Implementation of Programming Languages

Lecture Notes

November 15, 2004

Preface

Version spring 2001

These lecture notes are the result of a joint effort. In particular, the following people have contributed (in alphabetical order).

Arthur Baars	AG system and documentation
Atze Dijkstra	Code generation, SSM, editing of lecture notes, course design, exams
Jurriaan Hage	Type systems (intro)
Bastiaan Heeren	Type explanation lab exercise
Andres Löh	AG system and documentation
Piet van Oostrum	Bibtex lab exercise
Doaitse Swierstra	Parser combinators, syntax directed translation, AG system design, type systems (AG code), course design, exams.

Further, these lecture notes are not finished and are work in progress. Please keep this in mind when reading and using these notes.

Version spring 2002

This version adds no functionality, only bug fixes. At some places the used language is (still) Dutch, in particular at some exercises.

Version spring 2003

This version adds no functionality, only translation of Dutch parts to English. This has been done by Andres Löh.

Version spring 2004

This version contains an updated version of the compilers. They compile with the most recent version of the AG compiler and source code has been split up to improve modularity and sharing of code between compilers. This work has been done by Wouter Swierstra.

Atze Dijkstra
January 2004

Contents

Chapter 1

Introduction

1.1 Overview

The course *Implementation of Programming Languages* deals with the relation between programs written in a language and the way these programs perform their job on computer hardware. In these lecture notes two topics are dealt with:

- Code generation, the transformation of a program into its machine understandable counterpart (Chapter 6).
- Typing, making sure (or proving) that values are correctly used (Chapter 7).

Other material is included to support these topics. First, a small toy programming language is used (Section 1.2) which serves as the vehicle for the explanation of how compilers work. Parser combinators (Chapter 2) are used to describe syntax in an executable form. An attribute grammar system (Chapter 4) is used to specify the calculations necessary to (for example) generate code and typechecking. A simple compiler for the language described in the following section can be found in Chapter 5.

1.2 Example language: SL

These lecture notes use a small toy programming language to demonstrate issues involving the topics of this course. The language, called *Simple Language* (or *SL*), is informally defined using some examples. This is because SL is used to demonstrate the compilation of language constructs. SL is not to be used for serious programming tasks. Nevertheless, many of the standard constructs are available, its notation loosely based on Haskell (or, in general, functional languages) with some Java (or, in general, imperative languages) features added.

In these lecture notes compilers are written, specifically focussed on two problems: code generation and type inference. Because of these different focusses, the presented compilers do not both compile all language features presented here.

Program An SL program is just an expression:

```
5
```

In this case the expression is just an integer constant. Depending on the compiler and an execution environment the value 5 will be returned and possibly also be printed on output.

Operators and operands SL allows the use of operators:

```
4 * 3 - 5 / 6
```

with result `12`, or

```
3 < 4 && True
```

with result `True`

Types and expressions SL expressions have types. The following types are used:

- Integer type is denoted by `Int`. Integer expressions themselves are denoted as usual.
- Boolean type is denoted by `Bool`. Boolean expressions are denoted with `True` and `False`.
- Function types are denoted by t_1 `->` t_2, where t_1 and t_2 are types. Function (or lambda) expressions are described later.
- Tuple types are denoted by `(`t_1`,`t_2`)`, where t_1 and t_2 are types. Tuple expressions are denoted similarly by `(`e_1`,`e_2`)`, where e_1 and e_2 are expressions.

Conditional A conditional expression allows making a choice:

```
if 3 < 5 then 1 else 2 fi
```

Let A let expression (or block expression) allows variables to be defined:

```
let i :: Int = 3
  ; j :: Int = i + 1
 in i * j
 ni
```

The declaration of an identifier specifies its type (after `::`) as well as its initial value (after `=`). Depending on the compiler the type specification may be omitted.

Assigments and Sequencing Assignments (`:=`) allow the modification of variables:

```
let i :: Int = 3
  ; j :: Int = i + 1
 in j := j + i
  ; i * j
 ni
```

Assignments and other expressions are sequenced, that is, their evaluation order (in time) is specified, by using `;`. The last expression of a sequence is considered to be the result of the sequence.

Assignments and sequences only make sense in an imperative interpretation which allows side effects.

Functions A function encapsulates a calculation for reuse at multiple locations within a program (defining the factorial function):

```
let fac :: Int -> Int =
            \n -> if n > 0 then n * fac (n-1) else 1 fi
 in fac 4
 ni
```

A function is a variable which has as its value a lambda expression. A lambda expression is denoted by `\`p `->` e where p is the parameter and e is the resulting expression, possibly using the value of p.

A function may take multiple parameters:

```
let add :: Int -> (Int -> Int) = \x -> \y -> x + y
 in add 3 4
 ni
```

The function `add` is a function taking one parameter `x` and returning a function which will add to its parameter `y` the value of `x`. The parentheses in the type may be left out since `->` is right associative.

The lambda expression in the example also could have been written as:

```
let add :: Int -> (Int -> Int) = \x y -> x + y
 in add 3 4
 ni
```

Now `add` is a function taking two arguments and returning the addition. For the example this makes no difference, but for the compiler and an implementation it may make a difference.

Chapter 2

Parser Combinators

In this chapter we document the *parser combinator* library `UU_Parsing`. Besides this we provide many hints on its use, the extensions, and derived libraries. Where appropriate we will provide caveats, and hint at the implementation.

2.1 Introductory Notes

Preliminary note concerning the version and status of this chapter. *This chapter is still based on an older version of the parser combinator library. However, the interface of the library has not been changed significantly since this older version. From the material present here only the topics concerning the usage of the library should be considered valid, material related to the implementation (like error recovery) is interesting to read as a subject on its own, but differs in many aspects from the current implementation.*

For whom is this manual written? In this manual we try to (incrementally) collect all information about the use, the design considerations, and internal workings of the `UU_Parsing` library. As such it provides:

- A short and quick introduction to first time users, with a description of the basic interface. However, some familiarity with parser combinators is assumed.
- A more detailed description of the more advanced features, with many examples of how to use them.
- A section dealing with common mistakes.

Why these Combinators? Have you always been intrigued by combinator parsers because they allow you to:

- Use the abstraction, typing and naming mechanism of Haskell[27].
- Create parsers on the fly, that is, at execution time of a program.
- Keep life simple by not having to run a separate program in order to generate a parser.
- To use the same formalism for describing scanners and parsers.
- Use the same formalism for describing semantic routines and parsers.
- To work with (limited forms of) infinite grammars.

but did you not like:

- Slow backtracking implementations.
- Bad error reporting and error recovery properties.
- LL(1) or LALR(1) restrictions on grammars.
- Parsers that use incredible amounts of heap space and hang onto their input.

then these parser combinators might be something for you.

With the basic combinators you can easily write parsers that closely resemble the corresponding context free grammars. Once you get more experienced and your grammars become larger, you may want to learn more about the most effective way of writing your grammars and about the amount of heap space used by the constructed parsers. We want to stress however that you should first get the software, install it, and run some small examples.

Do not immediately proceed to large examples however, without going through these notes. It is as with many other powerful tools: at first the user gets the feeling of being omnipotent, learning about the dangerous aspects the hard way afterwards. So you may easily end up with parsers consuming enormous amounts of heap space or running extremely slow, or even worse get error messages from the runtime system like stacks running into heaps, or control stacks overflow.

We also want to warn you that the implementation contains many subtleties and hidden invariants, that may be easily be destroyed by seemingly innocent modifications.

The prerequisites for reading this tutorial is a basic knowledge of Haskell programming and some idea about the use of combinator parsers, as e.g. provided by [18].

Where to get the software You need the following files:

1. The definition of the basic machinery: `UU_Parsing_Core.hs`.
2. The definition of a collection of derived combinators you are quite likely to find useful: `UU_Parsing_Derived.hs`.
3. A parameterisable scanner `UU_Scanner.hs`, that can be used both as an example of the typical use of the combinators, and as a scanner module for a language with a Haskell-like tokenisation scheme.

Some (somewhat out of date) further description of the internals of the combinators can be found in [31][1] and [30]. You may be interested in our lecture notes on Grammars and Parsing [18] too.

I would be happy it you would let me know:

- Whether you are using the package and what for.
- What modifications you are interested in.
- What modifications you have made yourself.
- Whether you think the package is useful.
- Any interesting examples useful to others.

You can have yourself put on a mailing list (`mailto:doaitse@cs.uu.nl`), and you will be kept informed of new versions, bug fixes, and experiences of others.

[1] courtesy Springer Verlag

2.2 Basic Parser Combinators

The following basic ideas underly the combinator parsers:

- Every non-terminal of the grammar corresponds to a parser.
- Every parser is represented by a (Haskell) function.
- Special functions (called combinators) combine parsers into new parsers.
- Parsers are first class objects, so they may be passed as an argument, and returned as a result. As a consequence your language of Context Free Grammars (e.g. BNF) is extended with the conventional abstraction mechanism available of Haskell.
- Since your parsers are written in Haskell you get the type checking of the interface with the semantic routines for free, and there is no need to run a separate tool for generating the parsers.

The basic parsing technique used follows closely the conventional recursive descent, top-down parsing method. The special properties of advanced functional programming languages like Haskell allow you to write the combinators: parsers are inherently polymorphic functions, and the type inferencing mechanism checks that they are composed in a consistent way. If you make an error in your parser definitions (i.e. in describing your grammar) you will most likely find out through the error messages from the system. It may take a while, unfortunately, to decipher such messages and to interpret them in the context of the grammar formalism you are thinking you are programming in.

For our examples we will assume that you are running the Hugs-system in interactive mode and have loaded the `UU_Parsing_Demo` module. The function `t` has been included to allow you to experiment with writing parsers: it displays the repair steps that were taken in order to make your input be a sentence of the recognized language and the result itself.

Caveat: All parsers must correspond to **non-left recursive** grammars. Since the system cannot discover this, the system also cannot warn you. If you write a left recursive parsers you will most likely find out by a stack overrunning a heap, a control stack overflow, or not getting an error message at all. Some common errors, like parsing a sequence of possibly empty elements, will be flagged by the derived combinators.

Caveat: If you are getting error message when loading the parser modules, you have not set the -98 flag. The combinators use existential types. These are not part of Haskell 98, but can be found in all implementations of the language. When using GHC use the so-called Glasgow extensions.

In the rest of this chapter we will introduce the basic combinators that are defined in and exported from the module `UU_Parsing_Core`.

2.2.1 pSym

After providing:

```
:l UU_Parsing_Demo> t (pSym 'a') "a"
```

the result is:

```
Result:
'a'
```

and after typing

```
> UU_Parsing_Demo> t (pSym 'a') "b"
```

the result is:

```
Errors:
 Deleted : 'b'
 Inserted: 'a'
Result:
'a'
```

It should not come as a surprise that the function `pSym` constructs a parser that recognizes the symbol described by its argument. The parser returns two values:

1. A value acting as a witness that the parsing succeeded, which in this case is the recognized character itself.
2. A list of repair steps that were made made to the input in order to make the input conform to the language described by the grammar. In the second example you can see that such a correcting step may either be the insertion of a missing symbol or the deletion of a superfluous symbol.

You may wonder why, in the example above, the system did not first delete the symbol `'b'` and only then insert the symbol `'a'`; it would take to far to explain such details here already, but it is the purpose of this manual to answer all such questions in greater detail.

The type of the parser constructor `pSym` is quite complicated:

```
pSym :: Symbol symbol => symbol -> Parser symbol symbol
```

We see that `pSym` is a function that is polymorphic in the type `Symbol` and takes a parameter of type `symbol` from which it constructs a `Parser` that returns the recognized `symbol`, which is indicated by the second parameter of the type constructor `Parser`. The first type argument indicates that the input of the parser will be a sequence of `symbol`s.

2.2.2 pSucceed

The parser constructor `pSucceed` returns a parser that recognizes the empty string. Consequently it always succeeds (therefore its name) and returns a value. Its argument is the value that is returned as the witness that recognition has taken place:

```
UU_Parsing_Demo>  t(pSucceed 'a') "b"
Errors:
 Deleted at eof: 'b'
Result:
'a'
```

Note that the argument to pSucceed can be anything, and is in no way related to the type of the input:

```
pSucceed ::  a -> Parser s a
```

2.2.3 <|>

The next combinator we introduce is `<|>`, which combines two productions (represented by their parsers) into a parser recognizing either alternative:

```
UU_Parsing_Demo> t (pSym 'a' <|> pSym 'b') "ab"
Errors:
 Deleted at eof: 'b'
Result:
'a'
```

The type of this combinator is:

```
(<|>) :: Symbol s => Parser s a -> Parser s a -> Parser s a
```

Note that it is required, as is to be expected, that both alternatives return a value of the same type. One might compare this with a similar situation in conditional expressions, where it is required that the **then** and the **else** part return a value of the same type; the constructed parser constructs a value of the same type as its arguments do.

Again you may wonder why the `'a'` was accepted and the `'b'` was deleted. The answer here is that the parsing process is greedy, and proceeds as long as possible without performing repair steps. Repair steps are only considered when no further progress can be made without. Since the `pSym 'a'` parser can succeed, the corresponding choice is made. Only after this alternative has consumed its symbol it is discovered that parsing has finished and that there are still input symbols left, and so these have to be deleted since there is no other way to consume them.

