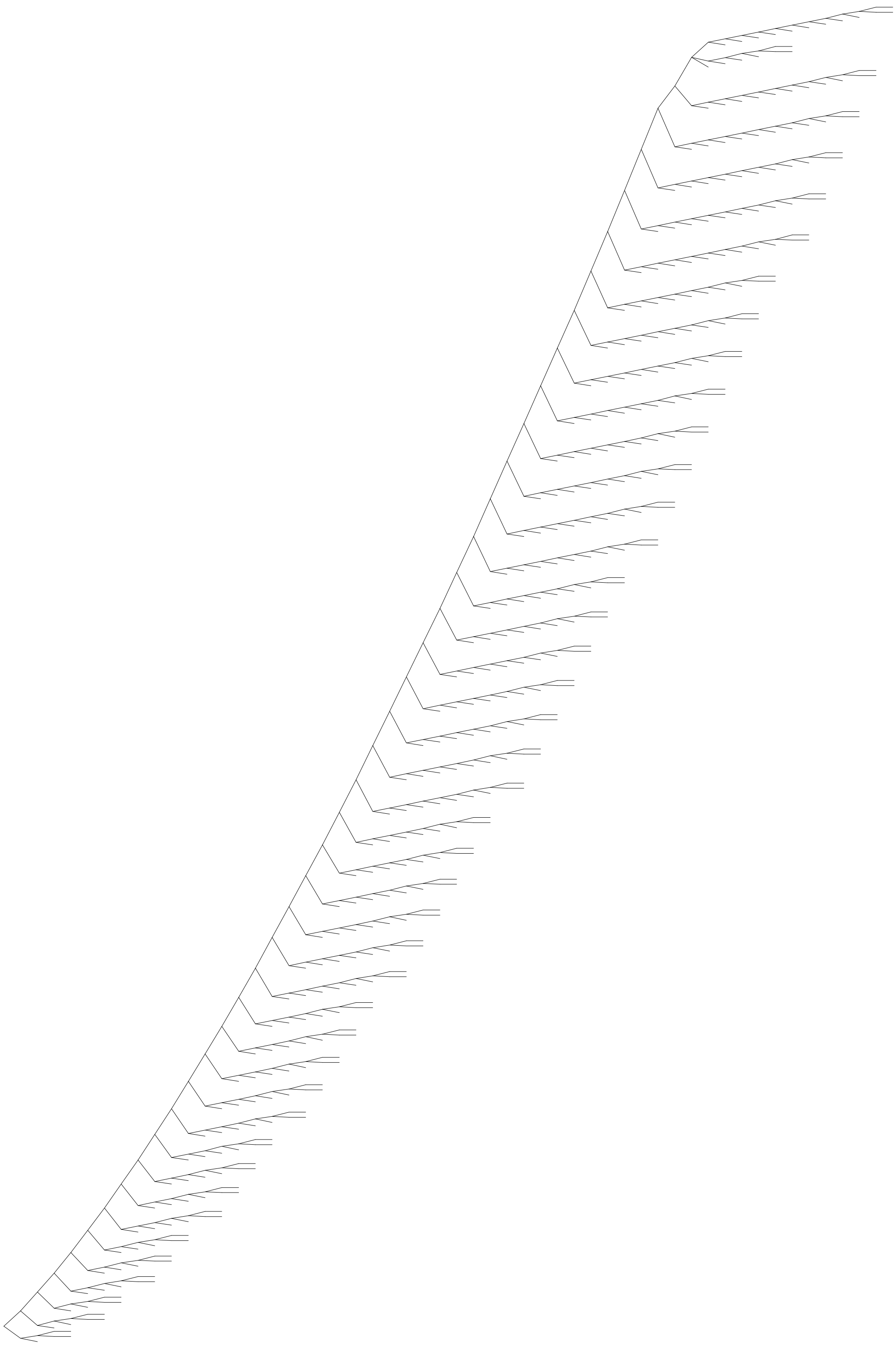
Part IV

Appendix



S T T (**S** T T)
after 50 reductions

Appendix A

RX User Manual

A.1 Introduction

This chapter describes the program RX, written within the framework of this thesis. RX handles rational tree languages in term algebras with arbitrary signatures. RX performs the usual operations on rational languages.

RX has been used to guess, and later verify, a lot of the relations between languages that are used in the decision procedure for $\mathbf{CL}(\mathbf{S})$, and the grammar for normalizing terms, given in chapters 8 and 9 of this thesis.

