



The TXL Programming Language

Filippo Ricca & Mariano Ceccato

ITC-Irst

Istituto per la ricerca Scientifica e
Tecnologica

ricca@itc.it ceccato@itc.it

What is TXL?

- ◆ TXL is a programming language specifically designed to support **software analysis** and **program transformation**.
- ◆ The TXL programming language is a **functional rule-based language**.

example: functional rule-based program

functions: $F(x) = x-1;$

$G(x) = x+1;$

rules: $\text{If } x < 5 \text{ then } x := G(x);$

$\text{If } x > 5 \text{ then } x := F(x);$

$\text{If } x = 5 \text{ then stop;}$

Flow execution is not sequential!

Program Transformations

- ◆ Program transformation is the act of changing one program into another.



source language L

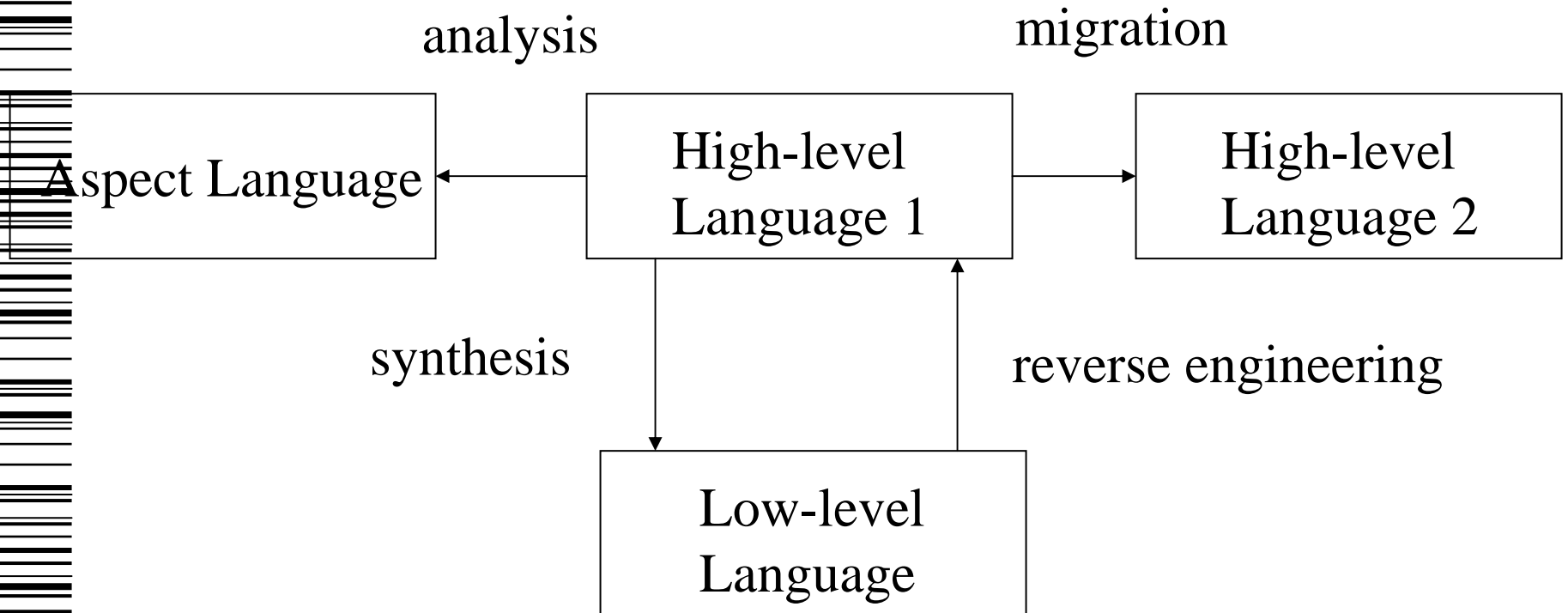
target language L'

L is different to L' -----> translation

L is equal to L' -----> rephrasing

What is TXL good for (1)?