2.2.4 pFail

For reasons that will become clear in a moment the library contains a parser that always fails, and that will generate an error message whenever you call it. It serves as the unit element for the `<|>` operator, so `pFail <|> p` is equivalent to `p`. This fact is indeed discovered by the combinators, and in this case `pFail` will never be called. If it is ever called this indicates that your grammar contains an error, since the system did not manage to eliminate the `pFail` occurrence.

2.2.5 <*>

For the sequential composition of two parsers we use the combinator `<*>`. One of the things to be decided upon in the design of the library is how to combine the results of the two parsers into a result of the resulting parser. We have decided to take the approach in which the result of the first parser is applied to the result of the second parser.

```
UU_Parsing_Demo> t (pSucceed (,) <*> pSym 'a' <*> pSym 'c') "ac"
Result:
('a','c')
```

Integrating the semantic processing with the parsing is usually done by starting an alternative with a parser that always succeeds by returning a function with arity equal to the number of right hand side members of the production. If we want to give a meaning to a conditional expression this is typically done as (the function `pKey` will be introduced later, but speaks for itself here):

```
sem_Expr_If i ce t te e ee f = ...  ce ...  te ...  ee
pIfExpr = pSucceed sem_Expr_If <*> pKey "if"   <*> pExpr
                               <*> pKey "then" <*> pExpr
                               <*> pKey "else" <*> pExpr
                               <*> pKey "fi"
```

You may read this as a function application: the function `sem_Expr_If` is applied to seven arguments, which are the the values that are returned by the seven parsers making up the rest of the parser. It is for this reason that the operator `<*>` was defined to be left associative: otherwise such expressions would be loaded with parentheses.

2.2.6 <..>

One might think that the function `pSym` is primitive in our library, but actually it is not. It is defined by:

```
infixl 5 <..>
(<..>) :: Symbol s => s -> s -> Parser s s
pSym s = s <..> s
pL       = 'a' <..> 'z'
pU       = 'A' <..> 'Z'
pLetter  = pL <|> pU
pDig     = (0 <..> (9::Int))
pAlfaNum = pLetter <|> pDig
```

The operator `<..>` takes a lower and and upper bound, and constructs a parser that recognizes any symbol in that range. For technical reasons also this combinator is not primitive either, as we will see later. You are however unlikely to need the really basic function, so we do not mention it here.

2.2.7 <<|>

As a special version of `<|>` we introduce `<<|>`. This combinator is special in the sense that if, during the parsing process, its left operand can proceed with accepting a symbol its right operand is not taken into account at all. As we will see there are good reasons for having this combinator, since it allows for the formulation of greedy parsing strategies, which will lead to more efficient parsers.

Caveat: On the other hand its semantics deviate from those of normal context free grammars, and as such its use may lead to unexpected results.

2.3 Derived Combinators

Although the combinators described so far can be used directly for constructing parsers there are many situations where this direct use is a bit laborious. In this section we introduce some extra combinators which are all defined in terms of the basic combinators. To understand these derivated parsers no knowledge of the basic parsers (or its underlying machinery) is needed.

2.3.1 Simple Derived Combinators

The <$>, <*, *>, and <$ combinators As we have seen before, we expect to have many occurrences of phrases of the form `pSucceed func <*> ...`. Furthermore, as we have seen in the example of the conditional statement we are not always interested in all values returned by elements of the right hand side of a production: once we have recognized the keywords we are not interested in the values returned by the keyword parsers. To cope with such situations we introduces the following special versions of `<*>`:

```
infixl 4 <$>, <$, <*, *>
f <$> p         = pSucceed f     <*> p
f <$  p         = pSucceed f     <$> p
p <*  q         = (\ x _ -> x)   <$> p <*> q
p *>  q         = (\ _ x -> x)   <$> p <*> q
```

The absence of a `<` or a `>` character indicates that the result that is returned by the parser at that side of the infix operator is of no further interest. As a result we can now formulate the parser for the conditional statement as:

```
sem_Expr_If ce te ee = ...  ce ...  te ...  ee
pIfExpr = sem_Expr_If <$ pKey "if"   <*> pExpr
                      <* pKey "then" <*> pExpr
```

```
                        <* pKey "else" <*> pExpr
                        <* pKey "fi"
```

This notation is probably as close as you can get to the conventional BNF formulation, while still writing a legal Haskell program.

The ‘opt‘ combinator As a further abbreviation we introduce:

```
infixl 2 ‘opt‘
p ‘opt‘ v       = mnz p (p <<|> pSucceed v)
```

The parser constructed with ‘opt‘ optionally recognizes the parser p. That is, if p cannot be recognized at some point in the parsing process, the value v is returned. If p can be recognized the return value of p is returned.

Note that we have formulated this in a greedy way. This is equivalent to the approaches taken in most parser generators where, in the case of a shift-reduce conflict, preference is given to shifting over reducing. If both alternatives fail the repairing process is equivalent to the non-greedy version of the choice combinator.

It is essential that the left operand cannot recognize the empty string. The function mnz (must-not-be-zero) checks for this, and generates an error message if needed. Note that if such an error message is generated, the grammar is ambiguous!

The <> and <??> combinators** Although the combinators analyze your grammar and perform a one-symbol left factorization you can usually do a far better job yourself. In order to facilitate this we introduce the following three combinators:

```
infixl 4 <**>, <??>, <$$>
p <**> q        = (\ x f -> f x) <$> p <*> q
p <??> q        =                    p <**> (q ‘opt‘ id)
f <$$> p        = pSucceed (flip f)  <*> p
```

Suppose we have a parser a with two alternatives that both start with recognizing a non-terminal p, then we will typically rewrite

```
a =       f <$> p <*> q
     <|>  g <$> p <*> r
```

into:

```
a = p <**> (f <$$> q <|> g <$$> r)
```

The combinator <??> takes care of the situation where the parser r is absent.

2.3.2 Sequences and Folding Combinators

When using a conventional parser generator one is usually not restricted to the pure BNF-notation, and can use extensions of this formalism to deal with frequently occurring patterns. Here we see the advantage of the combinator approach: instead of extending the input language of the parser generator, we may simply define new combinators in terms of existing ones, thus moving the facility to capture frequently occurring patterns to the user of the library. Since some patterns are likely to be used by many, we have assembled a set of them in a separate module UU_Parsing_Derived.

pFoldr and variants The basic sequencing combinator is `pFoldr`, that takes as arguments a monoid `alg@(op,e)` and a parser `p`. It recognizes a sequence of `p` derivations, and combines the results with the operator `op`. The value to be used for the empty sequence is `e`.

The combinator `pFoldr` is similar to the Haskell function `foldr` in that both combine elements of a list to a result value. The difference is that `foldr` accepts a list whereas `pFoldr` accepts a parser which will produce the elements of a list. The length of this list is determined by the parsing process.

Since we are almost always interested in matching the longest possible prefix of the input, we have formulated `pFoldr` in a greedy way: as long as the next input symbol can be recognized by `p` we continue constructing the list.

If you are really interested in the simultaneous parsing of another `p` and the parsers following the `pFoldr` application you should use `pFoldr_ng`, in which the *ng*-suffix stand for "non greedy"; in this case all possible parsing paths are explored in a breadth-first way, until only one possibility is left over.

```
pFoldr_gr      alg@(op,e)     p = mnz p pfm
                                  where pfm = (op <$> p <*> pfm) `opt` e
pFoldr_ng      alg@(op,e)     p = mnz p pfm
                                  where pfm = (op <$> p <*> pfm)  <|> pSucceed e
pFoldr         alg            p = pFoldr_gr alg p

pFoldr1_gr     alg@(op,e)     p = op <$> p <*> pFoldr_gr  alg p
pFoldr1_ng     alg@(op,e)     p = op <$> p <*> pFoldr_ng  alg p
pFoldr1        alg            p = pFoldr1_gr alg  p
```

The following combinators construct folding parsers, but take an extra argument that is a parser that should be used between the elements we are interested in. The results of these parsers (e.g. semicolons) are discarded and do not show up in the result.

```
pFoldrSep_gr  alg@(op,e) sep p
               = mnz p ((op <$> p <*> pFoldr_gr alg (sep *> p)) `opt` e         )
pFoldrSep_ng  alg@(op,e) sep p
               = mnz p ((op <$> p <*> pFoldr_ng alg (sep *> p))  <|>  pSucceed e)
pFoldrSep     alg        sep p
              = pFoldrSep_gr alg sep p

pFoldr1Sep_gr alg@(op,e) sep p = if acceptsepsilon sep then mnz p pfm else pfm
                                 where pfm = op <$> p <*> pFoldr_gr alg (sep *> p)
pFoldr1Sep_ng alg@(op,e) sep p = if acceptsepsilon sep then mnz p pfm else pfm
                                 where pfm = op <$> p <*> pFoldr_ng alg (sep *> p)
pFoldr1Sep    alg        sep p = pFoldr1Sep_gr alg sep p
```

The predicate `acceptsepsilon` checks if a parser can accept the empty string of symbols (called epsilon), and is used in combination with `mnz` to check for erroneous situations.

Since we quite often want to construct a list as a result of a sequence parser we define special functions for this:

```
list_alg = ((:), [])

pList_gr        p = pFoldr_gr     list_alg   p
pList_ng        p = pFoldr_ng     list_alg   p
pList           p = pList_gr p

pList1_gr       p = pFoldr1_gr    list_alg   p
pList1_ng       p = pFoldr1_ng    list_alg   p
pList1          p = pFoldr1_gr    list_alg   p
```

```
pListSep_gr   s p = pFoldrSep_gr  list_alg s p
pListSep_ng   s p = pFoldrSep_ng  list_alg s p
pListSep      s p = pFoldrSep_gr  list_alg s p

pList1Sep_gr  s p = pFoldr1Sep_gr list_alg s p
pList1Sep_ng  s p = pFoldr1Sep_ng list_alg s p
pList1Sep     s p = pFoldr1Sep_ng list_alg s p
```

Examples

```
pSpaces   = pList pSpace
```

`pChainr`, `pChainl` and variants In the previous sequencing combinators like `pFoldr` the operator which combines results was passed as an argument. Here we introduce the combinators in which the separator also has a meaning. We handle this by having the parser for the operator return a binary function. This function is then applied to the results of the operand parser:

```
pChainr_gr op x    =  if acceptsepsilon op then mnz x r else r
                      where r = x <??> (flip <$> op <*> r)
pChainr_ng op x =  if acceptsepsilon op then mnz x r else r
                      where r = x <**> ((flip <$> op <*> r)  <|> pSucceed id)
pChainr    op x = pChainr_gr op x

pChainl_gr op x    =  if acceptsepsilon op then mnz x r else r
                      where
                       r      = (f <$> x <*> pList_gr (flip <$> op <*> x) )
                       f x [] = x
                       f x (func:rest) = f (func x) rest

pChainl_ng op x    =  if acceptsepsilon op then mnz x r else r
                      where
                       r      = (f <$> x <*> pList_ng (flip <$> op <*> x) )
                       f x [] = x
                       f x (func:rest) = f (func x) rest
pChainl    op x    = pChainl_ng op x
```

An example of the use of `pChainr` is when using the combinators to describe a scanner, where the parser `symbols` only returns the elements we are interested in. In the example the `drop`-symbols are to be recognized, but should not show up in the result:

```
valid  = (\v l r -> v:r) <$> (   pStrings  <|> pChars  <|> pDigits <|> pSymbols
                             <|> pOperator <|> pInfxOp <|> pVarids <|> pKeywords
                             <|> pConids
                             )
drop   = const [] <$> pList (pSpaces <|> pNewLine <|> pComments)
symbols = pChainr valid drop
```

Each recognized `valid` symbol is returned as a function value, that discards the drop-symbols recognized to its left (`l`) and prefixes the valid symbols recognized to its right (`r`) with the recognized valid symbol (`v:`).