Furthermore, RX's input and output behaviour can be customized so that it integrates smoothly with a typesetting system, much in the spirit of a *literate programming* tool. RX input may be contained in a text document, and RX output may be embedded in such a document. Additionally, RX output can contain formatting instructions to the typesetter. In fact RX is used to format the very thesis you are now reading.

A.2 RX internals

RX internally represents rational tree languages by finite tree automata. They can be non-deterministic (bottom-up or top-down) or deterministic (bottom-up).

For details of the underlying theory, see chapter 4.

The external form, used for input and output, is a non-deterministic top-down automaton that could also be called a rational grammar.

RX is written entirely in the pure, lazy functional programming language Haskell. It uses Haskell-1.3 features. RX is known to work with these compilers/interpreters: hugs, ghc-2.02, hbc-0.9999.4. It does not depend on compiler-specific extensions or libraries.

RX uses publically available libraries for

1. Finite Maps and Sets
2. Parsing
3. Pretty Printing

(Copies of these are included with the RX source code distribution.) These libraries helped me a lot when programming RX, as they provide tried and tested and optimized solutions for standard tasks, and allowed me to focus on finite automata programming.

The source code of RX, and some additional information, is available under terms of the GNU public license from `http://www5.informatik.uni-jena.de/~joe/rx/`. [1]

A.3 RX syntax

RX lexically and syntactically borrows from the Haskell programming language.

RX lexic

RX code consists of lines of characters.

The input may contain comments. A comment extends from `--` to the end of the line, or it is enclosed in between `{-` and `-}`.

RX has these lexical tokens:

1. number (a sequence of digits)
2. name (an alphabetic character, followed by a sequence of alphanumeric characters)
3. operator (a sequence of non-alphanumeric non-blank characters)
4. string (enclosed in double quotes)
5. group symbols `(`, `)`, `[`, `]`, `{`, `}`
6. group separator `,` (comma). Note that `;` (semi-colon) is not a separator but an operator.

[1] now `http://www.informatik.uni-leipzig.de/~joe/rx/`

The names `arity` and `form` are reserved.

A name may contain characters `"_"`, `"^"`, and `"'"`, and they'll have their expected interpretation when the output is embedded in a LaTeX document.

Names and operators together are called *identifiers*.

A name enclosed in back-quotes behaves as if it were an operator. From now on, *operator* encompasses that meaning.

An operator enclosed in parentheses behaves as if it were a name. From now on, *name* encompasses that meaning.

`RX` syntax

`RX` code is a sequence of *declarations* and *expressions*.

A *declaration* declares one of these things:

1. the form of an identifier (see below),

2. the arity (number of required arguments) of names.

   Example: `arity 2 power, times`

3. the precedence and associativity of operators.

   Example:

   ```
   infixl 7 *
   infixl 5 +
   ```

An *expression* is a sequence of *clauses* separated by operators.

A *clause* is a name followed by a sequence of *atoms*.

An *atom* is a name or a *group*.

A *group* is a sequence of expressions, separated by group separators, enclosed in group symbols. The group `(x)`, where `x` is an expression, is semantically equivalent to `x` alone.

`RX` special syntax

In `RX`, there is no partial application and there are no higher order functions. In a function application, the function must get exactly the number of arguments that its arity requires.

Arities that have not been explicitly declared (see above) are implicitly set to 0 for names and 2 for operators.

This is more restrictive that Haskell but it allows for a nice shorthand notation: the operator `@` (called *apply*) is implicitly inserted whenever a function has *more* arguments than necessary. Let `f` have arity 2, then `f x y b c` is parsed as `((f x y) @ b) @ c`. This corresponds to the convention used in combinatory logic and lambda calculus.

This behavior can be controlled by the switch `implicit` and by the option `apply`. If `implicit` is `on`, then the binary operator that is the value of the option `apply` is inserted. Normally, this is `@`.

A.4 RX semantics

In RX, the semantics of each identifier is a function from state × sequences of values to state × values. The length of the sequence is the arity of the identifier.

Currently, the values are finite tree automata, but this could be extended to a wider range of types.

The state is just the mapping from identifiers to their meaning (semantics). The pair (identifier, meaning) is called binding.

The state map can be extended by new bindings, but existing meanings cannot be overridden.

The operator `=` adds a new binding. Examples:

```
q = 0
f a b = a + b * b
```

If the arity of the bound identifier is 0 (a constant declaration), then the value of the whole expression is that of the right hand side. If the arity is more than 0 (a function definition), then the value of the whole expression is undefined.

The operator ; combines the evaluation of two expressions sequentially. In `a; b`, first `a` is evaluated, then `b` is evaluated in the state that resulted from the evaluation of `a`. The value of `b` is the final result.

