A strategy recogniser

Johan Jeuring
Joint work with Alex Gerdes and Bastiaan Heeren

Utrecht University and Open Universiteit Nederland
Computer Science

CEFP, Budapest, Hungary

June 2011

Open Universiteit
www.ou.nl

Running a strategy

A sequence of rules follows a strategy iff the sequence of rules is (a prefix of) a sentence in the language generated by the strategy.

An exercise gives us an initial term (say t_0), and we are only interested in sequences of rules that can be applied successively to this term.

A possible derivation that starts with t_0 can be depicted in the following way:

$$t_0 \xrightarrow{r_0} t_1 \xrightarrow{r_1} t_2 \xrightarrow{r_2} t_3 \xrightarrow{r_3} \ldots$$

Running a strategy is defined by:

$$\mathit{run}\ \sigma\ t_0 = \{ t_{n+1} \mid r_0 \ldots r_n \in \mathcal{L}\ (\sigma), \forall_{i \in 0 \ldots n} : t_{i+1} \in \mathit{apply}\ r_i\ t_i \}$$

Implementing function *run*

- *run* specifies how to run a strategy
- It amongst others enumerates all sentences in the language of a strategy
- Enumerating all sentences is infeasible in practice.
- This lecture defines a practical implementation of a strategy recogniser.

Outline of presentation

Reusing parser libraries?

Representing grammars

Functions *empty* and *firsts*

Strategies

Smart constructors

Running a strategy

Tracing a strategy

Feedback scripts

Reusing parser libraries?

Reusing existing parser libraries?

Instead of designing our own recogniser, we could reuse existing parsing libraries and tools.

Our problem is quite different from other parsing applications:

- efficiency is not a key concern as long as we do not have to enumerate all sentences. The length of the input is very limited.
- we are not building an abstract syntax tree of the solution to the exercise.

Still, one of the 67 parsing libraries on Hackage should work?

Problems with parsing libraries

- We are only interested in sequences of transformation rules that can be applied to some initial term
- Grammar analyses for constructing a parsing table cannot take the term from which you start into account
- The ability to diagnose errors in the input highly influences the quality of the feedback services
- We have to recognise prefixes
- We cannot use backtracking and look-ahead because we want to recognise strategies at each intermediate step
- Labels help to describe the structure of a strategy in the same way as non-terminals do in a grammar
- Parsing libraries do not offer combinators for interleaving
- A strategy should be serialisable

Representing grammars

Representing grammars

$$
\begin{aligned}
\textbf{data}\ \mathit{Grammar}\ a = {} & \mathit{Symbol}\ a \\
\mid\ & \mathit{Succeed} \\
\mid\ & \mathit{Fail} \\
\mid\ & \mathit{Grammar}\ a\ \ {:}{|}{:}\ \ \mathit{Grammar}\ a \\
\mid\ & \mathit{Grammar}\ a\ \ {:}{\star}{:}\ \ \mathit{Grammar}\ a \\
\mid\ & \mathit{Grammar}\ a\ \ {:}\%{:}\ \ \mathit{Grammar}\ a \\
\mid\ & \mathit{Grammar}\ a\ {:}\%{>}{:}\ \mathit{Grammar}\ a \\
\mid\ & \mathit{Atomic}\ (\mathit{Grammar}\ a) \\
\mid\ & \mathit{Rec}\ \mathit{Int}\ (\mathit{Grammar}\ a) \\
\mid\ & \mathit{Var}\ \mathit{Int}
\end{aligned}
$$

a is used for symbols: for strategies, the symbols are rules, but also Enter and Exit steps associated with a label.

Alternative representations for recursion are higher-order fixed point functions, or nameless terms using De Bruijn indices.

[CEFP: **A strategy recogniser**]

many again

$$
\begin{aligned}
&\mathit{many} &&:: \mathit{Grammar}\ a \rightarrow \mathit{Grammar}\ a\\
&\mathit{many}\ \sigma &&= \mathit{Rec}\ 0\ (\mathit{Succeed}\ {:}|{:}\ (\sigma\ {:}\star{:}\ \mathit{Var}\ 0))
\end{aligned}
$$

Later we will see that smart constructors are more convenient for writing such a combinator.

