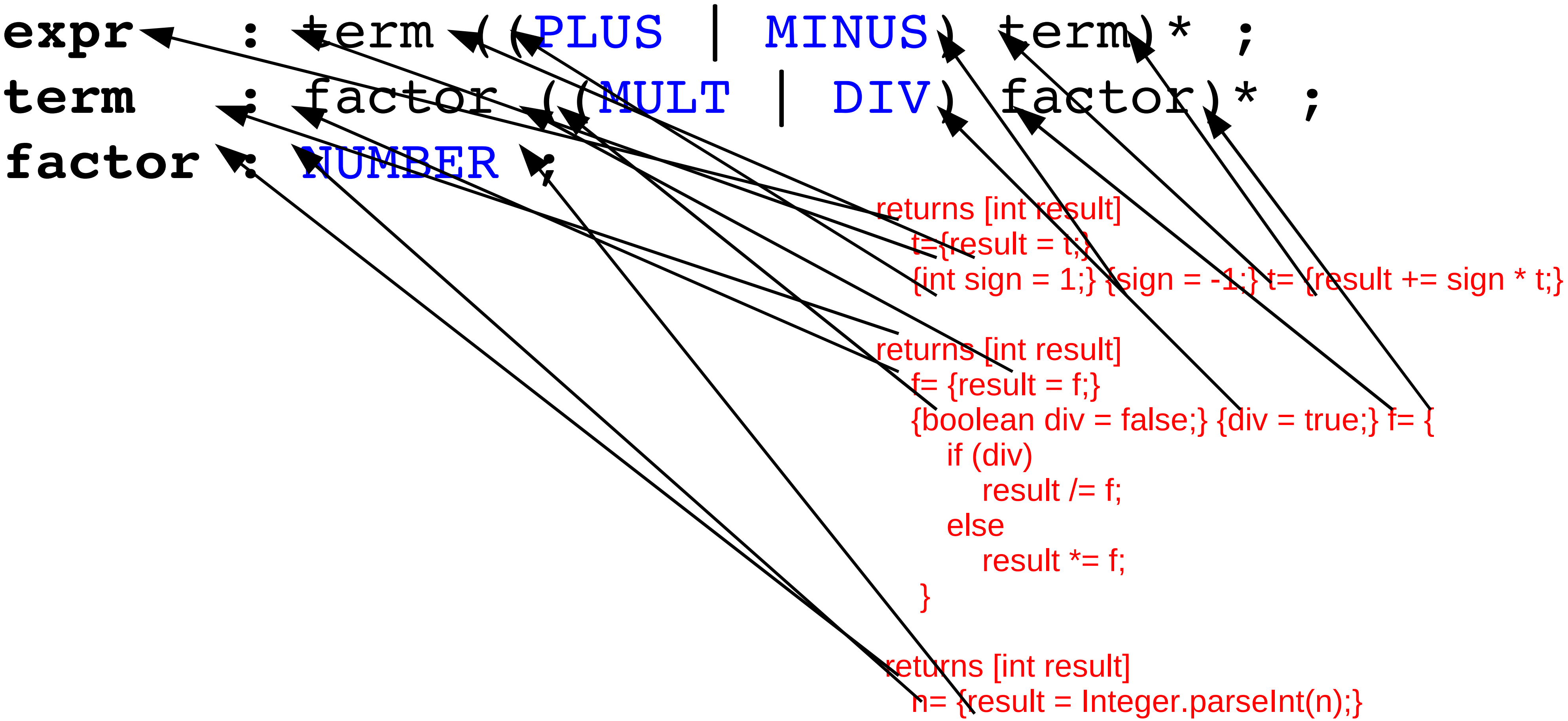
Grammar AND **semantic actions**

```
expr   : term ((PLUS | MINUS) term)* ;
term   : factor ((MULT | DIV) factor)* ;
factor : NUMBER ;
```

```
returns [int result]
    t={result = t;}
    {int sign = 1;} {sign = -1;} t= {result += sign * t;}

returns [int result]
    f= {result = f;}
    {boolean div = false;} {div = true;} f= {
        if (div)
            result /= f;
        else
            result *= f;
     }

 returns [int result]
    n= {result = Integer.parseInt(n);}
```

