Unification of Parsing and Reflective Printing

Zirun Zhu
National Institute of Informatics

joint work with Josh Ko, João Saraiva, Zhenjiang Hu

Spring Festival Workshop @ Karuizawa
21 March 2017
Parsers and printers

- Parsers and printers are needed for programming languages
  - parser: program text $\rightarrow$ tree structures
  - printer: tree structures $\rightarrow$ program text

```
1 * 2 + 3 / 4
```

```
parser

printer
```

```
Add

Mul

Div

1

2

3

4
```
Parsers and printers

- Parsers and printers are needed for programming languages
  - parser: program text $\rightarrow$ tree structures
  - printer: tree structures $\rightarrow$ program text

intuitive feeling: $\text{print . parse} = \text{id}$
$\text{parse . print} = \text{id}$
Mainstream practice

- Implement parser and printer independently:
  - Happy, parsec, etc. for building parser
  - pretty package, etc. for building printer
Disadvantages of mainstream practice

• Obviously, need to design and maintain two components, and

• at a **high risk** of failing to preserve consistency properties

![Diagram of text, parser, and tree with arrows indicating build by Happy and build by pretty.]
New attempt

• Ideas for unifying parsing and printing have already been proposed, with tools implemented.
  • industrial tools: Xtext\textsuperscript{[1]}, JetBrain MPS\textsuperscript{[2]}, ...
  • research papers: Invertible Syntax Descriptions\textsuperscript{[3]}, FliPpr\textsuperscript{[4]}, ...

\begin{itemize}
  \item [1] Eysholdt, Moritz, and Heiko Behrens. "Xtext: implement your language faster than the quick and dirty way.", 2010.
  \item [2] https://www.jetbrains.com/mps/
\end{itemize}
New attempt

- Ideas for unifying parsing and printing have already been proposed, with tools implemented.
  - industrial tools: Xtext\textsuperscript{[1]}, JetBrains MPS\textsuperscript{[2]}, ...
  - research papers: Invertible Syntax Descriptions\textsuperscript{[3]}, Flipp\textsuperscript{[4]}, ...

- Do not pay enough attention to the properties.

\[\text{print . parse } = \text{id}\]
\[\text{parse . print } = \text{id}\]
New attempt

• Ideas for unifying parsing and printing have already been proposed, with tools implemented.
  • industrial tools: Xtext\textsuperscript{[1]}, JetBrain MPS\textsuperscript{[2]}, …
  • research papers: Invertible Syntax Descriptions\textsuperscript{[3]}, FliPpr\textsuperscript{[4]}, …

• Do not pay enough attention to the properties.

\[
\begin{align*}
\text{print . parse} &= \text{id} \\
\text{parse . print} &= \text{id}
\end{align*}
\]
New attempt

• Ideas for unifying parsing and printing have already been proposed, with tools implemented.
  • industrial tools: Xtext\textsuperscript{[1]}, JetBrain MPS\textsuperscript{[2]}, …
  • research papers: Invertible Syntax Descriptions\textsuperscript{[3]}, FliPpr\textsuperscript{[4]}, …

• Do not pay enough attention to the properties.

\[
\begin{align*}
\text{print} \cdot \text{parse} &= \text{id} \\
\text{parse} \cdot \text{print} &= \text{id}
\end{align*}
\]
Importance of consistency properties

• Two properties:
  \[
  \begin{align*}
  \text{print} \cdot \text{parse} &= \text{id} \\
  \text{parse} \cdot \text{print} &= \text{id}
  \end{align*}
  \]

• Semantic equivalence: of course, the bottom line!
  • we do not want \(1 + 1\) becomes \(75 \times 3\) or something worse...

