Strategy Parser
Technical Reference Manual

W.Pasman

Manual version 20.9.7 for Strategy Parser version 24.7.7

About this Manual
This manual aims at giving a technical explanation of the strategy parser, to enable maintenance and extension of the parser. It is assumed that the reader knows how to operate the parser, as described in the Strategy Parser User Manual [Pasman07].

Table of Contents

1. Introduction

Overview

This section discusses the major functional blocks of the strategy parser. Figure 1 shows an overview of the most important data structures and functions in the strategy parser.

To start with, the strategy parser needs a strategy specification. Strictly speaking this is not the topic of this manual. However there is not yet a final report on the strategy specification and in fact this is still under research. Therefore, the strategy specification will be presented on a technical level here in Chapter 4: Strategy Specification Objects. It is assumed that the reader is familiar with the grammar description used in our parser, as described in the user manual [Pasman07].

The core component of the parser is the parse state. The parse state stores the entire parse situation after the student applied a given sequence of rewrite actions to the problem term. The parse state can be extended incrementally, using Scanner with the latest student rewrite action. The current parse situation can be evaluated at all times using the Status, OnTrack and Finished functions.

Internally the parse state is represented using dotted rules and shift possibilities, objects derived from the strategy. This manual will refer to them as the strategy parser objects.

Finally, the complex internal parse state can be converted into a non-ambiguous parse tree. This tree makes it simpler to determine which feedback is appropriate, by expliciting the next step(s) the student needs to take within the hierarchy of strategic steps.

Acknowledgments

This project was made possible by the support of the SURF Foundation, the higher education and research partnership organisation for Information and Communications Technology (ICT). For more information about SURF, please visit www.surf.nl

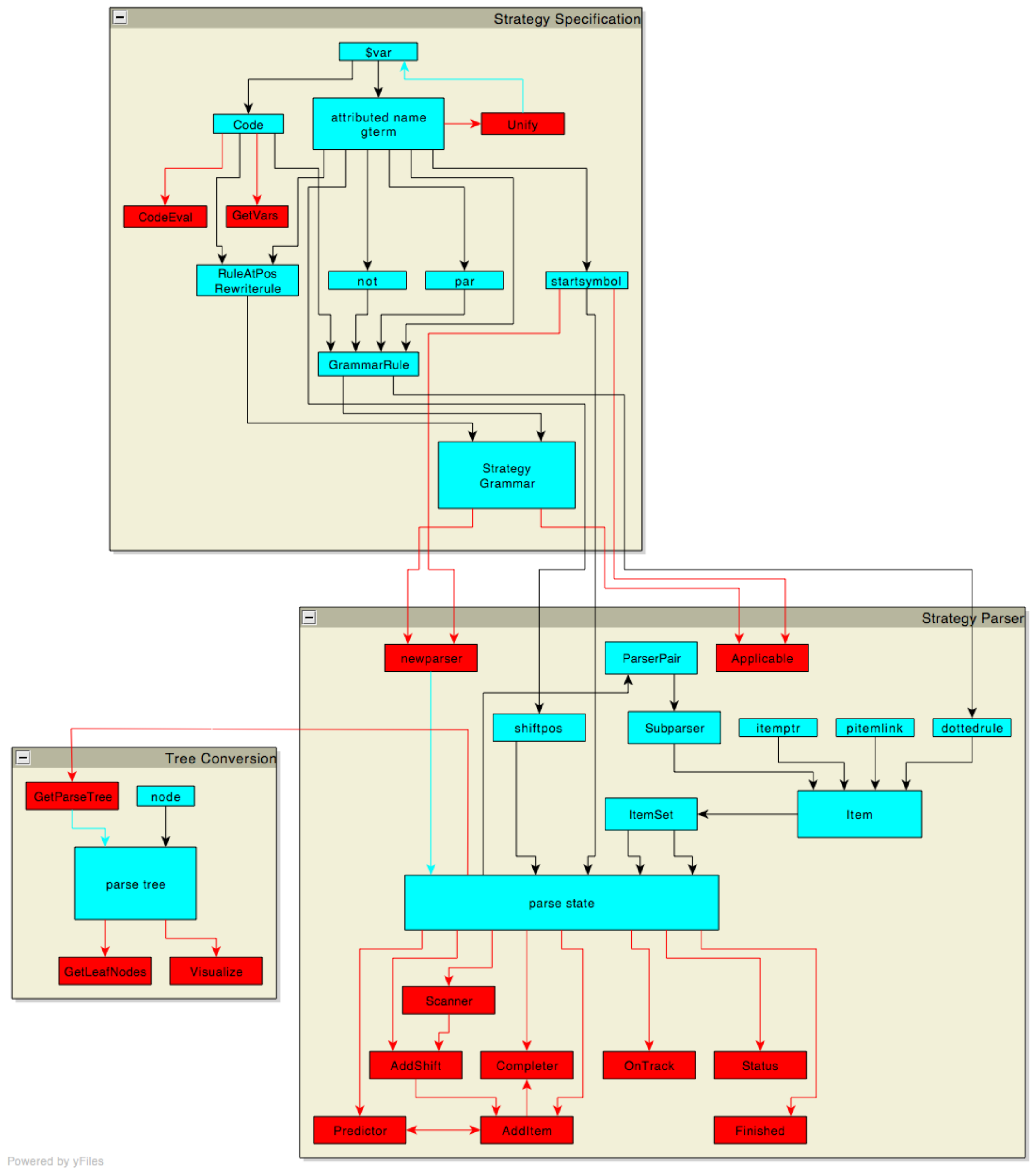

Figure 1. Overview of main data objects (blue) and functions (red), categorized in two main clusters "strategy specification" and "strategy parser", with a small third cluster for a tree-converted/simplified parse. Black arrows from A to B indicate a "A used in B" relation. Red arrows to a function indicate important input argument, and light blue arrows out of a function indicate important output arguments.

2. About Parsing

This chapter give a very short but necessary background for readers new to parsing. No attributes are used in the first sections, to keep the explanation simple. The discussion starts with grammars containing only grammar rules, and no rewrite rules.

Context Free Grammar

A **context-free grammar** G represents a set of strings of symbols. A string of symbols $\alpha_1 \ldots \alpha_n$ can be in this grammar, also written as $\alpha_1 \ldots \alpha_n \in G$. What makes the grammar context-free is the way this set is defined.
The context free grammar G is defined by means of a set of grammar rules, plus a start symbol. Each **grammar rule** is an object $N \to \alpha_1 \ldots \alpha_n$ where N is a nonterminal and each of the string of symbols α_i is either a nonterminal or a terminal. **Nonterminals** will be written as upper case characters, and **terminals** as lower case characters. The **start symbol** is the nonterminal of one of the grammar rules.
Now we can repeatedly apply the grammar rules to the start symbol, as if they are rewrite rules. One such application of a grammar rule on a string, giving another string of symbols, is written as $\alpha_1 \ldots \alpha_n \Rightarrow \beta_1 \ldots \beta_m$. For instance, if we have this grammar

$$G = \begin{cases} E \to E + E \\ E \to E * E \\ E \to a \end{cases}$$

and start symbol E, we can write

$$E \Rightarrow E + E \Rightarrow E + E * E \Rightarrow a + E * E \Rightarrow a + E * a \Rightarrow a + a * a$$

It is said that grammar G **produces** a+a*a, or shortly $G \Rightarrow^* a + a * a$.
Now it is clear why nonterminals are called that way: because the left hand side of a grammar rule is always a nonterminal, only nonterminals in the symbol string can be rewritten further. The lower case terminals will never be rewritten any more.
This definition of production is used to define when a string of symbols is in the grammar: $\alpha_1 \ldots \alpha_n \in G$ if and only if $G \Rightarrow^* \alpha_1 \ldots \alpha_n$.

Basic Chart Parsing

Parsing is a method to answer the question whether a given **input string** of symbols is in the grammar: is $\alpha_1 \ldots \alpha_n \in G$ or not?. For instance, can G produce *a*a+a*a*?
One procedure to answer such a question, or to **parse** the string, is with a chart parser. Instead of the usual Earley chart parser [Earley70] we use a simpler chart parser that works directly with the grammar [Pasman91]. We will refer to this parser as **basic parser**.

Basic Components

The basic parser contains three basic components: the itemsets that in turn contain dotted rules, and shift possibilities.

To track the parse state, sets are created containing dotted rules. A **dotted rule** is a grammar rule with a dot ('•') somewhere in the string of symbols. So for example if $N \rightarrow a\ \ b\ \ c$ is a grammar rule then $N \rightarrow a\ \ b \bullet\ c$ could be a dotted rule.
Sets of dotted rules, also called **item sets**, will be placed at every position between two symbols of the input string, before the first and after the last.
Furthermore, a set of shift possibilities is created between the item sets. A **shift possibility** is a triple <symbol,start,end> where symbol is the (non)terminal that was succesfully parsed, and start and end point to an item set. A shift possibility represents a parsed symbol.
Finally, to make parsing easier, **links** are created between dotted rules. Each dotted rule may have multiple such links. Links point to dotted rules that have the dot more to the right. An item is chain-linked to another item if one can get from the first to the second item following the links.

Parsing Procedure

With these three objects the parsing procedure can be described with a a few procedure rules that should be applied *exhaustively* to complete the parse (in any order):

rulename	procedure
scaninput	**if** symbol α_i in the input string $\alpha_1...\alpha_n$, **then add** a shift possibility <α_i,i,i+1>.
startpoint	**if S** is the start symbol **and** $S \rightarrow \alpha_1...\alpha_n$ is a grammar rule **then add** a dotted rule $S \rightarrow \bullet\alpha_1...\alpha_n$ to item set 1
scanner	**if** there is a dotted rule $M \rightarrow \alpha_1...\bullet\,\alpha_m...\alpha_n$ in some itemset L **and** there is a shift possibility <α_m,L,R> for some R **then add** the dotted rule $M \rightarrow \alpha_1...\alpha_m \bullet ...\alpha_n$ to itemset R **and link** this the rule in set L to this new dotted rule.
predictor	**if** there is a dotted rule $M \rightarrow \alpha_1...\bullet\,\alpha_m...\alpha_n$ in some itemset L **and** $\alpha_m \rightarrow \beta_1...\beta_l$ is a grammar rule **then add** $\alpha_m \rightarrow \bullet\beta_1...\beta_l$ to itemset L
completer	**if** there is a dotted rule $N \rightarrow \alpha_1...\alpha_n \bullet$ in set R **and** dotted rule $N \rightarrow \bullet\alpha_1...\alpha_n$ in set L is chain-linked to it **then** add a shift possibility <N,L,R>

These rules are pretty intuitive. To complete a parse, the dot has to be shifted over each of the symbols in a dotted rule. This is done by the scanner. Shifting the dot over a symbol X is possible only if the symbol X could actually be parsed at that point. If that was possible, a shift possibility was created for X to reflect this.
Initially the parser only searches for the start symbol S, not for other symbols X. It is the job of the predictor to recognise that a rule needs symbol X to be recognised and to start a parser for it.
The completer recognises that the dot has reached the right end of a dotted rule and copies this knowledge into a new shift possibility.
Parsing succeeds, or $\alpha_1 \ldots \alpha_n \in G$, if and only if the parsing procedure results in a shift possibility <S,1,n+1>.

Example, Graphical Representation

This section discusses an example to illustrate the parser and parsing procedure. Assume we have the grammar

$$G = \begin{cases} S \to b \\ S \to c\ \ S \end{cases}$$

and the input string is "c b". We start running the scaninput rule and the startpoint rule both once. The result is shown in Figure 2. In these parse state figures, itemsets are denoted with rounded-square blocks with on top the item set number for reference. Shift possibilities are denoted with a two-sided arrow with a symbol on it below the itemsets.

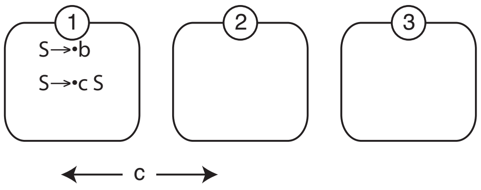

Figure 2. Parser state after running scaninput and startpoint once.

Now, the scanner can be applied, using the shift possibility <c,1,2>, adding the dotted rule $S \to c \bullet S$ to set 2. The link is indicated with an arrow to the new dotted rule (Figure 3).

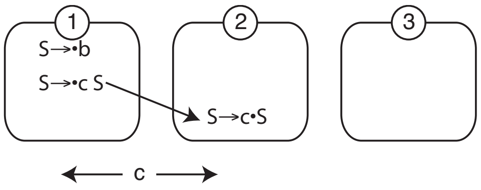

Figure 3. Result after applying the scanner.

Next, it is possible to apply the predictor rule, which adds new dotted rules to set 2, resulting in the state of Figure 4:

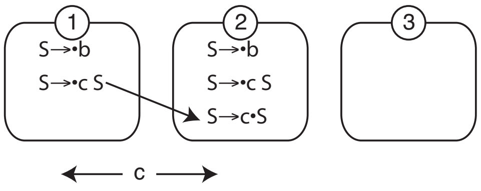

Figure 4. Repeated application of the predictor introduces new items to itemset 2.

Applying the scaninput rule allows the just introduced dotted rule $S \to \bullet b$ to shift its dot and proceed to itemset 3. The new item $S \to b \bullet$ has the dot on the right, and the completer can kick in, creating a shift possibility for S between set 2 and 3 (Figure 5).

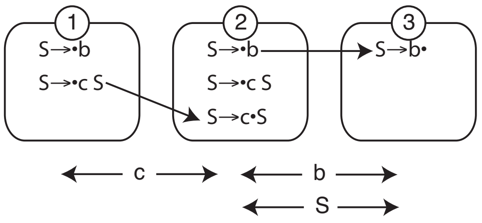

Figure 5. scaninput and completer were applied.

The new shift possibility for S enables the scanner to shift the dot in the item $S \rightarrow c \bullet S$. The new item $S \rightarrow c\ \ S\bullet$ in set 3 has the dot on the right, and the completer now can create a shift possibility <S,1,3> (Figure 6).
At this point, none of the rules is applicable anymore and the parser stops. The parse succeeded, because the final result contains a shift possibility <S,1,3>.

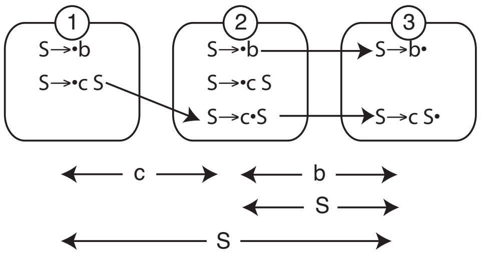

Figure 6. Final result of the parsing procedure

Parse State, Incremental Parsing

The parser as described is incremental in the sense that once a parse was done, the parse can easily be extended to a parse for the same input string extended with one symbol. We saw this already in the example, where a partial parse was first done for the 'c' alone, after which the 'b' was handled in the second phase. Extending the parse after entering a new symbol is then done just by again exhaustively applying the rules. The result of a parse (that is, exhaustively applying the rules on a given input string) can be considered as one step of a cycle of parsing, extending the input string by one symbol, extending the parse, etcetera. The result after one such a cycle is then called a **parse state**.

Parallel Parsing

A first extension of the basic parsing algorithm is to introduce the parallel operator. The parallel operator allows arbitrary permutations of the terminal symbols produced by a nonterminal. The notation for having two nonterminals N and M in parallel is N//M.
For example, the grammar

$$G = \begin{cases} S \rightarrow N // M \\ N = aa \\ M = b \end{cases}$$

produces any permutation of two 'a' and one 'b' symbols.
To handle this, first the predictor has to be modified for when the dot is in front of the parallel operator. It introduces a new object called a **subparser**. A subparser contains a set of *parser pairs*. A parser pair is a tuple <N,M,eatpattern> where N and M are parsers for each of the two nonterminals in the parallel operator. Each of these parser

pairs has a different eatpattern. An **eatpattern** is a specific way in which the parser pair eats the symbols from the input string.
The following example hopefully makes this more concrete. We take the grammar as above, and try to parse input "aba". After running the startpoint and the predictor once, the result is as in Figure 7. The subparser object has replaced a would-be rule of the form $N // M \rightarrow ...$ which is not possible in context free grammars. The single orange block is a parser pair, containing two full parsers and the empty eat pattern notated with "-". Both parsers in the parser pair ran the startpoint rule. No more rules can be applied at this point.

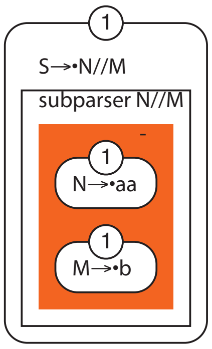

Figure 7. Parser state after running the startpoint and predictor once.

Applying the scaninput rule now becomes interesting. The dotted rule $S \rightarrow \bullet N // M$ can not do anything with the first input symbol a. But the subparser takes new scaninput in a special way: it splits each available parser pair in two new parser pairs. In the first of the two parserpairs (marked with eatpattern "0"), the first parser gets the new symbol, and in the second of the two parserpairs (marked with eatpattern "1") the second parser gets the new symbol. The other parser in the parser pair does not do anything, its state is just copied to a new itemset using a special shift possibility "*". In the running example, the first parser can do something useful with an 'a' but the second parser can not. The result is shown in Figure 8.