◆ Translation





Translations



- ◆ Program synthesis: compilation, code generation, ...
- ◆ Program migration: porting a Pascal program to C, translation between dialects (Fortran77 to Fortran90), ...
- ◆ Reverse engineering: architecture extraction, design recovery, ...
- ◆ Program analysis: measurement, clone detection, type inference, call graph, control flow graph, ...

Reverse engineering

Example: design recovery

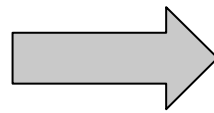
Java code

Class A;

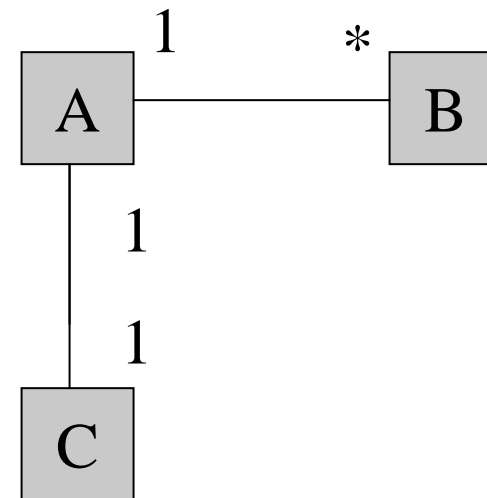
Class B;

Class C,

Reverse engineering



Class Diagram



Program analysis

◆ Example: clone analysis

Clones:

...

Lines 20-50 and 100-130;

...

...

```
20: FOR I=1 TO 10
```

```
30: V[I] = V[I] +1;
```

```
40: PRINT V[I]
```

```
50: ENDFOR
```

```
60: PRINT X;
```

```
70: CALL F;
```

...

```
100: FOR J=1 TO 10
```

```
110: W[J] = W[J] +1;
```

```
120: PRINT W[J]
```

```
130: ENDFOR
```

...

What is TXL good for (2)

Rephrasing:

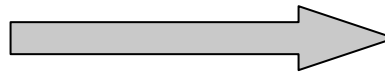
- Normalization: reduces a program to a program in a sub-language to decrease its syntactic complexity (ex. Pascal to “core Pascal”).
- Optimization: improves the run-time and/or space performance of a program (ex. Code motion optimization).
- Refactoring or restructuring: changes the structure of the program to make it easier to understand.
- Renovation: Error repair (ex. Year 2000) and changed requirements (ex. “lira” to “euro”). **Does not preserve semantics.**

Optimization

Example: Code motion optimization: moves all loop-independent assignment statements outside of loops.

```
Loop
  x := a + b;
  y := x;
  a := y + 3;
  x := b + c;
  y := x - 2;
  z := x + a * y;
End loop
```

Code motion optimization



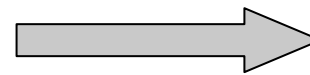
```
x4 := b + c;
y2 := x4 - 2;
Loop
  x := a + b;
  y := x;
  a := y + 3;
  z := x4 + a * y2;
End loop
```

Restructuring

Example: goto elimination

```
f := 0;
A_0: if x > n goto B_3;
      x := x - 1;
      f := f * x;
      goto A_0;
B_3: print f
```

goto elimination



```
f := 0;
while x <= n do
  x := x - 1;
  f := f * x;
end while
print f
```



TXL Components

Each TXL program has two components:

- ◆ A description of the structure to be transformed specified as an **EBNF grammar**, in context-free ambiguous form.
- ◆ A set of Transformation Rules specified by example, using pattern/replacement pairs.

Syntax definition

- ◆ A grammar G describes the syntax of a language.
- ◆ $L(G)$ is the language defined by the grammar G .
- ◆ Usually a grammar G is given in (E)BNF notation.