Functions *empty* and *firsts*

Generating sentences

We use the functions *empty* and *firsts* to generate sentences of a grammar.

Function *empty*: specification

- $empty\ (\sigma) = \epsilon \in \mathcal{L}\ (\sigma)$

Function *empty*: implementation

$$
\begin{aligned}
&\mathit{empty} && :: \mathit{Grammar}\ a \rightarrow \mathit{Bool} \\
&\mathit{empty}\ (\mathit{Symbol}\ a) && = \mathit{False} \\
&\mathit{empty}\ \mathit{Succeed} && = \mathit{True} \\
&\mathit{empty}\ \mathit{Fail} && = \mathit{False} \\
&\mathit{empty}\ (\sigma :|: \tau) && = \mathit{empty}\ \sigma \vee \mathit{empty}\ \tau \\
&\mathit{empty}\ (\sigma :\star: \tau) && = \mathit{empty}\ \sigma \wedge \mathit{empty}\ \tau \\
&\mathit{empty}\ (\sigma :\%: \tau) && = \mathit{empty}\ \sigma \wedge \mathit{empty}\ \tau \\
&\mathit{empty}\ (\sigma :\%>: \tau) && = \mathit{False} \\
&\mathit{empty}\ (\mathit{Atomic}\ \sigma) && = \mathit{empty}\ \sigma \\
&\mathit{empty}\ (\mathit{Rec}\ i\ \sigma) && = \mathit{empty}\ \sigma \\
&\mathit{empty}\ (\mathit{Var}\ i) && = \mathit{False}
\end{aligned}
$$

Function *firsts*: specification

$$\forall a, x : ax \in \mathcal{L}(\sigma) \Leftrightarrow \exists \sigma' : (a, \sigma') \in \mathit{firsts}(\sigma) \land x \in \mathcal{L}(\sigma')$$

Splitting off an atomic part

In *firsts* we deal with interleaving and atomicity.

For the case σ :%>: τ we need to split σ into an atomic part and a remainder: $\mathit{Atomic}\ \sigma' :\!\star\!: \sigma''$. After σ' we continue with σ'' :%: τ. Here we use the property:

$$(\langle a :\!\star\!: \sigma\rangle :\!\star\!: \tau) \;\text{:\%>:}\; u = \langle a :\!\star\!: \sigma\rangle :\!\star\!: (\tau \;\text{:\%:}\; u)$$

Function *split*

Function *split* transforms a strategy into (a, x, y), which should be interpreted as $\langle a :\!\star\!: x\rangle :\!\star\!: y$.

$$
\begin{array}{ll}
\mathit{split} :: \mathit{Grammar}\ a \rightarrow [(a, \mathit{Grammar}\ a, \mathit{Grammar}\ a)] & \\
\mathit{split}\ (\mathit{Symbol}\ a) & = [(a, \mathit{Succeed}, \mathit{Succeed})] \\
\mathit{split}\ \mathit{Succeed} & = [\,] \\
\mathit{split}\ \mathit{Fail} & = [\,] \\
\mathit{split}\ (\sigma :|: \tau) & = \mathit{split}\ \sigma +\!\!+\ \mathit{split}\ \tau \\
\mathit{split}\ (\sigma :\!\star\!: \tau) & = [(a, x, y :\!\star\!: \tau) \mid (a, x, y) \leftarrow \mathit{split}\ \sigma] +\!\!+ \\
 & \quad \textbf{if}\ \mathit{empty}\ \sigma\ \textbf{then}\ \mathit{split}\ \tau\ \textbf{else}\ [\,] \\
\mathit{split}\ (\sigma :\%: \tau) & = \mathit{split}\ (\sigma :\%>: \tau) +\!\!+\ \mathit{split}\ (\tau :\%>: \sigma) \\
\mathit{split}\ (\sigma :\%>: \tau) & = [(a, x, y :\%: \tau) \mid (a, x, y) \leftarrow \mathit{split}\ \sigma] \\
\mathit{split}\ (\mathit{Atomic}\ \sigma) & = [(a, x :\!\star\!: y, \mathit{Succeed}) \mid (a, x, y) \leftarrow \mathit{split}\ \sigma] \\
\mathit{split}\ (\mathit{Rec}\ i\ \sigma) & = \mathit{split}\ (\mathit{replaceVar}\ i\ (\mathit{Rec}\ i\ \sigma)\ \sigma) \\
\mathit{split}\ (\mathit{Var}\ i) & = \mathit{error}\ \texttt{"unbound Var"}
\end{array}
$$