• Syntactic equivalence:
  • suppose \(a \land b\) is equivalent to \(\text{if } a \text{ then } b \text{ else } 0\) \[^1\]

\[
\begin{align*}
\text{let } x &= \text{if } a \text{ then } b \text{ else } 0 \\
\text{in } &\ldots
\end{align*}
\quad \longleftrightarrow \quad
\begin{align*}
\text{let } x &= a \land b \\
\text{in } &\ldots
\end{align*}
\]

\[^1\] For example, in Tiger language. (Introduced by Appel, Andrew W, Modern Compiler Implementation in C, 2004.)
A typical situation in code refactoring

program text

```plaintext
// the expression
// evaluates ...
(- α) / (1 + α)
```
A typical situation in code refactoring

program text

```
// the expression
// evaluates ...
(- α) / (1 + α)
```

syntactic sugar

![AST diagram]

```
Div
  Sub
    0
    α
  Add
    1
    α
```
A typical situation in code refactoring

program text

```
// the expression
// evaluates ...
(- \alpha) / (1 + \alpha)
```

parse

variable renaming
A typical situation in code refactoring

Program text:

// the expression
// evaluates ...
(- α) / (1 + α)

AST:

```
Div
    Sub
        0
        α
    Add
        1
        α
```

Variable renaming:

```
Div
    Sub
        0
        β
    Add
        1
        β
```
A typical situation in code refactoring

program text

// the expression
// evaluates ...
(- α) / (1 + α)

// the expression
// evaluates ...
(- β) / (1 + β)

(0 - β) / (1 + β)
A typical situation in code refactoring

program text

```c
// the expression
// evaluates ...
(- α) / (1 + α)
```

syntactic sugar

```c
// the expression
// evaluates ...
(- β) / (1 + β)
```

```
(0 - β) / (1 + β)
```

AST

```
Div
Sub  Add
0 α  1 α
```

parse

variable renaming

```
Div
Sub  Add
0 β  1 β
```

print ?
A typical situation in code refactoring

program text

```
// the expression
// evaluates ...
(- α) / (1 + α)
```

syntax sugar, comments

```
// the expression
// evaluates ...
(- β) / (1 + β)
```

```
(0 - β) / (1 + β)
```
A typical situation in code refactoring

program text

```plaintext
// the expression
// evaluates ...
(- α) / (1 + α)
```

syntactic sugar, comments, layouts

```plaintext
// the expression
// evaluates ...
(- β) / (1 + β)
```

```
(0 - β) / (1 + β)
```

AST

parse

```
Div
   Sub
     0
     α
   Add
     1
     α
```

variable renaming

print ?

```
Div
   Sub
     0
     β
   Add
     1
     β
```
A typical situation in code refactoring

program text

```plaintext
// the expression
// evaluates ...
\( (-\ \alpha) \div (1 + \alpha) \)
```

syntactic sugar, comments, layouts

```plaintext
// the expression
// evaluates ...
\( (-\ \beta) \div (1 + \beta) \)
```

```plaintext
(0 - \beta) / (1 + \beta)
```

AST

```
Div
  Sub
    0
  Add
    1
    \beta
```

variable renaming

```
Div
  Sub
    0
  Add
    1
    \beta
```

parse

```
Div
  Sub
    0
  Add
    \alpha
    1
    \alpha
```

print ?
Reflective printer

When an abstract syntax tree corresponding to a piece of program text is modified, the printer can reflect the modification to the program text while preserving syntactic sugar, comments, and layouts.
How to preserve syntactic equivalence

• To preserve print . parse = id, we need to know "which part of the AST comes from which syntactic object"

• Mainstream practice: contaminate ASTs with more information
  • add line and column numbers for debugging
  • augment terms in ASTs with tags for resugaring[1]
  • ...