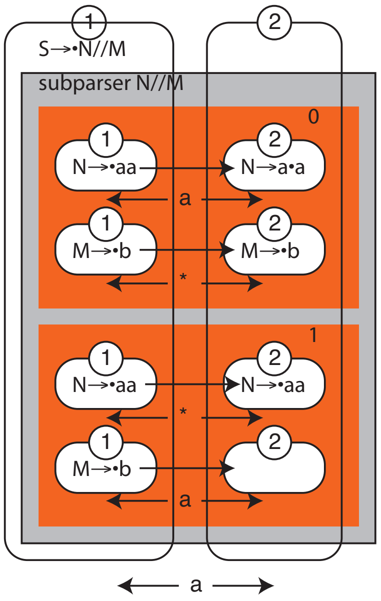

Figure 8. Parser state after running scaninput. This results in splitting the parser pairs in the subparser.

The parser can detect that the second parserpair in the subparser (the lower orange block) will never complete succesfully anymore, and can further ignore this parserpair. We delete it from the diagram to save some space.
The next input symbol is a 'b'. As before, the $S \rightarrow \bullet N // M$ can not do anything. The subparser again splits the parsers, resulting potentially in four parsers now, but the second one was dropped as it was dead anyway. This time, the second parser of the split survives as it can eat the new input 'b', while the first dies.

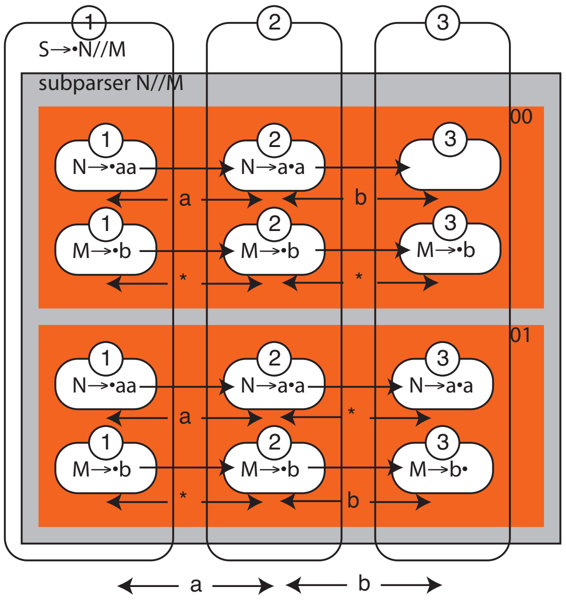

Figure 9. Result after scanning the 'b'.

Again we drop the dead parser from the pictures. The completer for the second parser notices that a "M" has been parsed, but now this has no immediate effects as the other half of the parser pair has not yet been completed.

The next input symbol "a" results in a state like Figure 10. The subparser with eatpattern 010 has succeeded parsing with both subparsers. This results in a shift possibility for N//M in the top level parser. This enables the scanner to finally shift the $S \rightarrow \bullet N // M$ to set 4, and then the completer recognises that it is complete and adds a shift possibility for S completing the parse.

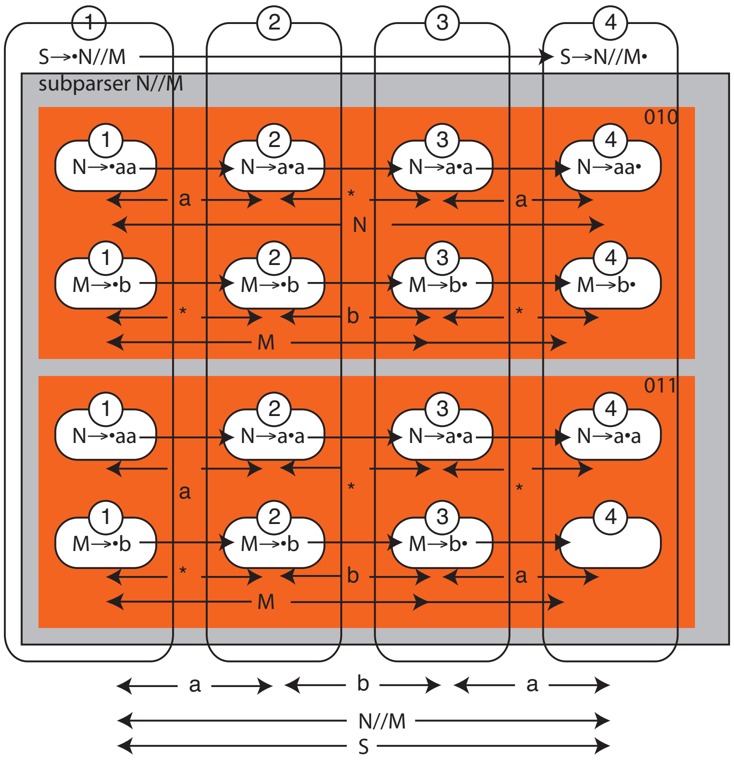

Figure 10. Result after final input symbol 'a'.

Strategy Parsing

The grammar produces a (potentially infinite) set of strings of terminal symbols starting from the start symbol. The idea behind strategy parsing is that each of these terminal symbols refers to a **rewrite rule** (see Rewrite Rule, p. 24 and [Pasman07]) that can be applied to a **problem term** that the student is trying to solve. Thus, the grammar now produces a set of strings of references to rewrite rules, or shortly a set of **rewrite sequences**. And this grammar including the rewrite rules and start symbol is called a **strategy specification**.

To relax the requirements on a strategy specification, it was decided that a rewrite sequence produced by a strategy may actually not be **applicable** to the problem term. Being not applicable is an issue arising from the rewrite rules: a rewrite rule may be not applicable, for instance if the rewrite rules is $a \rightarrow b$ but the term a does not actually apppear in the problem term. In this case, the rewrite rule fails.

On the other hand, the strategy should not produce rewrite sequences that lead to non-solutions, as the strategy specification is the currently the only object that determines whether a solution generated by the student is complete or not.

To give the strategy specification a solid meaning, it is required that

a rewrite sequence is a solution to the problem term
if and only if
the rewrite sequence is produced by a strategy specification and
the sequence is applicable to the term

Attribute Parsing

I would be very useful if grammar rules (and rewrite rules as well) could contain variables. For example, a rule to reduce a matrix may look like

```
Rowreduce($matrix,$rownr)→RedOneRow($matrix,$rownr)
Rowreduce($matrix,$rownr+1).
```

The variables in this rule are matrix and rownr, and are called **attributes**. To keep things clear all attributes have to start with a $. The parser now also has to find the appropriate attribute values. For a rule, being applicable means that there is some attribute value that makes the rule applicable.
In general, the parser can not just guess the appropriate attribute values. Instead, the strategy specification has to *compute* the values where needed. This is typically done in Code blocks (p.22).

Parse Order and Parse Modes

Attributes may be kept unassigned, they only have to be fixed whenever they are actually used in a rewrite rule. Of course this works only when a known order of evaluation exists. This order is left-to-right parsing at all levels.
Normally, the parsing is done on incomplete strings of rewrite rules: the student is not yet finished and has applied only part of the rewrite actions. In order to determine how the student is doing, the parser has to answer the question whether and how the problem can still be completed, starting with what the student already did.
To implement this, the parser operates in two **parse modes**.
In the **tracking mode**, the parser is trying to recognise rewrite steps that were actually done by the student.
in the **predicting mode**, the parser is trying to invent subsequent actions in order to complete the problem: it tries to extend the student's rewrite actions with new rewrite steps, such that the extended sequence is a solution to the problem.

Inherited and Synthesized attributes

An attribute value may be passed downwards, or **inherited**, from a top level grammar rule, as for example in this grammar:

$$G = \begin{cases} S \to T(3) \\ T(\$x) \to a \end{cases}$$

where S is the start symbol of the grammar. The parser will determine that the symbol T(3) can be parsed by setting $x to 3 in the rule T($x). In this case, only the rule for T has access to the $x.
Alternatively an attribute value may be passed upwards, or **synthesized**, from a lower level grammar rule. This is shown in these example grammars:

$$G = \begin{cases} S \to T(\$x) \\ T(3) \to a \end{cases} \qquad G' = \begin{cases} S \to T(\$x) \\ T(\$y) \to a \;\; Code[\$y = 3] \end{cases}$$

Here the parser will notice that it can parse the T($x) by setting $x=3 in the rule for S. In the grammar G, it will discover this requirement already when looking for a fitting rule, while it follows only in a later stage after the Code block sets $y=3.
To complete these examples, the attribute may also be left unassigned, for instance in

$$G = \begin{cases} S \to T(\$x) \\ T(\$x) \to a \end{cases}$$

Depending on the parse mode, an attribute value may have to be calculated or not. This is because attributes occuring in rewrite rules are set unambiguously when the student applies that rewrite rule and the parser is in tracking mode, but are not set when the parser is in predicting mode. The plan is to introduce a global attribute $ParserMode that gets set to either tracking or predicting, but this is not yet implemented.

Parsing Not

As was described in the section Strategy Parsing, a string of symbols produced from the start symbol may not be applicable to the problem term at hand. Testing for applicability can also be done starting with other symbols than the start symbol. For instance, to do an exhaustive repeat of strategy P one has to test whether strategy P is applicable or not. This is roughly what the "not(P)" grammar item does.
The main differences with applicability of the entire strategy specification is (1) that with the not(P) the parser already is in a certain state where the student did a number of rewrites and (2) when P is applicable, not the entire strategy fits but only the substrategy P.
So to be precise, when the parser encounters a not(P), the problem term has a current form and the attributes have a certain value. The parser then tests whether P is applicable to that current form, with that attributes. Note that 'P is applicable' in this case means: check whether P produces a rewrite sequence that can be applied to the current form of the problem term. Of course the result of applicability of P does not imply having found a solution to the entire problem.
The "not" is handled in the predictor. Whenever a rule with the dot right before a not(P) is encountered in set L, it checks directly for applicability. If the P is not applicable a shift possibility <not(P),L,L> is created. This will cause the dot to be shifted over the not by the scanner, as required.

Example

To give an example, consider this grammar:

$$G = \begin{cases} S \rightarrow Subs\ \ S \\ S \rightarrow not(Subs) \\ Subs \rightarrow SubsA \\ Subs \rightarrow SubsB \\ SubsA : a \Rightarrow 1 \\ SubsB : b \Rightarrow 2 \end{cases}$$

The last two rules are rewrite rules, without attributes to keep it simple. The two rules for start symbol S say that Subs has to be applied as long as possible (the only way to stop is via the not(Subs). Now we consider what happens with problem term "a+b=3". The predictor for the start symbol adds the two rules for S with the dot at the start. The predictor then finds the dot before the Subs and it adds the two rules for Subs. The predictor does nothing with the dot before items SubsA and SubsB, because SubsA and SubsB are rewrite rules and not grammar rules. The state so far looks as in Figure 11:

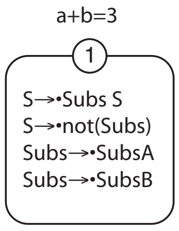

Figure 11. Initial parse state.

The predictor here notices the dot in front of the not(Subs), and at this point it gets interesting. A completely new parser is launched to check applicability of Subs on the current term "a+b=3" and no attribute values. This parser is run in predicting mode. The predictor for this parser again adds a number of new items, just as the main parser just did. The result is in Figure 12:

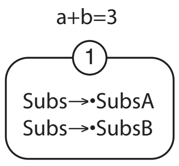

Figure 12. New parser for start symbol Subs, launched to check applicability of Subs.

The parser is in predicting mode, and therefore it does not wait for input symbols but starts guessing. The guesses are determined by the requirements of the rewrite rules. At this point, there are two rewrite rules: rule SubsA looking for an 'a', and SubsB looking for a 'b'. A check with the term at hand shows that both rules can be applied, as both an a and b are present in "a+b=3". The parser starts with trying the action SubsA, resulting in Figure 13:

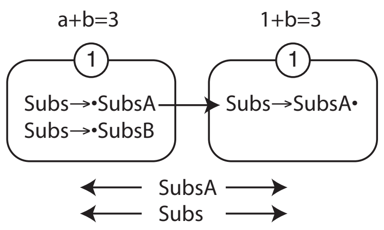

Figure 13. Result after guessing SubsA as next action.

At this point the parser notices that there is a shift possibility for the start symbol Subs, and hence Subs is applicable. SubsB would also have been applicable but that is not even tested.
Therefore the "not(Subs)" fails in Figure 11, and no further actions are taken at this point.
Next, assume the student takes actions SubsA and SubsB. After these two steps, Figure 11 has been transformed into Figure 14:

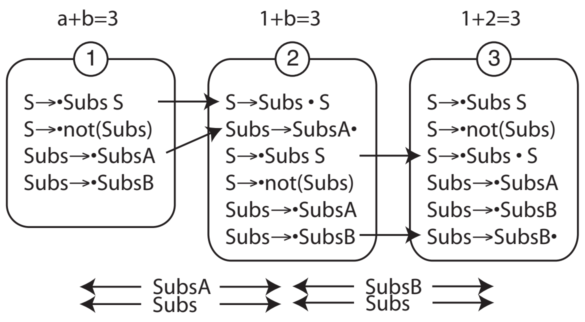

Figure 14. intermediate parser state after student took actions SubsA followed by SubsB.

But the parser is not yet finished at this point, the not still has to be tested. Again a new parser is launched, trying to parse Subs, now with current term "1+2=3" (Figure 15):

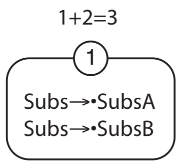

Figure 15. New parser for start symbol subs, launched to check applicability of Subs on the term "1+2=3".

The parser tries to apply SubsA on the term, but find that it fails. Next, SubsB also fails. There are no other rewrite rules that could forward the parser at this point.
All options are exhausted and no shift possibility for Subs was found. Therefore the conclusion is that Subs is not applicable. Hence the not(Subs) *succeeds* and a shift possibility is created for not(Subs) in the main parser resulting in Figure 16:

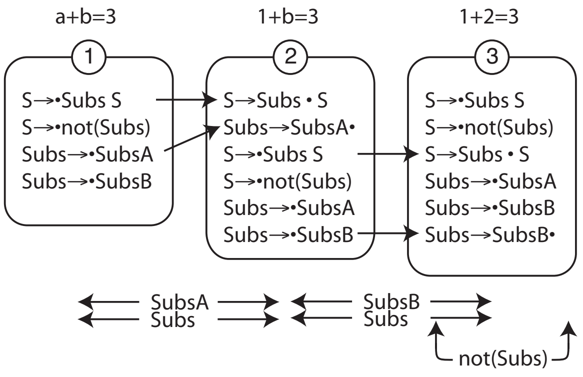

Figure 16. not(Subs) has been recognised and a shiftpossibility for it was added.

The scanner now recognises an S, and creates a shift possiblity <S,3,3>. This results in another shift possibility <S,2,3>, resulting in yet another shift possibility <S,1,3>, completing the parse as in Figure 17:

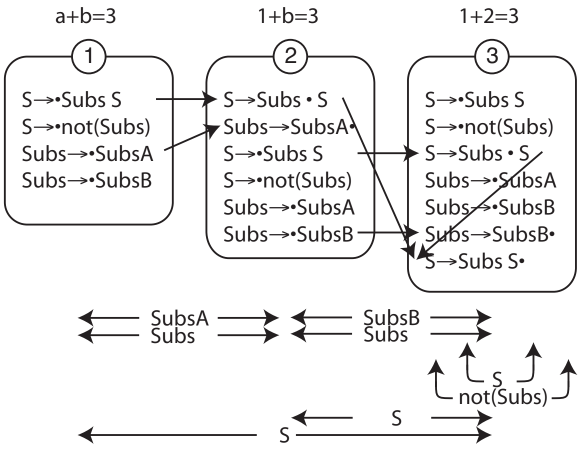

Figure 17. S has been parsed succesfully.

3: Installation

This chapter discusses system requirements and how the parser can be installed.

System Requirements

The parser has been run succesfully on Mathematica 4 to Mathematica 6. Mathematica 6 currently is supported for a range of operating systems, as shown in Table 1:

Table 1. Operating systems under which Mathematica 6 can be run (at 27 august 2007).

Operating System	32/64 bit
Windows Vista, Windows XP, Windows Server 2003	32 and 64
Windows Compute Cluster Server 2003	64
Windows 2000, ME	32
Mac OSX 10.3 Intel	32 and 64
Mac OSX 10.3.9, 10.4 PPC	32
Linux 2.4 or later	32 and 64

The parser was developed on Mathematica 6 on Mac OSX 10.4, and was load and run succesfully in Mathematica 6 on Debian "Edge" on Linux 2.6, and in Mathematica 4 on Mac OSX 10.3.
Mathematica 4, 5 and 6 have minor changes in several system functions. Therefore, the code in the parser in some cases has to call separate functions depending on the actual version of Mathematica. This is done via the IfVersion[version4code, version6code] function. We have not tested the parser on Mathematica 5, but from tests with other Mathematica applications it is expected that the version 4 code, which is currently selected also under Mathematica 5, will work.
There is a minor issue when loading the Mathematica 6 modules (.m files) in Mathematica 4: Mathematica 6 adds a few $CellContext`Code fragments that have to be deleted manually before Mathematica 4 can open the files.
Parsing more serious grammars can take a lot of memory and CPU power, hence a fast modern machine is recommended. For example, the row reduce strategy alone can take 15 seconds just to create a new parser object, using Mathematica 4 on a 1GHz Powerbook G4, while it takes less than a second in Mathematica 6 on a Macbook Pro 2.33 GHz Intel Core 2 Duo.