As a second example we provide a parser for an expression language with a number of different operator priorities (for the sake of simplicity we assume all operators to be left associative). The example is taken from Section 6.4.

```
pExpr
  =     pAndPrioExpr
```

```
        <|> sem_Expr_Lam
            <$ pKey "\\"  <*> pFoldr1 (sem_LamIds_Cons, sem_LamIds_Nil) pVarid
            <* pKey "->"  <*> pExpr
    pAndPrioExpr
      = pChainl (sem_Expr_Op <$> (pOper "&&" <|> pOper "||")) pCmpPrioExpr
    pCmpPrioExpr
      = pChainl (sem_Expr_Op <$> (   pOper "=="
                                 <|> pOper "/="
                                 <|> pOper "<"
                                 <|> pOper ">"
                                 <|> pOper "<="
                                 <|> pOper ">="
                                 )
                ) pAddPrioExpr
    pAddPrioExpr
      = pChainl (sem_Expr_Op <$> (pOper "+" <|> pOper "-")) pMulPrioExpr
    pMulPrioExpr
      = pChainl (sem_Expr_Op <$> (pOper "*" <|> pOper "/")) pLamApply
    pLamApply
      = (\fs -> case fs of
                  [fe]     -> fe
                  (fe:fes) -> sem_Expr_Lamcall (foldl1 sem_Expr_Apply fs)
        ) <$> pList1 pFactor
    pFactor
      =     pParens pExpr
        <|> (sem_Expr_Intexpr . string2int) <$> pInteger
        <|> sem_Expr_Boolexpr True  <$  pKey "True"
        <|> sem_Expr_Boolexpr False <$  pKey "False"
        <|> sem_Expr_Ident <$> pVarid
        <|> sem_Expr_If
            <$ pKey "if"    <*> pExpr
            <* pKey "then"  <*> pExpr
            <* pKey "else"  <*> pExpr
            <* pKey "fi"
        <|> sem_Expr_Let
            <$ pKey "let" <*> pDecls
            <* pKey "in"  <*> pExprSeq
            <* pKey "ni"
```

pAny and pAnySym

When constructing the expression parser the following pieces of text are encountered:

```
(   pOper "=="
<|> pOper "/="
<|> pOper "<"
<|> pOper ">"
<|> pOper "<="
<|> pOper ">="
)
```

In these situations `pAny` can be used:

```
pAny  f l = if null l then usererror "pAny: argument may not be empty list"
                      else foldr1 (<|>) (map f l)
pAnySym l = pAny pSym l
```

so we may now write:

```
pOps = pAny pOper
pCmp = pOps ["==", "/=", "<", ">", "<=", ">="]
```

Using some further abstraction we can now replace the whole hierarchy of functions that were introduced for describing the different priority levels directly by:

```
pAndPrioExpr
  = foldr (.) pLamApply
          (map pChainl (map (sem_Expr_Op.pOps) [ ["==", "/=", "<", ">", "<=", ">="]
                                               , ["&&", "||"]
                                               , ["+" , "-" ]
                                               , ["*" , "/" ]
                                               ]
                       )
```

2.4 Error Recovery

In general you will not have to worry too much about controlling your error recovery, and we have taken some precautions to guarantee that the repair process will not derail. Nevertheless it is good to have some basic understanding of how the recovery process works.

As we have seen a parser generates a list of error messages, indicating the corrections made to the input. There are two kind of correcting steps:

1. insertion steps, that insert a symbol into the stream of input symbols
2. deletion steps that remove a symbol from the input stream

The underlying idea of the repairing process is the concept of an editing sequence, which may be defined as:

```
data EditStep s = Shift
                | Insert s
                | Delete
type EditSeq s = [EditStep s]
```

with which an editing function can be defined that takes an `EditSeq s` and a list of tokens `[s]` and returns an updated sequence as follows:

```
edit :: EditSeq s -> [s] -> [s]
edit []         inp  = inp
edit (Insert a:r) inp  = a:edit r inp
edit (Shift   :r) (i:ii) = i:edit r ii
edit (Delete  :r) (i:ii) =   edit r ii
edit _            []     = undefined
```

One can now think of the repairing process as the construction of an editing sequence that maps the input into a sentence of the recognized language. Of course there may be many different editing sequences doing so. Ideally we should like to take the least costly one for some definition of a function:

```
cost :: EditSeq s -> Int
```

In the current version of our combinators we have chosen to assign a fixed cost[2] to the repairing steps and zero cost to the `Shift` step. Since finding the least costly editing sequence may be

[2] In an earlier version of our combinators the user of the library could specify these costs. For the sake of efficiency and because this freedom was abused quite a bit – leading to complicated questions – we have chosen to fix these costs and to provide some other ways of specifying e.g. greedy behaviour.

prohibitively expensive we order the editing sequences lexicographically, and construct the smallest one in that order (later we will adapt this a bit).

As an example consider:

```
lift x = [x]

b  = lift <$>  pSym  'b'
a  = lift <$>  pSym  'a'

ab  = (++) <$> a   <*> b

ab_2  = (++) <$> ab   <*> ab  <|> ab
```

The parser `ab_2` accepts both "ab" and "abab". The question is what happens with the input "abb": will the last 'b' be deleted or will an 'a' be inserted in front of it? Since both repairs are just as expensive preference is given here to the shortest alternative, and thus we get:

```
UU_Parsing_Demo> t ab_2 "abb"
Errors:
 Deleted at eof: 'b'
Result:
"ab"
```

Now let us try:

```
UU_Parsing_Demo> t ab_2 "aba"
Errors:
 Inserted: 'b' before eof
Result:
"abab"
```

Although we again here have two possibilities, i.e. deleting the extra `a`, or inserting a `b` at the end, now the second alternative is chosen. This is caused by the lexical ordering of the editing sequences that makes the underlying parsing machine greedy: as long as it can make progress without performing repairs that alternative is chosen. So here the second `a` actually forms the decision point at which in chosen for the second alternative of `ab_2`.

A more interesting example is formed by:

```
aaa    = a <* a <* a
bbb    = b <* b <* b
aaabbb = aaa <|> bbb
UU_Parsing_Demo> t aaabbb "xabb"
Errors:
 Deleted : 'x' before 'a'
 Deleted : 'a' before 'b'
 Inserted: 'b' before eof
Result:
"b"
```

Here we see that the repairing process is quite subtle: instead of just discarding the superfluous `x` and then greedily accepting the `a`, it discovers, by looking ahead a few steps more, that also discarding the `a` will lead to a lower total repair cost. This extended look-ahead is only done when it has been concluded that a repair is unavoidable. The current combinators base the decision on the accumulated repair cost for the next four parsing steps.

2.5 Scanner

When using combinator based parsers it is common practice to embed the scanner in the parser, and to do away with a special lexer. This is possible because many peculiarities associated with tokenizing an input stream can be dealt with by the introduction of special combinators, which perform semantic checks on the possible tokens. Such combinators extend the expressive power of the formalism far beyond what is expressible in conventional BNF. So it is quite easy to construct a combinator that will accept "if" as a keyword and not as an identifier, but expressing the same fact in ordinary BNF is quite a challenge and definitely no fun.

Since the `UU_Parsing` parsing combinators analyse a grammar it is required that the underlying formalism is much closer to ordinary BNF and assumed that there are no hidden semantic conditions. This makes combining a scanner and parser more difficult. In this case a better approach is to use a separate scanner in order not to burden a parser with the tasks a scanner usually does: recognizing individual symbols, also called *lexeme*s.

Another reason for using a separate scanner is that a proper lexer (scanner) does much more than just tokenizing the input stream: it keeps track of line numbers, column numbers, layout information for handling the offside rule, the name of the file the token is coming from and knows about (nested) comment conventions.

In order to make it easy to construct scanners for use with our parser combinators we provide a parameterisable scanner, which – if needed– can be easily adapted to different lexing conventions.

The main function exported from `UU_Scanner` is the function scanner:

```
scanner :: Bool             -- should the scanner recognise literal text or not
           -> [String]      -- those identifiers that are keywords, e.g. "if"
           -> [String]      -- those operators that are keywords, e.g. "=" and "->"
           -> String        -- those characters that stand by themselves, e.g. '('
           -> String        -- those characters that are used to construct operators
           -> String        -- the name of the file to be tokenised
           -> Maybe String  -- if the file was already opened
                            -- then Just  <rest_of_input> else Nothing
           -> IO [Token]    -- the resulting IO action, returning a sequence of tokens
```

For the small, Haskell-inspired language, for which fragments of the compiler (see also Chapter 1, Chapter 6 and Chapter 7) were given above, such a specialization looks like:

```
keywordstxt      = [ "if", "then", "else", "fi"
                   , "while", "do", "od"
                   , "let",  "in", "True", "False"
                   , "module", "infixl", "infixr", "infix"]
keywordsops      = ["=", "\\", "->", "@", "?"]
specialchars     = "();,"
opchars          = "!#$%&*+/<=>?@\\^|-"
```

The main combinators exported from this module are:

```
pKey   :: String -> Parser Token String  -- parse a keyword
pOper  :: String -> Parser Token String  -- parse an operator
pVarid ::           Parser Token String  -- parse a lower case identifier
pConid ::           Parser Token String  -- parse an upper case identifier
pSpec  :: Char   -> Parser Token String  -- parse a special character

pString    :: Parser Token String -- e.g. "doaitse"
pChar      :: Parser Token String -- e.g. '\n' or 'a'
pInteger8  :: Parser Token String -- e.g.  0o7 or 007
pInteger10 :: Parser Token String -- e.g.  3457
pInteger16 :: Parser Token String -- e.g.  0xA
```

```
pTextnm    :: Parser Token String -- from \\begin{string}
pTextln    :: Parser Token String -- line from a text block

pInteger       =   pInteger10
pComma :: Parser Token String
pComma  = pSpec ','
pSemi   = pSpec ';'
pOParen = pSpec '('
pCParen = pSpec ')'
pOBrack = pSpec '['
pCBrack = pSpec ']'
pOCurly = pSpec '{'
pCCurly = pSpec '}'
```

And using these, the following extra combinators are exported that are covering some prominent cases:

```
pCommas  = pListSep pComma
pSemics  = pListSep pSemi
pParens  = pPacked pOParen pCParen
pBracks  = pPacked pOBrack pCBrack
pCurly   = pPacked pOCurly pCCurly

pParens_pCommas :: Parser Token a -> Parser Token [a]
pBracks_pCommas :: Parser Token a -> Parser Token [a]
pCurly_pSemics  :: Parser Token a -> Parser Token [a]
pParens_pCommas = pParens.pCommas
pBracks_pCommas = pBracks.pCommas
pCurly_pSemics  = pCurly .pSemics
```

2.6 A Bit of Internal Machinery

The underlying basic machinery of the parser combinators actually is quite complicated. Fortunately one does not have to understand everything in order to be able to interface with it. Some understanding may however help to understand error messages and how to make use of the low level interface. The description will be given at two levels. The first level describes the interface to the basic combinators, and how to access them. The second level contains a description of the real machinery, and provides sufficient understanding of what is needed in case one wants to extend this or make changes.

2.6.1 Level 1

The class `Symbol`

At many places of the basic machinery we need information about the kind of symbols that we are actually parsing. The class `Symbol` is used to parameterize the parsing process at several different places:

```
class (Ord s,Show s) => Symbol s where
```

In the first place we require that the symbols to be recognized can be ordered in some way. Strictly spoken this requirement is bit stronger than one might expect: equality should be sufficient for a parser. Since we use binary search trees internally we have to put up this stronger `Ord` requirement. The resulting speed improvement is likely to be preferred over this small loss of generality. Furthermore we need to be able to convert a symbol into a string in order to generate error messages.

Ranges

The basic combinators are expressed in terms of ranges instead of symbols. Ranges may include their bounding values or not. The data type `OC` is used to indicate whether this is the case or not:

```
data OC = RO -- )
        | CL -- [ and ]
        | LO -- (
        deriving (Show,Eq,Ord)
type Point s = (s,OC)
data  SymbolR s  =  Range (Point s) (Point s)
                 | EmptyR
                 deriving (Show,Eq,Ord)
mk_range l@(lv, lb) r@(rv,rb) = if lv > rv then EmptyR
                                           else Range l r
```

So the range [3..5) is represented by `Range (3,CL) (5,RO)`.

One will usually not use the open ranges, but they naturally show up when taking intersections of ranges, which is done when computing common prefixes of parsers.

pLRange

The basic parsing function is pLRange:

```
pLRange :: Symbol s
        => SymbolR s  -- the range to be recognized
        -> s          -- the symbol used as an insertion symbol
        -> Parser s s -- the constructed parser
```

Using this basic combinator we can now express more commonly used parsing combinators:

```
pRange lb l rb r ins_sym = pLRange (mk_range (l,lb)      -- left symbol left bound
                                             (r,rb))     -- right symbol right bound
                                   (to_inserted ins_sym) -- insertion symbol
a <..> b        = pLRange (Range (a,CL) (b,CL)) a
pSym a          = a <..> a
```

2.6.2 The type `RealParser`

The parser combinators do not only construct the actual parsers, but tuple them with a `Nat` describing the minimal length of a sentence for the associated non-terminal and a data structure of type `Parser` that contains information collected about the actual parser. The function used in the parsing process is of type `RealParser`:

```
newtype RealParser s a = P (forall r .([s] -> Result r) -> [s] -> Result (a,r))
type Result v = ((v,String),[Int])
data Parser     s a= Parser { leng   :: Nat
                            , p      :: RealParser s a
                            , descr  :: Descr s a
                            }
```

A `RealParser` is continuation based: its first argument is the parser to be called when the parser has completed its job and the second argument the list of symbols to be parsed.

The `Result v` is a cartesian product containing:

1. The computed result of type `v`.

2. A `String` describing what repairs were made to the input during parsing.

3. A sequence of editing step costs.

The use of the continuation is a bit unusual, in the sense that the result of the continuation call is modified before being passed back to the caller: a parser prefixes the result of the future parsing –that is returned by its continuation– with its own contribution. If the continuation constructs a value of type `Result r` then the parser of type `Realparser s a` will return a value of type `Result (a, r)`. Lazy evaluation makes that we can access part of this result without the continuation actually having returned anything yet. One might be inclined to think that we are dealing here with a conventional parse stack containing the intermediate results computed thus far. However, this is not the case because a parser pushes its result onto a "stack" of still to be computed values instead of already computed values!

