# The Construction of an **SASL**-Compiler

Torsten Grust

43/1997

Department of Mathematics and Computer Science
University of Konstanz, Germany

e-mail: `Torsten.Grust@uni-konstanz.de`
web home: `http://www.informatik.uni-konstanz.de/~grust/SASL/`

September 24, 1997

## 1 Introduction

These notes are intended to guide students during the construction of a compiler for a lazy pure functional language. The material covered here includes the lexing, parsing, and parse tree construction phase, as well as the combinator compilation and reduction component of a combinator reduction implementation of such a language.

When there has been the choice between the exhaustive theoretical presentation of a topic or the discussion of the issues of its implementation, we chose the latter. After all it is the goal of this course to get a working compiler within reasonable time. However, topics like grammar transformation (for recursive descent parsing), combinator compilation, and reduction are discussed to a depth that should be sufficient to wake interest for the theoretical foundations.

The students are expected to be familiar with formal language concepts like BNF grammars, production rules, and derivation. The knowledge of basic data structures, such as trees and stacks, is a prerequisite. Experience with functional programming languages is not assumed, though.

Starting with Section 2, the material is intended as a handout to students. These notes have been the successful basis for a second year student's programming course in Konstanz. I make them available in the hope that they turn out to be useful or might speed up the preparation of a similar assignment.

**SASL.** In contrast to various other compiler construction courses, we did not invent SASL as a toy language specific to this course. SASL (St. Andrews Static Language) has been developed by Prof. David Turner in 1976 when Turner was with St. Andrews University, Scotland. Turner published a series on articles on the language, and one of these reports [Tur79] provides the basis this course is built on. Apart from the lexer and parser modules (i.e. the complete syntactical analysis phase), this article describes all techniques needed to compile and execute SASL programs. In this article, Turner established a technique called *combinator reduction*, variants of which are still successfully employed in the translation of modern functional languages like Haskell or Miranda.

The combinator reduction technique is especially well suited for this course, because the SASL compiler does not produce machine level code for some specific CPU. Rather, a reduction machine—realized in software—is used, to perform a stepwise transformation of the *graph representation* of the compiled program, until the final result is obtained. SASL is particularly *lazy* in performing these transformations, a fact which lends interesting semantics to the language. In SASL, it is perfectly reasonable to define functions that compute *infinite* lists (the list of all prime numbers, say). As long as the final result may be computed by examining only a finite number of list elements, the runtime system faces no problems. The graph representaion of compiled programs is crucial for combinator reduction. This will have an impact on the programs that have to be constructed during the course. The construction and transformation of these graphs will be a dominating task—the parser will be responsible to compile an SASL source text into its corresponding graph representation, while the reduction machine will implement program execution by means of graph transformations (very often, it will reduce the graph to one single node).

## 1.1 Implementation Language

At the University of Konstanz, students implemented the SASL compiler using the object-oriented language Eiffel, which has been the first-year language for the students. This course description does not depend in any way on a specific implementation language, although the few small code examples given use Eiffel syntax. However, we are largely using the simple procedural elements of Eiffel only, so that the code snippets should be easy to comprehend. The students were referred to Bertrand Meyer's "classic" *Eiffel – The Language* [Mey92] as a reference. No further Eiffel material has been provided.

## 1.2 Running the Course

In order to complete the assignment in due time, students should not work as sole fighters. In Konstanz, two students made up a development group each. Using the overall architecture of the compiler as an orientation, we proposed to cut the work to be done roughly in half, as Figure 1 suggests:

The dashed line in this drawing suggests that the two halves are not independent of each other. The two co-workers were forced to exchange their ideas and plans in great detail. This was especially true when it came to the design of the graph representation data structure which runs like the red thread through the whole project.

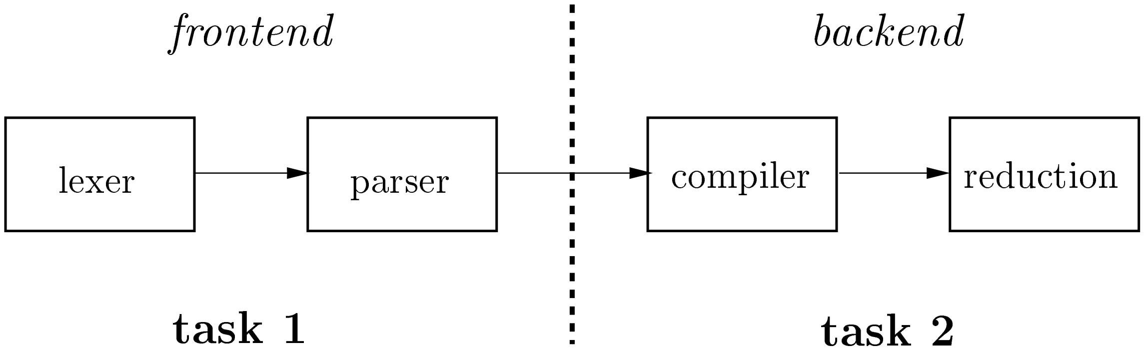



Figure 1: The four stages of the SASL compiler.

When the project started, the co-workers commited themselves to work on either the *frontend* part (lexer, parser) or the *backend* (compiler, reduction machine) of the compiler. During the course, however, it was perfectly okay for the students to "switch sides" and help out if needed.

### 1.3 Time Schedule

The course was scheduled to run for 12 weeks. After two to three weeks the students were expected to finish their initial planning and design phase. The actual coding phase then lasted for the rest of the time.

### 1.4 Organization of this Document

Material that is suitable to hand out to students starts with Section 2, in which we introduce SASL to the depth needed: while types are only marginally mentioned (we will not dicuss the implementation of a type checker for SASL here), we review expressions and function application in their gory details. Global and local definitions are covered in their own subsections, since these pose some challenge during the backend implementation. An SASL grammar given in extended Backus-Naur form completes this part.

Section 3 then walks through the four compiler stages we outlined in Figure 1. Basic techniques and mechanisms are introduced, including grammar transformations for LL(1) parsing, recursive descent parsers, **SK** combinator reduction, and the corresponding reduction machine. Optional optimzations to the **SK** reduction machine are finally presented in Section 4.

Two appendices conclude this paper: Appendix A contains a small library of functions intended to serve as a small SASL prelude. Appendix B presents the above mentioned SASL grammar in a form more suitable for the impatient parser implementor.

The material presented here is intended to be closed with respect to references to further literature. However, reading (parts of) David Turner's seminal original SASL paper [Tur79] as well as Simon Peyton-Jones' excellent book on implementing lazy functional programming languages [PJ87] is always a pleasure and may turn out to be very useful.

**Acknowledgements.** I am particularly grateful for the feedback the students of the course "Informatik IV" (winter 1996/97) provided me with. Special thanks go to Andres Löh for his useful remarks and considerations. Finally, I would like to thank Dave Turner for his immediate and most helpful response to a question of mine.

# 2 The Functional Programing Language SASL

As almost all functional languages, SASL is a conceptually simple and elegant programming language. In fact, it is so simple that we can introduce the language completely in this section. It is not necessary to be fluent in any other functional language to follow the upcoming introduction of SASL. The *definition* and *application of functions* is the single major concept in SASL. The term *functional programming language* stems from this observation.

Typing is not our primary concern here but we will start with a few words on types of expressions. Expressions are built from SASL's built-in operators as well as globally and locally user-defined functions. These concepts are covered next. We conclude this section by giving a grammar for SASL programs so that we account for SASL's syntax, too.

## 2.1 Data Types

**Atomic types.** SASL provides `num` (negative and positive integers), `bool` (consisting of the two values `true` and `false` only), and `string` (finite sequences of characters enclosed in `""`) as standard data types.

**Type constructors.** SASL allows for the construction of lists that may be built from values of *arbitrary types*. This is a relaxation of the typing rules of most functional programming languages which require lists to be built of values of a single type only. Consequently, there is no type constructor $\alpha$ `list` in our implementation of SASL since, in general, the list elements will not agree on a single type $\alpha$. Rather, we will simply assign the type `list` to any value constructed with the list constructors `:` and `nil` (see Subsection 2.2 on expressions below).

Note that this is *not* the way real-world functional programming languages go about the problem. We can live with this simplified view of things because we are not concerned with type checking at all.

Functions map arguments to function result; we will denote their types as $\tau_1$ `->` $\tau_2$, where $\tau_1$ and $\tau_2$ are the types of the argument and the result, respectively. For example, the function *is_even* has type `num -> bool`.

## 2.2 Expressions

**Operators.** SASL is equipped with the following arithmetic operators which only have sensible definitions if applied to arguments of type `num` (in the following $a$, $b$, and $c$ represent expressions of type `num`):

| | |
|---|---|
| $a$+$b$ | addition |
| +$a$ | unary plus |
| $a$-$b$ | subtraction |
| -$a$ | unary minus |
| $a$*$b$ | multiplication |
| $a$/$b$ | integer division (5/3 $=$ 1) |

The relational operators listed in the following table may be applied to values of type `num` and `string`, `=` and `~=` are also applicable to `bool`eans or `list` values:

| | |
|---|---|
| $a$=$b$ | equality |
| $a$~=$b$ | inequality |
| $a$<$b$ | less than |
| $a$>$b$ | greater than |
| $a$<=$b$ | less than or equal |
| $a$>=$b$ | greater than or equal |

The `bool`ean operators are listed below:

| | |
|---|---|
| `not` $a$ | logical negation |
| $a$ `and` $b$ | conjunction |
| $a$ `or` $b$ | disjunction |
| `if` $a$ `then` $b$ `else` $c$ | conditional expression |

Please note that `if` $a$ `then` $b$ `else` $c$ is to be understood as an *expression* and not as a *statement*. An expression always has a unique value. The value of `if` $a$ `then` $b$ `else` $c$ is $b$ if $a$ `= true`, and $c$ otherwise. As a consequence, there is no conditional expression of the form `if` $a$ `then` $b$, which would be undefined for the case $a$ `= false`.

The infix operator `:` (read: *cons*) must be used to construct lists. The expression `x:xs` builds a new list by prepending the element `x` to the already existing list `xs`. The constant value `nil` denotes the empty list.

Examples:

- `1:nil` – a list containing element `1` only.
- `true:false:true:nil` – list of three elements of type `bool`; `:` associates to the right, so that the expression is equivalent to `true:(false:(true:nil))`.
- In SASL, you may use the abbreviation [$a$,$b$,$c$] for the expression $a$:$b$:$c$:`nil`. Thus, we can equivalently write the above examples as `[1]` resp. `[true,false,true]`. Lists may contains lists as elements, making `[[1,2],nil,[[true]],["a","b"]]` a valid expression. The empty list `nil` may be alternatively written as `[]`.

If $a$ is an expression then ($a$) is an expression as well (with the same value as $a$). You may use parentheses to circumvent the default operator priorities and associativites. We list the priorities here:

| priority | operator |
|---|---|
| 8 | `f a` (*juxtaposition*, see below) |
| 7 | `not + -` (prefix) |
| 6 | `* /` |
| 5 | `- +` (infix) |
| 4 | `= ~= < > <= >=` |
| 3 | `and` |
| 2 | `or` |
| 1 | `:` |
| 0 | `if·then·else` |

All binary operators associate to the left, with `:` (*cons*) being the only exception. See above.