Installing the Parser

The current version of the parser consists of three files: parstratparser.m, code.m and unification.m. These are files automatically compiled by Mathematica from three files with the same name but with the extension ".nb".
To use the parser, the .m files have to be placed in the proper directory which we will abbreviate with TDIR. TDIR is dependent on the operating system as shown in Table 2:

Table 2. Different operating systems and required target directory TDIR for the parser modules.

OS	TDIR
Mac OSX	`$HOME/Library/Mathematica/Applications`
Linux	`$HOME/.Mathematica/Applications`
Windows 98/Me	`C:\Windows\Application Data\Mathematica\Applications`
Windows NT	`C:\WINNT\Profiles\username\ Application Data\Mathematica\Applications`
Windows 2000/XP	`C:\Documents and Settings\username\ Application Data\Mathematica\Applications`

In general the directory is $UserBaseDirectory/Applications, and $UserBaseDirectory is a Mathematica variable pointing to the system-dependent directory where Mathematica expects the files.
Just drag all .m files into the TDIR directory, or use a shell command like

```
cd <directory where package was downloaded>
cp *.m $TDIR
```

If you want to use symbolic links instead of a copy: only *symbolic* links (made with the -s option in ln) work properly. In OSX, it is possible to create another type of links called 'alias' using option+apple+drag, but these links will not work for Mathematica.

4: Strategy Specification Objects

The strategy specification objects are the objects that are used to specify the strategy. A strategy specification consists of a set of rules. Every rule can be either a grammar rule or a rewrite rule.
A rewrite rule can manipulate the problem term, using (Mathematica) code to compute the term after manipulation if necessary.
A grammar rule does not manipulate the problem term, but it determines the order in which rewrite rules can be applied. To do this, there are a number of options:

- multiple grammar rules with the same attributed name can be used to indicate alternative strategies ("OR")
- Each grammar rule contains a sequence of attributed names, referring to other grammar rules and rewrite rules that further detail on each step in the sequence
- A "not" operator to test whether a rule can be applied anyway (without actually applying it)
- A "parallel" operator that runs two strategies in parallel.

These objects and their semantics were already discussed in the User Manual [Pasman07]. The following sections discuss the functions related to these objects in detail.

Attributes

In the context of parsing and grammar definitions, attributes refer to variables that are used inside rules. Attribute values can be passed to substrategies via the attributes in the attributed name of the rule (see next section). Attribute values can be passed both upwards and downwards. Downwards, also called inherited attributes, is when the attribute is known in the top level strategy and passed down to a substrategy. Upwards, also called synthesized attributes, is when the attribute value is calculated in a bottom level strategy and its value is passed upwards.
The attributes can be any mathematica variable starting with a $ followed by a lower case character. For example, `$x`, `$someName`. As far as mathematica is concerned, these are just standard variables, just as ones starting not with $. The purpose of the $ is twofold: (1) avoid any conflicts with usual variable names (that do not start with $) and (2) make it easy to determine which variables are used.
Standard Mathematica already provides a number of variables starting with $, such as $RecursionLimit, $SystemID, $MaxNumber, etc. All these have an upper case character after the $, and will never conflict with the attributes.
The parser introduces a special variable $CurrentTerm. It corresponds to the current term at hand. Due to lack of time, currently $CurrentTerm only is set within Code blocks but it would be better if it were set always.
Generally, attribute values are local to a rule, so each rule has its own copy of attributes. More accurately, every item within the parser has its own copy of the attribute values. The only exception where attributes are shared is within parallel constructs.

Attribute Value List

An attribute value list is a plain Mathematica list of `Equal[attribute,mathematica value]` objects and pretty prints as attribute==value. For example,

```
{$x==5, $y==LZPlus($x,7),$pos=={2}}
```

is an attribute value list. We use "Equal" (==) instead of "Set" (=) because "Set" will immediately be evaluated by Mathematica while Equal is not evaluated.

Attributed Name

Attributed names are objects used to refer to rules. They are stored with a **gterm** object with the name and a number of attributes as parameters. The first argument is normally a string: the name of the rule.
In this parser, any string is acceptable for a name, but it is recommended to stick with standard (upper and lower case) roman characters and numbers to avoid confusion and issues while pretty printing.
The parser is set up to recognise names starting with a lower case character as rewrite rules, and names starting with an upper case character as a grammar rule.
Table 3 shows a few examples of attributed names. On the right are the prettyprinted versions of these attributed names (see section Pretty Printing, p. 27).

Table 3. Few example attributed names. On the right are the prettyprinted versions of these rules

non pretty printed attributed name/gterm	pretty printed
`gterm["ruleexchrows",1,2]`	`ruleexchrows[1,2]`
`gterm["Strat25"]`	`Strat25`
`gterm["ruledeletevariable ",$x]`	`ruledeletevariable[$x]`
`gterm["RepExh",$s]`	`RepExh[$s]`
`gterm["aap",gterm["beer",$x],$pos]`	`aap[beer[$x],$pos]`

Code Block

A Code block is an object holding a fragment of raw Mathematica code. Table 4 shows a few example Code blocks. Formally, a code block is an object Code with a Mathematica CompoundExpression as its argument. Code blocks were already discussed in detail in the User Manual [Pasman07], so what follows is mainly a copy from there.
The Code block functions are collected in the "Code.nb" module.

Table 4. Some example Code blocks. The third example is showing the RuleAtPos embedding the Code block, to make clear that the pattern variables "mat" and "vars" used in the Code block get their values from the term at hand.

```
Code[If[$start > $end, $Failure, $var = $start]]
```
```
Code[Clear[$var]; $startp1 = $start + 1]
```
```
RuleAtPos[AugmentedMatrix[mat_LZMatrix,vars_],{},
   Code[If[$k === 0 , $Failure,
     $newmat = mat; $newmat[[$row]] = $k  mat[[$row]];
     AugmentedMatrix[$newmat, vars]]]]
```

Only attribute variables should be used as a computational variable inside a code block. Normal Mathematica variables (e.g. "k", as opposed to $k) should be avoided. The reason is that any value given to normal Mathematica variables would be stored

in the global scope, and appear to all other evaluations at once. Proper scope restriction to the rule at hand is provided only for the attribute variables.
If there is a problem evaluating the code contents by Mathematica, this will be caught. An error message will be printed and the evaluation continues as if the Code returned $Failure. For example, evaluating `Code[3 = 4]` (using `CodeEval[{}, Code[3 = 4]]`) gives

```
Set::setraw  : Cannot assign to raw object 3. >>
Problem with strategy. Code block is incorrect. Intercepted
   and returning $Failure instead. Code block: Code[3 = 4]
```

Code blocks have a number of important properties:

- A Code block has attribute HoldAll, so nothing inside a Code block is evaluated until it is given to the CodeEval function.
- A Code block that returns $Failure causes all associated parser attempts to fail.
- Inside a Code block one can refer to $CurrentTerm, to determine the actual term at that point.
- An attribute variable can be cleared using Mathematica's Clear[].

Code Evaluation

Evaluation of a Code block is done with `CodeEval[attrivalues, Code block]`. attrivalues is an Attribute Value List. CodeEval returns a pair {result, new attribute settings}.
Although a Code block looks pretty straightforward, the technical implementation is surprisingly tricky and may cause subtile problems.
First, all attribute variables occuring inside the Code block are collected, and shielded from the general scope by using a Module call. Inside this shielded scope, the attribute variables are set to their initial values. Then, Mathematica is asked to evaluate the program inside the Code block. Control is passed directly to Mathematica, and anything allowed by Mathematica can be done here. Next, the return value as well as the new values of the attribute variables are collected. The Code block is analysed, to determine which attribute variables occured on the left side of an equation or inside a Clear. Only for these variables, the new values are saved, the other values are discarded. While saving the values, it is checked for each variable if it actually still has a value (it may have been Cleared), and if it was cleared the variable is removed from the list of known variables instead of saved.
One known issue with this algorithm becomes clear in the following example with the start condition $g=f[$x] and $k<0:

```
Code[If[$k<0,$x=6,$g=3]]
```

In this code block, $g *might* be changed (because $g is at the left of an assignment, and we do no analysis of If-blocks), and therefore the value of $g is re-computed after evaluation of this code block. In this case $x=6 after evaluation, and therefore $g=f[6] after evaluation of this code block, *even though $g was never modified in the code block itself*. In this case, a Clear[$x] in a subsequent Code block will NOT return $g=f[$x] but leave us with $g=f[6].
A nice example where Clear is needed is the "For" strategy (Figure 18).

```
For[$var,$start,$end,$strategy] := Code[If[$start <= $end, $Failure]]
For[$var,$start,$end,$strategy]:=
    Code[If[$start > $end, $Failure, $var = $start]]
    $strategy
    Code[Clear[$var]; $startp1=$start + 1] For[$var, $startp1, $end,
    $strategy]
```

Figure 18. For strategy, illustrating the need for Clear in some cases. $var needs to be set in order to evaluate the $strategy. But if $var were not cleared before the next step of the For loop, and for instance be set to 0, $var=$start would evaluate as 0=$start which would fail.

EstimateAffectedVars

As discussed in Code Block, we need to estimate affected vars in code block, because for instance if $y = f[$x] we do NOT want to change $y if $x is set but $y is not touched, (maybe not even mentioned) in the code. EstimateAffectedVars returns a list of all attributes that may be changed by the Code block. It searches all attributes that are either in the left hand of an assignment (e.g.,$x in `$x=17`), or within a Clear[] call. Table 5 gives a few examples of EstimateAffectedVars.

Table 5. Three examples of EstimateAffectedVars. First line of each example gives the Mathematica call, the second line gives the resulting list of estimated affected variables.

```
EstimateAffectedVars[Code[$x = 3; $y = f[$x] + $w; Clear[$z]; Clear[$a,
$b]]]
 result: {$a, $b, $x, $y, $z}
EstimateAffectedVars[Code[$newmat[[2]] = 3]]
 result: {$newmat}
EstimateAffectedVars[Code[$y = f[$x]]]
 result: {$y}
```

GetVars

GetVars is a support function to extract from an arbitrary Mathematica term all the leaf nodes that look like attributes. "look like attribute" here means that, when converted to String, the leaf starts with $ followed by an arbitrary character.

Rewrite Rule

A rewrite rule is a Mathematica object / data structure that looks as follows:

```
RewriteRule[attributedname,RuleAtPos[mathpattern, pos,result]
```

Attributedname is a gterm as discussed in the section Attributed Name. RuleAtPos is another datastructure. Details about rewrite rules can be found in the user manual [Pasman07]. In this manual some functions related to rewrite rules are discussed.

RuleAtPos

RuleAtPos is a data structure looking as

```
RuleAtPos[mathpattern, pos,result]
```

It holds the actual rewrite rule in the form of a mathpattern that indicates the pattern in the term before rewriting, a position in the term where to apply the rule, and a result that is either a mathematica term or a Code block[1].

[1] This is one of the places where we chose to use the Mathematica implementation straight away. This causes the strategy specification to be system (Mathematica)

Note that the position only refers to the top level position. The mathpattern may still contain ambiguities concerning the positions of subterms. Figure 19 uses Mathematica's ReplaceList to show that Mathematica rewrite rules may be ambiguous on more places than just the position of the top of the rule.

```
ReplaceList[aap[1, 2, 3, 4], aap[x__, y__] -> aap[X[x], Y[y]]]
{aap[X[1], Y[2, 3, 4]], aap[X[1, 2], Y[3, 4]], aap[X[1, 2, 3], Y[4]]}
```

Figure 19. ReplaceList example showing that the rewrite rule `aap[x__, y__] -> aap[X[x], Y[y]]` can be applied in many – in this case, three – ways.

Currently, when such an ambiguity is present in the rewrite rule, there is no way to specify which of the rewrites is needed. Mathematica's Replace function is used, which uses the first replace that is possible. Citing the manual for Patterns and Transformation Rules:

> "The case that is tried first takes all the __ and ___ to stand for sequences of minimum length, except the last one, which stands for "the rest" of the arguments. When `x_: v` or `x_.` are present, the case that is tried first is the one in which none of them correspond to omitted arguments. Cases in which later arguments are dropped are tried next. The order in which the different cases are tried can be changed using Shortest and Longest."

ApplyRewriteRule

`ApplyRewriteRule[ruleatpos,attrivalues,term]` is a function that takes a RuleAtPos [mathpattern, pos, result], a list of current attributes with their values, and a term to be rewritten. attrivalues is a attribute value list and term the user's current problem term.
It checks whether the term at the mathpattern actually matches the given subterm of term at position pos (ignoring the attributes). If not, ApplyRewriteRule returns $Failure.
If it does, the subterm is replaced with result. Next, it is checked whether the replaced subterm is a Code block. If so, the Code block is evaluated, using the given attribute values The values of the attributes after evaluation of the code block are ignored.

AllRewritesForInstRule

`AllRewritesForInstRule[RuleAtPos[lhs,pos,rhs],attrivalues,term]` is a variant of ApplyRewriteRule that can handle non-instantiated $pos attribute in the RewriteRule. attrivalues is an attribute-value list. If the position value in RuleAtPos is an attribute ($var), the lhs of the RuleAtPos is matched against the term, and all matching positions are collected. The RuleAtPos is then applied on all these positions. If the given position is already instantiated the rule is applied on that single position.
The results of all possible positions are checked, and only those that were not $Failure are kept. A list of pairs {new attrivalues, result} is returned, where new attrivalues is a attribute-value list for that result. In the new attrivalue-list, the $pos attribute will be set according to the particular choice that was made for that case. Figure 20 shows an example.

---

dependent. We do not yet have enough experience with rewriting for strategy specification to make a better choice.

```
AllRewritesForInstRule[
RuleAtPos[aap[3], $posi, kat[3]], {$m == achttien},
beer[aap[4], aap[5], aap[3], aap[6], aap[3], aap[4]]]
```

```
{{{$m == achttien, $posi == {3}},
    beer[aap[4], aap[5], kat[3], aap[6], aap[3], aap[4]]},
{{$m == achttien, $posi == {5}},
  beer[aap[4], aap[5], aap[3], aap[6], kat[3], aap[4]]}}
```

Figure 20. Input for AllRewritesForInstRule, and result.

AllRewritesForRule

`AllRewritesForRule[RewriteRule[lhs,pos,rhs],neededhead,term]` is a light variant of AllRewritesForInstRule. It checks whether the left hand side lhs actually matches the given attributed name neededhead, by checking if neededhead and lhs can be matched with UnifyInherit. If it does not match it returns $Failure straight away. If it matches, it uses the variable assignments according to the UnifyInherit to call AllRewritesForInstRule. It finally returns a list of pairs, each pair of the form `{instantiated name,newterm}` where instantiated name is an attributed name with $pos instantiated as necessary, and newterm the result of applying the rule at that $pos. If the rule is not applicable at all, the empty list {} is returned.

```
AllRewritesForRule[
 RewriteRule[gterm["rulen", $subterm],
RuleAtPos[beer[x__], $pos, kat[x]]],
gterm["rulen", 3], beer[beer[kaas, aap, noot, mies]]]
```

```
{{gterm["rulen", {1}], beer[kat[kaas, aap, noot, mies]]},
{gterm["rulen", {}], kat[beer[kaas, aap, noot, mies]]}}
```

Figure 21. Example of application of AllRewritesForRule and result.

AllRewrites

AllRewrites[rules,neededhead,term] finds the matching rewrite rule from the rules list. rules is a list of RewriteRule[...] objects. neededhead is an attributed name, instantiated as needed. it returns the first rule in rules for which AllRewritesForRule succeeds, or else $Failure.
It returns only the first rule, because currently AllRewrites is only used for rewrite rules and not for grammar rules. Figure 22 shows an example.

```
AllRewrites[
{RewriteRule[gterm["rulebeer"],
RuleAtPos[beer[x__], $pos, kat[x]]],
RewriteRule[gterm["ruleaap"],
   RuleAtPos[aap[x__], $pos,
    Code[If[Length[$pos] > 2, $Failure, kat[$pos]]]]]},
gterm["ruleaap"],
 aap[beer[kaas, aap[4], noot, aap[aap[2]]]]]
```

```
{{gterm["ruleaap"],aap[beer[kaas, kat[{1, 2}], noot, aap[aap[2]]]]},
{gterm["ruleaap"], aap[beer[kaas, aap[4], noot, kat[{1, 4}]]]},
{gterm["ruleaap"], kat[{}]}}
```

Figure 22. Example of AllRewrites. Note that the rewrite rule fails if Length[$pos] >2, thus if the beer term is more than two levels below the top. rule "aap" is requested so the rule for beer is ignored. The aap[2] term is too low hence is not rewritten, but aap at higher level is rewritten.

Grammar Rules

Grammar rules are represented with the object

```
    GrammarRule[name,listofgrammaritems]
```

name is the attributed name of the grammar rule, and listofgrammaritems is a Mathematica list (ordered sequence) of grammar items. The listofgrammaritems represents the right hand side of a grammar rule.