Calling `RealParser` In the module `UU_Parsing_Core` we have defined a function `parse` that may be used to call a real parser. It may be used as it is or be used as a starting point to define other parser calling function.

```
parse (Parser _ root@(P rp) _) inp
 = let noErrors             = ""
          noEditingSteps       = []
          emptyStack           = ()
          handle_eof           = foldr delete'
                                       ((emptyStack, noErrors), noEditingSteps)
          (((res,()),errors),_)= rp handle_eof inp
   in (res,errors)
```

The function parse:

1. Selects the `RealParser` from the `root` parameter, names it `rp` and applies it to the continuation `handle_eof` and the provided input, and

2. Discards the editing steps and the bottom element of the stack of results, on top of which the desired result `res` is found.

3. Returns the result we are interested in and the generated error messages.

The `handle_eof` continuation discards any unconsumed input tokens by calling the function `delete'` for each of them.

2.6.3 Level 2

The type `Parser`

We will not go into the details of the type `Parser`, but only explain shortly what its three components stand for.

```
data Parser     s a= Parser { leng   :: Nat
                            , p      :: RealParser s a
                            , descr  :: Descr s a
                            }
```

The first component describes the minimal length of the sequence accepted by this parser. It is used in the repairing process to make sure that it will always terminate, despite a limited look-ahead. The formulation is typical for the techniques used in the core implementation:

```
data Nat  = Zero
          | Succ Nat

length_le Zero       _      = True
length_le _          Zero   = False
length_le (Succ l) (Succ r) = length_le l r

shortest Zero _     = Zero
shortest _    Zero  = Zero
shortest (Succ l) (Succ r) = Succ (shortest l r)

infinite = Succ infinite

addnat Zero r = r
addnat (Succ l) r = Succ (addnat l r)
```

These definitions allow us to add and compare natural numbers, some of which are infinite. We can however, with this representation, compute the minimum of some `Nat` values, provided they are not all infinite. Here we rely critically on the fact that our grammars must not be left recursive. In the function `addnat` we assume that the first argument is indeed available for pattern matching, and this is not always the case for a left recursive grammar.

Based on this component we may check whether a parser can recognize the empty sequence, and provide error messages if we try to recognize sequences of such empty sequences.

```
mnz p v = -- must not be zero
   if acceptsepsilon p
   then usererror (   "You are calling a list based derived combinator"
                   ++ "with a parser that may accept the empty string.\n"
                   ++ "This results in a hidden left recursive formulation"
                   ++ "(and is ambiguous too).\n"
                  )
   else v

acceptsepsilon (Parser Zero _ _) = True
acceptsepsilon _                 = False

usererror m = error ("Your grammar contains a problem.\n" ++m)
```

2.7 Tips

Since many tables are constructed and Hugs has a tendency to keep such data structures in memory it is a good idea to define subsidiary parsers inside a separate `where` construct, and let only the parser for the start symbol of your grammar be visible at the outside level.

It never does any harm to do some left-factorization, and to try to make the grammar as close to LL(1) as you can.

2.8 Exercises

2-1 **Pocket Calculator**. Use the parser combinators to write a parser for arithmetic expressions built from (constant) integer operands, the unary operators + and −, and the binary operators +, −, ∗, /, and parentheses. The operators should have the usual priortities. The parser must return the value of the expression – it is a bit like a pocket calculator.

2-2 **Pocket Calculator with Reverse Polish Notation (RPN) as output**. Modify the parser from exercise 2-1 in such a way that it no longer returns the value of the expression, but rather translates the expression into RPN (Reversed Polish Notation). In RPN, the operators are not written in between the arguments, but in the end. (This system is also used on HP calculators.) To distinguish the unary from the binary −, the unary minus should be represented as `~`. The unary + has to be removed. The advantage of RPN is that you no longer need parentheses. Therefore, parentheses have to be removed as well. Let the parser write each number and each operator on a separate line.

Example: Parsing `37*(237+18)/(-5)` returns:

```
37
237
18
+
*
5
~
/
```

2-3 **Pocket Calculator with Memory**. Modify the parser from exercise 2-1 in such a way that you can use 'memory slots'. Memory slots can be referred to by identifiers. Identifiers have the same syntax as variables names in Haskell or Java. A value can be stored in a memory slot by means of an *assignment*:

```
identifier = expression
```

In expressions, now also identifiers may occur, but only if there has been an assignment to that identifier before.

The parser now has to accept a sequence of assignments and expressions, separated by ';'. The results of the expressions in that sequence are returned.

Question: Is it possible to accomplish the same even without using ';'s as separators?

2-4 **Pocket Calculator with RPN and Memory**. Combine exercises 2-2 and 2-3.

Chapter 3

Syntax-Directed Translation

3.1 Beyond Parsing

Soon after it was discovered that it was possible to generate parsers from the description of of a language by a context free grammar the question arose how to do something similar for those aspects of a programming language that were beyond the expressive power of a context free language. Aspects to be dealt with were things like name processing, type checking, type inferencing, optimizations and code generation.

After many years of separate development several lines now seem to come together. In the next subsections we will shortly describe the different, rather independently evolving areas of research, and show how they are related.

3.1.1 Attribute Grammars

Attribute grammars (and similar formalisms like affix grammars) can be seen as a straightforward continuation of the use of parser generators. Parser generators construct parse trees and it seems only logical to proceed from there. The basic idea of an attribute grammar is that we start from a parse tree, and by describing how the nodes of the tree should be decorated with values, we describe further computations. Essential is that the description of what values such attributes should take is done either in terms of attributes of the children (*synthesized attributes*), or by attributes of the father (*inherited attributes*). Essential is that the description is in both cases a local affair, and that global analyses of the the total tree is done by describing how values are passed around along the branches of the tree.

This approach has been quite successful, and after many years of research has resulted in efficient systems that generate so-called tree walking automata that know about the evaluation order of the attributes, the lifetimes of attributes, and where to store and how to access them efficiently. The overall goal of this research has been to make the resulting compilers about as efficient as hand-written ones or even faster. Most compiler writing systems nowadays take some form of attribute grammar as input, perform an extensive analysis of the way attributes are expressed in terms of each other, and generate efficient evaluators.

Since the bulk of the work in compiling a program is not so much in the scanning or the parsing phase, as in the later phases, this approach is often referred to as *syntax directed translation*, making explicit that the parse tree – described by the syntax of the language– is taken as the starting point for the translation process.

Using attribute grammars has always been limited by the need to choose a specific language for describing the semantic functions and a specific target language. Fortunately, as we will show, it is nowadays quite straightforward to use the attribute grammar based way of thinking when

programming in the setting of a modern, lazily evaluated functional language: it is the declarative way of thinking in both formalisms that bridges the gap, and when using Haskell you get an attribute grammar evaluator for free ([19, 24]).

3.1.2 Algebraic Approach

A second line of development comes from the concept of an initial algebra, that closely correspond to an algebraic data type in a functional language. The syntax directed translation is here called *compositional description*, where we compute values associated with trees in terms of values computed for the children of the root of the tree. The way we look at it is not so much as assigning a value to a node as assigning a value to the whole tree. In the course on grammars and parsing [18] we have seen many examples of this approach, in which:

- the default "control structure" expressing the compositionality property, was a catamorphism (fold-function)
- the description how to compute a value for a father node in terms of the values computed for the children nodes was described by an algebra, and
- by going to higher order domains (i.e. assigning function values instead of just first order values) we could mimic the passing down of values along the branches of the tree. An example of this was given in the *repmin* example, and in the compiler for the small expression language.

One of the essential differences between the two approaches is that in most attribute grammar systems, as the result of the analysis, a forward evaluation order is constructed that describes how to compute further attributes in terms of already computed ones. The resulting evaluation order can be seen as a *data driven* computation. The formulation in a functional language, using the higher order domains, is essentially a demand driven computation. The reliance on a lazy evaluation strategy makes it straightforward to pass (part of) a computed result back as an argument to a function assigned to a node of the parse tree.

3.1.3 Aspect Oriented Programming

With software systems getting more and more complex the question arises how to master this complexity, and how to make changes in the implementation manageable, and how to develop parts of the system in relative isolation. It is here that recently the term *aspect oriented programming* was coined. Although there does not yet exists a formal definition of this concept, we claim that this concept has been with us for many years in the form of attribute grammars.

Writing a compiler for a modern programming language is no easy task, and deals with many different aspects:

- *name analysis*: what identifiers are introduced, where may they be used and under what restrictions or conditions.
- *type analysis*: we have to process the defined types, combine information that is provided at many different places in the program, and link this information with the name analysis.
- *providing error messages*: it is a fact of life that most compiler runs are dealing with incorrect input, since once the program is compiled successfully there is no need for recompilation. Despite the fact that not all language processing systems put priority here we think that the user of the system deserves as much feedback about the discovered error and the possible sources for the inconsistency as possible. The sparseness of many error messages generated by many contemporary systems still leaves much to be desired for.

- *labels*: in the course of code generation, labels have to be constructed in order to compile conditional and looping statements.
- *pattern matching*: many languages now contain a direct link between the expression language and the type language. In function definitions and case statements information from the type system may be referred to.
- *optimization*: the program may be transformed into a more efficient form. Although a compiler may be required to be able to process any correct input program, the program at hand is not just such a most general program, but only a special case of it. Quite often such special cases do not have to follow the most general compilation scheme, and much more efficient code can be generated for the special cases at hand. Common subexpressions may be joined, constant subexpressions may be evaluated by the compiler, functions may be inlined, data types may be deforestated, constants may be propagated, recursive formulations may be transformed into iterative ones, etc.
- *program specific checks*: the organization a programmer is working for may have decided not to use the full generality of the language, and have defined so-called "coding standards" which all programs are requested to adhere to. So one might require that every p-operation that claims a resource is paired with an associated v-operation in the same a procedure body that releases the resource. Another example is the requirement that all declared variables are properly initialized at the beginning of the block in which they have been declared.
- *checking code*: it is quite common for compilers to emit extra code that checks for specific conditions. We may e.g. require that the value of an array indexing expression is indeed between the array bounds, and if this cannot be verified statically by the compiler, extra code is to be generated.
- *condition generation*: a compiler may try to verify that specific conditions hold for the program. It is however quite likely that the system cannot verify all such conditions automatically, and that help from a human being has to be called for. In such a case a compiler may emit predicates that are verified offline, e.g. with the help of a proof checker.

It should be clear from the above list that writing a production quality compiler for a real language is no small task, and that any methodology that enables us to attack the total task in a number of phases could be of great help. An essential property of all the above aspects is that:

1. on the one hand we want to see them as issues to be dealt with in as much isolation of the other aspects as possible, and that
2. on the other hand not all these aspects are completely independent, and some information flow between the different aspects should be describable

Ideally we should be able to write a compiler taking care of all the above mentioned aspects except e.g. the last one, and then later be able to add this last aspect **without touching the existing code**.

The approach we take in these lecture notes is based on the two above approaches, in which we take the aspect oriented nature of attribute grammars, and combine it with an implementation model based on the algebraic approach. In this way we profit from the convenient attribute grammar notation, while on the other hand we profit from existing language processing systems like Haskell, in which care is taken of the scheduling of the computation and things like type checking etc.

3.1.4 Overview of the rest of this chapter

Based on the above starting points we will now describe different solutions to an example problem. In the example, we will make use of the attribute grammar notation of the `UU_AG` system. This system is described in Chapter 4.

3.2 Making a one time investment

3.2.1 Problem description

This problem is also known under the name *maximal segment sum*, and has served as an example for many years now in the area of the study of programming methodology. Here we have given it a more modern name, referring to the stock exchange (something which interests us all).

Suppose you are given the number of points the Dow Jones has gone up or down over the last ten years, and you are asked to compute the maximum profit you could have made by investing once a sum of money and selling it again at a later point in time.

3.2.2 Functional Solution

Both solutions are based on the simple, but effective investment strategy: "buy low, sell high". So one of the aspects we are interested in is how low the index has been, but in order to compute this we have to accumulate the given differences first, in order to compute the actual index (keep in mind that we have only been given the relative changes to the previous day).