• Our approach: adopt bidirectional transformations and take program text as an additional input

We adopt the state-based (asymmetric) lens framework of bidirectional transformations (BX for short), where:

- a BX consists of a pair of transformations — _get_ and _put_ — satisfying well-behavedness laws[1]:

- _get_ : _source_ → _view_ abstracts the source as a view

Background: bidirectional transformations

- We adopt the state-based (asymmetric) lens framework of bidirectional transformations (BX for short), where:

  - a BX consists of a pair of transformations — get and put — satisfying well-behavedness laws[1]:
  
    - put : $source \times view \rightarrow source$ updates the source

• We adopt the state-based (asymmetric) lens framework of bidirectional transformations (BX for short), where:
  
  • a BX consists of a pair of transformations — *get* and *put* — satisfying well-behavedness laws[1]:

  • *get*: *source* → *view* abstracts the source as a view

  • *put*: *source* × *view* → *source* updates the source

Well-behavedness

• _get_ and _put_ must satisfy:
  
  • PutGet (correctness): \( \text{get (put } s \ v) = v \)
  
  • GetPut (hippocraticness): \( \text{put } s \ (\text{get } s) = s \)
Well-behavedness

• *get* and *put* must satisfy:
  
  • **PutGet** (correctness): \( get \left( put \ s \ v \right) = v \)
  
  • **GetPut** (hippocraticness): \( put \ s \left( get \ s \right) = s \)
Well-behavedness

- \textit{get} and \textit{put} must satisfy:
  - \textbf{PutGet} (correctness): \( \text{get} (\text{put} s v) = v \)
  - \textbf{GetPut} (hippocraticness): \( \text{put} s (\text{get} s) = s \)

\[
\begin{align*}
\text{Div} & \quad \text{Add} \\
0 & \quad \alpha & 1 & \quad \alpha
\end{align*}
\]

1. put

\[
\frac{0 - \alpha}{1 + \alpha}
\]
Well-behavedness

- *get* and *put* must satisfy:
  - **PutGet** (correctness): \( \text{get (put } s \ v) = v \)
  - **GetPut** (hippocraticness): \( \text{put } s \ (\text{get } s) = s \)

\[
\begin{align*}
\text{1. put } & (0 - \alpha) / (1 + \alpha) \\
\text{2. get } & \text{L}
\end{align*}
\]
Well-behavedness

- **get** and **put** must satisfy:
  - **PutGet** (correctness): \( \text{get} \ (\text{put} \ s \ v) = v \)
  - **GetPut** (hippocraticness): \( \text{put} \ s \ (\text{get} \ s) = s \)

\[
\frac{0 - \alpha}{1 + \alpha} \quad \text{sub}
\]

\[
\frac{0}{1 + \alpha} \quad \text{add}
\]

\[
\frac{0}{1 + \alpha} \quad \text{div}
\]

\[
\frac{0}{1 + \alpha} \quad \text{add}
\]

\[
\frac{0}{1 + \alpha} \quad \text{div}
\]

\[
\frac{0}{1 + \alpha} \quad \text{add}
\]
Well-behavedness

• *get* and *put* must satisfy:
  • **PutGet** (correctness): \( \text{get} (\text{put} s v) = v \)
  • **GetPut** (hippocraticness): \( \text{put} s (\text{get} s) = s \)

// the expression
(- α) / (1 + α)

GetPut
Well-behavedness

- *get* and *put* must satisfy:
  - **PutGet** (correctness): \( get (put s v) = v \)
  - **GetPut** (hippocraticness): \( put s (get s) = s \)

// the expression \((- \alpha) / (1 + \alpha)\)
Well-behavedness

- *get* and *put* must satisfy:
  - **PutGet** (correctness): \( \text{get (put } s \ v) = v \)
  - **GetPut** (hippocraticness): \( \text{put } s \ (\text{get } s) = s \)

```plaintext
// the expression 
(- \alpha) / (1 + \alpha)

1. get

Div
Sub 0  a
Add 1  a

2. put
```
Well-behavedness

- *get* and *put* must satisfy:
  - **PutGet** (correctness): \( \text{get (put } s \ v) = v \)
  - **GetPut** (hippocraticness): \( \text{put } s \ (\text{get } s) = s \)
Well-behavedness $\rightarrow$ consistency

- *get* and *put* must satisfy:
  - **PutGet** (correctness): $\text{get} (\text{put} \ s \ v) = v$
  - **GetPut** (hippocraticness): $\text{put} \ s \ (\text{get} \ s) = s$