Grammar Item

A **grammar item** is an object as it can appear in a grammar rule. It can be a number of data types:

1. An attributed name, which then refers to the name of a grammar or rewrite rule
2. **not**[attributed name], where attributed name again refers to a name of a rule
3. **par**[name1,name2] where name1 and name2 are attributed names referring to a rule
4. **Code**[raw Mathematica code]

Table 6 shows a number of examples and their pretty printed form:

Table 6. Some example grammar rules and their pretty printed form.

Grammar Rule	pretty printed form
`GrammarRule[gterm["Strat25"], {gterm["Strat25d"]}]`	`Strat25 := Strat25d`
`GrammarRule[gterm["S"], {par[gterm["A", $x], gterm["B", $x]]}]`	`S:=(A[$x]//B[$x])`
`GrammarRule[gterm["RepExh", $s], {$s, gterm["RepExh", $s]}]`	`RepExh[$s] := $s RepExh[$s]`
`GrammarRule[ gterm["For", $var, $start, $end,$strategy], {Code[If[$start > $end, $Failure, $var = $start]], $strategy, Code[Clear[$var]; $startp1 = $start + 1], gterm["For", $var, $startp1, $end, $strategy]}]`	`For[$var, $start, $end, $strategy] := Code[ If[$start > $end, $Failure, $var =$start]] $strategy Code[ Clear[$var]; $startp1 = $start + 1] For[$var, $startp1, $end, $strategy]`

Pretty Printing

Pretty printing of most objects is pretty straightforward, best consult the Mathematica code for PToString in the pretty printing code section for the details. Here only a few interesting details are discussed.

- attributed names, that is gterm["name",attributes] are prettyprinted as name[attributes]. If there are no attributes, it pretty prints as "name".
- not[X] is pretty printed as ~X
- a list can be pretty printed with an item separator (a string) of choice, using PToString[list,separator].
- par[X,Y] is pretty printed as X//Y
- the parse state items and their pretty printing are discussed in the next section.

5. Unification

A basic action during parsing is unification of attributed names – gterms. Every time the parser shifts the dot and encounters a new item, it has to search through the grammar rules or rewrite rules to find a grammar rule with an attributed name that matches the new item. This is more than just matching, because both the grammar rule and the item may contain attributes.

General unification [Baader01] takes two terms, each with variables, and instantiates the variables in both terms in a minimal way such that the two terms are identical. The algorithm of Baader is implemented as `Unify1[term1,term2]`. In the terms, the variables have to be indicated explicitly, by wrapping them in an Att function. Unify1 returns a list of pairs {variable, value} pairs.

The Att label around each attribute is to easen further processing. Unification only looks at names of attributes, it does not know whether a certain attribute occured in term1 or term2. This poses problems in the parser. Therefore several derived functions are available for unification, that internally use the Att label to rename/wrap the actual variables enabling distinction betweel term1 and term2.

General unification can also be done with `Unify[term1,term2]`. This function just wraps all $-vars (attributes) in an Att function and calls Unify1. Table 7 shows a few examples.

In line with Baader's algorithm, there is no unification support for the BlankSequence (__) and the BlankNullSequence(___), but only plain ($var) attributes.

Table 7. A few unification examples.

unify call	result
Unify[f[$x, $x], f[3, 3]]	{{Att[$x], 3}}
Unify[f[$x, $x], f[3, 5]]	$Failure
Unify[f[3, $y], f[$h, f[3]]]	{{Att[$h], 3}, {Att[$y], f[3]}}
Unify[h[6, $y], h[$x, $y]]	{{Att[$x], 6}}
Unify[f[$x, f[$x]], f[$x, $z]]	{{Att[$z], f[Att[$x]]}}

Unification of Attributed Names

The basic idea is to use `Unify1[highlevel,lowerlevel]` where highlevel and lowerlevel are attributed names. However, unification can not be used straight away because of two reasons:

1. The parser has completely separate context variables for both attributed names, while the Unify1 function may return any collection of substitutions with attributes from both terms mixed up in any way.
2. Unification of two attributed names can not possibly be a one-run process, because synthesized variables will be available only after the called rule has been fully applied.

To work around these issues, a two-step unification is done, as follows.

First, a preliminary, potentially too general unification is done with UnifyInherit. This brings the values from the 'higher level' term1 to the 'lower level' term2. Then, the rule at lower level is evaluated, potentially changing the attribute values. Finally, the new attribute values are then re-unified, using UnifySynthesize, with the higher level term1, bringing the new attribute values back to the high level.

UnifyInherit

UnifyInherit creates a set of lower-level attribute assignments, using only those lower level attributes that get explicitly assigned after unification. All assignments involving more complex patterns to lower level attributes are simply ignored. This results potentially in a search with under-specified variables, giving too many results. This might cause problems, especially when testing for not. But so far, there seem to be no major issues with this approach.
`UnifyInherit[highlevel,lowerlevel]` determines the variable settings at *lowerlevel* tuch that highlevel is matching, or $Failure is such a match is not possible. Inherit refers to the 'passing down' of attribute values, which corresponds to inherited attributes in attribute grammar parsing.

UnifySynthesize

`UnifySynthesize[highlevel,lowerlevel]` is used after a rule completed, to bring the new attribute values back to the high level. It determines the variable settings at *highlevel* tuch that lowerlevel is matching, or $Failure is such a match is not possible. Synthesize refers to the 'passing up' of attribute values, which corresponds to synthesized attributes in attribute grammar parsing.
As with UnifyInherit, the Unify1 function is used for the elementary unification of the two levels. This time, dependencies of higher level on lower level attributes have to be removed entirely from the final result, which is done with the `TryRemoveRHS` function.

6: Strategy Parser Objects

This chapter discusses the technical details inside the parser, the involved data structures and functions. A global overview is given to sketch the overall process. Then the parser state and the three key components – the shift possibility, item set and subparser – are discussed.

Global Overview

This section aims at giving a high-level overview of the internals of the strategy parser.
The core of the parser is an Earley style chart parser [Earley70] and builds further on to earlier work [Pasman90]. The actions/rewrite rules are represented by the 'terminals' of the grammar. The strategy specification is represented by the nonterminals.
Parsing is started by creating an initial parse state. Next, the parse can be extended with the next student's action leading to a new parse state[2]. We use the term **parse state** to refer to the entire collection of itemsets after a given sequence if actions has been parsed. As with Earley's parser, we have an itemset representing the parse situation after each action that was done. An itemset is a set with items: an **item** is a grammar rule with a dot somewhere in it and with some links to predecessor and successor items.
More accurately, there are a number of types of 'items' that can be member of an itemset:

```
1. item[tag,unifiedhead,left,right,vars,pred,succ]
2. subparser(tag,par(X,Y),{sharedvars},{parserpair }]
3. pitem(setnr,tag)
```

Note that one of these items is called 'item', it would better have been called 'dottedrule'.
There are a number of major additions in this parser, compared to [Pasman90].
The first addition is that the student's actions act on a **problem term**. The problem term thus changes with each action, so we speak of the **current** problem term to refer to problem term for the itemset holding some item under discussion. An action can be parsed only when both the strategy specification generates that action as the next step *and* that particular action is actually possible on the term at hand. The scanner currently takes both the action and the resulting problem term as input, it does *not* check whether the action really gives that result. The only change to the parser for this modification is that the problem term is added to each itemset.
The second addition is the introduction of a not(Strat) primitive to the context free grammar. This primitive can be parsed only if the Strat (non)terminal can not be applied to the current problem term. To implement this, the given strategy is basically fed with every possible action sequence to see whether one of them fits both strategy and problem term. If one fits, the not fails, and if none of them fits, the not is accepted.

[2] Earley's paper refers to the parser state as 'set of states'. We use 'itemset' to refer to Earley's 'state set'. Earley also uses 'state' to refer to our items.

Third, attributes were added to the strategy specification. To do this, all (non)terminals are replaced by attributed names, and internally all items get a set of local attribute/value pairs.
Fourth, Code blocks were added, enabling arbitrary computations. These can be used both in rewrite rules, to calculate the result term of the rewrite, and within strategy rules, to compute attribute values.
The final addition is the introduction of a parallel(Strat1,Strat2) operator, that creates all permutations of the actions from strategies Strat1 and Strat2. To handle parallel operator, a special item called **subparser** is available. This subparser tries to distribute the actions over the two strategies, such that all actions can be accounted for. Because a subparser can account for multiple actions, it would be logical to have it distributed over itemsets just as other items. But for practical reasons, the subparser object is kept in the first itemset where it was needed, and subsequent itemsets refer back to the original with an pitem object.
For efficiency, the parser also keeps track of the **shift possibilities** that were created during the parse. Shift possibilities are created when a grammar rule has been 'completed': all its steps have been recognised and its attributed name ('nonterminal') can be parsed. The shift possibility then enables all items that were waiting for that name to continue.

Parser State

The parser state is the central object around which parsing revolves. Each parser state is an object containing all the details about a current parser process. It looks like

```
PS[Strat,Itemsets,shiftpos,rules,terms,startsym]
```

- Strat is a set (Mathematica list) of grammar rules, and rules (another Mathematica list) are the rewrite rules. These two together form the strategy specification.
- Itemsets is a set (Mathematica list) of itemset objects.
- Terms is a set of problem terms, for each itemset: {term1,term2,...}.
- startsym is the attributed name that the parser used as the start term.

There are a number of objects that are typically part of the Parser State, such as the shift possibility, item, itemset and subparser, a number of parsing support functions that further develop the parse state, and a number of functions to evaluate a parser state. These are the subject of the following three sub-sections. These might have been subsections of this section, but to avoid too deep section levels and cyclic dependencies (see Figure 1) they are presented as plain subsections.

Shift Possibility

A shift possibility comes in two varieties: shiftpos and varchange.

ShiftPos

The basic shift possibility looks like

```
shiftpos[gterm,leftitemset,rightitemset, creators]
```

It reflects the occurance a chain of linked items from a left-complete to a right-complete item. gterm is the attributed name of the grammar rule of the chain, instantiated with the final attribute values of the right-complete item. leftitemset and

rightitemset are the set numbers holding the left-complete and right-complete item, and creators is a list of itempointers to right-complete items that created this shift possibility. The creators list is mostly for convenience and speed when finding a parse tree from a parse state (see chapter 9. Parse Tree Generation).
When the shiftpos is created by a completed subparser (p.34), the creators list contains pitemlink objects.

pitemlink

`pitemlink[pitem[setnr, tag], eatpattern]` is a pointer to a subparser. It is used in shiftpos, to indicate which pitem created a shift. The pitem points to a subparser via a standard pitem pointer, and the eatpattern is indicating which of the subparsers is the actual parser that created the link, and is explained in the section ParserPair, p.35.
Note that, as parsing continues, the subparser eat pattern may be extended and its status may have changed from finished into anything. But any subparser having the same start pattern should have the same results, for instance if the eat pattern is "10", both parsers 100 and 101 should have the same state after two input symbols.

VarChange

A varchange is a variant of the shift possibility. Its form is

```
    varchange[newvars, left, right,creators]
```

It represents a special shift where "all items" in the left can move on to the right itemset *without* shifting the dot, but with making a few changes to (shared) variables. This occurs in subparsers, and represents the situation where the other half of a parser pair did a parse step and changed some global variables in that step.
A varchange pretty-prints as <newvars, left,right,creators>
Actually not "all items" but only the items that satisfy the CanInterruptHere[item] criterion can actually perform this varchange shift.

CanInterruptHere

`CanInterruptHere[item[tag,name,left,right,vars,pred,suc]]` is a function that determines at which places a parser pair can switch between the left and right parser. Currently the switch can be made always except when there is a code block directly right of the dot. So switching is also possible when the dot is entirely on the right side (right=={}).

Item Set

An itemset is a set (represented with a plain Mathematica list) of parser items. There is an initial itemset and one after each next rewrite action.

Parser Item

A **parser item** is an object as it can appear in an item set. It can be a number of objects:

1. **item**[...], a DottedRule item
2. **subparser**[...], a special construct to handle parallel objects par[...] in the grammar.
3. **pitem**[...], a pointer to a subparser object

DottedRule Item

The item object is what maybe should have been named dottedrule. A dotted rule is a grammar rule with a dot in it. The dot then represents how far the parser proceeded through that rule. It looks like

```
item[tag,head,left,right,attris,pred,succ]
```

where tag is a unique tag 'item$nnn' where nnn is a number, head is the attributed name of the rule that the item reflect, left and right are the grammar items (see p.27) before and after the dot, attris is an attribute value list, and pred and succ are lists of itemptr objects pointing to the direct predecessor and successor of the item. An item gets an immediate successor if its dot is in front of an attributed name and there is a shiftpossibility for that. The predecessor list is the inverse relation of successor.
A dottedrule item is called **right-complete** if it has the dot on the right, i.e. right={}.

Itempointer

An **itempointer** has the form `itemptr[setnr,tag]`. setnr is the set number and tag is the unique item$nnn tag of the item being pointed to. An itempointer always points to items in the current parser state, not across parsers to for instance an item in another pair of a parserpair.

getitem

getitem is a support function to retrieve an item given an itempointer. The call is `getitem[PS, it]` where PS is the parserstate and it the itempointer. It returns a copy of the item as it appears in the parserstate.

Subparser

A subparser is a contains a set of parser pairs. Because a subparser is a complex construct, recursively using parse states, we discuss the subparser in more detail in the next section.

Subparser

A subparser is an object in an itemset that handles the parsing of two parallel strategies as specified by the par[...] grammar item (p.27). The object stores a large number of parserpairs, exhaustively representing all different ways of distributing the input symbols/actions over the two parallel strategies. The object looks as

```
subparser(tag,par(X,Y),sharedvarslist,parserpairlist]
```

where tag is some unique label in the style `subp$nnn` where n some number, X and Y are attributed names, sharedvarslist is a list of attributes ($vars) and parserpairlist is a list of parserpair objects. As was discussed in the user manual, the sharedvars currently are those un-instantiated $vars that were common to both X and Y in the par(X,Y) call.
In worst case, the size of a subparser **doubles** with each action. This means exponential space (memory/swap space) requirements which can cause serious parsing problems. In practical grammars with limited ambiguity we hope that this is not really an issue.

The tag is to enable easy lookup avoiding unification. Note that the sharedvarslist contains only the shared variables, and NOT their values. The values are stored with each item in each parserpair.
The subparser object is always placed in the itemset in which the corresponding item/dotted rule with the dot just before the par[...] occured. If the subparser is succesful in eating the next symbol/action, a subparser pointer is placed in the new itemset, pointing back to the subparser.

Subparser pointer

A Subparser pointer is represented as **pitem**[setnr,tag][3]. This pointer works just as the itempointer object, except that it points to a subparser and not to an item object.

ParserPair

A parserpair represents the state of two parallel parsers, given a certain distribution pattern of the input symbols/actions over the two parsers: the **eat pattern**. The object looks as

```
   parserpair[firstparser,secondparser,eatpattern,status]
```

Firstparser and secondparser are complete parser state objects (p.32).
The eatpattern is a string of '0' and '1' characters. Each character indicates which of the two parsers ate the next input symbol/action: A 0 indicates that the first parser ate the symbol and a 1 that the second parser ate the symbol.
The status is one of three values depending on the status of the two parsers (see Statuson p.43).

- finished, if both parsers are finished
- ontrack, if one of the two parsers is ontrack and the other ontrack or finished
- dead, if one of the two parsers or both are dead

Pretty Printing

The entire parser state can be pretty printed as well, with the same PToString predicate used to pretty print a strategy. PToString is returning a string so it can be used for other purposes besides pretty printing.

- A parser state pretty prints as Head:S followed by a prettyprint of all itemsets. Shift possibilities are grouped with the itemset that the shift possibility starts at.
- An itemset first prints the normal items, then the shift possibilities, then the varchanges, and finally the subparsers.
- An itempointer is prettyprinted as tag@setnumber where tag is the unique tag associated with the item.
- A shift possibility is pretty printed as <symbol,L,R,creators>. The symbol is the instantiated head, L and R are item set numbers, and creators is a list of itempointers.
- A varchange is printed as <newvars,L,R,creators>, similar to a shift possibility but recognizable because newvars is an attribute-value list and therefore has curly brackets (and maybe a number of $x==y pairs)

[3] The name pitem is confusing. Feel free to coin a better name and modify the code!

- An item is shown as <tag: dottedrule vars: attrivlist pred: predlist suc:suclist>. The predlist and suclist are list of itempointers, the attrivlist is an attribute-value list, but without the curly brackets. The dottedrule prints as head→left•right where head, left and right are also pretty printed.
- A gterm[name,attributes] prints as name[attributes] with all attributes prettyprinted. If there are no attributes, only name is printed.
- a not[gterm] is shown as ~gterm, where gterm is prettyprinted.
- par[X,Y] is prettyprinted as X//Y, X and Y are both prettyprinted in turn.
- A subparser causes a pretty big mess in the output, as it recursively prints a list of full parser states. It prints "SUBPARSER for head" where head is the head that this subparser is looking for, and then it dumps all parser pairs in the eatlist.
- A parserpair prints as "***Pair with eatpattern xxx. status: yyy" where xxx is the eatpattern of this parserpair and yyy the status (alive, dead, etc). Then it shows "Parser 1" followed by a prettyprint of parser 1 of the pair, followed by "Parser 2" and parser 2 of the pair.
- Grammar rules are shown as discussed before (p.28)

7. Parse State Manipulation

This section describes the available functions to manipulate the parse state. The available functions are calling each other in a highly recursive way, but it is possible to distinguish three groups of functions. The first is the **core functions**, handling the basic context free grammar constructs as usual for Earley-style chart parsers. The second group is the related to checking whether a strategy can be applied: the **applicable functions**. The third group is related to handling the parallel construct, and is called the **parallel functions**.
All these functions do not change the given parse state, they only return a new, changed parse state.

