Source Transformation on Boolean Grammars

Advantages and Challenges

James R. Cordy
Andrew Stevenson

Queen's University
Kingston, Canada

Boolean grammars

$$A = \alpha_1 \mid \alpha_2 \mid \alpha_3 \ \ldots$$

Boolean grammars

$$A = \alpha_1 \mid \alpha_2 \mid \alpha_3 \ \ldots$$

A. Okhotin, "Boolean grammars", *Information and Computation* 194 (2004).

$$A = \alpha_1 \ \& \ \alpha_2 \ \& \ \alpha_3 \ \ldots$$

Boolean grammars

$$A = \alpha_1 \mid \alpha_2 \mid \alpha_3 \ \ldots$$

A. Okhotin, "Boolean grammars", *Information and Computation* 194 (2004).

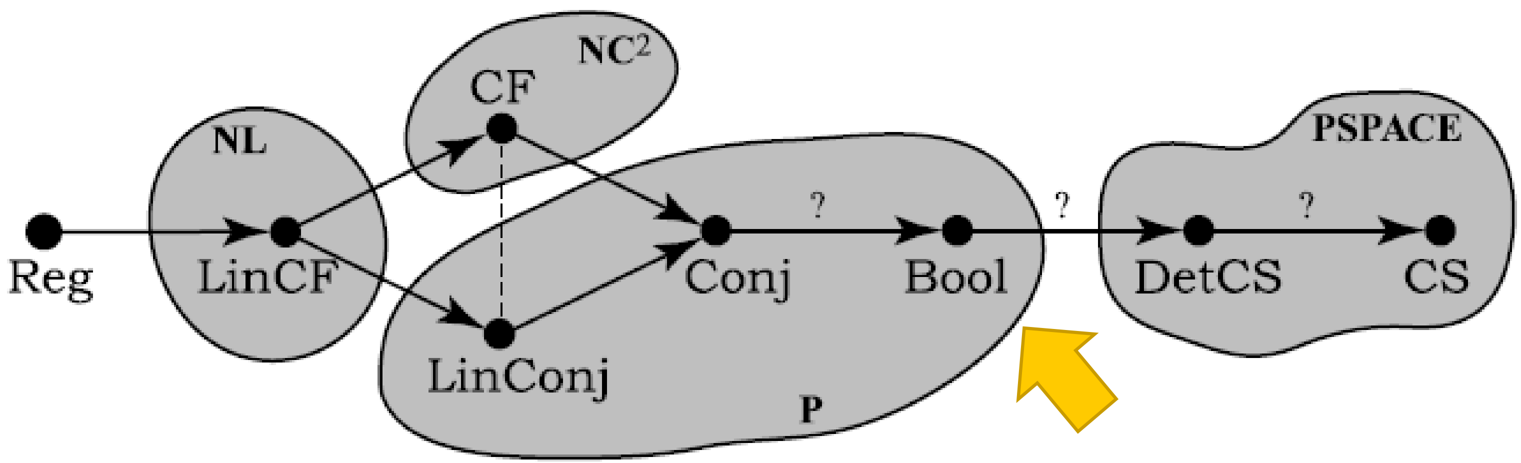
$$A = \alpha_1 \ \& \ \alpha_2 \ \& \ \alpha_3 \ \ldots$$

$$A = \alpha_1 \ \& \ \alpha_2 \ \& \ \alpha_3 \ \ldots \ \&\!\sim \beta_1 \ \&\!\sim \beta_2 \ \ldots$$

Boolean grammars

$A \ = \ \alpha_1 \ | \ \alpha_2 \ | \ \alpha_3 \ \ldots$

$A \ = \ \alpha_1 \ \& \ \alpha_2 \ \& \ \alpha_3 \ \ldots \ \&\sim \ \beta_1 \ \&\sim \ \beta_2 \ \ldots$



Boolean grammars

$$A \ = \ \alpha_1 \ | \ \alpha_2 \ | \ \alpha_3 \ \ldots$$

$$A \ = \ \alpha_1 \ \& \ \alpha_2 \ \& \ \alpha_3 \ \ldots \ \&\!\sim \beta_1 \ \&\!\sim \beta_2 \ \ldots$$

A. Okhotin, "LR parsing for Boolean grammars",
Int. J. Foundations of Computer Science 17:3 (2006) – $o\ (n^4)$

A. Okhotin, "Recursive descent parsing for Boolean grammars",
Acta Informatica 44:3-4 (2007) – $o\ (n)$ .. $o(2^n)$, LL-ish subset $o(n)$

Boolean grammars

$$ABC = AB\ c^* \ \&\ a^*\ BC$$

$$AB = a\ AB\ b \mid \epsilon$$

$$BC = b\ BC\ c \mid \epsilon$$

Boolean grammars

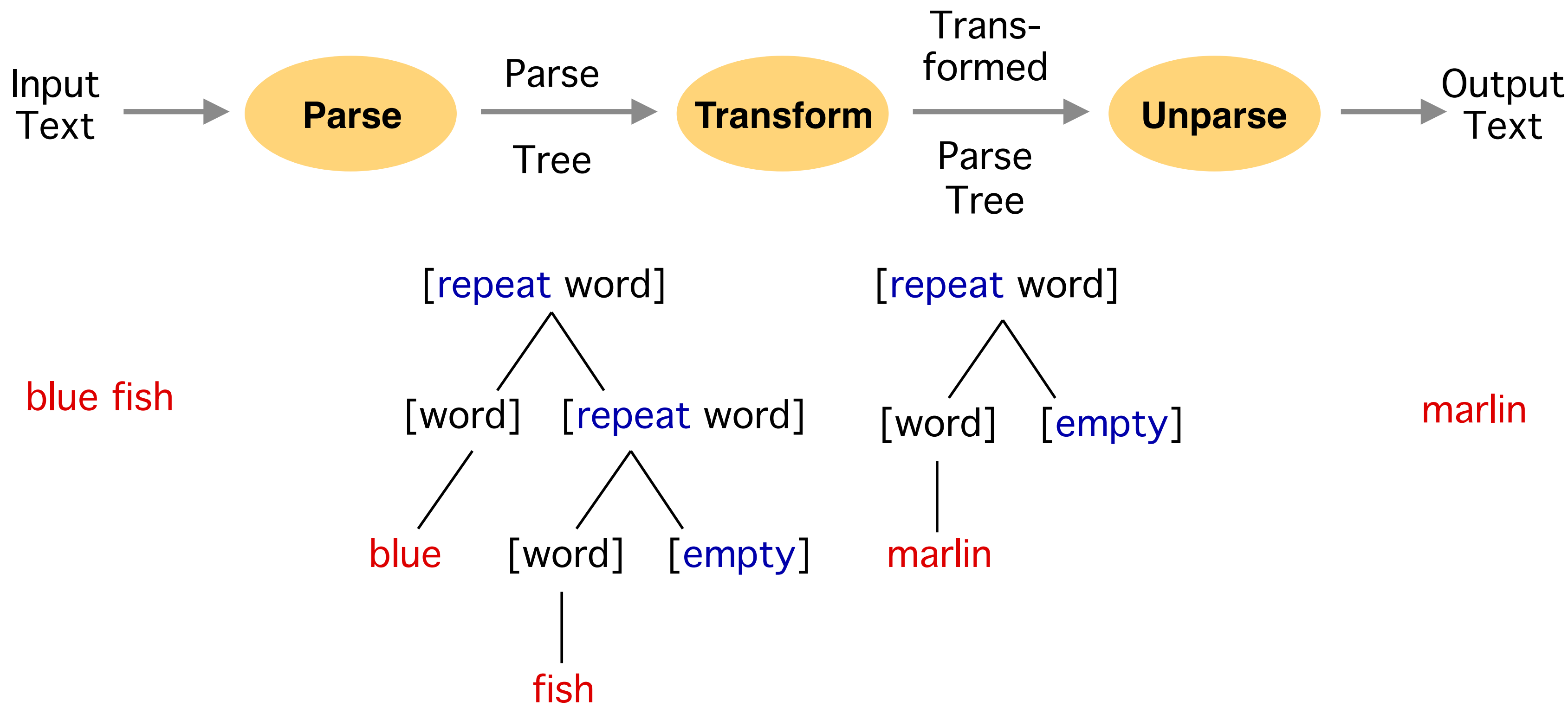
$$ABC = AB\ c^* \ \&\ a^*\ BC$$

$$AB = a\ AB\ b \mid \epsilon$$

$a^n b^n c^n$

$$BC = b\ BC\ c \mid \epsilon$$

TXL

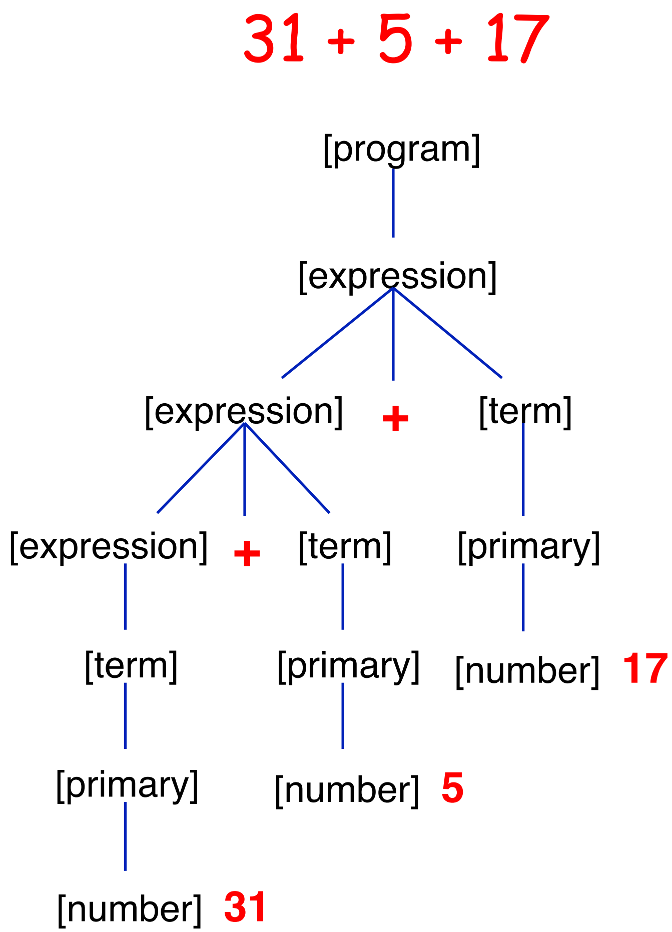


J.R. Cordy, C.D. Halpern and E. Promislow, "TXL: A Rapid Prototyping System for Programming Language Dialects", *Computer Languages* 16:1 (1991)