RX special semantics

So far, the description does not contain any hint on the range of values that RX operates on. This modularity is intentional, and could be explored when extending RX or re-using parts of it for other purposes.

But we are going to use RX for tree automata. The will be input and output as rational tree grammars.

First we need to specify the term algebra we're working in. This happens implicitly.

Each expression has a term algebra associated with it that consists of the *constructors* that happen to be contained in the expression's value.

A *constructor* is an identifier that is not bound.

Example:

```
arity 0 S
infixl 60 @
```

This declares two constructors, one nullary, one binary.

A grammar is built by the name `grammar` that gets two arguments: a start expression and a set of rules. Example:

```
grammar x { x -> S, x -> S x + S x x }
```

Remember that this is in fact `x -> S @ x + (S @ x) @ x` by the rule of implicit application.

Each rule has a variable on the left and an *rule expression* on the right. The start expression of the grammar also is an *rule expression*.

A *rule expression* looks like an expression (see above) but there are restrictions on where grammar variables might occur: Starting from the top of the syntax tree of a rule expression, on the path to any grammar variable contained therein, only constructors and `++` operators are allowed. In branches of rule expressions where no grammar variables occur, arbitrary expressions are permitted.

Note that `grammar` does not evaluate its arguments. Rather, it behaves like a LISP special form that gets to see the syntax tree of its argument.

`RX` predefined identifiers

There are predefined operators for rational languages:

1. `++` union
2. `\\` difference
3. `&` intersection

and predefined functions that only transform a language's representation, but keep its meaning (the set of its words).

1. `unify`: unify seemingly identical states in the automaton
2. `useful` only keep reachable and productive states
3. `det` make automaton deterministic
4. `min` make automaton deterministic and then minimal

Evaluation of expressions

If the switch `eval` is `on`, each expression is evaluated and the evaluation may result in state changes (because new bindings were added).

However, if evaluating an inline expression changes the current `RX` state, this change is not promoted to other code blocks or snippets.

That is, assignments cannot be hidden inside the text. Rather, they must be made visible in code blocks. (You can still cheat there, by turning the output off, see below.)

If the switch `exp` is `on`, the input expression is printed. For normal typesetting, you would

```
.set (eval=off,exp=on)
```

If the switch `res` is `on`, the result of evaluating the input is printed. (If evaluation took place at all.)

Preprocessing

The switches `unify,useful,det,min` tell whether the so named functions should be automatically inserted on top of all expressions that are being evaluated. In bindings, this refers to expressions right of the `=`.

The switch `expand` tells whether the input expansion is shown or hidden.

Postprocessing

The result of an evaluation is a finite tree automaton. It is shown as a rational grammar. There are switches that control the appearance of that grammar.

1. `foldconst` rules `x -> e` where `e` does not contain grammar variables, are applied to all other rules, and the deleted.

2. `foldnonrec` rules `x -> e` where `e` does not contain `x`, are applied to all other rules, and the deleted.

3. `hidegrammar` if the rule set is empty, don't say `grammar x {}`. Rather, use `x` instead.

A.5 Controlling RX's behaviour

In addition to the core language, RX has means for preprocessing its input, and specifying aspects of its behaviour.

Options

While processing input, RX keeps a set of *options*.
An *option set* is a mapping from strings to strings. It maps *options* to *values*.
A *switch* is an option that only takes on values of `on` or `off`.
Options can be set by *directives*, or on the command line when invoking RX.
A *directive* starts with a dot as the very first character in a line.
Options can be set globally by

```
.set (opt1=val1, opt2=val2, ...)
```

A `.set` is effective from where it stands to the end of the file it is located in.
Option names and values can be given as RX names or strings.
A value can also refer to another option's value by prepending a dollar sign to it, as in `.set (foo=$bar)`.
Options can be set locally by

```
.begin (opt1=val1,...)
   ...
.end
```

The new option set is effective only in between `.begin` and `.end`.

Default values for options

The file `Defaults.hs` from the RX source distribution contains the option set that is compiled in.
All of those options can be changed, however.

Importing files

RX input can be split over several files. The `.import` directive includes the contents of a file into the input stream.

```
.import file
```

`file` must be the file's complete name as an RX string.

The string `"-"`, when used as a file name, means the standard input stream. This is only really useful on the command line, see below.

It is possible to set some options that are only in effect while the imported file is processed

```
.import (opt1=val1,...) file
```

In any case, none of the possible `.set` commands in the *imported* file has an effect on the options in the *importing* file.

RX command line invocation

When the RX executable is invoked, it analyzes its command line, takes appropriate actions, and writes its output to the standard output stream, from where it may be redirected by other means.