The Core Functions

Scanner

Scanner extends a given parse state with an additional symbol/action and returns the resulting new parse state. It consists of a call to AddItemset followed by calls to Addshift and then a call to Subscanner for all pitems in the last itemset. The call looks like

```
   Scanner[PS, rewritename, newterm]
```

where PS is the parse state to be extended, rewritename is the attributed name of the rewrite rule being applied, and newterm is the new problem term after the rewritename was applied to the old problem term. If the parser has access to the given rewritename, the newterm could be computed instead of passed. However, there may be situations where the applied rewrite rule as specified by the rewritename is not available to the parser (so the parse should fail but not crash), and in such a case the parser can not compute the newterm.

AddItemset

`AddItemset[PS,newterm]` appends (at the right side) a new, empty itemset to the list of itemsets in the given parsestate PS. The associated problem term newterm is added to the terms list of the parsestate. The resulting modified parsestate is returned as a result.

Predictor

Predictor creates items from the grammar rules. The call is

```
   Predictor[PS, head, setnr]
```

PS is the old parse state, head the parseritem to be predicted, and setnr the item set number in which the head occured.
Predictor is called by the other functions whenever a new parseritem was added to an itemset. Predictor then looks in the parseritem, at the the grammar item immediately after the dot and takes actions as follows:

- if parseritem of the form **not**[S]: Test whether S is applicable by calling Applicable (p.39). If it is applicable, create a shift possibility for not[S] using Addshift.
- if parseritem of the form par[S,T]: Test if there is already a subparser for S//T in the last itemset. If not, add a subparser:
  `subparser[tag,par[S,T],sharedvars,{parserpair[p1,p2,"",ontrack]}]`
  where tag is a new unique tag "subp$nnn", sharedvars is determined by taking the intersection of the variables in S and in T (as determined by GetVars) and "" is the empty eat pattern. Also a subparser pointer pointing to this new subparser is added to the itemset.
- if the parseritem is an attributed name N: select all grammar rules that have an attributed name that unifies with N. For each of these rules, create a new parser item with the dot on the left of the grammar rule, and add it using Additem.

Additem

Additem tries to add an item to a set. The call is

```
  Additem[PS,item[tag,head,left,right,vars,pred,suc],
  setnr]
```

where PS is the old parsestate, item is a usual parser item, and setnr is the target item set number. Additem returns a new parserstate in which the item has been added.
Additem first checks whether the same item, but with maybe another tag, is already in that item set.
If it is already there, the existing item is extended with the pred and suc links from the new item. Then, it is checked if the updated parser item has links to a right-complete item using endsets. If so, the Completer is called for each right-complete item.
If it is not yet there, the item is added to the set. Then, it is checked whether the item is right-complete. If it is, the completer is called for the item. If it is not right-complete, it is checked which grammar item is immediately after the dot. If it is a Code block, the code block is evaluated. If the code block does not return $Failure, a new item is created with the dot shifted over the code block, the vars set according to the results of the Code block, and Additem is called to add this item to the itemset as well.

startsets

startsets returns all the left-complete items linked to a given item. The call is

```
   startsets[PS, itemptr]
```

where PS is the parser state, and itemptr is an item pointer.

endsets

Endsets returns all the right-complete items linked to a given item. The call is

```
   endsets[PS, itemptr]
```

where PS is the parser state, and itemptr is an item pointer.

Completer

Completer is called when a parser item is right-complete, indicating that some attributed name has been recognised succesfully. The call is

```
Completer[PS, itemptr[set, tag]]
```

where PS is the old parserstate, and itemptr is a pointer to the right-complete item.
The completer determines the left-complete parser items that are linked to the given right-complete item using startsets. The attributed name of each left-complete item is then instantiated with the final attribute settings according to the right-complete item, and a shift is created for each of these instantiated attributed names using Addshift.

AddShift

Addshift adds a new shift possibility and checks which items can use it. The call is

```
   Addshift[PS,shiftpos[head, left, right, creators]]
```

where PS is the old parserstate, and shiftpos a shift possibility. It returns a new parsestate with the shiftpos added and all consequences handled.
Addshift first checks if the shiftpossibility is already available. If so, the creators list is merged with the creators list of the existing shift possibility.
If the shift possibility is not yet available, it is added. Next, it is checked if any items in the left itemset have the dot just before a grammar item that can be unified with head (using UnifySynthesize1). For each of these grammar items, a new item is added with the dot is shifted one place to the right, using Additem.

UnifySynthesize1

UnifySynthesize1 is a support function that checks whether a parser item can be unified with the head of a shift possibility. The call is

```
   UnifySynthesize1[item, head]
```

If the item is a dottedrule item and it has an attributed name, not[...] or par[...] right of the dot, then UnifySynthesize1 returns the result of unification of the item and head. In all other cases $Failure is returned.

Applicable functions

The applicable functions is a small collection of functions that can check whether a given strategy can be applied to a given problem term. This is needed for instance when the not is encountered (see Predictor, p. 37) or when the status of a parser state has to be determined.

Applicable

Applicable returns a shortest complete rewritesequence on term for start if there is one, and "fail" if not. The call is

```
   Applicable[grammar, rewriterules, problemterm,
   startsym]
```

where grammar and rewriterules describe the strategy, problemterm is the current problem term, and startsym is the start symbol (an attributed name) of the strategy.
The real work is done by the Applicable function. Applicable however only accepts the startsymbol to refer to grammar rules . Therefore, if the startsymbol is referring to a rewrite rule, the grammar is extended with the rule `UnusedGrammarSymbol54::=startsym` and the startsymbol is set to `UnusedGrammarSymbol54`. Note that for this reason, one should not use `UnusedGrammarSymbol54` in a strategy specification.

TryToComplete

TryToComplete determines a sequence of additional actions (**extensions**) to complete a parse. Completing the parse means that the start symbol of the parser has been recognised after doing these additional actions. It returns $Failure if such a sequence does not exist. The full call is

```
TryToComplete[PS]
```

where PS is the parser state to be extended/completed.
In order to find this, TryToComplete first tests if the given parser state is in fact already finished. If so, it returns immediately.
If not, it does a breath-first search of all possible extensions. To do this, it first calls AllBlockedItems to determine the candidates that extend the parse with one step. Each candidate is tested, by applying it to the parser state (using Scanner) and then recursively calling TryToComplete to see if the resulting state can be completed. The first sequence that completes the parse is returned as the result.

AllBlockedItems

AllBlockedItems returns a set (Mathematica List) with all parser items that are waiting for a terminal (a rewrite action). All means: all items in the last itemset plus all items in the last itemsets of still-alive threads of the subparsers in the last itemset.

Parallel functions

The parallel functions is a small collection of functions to support parsing of two parallel strategies. The main function is the SubScanner, which is called from the Scanner function (p.37). It works on the subparser objects.

SubScanner

Subscanner takes care of the scanner job for subparser objects. It extends the parallel parse state within the subparser object with one symbol/action and returns a new parser state with this extended subparser. The call is

```
SubScanner[PS, paritem, rewrite, newterm]
```

where PS is the parser state, paritem a subparser pointer to the subparser being updated, rewrite the attributed name of the rewrite being applied, and newterm the resulting problem term after the rewrite was applied on the current problem term.
First, SubScanner fetches the subparser to be updated (p.34).
Then, each of the parser pairs in the subparser is extended with the new rewrite action, using ParScan.

Next, it is checked whether there are finished parser pairs using FinishedShiftsOfPair, and for each finished pair Addshift is called to add a shift for the paritem.
Finally, it is checked whether there are still alive parser pairs, and if so a pointer to this subparser is added the next itemset, to indicate that the subparser has to be checked when the next action comes in.

ParScan

ParScan extends a parserpair with one rewrite action. The call is

```
ParScan[parserpair[par1, par2, eatpatt, stat],
     rewrite, newterm, sharedattri]
```

where parserpair as on p.35, rewrite the attributed name of the rewrite being applied, newterm the resulting problem term after the rewrite was applied on the current problem term, and sharedattri the list of shared attributes between par1 and par2.
ParScan works straightforward: it calls ScanEF to determine the result parserpair1 when par1 eats the rewrite action, and parserpair2' when par2 eats the rewrite action, and returns a tuple { parserpair1',parserpair2' }.

ScanEF

ScanEF determines what happens when one parser in a parserpair eats a rewrite action. The non-eating parser may also be affected, because of shared attributes. The call is:

```
ScanEF[eatingparser, followingparser,
     rewrite, newterm, sharedvars]
```

First it calls Scanner to extend the eating parser. Then, the shared variables are extracted from the resulting parse state of the eating parser.
The following parser is extended with a new empty itemset and a varchange shift possibility is created for the extracted shared variables to extend the following parser as well.
ScanEF then returns a tuple {neweatingparser,newfollowingparser}.

GetSharedVars

GetSharedVars extracts the shared vars from the last item set of a parser state. Usually there is not one unique set of shared var, and therefore GetSharedVars returns a set (List) of objects **shared**[items,attribute value list]. In each shared object, items is a list of item pointers that have the same attribute values as given in the attribute value list. The call is

```
GetSharedVars[PS,sharedattri]
```

GetSharedVars checks all parser items, and for items that satisfy CanInterruptHere (p.33) the shared attributes are extracted and grouped in the 'shared' objects.
There is currently no code to extract shared variables from subparsers within subparsers. This means that **shared variables in nested subparsers are not properly supported**.

FinishedShiftsOfPair

FinishedShiftsOfPair returns all shift possibilities to the last itemset, made possible by a finished parserpair. The call is

```
   FinishedShiftsOfPair[parserpair[par1,par2,...]]
```

where parserpair is a parserpair object.
A list of instantiated par[S,T] objects are returned, with S and T all possible tuples where S the result of FinishedShifts[par1] and T=FinishedShifts[par2].
Note that FinishedShiftsOfPair can not return startset and endset numbers, as the parserpair does not know at what point the parse started at a higher level.

8. Checking the Parse State

In order to determine the status of the result of scanning student actions, a number of test functions are available. In principle the Status function does it all, but Status can be very expensive. Therefore, a number of cheaper options are also available, doing a partial status check.

Finished

`Finished[parsestate]` checks that the given parsestate contains a shift possibility for the start symbol from the first to the last itemset in the given parse state. It returns True if there is at least one, and False if none exists. In fact it just returns FinishedShifts[..]≠{}.

FinishedShifts

`FinishedShifts[parsestate]`returns all the shift possibilities from the first to the last itemset that can be unified with the start symbol (and {} if no such shift possibility exists).

OpenEnded

`OpenEnded[parserstate]` returns True if there is any item in the last itemset. Note that this is weaker than OnTrack, because the presence of an item in the last itemset does not guarantee that any of these items is going to succeed.

OnTrack

`OnTrack[parserstate]` returns True if there is an extension possible to complete the problem. It is just equivalent to TryToComplete[parserstate]=!=$Failure. TryToComplete is an expensive operation and therefore OnTrack is also expensive.

Status

`Status[parserstate]` returns **finished** if the parserstate is Finished, **ontrack** if the parserstate is OnTrack but not finished, and **dead** otherwise. Status may be expensive, if the OnTrack test is needed.

PairStatus

`PairStatus[par1,par2]` determines the status of a parser pair with parserstates par1 and par2.

- If both parsers are finished, it returns finished
- If both parsers are on track, it returns ontrack.
- Otherwise, it returns dead.

This usually will be an expensive operation.

9. Parse Tree Generation

Parser states are pretty complex objects, containing all ambiguities, dead ends, and potentially many different paths to a solution. In many cases, but particulary when generating a hint for the student (e.g., how to proceed, determining where he is in the strategy hierarchy, etc) only *one* succesful parse would be sufficient. A parse tree is the representation of choice for one such succesful parse.
This section describes how a parse state can be converted to a parse tree, and how hints can be extracted from such a tree. It contains two sections: the parse tree datastructure objects and the parse tree functions.

Tree Datastructure

This section describes the datastructures to build a parse tree: the seqnode and the parnode. Each can hold a list of these nodes. The tree then is simply a nested hierarchy of such nodes. Of the grammar items, only the attributed name and the par constructors are reflected in the tree, while the not and the Code items are ignored.
If a node has no children, the subnodelist is empty ({}). Not all leaf nodes are rewrite rules, some grammar rules also have no children.

Sequence Node

`node[term,subnodelist]` represent a node that has a sequence of children. This reflects the application of a grammar or rewrite rule where the term is the attributed name of the rule, and the subnodelist is a list of nodes, each representing one of the grammar items in the grammar rule (rewrite rules have no children hence the subnodelist is empty in that case).

Parallel Node

`parnode[X//Y,eatpattern,subparsetree1,subparsetree2]`
represent a par[...] grammar item. subparsetree1 and subparsetree2 are the subnodes of the strategies X and Y, and eatpattern indicates in which particular order X and Y ate the subsequent rewrite actions.

Tree Functions

Tree functions support the conversion and manipulation of parse trees.

GetSuggestionTree

`GetSuggestionTree[parsestate]` tries to complete the given parse state using TryToComplete, and to generate a parse tree for the result. It returns a parse tree. It returns $Failure if the parsestate can not be completed.
GetSuggestionTree first calls TryToComplete to get a completed parse state. Then it calls FinishedShifts to pick one (the first) instantiation of the start symbol that succeeded. Finally it calls ParseTreeForShift to compile a parse tree for the chosen instantiated start symbol.

ParseTreeForShift

ParseTreeforShift takes a shift possibility, and returns a parse tree. The call is

```
ParseTreeForShift[PS,
shiftpos[item,left,right,creators]]
```

A few cases are distinguished

- item is an attributed name:
    - o if the creators list is empty: ParseTreeForShift then returns node[item,{}]
    - o not empty: ParseTreeForShift calls FindPathBack to determine a path (a sequence of shift possibilities) from left to right for the first alternative in the creators list. For each of these shifts, ParseTreeForShift is called recursively to generate subtrees for that shift.
- item is a par[...] item. ParseTreeForShift has to dig into the subparsers to find the details. In this case, the creators list contains pitemlink objects which points into a subparser. The subparser is fetched, and the first (any will do) subparser with an eatpattern starting with the pattern specified in the pitemlink is selected. ParseTreeForShift is called to create a parse tree for the two parsers,

MatchingShift

MatchingShift returns a (just one) matching shift for a given shiftpos. "not" is not checked, because that has been done already when a link was created, and MatchingShift assumes that the link is there. The call is

```
MatchingShift[PS, shiftpos[name, left, right,
creators]]
```

The requested shift is unified (with Unify) with each of the shift possibilities in the given parsestate PS. The first shift that unifies is returned. The 'creators' field of the given shiftpos is ignored in this unification. Internally, the creators list is replaced with the suppusedly never used attribute $anythingcreators612 before unification is attempted.

FindPathBack

FindPathBack returns a (one) path from a given item back to the start set. It returns this in the form of an (ordered) list of shift possibilities, or Null if there is no path back to the start set. The shift possibilities only concern the grammar items in the given item, and therefore may need further refinement. The call is

```
FindPathBack[PS,itemptr, startset]
```

where PS is the parser state, itemptr is an itempointer, and startset is the number of the set where the left-complete item should be.
FindPathBack checks if the item is left-complete. If it is, it can return {} straight away.
If it is not left-complete, each of the predecessors of the item is checked recursively with FindPathBack. If none of these items has a path back to the startset, Null is returned.

The first of the predecessors that has a path back to the startset is used as a start for the complete path back.To complete the path, only the last step has to be added. There are a number of possibilities for this last step:

1. The predecessor has a dottedrule with the dot in the same position. This is possible if the parser is in a parser pair and the shift possibility was actually a varchange. In that case, the shift was done by the other parser of the pair, and the path does not need extension.
2. The dot was shifted over a Code grammar item. There is no associated shift possibility for such a shift, and the path does not need extension.
3. The dot was shifted over a not[...] item. There is an associated shift possibility for such a shift but there is no further proof of the 'not' available in the parse tree. Therefore again the path is not extended.
4. The dot was shifted over a regular attributed name or over a par[..] item. In this case, the shift possibility that created this shift is recovered using MatchingShift, using the item instantiated with the values of the attributes as they were in the direct predecessor that we found.

GetTerminalsFromTree

GetTerminalsFromTree extracts the leaves from a parse tree, to form a rewrite sequence that matches that tree. The call is

```
GetTerminalsFromTree[rootnode]
```

where treenode is the root node of the tree.
We can not just traverse the tree depth first, left to right, because the par[..] blocks specify the proper traversal order.
If the given root is sequence node, it must contain a gterm referring to a shift possibility. This gterm is checked. If the name of the gterm refers to a terminal/rewrite rule, then that attributed name is added as the next terminal in the rewrite sequence.
If treenode is a parallel node, the terminals are extracted from both trees in the parshift node, and then these terminals are permutated as specified in the parshift node[4].