[CEFP: **A strategy recogniser**]

firsts in terms of *split*

$$\begin{aligned}
&\mathit{firsts} && :: \mathit{Grammar}\ a \rightarrow [(a, \mathit{Grammar}\ a)] \\
&\mathit{firsts}\ \sigma && = [(a, x :\star: y) \mid (a, x, y) \leftarrow \mathit{split}\ \sigma]
\end{aligned}$$

Left-recursion again

- The definition of *firsts* (*split*) shows why left-recursion is problematic.
- if grammar σ accepts the empty sentence, then running the grammar *many* σ may result in non-termination.
- The problem with left recursion can be partially circumvented by restricting the number of recursion points (*Rec*s and *Var*s) that are unfolded in the definition of *split* (*Rec* i σ).

Strategies

Labels

- *Grammar* has no alternative for labels
- We use label information to trace where we are in a strategy by inserting Enter and Exit steps for each labelled substrategy
- We attach feedback messages to labels

$$\textbf{data}\ \mathit{Step}\ l\ a = \mathit{Enter}\ l \mid \mathit{Step}\ (\mathit{Rule}\ a) \mid \mathit{Exit}\ l$$

l represents the type of information associated with each label.

The type *Rule* is parameterised by the type of values on which the rule can be applied.

Strategy

With the *Step* datatype, we can now specify a type for strategies:

$$\mathbf{type}\ \mathit{LabelInfo}\ \ = \mathit{String}$$
$$\mathbf{data}\ \mathit{Strategy}\ a = S\ \{\mathit{unS} :: \mathit{Grammar}\ (\mathit{Step}\ \mathit{LabelInfo}\ a)\}$$

The *Strategy* datatype wraps a grammar, where the symbols of this grammar are steps.

From *Step* to *Strategy*

$$\mathit{fromStep} :: \mathit{Step}\ \mathit{LabelInfo}\ a \rightarrow \mathit{Strategy}\ a$$
$$\mathit{fromStep} = S \circ \mathit{Symbol}$$

IsStrategy

The (un)wrapping of strategies quickly becomes cumbersome when defining functions over strategies.

$$
\begin{array}{l}
\textbf{class}\ \mathit{IsStrategy}\ f\ \textbf{where} \\
\quad \mathit{toStrategy} :: f\ a \rightarrow \mathit{Strategy}\ a \\
\textbf{instance}\ \mathit{IsStrategy}\ \mathit{Rule}\ \textbf{where} \\
\quad \mathit{toStrategy} = \mathit{fromStep} \circ \mathit{Step} \\
\textbf{instance}\ \mathit{IsStrategy}\ \mathit{Strategy}\ \textbf{where} \\
\quad \mathit{toStrategy} = \mathit{id}
\end{array}
$$

IsStrategy

LabeledStrategy represents strategies that have a label.

$$\textbf{data}\ \mathit{LabeledStrategy}\ a = \mathit{Label}\ \{\mathit{labelInfo} :: \mathit{LabelInfo}\ ,\ \mathit{unlabel} :: \mathit{Strategy}\ a\}$$

A labelled strategy is turned into a (normal) strategy by surrounding its strategy with *Enter* and *Exit* steps.

$$\begin{aligned} &\textbf{instance}\ \mathit{IsStrategy}\ \mathit{LabeledStrategy}\ \textbf{where} \\ &\quad \mathit{toStrategy}\ (\mathit{Label}\ a\ \sigma) = \mathit{fromStep}\ (\mathit{Enter}\ a) \\ &\qquad\qquad\qquad\qquad\qquad :\!\star\!:\ \sigma \\ &\qquad\qquad\qquad\qquad\qquad :\!\star\!:\ \mathit{fromStep}\ (\mathit{Exit}\ a) \end{aligned}$$