J.R. Cordy, “The TXL source transformation language”, *Science of Computer Programming* 61:3 (2006)

TXL

```
define program                  % goal symbol of input
    [expression]
end define

define expression
    [term]
  | [expression] + [term]
  | [expression] – [term]
end define

define term
    [primary]
  | [term] * [primary]
  | [term] / [primary]
end define

define primary
    [number]
  | ( [expression] )
end define
```

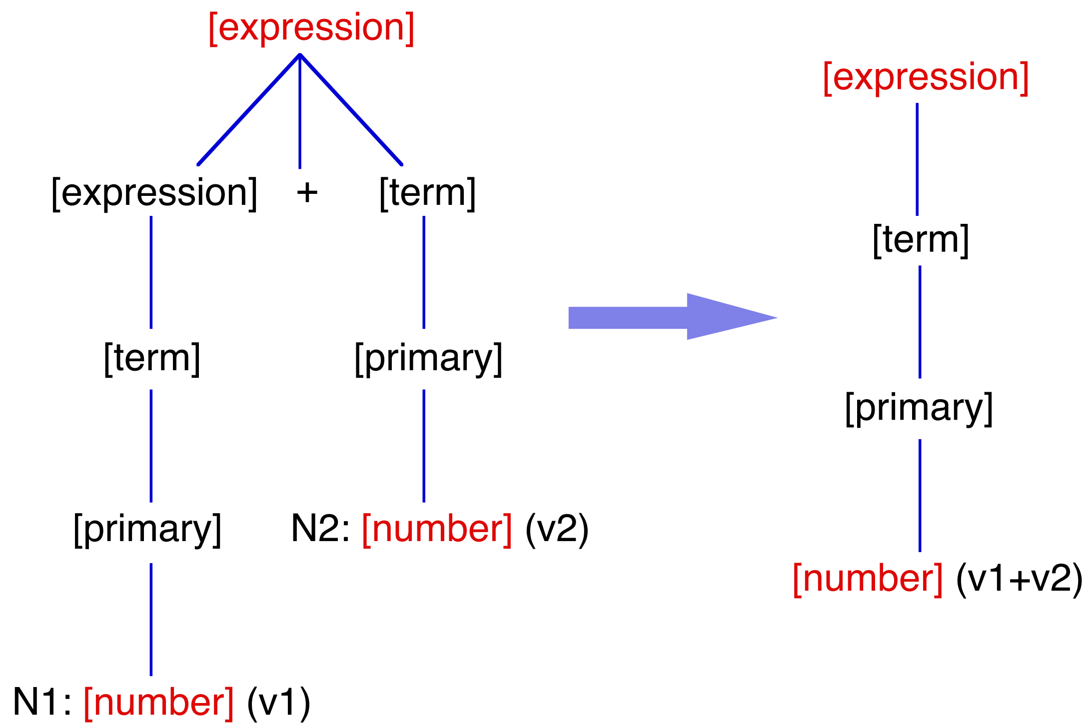


Top down, backtracking recursive descent parser (similar to ANTLR)

Left-recursion heuristic

TXL

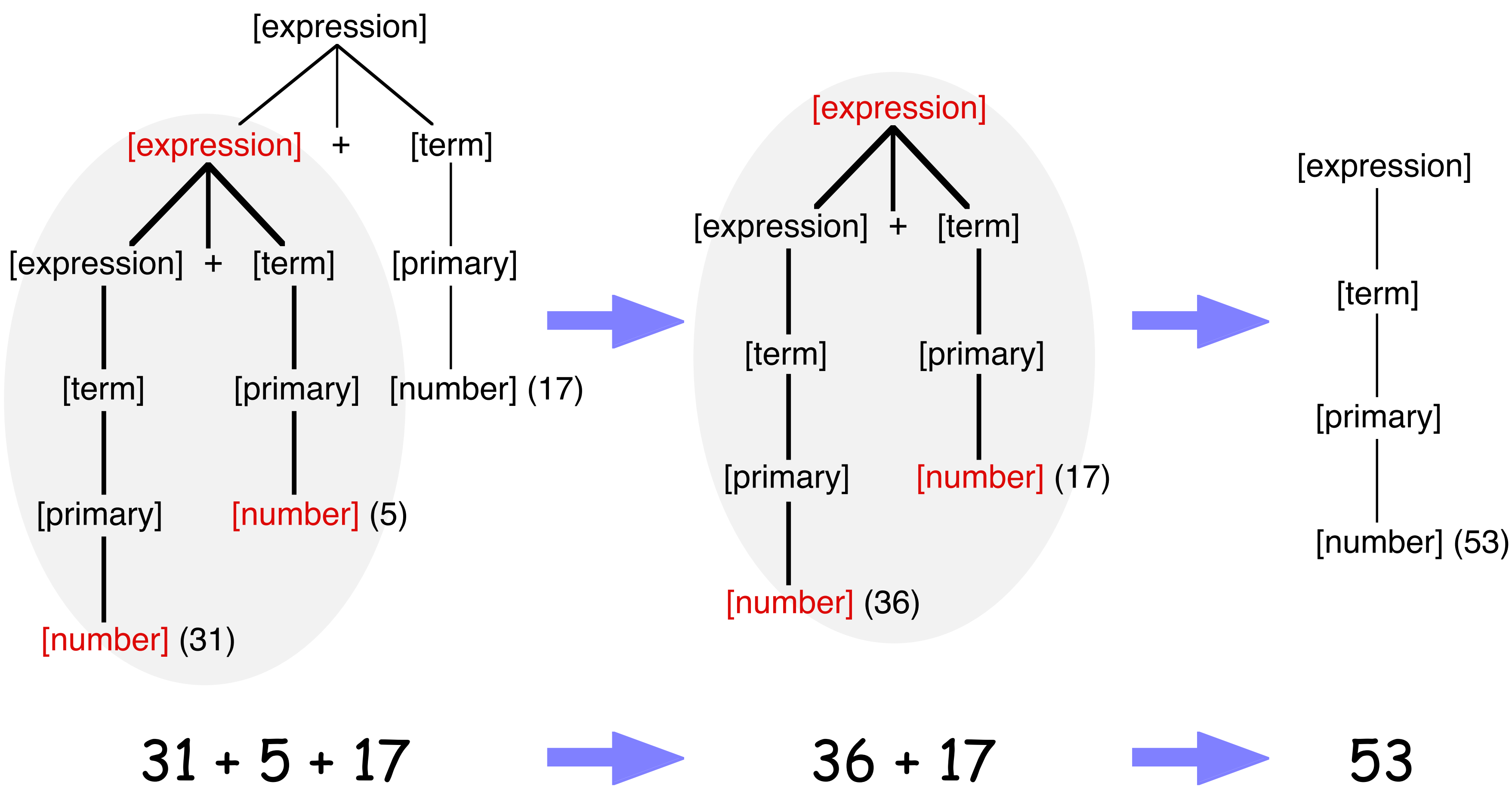
```
rule resolveAdditions
   replace [expression]
      N1[number] + N2[number]
   by
      N1 [+ N2]
end rule
```



Strongly typed patterns & replacements guarantee WF result

Contrast with ASF, others

TXL


[expression]
[expression]
+
[term]
[expression]
+
[term]
[primary]
[term]
[primary]
[number] (17)
[primary]
[number] (5)
[number] (31)
[expression]
[expression]
+
[term]
[term]
[primary]
[primary]
[number] (17)
[number] (36)
[expression]
[term]
[primary]
[number] (53)
31 + 5 + 17
36 + 17
53

TXL

```
include "TIL.grm"

% Begin-end dialect of TIL
redefine statement
    ...                   % refers to all existing forms
  | [begin_statement]  % add alternative for our new form
end redefine

define begin_statement
    begin
        [statement*]
    'end
end define
```

Grammar extensions and language variants using redefine

Agile Parsing

```
define declaration
        [init_declarator,+]
    |   [decl_specifiers?] [declarator] [ctor_initializer?]
        [compound_statement?] [‘; ?]
end define

define init_declarator
    [declarator] [initializer?]
end define

define declarator
        [repeat ptr_operator] [dname] [declarator_extension*]
    |   ( [declarator] ) [declarator_extension*]
end define
```