Separation of concerns: Attaching annotations **(Join Points)**

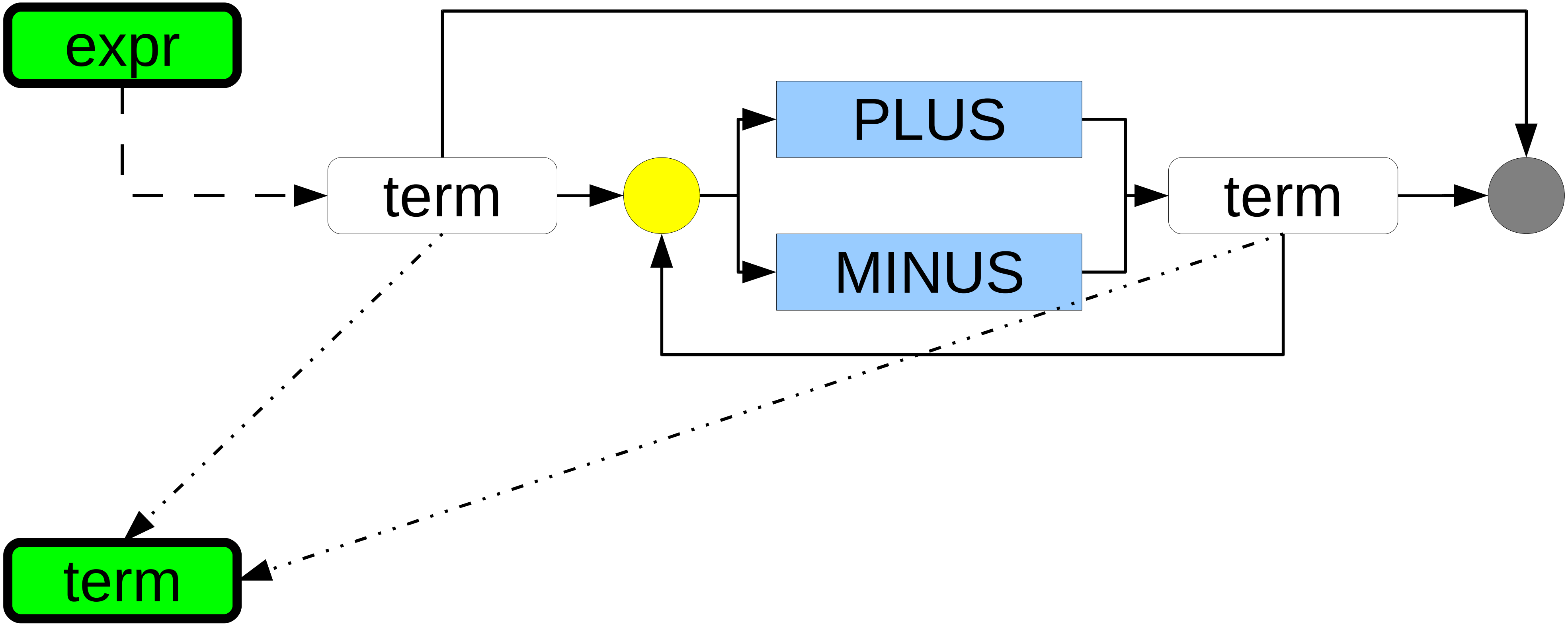
Grammar AND **semantic actions**



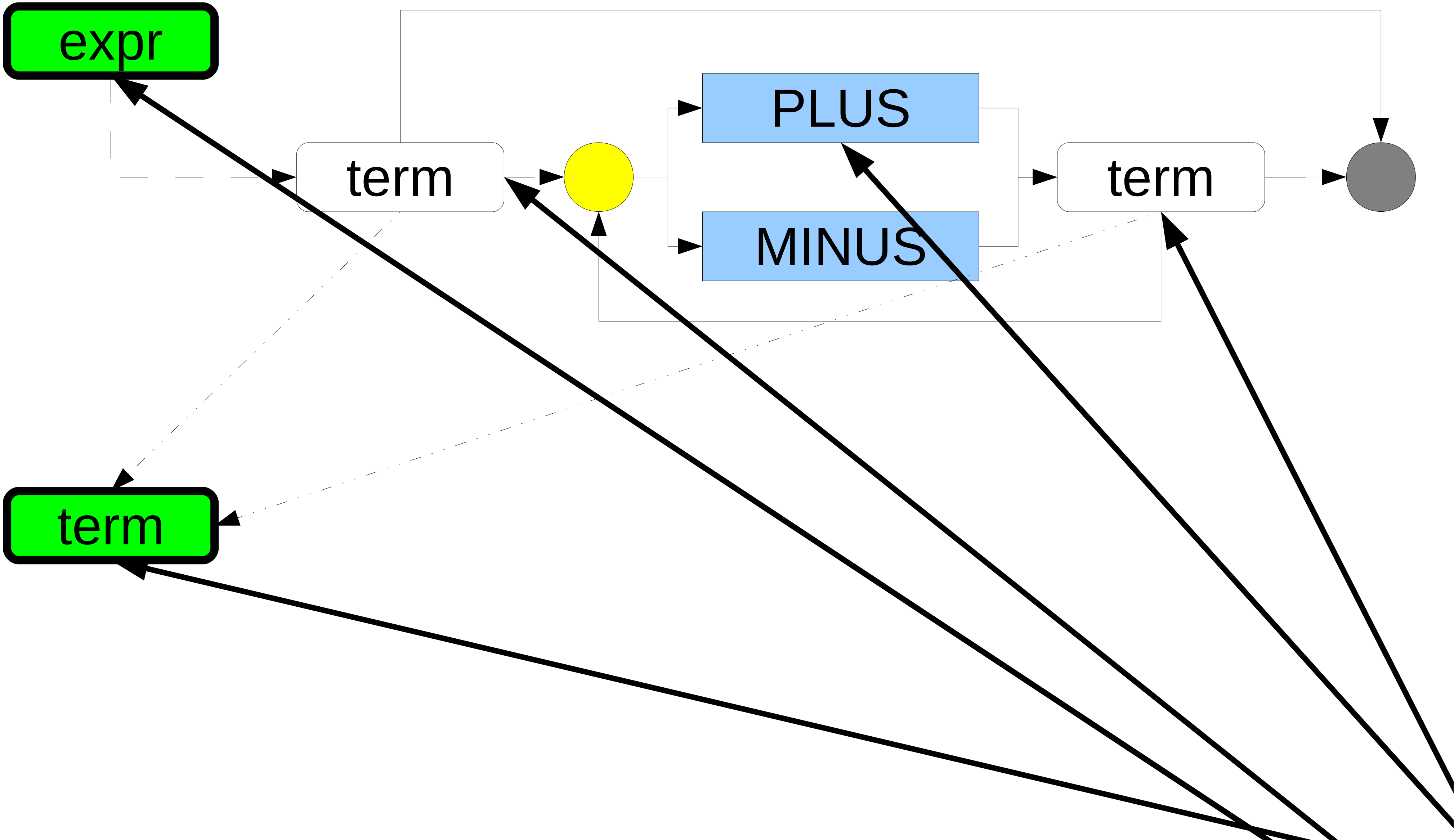
```
expr   : term ((PLUS | MINUS) term)* ;
term   : factor ((MULT | DIV) factor)* ;
factor : NUMBER ;
```

```
returns [int result]
   t={result = t;}
   {int sign = 1;} {sign = -1;} t= {result += sign * t;}

returns [int result]
   f= {result = f;}
   {boolean div = false;} {div = true;} f= {
       if (div)
           result /= f;
       else
           result *= f;
    }

returns [int result]
   n= {result = Integer.parseInt(n);}
```

Grammar is not a Text



Grammar is not a Text



Attaching annotations: Pattern language **(Point Cuts)**

Pattern name	**Sign**	**What it matches**
Grammar element		An element of the same structure
Sequence wildcard	**..**	A sequence of elements
Alternative wildcard	**...**	Any number of alternatives
Production wildcard	**{...}**	Any number of productions
Symbol wildcard	**#**	Any symbol
Lexical wildcard	**#lex**	Any lexical element
Variable	**$v**	The same thing every time

Pattern examples

- **expr** : term ((PLUS | MINUS) term)* ;
  - Exact match
- **expr** : {...} ;
  - Production wildcard
- # : term .. ;
  - Sequence wildcard
- **#** : **$t**=# ((PLUS | MINUS) **$t**)* ;
  - Symbol wildcard and a variable