Smart constructors

Smart constructors

- A smart constructor is a function that in addition to constructing a value performs some checks, simplifications, or conversions
- We use smart constructors for simplifying grammars.
- We introduce a smart constructor for every alternative of the strategy language
- Definitions for *succeed* and *fail* are straightforward:

$$
\begin{aligned}
&\mathit{succeed}, \mathit{fail} :: \mathit{Strategy}\ a \\
&\mathit{succeed} = S\ \mathit{Succeed} \\
&\mathit{fail} \ = S\ \mathit{Fail}
\end{aligned}
$$

A smart constructors for labels

$$\mathit{label} :: \mathit{IsStrategy}\ f \Rightarrow \mathit{LabelInfo} \rightarrow f\ a \rightarrow \mathit{LabeledStrategy}\ a$$
$$\mathit{label}\ \mathit{str} = \mathit{Label}\ \mathit{str} \circ \mathit{toStrategy}$$

A smart constructors for choice

The other constructors return a value of type *Strategy*, and overload their strategy arguments.

For choices, we remove occurrences of *Fail*, and we associate the alternatives to the right.

$$
\begin{array}{l}
(\lhd\!|\!\rhd) :: (\mathit{IsStrategy}\ f, \mathit{IsStrategy}\ g) \Rightarrow f\ a \rightarrow g\ a \rightarrow \mathit{Strategy}\ a \\
(\lhd\!|\!\rhd) = \mathit{lift2}\ \mathit{op} \\
\quad \mathbf{where} \\
\qquad \begin{array}{lll}
\multicolumn{3}{l}{\mathit{op} :: \mathit{Grammar}\ a \rightarrow \mathit{Grammar}\ a \rightarrow \mathit{Grammar}\ a} \\
\mathit{op}\ \mathit{Fail} & \tau & = \tau \\
\mathit{op}\ \sigma & \mathit{Fail} & = \sigma \\
\mathit{op}\ (\sigma :|: \tau) & u & = \sigma\ \text{`}\mathit{op}\text{`}\ (\tau\ \text{`}\mathit{op}\text{`}\ u) \\
\mathit{op}\ \sigma & \tau & = \sigma :|: \tau
\end{array}
\end{array}
$$

Lifting functions

Lifting functions turn a function that works on the *Grammar* datatype into an overloaded function that returns a strategy.

$$
\begin{aligned}
&\mathit{lift1}\ \mathit{op} = S \circ \mathit{op} \circ \mathit{unS} \circ \mathit{toStrategy}\\
&\mathit{lift2}\ \mathit{op} = \mathit{lift1} \circ \mathit{op} \circ \mathit{unS} \circ \mathit{toStrategy}
\end{aligned}
$$

A smart constructors for sequence

The smart constructor $<\star>$ for sequences removes the unit element *Succeed*, and propagates the absorbing element *Fail*.

$$
\begin{array}{l}
(<\star>) :: (\mathit{IsStrategy}\ f, \mathit{IsStrategy}\ g) \Rightarrow f\ a \rightarrow g\ a \rightarrow \mathit{Strategy}\ a \\
(<\star>) = \mathit{lift2}\ \mathit{op} \\
\quad \textbf{where} \\
\qquad \mathit{op} :: \mathit{Grammar}\ a \rightarrow \mathit{Grammar}\ a \rightarrow \mathit{Grammar}\ a \\
\qquad \begin{array}{lll}
\mathit{op}\ \mathit{Succeed} & \tau & = \tau \\
\mathit{op}\ \sigma & \mathit{Succeed} & = \sigma \\
\mathit{op}\ \mathit{Fail} & \_ & = \mathit{Fail} \\
\mathit{op}\ \_ & \mathit{Fail} & = \mathit{Fail} \\
\mathit{op}\ (\sigma :\star: \tau) & u & = \sigma\ \text{`}\mathit{op}\text{`}\ (\tau\ \text{`}\mathit{op}\text{`}\ u) \\
\mathit{op}\ \sigma & \tau & = \sigma :\star: \tau
\end{array}
\end{array}
$$

A smart constructor for atomic

The binary combinators for interleaving, <%> and %>, are defined in a similar fashion.