A few examples of valid expressions (and the values SASL evaluates them to):

```
2+3                                           5
if true then 42 else 0                        42
1 : if not ("a" < "b") then [2,3] else nil    [1]
-(4+2)*3                                      −18
```

**Function application.** Applying a function is the major operation in SASL. Therefore, we represent application by the simplest syntactic construct we can provide: the function and its argument are simply written in sequence (*juxtaposition*). $f\,a$ represents the application of $f$ to $a$, where $f$ is a function-valued expression (e.g. a function name) and $a$ an arbitrary expression. Function application has the highest operator priority.

The former paragraph talked about an expression $f$ having a *function* as its value. Expressions of this kind are unique to functional programming languages. SASL treats functions like ordinary values, that is, there is no essential difference between function values and numbers or strings. Functional programming languages draw their power and elegance from this fact. A function may be argument to another function, functions may return functions (such functions are called *higher-order functions*), you may build lists containing functions, etc. You can even compute a function on the fly and then pass this new function value around in your program. This is essentially different from imperative programming languages, where you can define and then call a function; you may not, however, assign a function to a variable for example.

A simple example is (let `f` and `g` be function values, e.g. names of user-defined functions)

```
(if 1 ~= 0 then f else g) 42
```

in which we apply a function, namely `if 1 ~= 0 then f else g`, to the argument `42`. The `if·then·else` expression itself evaluates to a function, `f` in this case, which in turn is then applied to `42`. We will have a closer look at examples of this flavour and you will quickly get used to it.

In addition to the SASL built-ins, functions may be defined by the programmer using the keywords `def` and `where` (see below). The operators `+`, `-`, `<`, `or`, ... are pre-defined functions built into SASL. The syntax allows you to use these in the more operator-wise infix notation, which exceptionally breaks the juxtaposition rule.

**Currying.** Every SASL function may take more than one argument. It may be instructive to think about an application like $f\,a\,b$ for a minute. What is actually meant here? Since juxtaposition is just another operator (associating to the left as explained above), we have to read the above expression as $(f\,a)\,b$ : $f\,a$ is evaluated to a function which in turn is applied to the argument $b$. This technique, well-known as *Currying*, makes use of SASL's higher-order functions: $f$ is a function that, when applied to an argument (here $a$), returns a function value (which we can apply to $b$).

The type of $f$ is $\alpha$ `->` $(\beta$ `->` $\gamma)$ if $\alpha$, $\beta$, and $\gamma$ denote the types of $a$, $b$, and the result's type, respectively. We provide one argument $a$ of type $\alpha$ and obtain a function of type $\beta$ `->` $\gamma$. By applying this function to the "second" argument $b$ we finally get the result of type $\gamma$.

An example (with a leap ahead to `def`). A programmer defines the function `plus` as

```
def plus x y = x+y
```

The currying principle instructs us to read an expression like `plus 2 3` as `(plus 2) 3`. However, the expression `(plus 2)` itself already has a sensible value, namely the function that adds 2 to its argument. You can intuitively see that if you replace the variable `x` by `2` in the function body of `plus`. The complete expressions `plus 2 3` therefore evaluates to 5. The type of `plus` unquestionable is `num -> (num -> num)`. Nobody stops us from defining

```
def incr = plus 1
```

Given this, `incr` (*increment*) is the function that adds 1 to its argument, e.g. `incr 6` evaluates to 7. `incr`'s type is `num -> num`, as expected.

Being so far, let us introduce a graphical notation that will turn out to be useful during the whole course.

We will represent SASL expressions as binary trees. The leafs are labeled with values of any type (`num`, `bool`, `string`, but also functions). An inner node always represents a function application, indicated by an `@` sign. The expression $f\,a$ is drawn as

```
    @
   / \
  f   a
```

We will sometimes denote the above tree as $f$ `@` $a$ (read: $f$ at $a$). Our recent example `(plus 2) 3` consequently is represented by the tree

```
          @
         / \
        @   3
       / \
   plus   2
```

More general, $f\,x_1\,x_2\,\cdots\,x_n$ has the presentation

```
                @
               / \
              @   x_n
             / \
            @   ...
           / \
          @   x_2
         / \
        f   x_1
```

Tree structures of this kind will be generated by the parser and then manipulated by the reduction machine. Section 3 will cover these issues to the depth needed.

### 2.3 Global Definitions (`def`)

The programmer may assign globally visible names to values via a series of `def` definitions (remember that values may be booleans, strings, and numbers, but also functions). The last `def` definition has to be terminated by a dot '`.`'. A subsequent expression may then refer to any global name defined. Any SASL program comprises an optional sequence of `def`s followed by a single expression to be evaluated.

Some examples involving `def`:

```
def answer   = double 21
def list     = [1,2,3,4]
def double x = 2*x
def twice    = double.
```

A definition may refer to names that are introduced later on (see `double` in the definition of `answer`; there is no "forward" declaration or the like). The last definition makes `twice` an alias for the function `double` defined earlier. `twice 2` evaluates to 4.

`def` may be (mutually) recursive:

```
def fac n = if n=0 then 1 else n * fac (n-1)
def one   = 1 : two
def two   = 2 : one.
```

While the definition of `fac` should be obvious, an evaluation of the expression `one` results in the *infinite* list $[1, 2, 1, 2, 1, 2, 1, \ldots$ . Thanks to SASL's semantics of expression evaluation (which is performed *lazily*) we can actually do sensible operations on such "infinitely large" values. Section 3.4 comments on that.

### 2.4 Local Definitions (`where`)

Every expression may be followed by local definitions that introduce names whose visibility is restricted to just this expression. Local definitions are introduced by the keyword **`where`**, multiple definitions are seperated by a semicolon (`;`). Arbitrary (mutual) recursion is allowed, just like in the case of **`def`**.

Examples:

```
     x where x = 3                          3
   x+y where x = 3;                         9
             y = 2*x
answer where answer = double 21;            42
                      double x = 2*x
double 2                                    error: double not visible
```

### 2.5 Predefined Functions

Our somewhat restricted version of the SASL language will further predefine two functions: **`hd`** (*head*) and **`tl`** (*tail*). Both operate on lists and are defined as follows:

$$\begin{array}{rcl} \texttt{hd}\ (x\!:\!xs) & = & x \\ \texttt{tl}\ (x\!:\!xs) & = & xs \end{array}$$

Both functions are allowed to return anything (read: are undefined) if applied to an empty list. By virtue of **`hd`** and **`tl`** we are able to take apart a non-empty list into its head and tail again. Together with the list constructors `:` and **`nil`** we now have a complete "list toolbox" which allows us to define any list operation we might wish for.

Example (of a function that determines the first **`n`** elements of a list **`l`**):

```
def first n l = if n=0 or l=nil then nil
                                else x:(first (n-1) xs)
                where x  = hd l;
                      xs = tl l.
first 2 [1,2,3,4]                                              [1,2]
```

**Note.** SASL's *lazy* evaluation enables us to actually compute the value of e.g. `first 3 one` and to print the correct result $[1, 2, 1]$ (remember the definition of the infinite value `one` from above). A non-lazy programming language would rather try to evaluate all arguments to the function `first` before `first` itself is applied. The argument `3` poses no problems, the evaluation of `one` takes "forever", however. These languages (normally attributed as being *strict* or *eager*; all imperative languages like Pascal or C belong to this class) are unable to evaluate expressions involving infinite values like `first 3 one`. SASL's laziness is one of the core virtues of the language.

If your SASL implementation project works out well, the library of predefined functions is a good point where you can extend your compiler. But see Appendix A which describes a minimal SASL *prelude*.

## 2.6 An SASL Grammar in EBNF

Table 1 displays the syntax of the SASL subset which is relevant to this course. It is given in EBNF (extended Backus-Naur form). You should scan through the grammar rules to get a clear concept of what parts of the grammar represent specific parts of the SASL syntax we described only informally up to now. The nonterminal ⟨*list*⟩, for example, describes the handy abbreviating list notation using square brackets `[·]`.

A sequence of characters is a syntactically correct SASL program if it can be derived from the grammar's start symbol ⟨*system*⟩. Grammars, rules, and derivation are concepts you already should be familiar with. There are programs derivable from ⟨*system*⟩, however, which do not make much sense. We may derive `3*true` for example. In a real-world compiler, typing flaws like this would have been detected by the the compiler's *type checker*.

This course is not concerned with type checking in order to focus on the actual compilation matters. Your compiler is allowed to do anything with a program like `3*true`. It may choke or dump core—it may, however, also print a sensible error message *if* you implement a type check on your own; Chapter 9 of [PJ87] covers the type checking phase. Otherwise, we follow the *gigo principle* here (*garbage in—garbage out*).

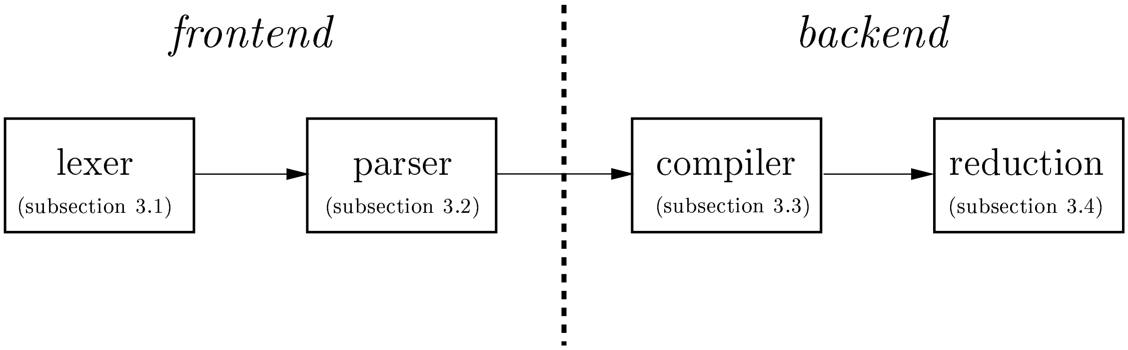
Notes and a series of examples on the SASL language itself may be found in one of the original references [Tur79]. In these course notes you should only find minor syntactical differences from what is described there (e.g. we substituted the somewhat more familiar form `if·then·else` for Turner's conditional expression operator `·->·;·`).