Agile Parsing

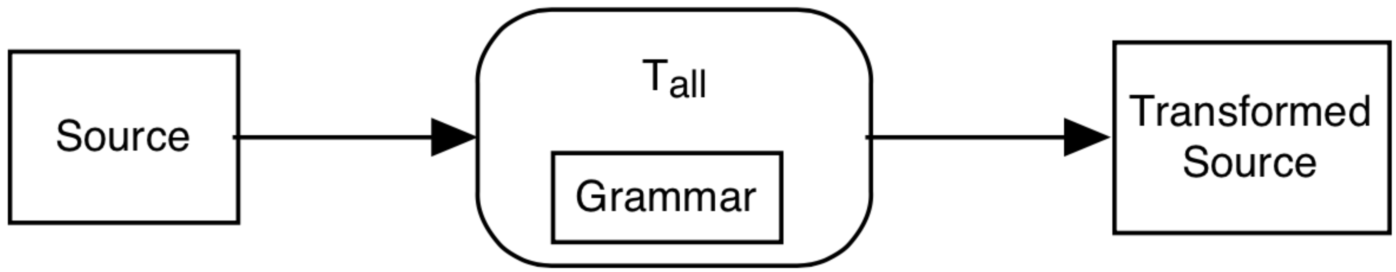
```
define declaration
        [init_declarator,+]
    |   [decl_specifiers?] [declarator] [ctor_initializer?]
        [compound_statement?] [‘; ?]
end define

define init_declarator
    [declarator] [initializer?]
end define

define declarator
        [repeat ptr_operator] [dname] [declarator_extension*]
    |   ( [declarator] ) [declarator_extension*]
end define
```

```
include "Cpp.Grammar"

redefine declaration
        [function_definition]
    |   ...
end redefine

define function_definition
    [function_header]
    [opt exception_specification]
    [function_body]
end define

define function_header
    [decl_specifiers?] [function_declarator] [ctor_initializer?]
end define
```

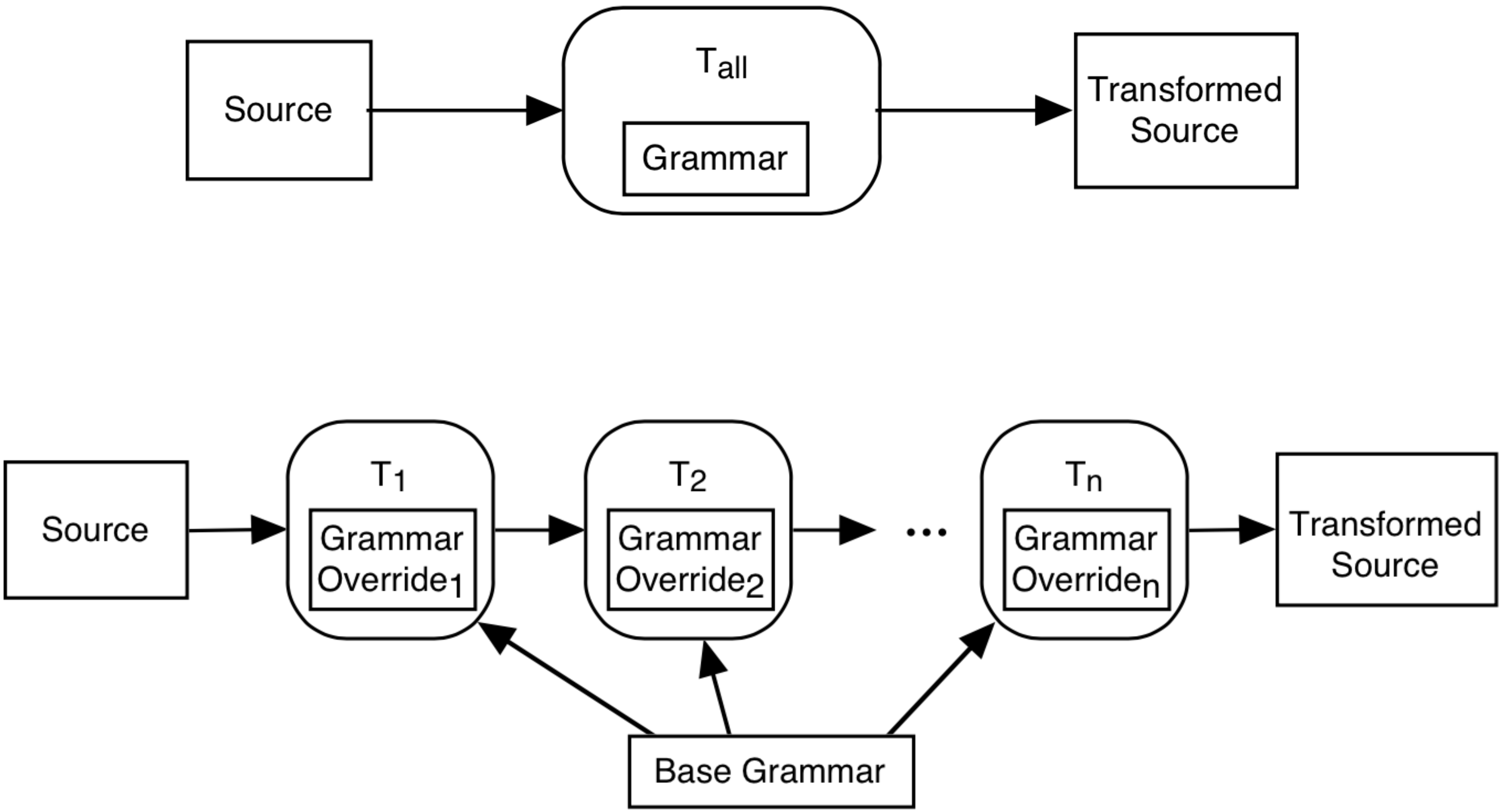
Custom parse of input, tailored to the task

T.R. Dean, J.R. Cordy, A.J. Malton and K.A. Schneider, "Agile Parsing in TXL", *Automated Software Engineering* 10:4 (2003)

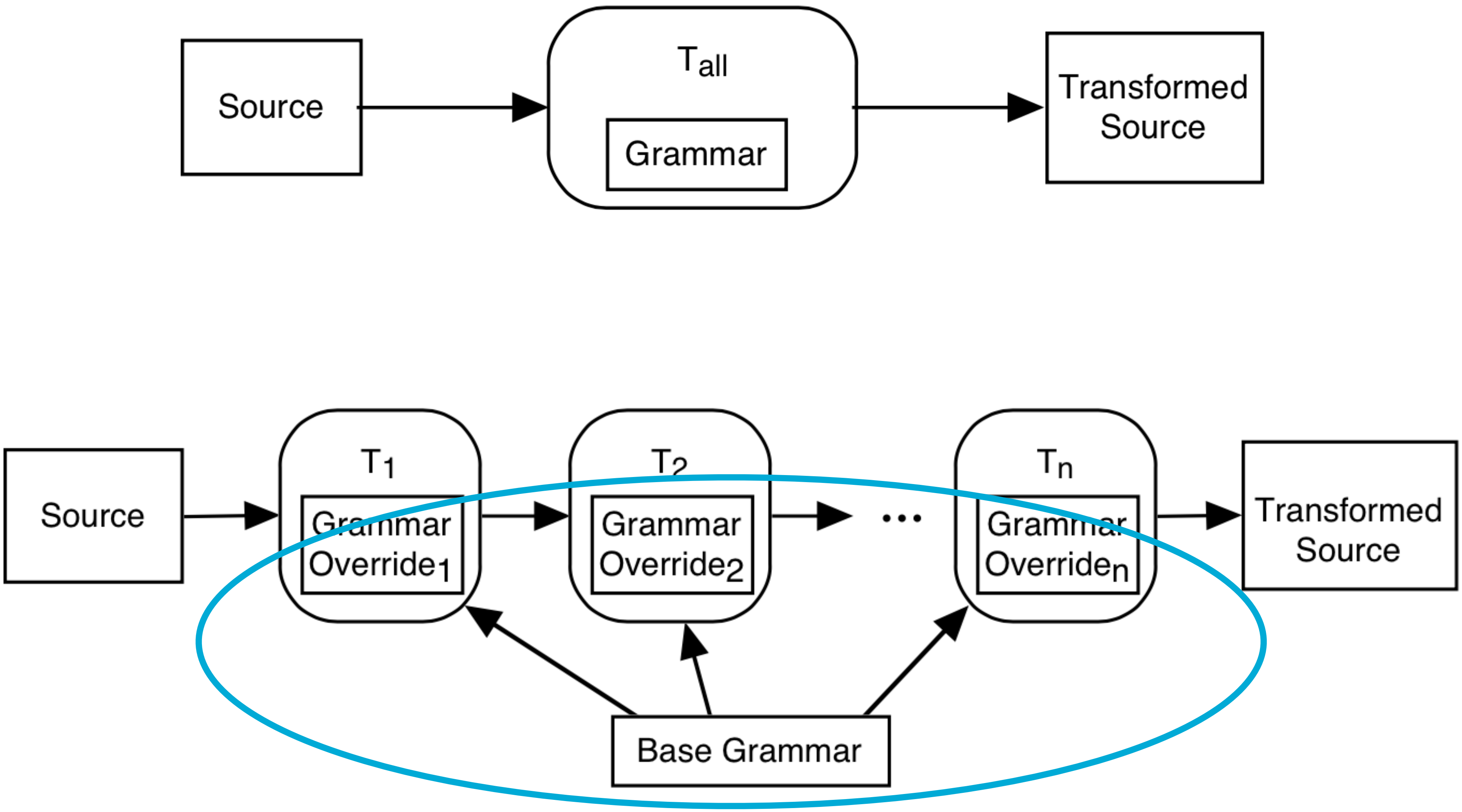
Agile Parsing



Agile Parsing



Agile Parsing



Boolean Grammar?