- Consistency properties between parser and printer

\[
\text{parse} \ (\text{print} \ s \ t) = t
\]
\[
\text{print} \ s \ (\text{parse} \ s) = t
\]
Looks good and we implemented a DSL
Overview of the implementation

Overview of the implementation

BiYacc program

special lexer

parser (Happy)

BiGUL\textsuperscript{1} program

text

tokens

CST

AST

isomorphism, trivial bx

simple printer

generate
Overview of the implementation

Example: parser & printer for arithmetic expressions

• In our DSL, users need to
  • describe the abstract syntax, in valid Haskell syntax:

```haskell
data Arith = Num Int
           | Add Arith Arith
           | Sub Arith Arith
           | Mul Arith Arith
           | Div Arith Arith
```
Example: parser & printer for arithmetic expressions

- In our DSL, users need to
  - describe the abstract syntax, in valid Haskell syntax,
  - the concrete syntax, using production rules:

```
%LineComment '//' ;
%BlockComment '/*' '*/' ;

Expr -> Expr '+' Term
   | Expr '-' Term
   | Term ;

Term -> Term '*' Factor
   | Term '/' Factor
   | Factor ;

Factor -> '-' Factor
   | Int
   | '()' Expr ')' ;
```
Example: parser & printer for arithmetic expressions

• In our DSL, users need to
  • describe the abstract syntax, in valid Haskell syntax,
  • the concrete syntax, using production rules,
  • and a short program describing relations between the two
Arith -> Expr
Add l r -> (l -> Expr) '+' (r -> Term)
Sub l r -> (l -> Expr) '-' (r -> Term)
e -> (e -> Term)

Arith -> Term
Mul l r -> (l -> Term) '*' (r -> Factor)
Div l r -> (l -> Term) '/' (r -> Factor)
e -> (e -> Factor)

Arith -> Factor
Sub (Num 0) r -> '-' (r -> Factor)
Num n -> (n -> Int)
e -> '=(' (e -> Expr) ')')
Arith -> Expr

Add l r -> (l -> Expr) '+' (r -> Term)
Sub l r -> (l -> Expr) '-' (r -> Term)
e -> (e -> Term)

Arith -> Term

Mul l r -> (l -> Term) '*' (r -> Factor)
Div l r -> (l -> Term) '/' (r -> Factor)
e -> (e -> Factor)

Arith -> Factor

Sub (Num 0) r -> '-' (r -> Factor)
Num n -> (n -> Int)
e -> '()' (e -> Expr) ')


Arith ➔ Expr

<table>
<thead>
<tr>
<th>Rule</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Add l r</td>
<td>$(l \rightarrow \text{Expr}) , '+' , (r \rightarrow \text{Term})$</td>
</tr>
<tr>
<td>Sub l r</td>
<td>$(l \rightarrow \text{Expr}) , '-' , (r \rightarrow \text{Term})$</td>
</tr>
<tr>
<td>e</td>
<td>$(e \rightarrow \text{Term})$</td>
</tr>
</tbody>
</table>

Arith ➔ Term

<table>
<thead>
<tr>
<th>Rule</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mul l r</td>
<td>$(l \rightarrow \text{Term}) , '*' , (r \rightarrow \text{Factor})$</td>
</tr>
<tr>
<td>Div l r</td>
<td>$(l \rightarrow \text{Term}) , '/' , (r \rightarrow \text{Factor})$</td>
</tr>
<tr>
<td>e</td>
<td>$(e \rightarrow \text{Factor})$</td>
</tr>
</tbody>
</table>