$$
\begin{array}{l}
\mathit{atomic} :: \mathit{IsStrategy}\ f \Rightarrow f\ a \rightarrow \mathit{Strategy}\ a \\
\mathit{atomic} = \mathit{lift1}\ \mathit{op} \\
\quad \textbf{where} \\
\qquad \begin{array}{ll}
\mathit{op} :: \mathit{Grammar}\ a \rightarrow \mathit{Grammar}\ a & \\
\mathit{op}\ (\mathit{Symbol}\ a) & = \mathit{Symbol}\ a \\
\mathit{op}\ \mathit{Succeed} & = \mathit{Succeed} \\
\mathit{op}\ \mathit{Fail} & = \mathit{Fail} \\
\mathit{op}\ (\mathit{Atomic}\ \sigma) & = \mathit{op}\ \sigma \\
\mathit{op}\ (\sigma :|: \tau) & = \mathit{op}\ \sigma :|: \mathit{op}\ \tau \\
\mathit{op}\ \sigma & = \mathit{Atomic}\ \sigma
\end{array}
\end{array}
$$

A smart constructor for recursion

$$
\begin{aligned}
&\mathit{fix} :: (\mathit{Strategy}\ a \rightarrow \mathit{Strategy}\ a) \rightarrow \mathit{Strategy}\ a \\
&\mathit{fix}\ f = \mathit{lift1}\ (\mathit{Rec}\ i)\ (\mathit{make}\ i) \\
&\textbf{where} \\
&\quad \mathit{make} = f \circ S \circ \mathit{Var} \\
&\quad \mathit{is} \quad\;\; = \mathit{usedNumbers}\ (\mathit{unS}\ (\mathit{make}\ 0)) \\
&\quad i \quad\;\;\; = \textbf{if}\ \mathit{null}\ \mathit{is}\ \textbf{then}\ 0\ \textbf{else}\ \mathit{maximum}\ \mathit{is} + 1
\end{aligned}
$$

- First, we pass f a strategy with the grammar $\mathit{Var}\ 0$, and we inspect which numbers are used (variable is of type $[\mathit{Int}]$). Based on this information, we determine the next number to use (variable i)
- We apply f for the second time using grammar $\mathit{Var}\ i$, and bind these *Var*s to the top-level *Rec*

many again

We define the repetition combinator *many* with the smart constructors.

$$many :: IsStrategy\ f \Rightarrow f\ a \rightarrow Strategy\ a$$
$$many\ \sigma = fix\ \$\ \lambda x \rightarrow succeed <|> (\sigma <\star> x)$$

Running a strategy

Applying a rule

To run a strategy, we apply the rules.

$$
\begin{aligned}
&\textbf{class}\ \mathit{Apply}\ f\ \textbf{where}\\
&\quad \mathit{apply} :: f\ a \to a \to [a]\\
&\textbf{instance}\ \mathit{Apply}\ \mathit{Rule}\\
&\qquad \text{-- implementation provided in framework}\\
&\textbf{instance}\ \mathit{Apply}\ (\mathit{Step}\ l)\ \textbf{where}\\
&\quad \mathit{apply}\ (\mathit{Step}\ r) = \mathit{apply}\ r\\
&\quad \mathit{apply}\ \_ \qquad\quad\ \ = \mathit{return}
\end{aligned}
$$

Running a strategy

A strategy is a grammar over rewrite rules and *Enter* and *Exit* steps for labels.

$$
\begin{aligned}
&run :: Apply\, f \Rightarrow Grammar\ (f\ a) \rightarrow a \rightarrow [a]\\
&run\ \sigma\ a = [a \mid empty\ \sigma]\\
&\qquad\qquad ++ [c \mid (f, \tau) \leftarrow firsts\ \sigma\\
&\qquad\qquad\qquad\ \ , b \qquad \leftarrow apply\ f\ a\\
&\qquad\qquad\qquad\ \ , c \qquad \leftarrow run\ \tau\ b\\
&\qquad\qquad\quad\ ]
\end{aligned}
$$

Applying a strategy

Now that we have defined the function *run* we can also make *Strategy* and *LabeledStrategy* instances of class *Apply*:

$$
\begin{aligned}
&\textbf{instance}\ \mathit{Apply}\ \mathit{Strategy}\ \textbf{where}\\
&\quad \mathit{apply} = \mathit{run} \circ \mathit{unS}\\
&\textbf{instance}\ \mathit{Apply}\ \mathit{LabeledStrategy}\ \textbf{where}\\
&\quad \mathit{apply} = \mathit{apply} \circ \mathit{toStrategy}
\end{aligned}
$$

A breadth-first *run*

run returns results in a depth-first manner.