The Plan

Boolean grammars for TXL – extend power of TXL parser

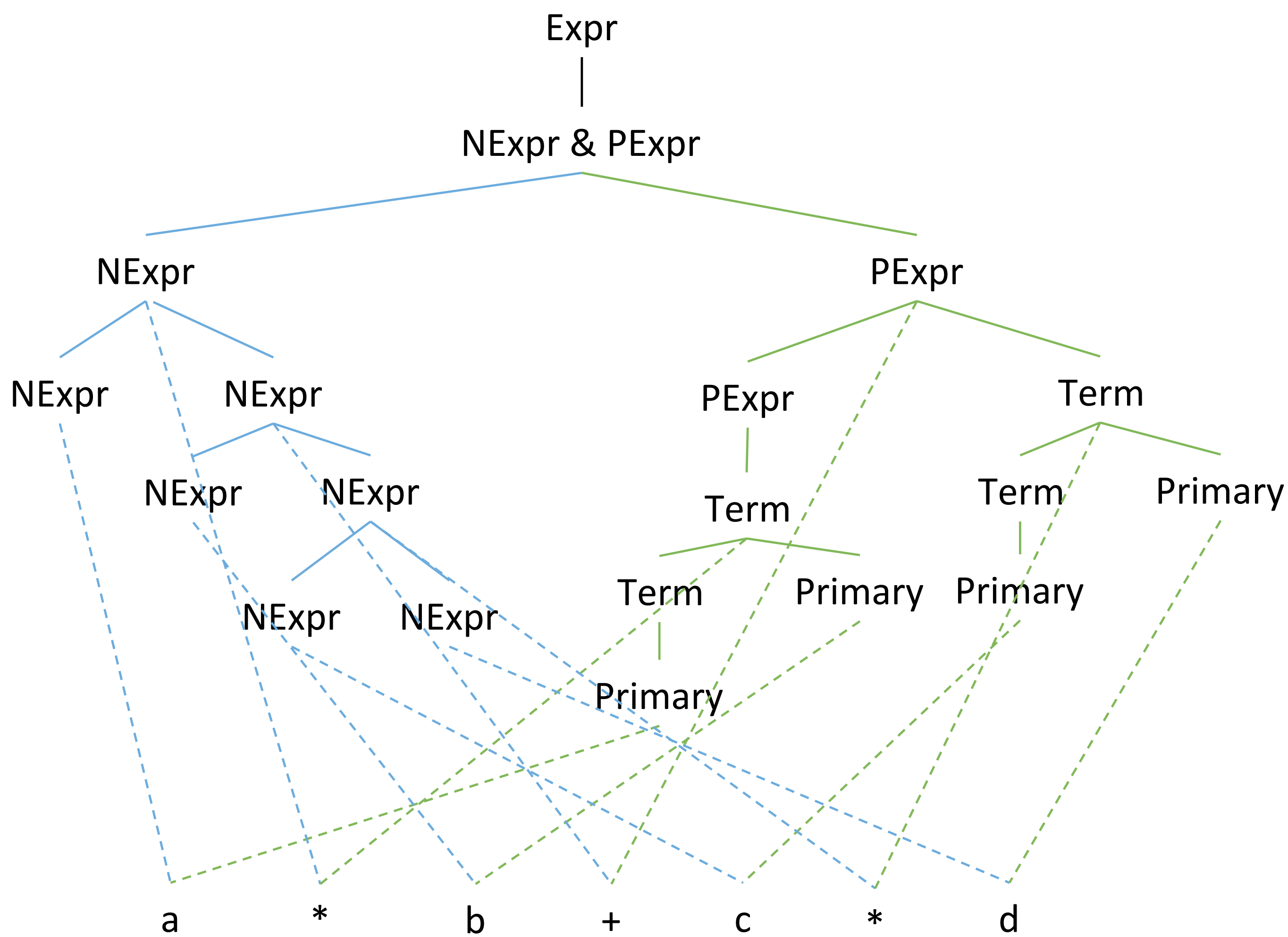
Multiple grammatical views in TXL – transformations on Boolean grammars

The Plan

Expr = NExpr & PExpr

NExpr = NExpr + Nectar
  | NExpr * NExpr
  | (NExpr)
  | Ident

PExpr = PExpr + Term
  | Term
Term = Term * Primary
  | Primary
Primary = (PExpr)
  | Ident



1. Boolean language recognition for TXL

$$ABC = AB\ c^* \ \&\ a^*\ BC$$

$$AB = a\ AB\ b \mid \epsilon$$

$$BC = b\ BC\ c \mid \epsilon$$

1. Boolean language recognition for TXL

$ABC = AB\ c^* \ \&\ a^*\ BC$

$AB = a\ AB\ b \mid \epsilon$

$BC = b\ BC\ c \mid \epsilon$

```
define ABC
    [AB] ['c*]
  & ['a*] [BC]
end define

define AB
    'a [AB] 'b
  | [empty]
end define

define BC
    'b [BC] 'c
  | [empty]
end define
```

1. Boolean language recognition for TXL

$$ABC = AB\ c^* \ \&\ a^*\ BC$$

$$AB = a\ AB\ b \mid \epsilon$$

$$BC = b\ BC\ c \mid \epsilon$$

```
define ABC
    [AB] ['c*]
  & ['a*] [BC]
end define

define AB
    'a [AB] 'b
  | [empty]
end define

define BC
    'b [BC] 'c
  | [empty]
end define
```

a a a b b b c c c

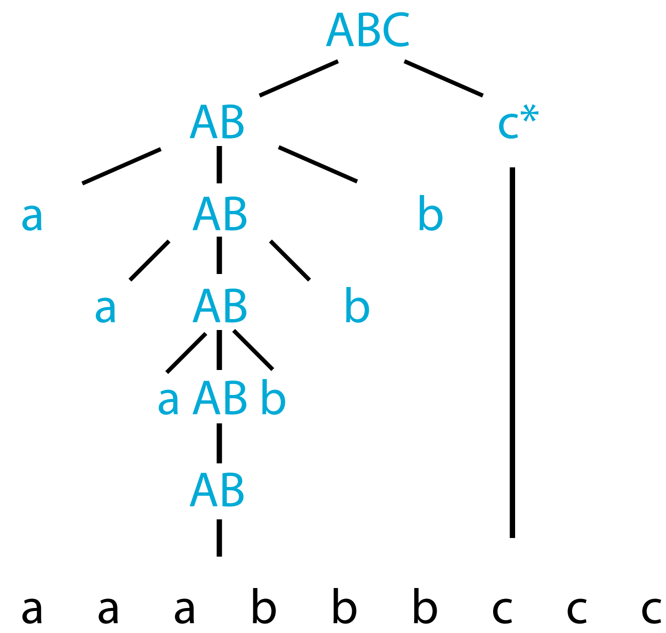
1. Boolean language recognition for TXL

ABC = AB c* & a* BC

AB = a AB b | ϵ

BC = b BC c | ϵ

```
define ABC
    [AB] [‘c*]
  & [‘a*] [BC]
end define

define AB
    ‘a [AB] ‘b
  | [empty]
end define

define BC
    ‘b [BC] ‘c
  | [empty]
end define
```



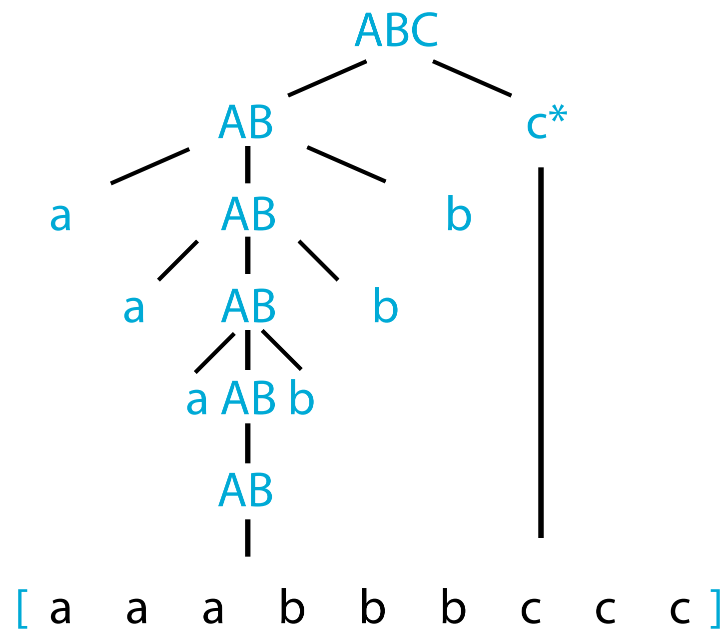
1. Boolean language recognition for TXL

ABC = AB c* & a* BC

AB = a AB b | ϵ

BC = b BC c | ϵ

```
define ABC
    [AB] [‘c*]
  & [‘a*] [BC]
end define

define AB
    ‘a [AB] ‘b
  | [empty]
end define

define BC
    ‘b [BC] ‘c
  | [empty]
end define
```