First we define a data type representing the given input[1], similar to the list data type.

```
data DJSequence a = NextDay a (DJSequence a) | Start deriving Show
foldrDJSequence (_, start) Start = start
foldrDJSequence alg@(nextday, _) (NextDay i h) = nextday i (foldrDJSequence alg h)
```

Accumulating the delta's we can now compute a list containing the actual (Dow Jones) index values, in which we assume that at the start of the period of interest the index was at level 0.

```
dj_index :: DJSequence Int → DJSequence Int
dj_index deltas
  = snd (foldrDJSequence (nextday, (0, Start)) deltas)
    where nextday today (yesterday, history)
            = (index, NextDay index history)
              where index = today + yesterday
```

In the next step we compute for each point in time the cheapest deal we could have made in the past, so we compute the running minimal value (the lowest Dow Jones):

```
dj_lowest :: DJSequence Int → DJSequence Int
dj_lowest index
  = snd (foldrDJSequence (nextday, (0, Start)) index)
    where nextday today (yesterday, history)
            = (lowest, NextDay lowest history)
              where lowest = today `min` yesterday
```

The next aspect to be computed is how much money we could have made on each specific day if we would sell on that day. This aspect depends on two other values, i.e. the lowest price we could have bought for thus far and the current index value. Unfortunately we need for this both previously computed aspects, for which we define an "aspect-combining" function:

```
dj_zip :: DJSequence Int → DJSequence Int → DJSequence (Int, Int)
dj_zip Start Start = Start
dj_zip (NextDay lv lh) (NextDay rv rh) = NextDay (lv, rv) (dj_zip lh rh)
```

Using this function we can now compute for each day in the past how much money we would have made (or lost) if we had sold on that day, assuming we had bought at the lowest point before that:

```
dj_profit :: DJSequence Int → DJSequence Int → DJSequence Int
dj_profit index lowest
= foldrDJSequence (nextday, Start) (dj_zip index lowest)
```

[1] Those who think that time is running from left to right should stop doing so

```
  where nextday (currentindex, currentlow) histprof
    = NextDay (currentindex − currentlow) histprof
```

and finally we can now compute the maximum values of all the potential profits and losses to compute our best investment:

```
dj_maxprofit :: DJSequence Int → Int
dj_maxprofit profits
  = foldrDJSequence (nextday, 0) profits
    where nextday prof hist = prof `max` hist
```

Finally we can now combine all these functions into our main program:

```
dj_bestinvest :: DJSequence Int → Int
dj_bestinvest deltas
  = dj_maxprofit profits
    where profits = dj_profit indexes lowests
          lowests = dj_lowest indexes
          indexes = dj_index deltas
```

The question that arises is whether it is really necessary to iterate six (do not forget the function `dj_zip`) times over a sequence, and the answer is of course "no". By combining the computations we can write directly:

```
dj_bestinvest' :: DJSequence Int → Int
dj_bestinvest' deltas
= maxprofit
  where (maxprofit, lowest, index) = foldrDJSequence (nextday, (0, 0, 0)) deltas
        nextday delta (mp, low, ind) = let newindex = delta + ind
                                       newlow = newindex `min` low
                                       todayprofit = newindex − newlow
                                       maxprofit = todayprofit `max` mp
                                       in (maxprofit, newlow, newindex)
```

Running the following then gives 10 as answer:

```
dj :: DJSequence Int
dj = NextDay 3
  (NextDay 4
    (NextDay 1
      (NextDay (−1)
        (NextDay 3
          (NextDay (−5)
            (NextDay 0
              (NextDay 4 Start
))))))))
dj_bestinvest' dj
```

Let us compare the two approaches now. In the first approach we have managed to keep all the different aspects relatively well separated, but:

1. the resulting program in relatively inefficient, because we have to take extra precautions for routing the available values to the right places and the multiple traversals of the lists.
2. we had to invent an extra function (`dj_zip`) for helping us in routing the information.

One can easily imagine that with more and more aspects playing their role this routing problem becomes more and more a burden for the programmer. On the other hand, if we had written the more efficient program by taking the aspects into account one by one, we would have had to modify the original program several times, each time weaving in another aspect.

In order to have to best of both worlds we introduce a special notation: attribute grammars.

3.2.3 Attribute grammar solution

In an attribute grammar we describe how to decorate a tree, but before doing so we have to define the shape of a tree. The notation we use has been inspired by Haskell but differs slightly for reasons that will become clear later:

DATA *Sequence* | *NextDay delta* : *Int past* : *Sequence*
| *Start*

The identifiers `delta` and `past` refer to the two children of a `Sequence` node. The first aspect we add now is the computation of the current value of the index. Since this value depends only on the value of the children of a node, this is a so-called synthesized attribute, that is specified as follows:

ATTR *Sequence* [{-no inherited attribute -} | {-... -} | *index* : *Int*]

SEM *Sequence*
| *Start* **lhs**.*index* = 0
| *NextDay* **loc**.*index* = @*delta* + @*past*.*index*
lhs.*index* = @*index*

The keyword `lhs` refers to the non-terminal at the "Left Hand Side" of a production, and a rule referring to this element thus defines a synthesized attribute. The right-hand sides of attribute definitions (i.e. the piece of code that is located to the right of the `=` symbol) have to be proper Haskell expressions, and they will show up unmodified in the generated program. In these expressions we may refer to synthesized attributes of the children by using identifiers of the form `@<child name>.<attribute name>`. In order to refer to inherited attributes (which we will not use in our example) we write `@lhs.<attribute name>`. Locally defined attributes (via `loc`) are referred to by `@<attribute name>`.

The next aspect we introduce is the running minimal value:

ATTR *Sequence* [{-no inherited attribute -} | {-... -} | *low* : *Int*]

SEM *Sequence*
| *Start* **lhs**.*low* = 0
| *NextDay* **loc**.*low* = @*index* ‘*min*‘ @*past*.*low*
lhs.*low* = @*low*

To this we add the definitions for the running maximum profits:

ATTR *Sequence* [{-no inherited attribute -} | {-... -} | *maxprofit* : *Int*]

SEM *Sequence*
| *Start* **lhs**.*maxprofit* = 0
| *NextDay* **lhs**.*maxprofit* = (@*index* − @*low*) ‘*max*‘ @*past*.*maxprofit*

The code generated from this input is:

```
data Sequence = Sequence_NextDay (Int) (Sequence)
  | Sequence_Start
  -- semantic domain
type T_Sequence = ((Int), (Int), (Int))
  -- cata
sem_Sequence :: (Sequence) →
  (T_Sequence)
sem_Sequence ((Sequence_NextDay (_delta) (_past))) =
  (sem_Sequence_NextDay (_delta) ((sem_Sequence (_past))))
sem_Sequence ((Sequence_Start)) =
  (sem_Sequence_Start)
sem_Sequence_NextDay :: (Int) →
  (T_Sequence) →
  (T_Sequence)
sem_Sequence_NextDay (delta_) (past_) =
```

```
  let _lhsOindex :: (Int)
      _lhsOlow :: (Int)
      _lhsOmaxprofit :: (Int)
      _pastIindex :: (Int)
      _pastIlow :: (Int)
      _pastImaxprofit :: (Int)
      (_pastIindex, _pastIlow, _pastImaxprofit) =
        (past_)
        -- "DowJones4AG.ag"(line 14, column 14)
      (_lhsOindex@_) =
        _index
        -- "DowJones4AG.ag"(line 13, column 14)
      (_index@_) =
        delta_ + _pastIindex
        -- "DowJones4AG.ag"(line 21, column 14)
      (_lhsOlow@_) =
        _low
        -- "DowJones4AG.ag"(line 20, column 14)
      (_low@_) =
        _index `min` _pastIlow
        -- "DowJones4AG.ag"(line 27, column 14)
      (_lhsOmaxprofit@_) =
        (_index - _low) `max` _pastImaxprofit
  in (_lhsOindex, _lhsOlow, _lhsOmaxprofit)
sem_Sequence_Start :: (T_Sequence)
sem_Sequence_Start =
  let _lhsOindex :: (Int)
      _lhsOlow :: (Int)
      _lhsOmaxprofit :: (Int)
        -- "DowJones4AG.ag"(line 12, column 14)
      (_lhsOindex@_) =
        0
        -- "DowJones4AG.ag"(line 19, column 14)
      (_lhsOlow@_) =
        0
        -- "DowJones4AG.ag"(line 26, column 14)
      (_lhsOmaxprofit@_) =
        0
  in (_lhsOindex, _lhsOlow, _lhsOmaxprofit)
```

Inspection of this output shows that the AG compiler creates a Haskell program which is otherwise difficult to write. Data definitions and types are introduced for the abstract syntax (the DATA definitions), inherited attributes are passed as parameters, synthesized attributes are combined into a tuple and attribute definitions are put together.

Note that by explicitely naming argument and result positions (by the introduction of attribute names) in the AG notation, we are no longer restricted to the implicit positional argument passing enforced by conventional function definitions.

3.3 Full example code

```
{
import UU.Parsing
}
```

```
DATA Sequence | NextDay delta : Int past : Sequence
              | Start
ATTR Sequence [|  | index : Int]
SEM Sequence
  | Start   lhs.index   = 0
  | NextDay loc.index   = @delta + @past.index
            lhs.index   = @index
ATTR Sequence [|  | low : Int]
SEM Sequence
  | Start   lhs.low     = 0
  | NextDay loc.low     = @index `min` @past.low
            lhs.low     = @low
ATTR Sequence [|  | maxprofit : Int]
SEM Sequence
  | Start   lhs.maxprofit = 0
  | NextDay lhs.maxprofit = (@index - @low) `max` @past.maxprofit

{
dj :: Sequence
dj = Sequence_NextDay (-2)
  (Sequence_NextDay 4
    (Sequence_NextDay 1
       (Sequence_NextDay (-1)
          (Sequence_NextDay 3
             (Sequence_NextDay (-5)
                (Sequence_NextDay 0
                   (Sequence_NextDay 4 Sequence_Start
  )))))))
dj_bestinvest :: Sequence -> Int
dj_bestinvest deltas
  = let (_, _, maxprofit) = sem_Sequence deltas
    in  maxprofit
instance Symbol Char
string2int = foldl (\val dig -> (10 * val + ord dig - ord '0')) 0
pPosInt :: IsParser a Char => a Int
pPosInt = string2int <$> pList1 ('0' <..> '9')
pInt    :: IsParser a Char => a Int
pInt    = (negate <$ (pSym '-') `opt` id) <*> pPosInt
pSpaces :: IsParser a Char => a String
pSpaces = pList (pSym ' ')
pDJ     :: IsParser a Char => a (Int, Int, Int)
pDJ     = pFoldrSep (sem_Sequence_NextDay, sem_Sequence_Start) pSpaces pInt
dj_bestinvest' inp
  = do {(_, _, maxprofit) <- parseIO pDJ inp
       ; putStr (show maxprofit)
       }
main = dj_bestinvest' "-2 4 1 -1 3 -5 0 4"
}
```

3.4 Exercises

3-1 **Bitstring conversion**. In this exercise we concern ourselves with computing the decimal equivalent of a binary number. The exercise has been done before in G & O (Grammatica's en Ontleden [18]), and can be used to refresh your memory concerning parser combinators and abstract syntax trees. It is also an exercise where we make the transition from writing the semantics of a language directly in the grammar, to a way where we use the AG system after computing an abstract syntax tree using parser combinators. This latter part will be done in exercise 4-1.

1. Give a non-leftrecursive BNF grammar for bitstrings. You may accept the empty string as a valid bitstring. Give also the derivation tree for the string `1011`.
2. In this tree, we want to compute the decimal value of the bitstring in the root of the tree. How can you do this bottom-up? Give the intermediate values in the internal nodes. Do you think that one intermediate value per node is enough?
3. Give a parser for binary numbers using parser combinators as constructed from the above BNF grammar (you are not allowed to use the chain, list or fold combinators). The outcome of the parser should be an abstract syntax tree for the data type

   ```
   data Bin = Bin_Empty | Bin_Zero Bin | Bin_One Bin
   ```

   Give the abstract syntax tree for the bitstring `1011` and find out how the computations on the concrete syntax tree carry over to the abstract one.
4. Add the computations as semantic functions to the parser you wrote earlier, so that the necessary values are computed without first constructing the abstract syntax tree.

At this point you should understand what information is necessary where and even how it is moved around in the tree. In exercise 4-1 we shall do the same using the AG system.

Chapter 4

UU AG System User Manual

The material of this chapter is copied from the UU AG System User Manual by Arthur Baars, Doaitse Swierstra and Andres Löh.

The `UUAG` system is an attribute grammar system developed at the University of Utrecht.

4.1 About this document

After the introduction, this document contains a user guide. This guide is divided in two parts, the first consists of an example introducing most language features, the second part covers the language constructs and the `AG` compiler in more detail.

4.2 Reporting bugs

Any bugs (or fixes!) can be reported to the author, Arthur Baars (`arthurb@cs.uu.nl`). Any feedback on:

- what modifications you are interested in
- what modifications you have made yourself

is greatly appreciated too. Besides that, I am also quite interested in any applications, that are created using this system.

4.3 The "Artistic License"

4.3.1 Preamble

The intent of this document is to state the conditions under which a Package may be copied, such that the Copyright Holder maintains some semblance of artistic control over the development of the package, while giving the users of the package the right to use and distribute the Package in a more-or-less customary fashion, plus the right to make reasonable modifications.

4.3.2 Definitions:

- "Package" refers to the collection of files distributed by the Copyright Holder, and derivatives of that collection of files created through textual modification.

- "Standard Version" refers to such a Package if it has not been modified, or has been modified in accordance with the wishes of the Copyright Holder as specified below.
- "Copyright Holder" is whoever is named in the copyright or copyrights for the package.
- "You" is you, if you're thinking about copying or distributing this Package.
- "Reasonable copying fee" is whatever you can justify on the basis of media cost, duplication charges, time of people involved, and so on. (You will not be required to justify it to the Copyright Holder, but only to the computing community at large as a market that must bear the fee.)
- "Freely Available" means that no fee is charged for the item itself, though there may be fees involved in handling the item . It also means that recipients of the item may redistribute it under the same conditions they received it.