On the command line, you may specify option values and files to read.

Example:

```
RX opt1=val1 opt2=val2 filea opt3=val3 fileb filec -
```

is handled as

```
.set (opt1=val1)
.set (opt2=val2)
.import "filea"
.set (opt3=val3)
.import "fileb"
.import "filec"
.import "-"
```

Note that the last line means "read standard input".

Additionally, RX might understand options for its underlying runtime system. They depend on the compiler that had been used when building RX. A common idiom is

```
RX +RTS -H50M -RTS opt1=val1 ...
```

telling the system to use lots of heap space.

A.6 Literate Programming

A literate document contains text and code. From this, RX generates a text document that has code expressions and possibly their values converted to formatted text.

The switch `output` says whether to generate any output at all. Beware that even if this is off, imports, sets, and evaluations may take place.

RX knows whether it is currently looking at text or code by looking up the value of the option `current`. So a typical code block is

```
.begin(current=code)
   ...
.end
```

The physical lines inside such a block are glued together to form logical lines that are fed into the RX evaluator one by one.

A *logical line* consists of one or more *physical lines*. Each physical line that starts with a non-blank character begins a new logical line. Physical lines that start with a blank character are called *continuation lines*. They are appended to the most recent logical line.

Each logical line must be a syntactically valid RX expression.

Apart from code blocks, there may be code snippets contained inside lines of text.

Example:

```
 ... the expression $S Q1 Q0$ denotes ...
```

Such inline code must be a valid expression. It is also sent to the RX evaluator.

There are two variations of inline code. They differ in whether the separator is omitted or kept. Per default,

```
.set (keepsep="$",omitsep="|")
```

and `$x$` will result in `$y$`, while `|x|` will result in `y`, where `y` denotes the result of typesetting `x`.

The default setting should prove reasonable, and in fact RX inline code mode can be seen as a replacement for LaTeX's math mode.

Typesetting customization

RX has two styles for typesetting its output: `plain` and `latex`.

Both can be assigned independently to the options `code` and `text`.

Example:

```
.set (text=latex,code=plain)
```

tells RX to use `\verb` for inline code and the `{verbatim}` environment for code blocks.

You can assign a special LaTeX expansion to RX identifiers. This will be used when `code=latex`. It is achieved by form declarations, of which there are two kinds:

A *passive form* is just a string that gets printed instead of the identifier's name. Example:

```
.begin(current=code,output=off)
form S = "\textbf{S}"
form (+) = "\cup"
.end
```

Note that form declarations happen inside code, not text.

An *active* form is a formatting macro that takes arguments. Example:

```
.begin(current=code,output=off)
form (/) 2 = "\frac#1#2"
.end
```

In the arguments of the operators, parentheses are inserted so that the output is correctly parseable with the given operator precedences and associativities. This remains unaffected when a passive form is used for an operator.

When an active form is in effect, its arguments are typeset for a very low precedence. So, there will be no outer parentheses, because it is assumed that the form takes care of visual grouping. For instance, `4 / (1 + 3)` will come out as `\frac{4}{1 + 3}`, given the above declaration.

(As an aside, this idea of omitting some parentheses was already present in the `smugweb` system [Wal95].)

Typesetting trees

As seen in this thesis, `RX` can be conveniently used to read and typeset trees. Reading is eased by the *implicit apply* convention which inserts `@` nodes. These nodes can be visually typeset by giving `@` an active form. In this paper we use `tree.sty` written by Edward M. Reingold and Nachum Dershowitz which in turn uses PICTEX.

A.7 RX example

Here is some input that is used in the thesis itself.
First, describe the set of all S-terms.

```
all = grammar t {t -> S ++ t  t}
== grammar x1 {x1 -> x1  x1 ++ S}
```

So `all` is a constant whose value is an automaton that accepts (produces) the set of all terms.
The set of all redexes obviously is

```
redex = S  all  all  all
==
grammar
x0
{x0 -> x1  x0 ++ x1  x1 ++ x1  x2 ++ x1  S ++ x1  x4,
 x1 -> x2  x0 ++ x2  x1 ++ x2  x2 ++ x2  S ++ x2  x4,
 x2 -> S  x0 ++ S  x1 ++ S  x2 ++ S  S ++ S  x4,
 x4
 ->
 x0  x0 ++ x0  x1 ++ x0  x2 ++ x0  S ++ x0  x4 ++ x4  x0 ++ x4  x1
 ++ x4  x2 ++ x4  S ++ x4  x4}
```

Now we define a function with one argument.