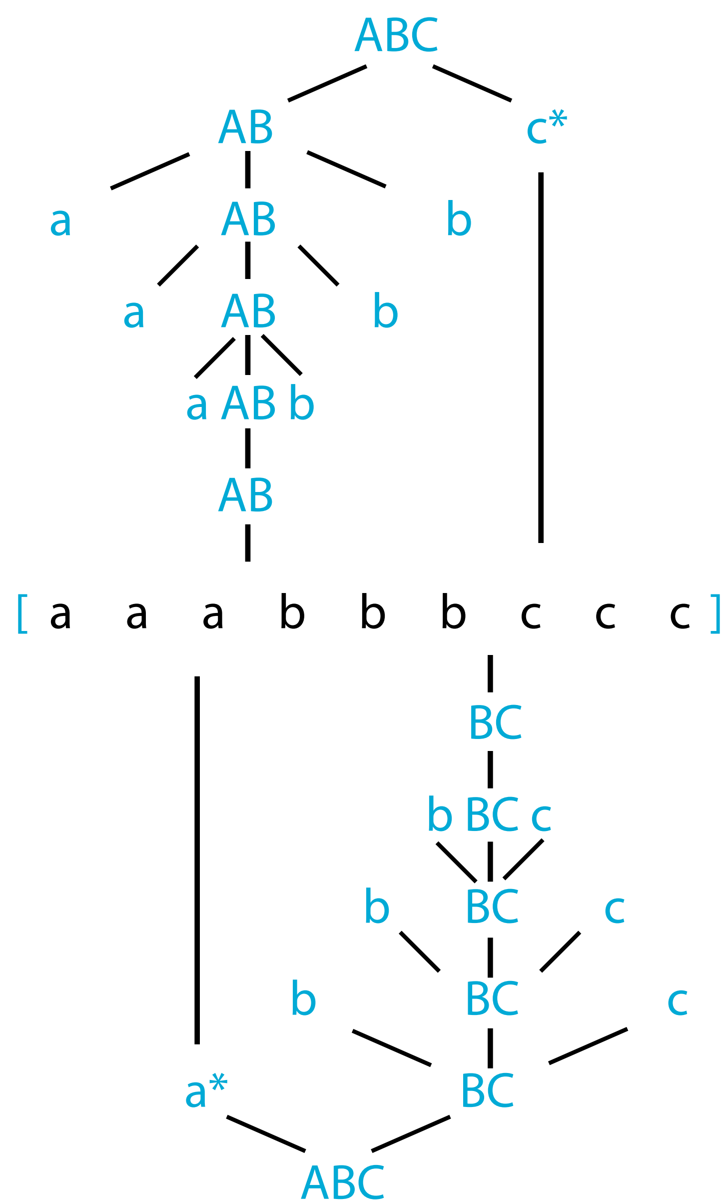
1. Boolean language recognition for TXL

ABC = AB c* & a* BC

AB = a AB b | ϵ

BC = b BC c | ϵ

```
define ABC
    [AB] [‘c*]
  & [‘a*] [BC]
end define

define AB
    ‘a [AB] ‘b
  | [empty]
end define

define BC
    ‘b [BC] ‘c
  | [empty]
end define
```



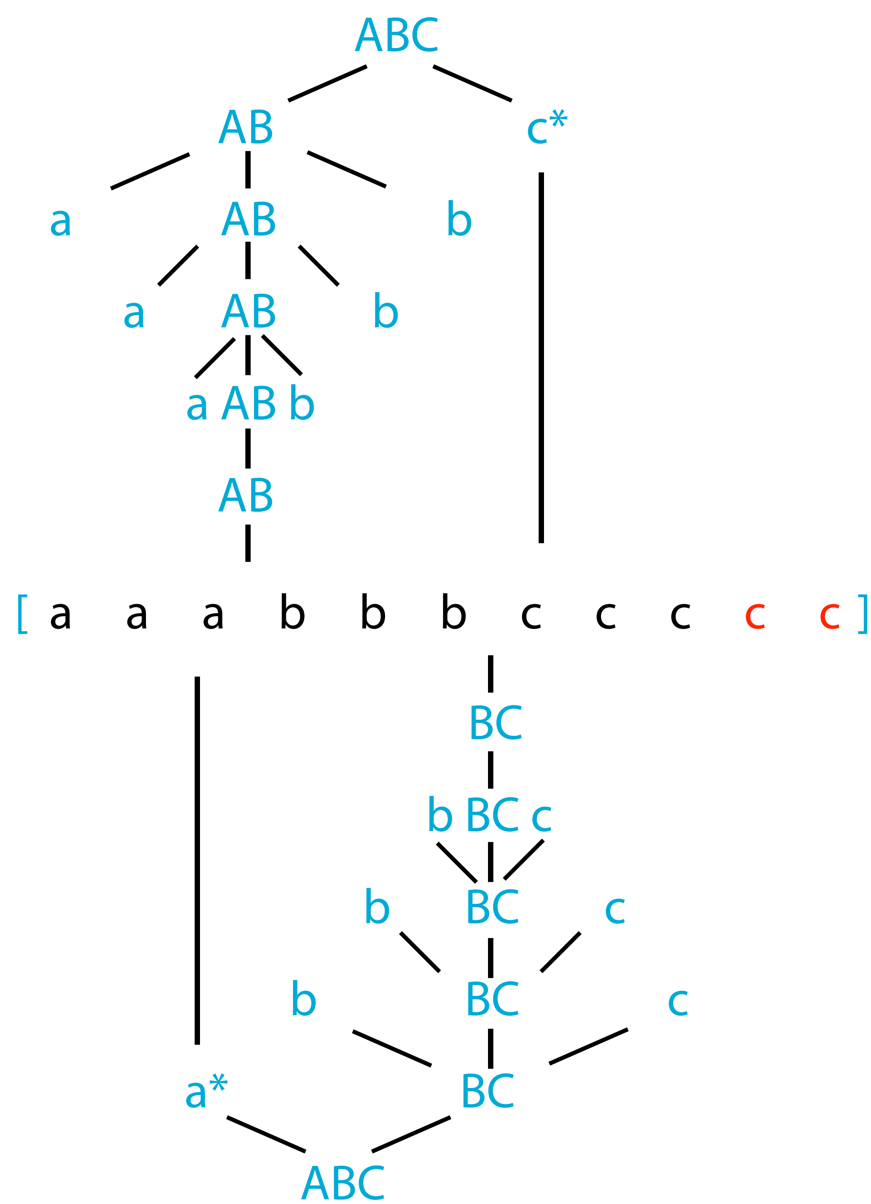
1. Boolean language recognition for TXL

ABC = AB c* & a* BC

AB = a AB b | ϵ

BC = b BC c | ϵ

```
define ABC
    [AB] ['c*]
  & ['a*] [BC]
end define

define AB
    'a [AB] 'b
  | [empty]
end define

define BC
    'b [BC] 'c
  | [empty]
end define
```



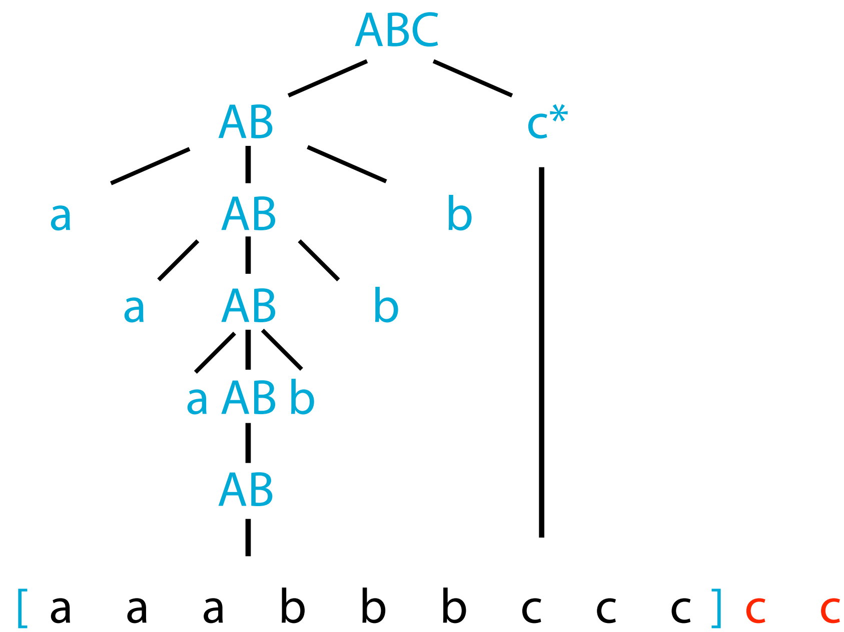
1. Boolean language recognition for TXL

$$ABC = AB\ c^* \ \&\ a^*\ BC$$

$$AB = a\ AB\ b \mid \epsilon$$

$$BC = b\ BC\ c \mid \epsilon$$

```
define ABC
    [AB] [‘c*]
  & [‘a*] [BC]
end define

define AB
    ‘a [AB] ‘b
  | [empty]
end define

define BC
    ‘b [BC] ‘c
  | [empty]
end define
```