1. You may make and give away verbatim copies of the source form of the Standard Version of this Package without restriction, provided that you duplicate all of the original copyright notices and associated disclaimers.
2. You may apply bug fixes, portability fixes and other modifications derived from the Public Domain or from the Copyright Holder. A Package modified in such a way shall still be considered the Standard Version.
3. You may otherwise modify your copy of this Package in any way, provided that you insert a prominent notice in each changed file stating how and when you changed that file, and provided that you do at least ONE of the following:
   (a) place your modifications in the Public Domain or otherwise make them Freely Available, such as by posting said modifications to Usenet or an equivalent medium, or placing the modifications on a major archive site such as uunet.uu.net, or by allowing the Copyright Holder to include your modifications in the Standard Version of the Package.
   (b) use the modified Package only within your corporation or organization.
   (c) rename any non-standard executables so the names do not conflict with standard executables, which must also be provided, and provide a separate manual page for each non-standard executable that clearly documents how it differs from the Standard Version.
   (d) make other distribution arrangements with the Copyright Holder.
4. You may distribute the programs of this Package in object code or executable form, provided that you do at least ONE of the following:
   (a) distribute a Standard Version of the executables and library files together with instructions (in the manual page or equivalent) on where to get the Standard Version.
   (b) accompany the distribution with the machine-readable source or the Package with your modifications.
   (c) give non-standard executables non-standard names, and clearly document the differences in manual pages (or equivalent), together with instructions on where to get the Standard Version.
   (d) make other distribution arrangements with the Copyright Holder.

5. You may charge a reasonable copying fee for any distribution of this Package. You may charge any fee you choose for support of this Package. You may not charge a fee for this Package itself. However, you may distribute this Package in aggregate with other (possibly commercial) programs as part of a larger (possibly commercial) software distribution provided that you do not advertise this Package as a product of your own. You may embed this Package's code within an executable of yours (by linking); this shall be construed as a mere form of aggregation, provided that the complete Standard Version is so embedded.

6. Aggregation of this Package with a commercial distribution is always permitted provided that the use of this Package is embedded; that is, when no overt attempt is made to make this Package's interfaces visible to the end user of the commercial distribution. Such use shall not be construed as a distribution of this Package

7. The name of the Copyright Holder may not be used to endorse or promote products derived from this software without specific prior written permission.

8. THIS PACKAGE IS PROVIDED "AS IS" AND WITHOUT ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, WITHOUT LIMITATION, THE IMPLIED WARRANTIES OF MERCHANTIBILITY AND FITNESS FOR A PARTICULAR PURPOSE.

The End

4.4 Getting Started

4.4.1 Running the AG system

We assume that `uuagc`, `AG` compiler is installed on your system. If you run the compiler without arguments it will show you a short help message, and a list of options.

```
> uuagc
Usage info:
  uuagc options file ...

List of options:
  -m                        generate default module header
          --module[=name]   generate module header, specify module name
  -d      --data            generate data type definitions
  ...
```

In this user manual all the compiler switches and language features are introduced and explained in the examples.

4.4.2 Simple Attribute Grammar

As a first example we take the well known RepMin problem. The input of the program is a binary tree, and it produces a binary tree of the same shape. In the new tree however all values in the leaves are equal to the minimum of the values in the leaves in the original tree.

A grammar is defined as a collection of `DATA` declarations. The types correspond to the nonterminals and the constructors to the productions of the grammar. The grammar of binary trees is defined as follows:

```
DATA Tree
 | Node left:Tree right:Tree
 | Leaf int:Int
```

As in Haskell the names of the types and constructors start with an uppercase letter. The difference with a Haskell data type definition is that the fields of a constructor are associated with a name, and not only by position.

4.4.3 Adding attributes

In this section we define attributes to solve the Repmin problem. We split the computation to be performed into three different aspects:

- computing the minimal value
- making the minimal value available at the leaves
- constructing the final result

For each of the aspects we introduce an attribute and attribute computation rules.

Firstly we introduce a synthesized attribute *minval* representing the minimum value of a *Tree* by an `ATTR` declaration.

```
ATTR Tree [ | | minval:Int]
```

That *minval* is a synthesized attribute follows from the fact that its declaration is located after the second vertical bar. In an `ATTR` declaration there are three places to put attributes declarations.

$$[\ \textit{inherited} \mid \textit{inherited/synthesized} \mid \textit{synthesised}\ ] \tag{4.1}$$

Attributes in the first position are inherited attributes, attributes in the last position are synthesized attributes, and attributes in the middle are inherited as well as synthesized.

Next we specify the computation of the minimum value by providing semantic rules.

```
SEM Tree
  | Leaf lhs.minval = { @int }
  | Node lhs.minval = { min @left.minval @right.minval }
```

To compute the minimum value of a *Leaf* we simply return the value of the *Leaf*. For a *Node* the minimum value is the minimum of *left*'s *minval* and *right*'s *minval*. The right-hand side of a semantic rule is a Haskell expression between braces. The references to attribute and field values are all marked with an '@' symbol. The left-hand side of a semantic rule is a reference to an attribute. In this case the *minval* attribute of *Tree*, which is the left hand side of the productions *Leaf* and *Node*, hence the name *lhs*.

4.4.4 Compiling an attribute grammar

The example code developed thusfar is can be found in `examples/Repmin1.ag`. This simple attribute grammar is compiled into a `Haskell` source file as follows:

```
> uuagc --module --data --semfuns --catas --signatures Repmin1.ag

Repmin1.hs generated
```

Using the functions in the generated `Haskell` program we can compute the minimum of a *Tree* as is shown in the following example:

```
Repmin1> sem_Tree (Node (Node (Leaf 2 )(Leaf 3))(Node (Leaf 1)(Leaf 2)))
1
```

4.4.5 Generated code

In this section we explain the following compiler options a and take a brief look at the code generated by the `UUAG` compiler.

short option	long option	description
`-m`	`--module[=name]`	generate module header, specify module name
`-d`	`--data`	generate data type definitions
`-f`	`--semfuns`	generate semantic functions
`-c`	`--catas`	generate catamorphisms
`-s`	`--signatures`	generate type signatures for semantic functions

The option `--module` tells the `UUAG` compiler to generate a `Haskell` module header. If a name is specified this name is used as the module name. If no name is specified or when the short option (`-m`) is used the module name is the name of the `UUAG` source file, without its extension. Hence the generated code for `RepMin1.ag` code contains the following module header:

```
module Repmin1 where
```

The option `--data` tells the `UUAG` compiler to generate `Haskell` data type definitions corresponding to the `DATA` statements in the attribute grammar. The data type definition generated for `RepMin1.ag` is:

```
data Tree = Leaf Int
          | Node Tree Tree
```

The `SEM` rules are compiled into semantic functions that compute the output attributes from the input attributes. For each nonterminal a type synonym, named `T_`*Type*,is introduced for the type of its semantics. In our example there are no inherited attributes and only a single synthesized attribute, namely *minval* with type *Int*. Hence the type synonym for the nonterminal *Tree* is:

```
type T_Tree = Int
```

The option `semfuns` tells the compiler to generate a semantic function for each constructor. They are named as follows: `sem_`*Nonterminal_Constuctor*. A semantic function takes the semantics of the constructor's children as argument to compute the semantics of the nonterminal. By providing the `--catas` the `UUAG` compiler generates catamorphisms for each data type in the attribute grammar. A catamorphism takes a data type and computes its semantics. This is achieved by and applying the appropriate semantic functions. The generated catamorphisms are named as follows: `sem_`*Type*. The option `--signatures` tells the compiler to emit type signatures for all semantic functions and catamorphisms. For our example these signatures are:

```
sem_Tree_Node :: T_Tree -> T_Tree -> T_Tree

sem_Tree_Leaf :: Int -> T_Tree

sem_Tree :: Tree -> T_Tree
```

The semantics of a child with a type that is not defined using a `DATA` statement is simply its value. Hence the type *Int* for the semantics of the value in a *Leaf*.

The actual code generated for semantics functions and catamorphisms is discussed in section 4.7.

4.4.6 RepMin continued

The attribute grammar developed thusfar computes the minimum of a tree. This computation is done bottom-up using a single attribute *minval*. The global minimum of a tree is the value of *minval* at the root node. To solve the "repmin" problem we need to distribute the global minimum to all the leaves and, then reconstruct the tree with each value replaced by the global minimum.

Distribute global minimum

The global minimum is the minimum value of the root node of the tree. In order to make the global minimum available at all the leaves we need to push the minimum value of the root node down to all the leaves.

Firstly we declare an inherited attribute *gmin* that holds the global minimum.

```
ATTR Tree [ gmin:{Int} | | ]
```

At each *Node* the global minimum is distributed to both children. Their is no rule for the constructor *Leaf* because it does not have children.

```
SEM Tree
  | Node left.gmin  = { @lhs.gmin }
         right.gmin = { @lhs.gmin }
```

The global minimum is passed down from parent nodes to their children. The root node of a *Tree*, however, does not have a parent, so we cannot set its inherited attribute *gmin*. We introduce a data type *Root* that serves as the parent of a *Tree*. It uses the synthesized attribute *minval* of the *Tree* to define the inherited attribute *gmin*.

```
DATA Root | Root tree:Tree

SEM Root
  | Root tree.gmin = { @tree.minval }
```

Construct the result

Now the global minimum is available everywhere in the tree we can construct the final result, that is a tree with the same structure as the original, but with the value stored in each leaf replaced by the smallest integer stored in the entire tree.

First we declare a synthesized attribute *result* for both *Tree* and *Root*.

```
ATTR Tree [ | | result:Tree ]
ATTR Root [ | | result:Tree ]
```

In a *Node* the resulting trees of both children are combined into a new *Node*. For a *Leaf* a new *Leaf* is returned containing the minimum value.

```
SEM Tree
  | Node lhs.result  = { Node @left.result @right.result }
  | Leaf lhs.result  = { Leaf @lhs.gmin }
```

At a *Root* the resulting tree is returned.

```
SEM Root
  | Root lhs.result = { @tree.result }
```

Haskell code blocks

To finish the rep-min example we define a number of `Haskell` functions. These definitions are written between braces and are copied literally into the output of the `AG` System. The following code block defines an instance of *Show* for *Tree*, a sample *Tree* and a *main* function.

```
{
instance Show Tree where
  show tree = case tree of
                Leaf val -> "Leaf " ++ show val
                Node l r -> "Node (" ++ show l ++ ") (" ++ show r ++ ")"

example :: Tree
example =  Node (Leaf 3)(Node (Leaf 6)(Leaf 2))

main :: IO ()
main = do putStrLn "input tree:"
          print example
          putStrLn "result tree:"
          print (sem_Root (Root example))
}
```

Compile and Run

The example code developed thusfar is can be found in `examples/Repmin2.ag`. This attribute grammar is compiled into a `Haskell` source file as follows:

```
> uuagc --module=Main --signatures --data --semfuns --catas Repmin2.ag

Repmin2.hs generated
```

The generated code is a module named *Main* containing the *Tree* datatype, semantic functions, catamorphisms, and some additional `Haskell` definitions. The program can be run using `runhugs` as follows:

```
> runhugs Repmin2.hs
input tree:
Node (Leaf 3) (Node (Leaf 6) (Leaf 2))
result tree:
Node (Leaf 2) (Node (Leaf 2) (Leaf 2))
```

4.5 Language Constructs

This section gives an overview of the `UUAG` language. Lines printed in bold are grammar rules and show what the language construct looks like in general. Subscripts and "..."-notation are used in the syntax rules. For example:

$$\textbf{constructor name}_1\textbf{:type}_1 \ldots \textbf{name}_n\textbf{:type}_n \quad (n \geqslant 0)$$

This means that a constructor has zero or more fields. Valid instantiations are:

```
Leaf val:Int
Bin left:Tree right:Tree
Empty
```

The following sections show the syntax of each construct as a grammar rule, followed by an explanation of its semantics and a number of examples. The `UUAG` language provides many shorthand notations. These abbreviations are explained by example, as including them in the grammar rules would clutter the presentation. A complete reference in `EBNF` of the `UUAG` language can be found in Section 4.8.

4.5.1 DATA declaration

$$
\begin{array}{ll}
\texttt{DATA} & \textbf{nonterminal} \\
& \textbf{|constructor}_1\textbf{field}_{1,1}\textbf{:type}_{1,1}\ldots\textbf{field}_{1,i}\textbf{:type}_{1,i} \\
& \textbf{|}\vdots \\
& \textbf{|constructor}_n\textbf{field}_{n,1}\textbf{:type}_{n,1}\ldots\textbf{field}_{n,j}\textbf{:type}_{n,j} \\
& (i \geqslant 0, j \geqslant 0, n \geqslant 0)
\end{array}
$$

A `DATA` declares a number of productions for a nonterminal. Each production is labelled with a constructor name. In contrast to `Haskell` it is allowed to use the same constructor name for more than one nonterminal. However, the names of all constructors of the same nonterminal must be different. Giving multiple `DATA` declarations for the same nonterminal is allowed, provided that the constructor names in the declarations do not clash. The fields of each production all have a name and a **type**. The type can be a nonterminal or a `Haskell` type. All fields of the same constructor must have different names.