$$
\begin{array}{rcl}
\langle \mathit{system} \rangle & \rightarrow & \langle \mathit{funcdefs} \rangle\ \texttt{.}\ \langle \mathit{expr} \rangle \\
& | & \langle \mathit{expr} \rangle \\
\langle \mathit{funcdefs} \rangle & \rightarrow & \texttt{def}\ \langle \mathit{def} \rangle \\
& | & \langle \mathit{funcdefs} \rangle\ \texttt{def}\ \langle \mathit{def} \rangle \\
\langle \mathit{defs} \rangle & \rightarrow & \langle \mathit{def} \rangle \\
& | & \langle \mathit{def} \rangle\ \texttt{;}\ \langle \mathit{defs} \rangle \\
\langle \mathit{def} \rangle & \rightarrow & \langle \mathit{name} \rangle\ \langle \mathit{abstraction} \rangle \\
\langle \mathit{abstraction} \rangle & \rightarrow & \texttt{=}\ \langle \mathit{expr} \rangle \\
& | & \langle \mathit{name} \rangle\ \langle \mathit{abstraction} \rangle \\
\langle \mathit{expr} \rangle & \rightarrow & \langle \mathit{expr} \rangle\ \texttt{where}\ \langle \mathit{defs} \rangle \\
& | & \langle \mathit{condexpr} \rangle \\
\langle \mathit{condexpr} \rangle & \rightarrow & \texttt{if}\ \langle \mathit{expr} \rangle\ \texttt{then}\ \langle \mathit{condexpr} \rangle\ \texttt{else}\ \langle \mathit{condexpr} \rangle \\
& | & \langle \mathit{listexpr} \rangle \\
\langle \mathit{listexpr} \rangle & \rightarrow & \langle \mathit{opexpr} \rangle\ \texttt{:}\ \langle \mathit{listexpr} \rangle \\
& | & \langle \mathit{opexpr} \rangle \\
\langle \mathit{opexpr} \rangle & \rightarrow & \langle \mathit{prefix} \rangle\ \langle \mathit{opexpr} \rangle \\
& | & \langle \mathit{opexpr} \rangle\ \langle \mathit{infix} \rangle\ \langle \mathit{opexpr} \rangle \\
& | & \langle \mathit{comb} \rangle \\
\langle \mathit{comb} \rangle & \rightarrow & \langle \mathit{comb} \rangle\ \langle \mathit{simple} \rangle \\
& | & \langle \mathit{simple} \rangle \\
\langle \mathit{simple} \rangle & \rightarrow & \langle \mathit{name} \rangle \\
& | & \langle \mathit{builtin} \rangle \\
& | & \langle \mathit{constant} \rangle \\
& | & \texttt{(}\ \langle \mathit{expr} \rangle\ \texttt{)} \\
\langle \mathit{name} \rangle & \rightarrow & \langle \mathit{id} \rangle \\
\langle \mathit{builtin} \rangle & \rightarrow & \texttt{hd} \\
& | & \texttt{tl} \\
\langle \mathit{constant} \rangle & \rightarrow & \langle \mathit{num} \rangle \\
& | & \langle \mathit{bool} \rangle \\
& | & \langle \mathit{string} \rangle \\
& | & \texttt{nil} \\
& | & \langle \mathit{list} \rangle \\
\langle \mathit{list} \rangle & \rightarrow & \texttt{[ ]} \\
& | & \texttt{[}\ \langle \mathit{listelems} \rangle\ \texttt{]} \\
\langle \mathit{listelems} \rangle & \rightarrow & \langle \mathit{expr} \rangle \\
& | & \langle \mathit{listelems} \rangle\ \texttt{,}\ \langle \mathit{expr} \rangle \\
\langle \mathit{prefix} \rangle & \rightarrow & \texttt{-} \mid \texttt{+} \mid \texttt{not} \\
\langle \mathit{infix} \rangle & \rightarrow & \texttt{+} \mid \texttt{-} \mid \texttt{*} \mid \texttt{/} \mid \texttt{=} \mid \texttt{\textasciitilde=} \mid \texttt{<} \mid \texttt{>} \mid \texttt{<=} \mid \texttt{>=} \mid \texttt{and} \mid \texttt{or} \\
\langle \mathit{id} \rangle & \rightarrow & [\texttt{a} - \texttt{zA} - \texttt{Z\_}][\texttt{a} - \texttt{zA} - \texttt{Z\_0} - \texttt{9}]^* \\
\langle \mathit{num} \rangle & \rightarrow & [\texttt{0} - \texttt{9}]^+ \\
\langle \mathit{bool} \rangle & \rightarrow & \texttt{true} \mid \texttt{false} \\
\langle \mathit{string} \rangle & \rightarrow & \texttt{"}\langle \mathit{ASCII\ character} \rangle^* \texttt{"}
\end{array}
$$

Table 1: SASL grammar in EBNF

# 3 The Compiler Stages

In what follows, we will have a close look at the several components of the SASL-Compiler. Subsection 3.3 then describes the actual compilation process that translates SASL programs into the tree structure we introduced in the previous section. We will use the previously shown figure as a roadmap for the next few sections:




A special part of the compiler—you may think of some superordinate control, often called the *compiler driver*—will initiate the four compiler stages in sequence and takes care of passing the intermediate compilation results from one phase to the next. In Eiffel, the driver could look similar to the following skeleton:

```
lex   : LEXER
parse : PARSER
comp  : COMPILER
sk    : REDUCER

p     : PARSETREE
c     : COMBINATOR_GRAPH
...

!!lex.make(filename);        -- open the LEXER on the source
!!parse.make(lex);           -- initialize the parser
p := parse.parse;            -- build the parse tree
!!comp.make(p);              -- initialize the compiler
c := comp.compile;           -- compile the parse tree into a graph
!!sk.make(c);                -- initialize the SK reducer
sk.reduce;                   -- start the graph reduction
```

**Invoking the compiler.** Users should give a command like

`sasl` ⟨*sasl-source-file*⟩

to invoke the compiler from the shell. The compiler should print a short but helpful message about its proper usage if the argument is missing or otherwise incorrect (e.g. ⟨*sasl-source-file*⟩ is unreadable). The usage message, like any other warning or error message, should go to the `stderr` (standard error) I/O channel.

The compiler will read the SASL source from the file named ⟨*sasl-source-file*⟩, compile it, and then execute (i.e. reduce) the compiled program. The result of the reduction phase—a single SASL value—is finally output to the `stdout` (standard output) channel. This completes the compiler run.

## 3.1 Lexer

The lexer has the single task to open and read the source file. While it proceeds with reading the file, it converts the stream of characters in a more coarse grained (not character-wise) stream of symbols (*token*) which are then consumed by the parser.

Example: suppose the lexer detects the three character sequence '`d e f`' in its input. The lexer will then produce the single token ◁`def`▷ as its output. The parser operates on tokens only; it does not cope with single characters. Imagine we decide to replace the keyword `def` by `define` later. This replacement affects the lexer only. As long as it produces the ◁`def`▷ token when encountering '`d e f i n e`' we do not have to change a single bit of the parser.

**Token classification.** The following table suggests a classification of the tokens that need to be generated by the lexer:

| token type | example |
|---|---|
| keyword | ◁`def`▷, ◁`if`▷, ◁`where`▷ |
| identifier | ◁`id:"answer"`▷, ◁`id:"x"`▷ |
| constant | ◁`num:`42▷, ◁`bool:`*true*▷, ◁`string:"foo"`▷ |
| symbol | ◁`leq`▷ (<=), ◁`plus`▷ (+) |
| special | ◁`eof`▷ (end of input) |

Some tokens carry additional information with them (such as the value of an integer constant) which can be used by the parser when it comes to the construction of a parse tree for the program. Note that if would be sufficient to map all integer constants to the token ◁`num`▷ if we were only interested in syntax checking. However, this does not suffice if we want to build a complete representation of the program.

Example: the lexer transforms the input '`if 1 ~= 0 then f else g`' into

◁`if`▷ ◁`num:`1▷ ◁`neq`▷ ◁`num:`0▷ ◁`then`▷ ◁`id:"f"`▷ ◁`else`▷ ◁`id:"g"`▷

The interface to the lexer will be as simple as a single routine that delivers the next token on demand. The parser can *look ahead* into the source file by requesting more and more tokens from the lexer. Fortunately, the SASL grammar is simple in the sense that at any time a *lookahead* of just one token is sufficient for the parser to decide what to do next (see below). The last token delivered by the lexer is ◁`eof`▷.

### 3.2 Parser

The parsing phase requests tokens from the lexer (by calling the lexer's lookahead routine) on demand. The output of this phase is a *parse tree*, a tree structure that represents the abstract syntax of the program. In this course, we will employ the *recursive descent* parsing technique. This methods allows for the construction of parsers by translating the BNF grammar's production rules into code in an almost one-to-one manner. However, we need to "massage" the grammar rules before we can deduce the parser from its grammar. We come to this next.

Recursive descent parsers contain exactly one routine for each nonterminal the grammar features (nonterminals are enclosed in $\langle\cdot\rangle$ in Table 1). The routine body implements the right-hand side (rhs) of the corresponding production rule:

- each nonterminal symbol on the rhs is translated into a call to the corresponding parser routine,
- each terminal symbol on the rhs is compared to the next lookahead token. If the comparison succeeds all is well and the parse goes on. On a failed match a syntax error has been detected and the parser should give appropriate diagnostics about the failure. In the code snippets below, the routine `match` takes care of these tasks.

This recipe leads to recursive calls of the parser routines in general, an observation from which the technique's name is deduced.

Example: the production rule

$$\langle \mathit{condexpr} \rangle \quad \rightarrow \quad \texttt{if}\ \langle \mathit{expr} \rangle\ \texttt{then}\ \langle \mathit{condexpr} \rangle\ \texttt{else}\ \langle \mathit{condexpr} \rangle$$

may be implemented as follows:

```
lex : LEXER
...
condexpr is
   do
      match(◁if▷);
      expr;
      match(◁then▷);
      condexpr;
      match(◁else▷);
      condexpr;
   end -- condexpr
```

However, the complete production rule for $\langle \mathit{condexpr} \rangle$ reads

$$
\begin{array}{rcl}
\langle \mathit{condexpr} \rangle & \rightarrow & \texttt{if}\ \langle \mathit{expr} \rangle\ \texttt{then}\ \langle \mathit{condexpr} \rangle\ \texttt{else}\ \langle \mathit{condexpr} \rangle \\
& | & \langle \mathit{listexpr} \rangle
\end{array}
$$

Which of the two alternatives seperated by | is the right one for the parser to follow? The parser resolves this dilemma by looking at the lookahead token. We will transform the grammar to ensure that it is sufficient to look only one token ahead to make the right decision (see *left factorization* below). In our example, the parser has to check for the occurence of an ◁`if`▷ token. The corresponding Eiffel code might look like as follows:

```
condexpr is
   do
      if lex.lookahead = ◁if▷ then
         match(◁if▷);
         expr;
         match(◁then▷);
         condexpr;
         match(◁else▷);
         condexpr
      else
         listexpr
      end
   end -- condexpr
```

Note that in the previous example `condexpr` has been introduced as a routine, which, of course, is not the full truth. Actually it has to be implemented as a function that returns a parse tree representing the conditional expression.

**Constructing parse trees.** An important hint. Construct the parse tree using the explicit function application convention (using `@`) introduced when we discussed currying. Understand SASL's built-in operators as ordinary functions. The expression `2 + 3` should be translated into the parse tree

```
           @
          / \
         @   3
        / \
    plus   2
```

and not—as you might be used to—into

```
   plus
   /  \
  2    3
```

Life in the compilation phase (see Subsection 3.3) is a lot easier if you follow this convention when constructing the trees. Remember currying: how would you represent the partial application `plus 2` using the latter variant? Stick to the explicit function application convention. A final example: translate `if` $a$ `then` $b$ `else` $c$ into the tree

```
             @
            / \
           @   c
          / \
         @   b
        / \
    cond   a
```

i.e. transform the conditional expression into the function application `cond` $a$ $b$ $c$.