Example:

$E \rightarrow E + E \mid E * E \mid 0 \mid 1 \mid 2$

non-terminal terminal

$0+1*2$ is in $L(E)$
 $3+0$ is not in $L(E)$



BNF vs. EBNF

BNF

List --> List Element ;

List --> Element ;

Element --> number

Element --> word

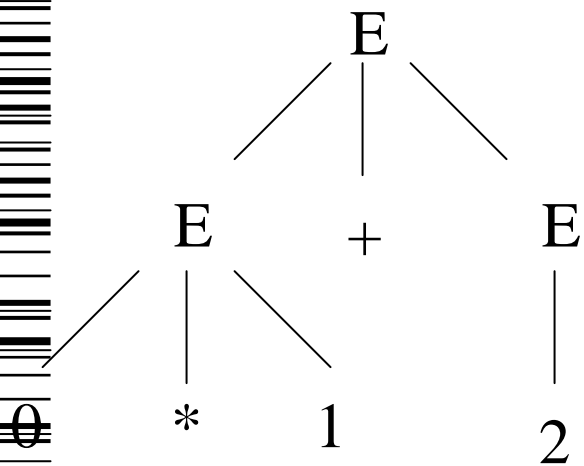
Element --> word sign word

EBNF

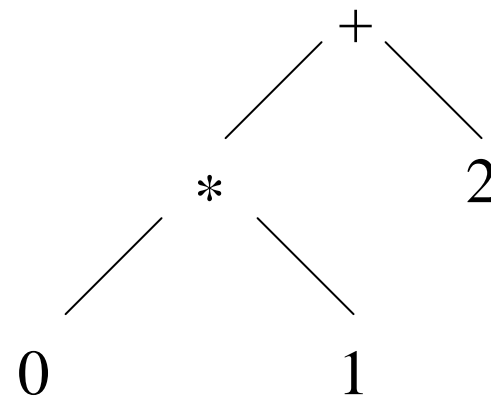
List --> ((word [sign word] | number) ;)*

Parse Tree vs. AST

Parse tree



Abstract syntax tree



$E \rightarrow E + E \mid E * E \mid 0 \mid 1 \mid 2$

Ambiguity

- ◆ A grammar that produces more than one parse tree for some term is said to be ambiguous.

Example:

$E \rightarrow E + E \mid E * E \mid 0 \mid 1 \mid 2$ is ambiguous.

$0+1*2$ has two parse trees.

Transformation rules

- A transformation rule is a rule of the form:

Lhs \rightarrow RhS if cond

where Lhs and RhS are term patterns

the condition is optional

The application of a rule to a term succeeds if the term matches (pattern matching) with the Lhs pattern and the condition is true.

The result is the instantiation of the RhS pattern.