Example: Highlighting specification

```
class Time {
  hours, minutes;
  midnight() {
    (this.hours == 0) &&
    (this.minutes == 0);
  }
}

t = new Time();
t.hours = 10;
print(t.midnight());
```

Roles

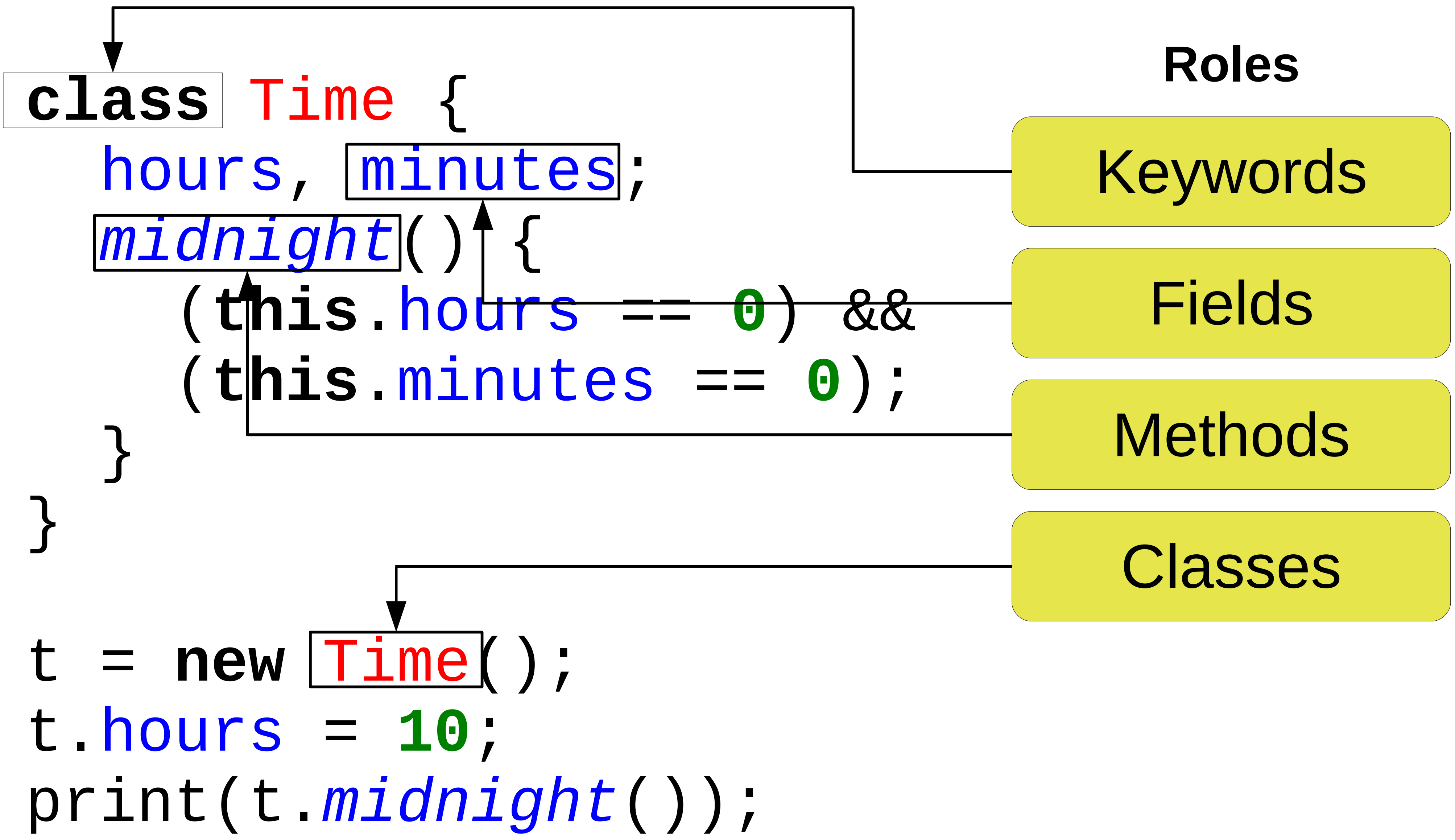
Keywords

Fields

Methods

Classes

Example: Highlighting specification



```
class Time {
  hours, minutes;
  midnight() {
    (this.hours == 0) &&
    (this.minutes == 0);
  }
}

t = new Time();
t.hours = 10;
print(t.midnight());
```

Roles

Keywords

Fields

Methods

Classes

The Toy-Language Grammar (1)

```
class
   : 'class' ID '{'
        fields?
        methods?
     '}'
   ;

newExpr
   : 'new' ID '(' ')'
   ;
```