**Elimination of left recursion.** For production rules like the one below, we run into problems if we mechanically transform the rhs of the rule into a parser routine:

$$\begin{array}{rcl} \langle \textit{funcdefs} \rangle & \rightarrow & \texttt{def}\ \langle \textit{def} \rangle \\ & | & \langle \textit{funcdefs} \rangle\ \texttt{def}\ \langle \textit{def} \rangle \end{array}$$

Note that the latter alternative on the rhs starts with the nonterminal that is just to be defined! This leads to endless recursive calls of the routine corresponding to $\langle \textit{funcdefs} \rangle$ (verify this by translating the production's rhs into a parser routine body). We resolve this problem by grammar rule rewriting. Rules of the general form ($\alpha$ denotes an arbitrary sequence of (non)terminals, $\beta$ denotes an arbitrary sequence of (non)terminals not starting with $A$)

$$\begin{array}{rcl} A & \rightarrow & A\ \alpha \\ & | & \beta \end{array}$$

are transformed into the equivalent form ($\epsilon$ represents the *empty word* i.e. a zero character sequence)

$$\begin{array}{rcl} A & \rightarrow & \beta\ A' \\ A' & \rightarrow & \alpha\ A' \\ & | & \epsilon \end{array}$$

You can verify that both variants allow the derivation of $\beta\alpha\alpha\cdots\alpha$. For the $\langle \textit{funcdefs} \rangle$ example we obtain

$$\begin{array}{rcl} \langle \textit{funcdefs} \rangle & \rightarrow & \texttt{def}\ \langle \textit{def} \rangle\ \langle \textit{funcdefs'} \rangle \\ \langle \textit{funcdefs'} \rangle & \rightarrow & \texttt{def}\ \langle \textit{def} \rangle\ \langle \textit{funcdefs'} \rangle \\ & | & \epsilon \end{array}$$

after the rewriting, which is readily implemented as

```
funcdefs is
   do
      match(◁def▷);
      def;
      funcdefs1
   end -- funcdefs

funcdefs1 is
   do
      if lex.lookahead = ◁def▷ then
         match(◁def▷);
         def;
         funcdefs1
      else
         -- epsilon
      end
   end -- funcdefs1
```

The $\epsilon$ alternative is always taken as the last resort.

**Left factorization.** The following grammar rule poses another difficulty:

$$\begin{array}{rcl} \langle \mathit{defs} \rangle & \rightarrow & \langle \mathit{def} \rangle \\ & | & \langle \mathit{def} \rangle \ \mathtt{;} \ \langle \mathit{defs} \rangle \end{array}$$

The parser is not able to make the decision for one of the two alternatives before he has seen all the tokens normally consumed by the $\langle \mathit{def} \rangle$ routine (does a ; follow or not?). This is a contradicition to our claim that it is sufficient to look ahead one token only at any time. Once again we are able to equivalently transform the production into a more convenient form for the parser. Rules of the general form

$$\begin{array}{rcl} A & \rightarrow & \alpha\ \beta_1 \\ & | & \alpha\ \beta_2 \end{array}$$

are rewritten into

$$\begin{array}{rcl} A & \rightarrow & \alpha\ A' \\ A' & \rightarrow & \beta_1 \\ & | & \beta_2 \end{array}$$

before we implement the corresponding parser routines for the rhs.

**Lookahead (*first* sets).** We are facing a final complication with productions rules like $\langle factor \rangle$ shown below. $\langle factor \rangle$ is neither left recursive nor a case for left factorization:

$$\begin{array}{rcl} \langle factor \rangle & \rightarrow & \langle prefix \rangle \ \langle comb \rangle \\ & | & \langle comb \rangle \end{array}$$

The problem lies in the fact that $\langle prefix \rangle$ and $\langle comb \rangle$ are *nonterminals*. We cannot compare nonterminals with the current lookahead token directly. We can solve the problem by analyzing $\langle prefix \rangle$:

$$\langle prefix \rangle \quad \rightarrow \quad \texttt{+} \mid \texttt{-} \mid \texttt{not}$$

The parser routine for $\langle factor \rangle$ should obviously choose the first alternative if the lookahead token is `-`, `+`, or `not`. Otherwise the second alternative is the way to go. For this to work correctly, it is crucial that we cannot derive strings starting with `-`, `+`, or `not` from $\langle comb \rangle$ as well because our decision for one of the alternatives would be ambigious then.

To formalize the matter, parser construction theory associates every nonterminal with its so-called *first set* which contains the first token of every token sequence that may derived from that nonterminal. In our case we have $\mathit{first}(\langle prefix \rangle) = \{\triangleleft\texttt{plus}\triangleright, \triangleleft\texttt{minus}\triangleright, \triangleleft\texttt{not}\triangleright\}$ and $\mathit{first}(\langle prefix \rangle) \cap \mathit{first}(\langle comb \rangle) = \emptyset$. The latter is the essential condition that lets us safely decide for one alternative (spend a minute and check this condition on your own).

```
prefixop(t : TOKEN) : BOOLEAN is
   do
      -- check for a token in the set first(⟨prefix⟩)
      Result := t = ◁plus▷ or else t = ◁minus▷ or else t = ◁not▷
   end -- prefixop

factor is
   do
      if prefixop(lex.lookahead) then
         prefix;
         comb
      else
         comb
      end
   end -- factor
```

**Operator precedence.** The $\langle opexpr \rangle$ production rule represents SASL's prefix and infix operator application. Operator precedence (or priority), however, is not reflected by the grammar in Table 1. We need a means of expressing operator precedence in the grammar in order to generate parse trees that reflect the binding power of operators. Once again, grammar transformations are the key to the problem.

For every level of precedence (see Section 2) we introduce a separate grammar rule.

Example: to implement the precedence levels 6 (`*` und `/`) and 5 (`+` und `-`) we would transform the original production

$$\begin{array}{rcl} \langle \mathit{opexpr} \rangle & \to & \langle \mathit{opexpr} \rangle \ \langle \mathit{infix} \rangle \ \langle \mathit{opexpr} \rangle \\ & | & \langle \mathit{comb} \rangle \end{array}$$

into the following rules (left recursion has been eliminated already)

$$\begin{array}{rcl} \langle \mathit{add} \rangle & \to & \langle \mathit{mul} \rangle \ \langle \mathit{add'} \rangle \\ \langle \mathit{add'} \rangle & \to & \langle \mathit{addop} \rangle \ \langle \mathit{mul} \rangle \ \langle \mathit{add'} \rangle \\ & | & \epsilon \\ \\ \langle \mathit{mul} \rangle & \to & \langle \mathit{factor} \rangle \ \langle \mathit{mul'} \rangle \\ \langle \mathit{mul'} \rangle & \to & \langle \mathit{mulop} \rangle \ \langle \mathit{factor} \rangle \ \langle \mathit{mul'} \rangle \\ & | & \epsilon \\ \\ \langle \mathit{addop} \rangle & \to & \texttt{+} \mid \texttt{-} \\ \langle \mathit{mulop} \rangle & \to & \texttt{*} \mid \texttt{/} \end{array}$$

(for $\langle \mathit{factor} \rangle$ see above). The method applies analogously for more than two precedence levels. It is instructive to check that an expression like `2+3*4` is actually parsed as $2 + (3 * 4)$.

The next step in implementing the compiler should be clear by now: apply the several transformations to the SASL grammar in Table 1 until it reaches a form that can be (rather easily) converted into a recursive descent parser for it. Recursive descent parser tend to be quite fast. However, for the sake of brevity, we entirely skipped error handling and recovery, so that *gigo* strikes again here.

You should a find detailed treatment of the grammar transformation techniques in any book on compiler construction—[ASU86] devotes several pages to this topic.

### 3.3 SASL Compilation and Combinators

Let us now turn to the core compilation phase—i.e., the backend—of our project. This phase expects the parse tree constructed by the frontend as input. It will construct a graph representation of the program to be passed to the graph reduction machine for execution (see Subsection 3.4).

**Removal of variables.** If you have a closer look at the parse trees you will find that they merely represent the application of built-in SASL functions to constant values. Inner nodes are always labeled with `@`. In what follows, we will understand operators like `+`, `:`, and `not` as built-in functions just like `hd` and `tl`. If this would be really all about it, we could pass the parse tree to the reduction machine directly.

Additional elements of SASL programs are user-defined functions that have been introduced by `def` resp. `where`, as well as variables used in function definitions. The compilation phase will remove all occurences of user-defined names and variables. The reduction machine operates on variable-free programs only.

At first glance, the complete removal of variables may seem odd. The technique we will explain in the following has been developed in David Turner's paper [Tur79]. Chapter 16 of [PJ87] is a useful reference, too.

**Global `def` definitions.** Let

$$\begin{array}{l} \texttt{def}\ v_1\ \texttt{=}\ E_1 \\ \texttt{def}\ v_2\ \texttt{=}\ E_2\,\texttt{.} \\ E \end{array}$$

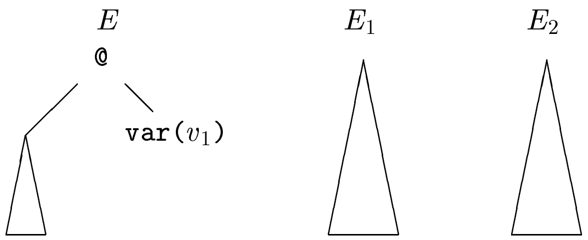
be a program with `def` definitions. The compiler undertakes the following two steps:

Step I. Construct the parse trees for $E_1$, $E_2$, and $E$ (this is the frontend's task).

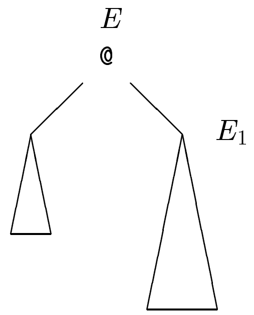
Step II. Replace every occurence of $v_i$ (nodes labeled $\texttt{var}(v_i)$ in the parse tree) in $E_1$, $E_2$, or $E$ by a reference to the parse tree of $E_i$. This may result in cycles if the definitions of the $v_i$ are (mutual) recursive. We end up with a graph for $E$.

The graph for $E$ is then passed on to the reduction machine. An example might clarify the matter:

Step I:



Step II:




Note that we only deal with a single copy of the parse tree of $E_1$. If there is more than one occurence of $v_1$ in $E$, $E_1$, or $E_2$, there will be the same number of references to $E_1$. Our toy example program does not reference the parse tree of $E_2$ at all. Hence $E_2$ is not part of the final graph (since there are no references to it, a garbage-collected implementation language like Eiffel will eventually free the memory occupied by $E_2$).

The definitions of $v_1$ and $v_2$ were simple in the sense that they did not define functions with parameters but simply introduced names for constant values (namely $E_1$ resp. $E_2$). Things get a bit more complicated if we have to remove variables from definitions like

$$\texttt{def}\ f\ x\ \texttt{=}\ E$$

In this case we are left with the task of "freeing" $E$ of all occurences of $x$. We turn to this now.