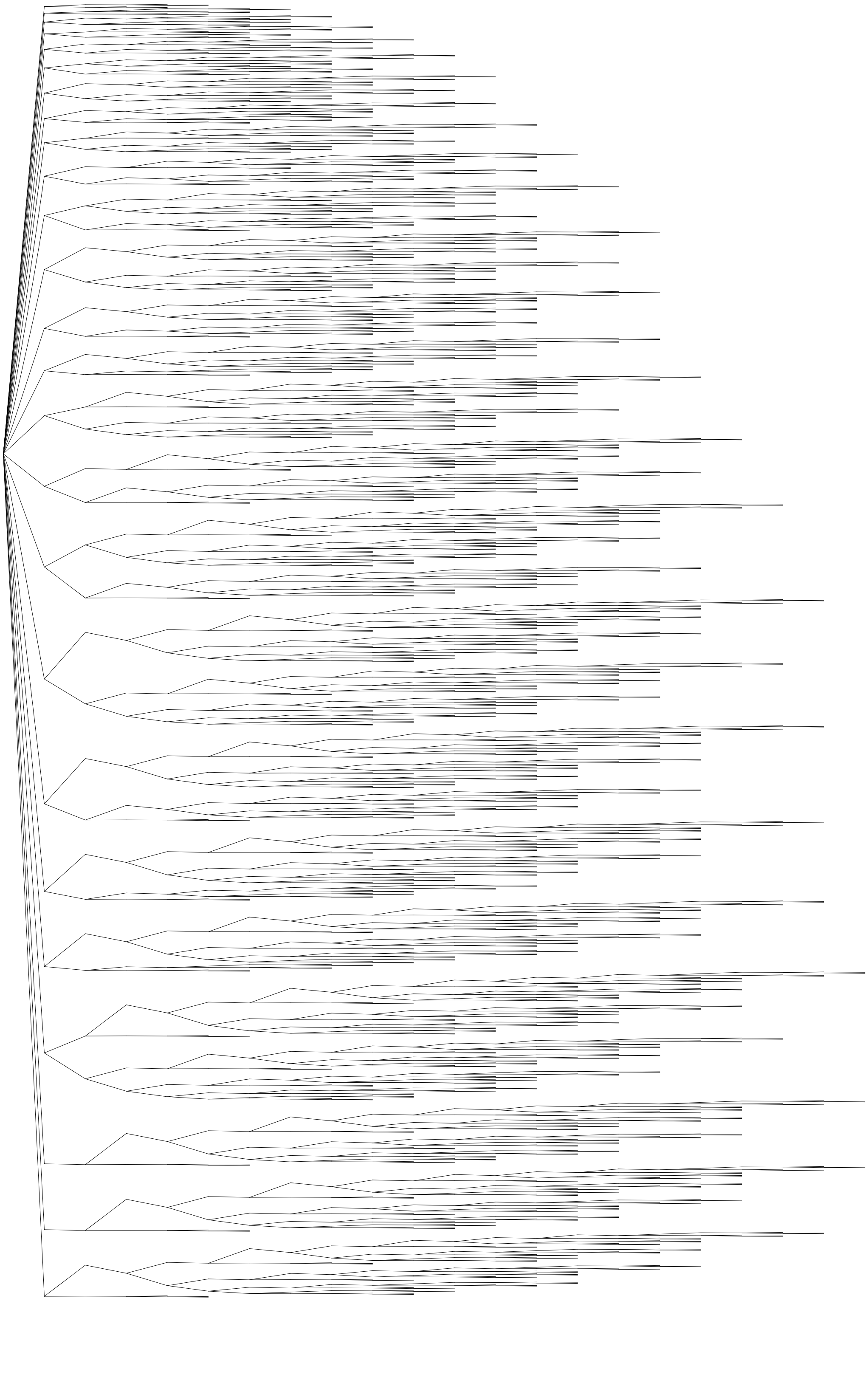
```
contains x = grammar y {y -> x, y -> y  all ++ all  y}
== defined
```

The argument is any language `x`, and the result is the language of all terms `y` that contain `x` as a subterm.
This allows to compute the set of normal forms.

```
normal = all \\ contains redex
==
grammar
(x0 ++ x1 ++ S)
{x0 -> x1  x0 ++ x1  x1 ++ x1  S, x1 -> S  x0 ++ S  x1 ++ S  S}
```

Finally, this could be compared to the representation

```
normal' = grammar t {t -> S, t -> S  t, t -> S  t  t}
==
grammar
(x0 ++ S ++ x2)
{x0 -> S  x0 ++ S  S ++ S  x2, x2 -> x0  x0 ++ x0  S ++ x0  x2}
```



S A S (S A S)
after 50 reductions