Roles

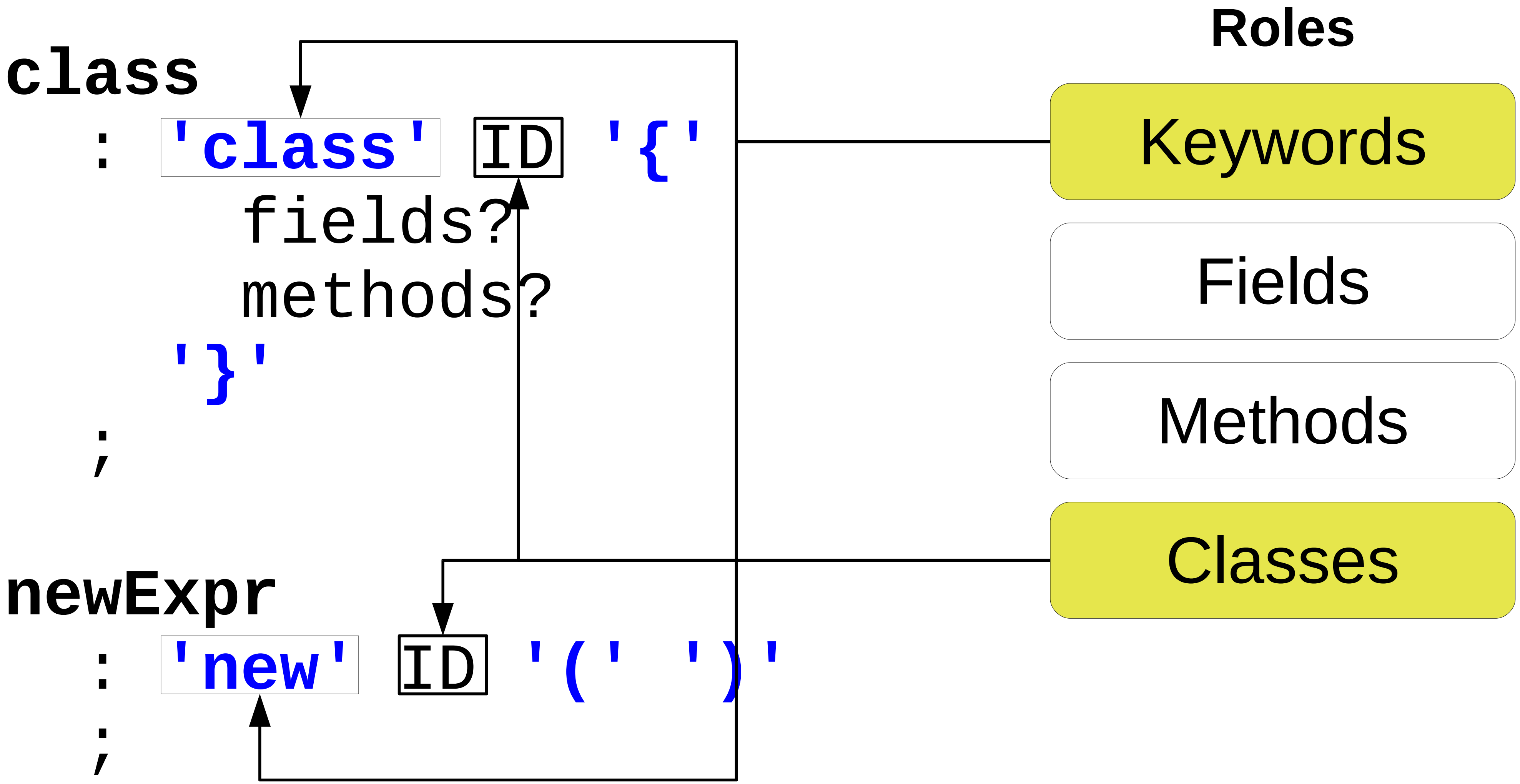
Keywords

Fields

Methods

Classes

Roles in the grammar (1)

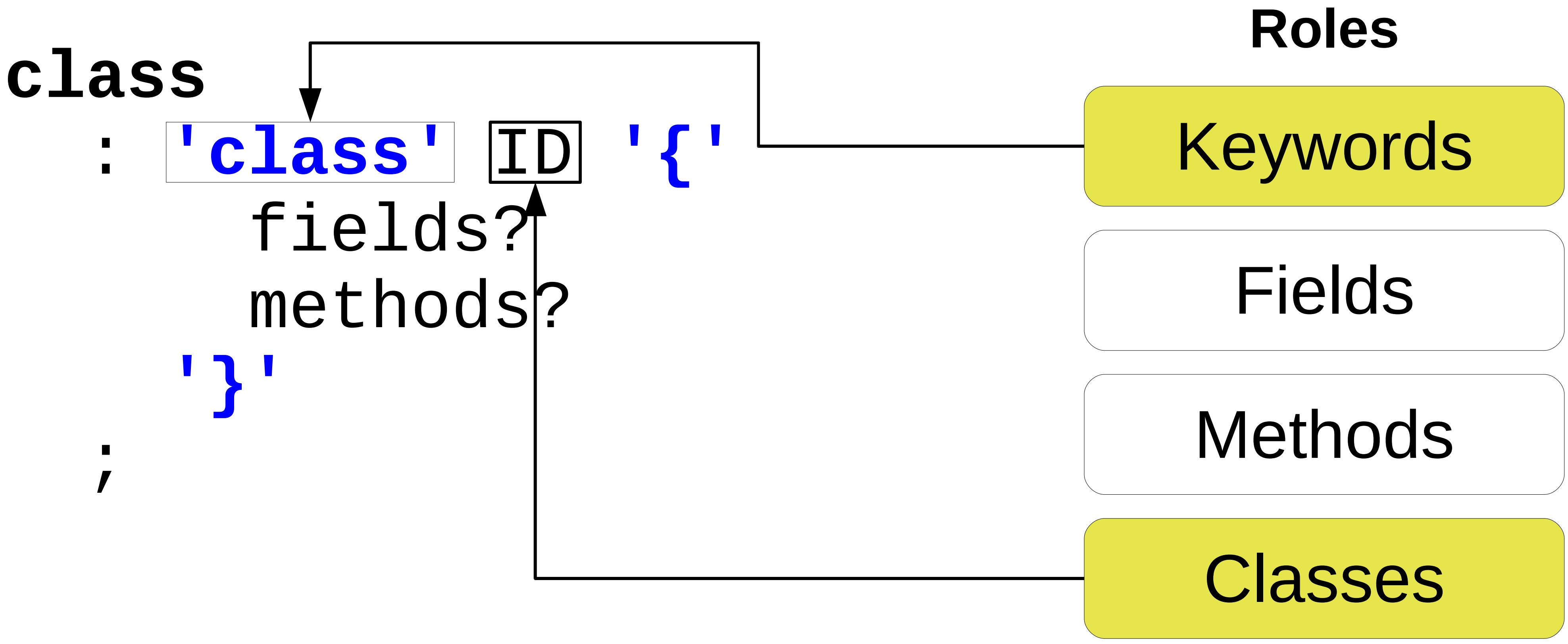


```
class
   : 'class' ID '{'
       fields?
       methods?
     '}'
   ;

newExpr
   : 'new' ID '(' ')'
   ;
```

Roles

- Keywords
- Fields
- Methods
- Classes

Roles in the grammar (1)



```
class
  : 'class' ID '{'
      fields?
      methods?
    '}'
  ;
```