**Combinators.** $E$ is either built of

- constants $c$,
- variables $\texttt{var}(v)$, or
- function applications $f\ \texttt{@}\ a$

(we do not use the tree notation here to save space; `@` associates to the left). In order to compile $\texttt{def}\ f\ x\ \texttt{=}\ E$ we apply the operation $[x]$ (to be defined below; read $[x]$ as "*abstract* $x$") to $E$. Most importantly, we have $f = [x]E$, i.e. variable abstraction does not alter the meaning of the program. $[x]E$ is free of all occurences of $x$.

Let us define $[x]$ for the three possible cases:

$$\begin{aligned}
[x]c &= \mathbf{K}\ \texttt{@}\ c \\
[x]\texttt{var}(v) &= \begin{cases} \mathbf{I}, & \text{if } x = v \\ \mathbf{K}\ \texttt{@}\ \texttt{var}(v) & \text{otherwise.} \end{cases} \\
[x](f\ \texttt{@}\ a) &= \mathbf{S}\ \texttt{@}\ [x]f\ \texttt{@}\ [x]a
\end{aligned}$$

The three symbols **S**, **K**, and **I** are special built-in functions of the reduction machine. We will refer to them as *combinators*, a term which stems from the theory of the *lambda calculus*. They are defined as follows:

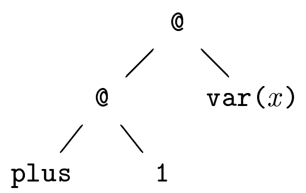
$$\begin{aligned} \mathbf{S} \mathbin{\texttt{@}} f \mathbin{\texttt{@}} g \mathbin{\texttt{@}} x &= f \mathbin{\texttt{@}} x \mathbin{\texttt{@}} (g \mathbin{\texttt{@}} x) \\ \mathbf{K} \mathbin{\texttt{@}} x \mathbin{\texttt{@}} y &= x \\ \mathbf{I} \mathbin{\texttt{@}} x &= x \end{aligned}$$

**S** is a mnemonic for substitution, **K** represents a function being constant with respect to its second argument (think of the german word *konstant*), while **I** is the identity.

Example:

$$\texttt{def incr } x \texttt{ = 1 + } x$$

The frontend generates the following parse tree for `incr`'s body:




The compiler abstracts $x$ away and therefore computes

$$[x]((\texttt{plus @ 1}) \texttt{ @ var}(x))$$

which we show step-wise here:

$$\begin{aligned} & [x]((\texttt{plus @ 1}) \texttt{ @ var}(x)) \\ &= \mathbf{S} \texttt{ @ } [x](\texttt{plus @ 1}) \texttt{ @ } [x]\texttt{var}(x) \\ &= \mathbf{S} \texttt{ @ } (\mathbf{S} \texttt{ @ } [x]\texttt{plus @ } [x]\texttt{1}) \texttt{ @ } \mathbf{I} \\ &= \mathbf{S} \texttt{ @ } (\mathbf{S} \texttt{ @ } (\mathbf{K} \texttt{ @ plus}) \texttt{ @ } (\mathbf{K} \texttt{ @ 1})) \texttt{ @ } \mathbf{I} \end{aligned}$$

Global definitions of the form

$$\texttt{def } f \ x_1 \cdots x_n \texttt{ = } E$$

(i.e. definitions of function with more than one parameter) are transformed in $n$ steps:

$$[x_1](\cdots([x_{n-1}]([x_n]E))\cdots)$$

The inner abstractions have to be performed first.

During abstraction, we treat built-in functions (like `plus` in this example, but also **S**, **K**, and **I**) as constants. This is no surprise if you remember that functions are ordinary values in SASL.

When the abstraction step has been completed, remaining occurences of `var(`$v$`)` are replaced by references to their definitions as explained under Step II above.

We shall spend the time to convince ourselves that the combinator expression—the compiled program—actually computes $1 + x$ when applied to the argument $x$, i.e. we will check that the combinator expression implements `incr`. This involves the manual step-wise reduction of the combinator expression, a task which the reduction machine will carry out later. As a rule of thumb we will evaluate the left-most function first (we emulate the so-called *normal order reduction*). Details are to be found in Subsection 3.4.

Let us evaluate (the combinator reduced is shown below the = sign; only the grey part of the expression is affected by the next reduction step):

$$
\begin{aligned}
&\mathbf{S} \mathbin{@} (\mathbf{S} \mathbin{@} (\mathbf{K} \mathbin{@} \mathtt{plus}) \mathbin{@} (\mathbf{K} \mathbin{@} 1)) \mathbin{@} \mathbf{I} \mathbin{@} x \\
\underset{\mathbf{S}}{=}\ & \mathbf{S} \mathbin{@} (\mathbf{K} \mathbin{@} \mathtt{plus}) \mathbin{@} (\mathbf{K} \mathbin{@} 1) \mathbin{@} x \mathbin{@} (\mathbf{I} \mathbin{@} x) \\
\underset{\mathbf{S}}{=}\ & \mathbf{K} \mathbin{@} \mathtt{plus} \mathbin{@} x \mathbin{@} (\mathbf{K} \mathbin{@} 1 \mathbin{@} x) \mathbin{@} (\mathbf{I} \mathbin{@} x) \\
\underset{\mathbf{K}}{=}\ & \mathtt{plus} \mathbin{@} (\mathbf{K} \mathbin{@} 1 \mathbin{@} x) \mathbin{@} (\mathbf{I} \mathbin{@} x) \\
\underset{\mathbf{K}}{=}\ & \mathtt{plus} \mathbin{@} 1 \mathbin{@} (\mathbf{I} \mathbin{@} x) \\
\underset{\mathbf{I}}{=}\ & \mathtt{plus} \mathbin{@} 1 \mathbin{@} x
\end{aligned}
$$

Because `plus` is a built-in function, the reduction machine is able to directly apply `plus` to the two arguments given. The value of `plus 1` $x$ is then returned as the result.

**Local `where` definitions.** Since a `where` introduces local (potentially function-valued) variables, abstraction plus the fact that functions are first-class citizens in SASL immediately provides us with a way to compile `where` definitions away.

Let us split the matter in two cases: `where`s that contain exactly one definition, i.e. are of the general form

$$E_1 \ \mathtt{where} \ f \ \mathtt{=} \ E_2$$

and `where` expressions featuring a list (separated by `;`) of more than one definition. We will dicuss the latter form later on.

The compilation rule for the above expression simply is

$$([f]E_1) \mathbin{@} E_2$$

We turn $E_1$ into a function of $f$ (by abstracting $f$ from $E_1$) and apply this new function to $E_2$, which results in replacing every occurence of $f$ in $E_1$ with $E_2$. Note that this replacement of expressions is just what `where` definitions are all about.

If $f$ is function-valued, i.e. if the program has the form

$$E_1 \ \mathtt{where} \ f \ x \ \mathtt{=} \ E_2$$

we proceed by combining the methods we learned so far: we abstract $x$ from $E_2$ in order to compile the local function and additionally remove occurences of $f$ from $E_1$. We then have:

$$([f]E_1) \mathbin{@} ([x]E_2)$$

We generalize the `where` definition once more: how to compile a *recursive* local definition? In the definition of $f$ we refer to $f$ itself as in:

$$E_1 \texttt{ where } f\ x \texttt{ = } \ldots f \ldots$$

In order to be able to detect this case, the compiler needs a means to check for the use of a certain variable name (here $f$) in an expression. A simple parse tree traversal should do the job.

At this point we employ a new combinator, the *fixpoint combinator* $\mathbf{Y}$, whose definition is

$$\mathbf{Y} \texttt{ @ } f \quad = \quad f \texttt{ @ } (\mathbf{Y} \texttt{ @ } f)$$

Note that $\mathbf{Y}$'s definition itself is recursive (it realizes the repeated application of $f$) and we would have to expand $\mathbf{Y}$ an infinite number of times to implement it properly. The reduction machine will implement $\mathbf{Y}$ by rewriting it into a cycle in the program's graph. See Subsection 3.4 on this issue.

All we need to know during the compilation phase is that the above expression involving local recursion has to be rewritten into

$$([f]E_1) \texttt{ @ } (\mathbf{Y} \texttt{ @ } [f]([x]E_2))$$

We abstract $f$ from $E_1$ (as usual) *and* $E_2$, making $E_2$ a function of $f$. The fixpoint combinator then realizes the repeated application of this function to itself which implements the recursion. This completes the case of a `where` definition introducing a single name only.

**Multiple local `where` definitions.** The most general form of a `where` expression, namely a `where` followed by a list of definitions separated by semicolons, demands special consideration. Consider:

$$\begin{array}{lll} E_1 \texttt{ where } & f\ x \texttt{ = } E_2\texttt{;} \\ & g\ y \texttt{ = } E_3 \end{array}$$

The compilation method will be similar to the single definition case but we have to employ yet another combinator, $\mathbf{U}$, which we will define below. Do not be alarmed by the "complexity" of the compilation rule, its structure is rather simple. The above expression has to be compiled into

$$\mathbf{U} \texttt{ @ } ([f](\mathbf{U} \texttt{ @ } [g](\mathbf{K} \texttt{ @ } E_1))) \texttt{ @ } \texttt{[}[x]E_2\texttt{,}[y]E_3\texttt{]}$$

Remember that `[`$x$`,`$y$`]` is a shorthand for `:` `@` $x$ `@` (`:` `@` $y$ `@` `nil`).

If the definition of $f$ and/or $g$ is recursive (this includes expression like

$$\begin{array}{lll} E_1 \texttt{ where } & f\ x \texttt{ = } \ldots g \ldots\texttt{;} \\ & g\ y \texttt{ = } \ldots f \ldots \end{array}$$

i.e. mutual recursion) we proceed just like in the case of single `where` definitions: we use $\mathbf{Y}$ to express recursion. The compiled expression then is

$$\mathbf{U} \texttt{ @ } ([f](\mathbf{U} \texttt{ @ } [g](\mathbf{K} \texttt{ @ } E_1))) \texttt{ @ } (\mathbf{Y} \texttt{ @ } (\mathbf{U} \texttt{ @ } ([f](\mathbf{U} \texttt{ @ } [g](\mathbf{K} \texttt{ @ } \texttt{[}[x]E_2\texttt{,}[y]E_3\texttt{]})))))$$

The combinator **U** is a mnemonic for *uncurrying*. It is defined (and thus implemented in the reduction machine) by the equation

$$\mathbf{U} \mathbin{@} f \mathbin{@} z \quad = \quad f \mathbin{@} (\texttt{hd} \mathbin{@} z) \mathbin{@} (\texttt{tl} \mathbin{@} z)$$

**U** realizes the application of $f$ to two arguments (the head and tail of the list $z$) by applying $f$ to (`hd` $z$) first. The result is then applied to (`tl` $z$).

The following subsection will sketch the corresponding reduction machine which completes the implementation of the backend. The reduction machine constitutes the SASL *runtime* if you like. It will realize the five combinators **S**, **K**, **I**, **Y**, and **U** as well as the SASL built-in functions.