We define a variant of *run* which exposes breadth-first behaviour:

$$
\begin{aligned}
&\mathit{runBF} :: \mathit{Apply}\ f \Rightarrow \mathit{Grammar}\ (f\ a) \rightarrow a \rightarrow [[a]] \\
&\mathit{runBF}\ \sigma\ a = [a \mid \mathit{empty}\ \sigma] \\
&\qquad\qquad : \ \mathit{merge}\ [\mathit{runBF}\ \tau\ b \mid (f, \tau) \leftarrow \mathit{firsts}\ \sigma \\
&\qquad\qquad\qquad\qquad\qquad\qquad\quad , b \qquad \leftarrow \mathit{apply}\ f\ a \\
&\qquad\qquad\qquad\quad] \\
&\quad \mathbf{where}\ \mathit{merge} = \mathit{map}\ \mathit{concat} \circ \mathit{transpose}
\end{aligned}
$$

Tracing a strategy

Tracing a strategy

We extend *run*'s definition to keep a trace of the steps that have been applied:

$$
\begin{array}{l}
\mathit{runTrace} :: \mathit{Apply}\ f \Rightarrow \mathit{Grammar}\ (f\ a) \rightarrow a \rightarrow [(a, [f\ a])] \\
\mathit{runTrace}\ \sigma\ a = [(a, [\,])\qquad\quad \mid \mathit{empty}\ \sigma] \\
\qquad\qquad\qquad + [(c, (f : \mathit{fs})) \mid (f, \tau) \leftarrow \mathit{firsts}\ \sigma \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad\ , b \qquad\ \leftarrow \mathit{apply}\ f\ a \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad\ , (c, \mathit{fs}) \leftarrow \mathit{runTrace}\ \tau\ b \\
\qquad\qquad\qquad\quad\ ]
\end{array}
$$

In case of a strategy, we can thus obtain the list of *Enter* and *Exit* steps seen so far.

Tracing *addFractions*

We run *addFractions* on the term $\frac{2}{5} + \frac{2}{3}$.

$$\frac{2}{5} + \frac{2}{3} \;=\; \frac{6}{15} + \frac{2}{3} \;=\; \frac{6}{15} + \frac{10}{15} \;=\; \frac{16}{15} \;=\; 1\frac{1}{15}$$

Tracing *addFractions*

$$
\begin{array}{lllll}
[& \mathit{Enter}\ \ell_0, & \mathit{Enter}\ \ell_1, & \mathit{Step}\ \mathsf{LCD}, & \mathit{Exit}\ \ell_1 \\
, & \mathit{Enter}\ \ell_2, & \mathit{Step}\ \mathit{down}_{(0)}, & \mathit{Step}\ \underline{\mathsf{Rename}}, & \mathit{Step}\ \mathit{up} \\
, & \mathit{Step}\ \mathit{down}_{(1)}, & \mathit{Step}\ \underline{\mathsf{Rename}} & \mathit{Step}\ \mathit{up}, & \mathit{Step}\ \mathit{not} \\
, & \mathit{Exit}\ \ell_2, & \mathit{Enter}\ \ell_3, & \mathit{Step}\ \underline{\mathsf{Add}}, & \mathit{Exit}\ \ell_3 \\
, & \mathit{Enter}\ \ell_4, & \mathit{Step}\ \underline{\mathsf{Simpl}}, & \mathit{Exit}\ \ell_4, & \mathit{Exit}\ \ell_0 \\
]
\end{array}
$$

We determine at each point in the derivation where we are in the strategy by enumerating the *Enter* steps without their corresponding *Exit* step.