Roles

Keywords

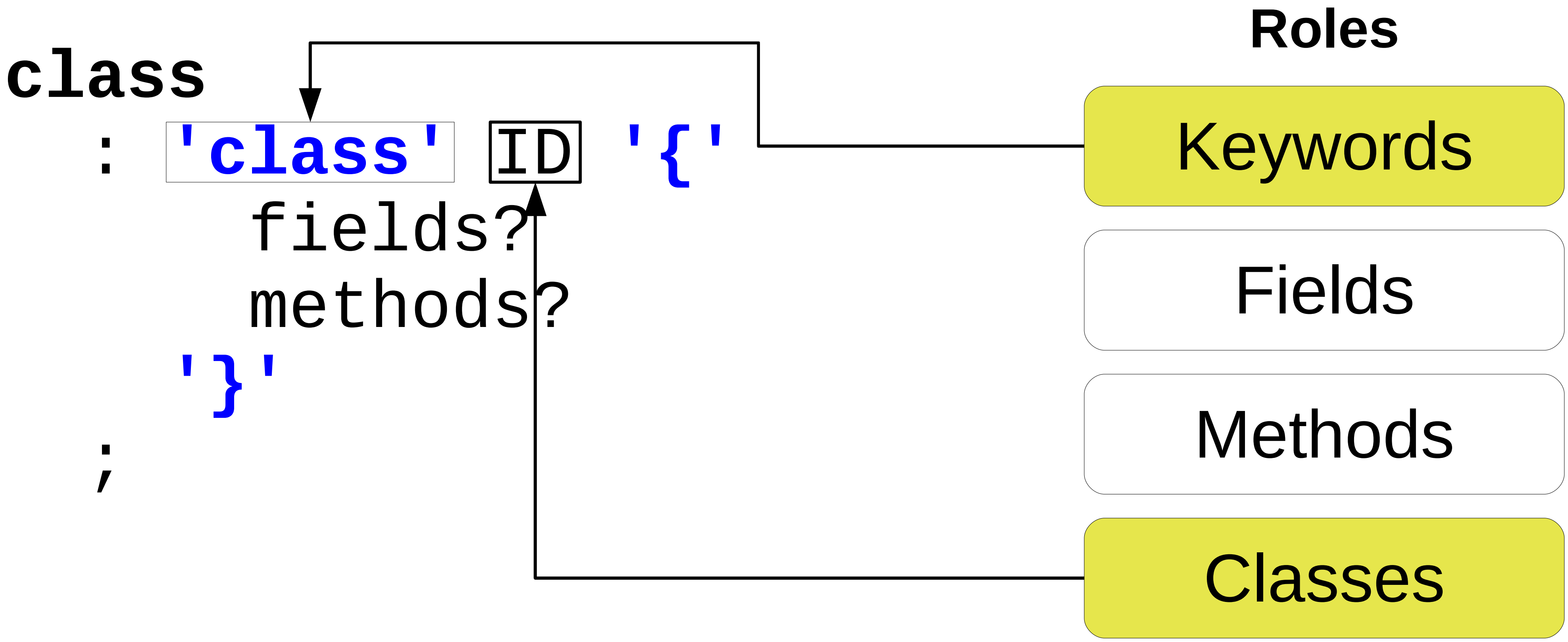
Fields

Methods

Classes

```
# : 'class' ID .. ;
```

Highlighting specification **(Aspect)**



```
class
   : 'class' ID '{'
       fields?
       methods?
     '}'
   ;
```

Roles

Keywords

Fields

Methods

Classes

```
# : 'class' ID .. ;
   @'class'.role = keyword;
   @ID.role = className;
```

The Toy-Language Grammar (2)

```
fields
   : ID (',' ID)* ';'
   ;

memberAccess
   : ID '.' ID
   : ID '.' ID '(' ')'
   ;

method
   : ID '('')' '{' ... '}'
   ;
```