## 3.4 SK Reduction Machine

The **SK** reduction machine (which draws its name from the two combinators) constitutes the runtime system of our SASL project. It acts like a processor entirely realized in software. The machine is remarkable because it does not operate on a machine code representation but rather executes the compiled program (the graph) by *transforming the program itself.*

It is reasonable to think of a *simplifier* transforming and simplyfying the graph built from combinators, built-in functions, and constants. The simplification process goes on until the graph has been reduced to a single constant value (of type `num`, `bool`, `string`, or `list`). This value is then printed to `stdout`—which completes the compiler run.

In what follows we will use the terms "evaluation", "simplification", and "reduction" as synonyms. This subsection presents the simplification rules used for the graph reduction process. The rules implement the defining equations for the combinators **S**, **K**, **I**, **Y**, and **U**. There will be additional rules for the built-in functions (like `+`, `cond`, or `tl`).

**Lazy evaluation.** The evaluation of function applications in SASL happens *lazyily.* This is in contrast to the vast majority of programming languages whose evaluation semantics are known to be *strict.* What does lazy evaluation mean?

A programming language with strict semantics evaluates the arguments of a function call *before* the function is actually applied. An example might help here (using Eiffel syntax):

```
feature first(x,y : T) : T is
   -- this implements K, did you notice?
   do
     Result := x
   end -- first
...
first(1+2, 3*4)
```

Before we actually apply `first`, we evaluate the arguments `1+2` and `3*4` and then pass them to `first`, i.e. execute the call `first(3,12)`. Half of the argument evaluation effort is for nothing, of course, because `first` simply throws `y`'s value away. The following example is even worse. Eiffel (read: any strict language) is not able to complete the evaluation at all:

```
feature bomb(n : INTEGER) : INTEGER is
   do
     Result := bomb(n+1)
   end -- bomb
...
first(42, bomb(0))
```

The evaluation of the second argument does not terminate—`first` will never be called.

SASL is lazy when it comes to the evaluation of arguments to a function call: every argument is passed *as-is* to the function. The general principle is: perform reductions only when they are absolutely necessary to compute the result. For example, in order to evaluate $x$ `+` $y$ it is indispensable to reduce $x$ and $y$ to numbers before we can caryy out the addition (`+` is said to be *strict in $x$ and $y$*. The function `first` is neither strict in its first nor in its second argument. SASL computes the correct results for both examples:

```
def first x y = x
def bomb n = bomb (n+1).

[first 1+2 3*4, first 42 (bomb 0)]
```

This program reduces to `[3,42]`. It is interesting to note that `first` returns its first argument `1+2` unevaluated. The addition is not done before the result has to be actually printed.

To conlcude, it is the point of lazy evaluation to delay the reduction of function arguments as long as possible[1]. Arguments are passed as-is instead. If we return to the graphical representation of function application,

```
   @
  / \
 f   a
```

we reduce the left branch before we reduce right branch (if we have to do so at all). Literature refers to this as *normal order reduction*. Strict languages evaluate the right branch first (*applicative order reduction*). Examples of lazy languages are Haskell, Miranda[2], or Lazy ML. Algol's *call-by-name* implements a similar but more inefficient evaluation strategy. More on that later.

[1] Lazy functional programmers would add the requirement that arguments are evaluated at most once.

[2] Miranda is a trademark of Research Software Ltd.

**Control of reduction.** The principal operation of the reduction machine is controlled by the so-called *left ancestors stack*. The following algorithm implements normal order reduction of our compiled programs with the help of this stack.

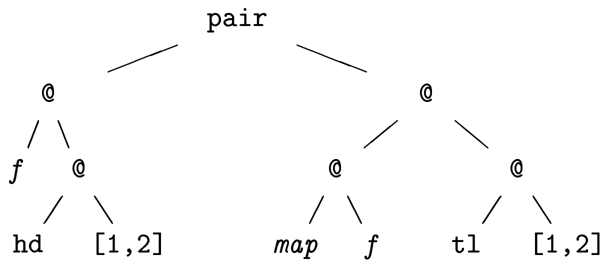
- When execution starts, a pointer to the (graph representation of the) complete compiled program is the only element—the stack top—on the stack.
- As long as the operation at the stack top is a function application, i.e. a `@`-node, we push the left ancestor of that `@`-node onto the stack. The stack derives its name from this step.
- At some point in time the stack top will contain a combinator or another built-in function. If so, we apply the graph reduction rule (see below) that corresponds to the stack top. The arguments to the function call are easily accessed: a pointer to the $n$-th argument may be found at the $n$-th position above the stack top. The function and its arguments are popped from the stack and replaced by the reduction result.
- After a reduction step has been completed, the machine continues its operation by examining the stack top again: if an `@`-node is encountered, push the left ancestor, else reduce.

The machine halts if the stack top contains a "printable" object, i.e. a value of type `num`, `bool`, `string`, or a `list` value constructed by a `pair` node (`pair` nodes are introduced on the evaluation of the : list constructor and are used to represent list values as trees; see :'s reduction rule below). Since SASL's evaluation is lazy it might well be that some or all elements of a list value have not yet been reduced, i.e. the left (head) and/or right (tail) ancestor of a `pair` constructor still is a function application. Thus, in order to print the list value, the reduction machine has to call itself recursively on these elements: it is not uncommon for a list-valued SASL program to complete its evaluation in very few reduction steps. However, the "real work" starts when the routine has to reduce the single list elements to be able to print their value.

Let us shed some light on this. Suppose the following global definitions (`length l` determines the lenght of its list argument `l`, while `map f l` applies function `f` to every element of the list `l`):

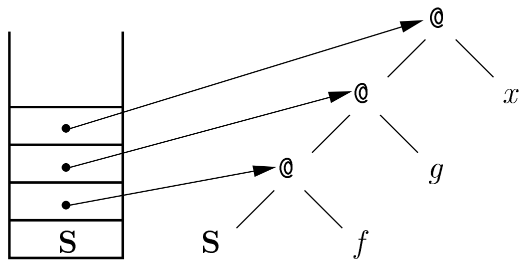
```
def length l = if l = nil then 0
                          else 1+length (tl l)

def map f l  = if l = nil then nil
                          else f x : map f xs
               where x  = hd l;
                     xs = tl l.
```

A call like `length ["a","b","c"]` eventually leads to the evaluation of the expression $1 + (1 + (1 + 0)) = 3$ whose size is linear in the size of input list. The result can be printed after the summation took place. The reduction of `map f [1,2]`, however, produces the following intermediate result after only a few reduction steps—no matter how long the argument list may be! We abbreviate the combinator expressions for `map` and `f` by *`map`* resp. *`f`* here:




The top `pair` node represents the list-valued result produced by our call to `map` and the reduction algorithm starts to print the result. However, to succeed in this there still remains the reduction work for the head of the list, `f (hd [1,2])`, as well as its tail, `map f (tl [1,2])` to be done: the print routine calls the reduction machine on both.

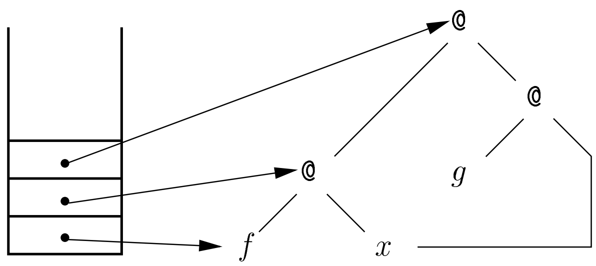
**A reduction step.** To further illustrate how reduction proceeds, let us re-enact a single reduction step. Let $f$, $g$, and $x$ denote arbitrary compiled expressions. The combinator expression $\mathbf{S}$ @ $f$ @ $g$ @ $x$ eventually leads to the following configuration of the left ancestors stack after we have walked down the "spine" of @-nodes (stacks grow downwards):




The combinator **S** at the stack top instructs us to apply its corresponding reduction rule

$$\mathbf{S} \texttt{ @ } f \texttt{ @ } g \texttt{ @ } x = f \texttt{ @ } x \texttt{ @ } (g \texttt{ @ } x)$$

This leads to the following combinator graph resp. stack content:

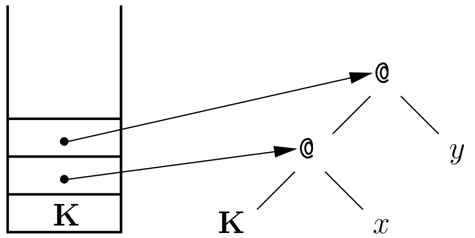


The stack top now contains the root node of $f$'s combinator representation. It determines how the reduction machine will proceed.

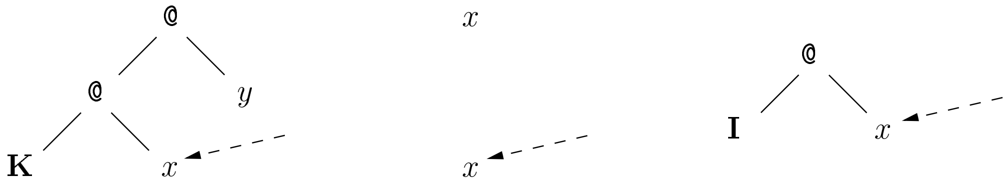
Note that, after the reduction, the code for $x$ has not been copied but is shared by $f$ and $g$. If $x$ has to be evaluated at all we have to do so at most once. The sharing of common code (or the detection of *common subexpressions*) is one crucial optimization the efficiency of the reduction machine relies on. Normal order reduction and sharing together make SASL a fully lazy language. Optimizations of this kind are equivalence-preserving because SASL is side-effect free (aka *pure*).

**Indirection Nodes.** To get the full benefit from this self-optimzing behaviour of the reduction machine one has to be careful when rewriting the combinator graph.

Let us trace the reduction of a **K** combinator. The stack configuration is depicted below:




Suppose that the dashed arrow in (a) below is an additional second reference to $x$ (this may happen when an **S** combinator has been reduced before, for example):

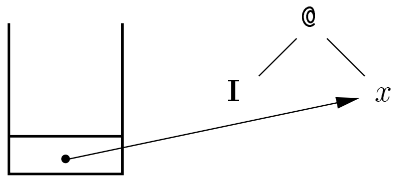



(a) (b) (c)

Following the reduction rule for **K**, i.e. $\mathbf{K} \mathbin{@} x \mathbin{@} y = x$, we obtain the situation (b) in which two copies of $x$ exist. If we reduce one of the copies, the other copy does not benefit from this effort and laziness is lost. In situations like these we therefore introduce *indirection nodes* (see (c)) that allow to preserve the sharing property. The identity combinator **I** is well suited for this task:

$$\mathbf{K} \mathbin{@} x \mathbin{@} y = \mathbf{I} \mathbin{@} x$$

still is a correct reduction. Indirection nodes affect the graph transformation only. The left ancestors stack may of course point to $x$ directly:



Every time a reference to **I** @ $x$ is encountered, the machine may elide the indirection by replacing it with a reference to $x$. After the last reference to the indirection has been removed it will be reclaimed by the garbage collector (if present).

**Reduction rules.** There remains nothing to be done but giving the rules for the reduction of the combinators as well as SASL's built-in functions—which are nothing else but rather specialized combinators. If a reduction involves the generation of indirection nodes, they are explicitly shown in the following table.