VisualizeTree

VisualizeTree shows a 2D tree plot of the parse graph. The call is

```
VisualizeTree[rootnode]
```

First, ConvertTree is called to get the root label and a tree description that is ready for a call to Mathematica's LayeredGraphPlot. Next, LayeredGraphPlot is called with the LayeredTop packing method.
Using Mathematica's LayeredGraphPlot creates a few problems. First, it will not always put the root term on top. Second, the order of the children is not respected, causing mixing up of the children in the graph.

ConvertTree

ConvertTree converts given parse tree into a tree with nodes of the form {nodelabel, {subnodes}}. The call is

[4] I'm not sure whether this works properly with trees with nested parshift nodes.

```
    ConvertTree[treenode]
```

The process is straightforward. First, prettyprinter is used to create a name given the node's attributed name. The name is made unique by appending an unique number. Then,

- for sequence nodes, ConverTree is then called recursively for the children of the node.
- for parallel nodes, two children are created: one for the first and one for the second parser of the parallel node. In this case, the first link is labelled "//" and the second link is labelled with the eating pattern.

References

[Pasman91] Incremental Parsing. Master's thesis, University of Amsterdam, Department of Computer Science, Amsterdam, the Netherlands. Available Internet: http://graphics.tudelft.nl/~wouter/publications/publ.html.

[Pasman07] Strategy Parser User Manual. Technical Report, Delft University of Technology, August.

[Pasman07c] Pasman, W. (2007). CFG Deep Permutation Parser with Not. Technical Report, Delft University of Technology, March.

[Baader01] Baader, F., & Snyder, W. (2001). Unification Theory. In J.A. Robinson and A. Voronkov (Eds.), Handbook of Automated Reasoning, 447–533. Elsevier Science Publishers.

[Earley70] Earley, J. (1970). An efficient context-free parsing algorithm. Comm ACM, 13:2:94-102.

Parallel+Not Strategy Attribute Parser

W.Pasman, may 2007.. .july 2007

```
Needs["code`"];

Needs["unification`"];

SetAttributes[IfVersion, HoldAll];

IfVersion[old_, new_] := If[$VersionNumber < 6, old, new]

MyReplacePart[exp_, new_, n_] :=
  IfVersion[ReplacePart[exp, new, n].ReplacePart[exp, n → new]];
```

Globals and support funcs

```
GRAMMAR = 1; ITEMSETS = 2; SHIFTPOSS = 3; REWRITERULES = 4; TERMS = 5; STARTSYMBOL = 6;
```

Get itemset n. 1 is first set.

```
Assert[True, _];
Assert[False, message_] := Throw["Internal Error: Assert failed. " <> message];

terminal[not[_]] := False;
terminal[t_] := LowerCaseQ[StringTake[t, 1]];

ParserLevel = 1; (*For better prettyprinting,  counts subparser level.*)
DPPrint[x___] := Print["Parser:", ParserLevel, ":", x];
DPLevel[add_] := ParserLevel = ParserLevel + add;
DPError[x___] := Print["Parser SERIOUS ERROR:", x];

(*Debug Hint system printer*)
DHPrint[x___] := Print["Hint gen:", x];

ExtractX[exp_, {}] := exp;
ExtractX[exp_, pos_] := Extract[exp, pos];
ReplacePartX[exp_, new_, {}] := new;
ReplacePartX[exp_, new_, pos_] := ReplacePart[exp, new, pos];
```

■ TaggedUnion

```
TaggedUnion::usage =
  "TaggedUnion[list1,list2,..., TagPosition] gives a list with the union of list1,
    list2..., in which all duplicated elements have been dropped. The TagPosition
    indicates the position of an indicator label within each of the elements in
    the lists. TaggedUnion converts each of these tags into a taglist. The tags of
    the dropped duplicates are added to the taglists of the remaining element.";

TaggedUnion[union__, TagPosition_] := Module[{result = {}},
  (result = add[#, result, TagPosition]) & /@ Flatten[{union}, 1];
  result
 ]
```

```
add[el_, list_, tagpos_] := Module[{p, newlist},
  p = Position[list, ReplacePart[el, tagpos -> Blank[]]];
  If[p === {},
   (*Print["elem not there yet. adding"];*)
   newel = ReplacePart[el, tagpos -> {Extract[el, tagpos]}];
   (*Print["newel-=",newel];*)
   newlist = Append[list, newel],

   (*add tag to the existing element*)
   p = p[[1]];
   (*Print["elem already there, at ",p];*)
   newel = Extract[list, p];
   newel =
    ReplacePart[newel, tagpos -> Union[Extract[newel, tagpos], {Extract[el, tagpos]}]];
   newlist = ReplacePart[list, p -> newel]
  ];
  newlist
 ]
```

StringStartsWith

```
StringStartsWith::usage = "StringStartsWith[string,start]
    returns True if string with start, and False otherwise.";

StringStartsWith[string_, start_] :=
 If[StringLength[string] < StringLength[start], False,
  StringTake[string, StringLength[start]] == start]
```

Parser Support Stuff

```
newparser[G_, R_, Term_, start_gterm] := Module[{ps},
  ps = PS[G, {{}}, {}, R, {Term}, start];
  ps = Predictor[ps, start, 1];
  ps]

getitem[PS_, itemptr[setnr_, tag_]] :=
  If[setnr < 0 || tag === Null, DPPrint["Internal error: getitem: setnr<0 or tag=Null"],
   Cases[PS[[ITEMSETS, setnr]], item[tag, ___]][[1]]
  ];

getpitem[PS_, pitem[setnr_, tag_]] :=
  If[setnr < 0 || tag === Null, DPPrint["Internal error: getpitem: setnr<0 or tag=Null"],
   Cases[PS[[ITEMSETS, setnr]], subparser[tag, ___]][[1]]
  ];

IndexOfLink[ps_, p : pitem[setnr_, tag_]] := Module[{},
   pos = Position[ps[[ITEMSETS, setnr]], subparser[tag, ___], {1}];
   If[pos === {}, DPError["Parser internal err. No item ", PToString[p]]; {},
    {ITEMSETS, setnr, pos[[1, 1]]}
   ]
  ];

replaceitem[PS_, itemptr[setnr_, tag_], newitem_item] := Module[{p},
  If[setnr < 0 || tag === Null,
   DPPrint["ERRR replaceitem: setnr<0 or tag=Null"], p = Position[PS, item[tag, ___]];
   (*DPPrint["replaceitem:",p];*)
   If[Length[p] =!= 1,
    DPPrint["Internal err: no item ", tag, " in set ", setnr],
    ReplacePart[PS, newitem, p]
   ]
  ]
 ]
```

```
startsets[PS_, it : itemptr[set_, tag_]] := Module[{theitem},
  theitem = getitem[PS, it]; (*item[tag,N,left,right,vars,pred,succ]*)
  If[theitem[[6]] === {}, {set},
   Union @@ (startsets[PS, #] & /@ theitem[[6]])
  ]
 ]

endsets[PS_, it : itemptr[set_, tag_]] := Module[{theitem},
  theitem = getitem[PS, it]; (*item[tag,unifiedhead,left,right,vars,pred,succ]*)
  If[theitem[[4]] === {}, {it},
   Union @@ (endsets[PS, #] & /@ theitem[[7]])
  ]
 ]

shiftdot[item[tag_, head_, {l___}, {s_, r___}, vars_, pred_, suc_], setnr_] :=
  item[Unique[item], head, {l, s}, {r}, vars, {itemptr[setnr, tag]}, {}];
(*shiftdotback[item[tag_,head_,{l___,s_},{r___},vars_,parents_],setnr]:=
  item[tag,head,{l},{s,r},more];*)

CanInterruptHere[item[tag_, unifiedhead_gterm,
   left_List, right_List, vars_List, pred_List, succ_List]] :=
 (right === {} || (Head[right[[1]]] =!= Code))
```

Interpreting parse states

```
FinishedShifts[PS_] := Module[{rules, n, goodshifts},
   n = Length[PS[[ITEMSETS]]];
   rules = {#, UnifySynthesize[shiftpos[PS[[STARTSYMBOL]], 1, n, $somecreator], #]} & /@
     PS[[SHIFTPOSS]];
   goodshifts = Select[rules, #[[2]] =!= $Failure &];
   (*11.6.7: at this point I can't clearly see whether
     we need futher testing on the goodshifts. Can stray items looking
     for something similar to the startsymbol be introduced by a tricky
     grammar? And would that matter anyway? For now I just get the shifts. *)
   Union[#[[1]] & /@ goodshifts]
  ];

Finished[PS_] := FinishedShifts[PS] =!= {}

Finished[$Failure] := False;

OnTrack[PS_] := TryToComplete[PS] =!= $Failure

OpenEnded[PS_] := Last[PS[[ITEMSETS]]] =!= {}

AllTuples[{}, _] := {};
AllTuples[{x_, rest___}, list2_List] := Join[AllTuples[{rest}, list2], {x, #} & /@ list2]

FinishedShiftsOfPair[parserpair[par1_, par2_, eat_, finished]] :=
 Module[{shifts1, shifts2},
  shifts1 = #[[1]] & /@ FinishedShifts[par1];
  shifts2 = #[[1]] & /@ FinishedShifts[par2];
  ({par @@ #, eat}) & /@ AllTuples[shifts1, shifts2]
 ]

Status[PS_] := Which[Finished[PS], finished, OnTrack[PS], ontrack, True, dead];

PairStatus[par1_, par2_] :=
  Module[{res},
   DPLevel[1];
   (*DPPrint["PairStatus checking.\n",par1,"\n EN \n",par2];*)
   res = Which[
     Finished[par1] && Finished[par2], DPPrint["FINISHED"]; finished,
     OnTrack[par1] && OnTrack[par2], DPPrint["ONTRACK"]; ontrack,
     True, DPPrint["DEAD"]; dead];
   DPLevel[-1];
   res];

alive[dead] := False;
alive[_] := True;
```

```
Instantiate[term_, vars_List] := term //. ((Rule @@ #) & /@ vars);
```

"Pretty"Printing the parse items etc

```
PFullInfo = True; (*Print extended item info*)

Clear[PToString]

PToString[shiftpos[symbol_, a_, b_, creator_]] := "<" <> PToString[symbol] <>
  "," <> ToString[a] <> "," <> ToString[b] <> " by " <> ToString[creator] <> ">"

PToString[varchange[newvars_, a_, b_, creator_]] := "<" <> PToString[newvars] <>
  "," <> ToString[a] <> "," <> ToString[b] <> ToString[creator] <> ">"

PToString[
   item[tag_, unifiedhead_gterm, left_List, right_List, vars_List, pred_List, suc_List]] :=
  "<" <> ToString[tag] <> ":" <> PToString[unifiedhead] <> "→" <> PToString[left, " "] <>
   "·" <> PToString[right, " "] <> If[PFullInfo, " vars:" <> PToString[vars, ","] <>
     " pred:" <> PToString[pred] <> " suc:" <> PToString[suc], ""] <> ">";

PToString[gterm[sym_String]] := sym;

PToString[gterm[symbol_String, attributes__]] :=
 symbol <> "[" <> PToString[{attributes}, ","] <> "]"

PToString[x_] := ToString[x, InputForm] (*InputForm in order to get 1/k print as 1/k *)

PToString[Equal[var_, value_]] := ToString[var] <> "=" <> ToString[value];

PToString[itemptr[setnr_, tag_]] := ToString[tag] <> "@" <> ToString[setnr]

PToString[s_String] := s;
PToString[not[s_gterm]] := "~" <> PToString[s];

PToString[a_List] := "{" <> PToString[a, ","] <> "}";

PToString[{}, separator_] := {};
PToString[{a_}, separator_] := PToString[a];
PToString[{a_, rest__}, separator_] :=
  PToString[a] <> separator <> PToString[{rest}, separator];

PToString[par[X_gterm, Y_gterm]] := "(" <> PToString[X] <> "//" <> PToString[Y] <> ")";

PToString[pitem[p_par, setnr_]] := "activesubp" <> PToString[p] <> "@" <> ToString[setnr]

PToString[GrammarRule[s_gterm, sym_List]] := PToString[s] <> ":=" <> PToString[sym, " "];

PToString[subparser[tag_, head_, sharedvars_, eatlist_]] :=
  "\nSUBPARSER " <> ToString[tag] <> " for " <>
   PToString[head] <> PToString[eatlist] <> "\nEND SUBPARSER " <> ToString[tag];
 (*don't print sharedvars, easy to see from the head.*)

PToString[parserpair[firstparser_PS, secondparser_PS, whoate_, stat_]] :=
  "\n***Pair with eatpattern '" <> whoate <> "'. status:" <> ToString[stat] <> "\nParser 1" <>
   PToString[firstparser] <> "\nParser 2" <> PToString[secondparser] <> "\n***";

PToString[PS[G_List, U_, S_, R_List, T_, N_]] := Module[{setnumber},
  StringJoin[(*leave out grammar, rules: takes much space*)
   "\nHead:", PToString[N],
   Table["\n<SET " <> ToString[setnumber] <> "---\n" <> "term:" <> ToString[T[[setnumber]]] <>
     "\nitems:" <> PToString[Select[U[[setnumber]], Head[#] =!= subparser &], "\n"] <>
     "\nshiftpos:" <> PToString[Cases[S, shiftpos[_, setnumber, __]], ","] <>
     PToString[Cases[S, varchange[_, setnumber, __]], ","] <> "\n" <>
     PToString[Cases[U[[setnumber]], _subparser], " "], {setnumber, 1, Length[U]}], "\n>"]

 ]

PToString[RuleAtPos[lhs_, pos_, rhs_]] :=
  ToString[lhs] <> " @" <> PToString[pos] <> "-> " <> ToString[rhs];
```

```
PToString[RuleAtPos[lhs_, rhs_]] := ToString[lhs] <> "->" <> ToString[rhs];
```

Rule rewriting

■ ApplyRewriteRule

```
ApplyRewriteRule[rule_RuleAtPos, vars_List, term_] :=
  ApplyInstRewriteRule[Instantiate[rule, vars], vars, term];

ApplyInstRewriteRule[RuleAtPos[lhs_, pos_, rhs_], vars_, term_] :=
 Module[{subterm, res, newvars},
  subterm = ExtractX[term, pos];
  (*Print["MatchQ[",subterm,",",lhs,"]=",MatchQ[subterm,lhs]];*)
  If[! MatchQ[subterm, lhs],
   $Failure, (*Fail if rule does not fit*)

   res = Replace[subterm, Rule[lhs, rhs]];
   If[Head[res] === Code,
    $CurrentTerm = term;
    {res, newvars} = CodeEval[vars, res]];
   If[res === $Failure, res, ReplacePartX[term, res, pos]]
  ]
 ]
```

Rewrite testing

```
AllRewritesForRule[RewriteRule[rulehead_gterm, rule_RuleAtPos],
  neededhead_gterm, term_] := Module[{vars, res},
  vars = UnifyInherit[neededhead, rulehead];
  If[vars === $Failure, $Failure,
   (*Print["AllRewritesForRule: vars=",vars];*)
   res = AllRewritesForInstRule[Instantiate[rule, vars], vars, term];
   (*Instantiate the head*)
   {Instantiate[rulehead, #[[1]]], #[[2]]} & /@ res
  ]
 ]

AllRewritesForInstRule[rule : RuleAtPos[lhs_, pos_, rhs_], vars_, term_] :=
 Module[{positions, unsetvars, res},
  If[Is$Var[pos], (*then we need to find out possible $pos ourselves*)
   positions = Position[term, lhs];
   res = {Append[vars, Equal[pos, #]],
       ApplyInstRewriteRule[rule /. pos → #, vars, term]} & /@ positions
   ,
   res = {{vars, ApplyInstRewriteRule[rule, vars, term]}}
  ];
  (*Print["candidates:",res];*)
  Select[res, #[[2]] =!= $Failure &]
 ]
```

■ AllRewrites

```
AllRewrites[rules_List, neededhead_gterm, term_] := Module[{res = $Failure, n = 1},
  While[n ≤ Length[rules] && res === $Failure,
   res = AllRewritesForRule[rules[[n]], neededhead, term];
   (*Print["AllRewrites next. res=",res];*)
   n = n + 1;
  ];
  res
 ]
```

Modified CFG parser

```
AddItemset[PS_, newterm_] := Module[{newps = PS},
   newps[[ITEMSETS]] = Append[newps[[ITEMSETS]], {}];
   newps[[TERMS]] = Append[newps[[TERMS]], newterm];
   newps];

Scanner[PS_, rewrite_gterm, newterm_] := Module[{newps = PS, n, subparsers, oldlast},
  DPPrint["scan ", PToString[rewrite]];
  oldlast = Last[newps[[ITEMSETS]]];
  newps = AddItemset[newps, newterm];
  n = Length[newps[[ITEMSETS]]];
  newps = Addshift[newps, shiftpos[rewrite, n - 1, n, {}]];

  (newps = SubScanner[newps, #, rewrite, newterm]) & /@ Cases[oldlast, pitem[___]];
  newps
 ]
```