Valid `DATA` declarations:

```
DATA Tree | Bin left:Tree right:Tree
          | Leaf value:Int

DATA Decl | Fun name:String args:{[String]} body:Expr
```

Several abbreviations exist for `DATA` declarations. Fields with the same type can be declared by listing their names separated by commas. Also the field name can be left out, in which case the name is defaulted to the type name with the first letter converted to lowercase. It is only allowed to leave out the field name if the type is an uppercase type identifier. You also need to make sure that the default name does not clash with the name of another field. The following example show correct abbreviations:

```
DATA Tree | Bin left,right:Tree -- 'left' & 'right' have type 'Tree'
          | Leaf Int            -- field name is 'int'
```

The following `DATA` statement is wrong:

```
DATA Tree | Bin Tree Tree    -- duplicate field name
          | Leaf {(Int,Int)} -- type is not a single type identifier
```

4.5.2 ATTR declaration

$$
\begin{array}{ll}
\texttt{ATTR} & \textbf{nonterminal}_1\ldots\textbf{nonterminal}_n \\
& \texttt{[}\ \textbf{attr}_1\textbf{:type}_1\ldots\textbf{attr}_i\textbf{:type}_i \\
& \textbf{|}\ \textbf{attr}_{(i+1)}\textbf{:type}_{(i+1)}\ldots\textbf{attr}_j\textbf{:type}_j \\
& \textbf{|}\ \textbf{attr}_{(j+1)}\textbf{:type}_{(j+1)}\ldots\textbf{attr}_k\textbf{:type}_k \\
& \texttt{]} \\
& (n \geqslant 1, 0 \leqslant i \leqslant j \leqslant k)
\end{array}
$$

An `ATTR` declaration declares attributes for one or more nonterminals. Each attribute has a name and a type. The position of an attribute in the declaration list (left of the bars, between the bars, or right of the bars) determines whether it is inherited, chained, or synthesized, respectivly. A chained attribute is just an abbreviation for an attribute that is both inherited and synthesized. The names of all inherited attributes declared by `ATTR` statements must be different. The same holds for synthesized attributes.

Valid `ATTR` declarations are:

```
ATTR Tree [ depth:Int  | minimum:Int | out:{[Bool]} ]
ATTR Tree [ count:Int  | | count:Int ]
ATTR Decl [ environment : {[(String,Type)]} | | ]
ATTR Decl [ | | code:Instructions ]
```

For attribute declarations the same abbreviations are permitted as for field in a `DATA` declaration. The name of an attribute can be left out, and attributes with the same type can be grouped. For example:

```
ATTR Tree [ | | min,max:Int ] -- 'min' and 'max' both have type 'Int'
ATTR Decl [ Environment | | ] -- attribute name is 'environment'
```

The following abbreviations are wrong:

```
ATTR Tree [ | | Int Int ]           -- duplicate attribute names
ATTR Decl [ {[(String,Type)]} | | ] -- complex type without name
```

A `USE` clause can be added to the declaration of a synthesized or chained attribute, to trigger a special kind of copy rule(see Section 4.6.3). The first expression must be an operator, and the second expression is a default value for the attribute.

attr `USE` **$\mathbf{expr_1}$ $\mathbf{expr_2}$: type**

For example:

```
DATA Tree
   | Bin left,right:Tree
   | Leaf value:Int
ATTR Tree [ | | value USE {+} {0} : Int ] -- compute sum of values
```

An attribute can be declared to be of type `SELF`. The type `SELF` is a placeholder for the type of the nonterminal for which the attribute is declared. For example:

```
ATTR Tree Expr [ | | copy:SELF ]
```

The `ATTR` statement above declares an attribute *copy* of type *Tree* for nonterminal *Tree*, and an attribute *copy* of type *Expr* for nonterminal *Expr*. Declaring a synthesized attribute of type `SELF` triggers a special copy-rule, that constructs a copy of the tree. Section 4.6.4 explains this type of copy-rule.

Attribute declarations can also be given in `DATA` or `SEM` statements after the name of the nonterminal. For example:

```
DATA Tree | Bin left,right:Tree
          | Leaf Int
ATTR Tree [ | | min:Int ]
```

can be combined into:

```
DATA Tree [ | | min:Int ]
   | Bin left,right:Tree
   | Leaf Int
```

4.5.3 SEM

In a `SEM` construct one can specify semantic rules for attributes. For each production the synthesized attributes associated with its corresponding nonterminal and the inherited attributes of its

children must be defined. If there is a rule for a certain attribute is missing, the system tries to derive a so called copy-rule. The `SEM` construct has the following form:

`SEM`	**nonterminal**	
	|constructor$_1$	**fieldref$_1$. attribute$_1$=expression$_1$**
		$\vdots$
	|constructor$_n$	**fieldref$_n$. attribute$_n$=expression$_n$**
	$(n \geqslant 0)$	

Semantic rules are organised per production. Semantic rules for the same production can be spread between multiple `SEM` statements. This has the same meaning as they were defined in a single `SEM` statement. A **fieldref** is `lhs`, or `loc`, or a **field** name. To refer to a synthesized attribute of the nonterminal associated with a production the special **fieldref** `lhs` is used together with the name of the attribute. To refer to an inherited attribute of a child of a production the **field** name of the child is used together with the attribute's name. The special **fieldref** `loc` is used to define a variable that is local to the production. It is in the scope of all semantic rules for the production.

The expressions in semantic rules are code blocks, i.e. `Haskell` expressions enclosed by { and }, see Section 4.5.6. They may contains references to values of attributes and fields. These references are all prefixed with an `@`-sign to distinguish them from `Haskell` identifiers. To refer to the value of a field one uses the name of the field. References to attributes are similar to the ones on the left-hand side of a semantic rule (**fieldref . attribute**). The difference is that they now refer to the synthesized attributes of the children and the inherited attributes of the nonterminal associated with the production. Local variables can be referenced using their name, optionally prefixed with the special **fieldref** `loc`.

Valid definitions:

```
ATTR Tree [ gmin:Int | | min:Int result:Tree ]
SEM Tree
   | Bin  left.gmin  = { @lhs.gmin }
     -- "left.gmin" refers to the inherited attribute "gmin"
     -- of the child "left"
   | Bin  right.gmin = { @lhs.gmin }
     -- "@lhs.gmin" refers to the inherited attribute "gmin"
     -- of nonterminal "Tree"
   | Bin  loc.min    = { min @left.min @right.min }
     -- "min" is a new local variable of the constructor "Bin"

SEM Tree
  | Bin  lhs.result = { Bin @left.result @right.result }
    -- "@left.result" refers to the synthesized attribute "result"
    -- of child "left"
  | Bin  lhs.min    = { @min }
    -- "@min" refers to the local variable "min"
  | Leaf lhs.result = { Leaf @lhs.gmin }
    -- "@lhs.gmin" refers to the inherited attribute "gmin"
    -- of nonterminal "Tree"
  | Leaf lhs.min    = { @int }
    -- "@int" refers to the value of field "int" of "Leaf"
```

For the `SEM` construct there exist a number of abbreviations. As for `DATA` statements one can write attribute declarations after the name of the nonterminal. Furthermore semantic rules for the same production can be grouped, mentioning the name of the production only once. For example:

```
SEM Tree
   | Bin  left.gmin  = { @lhs.gmin }
          right.gmin = { @lhs.gmin }
          loc.min    = { min @left.min @right.min }
```

In a similar way semantic rules for the same **fieldref** can be grouped. For example:

```
SEM Tree
  | Bin  lhs.result = { Bin @left.result @right.result }
           .min    = { @min }
```

When the same semantic rule is defined for two productions of the same nonterminal they can be combined by writing the names of both productions in front of the rule. For example:

```
SEM Tree
  | Node1 lhs.value = { @left.value + @right.value }
  | Node2 lhs.value = { @left.value + @right.value }
```

can be abbreviated as follows:

```
SEM Tree
  | Node1 Node2 lhs.value = { @left.value + @right.value }
```

Finally the curly braces around expressions may be left out. The layout of the code is then used to determine the end of the expression as follows. The column of the first non-whitespace symbol after the =-sign is the reference column. All subsequent lines that are indented the same or further to the right are considered to be part of the expression. The expression ends when a line is indented less than the reference column. An advantage of using layout is that problems with unbalanced braces, as described in Section 4.5.6 are avoided.

4.5.4 TYPE

The `TYPE` construct is convenient notation for defining list based types. It has the following form:

`TYPE` **nonterminal** `=` `[` **type** `]`

A `TYPE` construct is equivalent to:

```
DATA  nonterminal
      | Cons hd:type tl:nonterminal
      | Nil
```

Apart from a convenient notation the `TYPE` construct has effect on the code generated. Instead of generating data constructors `Cons` and `Nil` `Haskell`'s list constructors `:`, and `[]` are used.

Examples of `TYPE` constructs:

```
TYPE IntList = [ Int ]
TYPE Trees   = [ Tree ]
```

4.5.5 INCLUDE

Other `UUAG` files can be included using the following construct:

`INCLUDE` **string**

The **string** is a file name, between double quotes. The suffix of the file (`.ag` , or `.lag`) should not be omitted. The file should contain valid `UUAG` statements. These statements are inlined in the place of the `INCLUDE` statement.

4.5.6 Code Block

A code block is a piece of `Haskell` code enclosed by curly braces.

{ **haskellcode** }

There exist three kinds of code blocks: top-level, type, and expression code blocks. A top-level code block contains `Haskell` declarations, such as `import` declarations, and function and type definitions. A name can be writen before a top-level code block. The code blocks are sorted by their names, and appended to the code generated by the `UUAG` system. A special name `imports` is used to mark code blocks containing `import` declarations. These are copied to the start of the generated code, as `Haskell` only allows `import` declarations at the beginning of a file.

An example of two code blocks, an import declaration and a function definition:

```
imports
{
import List
}

quicksort
{
-- simple implementation of quicksort:
qsort :: Ord a => [a] -> [a]
qsort [] = []
qsort (x:xs) = let (l,r) = partition (<=x) xs
               in qsort l ++ [x] ++ qsort r
}
```

A type code block contains a `Haskell` type and may be used as **type**s in `DATA`, `TYPE`, and `ATTR` declarations. Examples:

```
DATA Module
   | Module name:{Maybe String} body:Declarations

TYPE Points = [ {(Int,Int)} ]

ATTR [ env:{[(String,Int)]} | | ]
```

Finally expression code blocks contain a `Haskell` expression and occur as the right-hand side of attribute definitions in `SEM` statements. Apart from normal `Haskell` code they may contain references to attributes. These references are prefixed with an `@`-symbol, to distinguish them from ordinary `Haskell` identifiers. Examples:

```
SEM Tree [ | | min:{Int} ]
  | Node lhs.min = { min @left.min @right.min } -- an expression code block
```

The contents of a block is the plain text between an open and a close brace. The text in a code block is not interpreted by the `UUAG` system.

any	::=	[“\0″..“\255″]	(*any character*)
codeblock	::=	“{″**codeblockcontent**∗“}″	
codeblockcontent	::=	**any**	*except* {, *and* }
		| **codeblock**	

Curly braces occurring inside the `Haskell` code must be balanced. This includes curly braces in comments, and in string and character literals.

An example of a code block containing a nested pair of braces:

```
{
f a b c = let { d = b*b - 4*a*c
              ; result1 = (-b + sqrt d) / 2*a
              ; result2 = (-b - sqrt d) / 2*a
              ; result  | d >  0 = [result1, result2]
                        | d == 0 = [result1]
                        | d < 0  = []
              }
          in result
}
```

All curly braces `Haskell` constructs, such as `do`, `let` must be matched. However, curly braces in string, or character literals may cause problems. The balancing rule forbids code blocks such as:

```
{
openbrace = "{"
}
```

This problem can be fixed by inserting a matching brace in comments. In the following code the curly braces are balanced:

```
{
openbrace = "{"
-- }, just to balance braces
}
```

4.5.7 Comments

One-line comments start with two dashes (`--`) and end at the end of the line. Multi-line comments start with `{-` and end with `-}`. As in `Haskell` comments can be nested.

```
{-
Definition of a datatype for binary trees
-}
DATA Tree
   | Leaf val:Int
   | Node left:Tree right:Tree -- a node has two subtrees
```

4.5.8 Names

Names start with a letter followed by a (possibly empty) sequence of letters, digits, and the symbols `_` and `'`. A name for a **nonterminal** or **constructor** must start with an upper-case letter. A name of a **field** or **attribute** must start with a lower-case letter. The following words are reserved and cannot be used as names: `DATA`, `EXT`, `ATTR`, `SEMTYPE`, `USE`, `loc`, `lhs`, and `INCLUDE`.