Roles

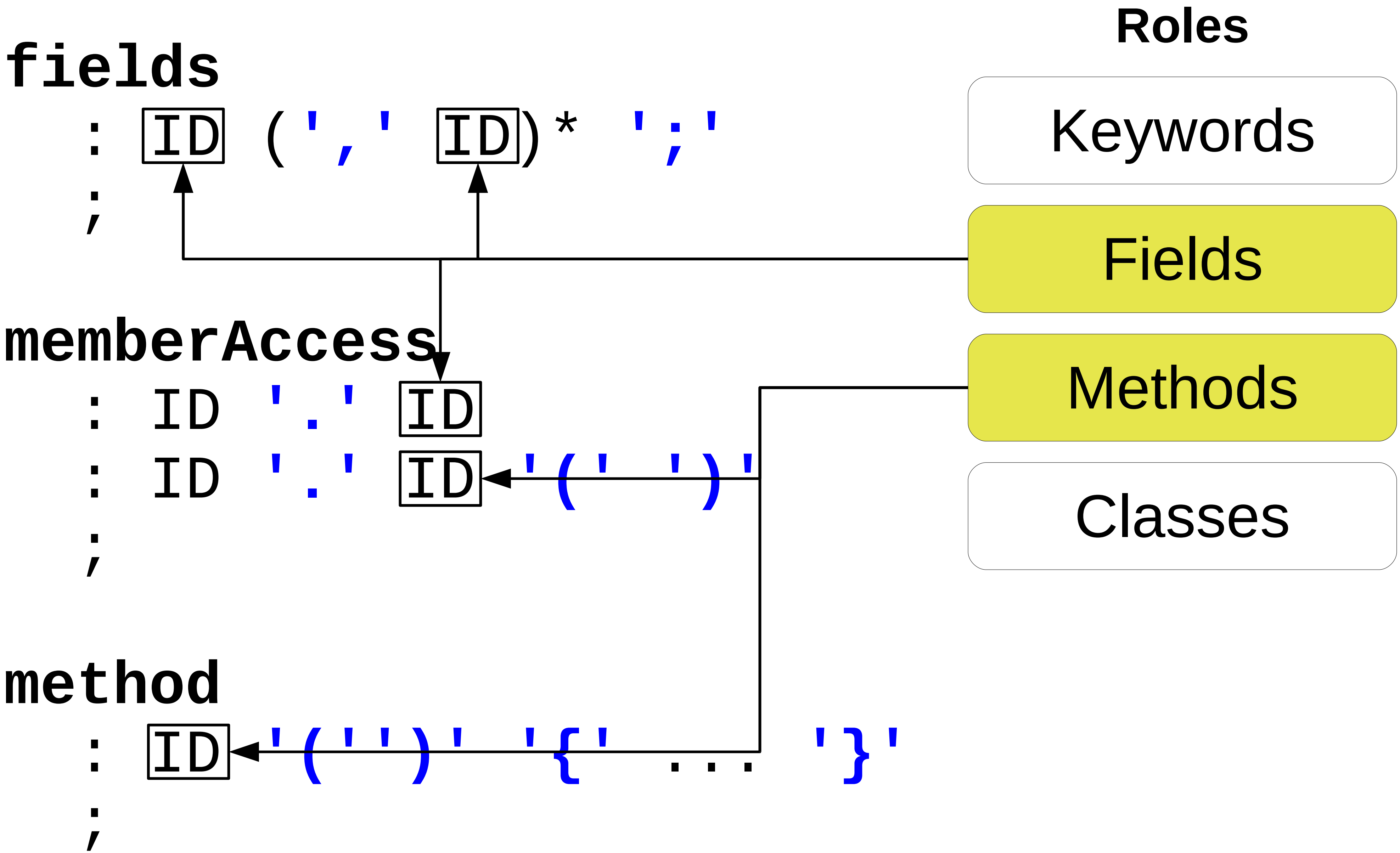
Keywords

Fields

Methods

Classes

Roles in the grammar (2)

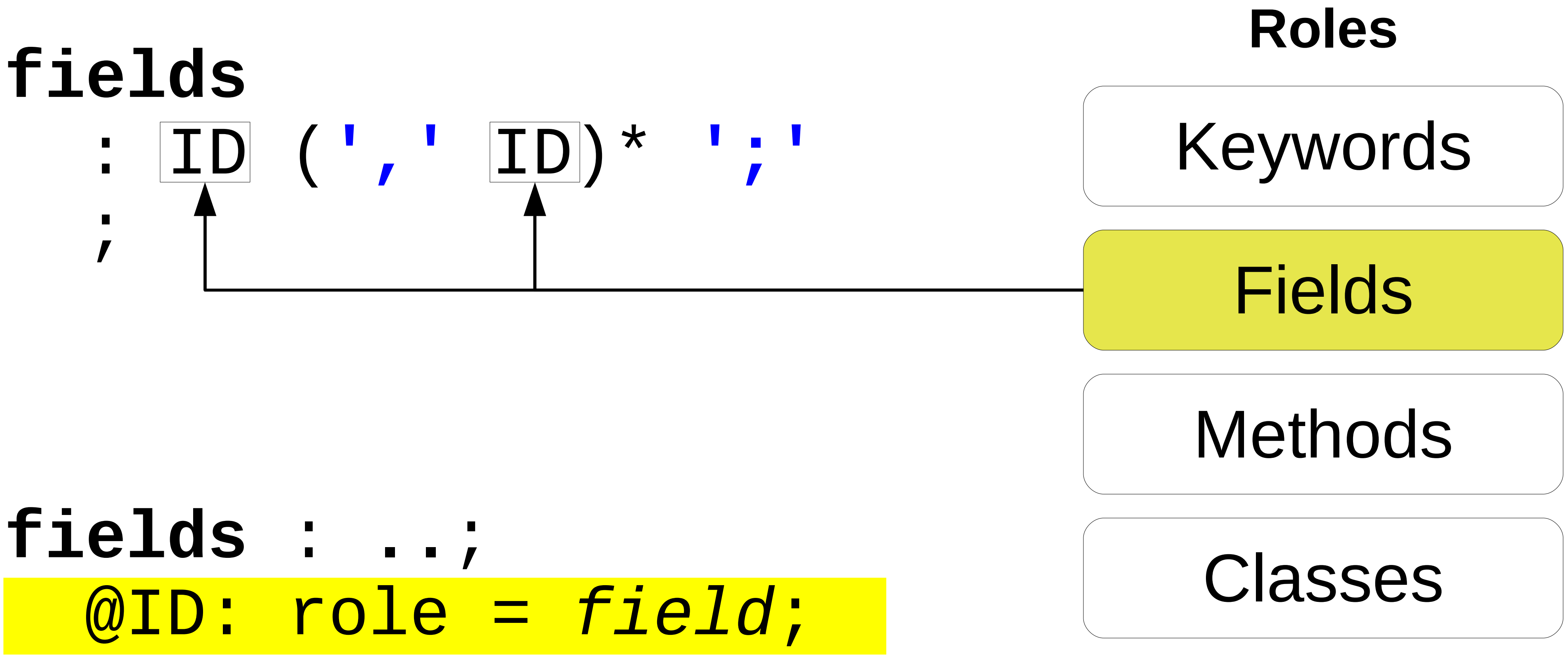


```
fields
   : ID (',' ID)* ';'
   ;

memberAccess
   : ID '.' ID
   : ID '.' ID '(' ')'
   ;

method
   : ID '(' ')' '{' ... '}'
   ;
```

Roles

- Keywords
- Fields
- Methods
- Classes

Highlighting specification **(Aspect)**



```
fields
   : ID (',' ID)* ';'
   ;
```

```
fields : ..;
  @ID: role = field;
```