GetSharedVars

```
GetSharedVars[it : item[tag_, unifiedhead_gterm, left_List, right_List, vars_List,
     pred_List, succ_List], sharedvars_, setnr_] := If[CanInterruptHere[it],
   {shared[itemptr[setnr, tag], Select[vars, MemberQ[sharedvars, #[[1]]] &]]},
   {}
  ];

GetSharedVars[PS[Gram_, {old___, lastitemset_}, shiftpos_, rules_, terms_, startsym_],
   sharedvars_] := Module[{shared},
   shared = GetSharedVars[#, sharedvars, Length[{old}] + 1] & /@ lastitemset;
   If[shared === {}, {},
    TaggedUnion[Sequence @@ shared, {1}]]
  ];
```

ParScan

```
ScanEF[eatingparser_, followingparser_, t_, newterm_, sharedvars_] :=
  Module[{neweatp, eatpsharedvars, newfollowp, n},
   (*vars change at the eating parser*)
   neweatp = Scanner[eatingparser, t, newterm];
   DPPrint["ScanEF: Scanner complete."];
   (*determine new var values*)
   eatpsharedvars = GetSharedVars[neweatp, sharedvars];
   (*this gives a list like {shared[{item$625,item$634},{$x=="1"},..}*)
   (*now, add shift for each shared var-list*)
   DPPrint["shared vars: ", eatpsharedvars];
   newfollowp = AddItemset[followingparser, newterm];
   n = Length[newfollowp[[ITEMSETS]]];
   (newfollowp = Addshift[newfollowp, varchange[#[[2]], n - 1, n, #[[1]]]]) & /@
    eatpsharedvars;
   (*TO DO: subparsers WITHIN subparsers have to be checked, just as in the Scanner.*)
   {neweatp, newfollowp}
  ];

ParScan[parserpair[par1_, par2_, whoate_, dead], t_, newterm_, sharedvars_] :=
  parserpair[par1, par2, whoate, dead];

ParScan[parserpair[par1_, par2_, whoate_, stat_], t_, newterm_, sharedvars_] :=
 Module[{par1eats, par2eats},
  DPPrint["ParScan eating parser 1"];
  par1eats = ScanEF[par1, par2, t, newterm, sharedvars];
  DPPrint["ParScan eating parser 2"];
  par2eats = ScanEF[par2, par1, t, newterm, sharedvars];
  DPPrint["ParScan complete."];
  {parserpair[par1eats[[1]], par1eats[[2]], whoate <> "0",
    PairStatus[par1eats[[1]], par1eats[[2]]]], parserpair[par2eats[[2]],
    par2eats[[1]], whoate <> "1", PairStatus[par2eats[[1]], par2eats[[2]]]]}]
```

```
SubScanner[PS_, paritem_pitem, t_, newterm_] :=
  Module[{newps = PS, newsubp, newsubparsers, S, subp, subparsers, neweatlist,
    startset, endset, shiftposs, finishedparsers, sharedvars, parindex},
   DPPrint["Subscanner called for ", PToString[paritem]];
   startset = paritem[[1]];
   (*pitem should point to last
    itemset. This is normal for the parsing process we use now. *)
   endset = Length[PS[[ITEMSETS]]]; parindex = IndexOfLink[PS, paritem];
   (*ptr to the parserpair list of the subparser in question:
     subparser[tag,par(X,Y),{sharedvars},***{parserpair}***]*)
   subp = Extract[PS, parindex];
   sharedvars = subp[[3]];
   subparsers = subp[[4]];
   (*DPPrint["subparser state:",PToString[subp]];*)
   newsubparsers = Flatten[ParScan[#, t, newterm, sharedvars] & /@ subparsers];
   (*DPPrint["new subparsers:",newsubparsers];*)
   newps = ReplacePart[newps, newsubparsers, Append[parindex, 4]];
   (*Check result of these parsers *)


   (*For each finished parserpair, add a shiftpos.*)
   (*Do this for each parserpair,
   as they all may have different settings for the variables *)

   finishedparsers = Cases[newsubparsers, parserpair[_, _, _, finished]];
   (*parserpair[firstp,secondp,whoate,stat]*)
   (* Get the Finished Shifts and the eat pattern for each finished parser.*)
   DPPrint["finishedparsers=", finishedparsers];
   shiftposs = Union @@ (FinishedShiftsOfPair /@ finishedparsers);
   (*Union removes duplicates (and makes a set of the set of sets)*)
   (*shiftpos is a list of pairs {par[..],eatpattern} *)
   DPPrint["tagged shiftpos found:", shiftposs];
   (* We use pitemlink[pitem,eatpattern] to refer to the 'creator' of this shift. *)
   (newps = Addshift[newps,
        shiftpos[#[[1]], startset, endset, {pitemlink[paritem, #[[2]]]}]]) & /@ shiftposs;

   (* Add pointer to subparser to last set if at least one parser alive,
   so that subparser keeps running next round. *)
   If[MemberQ[newsubparsers, parserpair[_, _, _, _?alive]],
    newps[[ITEMSETS, endset]] = Append[newps[[ITEMSETS, endset]], paritem]
   ];
   DPPrint["Subscanner complete"];
   (*DPPrint["res=",newps//PToString];*)
   newps
  ];

UnifySynthesize1[item[_, _, _, {}, __], head_] := $Failure;
UnifySynthesize1[item[_, _, _, {_Code, ___}, ___], head_] := $Failure;
UnifySynthesize1[item[_, _, _, {term1_, ___}, vars_, _, _], head_] :=
  UnifySynthesize[Instantiate[term1, vars], head];
UnifySynthesize1[item[_, _, _, {not[term1_], ___}, vars_, _, _], head_] :=
  UnifySynthesize[Instantiate[not[term1], vars], head];

UnifySynthesize1[_subparser, head_] := $Failure;
UnifySynthesize1[_pitem, head_] := $Failure;(*pitems handled by Subscanner*)
UnifySynthesize1[item[_, _, _, {par[terms__], ___}, vars_, _, _], head_] :=
  UnifySynthesize[Instantiate[par[terms], vars], head];
```

```
Addshift[PS_, shift : shiftpos[head_, left_, right_, creator_List]] :=
 Module[{newps = PS, checkitems, oldshift},
  DPPrint["addshift ", PToString[shift](*,"to ",newps*)];
  oldshift = Position[newps[[SHIFTPOSS]], shiftpos[head, left, right, _]];

  If[oldshift ≠ {},
   DPPrint["Shift is already there. Just merging creatorlists."];
   If[Length[oldshift] ≠ 1, DPError[
     "Internal inconsistency: multiple shifts for ", PToString[shift], "exist."]];
   oldshift = Sequence @@ oldshift[[1]];
   newps[[SHIFTPOSS, oldshift, 4]] = Union[newps[[SHIFTPOSS, oldshift, 4]], creator];
   ,
   newps[[SHIFTPOSS]] = Append[newps[[SHIFTPOSS]], shift];
   (*DPPrint["newps=",newps//PToString];*)
   checkitems = {#, UnifySynthesize1[#, head]} & /@ newps[[ITEMSETS, left]];
   (*checkitems is now a list of pairs {item,newvars} where newvars is
    $Failure if item can not use shift, or some var settings if it fits*)
   (*DPPrint["check results ",checkitems];*)
   checkitems = Select[checkitems, #[[2]] =!= $Failure &];
   DPPrint["items that can shift", checkitems];

   (*item[tag,gterm,left,right,vars,pred,succ]*)
   (*Incorporate the required variable requirements for each possible shift*)
   (*for each succesful shift, create shifted item and add it.*)
   (*Additem copes with cases when item already there.*)
   (newitem = shiftdot[#[[1]], left];
      newitem[[5]] = Join[newitem[[5]], #[[2]]];
      newps = Additem[newps, newitem, right]) & /@ checkitems
  ];
  newps
 ]

Addshift[PS_, shift : varchange[newvars_, left_, right_, creator_List]] :=
 Module[{newps = PS, checkitems, newitem, oldshift},
  DPPrint["addshift varchange ", PToString[shift](*,"to ",newps*)];
  oldshift = Position[newps[[SHIFTPOSS]], varchange[head, left, right, _]];
  If[oldshift ≠ {},
   DPPrint["Shift is already there. Just merging the creatorlists."];
   If[Length[oldshift] ≠ 1, DPError[
     "Internal inconsistency: multiple shifts for ", PToString[shift], "exist."]];
   oldshift = Sequence @@ oldshift[[1]];
   newps[[SHIFTPOSS, oldshift, 4]] = Union[newps[[SHIFTPOSS, oldshift, 4]], creator];
   ,
   newps[[SHIFTPOSS]] = Append[newps[[SHIFTPOSS]], shift];
   (*DPPrint["itemset=",newps[[ITEMSETS,left]]];*)
   checkitems = Select[newps[[ITEMSETS, left]], CanInterruptHere];
   DPPrint["items that can shift varchange", checkitems];

   (*Incorporate the required variable requirements for each possible shift*)
   (*for each succesful shift, create shifted item and add it.*)
   (*Additem copes with cases when item already there.*)
   (newitem = #(*create copy*); newitem[[1]] = Unique[item];
      newitem[[6]] = {itemptr[left, #[[1]]]}; (*set predecessor*)
      newitem[[5]] = SetVars[newitem[[5]], newvars];
      newps = Additem[newps, newitem, right]) & /@ checkitems
  ];
  DPPrint["completed addshift varchange:", newps // PToString];
  newps
 ]
```

Predictor

```
Predictor[PS_, head_, setnr_] :=
 Module[{newitems, n, newps = PS, term, res, newparser1, newparser2, tag, sharedvars},
  DPPrint["predictor called for ", head];
  Which[
   Head[head] === not,
   DPPrint["*******Testing not(", PToString[head], ")*******"];
   If[! Applicable[PS[[GRAMMAR]], PS[[REWRITERULES]], Last[PS[[TERMS]]], head[[1]]],
    DPPrint["******NOT() IS TRUE******"];
    newps = Addshift[PS, shiftpos[head, setnr, setnr, {}]],
    DPPrint["******NOT() IS FALSE******"];
   ]
   ,
   Head[head] === par,
   (*check if paritem not there yet. Just to avoid troubles.*)
   (*cheap check. Really should do Unification...*)
   (*This MAY cause problems at some point and may need correction*)
   If[MemberQ[newps[[ITEMSETS, setnr]], subparser[_, head, _]],
    DPPrint["Predictor warning: double occurance of " <>
      PToString[N] <> "in setnr" <> ToString[setnr]],

    newparser1 =
     newparser[PS[[GRAMMAR]], PS[[REWRITERULES]], Last[PS[[TERMS]]], head[[1]]];
    newparser2 = newparser[PS[[GRAMMAR]], PS[[REWRITERULES]],
      Last[PS[[TERMS]]], head[[2]]];
    tag = Unique[subp];
    (*determine the shared vars*)
    (*sharedvars do not have a value anyway*)
    sharedvars = Intersection[GetVars[head[[1]]], GetVars[head[[2]]] ];
    newitems = {pitem[setnr, tag], subparser[tag, head,
       sharedvars, {parserpair[newparser1, newparser2, "", ontrack]}]};
    DPPrint["subparser added!!!"];
    (*TODO: Check if head has been recognised right away. *)
    newps[[ITEMSETS, setnr]] = Join[newps[[ITEMSETS, setnr]], newitems];
   ]
   ,
   True, (*default case: predict nonterminal*)
   (*DPPrint["Searching for ",head," in ",newps[[GRAMMAR]]];*)
   newitems =
    Select[{#, UnifyInherit[head, #[[1]]]} & /@ newps[[GRAMMAR]], #[[2]] =!= $Failure &];
   (*newitems is list of pairs {grammarrule,unificationvars}*)
   DPPrint["new predictor items:", newitems];
   (newps = Additem[newps, item[Unique[item], Instantiate[#[[1, 1]], #[[2]]],
        {}, #[[1, 2]], #[[2]], {}, {}], setnr]) & /@ newitems;
  ];
  newps
 ]
```

Additem

```
addsuctoitem[ps_, item_itemptr, new_itemptr] := Module[{replacement},
   (*DPPrint["addsuctoitem",ps,item,new];*)
   replacement = getitem[ps, item];
   (*DPPrint["addsuctoitem replacing ",replacement];*)
   replacement[[7]] = Union[replacement[[7]], {new}];
   replaceitem[ps, item, replacement]
  ];
```

```
addpredtoitem[ps_, item_itemptr, new_itemptr] := Module[{replacement},
   (*DPPrint["addsuctoitem",ps,item,new];*)
   replacement = getitem[ps, item];
   (*DPPrint["addsuctoitem replacing ",replacement];*)
   replacement[[6]] = Union[replacement[[6]], {new}];
   replaceitem[ps, item, replacement]
  ];

Additem[PS_,
  it : item[tag_, head_gterm, left_, right_, vars_List, pred_List, suc_List], setnr_] :=
 Module[{newps = PS, sitem, p, existingitem, existingitemptr, rightends},
  DPPrint["Additem ", PToString[it], " to ", setnr];
  (*instantiatedhead=Instantiate[head,vars];
  instantitem=it;instantitem[[2]]=instantiatedhead;*)

  p = Position[newps[[ITEMSETS, setnr]], item[_, head, left, right, vars, __]];
  If[p === {} (*Item is NOT yet there*),
   newps[[ITEMSETS, setnr]] = Append[newps[[ITEMSETS, setnr]], it];
   If[right === {},
    DPPrint["Call Completer"];
    newps = Completer[newps, itemptr[setnr, tag]]
    ,
    (*DPPrint["head =",Head[right[[1]]]];*)
    If[Head[right[[1]]] === Code,
     DPPrint["code blok found"];
     sitem = shiftdot[it, setnr];
     $CurrentTerm = PS[[TERMS, setnr]];
     {res, sitem[[5]]} = CodeEval[vars, right[[1]]];
     (*DPPrint["result:",sitem[[5]]];*)
     If[res =!= $Failure, newps = Additem[newps, sitem, setnr]]
     ,
     DPPrint[right[[1]], " instantiated gives ", Instantiate[right[[1]], vars]];
     newps = Predictor[newps, Instantiate[right[[1]], vars], setnr]
    ]
   ];
    (*DPPrint["add finished. set now ",PToString[newps]];*)
    (*Update suc links in the predecessors. *)
    (*this update only OK if item was added in first place!!*)
   (newps = addsuctoitem[newps, #, itemptr[setnr, tag]]) & /@ pred;
    (*DPPrint["updated suc finished. set now ",PToString[newps]];*)


   ,
    (*Item already there. This can happen in various ways
    and pred, suc both may be wrong in the item already there.
      Strictly we need to check only new combinations of
    endsets  of already existing item with the startsets of this item.
      Here we are lazy: add the link and (re)check ALL endpoints.*)
   existingitem = Extract[newps[[ITEMSETS, setnr]], p[[1]]];
   DPPrint["ALREADY THERE. existing:", existingitem];
   existingitemptr = itemptr[setnr, existingitem[[1]]];

   (*create links between existing and all pred of new item.*)
   (newps = addsuctoitem[newps, #, existingitemptr];
      newps = addpredtoitem[newps, existingitemptr, #] ) & /@ pred;
   DPPrint["newps after addpredtoitem and addsuctoitem:", newps // PToString];
   (*and re-run completer on all right-complete ends*)
   rightends = endsets[newps, existingitemptr];
   DPPrint["right ends:", rightends];
   (newps = Completer[newps, #]) & /@ rightends;
  ];

  newps
 ]
```

```
Completer[PS_, it : itemptr[set_, tag_]] := Module[{newps = PS, theitem, start, head},
  DPPrint["item is complete. Checking"];
  theitem = getitem[newps, it]; (*item[tag,head,left,right,vars,pred,suc]*)
  start = startsets[newps, it];
  DPPrint["start items =", start];
  head = theitem[[2]] //. ((Rule @@ #) & /@ theitem[[5]]);
  (*DPPrint["instantiated head:",head];*)
  (newps = Addshift[newps, shiftpos[head, #, set, {it}]]) & /@ start;
  newps
 ]
```

The ApplicableParser

```
Applicable[grammar_, rewriterules_, startterm_, startsym_gterm] :=
 Module[{grammar1, startsym1, newps, rule, newterms,
   res = $Failure, checkitems, pickedrule, pickedvars},

  If[terminal[startsym[[1]]],
   (*If terminal, we can't use grammar because we need start symbol*)
   (*Then we tweak the grammar with a new start symbol.*)
   startsym1 = gterm["UnusedGrammarSymbol54"];
   grammar1 = Append[grammar, GrammarRule[startsym1, {startsym}]];
   ,
   grammar1 = grammar; startsym1 = startsym;
  ];

  newps = newparser[grammar1, rewriterules, startterm, startsym1];
  DPPrint["start parser:\n", PToString[newps]];
  res = TryToComplete[newps];
  DPPrint["****TryToComplete returned:", PToString[res]];
  Finished[res]
 ]
```

TryToComplete

■ AllBlockedItems

```
AllBlockedItems[ps_PS] := Module[{directitems, subitems, lastitemset},
  lastitemset = Last[ps[[ITEMSETS]]];
  directitems = Cases[lastitemset,
    item[tag_, head_, lhs_, {gterm[_?terminal, ___], ___}, vars_, prev_, next_]];
  subitems = Union @@ (AllBlockedItems[ps, #] & /@ Cases[lastitemset, _pitem]);
  Union[directitems, subitems]
 ]

AllBlockedItems[ps_PS, ptr_pitem] := AllBlockedItems[getpitem[ps, ptr]];

AllBlockedItems[subparser[tag_, par_, sharedvars_, ppairs_List]] :=
 Module[{aliveparsers, blockeditems},
  aliveparsers = Cases[ppairs, parserpair[_, _, _, _?alive]];
  blockeditems =
   Union[AllBlockedItems[#[[1]]], AllBlockedItems[#[[2]]]] & /@ aliveparsers;
  Union @@ blockeditems
 ]
```