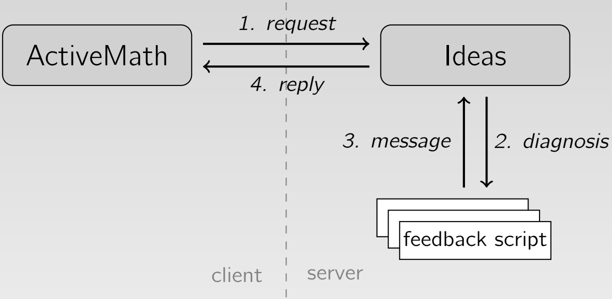
Feedback scripts

Textual feedback

Desirable features for textual feedback:

- support for different levels (abstract, concrete, bottom-out)
- messages available in multiple languages
- can contain dynamic parts such as formulas that depend on the exercise at hand
- should be easy for teachers to adapt feedback

Our approach: feedback scripts



- Server has feedback scripts containing textual messages
- Scripts are used to transform an abstract diagnosis into a message, which is returned to the learning environment
- Possible to select a specific script in a request (e.g. for choosing the language)
- Syntax of the scripts might slightly deviate from syntax presented here.

[CEFP: A strategy recogniser]

Translating rules

$$a \cdot (b + c) \quad \Rightarrow \quad a \cdot b + a \cdot c \qquad (\texttt{algebra.equations.linear.distr-times})$$

$$a = b \quad \Rightarrow \quad b = a \qquad (\texttt{algebra.equations.linear.flip})$$

```
namespace algebra.equations.linear

text distr-times = {distribute}
text flip = {flip equation around}
```

- All rules are organized in a math taxonomy
- Script provides a translation for all rules of an exercise
- Declaring a namespace prevents long identifier names

Example: worked-out solution

```
text scale-to-one = {divide by @arg1}
```

$4 \cdot (x - 1) = 7$

$\Rightarrow$ *distribute*

$4 \cdot x - 4 = 7$

$\Rightarrow$ *bring constants to right*

$4 \cdot x = 11$

$\Rightarrow$ *divide by 4*

$x = 2\frac{3}{4}$

- Rule translations are used in worked-out solutions
- Attributes (such as `@arg1`) are replaced by dynamic content

Hints at different levels

```
hint abstract = {Use the procedure for solving linear
   equations:  If present, remove parentheses, and
   isolate variable x}
hint concrete = {@expected}
hint bottom-out = {@expected:  this results in @after}
```

- Attribute `@expected` is replaced by the (translation of the) rule suggested by the strategy
- Attribute `@after` represents the term after application of the expected rule
- Feedback texts can be further tailored for a specific rule-level combination
- OpenMath is used for encoding mathematical objects

[CEFP: **A strategy recogniser**]

Feedback at different levels

```
feedback noteq = {This is incorrect.}
feedback buggy = {This is incorrect.  @recognized}

feedback ok    = {Well done!  You used @recognized}
feedback same  = {This is correct.}

# Messages for buggy rules
text buggy.distr-times-plus = {Did you try to use
   distribution?  One term was not multiplied.}
text buggy.negate-one-side = {It seems you have negated
   the terms on one side only.}
```

- The script contains messages for each type of diagnosis: *buggy*, *noteq*, *ok*, *same*, *detour*, and *unknown*
- Messages can again be specialized for the levels

More features

- String definitions and an include mechanism provide a way to reuse text fragments
- Conditionals make it possible to report tailor-made feedback messages for specific cases
- Many more attributes help to enrich the messages with dynamic content, including attributes for the number of steps remaining or the subexpression that is replaced
- Also strategy labels can be used to construct messages
- Feedback scripts can be analyzed for correctness:
  - Syntax errors are reported
  - Unknown attributes and non-existing rule identifiers result in warnings
  - Scripts can be tested for completeness, i.e., whether all cases are covered by the script

Conclusions

- We generate solutions to an exercise using the *run* function on a strategy
- The *run* function is defined in terms of *empty* and *firsts*: well-known functions on CFGs
- Smart constructors help in simplifying strategies
- Labels are used to trace (strategy) steps through an exercise
- Feedback scripts provide textual feedback to users solving exercises