Arith ➔ Factor

<table>
<thead>
<tr>
<th>Rule</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sub (Num 0) r</td>
<td>'$-' , (r \rightarrow \text{Factor})$</td>
</tr>
<tr>
<td>Num n</td>
<td>$(n \rightarrow \text{Int})$</td>
</tr>
<tr>
<td>e</td>
<td>$(e \rightarrow \text{Expr}) , ')'$</td>
</tr>
</tbody>
</table>
Arith \rightarrow Expr

\begin{align*}
\text{Add} & \rightarrow (l \rightarrow \text{Expr}) \ ' + ' \ (r \rightarrow \text{Term}) \\
\text{Sub} & \rightarrow (l \rightarrow \text{Expr}) \ ' - ' \ (r \rightarrow \text{Term}) \\
\text{e} & \rightarrow (e \rightarrow \text{Term})
\end{align*}

Arith \rightarrow Term

\begin{align*}
\text{Mul} & \rightarrow (l \rightarrow \text{Term}) \ ' * ' \ (r \rightarrow \text{Factor}) \\
\text{Div} & \rightarrow (l \rightarrow \text{Term}) \ ' / ' \ (r \rightarrow \text{Factor}) \\
\text{e} & \rightarrow (e \rightarrow \text{Factor})
\end{align*}

Arith \rightarrow Factor

\begin{align*}
\text{Sub (Num 0)} & \rightarrow \ ' - ' \ (r \rightarrow \text{Factor}) \\
\text{Num} \ n & \rightarrow (n \rightarrow \text{Int}) \\
\text{e} & \rightarrow \ ' ( (e \rightarrow \text{Expr}) ' ) '
\end{align*}
### Arith -> Expr

<table>
<thead>
<tr>
<th>Operator</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>Add L R</td>
<td>((L +&gt; \text{Expr})'+'(R +&gt; \text{Term}))</td>
</tr>
<tr>
<td>Sub L R</td>
<td>((L +&gt; \text{Expr})'-'(R +&gt; \text{Term}))</td>
</tr>
<tr>
<td>e</td>
<td>((e +&gt; \text{Term}))</td>
</tr>
</tbody>
</table>

### Arith -> Term

<table>
<thead>
<tr>
<th>Operator</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mul L R</td>
<td>((L +&gt; \text{Term})'\ast'(R +&gt; \text{Factor}))</td>
</tr>
<tr>
<td>Div L R</td>
<td>((L +&gt; \text{Term})'/'(R +&gt; \text{Factor}))</td>
</tr>
<tr>
<td>e</td>
<td>((e +&gt; \text{Factor}))</td>
</tr>
</tbody>
</table>

### Arith -> Factor

<table>
<thead>
<tr>
<th>Operator</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sub (Num 0) R</td>
<td>'-' (R +&gt; Factor)</td>
</tr>
<tr>
<td>Num n</td>
<td>((n +&gt; \text{Int}))</td>
</tr>
<tr>
<td>e</td>
<td>('(\text{Expr}'))</td>
</tr>
</tbody>
</table>
Looks good but problems remain ...
Looks good but problems remain ... 

- For unambiguous grammars, the story finished, however, regarding ambiguous grammars...

\[
\begin{align*}
\text{print} \cdot \text{parse} &= \text{id} \\
\text{parse} \cdot \text{print} &= \text{id}
\end{align*}
\]

where can be wrong?
Agreement on parsing and printing

• Given a Happy (or Yacc) program,

\[
\text{Expr} : \text{Expr} \ ' + ' \ \text{Expr} \ \{\text{Add} \ $1 \ $3\} \\
\quad | \quad \text{Expr} \ ' * ' \ \text{Expr} \ \{\text{Mul} \ $1 \ $3\} \\
\quad | \quad \ldots
\]

\[
\text{print} \ . \ \text{backward} \ . \ \text{forward} \ . \ \text{parse} = \text{id} \\
\text{forward} \ . \ \text{parse} \ . \ \text{print} \ . \ \text{backward} = \text{id}
\]
Agreement on parsing and printing