■ TryToComplete

```
TryToComplete[PS_] :=
  Module[{blocked, terminals, m, n, newterms, res = $Failure, ps2, rewriterule},
   DPPrint["TryToComplete trying next level on", PS // PToString];
   (*DPPrint["PS=",PS];*)
   If[Finished[PS],
    res = PS (*we are already finished.Return parse state*)
    ,
    (*Try all possible next rewrites. Before we tried ALL rules here,
    but since we use attributes we need to generate our tries from the state at hand,
    as we need to get the attribute values.*)
    blocked = AllBlockedItems[PS];
    DPPrint["TryToScan... trying to unblock ", blocked];
    terminals = Instantiate[#[[4, 1]], #[[5]]] & /@ blocked;
    DPPrint["terminals that may be applicable=", terminals];
    (*terminals= {gterm} List referring to terminals (rewrite actions) that
      may proceed the parse. Now check if we can apply one of these rewrite
      actions to the term at hand. Just finding a rule with matching name
      is insufficient: we have to apply the rule and see if it succeeds. *)

    For[m = 1, m ≤ Length[terminals] && res === $Failure, m++,
     t = terminals[[m]];
     allrewrites = AllRewrites[PS[[REWRITERULES]], t, Last[PS[[TERMS]]]];
     DPPrint["possible rewrites with rule ", t // FullForm, ":", allrewrites];
     (*allrewrites=List of pairs {newhead,newterm}. Try each possibility*)

     For[n = 1, n ≤ Length[allrewrites] && res === $Failure, n++,
      DPPrint["trying rewrite ", n, ":", allrewrites[[n]]];
      ps2 = Scanner[PS, allrewrites[[n, 1]], allrewrites[[n, 2]]];
      DPPrint["scanner gives\n", PToString[ps2]];
      res = TryToComplete[ps2];
     ]
    ];
   ];
   DPPrint["TrytoComplete returning "];
   res
  ];
```

Generating hints

FindPathBack

```
FindPathBack[PS_, it : itemptr[set_, tag_], startset_] :=
  Module[{theitem, preds, previousitem, previoussymbol, ms, n, res},
   theitem = getitem[PS, it];
   (*item[tag,N,left,right,vars,pred,succ]*)
   DHPrint["findpath back from ", tag, ":", theitem // PToString];
   preds = theitem[[6]];
   If[preds === {},
    (*We arrived at the start of a path *)
    If[set == startset, {}, Null],

    (*Not yet at start, check if there is solution for one of back steps*)
    res = Null; n = 0;
    While[ n < Length[preds] && res === Null,
     n = n + 1;
     res = FindPathBack[PS, preds[[n]], startset];
     DHPrint["findpathback returned for " tag, ":", res];
    ];
    If[res =!= Null, previousitem = getitem[PS, preds[[n]]]];
    DHPrint["cheching which case"];
    Which[
     (*Check if we actually found something. *)
     res === Null, Null, (*no, then return immediately*)

     (*Check dot position. If not changed, then *)
     (* this must have been a varchange, no shift*)
     (*We ignore the details of varshift, *)
     (*the eatpattern seems enough to reconstruct the parse*)
     Length[theitem[[3]]] == Length[previousitem[[3]]], res,

     (*else this was a regular shift.*)
     (*If it was a Code block, there is no related shiftpos, *)
     (*and we are ready*)
     Head[previousitem[[4, 1]]] === Code, res,

     (*If it was a not, there is no regular shift either*)
     Head[previousitem[[4, 1]]] === not, res,

     (* All cases checked, must be regular shift*)
     True,
     (*Instantiate the previous symbol with the vars at that time*)
     DHPrint["regular shift."];
     previoussymbol = Instantiate[previousitem[[4, 1]], previousitem[[5]]];
     DHPrint["MatchingSfhit called from FindPathBack"];
     Append[res, MatchingShift[PS, shiftpos[previoussymbol, preds[[n]][[1]], set, Null]]]
    ]
   ]
  ];

MatchingShift[PS_, s : shiftpos[not[t_], __]] := s;

MatchingShift[PS_, sp : shiftpos[t_, left_, right_, _]] := Module[{matches, shift},
   shift = ReplacePart[sp, 4 -> $anythingcreators612];
   DHPrint["Getting shift ", shift]; matches = {#, Unify[shift, #]} & /@ PS[[SHIFTPOSS]] ;
   matches = Select[matches, #[[2]] =!= $Failure &];
   DHPrint["matches found:", matches]; Assert[matches =!= {},
    "MatchingShifts failed. There seems no shift like " <> PToString[shift // PToString]];

   matches[[1, 1]]
  ];
```

▪ ParseTreeForShift

```
ParseTreeForShift[ps_PS, shiftpos[term_gterm, left_, right_, {}]] := node[term, {}];

ParseTreeForShift[ps_PS,
  shiftpos[term_gterm, left_, right_, {alternative1_, ___}]] := node[term,
  GenerateParseTree[ps, FindPathBack[ps, alternative1, left]]]

GenerateParseTree[PS_, shiftpos_List] := ParseTreeForShift[PS, #] & /@ shiftpos;

ParseTreeForShift[ps_PS,
  shiftpos[par[head1_, head2_], left_, right_, {pitemlink[pitem_, eatpattern_], ___}]] :=
 Module[{thesubparser, parserpairs, shiftpat, thepair},
  thesubparser = getpitem[ps, pitem];
  parserpairs = thesubparser[[4]];
  (*#[[3]] of each parserpair is the eatpattern*)
  thepair = Select[parserpairs, StringStartsWith[#[[3]], eatpattern] &];
  Assert[thepair =!= {}, "Subparser does not have requested eat pattern."];
  thepair = thepair[[1]];
  shiftpat = thesubparser[[2]];
  DHPrint["shiftpat=", shiftpat];
  parshift[shiftpat, eatpattern,
   ParseTreeForShift[thepair[[1]],
    MatchingShift[thepair[[1]], shiftpos[head1, left, right, {}]]],
   ParseTreeForShift[thepair[[2]], MatchingShift[thepair[[2]],
     shiftpos[head2, left, right, {}]]]
  ]
 ]
```

▪ GetSuggestionTree

```
GetSuggestionTree[ps_PS] := Module[{suggestion, tree},
   suggestion = TryToComplete[ps];
   DPPrint["returned to GetSuggestionTree."];
   If[suggestion === $Failure, $Failure,
    DHPrint["hint found, Building tree. suggestino=", suggestion];
    tree = ParseTreeForShift[suggestion, FinishedShifts[suggestion][[1]]];
    (*tree is a parse tree, now only get the suggestion out. NYI*)
    tree
   ]
  ];
```

▪ GetTerminalsFromTree

```
GetTerminalsFromTree[node[gterm[name_, para___], {}]] :=
  If[terminal[name], {gterm[name, para]}, {}];
GetTerminalsFromTree[node[gterm[name_, para___], children_]] :=
  Join @@ (GetTerminalsFromTree /@ children);
GetTerminalsFromTree[parshift[par[X_, Y_], eatpattern_, P1_, P2_]] := Module[{t1, t2},
  t1 = GetTerminalsFromTree[P1];
  t2 = GetTerminalsFromTree[P2];
  MakePermutation[t1, t2, eatpattern]
 ]

MakePermutation[l1_List, l2_List, perm_String] :=
  MakePermutation[l1, l2, Characters[perm]];

MakePermutation[{x_, l1___}, l2_List, {"0", perm___}] :=
  Join[{x}, MakePermutation[{l1}, l2, {perm}]];
MakePermutation[l1_List, {x_, l2___}, {"1", perm___}] :=
  Join[{x}, MakePermutation[l1, {l2}, {perm}]];
MakePermutation[{}, {}, {}] := {};
```

■ VisualizeTree makes a 2 D tree plot of the parse graph

```
VisualizeTree[tree_] := Module[{rootlabel, treeterms},
   {rootlabel, treeterms} = ConvertTree[tree];
   LayeredGraphPlot[treeterms, VertexLabeling → True,
    DirectedEdges → True, PackingMethod → "LayeredTop", AspectRatio → 0.5]
  ];

ConvertTree[node[N_, items_List]] :=
  Module[{node, subtreesplusroot, roots, subtrees},
   node = PToString[N] <> "   " <> ToString[Unique[]];
   If[items === {}, {node, {}}
    ,
    subtreesplusroot = Transpose[ConvertTree /@ items];
    roots = subtreesplusroot[[1]];
    subtrees = Join @@ subtreesplusroot[[2]];
    {node, Join[(node → #) & /@ roots, subtrees]}
   ]
  ];

ConvertTree[parshift[paritem_, eatpat_, P1_, P2_]] := Module[{node, r1, t1, r2, t2},
   node = PToString[paritem] <> "   " <> ToString[Unique[]];
   {r1, t1} = ConvertTree[P1]; {r2, t2} = ConvertTree[P2];
   {node, {{node → r1, "//"}, {node → r2, eatpat}, Sequence @@ t1, Sequence @@ t2}}
  ];
```

Appendix B : Code Module

```
SetAttributes[Code, HoldAll]

Protect[$Failure]

{$Failure}
```

VarSet handling (SetVars used by AddShift)

```
ClearVar[varset_List, x_] := DeleteCases[varset, Equal[x, _]];

SetVar[varset_List, v : Equal[x_, $VarWasCleared]] := ClearVar[varset, x];
SetVar[varset_List, v : Equal[x_, val_]] := Append[ClearVar[varset, x], v];

SetVars[varset_List, newvalues_List] := Module[{newvarset = varset},
   (newvarset = SetVar[newvarset, #]) & /@ newvalues;
   newvarset
  ];

SetEnv[varset_] := (Set @@ #) & /@ varset;

SetAttributes[ClearEnv, HoldAll]

ClearEnv[varlist_] := Clear[varlist];

SetAttributes[GetEnv, HoldAll]

GetEnv[varlist_] := Module[{values},
   values = varlist; Clear @@ varlist; res = {varlist, values}
  ];
```

GetVars

```
GetVars[snippet_] := Module[{leaves, vars},
  leaves = Cases[snippet, _, {-1}];
  (*Union filters out duplicates*)
  (*Union[Select[leaves,StringMatchQ[ToString[#],"$@"]&]]*)
  vars = Union[Select[leaves, StringMatchQ[ToString[#], "$*"] &]];
  Complement[vars, {$Failure}]
 ]
```

CodeEval

```
EstimateAffectedVars[c_Code] := Module[{c1},
  (*DPPrint["estimating affected vars of ",c];*)
  (*We remove rhs of Set elements,
  as Mathematica will try to evaluate it after the repeatreplace...*)
  c1 = c //. {Set → AssignOpera, AssignOpera[x766_, y_] → ChangedVar[x766],
     CompoundExpression → blabla, Clear → ChangedVar,
     ChangedVar[x766__] → Changed[Hold[x766]]};
  GetVars[Cases[c1, Changed[__], Infinity]]
 ]
```

■ CodeEval

```
CodeEval[startvars_List, snippet_Code] :=
 CodeEval[startvars, snippet, EstimateAffectedVars[snippet]]


CodeEval[startvars_List, snippet_Code, affectedvars_List] :=
 Module[{allvars, newvars, res},
  allvars = GetVars[{startvars, snippet}];
  DPPrint[CodeEvalLocal1[allvars, startvars, snippet, affectedvars]];
  DPPrint["$CurrentTerm=", $CurrentTerm];
  {res, newvars} = Check[CodeEvalLocal[allvars, startvars, snippet, affectedvars],
    Print["Problem with strategy. Code block is incorrect. Intercepted and
       returning $Failure instead. Code block:", snippet]; {$Failure, {}}];
  {res, SetVars[startvars, MapThread[Equal, {affectedvars, newvars}]]}
 ]

SetAttributes[checkvalue, HoldAll]

checkvalue[x_] := If[ValueQ[x], x, $VarWasCleared]

CodeEvalLocal[allvars_, startvars_List, snippet_Code, affectedvars_] := Module[allvars,
  (Set @@ #) & /@ startvars; (*set vars to init value*)
  (*run snippet and return var values*)
  (*Note, we rely here on strict leftright eval order
   of Mathematica to get this job done without using new local vars*)
  {Evaluate @@ snippet, ReleaseHold[Map[checkvalue, Hold[affectedvars], {-1}]]}
 ]
```

Appendix C: Unification

```
(*DUPrint[x__]:=Print[x];*)(*Debug printer for unify*)

Is$Var[t_] := StringMatchQ[ToString[t], "$@"]
```

Unify

```
Unify[s_, t_] :=
  Catch[Unify1[s /. x733_?Is$Var → Att[x733], t /. x733_?Is$Var → Att[x733], {}]];

Unify1[s1_, t1_, σ_] := Module[{σ1, s, t, n},
   DUPrint["s1=", s1, " t1=", t1, " σ=", σ];
   σ1 = Rule @@ # & /@ σ;
   DUPrint["σ1=", σ1];
   s = s1 //. σ1; t = t1 //. σ1;
   DUPrint["After substi: s=", s, ",t=", t];
   Which[
    (*case1: s=var & s==t*)
    Head[s] === Att && s === t, σ,
    (*case2: s,t both not a variable, so atom or function*)
    (*We check atoms separately, otherwise 3 and 5 will match*)
    AtomQ[s] && AtomQ[t], If[s === t, σ, Throw[$Failure]],
    Head[s] =!= Att && Head[t] =!= Att,
    If[Head[s] === Head[t] && Length[s] == Length[t],
     σ1 = σ;
     For[n = 1, n ≤ Length[s], n++, σ1 = Unify1[s[[n]], t[[n]], σ1]]; σ1,
     Throw[$Failure]
    ],
    (*case 3: s not a var (t might be)*)
    Head[s] =!= Att, Unify1[t, s, σ],
    (*case 4: s occurs in t*)
    Position[t, s] =!= {}, Throw[$Failure],
    (*Case 5: s is var, unify with t*)
    True,
    Append[σ, {s, t}]
   ]
  ];
```

UnifyInherit

```
RvarOnLeft[{Pat[R, rest__], more_}] := True;
```

```
UnifyInherit[s_, t_] := Module[{s1, s2, res, relevant, ass},
   DUPrint["UnifyInherit:", s, " with ", t];
   s1 = s /. x77_?Is$Var → Att[L, x77];
   t1 = t /. x77_?Is$Var → Att[R, x77];
   DUPrint["match ", s1, " ", t1];
   res = Catch[Unify1[s1, t1, {}]];
   DUPrint["unify gave ", res];
   If[res === $Failure, res,
    (*Check that all *)
    (*filter relevant but in wrong form and bring them in right form*)
    relevant = Union[Cases[res, {_, Att[R, __]}], Cases[res, {Att[R, __],
        _}]];
    (*Swap a rule if left hand is not the assignment to the Pat[R] variable*)
    relevant = (If[MatchQ[#, {Att[R, __], __}], #, {#[[2]], #[[1]]}] & /@ relevant);
    (*Drop cases where RHS still contains LHS variable.*)
    (*Dropping restrictions means we search too much *)
    relevant = Select[relevant, Position[#[[2]], Att[L, __]] == {} &];
    (*Print["relevant=",relevant];*)
    ass = (relevant /. Att[R, bla__] → bla);
    Equal @@ # & /@ ass
   ]
  ];
```

UnifySynthesize

```
TryRemoveRHS[{a___, {Att[R, v1__], Att[L, v2__]}, b___}] :=
  TryRemoveRHS[{a, {Att[L, v2], Att[R, v1]}, b}];
TryRemoveRHS[{a___, {Att[R, v1__], anything_}, b___}] :=
  TryRemoveRHS[{a, b} //. Att[R, v1] -> anything];
TryRemoveRHS[{a___, {Att[L, v1__], Att[R, v2__]}, b___}] :=
  TryRemoveRHS[{a, b} /. Att[R, v2] → Att[L, v1]];
TryRemoveRHS[finish_] :=
 If[Position[finish, Att[R, __]] =!= {}, $Failure, finish]
```

UnifySynthesize

```
UnifySynthesize[s_, t_] := Module[{s1, s2, res, relevant, ass},
   s1 = s /. x77_?Is$Var → Att[L, x77];
   t1 = t /. x77_?Is$Var → Att[R, x77];
   res = Catch[Unify1[s1, t1, {}]];
   DUPrint["res=", res];
   If[res === $Failure, res,
    (*filter relevant but in wrong form and bring them in right form*)
    (*relevant=Union[Cases[res,{_,Att[L,__]}], Cases[res,{Att[L,__],
        _}]];*)
    relevant = res;
    (*Swap an assignment if left hand is not the assignment to the Pat[L] variable*)
    (*relevant=(If[MatchQ[#,{Att[L,__],__}],#,{#[[2]],#[[1]]}]& /@ relevant);*)
    relevant = TryRemoveRHS[relevant];
    DUPrint["after tryremoveRHS we have ", relevant];
    (*relevant may be $Failure, but subsequent will not change that*)
    ass = relevant /. Att[L, bla__] → bla;
    Equal @@ # & /@ ass
   ]
  ];
```