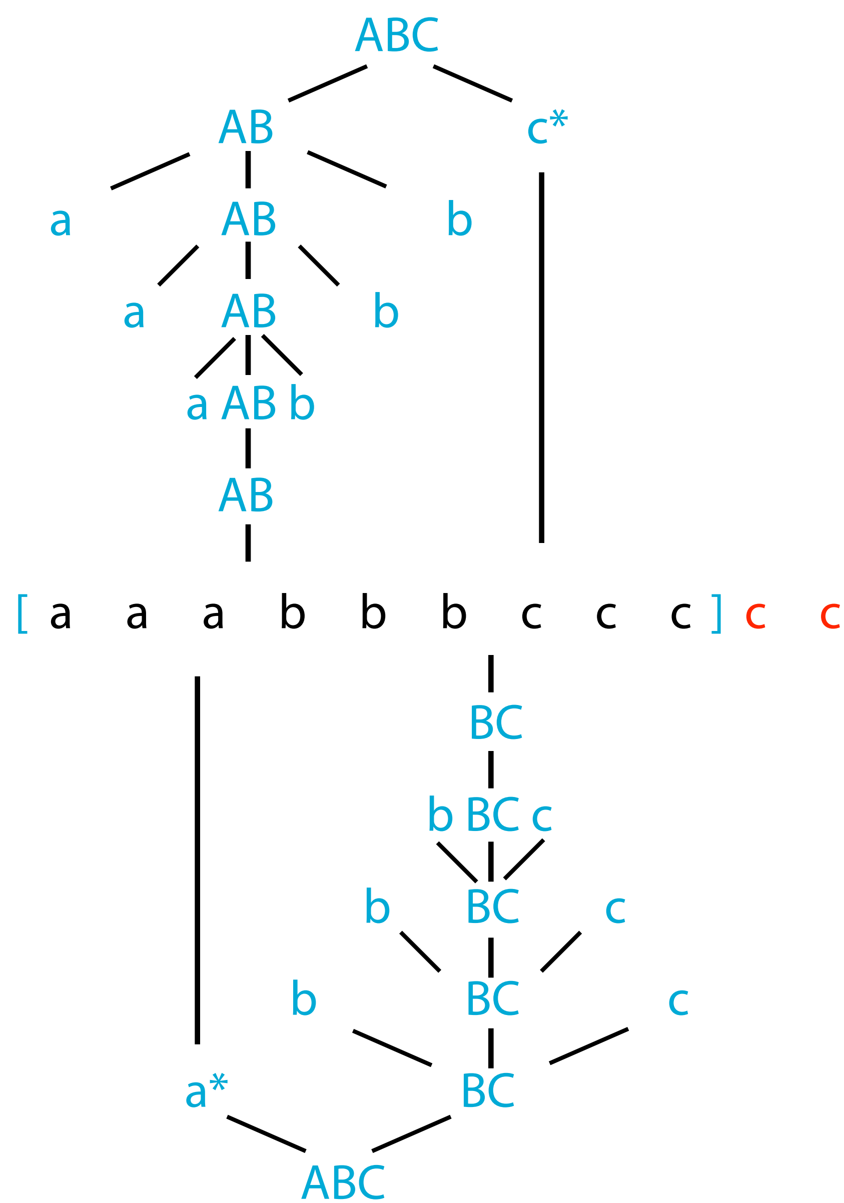
1. Boolean language recognition for TXL

$$ABC = AB\ c^* \ \&\ a^*\ BC$$

$$AB = a\ AB\ b \mid \epsilon$$

$$BC = b\ BC\ c \mid \epsilon$$

```
define ABC
    [AB] ['c*]
  & ['a*] [BC]
end define

define AB
    'a [AB] 'b
  | [empty]
end define

define BC
    'b [BC] 'c
  | [empty]
end define
```



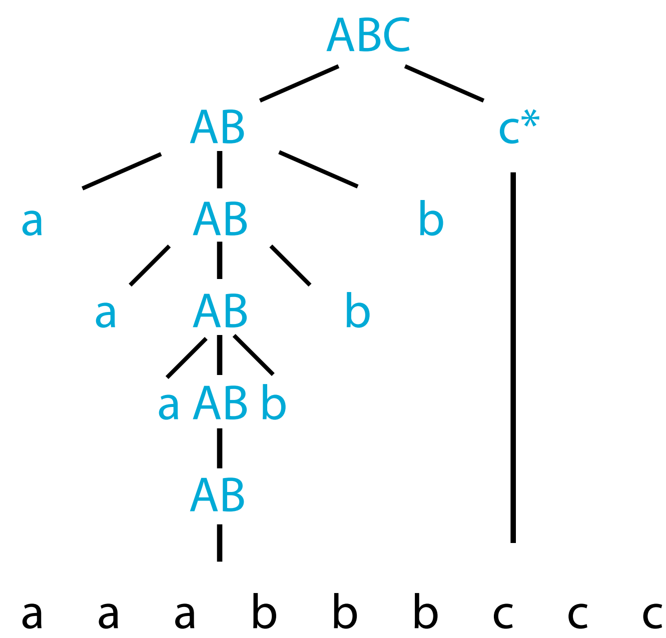
1. Boolean language recognition for TXL

ABC = AB c* & a* BC

AB = a AB b | ϵ

BC = b BC c | ϵ

```
define ABC
    [AB] ['c*]
  & ['a*] [BC]
end define

define AB
    'a [AB] 'b
  | [empty]
end define

define BC
    'b [BC] 'c
  | [empty]
end define
```



Summary: no problem.

& -> identify range of first parse, use regular parsing, backtracking, yield left hand tree, all is well

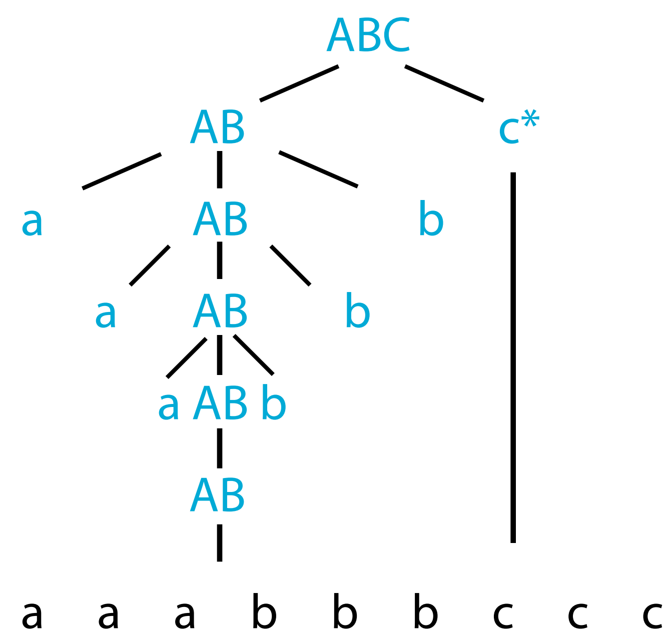
&! -> yields same original tree

Transformation can't tell anything has changed!

2. Boolean parsing for TXL, DAG result

$$ABC = AB\ c^* \ \&\ a^*\ BC$$
$$AB = a\ AB\ b \mid \epsilon$$
$$BC = b\ BC\ c \mid \epsilon$$

```
define ABC
    [AB] ['c*]
  & ['a*] [BC]
end define

define AB
    'a [AB] 'b
  | [empty]
end define

define BC
    'b [BC] 'c
  | [empty]
end define
```



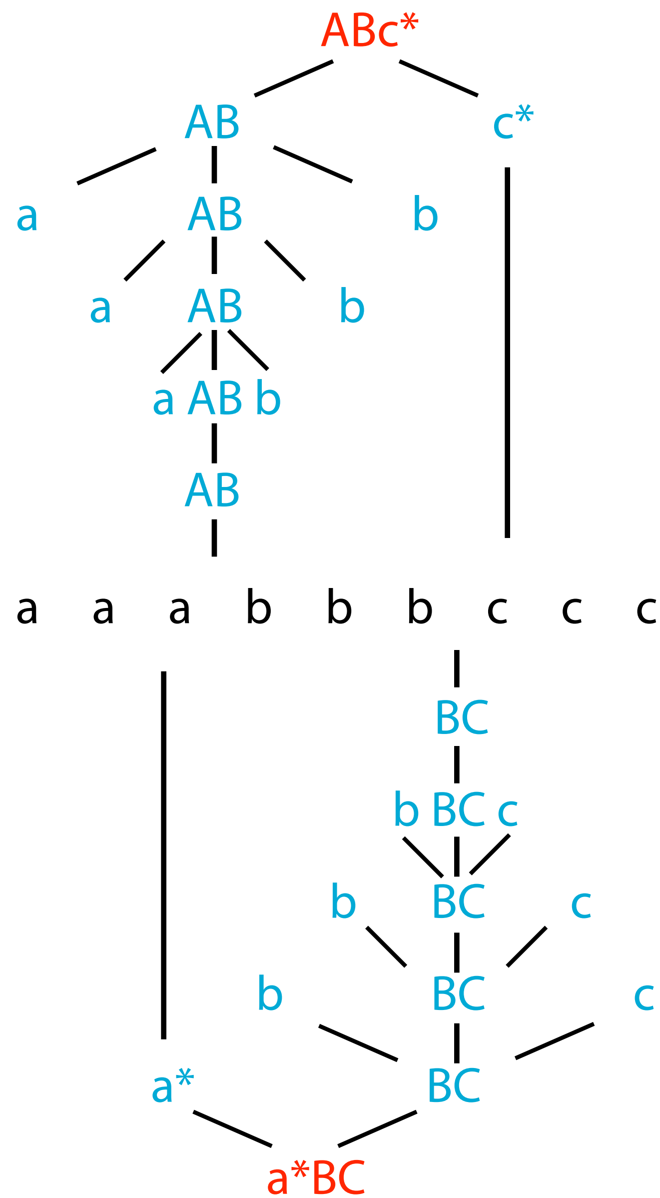
2. Boolean parsing for TXL, DAG result

ABC = AB c* & a* BC

AB = a AB b | ϵ

BC = b BC c | ϵ

```
define ABC
    [AB] ['c*]
  & ['a*] [BC]
end define

define AB
    'a [AB] 'b
  | [empty]
end define

define BC
    'b [BC] 'c
  | [empty]
end define
```