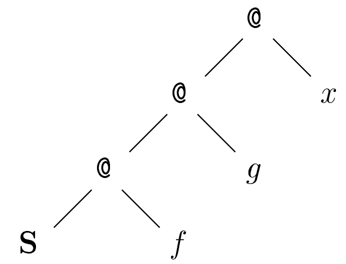
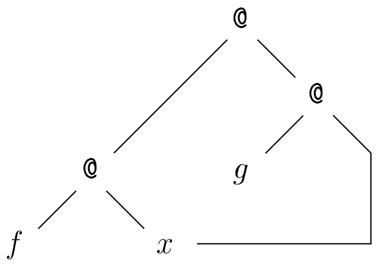
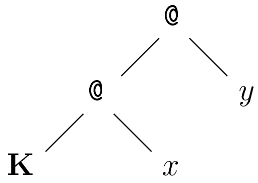
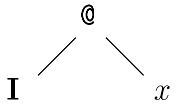
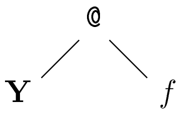
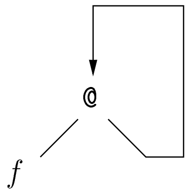
| combinator | reduction |
|---|---|
|  |  |
|  |  |
|  |  |

Table 2: Reduction rules

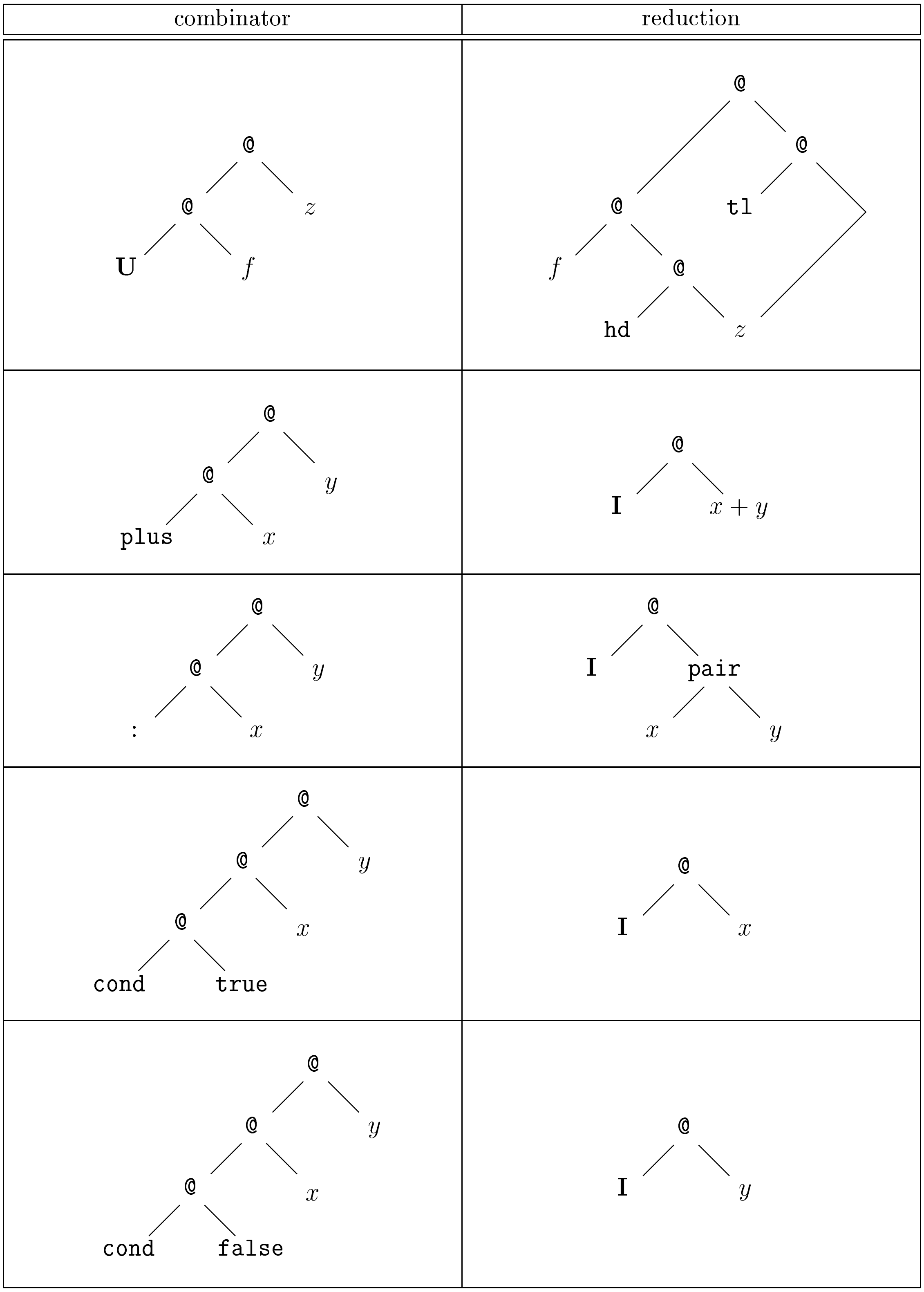



Table 2: Reduction rules

**Notes.**

- Realize that "tying the knot" in the reduction of **Y** actually implements the purpose of **Y**, i.e. the repeated application of a function to itself:

$$\mathbf{Y} \texttt{ @ } f = f \texttt{ @ } (\mathbf{Y} \texttt{ @ } f) = f \texttt{ @ } (f \texttt{ @ } (\mathbf{Y} \texttt{ @ } f)) = \cdots = f \texttt{ @ } (f \texttt{ @ } (f \texttt{ @ } (\cdots)\cdots))$$

- The implementation of the built-in operators is completely analogous to the rule for `plus`. Since these operators are strict you have to call the reduction machine recursively on their arguments: you cannot add two expressions until they have been actually reduced to numbers. `cond` is strict in its first argument only.

- A `pair` node represents a list $x$ : $y$ that has been built by the `:` (*cons*) list constructor. The built-in functions `hd` and `tl` reduce their argument to a `pair`—but not any further—and return $x$ resp. $y$; $x$ and $y$ themselves are not reduced.

# 4 Optimizing SK Compilation

The **SK** compilation and reduction scheme as stated here is simple and elegant, the compiled epxressions however tend to be quite large, resulting in space-consuming combinator graphs and high reduction times.

Fortunately, it is as easy to optimize the reduction process. The optimizations discussed in short below are not necessarily needed to get the compiler working, but they render the system a lot more usable.

**SK** optimzation involves the introduction of a series of new combinators, namely **B**, **B**$^*$, **C**, **C'**, and **S'**. We briefly motivate the introducion of **B** here and then give the optimization rewrite rules. For a full review of the optimization process, please refer to Chapter 16 of [PJ87].

Suppose we have to compile the function definition `def neg x = not x`. Variable abstraction produces the combinator equivalent **S** `@` (**K** `@` `not`) `@` **I** for `neg`. If you have a closer look at the reduction of the expression `neg` $x$, you will notice that in the second step **K** discards its second argument $x$:

```
        @                        @                      @                 @
       / \        ->          /     \        ->        / \       ->      / \
      @   x       S          @       @       K      not   @      I    not   x
     / \                    / \     / \                  / \
    @   I                  @   x   I   x                I   x
   / \                    / \
  S   @                  K   not
     / \
    K   not
```

Passing $x$ to **K** in this situation is wasted effort. In general, any combinator expression of the form $\mathbf{S} \mathbin{@} (\mathbf{K} \mathbin{@} f) \mathbin{@} g$ leads to such an unnecessary reduction step.

The new combinator **B**, defined by

$$\mathbf{B} \mathbin{@} f \mathbin{@} g \mathbin{@} x = f \mathbin{@} (g \mathbin{@} x)$$

avoids passing $x$ to $f$ and therefore saves a reduction step in a situation like above. A suitable **SK** optimizer will include an *optimzation rule*

$$\mathbf{S} \mathbin{@} (\mathbf{K} \mathbin{@} f) \mathbin{@} g \quad \rightarrow \quad \mathbf{B} \mathbin{@} f \mathbin{@} g$$

thus replacing a combination of **S** and **K** by **B** if possible. The new combinator code for `neg` now is **B** `@ not @` **I**. Compared to the original combinator expression, we get code that is of smaller size and additionally reducable in fewer steps.

The introduction of $\mathbf{B}^*$, **C**, **C'**, and **S'** pursues very similar goals. We conclude this section by giving their definitions as well as the optimization rules.

**Additional combinators.**

$$\begin{array}{rcl}
\mathbf{B} \mathbin{@} f \mathbin{@} g \mathbin{@} x & = & f \mathbin{@} (g \mathbin{@} x) \\
\mathbf{C} \mathbin{@} f \mathbin{@} g \mathbin{@} x & = & f \mathbin{@} x \mathbin{@} g \\
\mathbf{S'} \mathbin{@} c \mathbin{@} f \mathbin{@} g \mathbin{@} x & = & c \mathbin{@} (f \mathbin{@} x) \mathbin{@} (g \mathbin{@} x) \\
\mathbf{B}^* \mathbin{@} c \mathbin{@} f \mathbin{@} g \mathbin{@} x & = & c \mathbin{@} (f \mathbin{@} (g \mathbin{@} x)) \\
\mathbf{C'} \mathbin{@} c \mathbin{@} f \mathbin{@} g \mathbin{@} x & = & c \mathbin{@} (f \mathbin{@} x) \mathbin{@} g
\end{array}$$

**Optimization rules.**

$$\begin{array}{rcl}
\mathbf{S} \mathbin{@} (\mathbf{K} \mathbin{@} f) \mathbin{@} (\mathbf{K} \mathbin{@} g) & \rightarrow & \mathbf{K} \mathbin{@} (f \mathbin{@} g) \\
\mathbf{S} \mathbin{@} (\mathbf{K} \mathbin{@} f) \mathbin{@} \mathbf{I} & \rightarrow & f \\
\mathbf{S} \mathbin{@} (\mathbf{K} \mathbin{@} f) \mathbin{@} (\mathbf{B} \mathbin{@} g \mathbin{@} h) & \rightarrow & \mathbf{B}^* \mathbin{@} f \mathbin{@} g \mathbin{@} h \\
\mathbf{S} \mathbin{@} (\mathbf{K} \mathbin{@} f) \mathbin{@} g & \rightarrow & \mathbf{B} \mathbin{@} f \mathbin{@} g \\
\mathbf{S} \mathbin{@} (\mathbf{B} \mathbin{@} f \mathbin{@} g) \mathbin{@} (\mathbf{K} \mathbin{@} h) & \rightarrow & \mathbf{C'} \mathbin{@} f \mathbin{@} g \mathbin{@} h \\
\mathbf{S} \mathbin{@} f) \mathbin{@} (\mathbf{K} \mathbin{@} g) & \rightarrow & \mathbf{C} \mathbin{@} f \mathbin{@} g \\
\mathbf{S} \mathbin{@} (\mathbf{B} \mathbin{@} f \mathbin{@} g) \mathbin{@} h & \rightarrow & \mathbf{S'} \mathbin{@} f \mathbin{@} g \mathbin{@} h
\end{array}$$

You may expect *significant* speed-ups if you completely implement these rules in your reduction machine.