- Given a Happy (or Yacc) program,

\[
\text{Expr} : \begin{cases} \\
\text{Expr} + \text{Expr} \{\text{Add} \; \$1 \; \$3\} \\
\text{Expr} * \text{Expr} \{\text{Mul} \; \$1 \; \$3\} \\
\ldots
\end{cases}
\]

\[
\text{program text} \rightarrow (\text{concrete})\text{ parse} \rightarrow \text{CST} \rightarrow \text{forward trans.} \rightarrow \text{AST} \rightarrow (\text{concrete})\text{ print} \rightarrow (\text{concrete})\text{ print}
\]

print . backward . forward . parse = id
forward . parse . print . backward = id

CST: concrete syntax tree
AST: abstract syntax tree
Agreement on parsing and printing

• Given a Happy (or Yacc) program,

\[
\text{Expr} : \text{Expr} \, ' + ' \, \text{Expr} \{ \text{Add} \, \$1 \, \$3 \} \\
| \text{Expr} \, ' * ' \, \text{Expr} \{ \text{Mul} \, \$1 \, \$3 \} \\
| \ldots
\]

\begin{align*}
\text{print} \cdot \text{backward} \cdot \text{forward} \cdot \text{parse} & = \text{id} \\
\text{forward} \cdot \text{parse} \cdot \text{print} \cdot \text{backward} & = \text{id}
\end{align*}

CST: concrete syntax tree
AST: abstract syntax tree
Agreement on parsing and printing

• Given a Happy (or Yacc) program,

```
Expr : Expr '!' Expr
    | Expr '*' Expr {Mul $1 $3}
    | ...
```

(print . backward . forward . parse = id)

CST: concrete syntax tree
AST: abstract syntax tree
When the grammar is ambiguous

- Ambiguous grammar

```
(a & b & c) (concrete) parse
```

```
( & c)
( & b)
( & a)
```

```
( & a)
( & b)
```

```
( & c)
```

```
this ?
```

```
this ?
```

```
this ?
```
When the grammar is ambiguous

- Ambiguous grammar + disambiguation

(a & b & c) 

(Concrete) parse

(ban this?)
When the grammar is ambiguous

- Ambiguous grammar + disambiguation

a & b & c

1 print

2 parse

... parse . print . ... = id
is broken

ban this ?
When the grammar is ambiguous

• Consider the process as a whole, thus with CST hidden

(a & b) & c

(a & b) & c

A & A & c

a & b

a & b
When the grammar is ambiguous

- Consider the process as a whole, thus with CST hidden

\[(a \& b) \& c\]

add "proper parentheses"\[1\] according to the disambiguation

\[(a \& (b \& c))\]

When the grammar is ambiguous

- Consider the process as a whole, thus with CST hidden

\[(a \& b) \& c\]

add "proper parentheses"\textsuperscript{[1]}

according to the disambiguation

\[(a \& b) \& c\]

When the grammar is ambiguous

- Consider the process as a whole, thus with CST hidden

\[(a \& b) \& c\]

add "proper parentheses"[1] according to the disambiguation

\[\left((a \& b)\right) \& c\]

When the grammar is ambiguous

- Consider the process as a whole, thus with CST hidden

\[ a \& b \& c \]

add "proper parentheses"[1] according to the disambiguation

\[ a \& (b \& c) \]

When the grammar is ambiguous

- Consider the process as a whole, thus with CST hidden

 add "proper parentheses"\[^1\] according to the disambiguation

\[^1\] Van Den Brand, Mark, and Eelco Visser. "Generation of formatters for context-free languages.," 1996.
Solution

• Need clear definitions of disambiguation rules which works for both: disambiguation filters\[1\][2]

• In addition, we may build a **CST-centric framework** for building consistency parsers and printers:
  
  • program text and AST as two different views
  
  • disambiguation filters as additional restrictions on CSTs
  
  • ...

Looks good but could be "better" ...
Looks good but could be "better" ...

• PutGet law (get (put s v) == v) say nothing about how the program text changes with an updated AST.