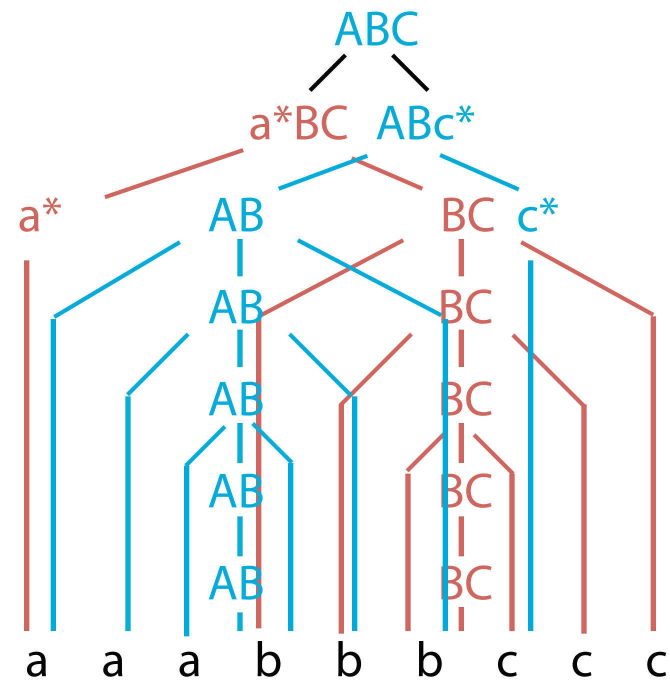
2. Boolean parsing for TXL, DAG result

ABC = AB c* & a* BC

AB = a AB b | ϵ

BC = b BC c | ϵ

```
define ABC
    [AB] [‘c*]
  & [‘a*] [BC]
end define

define AB
    ‘a [AB] ‘b
  | [empty]
end define

define BC
    ‘b [BC] ‘c
  | [empty]
end define
```



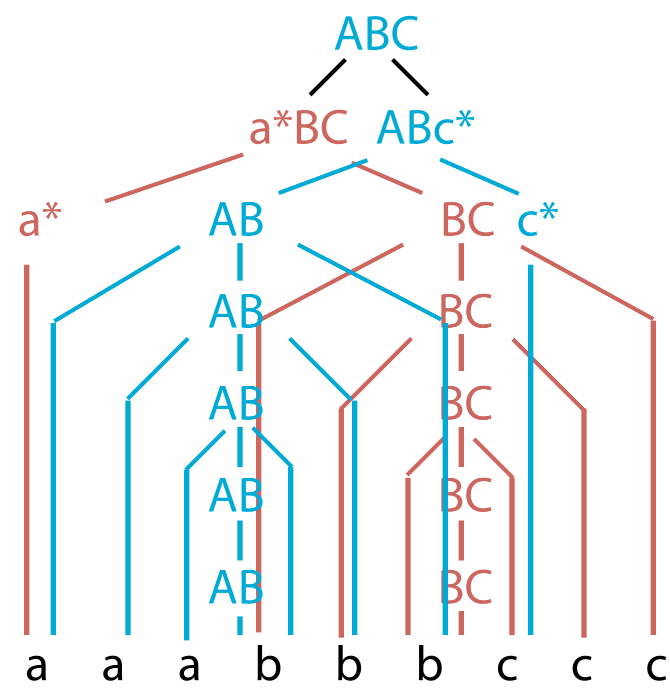
2. Boolean parsing for TXL, DAG result

ABC = AB c* & a* BC

AB = a AB b | ϵ

BC = b BC c | ϵ

```
define ABC
    [AB] [‘c*]
  & [‘a*] [BC]
end define

define AB
    ‘a [AB] ‘b
  | [empty]
end define

define BC
    ‘b [BC] ‘c
  | [empty]
end define
```



ABC = … | a* b* c*

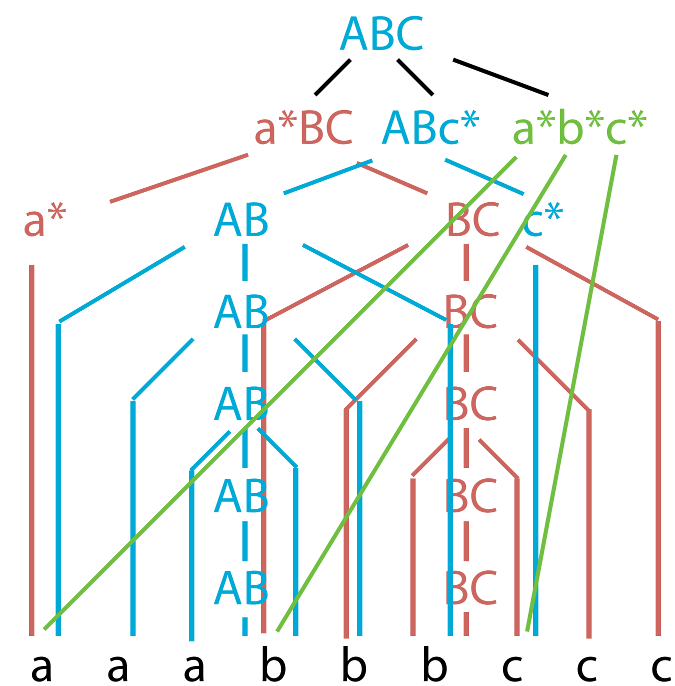
```
redefine ABC
    ...
  & [‘a*] [‘b*] [‘c*]
end redefine
```

2. Boolean parsing for TXL, DAG result

ABC = AB c* & a* BC

AB = a AB b | ϵ

BC = b BC c | ϵ




```
define ABC
    [AB] ['c*]
  & ['a*] [BC]
end define

define AB
    'a [AB] 'b
  | [empty]
end define

define BC
    'b [BC] 'c
  | [empty]
end define
```

ABC = ... | a* b* c*

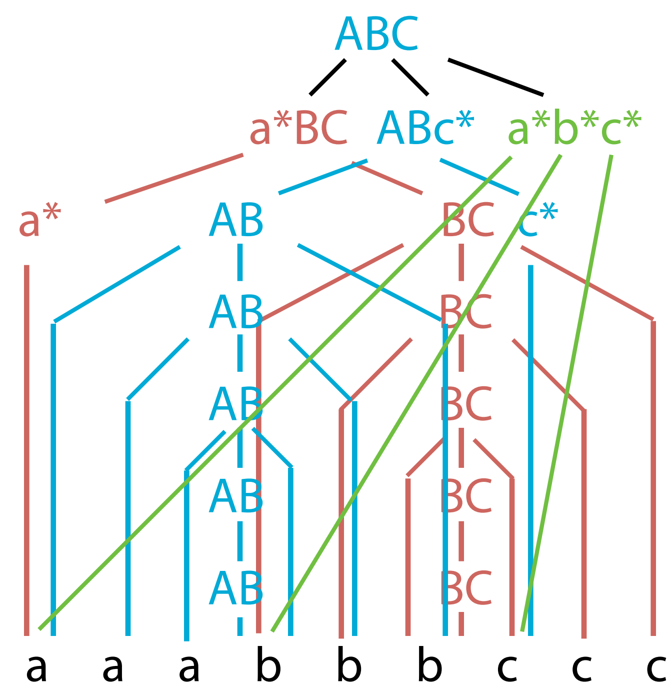
```
redefine ABC
    ...
  & ['a*] ['b*] ['c*]
end redefine
```

2. Boolean parsing for TXL, DAG result

ABC = AB c* & a* BC

AB = a AB b | ϵ

BC = b BC c | ϵ




```
define ABC
    [AB] ['c*]
  & ['a*] [BC]
end define

define AB
    'a [AB] 'b
  | [empty]
end define

define BC
    'b [BC] 'c
  | [empty]
end define
```

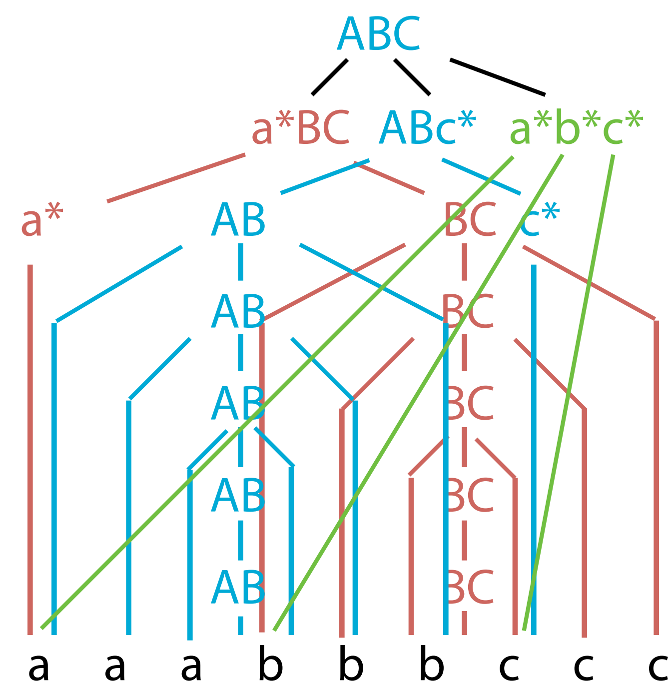
Again: no problem.