Roles

Keywords

Fields

Methods

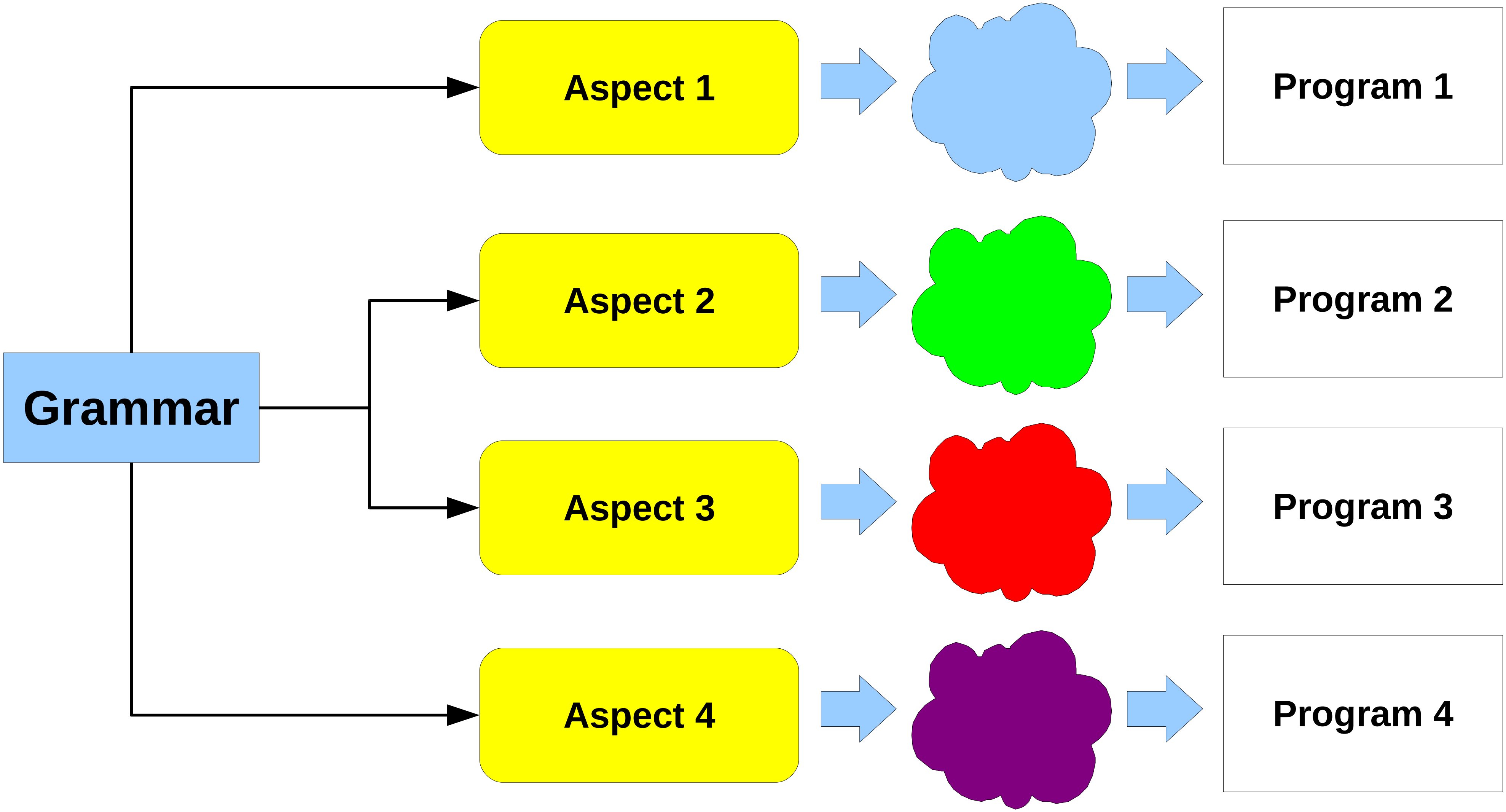
Classes

Generic Metadata **(Advice)**

- `size = 10`
- `name = John`
  - identifier
- `address = 'Nantes, France'`
- `date = {year = 2009; month = August; day = 28}`
  - tuple: set of attributes

-
  ```
  signature = {{
       int main(int argc, char[][] argv)
  }}
  ```
  - sequence of values
  - a small DSL framework

Recall the goal: We've reached it!



Problem 1: Readability

- Grammar
  - Pure
- Annotations
  - Semantically connected to the grammar
    - Hard to read when not seeing the grammar
  - Contain patterns
    - They outline the connection with the grammar
    - They introduce duplication
- A trade off between duplication and readability!

Problem 2: Duplication

- Many patterns may match the same object
  - This night be intended or not
  - Give the user some control
- Patterns duplicate parts of the grammar
  - Guarantee the consistency
  - Reduce the duplication
- Solutions:
  - Attribute assignment constraints
  - Match multiplicity constraints
  - Abstractions in patterns

Double assignment

```
fields : .. ;
  @ID : role = field;

# : ID .. ;
  @ID.role = method;
```

Attribute assignment constraints

```
fields : .. ;
  @ID : [once] role = field;

# : ID .. ;
  @ID.role = method;
```

- once – the second assignment is an error
- overwrite – the second assignment overwrites
- ignore – the second assignment is ignored

Match multiplicity constraints

```
[1..1] fields : .. ;
  @ID : [once] role = field;

[0..*] # : ID .. ;
  @ID.role = method;
```

Match multiplicity constraints

```
[1..1] fields : .. ;
  @ID : [once] role = field;

[0..*] # : ID .. ;
  @ID.role = method;
```

- The “fragile point cut” problem!

Abstraction in patterns

- Problems
  - Duplication of grammar objects
  - “Fragile” point cuts (patterns)
- Abstraction tools
  - Wildcards : **field** : **..** ;
    - Capture many sentences with the same structure
  - Subpatterns : @ID : role = *field*;
    - Capture all subsentences with the same structure
  - Abstract patterns : fieldRule();
    - Separate “models” and “views”

Abstract patterns

- Signature
  - Name
  - Variables (“public”)
- Aspect inheritance
  - Subaspects may implement pattern signatures
  - Can run only if everything is implemented
- Model-View analogy
  - A grammar is a “view”, UI
  - Annotations are a “model”, internal meaning

Approach Summary

- We obtain
  - Readable grammar specifications
  - Independent annotation sets (aspects)
- By using
  - A fixed **Grammar** language
    - universal for CF-grammars
  - A **Pattern** language
    - to locate grammar objects
  - A **Generic Metadata** language
    - To express custom annotations
    - May be replaced by DSLs <u>with the same AS</u>

Applications

- The approach is implemented in a tool called **Grammatic**, http://grammatic.googlecode.com
  - Grammar aspects
  - Grammar modularity & templates
- Already implemented with Grammatic
  - **ATF**, “*ANTLR, but with clean grammars*”
  - **G**rammar **T**ransformations **w**ith **A**spects
- In progress
  - IDE generation tool
    - Name analysis: navigation/renaming/completion