• Produce "better" program text
  • least surprise\(^1\) / least change\(^2\) for updated text
  • pretty layouts for newly generated text

Least surprise / least change

- The core part is still the alignment problem, example:

```
\begin{align*}
\text{e0} & \quad \text{parse} \quad \text{t0} \quad \text{modify and}\nonumber \\
\text{e1} & \quad \text{t1} \quad \text{rearrange} \quad \text{t2'} \\
\text{e2} & \quad \text{t0'} \\
\end{align*}
```
Least surprise / least change

- The core part is still the alignment problem, example:

naive behaviour: all is rebuilt, all is lost.

desired behaviour: smart matching.
Solutions

• Have new syntax to write users' own strategies
  Adaptive: [ | f :: s -> v -> Bool ]
  [ | f :: s -> v -> s ]

• For lists, we can align elements by
  • position, key, "similarity", edit distance, ... [1][2][3]...

• For trees, how to align nodes ...
  • delta-based BX may help [4][5]...


Summary

• Unification of parsing and printing using BX while guaranteeing
  
  \[ \text{print} \cdot \text{parse} = \text{id} \]
  
  \[ \text{parse} \cdot \text{print} = \text{id} \]

• Progress (need discussions) :
  
  • list alignment (95%)
  
  • disambiguation as filters for both directions (70%)
  
  • scannerless & generalised LR parsing (50%)
  
  • tree alignment (5%)
  
  • elegant and precise syntax for disambiguation filters
  
  • CST centric framework for parsing and printing (idea only : ) )
Thank you
Detailed construction of a BiYacc program
Program in BiYacc

\[
\text{Arith} \rightarrow \text{Expr} \\
\text{Add} \ l \ r \rightarrow \text{(l} \ +\rightarrow \text{Expr}) \ '+' \ (r \ +\rightarrow \text{Term}) \\
\text{Sub} \ l \ r \rightarrow \text{(l} \ +\rightarrow \text{Expr}) \ '-' \ (r \ +\rightarrow \text{Term}) \\
\text{e} \ +\rightarrow \text{(e} \ +\rightarrow \text{Term})
\]

\[
\text{Arith} \rightarrow \text{Term} \\
\text{Mul} \ l \ r \rightarrow \text{(l} \ +\rightarrow \text{Term}) \ '*' \ (r \ +\rightarrow \text{Factor}) \\
\text{Div} \ l \ r \rightarrow \text{(l} \ +\rightarrow \text{Term}) \ '/' \ (r \ +\rightarrow \text{Factor}) \\
\text{e} \ +\rightarrow \text{(e} \ +\rightarrow \text{Factor})
\]

\[
\text{Arith} \rightarrow \text{Factor} \\
\text{Sub} \ (\text{Num } 0) \ r \rightarrow \ '-' \ (r \ +\rightarrow \text{Factor}) \\
\text{Num} \ n \ +\rightarrow \ (n \ +\rightarrow \text{Int}) \\
\text{e} \ +\rightarrow \ '(\text{(e} \ +\rightarrow \text{Expr})')
\]
Arith $\rightarrow$ Expr

relate ASTs to CSTs of different shape

Arith $\rightarrow$ Term

Arith $\rightarrow$ Factor
Arith $\rightarrow$ Expr
Add $l \, r$ $\rightarrow$

shape of the AST

Arith $\rightarrow$ Term

Arith $\rightarrow$ Factor
\[
\text{Arith} \rightarrow \text{Expr} \\
\text{Add} \ l \ r \rightarrow \text{Expr} \ '++' \ \text{Term}
\]

\[
\text{Arith} \rightarrow \text{Term} \\
\text{shape of the CST (or program text)} \\
= \text{production rules}
\]

\[
\text{Arith} \rightarrow \text{Factor}
\]
\[
\begin{align*}
\text{Arith} & \rightarrow \text{Expr} \\
\text{Add } l \ r & \rightarrow (l \rightarrow \text{Expr}) \ '+' (r \rightarrow \text{Term})
\end{align*}
\]

use \textit{l} (left operand of Add) to update the \textit{Expr} part

Arith \rightarrow \text{Term}

keep the '+' (and following layouts/comments) there

Arith \rightarrow \text{Factor}
Arith $\rightarrow$ Expr
Add $l \, r \rightarrow (l \rightarrow \text{Expr}) \, '+' \, (r \rightarrow \text{Term})$
Sub $l \, r \rightarrow (l \rightarrow \text{Expr}) \, '-' \, (r \rightarrow \text{Term})$