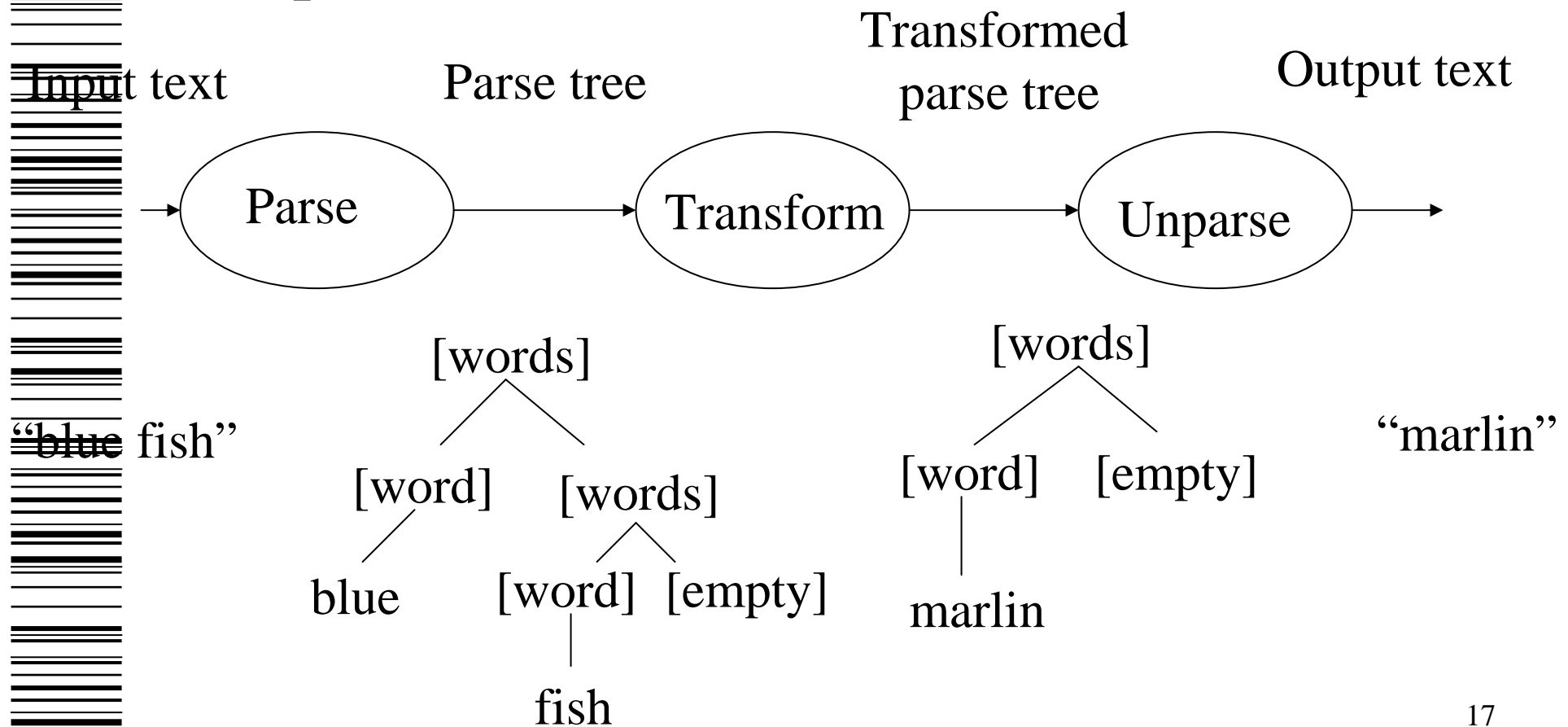
For example, if we have the term: $3 + 0$

and applying the rule: $x + 0 \rightarrow x$ to it

the result is 3 (the pattern variable x matches the number 3)

The three phases of TXL

txl “input file” “txl file”



First example: 'expr' grammar

```
% BNF: Expr --> Num | Expr+Expr | Expr*Expr | (Expr)
```

```
% Part I. Syntax specification
```

```
define program
```

```
  [Expr]
```

```
end define
```

```
define Expr
```

```
  [number]
```

```
  [Expr] '+' [Expr]
```

```
  [Expr] '*' [Expr]
```

```
  '(' [Expr] ')'
```

```
end define
```

First example: rules

$N + 0 \text{ -----} \rightarrow N$

```
rule removePlusZero
  replace [Expr]
    N [number] '+' 0
  by
    N
end rule
```

$(N) \text{ -----} \rightarrow N$

```
rule resolveBracketedExpressions
  replace [Expr]
    '( N [number] '
  by
    N
end rule
```

First example: main rule

% Part 2: main rule

rule main

 replace [Expr]

 E [Expr]

 construct NewE [Expr]

 E [removePlusZero]

 [resolveBracketedExpr]