& -> identify range of first parse,
use regular parsing, backtracking,
yield all view trees, all is well

&! -> still yields same original tree

But: what does it mean for transformation rules?

3. Transforming with Boolean views



Let us assume that the parser has generated the appropriate DAG for the Boolean parse

ABC = AB c* & a* BC & A B C

A = a*

B = b*

C = c*

Our goal is to allow transformation of views - but what does that mean for the DAG?

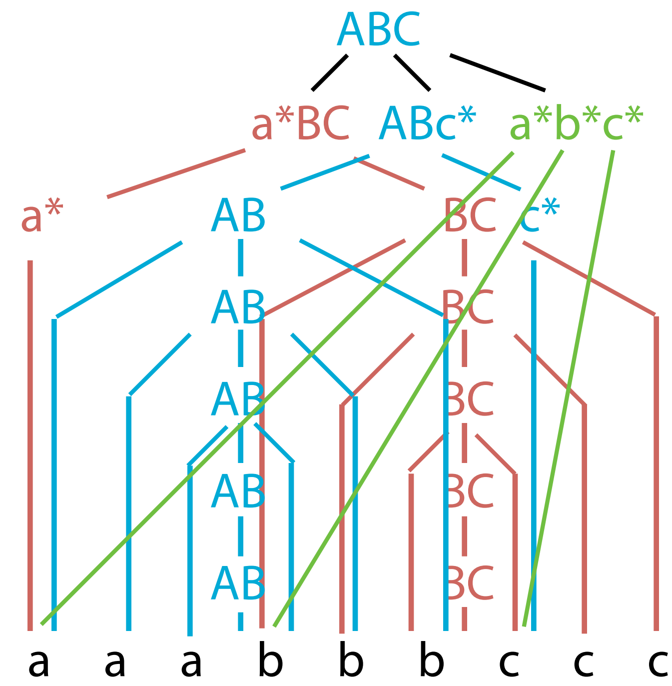
ABC : aaA bbB ccC => A B C

This rule targets ABC and uses the A B C view.

Recall that TXL rules are strongly typed in the target type of the rule.

Using TXL semantics, the rules replaces any match with either abc or ϵ – either way, all parses are still valid and the result is still a WF ABC

3. Transforming with Boolean views



ABC = AB c* & a* BC & A B C

A = a*

B = b*

C = c*

Suppose now we have an equivalent rule set.

R1. A : aaA => A

R2. B : bbB => B

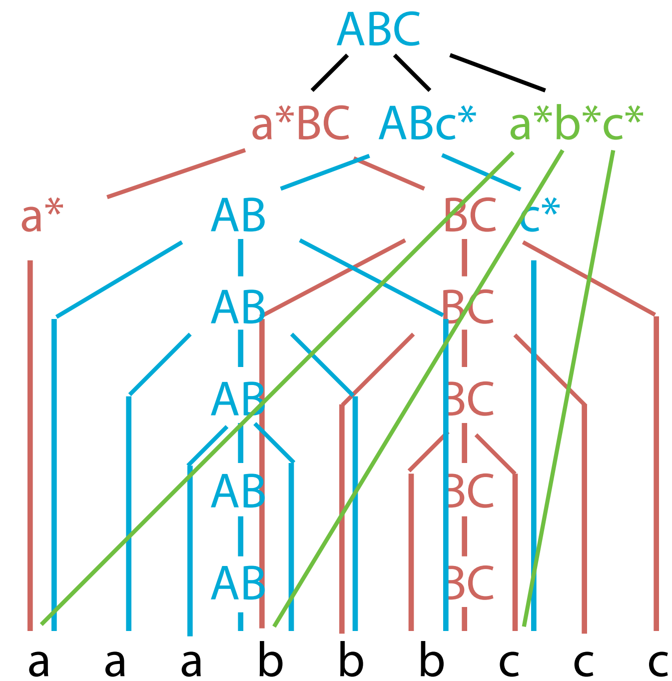
R3. C : ccC => C

ABC : A B C => A[R1] B[R2] C[R3]

This rule still targets ABC and uses the A B C view, and the result is still a WF ABC.

But …

3. Transforming with Boolean views



R1. A : aaA => A
R2. B : bbB => B
R3. C : ccC => C

Suppose instead we apply the subrules to the whole thing:

ABC : ABC => ABC [R1] [R2] [R3]

According to TXL, the result should be the same (and will be) - but we have a problem!

The intermediate results ABC [R1] and ABC [R1] [R2], are not WF ABCs.

a a a b b b c c c
a b b b c c c
a b c c c
a b c

3. Transforming with Boolean views

This becomes clearer when we consider the TXL semantics:

ABC : ABC => ((ABC [R1]) [R2]) [R3]

This is a new problem - a rule targeting a child can invalidate the parse of a parent (or any ancestor) which is not itself transformed!

We can also write rules that use views to always invalidate a result, as demonstrated by this rewriting of R1 and R3:

R1. A : A => a

R3. C : C => ϵ

a a a b b b c c c

a b b b c c c

a b c c c

a b

3. Transforming with Boolean views

So what is the meaning of transforming a view?

We could say that the solution is, when we transform a conjunctive type using a view, we must reparse the result to assure it still is WF

But what if not? We may throw away tons of work, possibly which will become WF again, after further rules are applied

It's also confusing: if the reparse fails,

- is that an error (and we abandon the entire transformation)?
- or is it simply a failed rule (normal in TXL since all rules are total, but confusing in this case)?
- in the worst case, we may successfully transform using many views, only to rescind all the changes at the root of the parse

3. Transforming with Boolean views

So what is the meaning of transforming a view?

It's also important to realize that, even if no rules target a conjunctive type or view directly, every conjunctive type must be reparsed whenever anything inside it may be transformed, because a rule may search through a conjunctive type

P = STMT*
STMT = ABC ;
ABC = AB c* & a* BC & A B C
A = a*
B = b*
C = c*

R1. A : A => a

R. P : P => P [R1] no mention of a conjunct or view at all!

3. Transforming with Boolean views

It gets worse …

OK, so let's say that we always reparse transformed conjunctive types,
if any rules have targeted anything inside them

What if two different rules transform inside two different views,
each transforming one of them?

Which of the DAG branches is the new source to be reparsed?

And what about co-transformations, where the result isn't valid until both
sub-transformations are complete?

3. Transforming with Boolean views

Alternative semantics

So basically TXL semantics are problematic with Boolean grammars –
what else might we do?

1) Abandon WF guarantees (like ASF): solves some problems,
   but still leaves us with the two-views transformed issue (also un-TXL-ish)

2) Allow conjuncts to turn into disjuncts on transformation (weak WF):
   works well when only one view is needed per program
   (but we could already do that!)

3) Require that patterns include all views (Boolean patterns):
   describe how all views change together, but only works for direct
   conjunctive patterns, and defeats purpose of views

4) Full reparse: analyze rule set to minimize, fail rule when reparse fails,
   rely on user to extend grammar to allow required intermediates

5) Give up.

For now, we are working with (4)

An Example

```
% basic precedence grammar - C has 18 precedence levels!
Expr = Expr BoolOp Comp
     | Comp
Comp = Comp CompOp Sum
     | Sum
Sum =  Sum AddOp Term
     | Term
Term = Term MulOp Factor
     | Factor
Factor = UnOp? Factor
     | Primary
Primary = Literal | Id | ( Expr )

BoolOp = and | or
CompOp = < | > | <= | >=
AddOp = + | -
MulOp = * | /
UnOp = -

% uniform expression view - every subexpression can be viewed
% as an expression
Comp =    ... & Expr
Sum =     ... & Expr
Term =    ... & Expr
Factor =  ... & Expr

% fully parenthesize all subexpressions using uniform expression view
rule Parenthesize.  Expr: Expr !Primary => ( Expr )
```

An Example

```
% uniform operator view - every kind of operator can be viewed as
% simple a binary operator
Op = BoolOp | CompOp | AddOp | MulOp
BoolOp =  ... & Op
CompOp =  ... & Op
AddOp =   ... & Op
MulOp =   ... & Op

% type attribution override
Primary =   ... TypeAttr?
TypeAttr =  :: Type

% type inference rule using fully parenthesized uniform expression
% and uniform operator views (initial leaf typing not shown here)
rule InferComps.
   Expr: ( Expr1 :: Type CompOp Expr2 :: Type ) =>
         ( Expr1 :: Type CompOp Expr2 :: Type } :: boolean

rule InferConsistent.
   Expr: ( Expr1 :: Type Op Expr2 :: Type ) =>
         ( Expr1 :: Type Op Expr2 :: Type } :: Type

% flag type errors
rule FlagRest.
   Expr: ( Expr ) => ( Expr } :: ERROR
```

Current work, and what's next

Current state

Still working on implementation and understanding opportunities for optimization – done steps 1 and 2 (parsing, DAGs), direct case of 3 (rules)

New implementation completely compatible with old – if you don't use & and &!, nothing changes

Still not certain all issues have been uncovered – building comprehensive functional test set

And as always …

We won't know what Boolean transformation should mean, or what it is for, until the TXL users start exploring it

Questions, comments, advice – more problems?