## References


[ASU86] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers—Principles, Techniques, and Tools.* Addison Wesley, 1986.

[Mey92] Betrand Meyer. *Eiffel – The Language.* Object-Oriented Series. Prentice-Hall, 1992.

[PJ87] Simon L. Peyton Jones. *The Implementation of Functional Programming Languages.* International Series in Computer Science. Prentice-Hall, 1987.

[Tur79] David A. Turner. A New Implementation Technique for Applicative Languages. *Software—Practice and Experience*, 9:31–49, 1979.

## A An SASL Prelude

Some programming tasks are so common and repeating when dealing with functional languages that it is a good habit to provide the compiler with a library of predefined functions, often referred to as *prelude*. A well-designed prelude can significantly increase the user-friendliness and usefulness of the SASL system. Traditionally, the prelude is loaded at the compiler's startup so that user's may refer to names defined in the prelude when they write their own programs.

Below we list some of the functions that deserve a place in a decent SASL prelude. This includes the higher-order list processing functions `map`, `filter`, `fold`, `append`, `reverse`, standard functions like `id` or `comp` ($\circ$ or function composition), several iterators, aggregation functions like `sum` and `product`, but also an insertion `sort` and finally curried versions of the arithmetic and relational operators (`plus`, `leq`, and friends). There is no limit.

The prelude functions may additionally serve as further SASL program examples as well as test cases for the student's compiler.

```
def id x = x

def until p f x = if p x then x else until p f (f x)

def comp f g x = f (g x)

def map f l = if l=nil then nil
                       else f x:map f xs where x  = hd l;
                                               xs = tl l

def fold m z l = if l=nil then z
                          else m x (fold m z xs) where x  = hd l;
                                                       xs = tl l

def append l1 l2 = if l1=nil then l2
                             else x:append xs l2 where x  = hd l1;
                                                       xs = tl l1

def reverse l = if l=nil then nil
                         else append (reverse (tl l)) [hd l]
```

```
def filter p l = if l=nil then nil
                     else if p x then x:filter p xs
                                 else filter p xs where x  = hd l;
                                                        xs = tl l

def sort p l = if l=nil then nil
                   else insert p (hd l) (sort p (tl l))
               where
                    insert pp e ll = if ll=nil then [e]
                                     else
                                          if pp e (hd ll) then e:ll
                                          else
                                               (hd ll):insert pp e (tl ll)


def drop n l = if n<=0 then l
                 else if l=nil then nil
                               else drop (n-1) (tl l)

def take n l = if n=0 or l=nil then nil
                         else x:take (n-1) xs where x  = hd l;
                                                    xs = tl l

def at n l = if n=0 then hd l
                else at (n-1) (tl l)

def length l = if l=nil then 0
                   else 1+length (tl l)

def null l = l=nil

def init l = if xs=nil then nil
                 else x:init xs where x  = hd l;
                                      xs = tl l

def iterate f x = x : iterate f (f x)

def repeat x = xs where xs=x:xs

def cycle xs = xs1 where xs1=append xs xs1

def splitAt n l = if n<=0 then []:l
                     else if l=nil then []:[]
                                   else ((hd l):xs1):xs2
                  where
                       xs  = splitAt (n-1) (tl l);
                       xs1 = hd xs;
                       xs2 = tl xs
```

```
def takeWhile p l = if l=nil then nil
                             else if p x then x:takeWhile p xs
                                         else nil
                    where
                          x  = hd l;
                          xs = tl l

def sum     = fold plus 0
def product = fold mul 1

def plus x y   = x+y
def mul x y    = x*y
def div x y    = x/y
def div2 y x   = x/y
def minus x y  = x-y
def minus2 y x = x-y
def lt x y     = x<y
def leq x y    = x<=y
def eq x y     = x=y
def neq x y    = x~=y
def geq x y    = x>=y
def gt x y     = x>y
```

The SASL input file containing the prelude may be downloaded from the web homepage of this course:

`http://www.informatik.uni-konstanz.de/~grust/SASL/`

## B The Transformed SASL Grammar

The SASL grammar presented here is equivalent to the one given in Table 1 in the sense that both grammars accept the same language. This version of the grammar has been left-factorized, it is free of left recursion, and accounts for operator precedence. It should be a straightforward task to construct its LL(1) recursive descent parser. See Section 3.2 on these issues.

$$
\begin{array}{rcl}
\langle \mathit{system} \rangle & \rightarrow & \langle \mathit{funcdefs} \rangle \ \texttt{.} \ \langle \mathit{expr} \rangle \\
 & | & \langle \mathit{expr} \rangle \\
\langle \mathit{funcdefs} \rangle & \rightarrow & \texttt{def} \ \langle \mathit{def} \rangle \ \langle \mathit{funcdefs'} \rangle \\
\langle \mathit{funcdefs'} \rangle & \rightarrow & \texttt{def} \ \langle \mathit{def} \rangle \ \langle \mathit{funcdefs'} \rangle \\
 & | & \epsilon \\
\langle \mathit{defs} \rangle & \rightarrow & \langle \mathit{def} \rangle \ \langle \mathit{defs'} \rangle \\
\langle \mathit{defs'} \rangle & \rightarrow & \texttt{;} \ \langle \mathit{def} \rangle \ \langle \mathit{defs'} \rangle \\
 & | & \epsilon
\end{array}
$$

Table 3: A massaged SASL grammar

| | | |
|---|---|---|
| ⟨*def*⟩ | $\rightarrow$ | ⟨*name*⟩ ⟨*abstraction*⟩ |
| ⟨*abstraction*⟩ | $\rightarrow$ | `=` ⟨*expr*⟩ |
| | $\mid$ | ⟨*name*⟩ ⟨*abstraction*⟩ |
| ⟨*expr*⟩ | $\rightarrow$ | ⟨*condexpr*⟩ ⟨*expr'*⟩ |
| ⟨*expr'*⟩ | $\rightarrow$ | `where` ⟨*defs*⟩ ⟨*expr'*⟩ |
| | $\mid$ | $\epsilon$ |
| ⟨*condexpr*⟩ | $\rightarrow$ | `if` ⟨*expr*⟩ `then` ⟨*condexpr*⟩ `else` ⟨*condexpr*⟩ |
| | $\mid$ | ⟨*listexpr*⟩ |
| ⟨*listexpr*⟩ | $\rightarrow$ | ⟨*opexpr*⟩ ⟨*listexpr'*⟩ |
| ⟨*listexpr'*⟩ | $\rightarrow$ | `:` ⟨*expr*⟩ |
| | $\mid$ | $\epsilon$ |
| ⟨*opexpr*⟩ | $\rightarrow$ | ⟨*conjunct*⟩ ⟨*opexpr'*⟩ |
| ⟨*opexpr'*⟩ | $\rightarrow$ | `or` ⟨*conjunct*⟩ ⟨*opexpr'*⟩ |
| | $\mid$ | $\epsilon$ |
| ⟨*conjunct*⟩ | $\rightarrow$ | ⟨*compar*⟩ ⟨*conjunct'*⟩ |
| ⟨*conjunct'*⟩ | $\rightarrow$ | `and` ⟨*compar*⟩ ⟨*conjunct'*⟩ |
| | $\mid$ | $\epsilon$ |
| ⟨*compar*⟩ | $\rightarrow$ | ⟨*add*⟩ ⟨*compar'*⟩ |
| ⟨*compar'*⟩ | $\rightarrow$ | ⟨*relop*⟩ ⟨*add*⟩ ⟨*compar'*⟩ |
| | $\mid$ | $\epsilon$ |
| ⟨*add*⟩ | $\rightarrow$ | ⟨*mul*⟩ ⟨*add'*⟩ |
| ⟨*add'*⟩ | $\rightarrow$ | ⟨*addop*⟩ ⟨*mul*⟩ ⟨*add'*⟩ |
| | $\mid$ | $\epsilon$ |
| ⟨*mul*⟩ | $\rightarrow$ | ⟨*factor*⟩ ⟨*mul'*⟩ |
| ⟨*mul'*⟩ | $\rightarrow$ | ⟨*mulop*⟩ ⟨*factor*⟩ ⟨*mul'*⟩ |
| | $\mid$ | $\epsilon$ |
| ⟨*factor*⟩ | $\rightarrow$ | ⟨*prefix*⟩ ⟨*comb*⟩ |
| | $\mid$ | ⟨*comb*⟩ |
| ⟨*comb*⟩ | $\rightarrow$ | ⟨*simple*⟩ ⟨*comb'*⟩ |
| ⟨*comb'*⟩ | $\rightarrow$ | ⟨*simple*⟩ ⟨*comb'*⟩ |
| | $\mid$ | $\epsilon$ |
| ⟨*simple*⟩ | $\rightarrow$ | ⟨*name*⟩ |
| | $\mid$ | ⟨*builtin*⟩ |
| | $\mid$ | ⟨*constant*⟩ |
| | $\mid$ | `(` ⟨*expr*⟩ `)` |
| ⟨*name*⟩ | $\rightarrow$ | ⟨*id*⟩ |
| ⟨*builtin*⟩ | $\rightarrow$ | `hd` |
| | $\mid$ | `tl` |
| ⟨*constant*⟩ | $\rightarrow$ | ⟨*num*⟩ |
| | $\mid$ | ⟨*bool*⟩ |
| | $\mid$ | ⟨*string*⟩ |
| | $\mid$ | `nil` |
| | $\mid$ | ⟨*list*⟩ |
| ⟨*list*⟩ | $\rightarrow$ | `[` ⟨*list'*⟩ |
| ⟨*list'*⟩ | $\rightarrow$ | `]` |
| | $\mid$ | ⟨*listelems*⟩ `]` |

Table 3: A massaged SASL grammar

| | | |
|---|---|---|
| $\langle listelems \rangle$ | $\rightarrow$ | $\langle expr \rangle$ $\langle listelems' \rangle$ |
| $\langle listelems' \rangle$ | $\rightarrow$ | `,` $\langle expr \rangle$ $\langle listelems' \rangle$ |
| | $\mid$ | $\epsilon$ |
| $\langle prefix \rangle$ | $\rightarrow$ | `-` $\mid$ `+` $\mid$ `not` |
| $\langle addop \rangle$ | $\rightarrow$ | `+` $\mid$ `-` |
| $\langle mulop \rangle$ | $\rightarrow$ | `*` $\mid$ `/` |
| $\langle relop \rangle$ | $\rightarrow$ | `=` $\mid$ `~=` $\mid$ `<` $\mid$ `>` $\mid$ `<=` $\mid$ `>=` |
| $\langle id \rangle$ | $\rightarrow$ | $[\mathtt{a-zA-Z\_}][\mathtt{a-zA-Z\_0-9}]^*$ |
| $\langle num \rangle$ | $\rightarrow$ | $[\mathtt{0-9}]^+$ |
| $\langle bool \rangle$ | $\rightarrow$ | `true` $\mid$ `false` |
| $\langle string \rangle$ | $\rightarrow$ | `"`$\langle ASCII\ character \rangle^*$`"` |

Table 3: A massaged SASL grammar