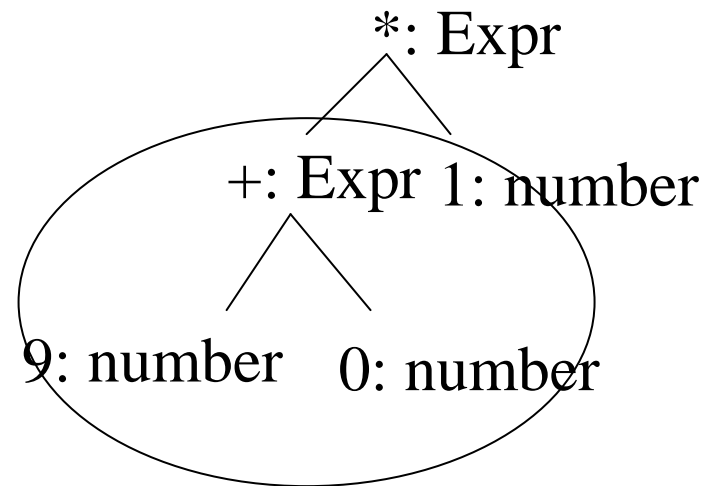
 where not

 NewE [= E]

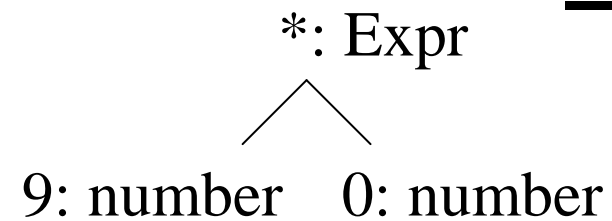
 by

 NewE

end rule

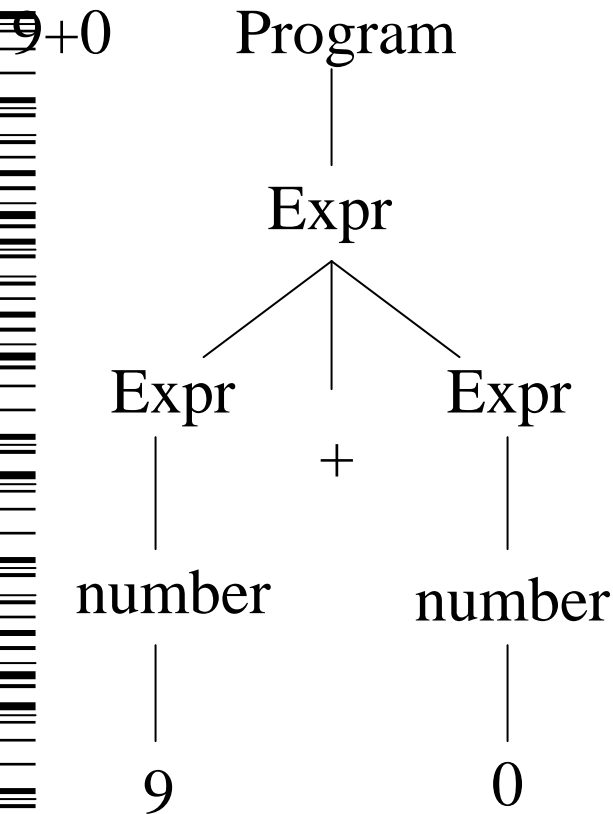


E



NewE

First example: parsing



----- Input Parse Tree -----

```
<program>
```

```
<Expr>
```

```
<Expr> <number text="9" </Expr>
```

```
<literal text="+"/>
```

```
<Expr> <number text="0" </Expr>
```

```
</Expr>
```

```
</program>
```

----- Output Parse Tree -----

```
txl -Dparse es.txt Expr.Grm
```

First example: transforming

Txl -Dapply es.txt Expr.txt

Input: $(9+0) * 1$

Transforming ...

$9 + 0 \implies 9$ [removePlusZero]

$(9) \implies 9$ [resolveBracketedExpressions]

$(9 + 0) * 1 \implies 9 * 1$ [main]

$9 * 1 \implies 9$ [removeMultiplicationByOne]

$9 * 1 \implies 9$ [main]

9

First example: unparsing

◆ [NL] force a new line of output.

◆ [IN] indent all following output lines by four spaces.

◆ [EX] extend all following output lines by four spaces.

Example:

```
define Procedure [id] [Parameters] [NL] [IN]
  'begin [NL] [IN]
    [body] [NL] [EX]
  'end [NL] [EX]
end define
```