Arith $\rightarrow$ Term

Arith $\rightarrow$ Factor
Arith $\rightarrow$ Expr
Add $l$ $r$ $\rightarrow$ $(l \rightarrow$ Expr $)'+' (r $\rightarrow$ Term)
Sub $l$ $r$ $\rightarrow$ $(l \rightarrow$ Expr $)'-' (r $\rightarrow$ Term)
e $\rightarrow$ (e $\rightarrow$ Term)

Arith $\rightarrow$ Term

otherwise, produce a term

Arith $\rightarrow$ Factor


\textbf{Arith} $\Rightarrow$ \textbf{Expr}
\begin{align*}
\text{Add} & \mid l \mid r \Rightarrow (l \Rightarrow \text{Expr}) \ ' + ' \ (r \Rightarrow \text{Term}) \\
\text{Sub} & \mid l \mid r \Rightarrow (l \Rightarrow \text{Expr}) \ ' - ' \ (r \Rightarrow \text{Term}) \\
\text{e} & \Rightarrow (e \Rightarrow \text{Term})
\end{align*}

\textbf{Arith} $\Rightarrow$ \textbf{Term}
\begin{align*}
\text{Mul} & \mid l \mid r \Rightarrow (l \Rightarrow \text{Term}) \ ' * ' \ (r \Rightarrow \text{Factor}) \\
\text{Div} & \mid l \mid r \Rightarrow (l \Rightarrow \text{Term}) \ '/ ' \ (r \Rightarrow \text{Factor}) \\
\text{e} & \Rightarrow (e \Rightarrow \text{Factor})
\end{align*}

\textbf{Arith} $\Rightarrow$ \textbf{Factor}
Arith \rightarrow Expr
Add \ l \ r \rightarrow (\ l \rightarrow Expr) \ '+' (\ r \rightarrow Term)
Sub \ l \ r \rightarrow (\ l \rightarrow Expr) \ '-' (\ r \rightarrow Term)
e \rightarrow (e \rightarrow Term)

Arith \rightarrow Term
Mul \ l \ r \rightarrow (\ l \rightarrow Term) \ '*' (\ r \rightarrow Factor)
Div \ l \ r \rightarrow (\ l \rightarrow Term) \ '/' (\ r \rightarrow Factor)
e \rightarrow (e \rightarrow Factor)

Arith \rightarrow Factor
Sub (Num 0) \ r \rightarrow '-' (\ r \rightarrow Factor)
Num \ n \rightarrow (n \rightarrow Int)
e \rightarrow (\ ' (e \rightarrow Expr) ')'

produce the negation
\textbf{Arith} \rightarrow \textbf{Expr}

Add \ l \ r \rightarrow (l \rightarrow \text{Expr}) \ '+' (r \rightarrow \text{Term})
Sub \ l \ r \rightarrow (l \rightarrow \text{Expr}) \ '-' (r \rightarrow \text{Term})
e \quad \rightarrow (e \rightarrow \text{Term})

\textbf{Arith} \rightarrow \textbf{Term}

Mul \ l \ r \rightarrow (l \rightarrow \text{Term}) \ '*' (r \rightarrow \text{Factor})
Div \ l \ r \rightarrow (l \rightarrow \text{Term}) \ '/' (r \rightarrow \text{Factor})
e \quad \rightarrow (e \rightarrow \text{Factor})

\textbf{Arith} \rightarrow \textbf{Factor}

Sub (\text{Num} \ 0) \ r \rightarrow ' - ' (r \rightarrow \text{Factor})
Num \ n \rightarrow (n \rightarrow \text{Int})
e \quad \rightarrow '( (e \rightarrow \text{Expr}) ')