Extra slides

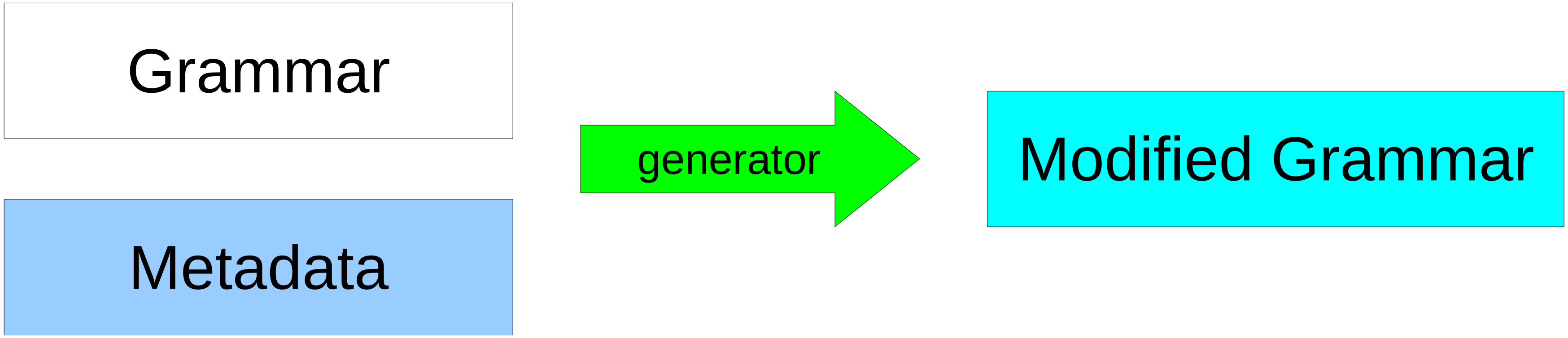
ATF: Attributed Translation Scheme

```
templateParameter
   : ('refer' | 'copy')? type operation NAME
   ;
# --> (TemplateParameter<?> result) {
   #before: {
      refer = Boolean.false();
      copy = Boolean.false();
   }
   'refer': refer = Boolean.true();
   'copy': copy = Boolean.true();
   #after: result = createTemplateParameter(
      refer, copy, type#, operation#, NAME#);
}
```

ATF Features

- Attributed Translation Framework (ATF)
  - Abstract
    - Functions are implemented externally
  - Language independent
  - Pluggable extensions
    - Type system
      - Local type inference
    - Declarations (types, imports, etc.)
  - Composable specifications
    - Many ATF advice applied to the same grammar may be merged together

Grammar Transformations



- Attributes are transformation instructions
  - (insert) a.before = << expression >>
  - (insert) a.after = << expression >>
  - a.instead = << expression >>
    - Remove: a.instead = <<>>

Creating Dialects

- Dialect
  - A language defined by providing changes for some other language
- Dialect types
  - Reductions
    - Language has a strictly smaller set of strings
  - Syntactical Extensions
    - Only “syntax sugar” is added
  - Semantical Extensions
    - New semantical notions are added

Toy language: Verifiable C

```
enum State { READY, WORKING, STOPPED };
void control()
{
    enum State state;
    state = READY;
    while (state != STOPPED) {
        switch (state) {
            case READY: state = WORKING;
                        onReadyToWorking();
                        break;
            case WORKING: state = STOPPED;
                          onWorkingToStopped();
                          break;
        }
    }
}
```

Reducing Dialect

```
typeSpecifier: structOrUnionSpecifier |
         enumSpecifier | typedefName | $types:...
   @types.instead = <<>>
   ;

typeSpecifier
   : structOrUnionSpecifier | enumSpecifier
                             | typedefName;

relationalExpression: shiftExpression
                            (.. shiftExpression)*
   @shiftExpression.instead = <<unaryExpression>>
   ;

relationalExpression
   : unaryExpression (relation unaryExpression)*;
```

Syntactical Extension

```
enum State { READY, WORKING, STOPPED };
void control()
{
    enum State state;
    state = READY;
    statemachine (state) {
        finish STOPPED;
        READY - onReadyToWorking() -> WORKING;
        WORKING - onWorkingToReady() -> STOPPED;
    }





}
```

Statemachine semantics

```
enum State { READY, WORKING, STOPPED };
void control()
{
    enum State state;
    state = READY;
    while (state != STOPPED) {
        switch (state) {
            case READY: state = WORKING;
                        onReadyToWorking();
                        break;
            case WORKING: state = STOPPED;
                          onWorkingToStopped();
                          break;
        }
    }
}
```

Integrating Syntax Sugar

```
statemachine :
  'statemachine' '(' $stateVar=ID ')' '{'
      'finish' $finishState=ID ';'
      $transitions=(
         $from=ID '-' functionCall '->' $to=ID
      )+
  '}' ;
#sugar {
    while (<stateVar> != <finishState>) {
        switch (<stateVar> {
        <transitions:
            case <from>:
                <functionCall>;
                <stateVar> = <to>;
                break;>
        }
    }
}
```

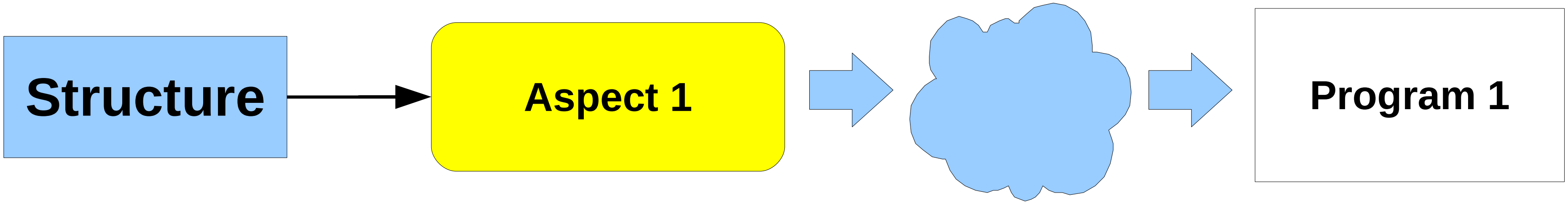
Semantical Extension

```
enum State { READY, WORKING, STOPPED };
void control()
  [(state==READY) -> F G (state==STOPPED)]
{
    enum State state;
    state = READY;
    statemachine (state) {
        finish STOPPED;
        READY - onReadyToWorking() -> WORKING;
        WORKING - onWorkingToReady() -> STOPPED;
    }




}
```

Dialects Summary

- Reductions
  - Almost seamless semantical transformation
- Syntactical Extensions
  - Syntax sugar is interpreted in terms of base syntax
- Semantical Extensions
  - Hopeless...

FW: Generalization



- Grammar → Structure
  - ADTs (Class diagram, Java interfaces, ...)
  - XML Schema (XSD)
  - RDB Schema (ER-diagram)
- Metadata is already generic
- We need a pattern language