Valid names:

```
-- nonterminals or constructors:
Node
Expression
Tree_Node
-- field names or attributes:
left
long_name
field2
```

4.5.9 Strings

A **string** in AG is sequence of characters enclosed by double quotes ("). The structure of strings is similar to Haskell strings. The escape character is a backslash (\). Below a table with the most common escape sequences:

\'	single quote (')
\"	double quote (")
\n	newline
\t	tab
nnn	character with ascii-code *nnn*

For a more detailed description of string and escape sequences see the Haskell Report[27]. Examples of valid strings:

```
"hello world"
"line 1\nline 2"
"hello\32world"
```

4.6 Copy Rule

When a definition for an attribute is missing, the UUAG can often derive a rule for it. These automatic rules, also known as copy rules, are based on name equality of attributes. They save a lot of otherwise trivial typing, thus making your programs easier to read by just leaving the essential parts in the code. If in the list of rules for a constructor a rule for an attribute $attr_1$ is missing then UUAG system tries to derive a rule for this attribute. This is done by looking for an attribute $attr_2$ with the same name as $attr_1$ in the sets of synthesised attributes of the children of the constructor and in the set of inherited attributes of the nonterminal it belongs to. If such an attribute $attr_2$ is found then the value of $attr_1$ is set to the value of $attr_2$. This section firstly shows two examples and then defines a generalisation that captures both(and others). There are also two special copy rules, the USE, and SELF rules, which are explained at the end of this section.

4.6.1 Examples

Very often one needs to pass a value from a node to all its children. Consider for example the following code, in which a inherited attribute *gmin* is declared.

```
DATA Tree | Bin left,right:Tree
          | Leaf val:Int

ATTR Tree [ gmin:Int | | ]
```

In this example rules for the syntesized attribute *gmin* of children of the constructor *Bin* are missing. This is however no problem. The nonterminal *Tree* has an inherited attribute with the same name and the UUAG system automatically inserts the following rules:

```
SEM Tree
  | Bin left.gmin  = @lhs.gmin
        right.gmin = @lhs.gmin
```

This kind of copy-rule is very convenient for copying an inherited attribute to all nodes in a top-down fashion.

Another kind of copy-rule is a co-called chain-rule. For a chain rule an attribute that is both inherited as well as synthesized is chained from left to right through all children of a constructor. Consider for example the following code that numbers all leaves in a *Tree* from left to right.

```
ATTR Tree [ | label:Int | ]

SEM Tree
  | Leaf lhs.label = @lhs.label+1
```

Because the attribute *label* is declared inherited as well as synthesized the `UUAG` system derives the following rules for the constructor *Bin*:

```
SEM Tree
  | Bin left.label  = @lhs.label
        right.label = @left.label
        lhs.label   = @right.label
```

4.6.2 Generalised copy rule

The `UUAG` system implements a more general copy rule of which the examples above are instances. If a rule is missing for an inherited attribute n of a child c of constructor con, the `UUAG` system searches for an attribute with the same name(n). The `UUAG` system searches for a suitable candidate in the following lists:

1. local attributes
2. synthesized attributes of children on the left of c
3. inherited attributes
4. fields

The search takes place in the order defined above, and the first occurrence of n is copied. Thus local attributes have preference over others. When there are multiple occurrences of n in the list of synthesized attributes of the children the rightmost is taken.

When a rule for a synthesized attribute is missing the search for a candidate with the same name takes place in a similar fashion. In the second step all children are searched, again taking the rightmost candidate if more than one is found.

4.6.3 `USE` rules

A `USE` rule can be derived for a synthesized attribute whose declaration includes a `USE` clause. A `USE` clause consists of two expressions; the first is an operator, and the second is a default value. Suppose s is a synthesized attribute of n, that is declared with a `USE` clause. If for a constructor c of n a definition of s is missing, a rule is derived as follows. Collect all synthesized attributes of constructor c's children with the same name as s. If this collection is empty the default value declared in the `USE` clause is taken. If this collection contains only a single attribute, then the value of this attribute is copied. Otherwise the values of the attributes are combined using the operator and the result is used to define s.

For example:

```
DATA Tree
   | Bin left,right:Tree
   | Single val:Int
   | Empty

ATTR Tree [ || sum USE {+} {0} : Int]
SEM Tree
  | Single lhs.sum = @val
```

The UUAG system derives the following rules:

```
SEM Tree
  | Bin   lhs.sum = @left.sum + @right
  | Empty lhs.sum = 0
```

4.6.4 SELF rules

The type SELF in an attribute declaration is equivalent to the type of the nonterminal to which the attribute belongs. A synthesized SELF attribute can for example be used if one wants a local copy of a tree, or wants to transform it. The SELF attribute then holds the transformed version of the tree. A SELF attribute usually holds a copy of the tree, except for a few places where a transformation is done. The semantic rules required for constructing a copy of a tree call for each production the corresponding constructor function on the copies of the children. The UUAG system implements a special copy rule to avoid writing these trivial rules. For each production of a nonterminal with a synthesized SELF attribute n, the UUAG system generates a local attribute containing the application of the corresponding constructor to the SELF attributes of the children with the same name as n. The value of the synthesized attribute is set to this local attribute.

For example for:

```
DATA Tree
   | Bin left,right:Tree
   | Leaf val:Int

ATTR Tree [ | | copy : SELF ]
```

the following semantic rules are generated:

```
SEM Tree
  | Bin  loc.copy = Bin @left.copy @right.copy
         lhs.copy = @copy
  | Leaf loc.copy = Leaf @val
         lhs.copy = @copy
```

The default definitions for the local and sythesized SELF attributes can be overriden by the programmer.

The following program is a complete attribute grammer for the rep-min problem using as many copy rules as possible. For constructing the transformed a SELF attribute *result* is used. Note that only for the production *Leaf* an explicit definition of this attribute is given. The definition for *Bin* is provided by an automatic rule.

```
DATA Tree
   | Bin left,right:Tree
   | Leaf val:Int

DATA Root
   | Root Tree

ATTR Tree [ gmin:Int | | lmin USE {‘min‘} {0}:Int ]
ATTR Root Tree [ | | result:SELF ]

SEM Tree
  | Leaf lhs.lmin   = @val
            .result = Leaf @lhs.gmin
SEM Root
  | Root tree.gmin = @tree.lmin
```

4.7 Code Generation

4.7.1 Module header

If the option `-m` or `--module=[name]` is provided to the `UUAG` compiler then a module header will be generated. If a name is provided to the `--module` flag then this name is used as module name, otherwise the module name will be the filename without the suffix `.ag` or `.lag`.

4.7.2 Data types

When the flag `--data` or `-d` is passed to the `UUAG` compiler, then a data type definition is generated for each nonterminal introduced in a `DATA` declaration and a type synonym is generated for each nonterminal introduced in a `TYPE` declaration. The `UUAG` system allows different nonterminals to have constuctors with the same names. For `Haskell` data types this is not allowed. To prevent clashes between constuctors of different data types the flag `--rename` or `-r` can be specified. All constructors will then be prefixed with their corresponding nonterminal(and an underscore).

For example for this fragment of `UUAG` code:

```
DATA Expr
   | Var name:String
   | Apply fun:Expr arg:Expr
   | Tuple elems:Exprs
   | ...

TYPE Exprs = [Expr]

DATA Type
   | Var name:String
   | Apply fun:Type arg:Type
   | ...
```

the following `Haskell` code is generated when the flags `--data` and `--rename` are switched on:

```
data Expr
   = Expr_Var String
   | Expr_Apply Expr Expr
   | ...

type Exprs = [Expr]

data Type
   = Type_Var String
   | Type_Apply Type Type
   | ...
```

If the `--rename` flag is not provided it is the responsibility of the programmer to make sure that are constructors are uniquely named.

4.7.3 Semantic functions

The semantic domain of a nonterminal is a mapping from its inherited to its synthesized attributes. When the flag `--semfuns` or `-f` is switched on, the `UUAG` compiler generates for each nonterminal a type synonym representing its semantic domain, and for each constructor a semantic function. A semantic function takes the semantics of its children as arguments and returns the semantics of the corresponding nonterminal. A semantic function is named as follows:

prefix_nonterminal_constructor.
The default prefix is `sem`, another prefix can be supplied with the `--prefix=name` flag.
Consider the following code fragment:

```
DATA Tree
   | Bin left,right:Tree
   | Leaf val:Int

ATTR Tree [ lmin:Int gmin:Int | | lmin:Int result:Tree ]

SEM Tree
  | Bin  lhs.result = Bin @left.result @right.result
  | Leaf lhs.lmin   = min @lhs.lmin @val
         lhs.result = Leaf @lhs.gmin
```

The semantic domain of the nonterminal **Tree** is defined as follows:

```
type T_Tree = Int -> Int -> (Int,Tree)
```

The inherited attributes are arguments and the synthesized attributes are packed together in a tuple as result. The `UUAG` system lexicographically sorts the attributes, hence the first `Int` stands for the inherited attribute *gmin*, and the second for the inherited attribute *lmin*. If the flag `--newtypes` is switched on, a `newtype` declaration is generated for the semantic domain instead of a `type` synonym.
The types of the generated semantic functions for the constructors *Bin*, and the *Leaf* are the following:

```
sem_Tree_Bin  :: T_Tree -> T_Tree -> T_Tree
sem_Tree_Leaf :: Int              -> T_Tree
```

Note that the semantics of a child that has a `Haskell` type is simply the value of that child. When the flag `--signatures` or `-s` is switched on then the type signatures of the semantic functions are actually emmited in the generated code.

4.7.4 Catamorphisms

When the flag `--catas` or `-c` is supplied, the the `UUAG` compiler generates catamorphisms for every nonterminal. A catamorphism is a function that takes a (syntax) tree as argument and computes the semantics of that tree. The catamorphism for a nonterminal **nt** is named as follows:
prefix_nonterminal.
As for semantic functions the prefix is `sem` by default, and can be changed with the `--prefix=name` flag. For example the type of the catamorphism for the nonterminal **Tree** is:

```
sem_Tree :: Tree -> T_Tree
```

When the flag `--signatures` or `-s` is switched on then the type signatures of the catamorphisms are actually emmited in the generated code.

4.7.5 Wrappers

The result of a semantic function or a catamorphism is a function from inherited to synthesized attributes. To be able to use such a result, a programmer needs to know the order of all the attributes. Wrapper functions for the semantic domains can be generated to provide access to the attributes by their names. When the flag `--wrappers` or `-w` is switched on the following is generated for each semantic domain:

- a record type with named fields for the inherited attributes
- a record type with named fields for the synthesized attributes
- a wrapper function that transforms a semantic domain in a function from a record of inherited attributes to a record of synthesized attributes

The two record types for a nonterminal *nt* are called *nt*_Inh, and *nt*_Syn, for the inherited and synthesized attributes, respectively. The labels of the records are the names of the attributes suffixed with the name of the record type. The generated wrapper function is named wrap_*nt*.

For the nonterminal **Tree** in the example above the following record types are generated:

```
data Tree_Inh = Tree_Inh{ lmin_Tree_Inh :: Int
                        , gmin_Tree_Inh :: Int
                        }
data Tree_Syn = Tree_Syn{ lmin_Tree_Syn   :: Int
                        , result_Tree_Syn :: Tree
                        }
```

The generated wrapper function has the following type:

```
wrap_Tree :: T_Tree -> Tree_Inh -> Tree_Syn
```

Using the generated wrapper code the function *repmin* can be defined as follows:

```
repmin :: Tree -> Tree
repmin tree = let synthesized = wrap_Tree (sem_Tree tree) inherited
                  inherited   =
                        Tree_Inh
                        { lmin_Tree_Inh = infty
                        , gmin_Tree_Inh = lmin_Tree_Syn synthesized
                        }
                  infty       = 1000
              in result_Tree_Syn synthesized
```

4.8 Grammar

Normal UUAG system source files have .ag as suffix. The UUAG system also supports literate programming. Literate UUAG files have .lag as suffix. In literate mode all text in a file is considered to be comments, except for those blocks enclosed between: \begin{Code}, and \end{Code}. The begin and end commands should be placed at the beginning of a line.

The remainder of this section presents the grammar of the UUAG system as EBNF production rules. Parenthesis are used for grouping, nonterminals are printed **boldface**, and terminal symbols are printed between "quotes". A rule of form $X*$ means zero or more occurrences of X, $X+$ means one or more occurrences of X, and $X?$ is an optional occurrence of X. In the lexical syntax character ranges are written between square brackets. For example ["A"."Z"] represents the range of uppercase letters.

4.8.1 Lexical Syntax

keywords = { "DATA", "EXT", "ATTR", "SEM", "TYPE"
, "USE", "loc", "lhs", "INCLUDE" }

uppercase ::= ["A" .. "Z"]

lowercase ::= ["a" .. "z"]

any	::=	["\0" .. "\255"] (any character)
conid	::=	**uppercase identletter*** except **keywords**
varid	::=	**lowercase identletter*** except **keywords**
identletter	::=	**uppercase**
	|	**lowercase**
	|	"'"
	|	"_"
string	::=	"`"` **stringcontents** "`"`
codeblock	::=	"{" **codeblockcontent*** "}"
codeblockcontent	::=	**any** except "{", and "}"
	|	**codeblock**
layoutcodeblock	::=	**layoutcontent***
layoutcontent	::=	**any** (except letters that are less indented than reference column)

4.8.2 Context-free Grammar

ag	::=	**elem***
elem	::=	"`DATA`" **conid attrDecls**? **dataAlt***
	|	"`ATTR`" **conid+ attrDecls**
	|	"`TYPE`" **conid** "`=`" "`[`" **type*** "`]`"
	|	"`SEM`" **conid attrDecls**? **semAlt***
	|	**varid**? **codeblock**
	|	"`INCLUDE`" **string**
attrDecls	::=	"`[`" **inhAttrDecl*** "`|`" **synAttrDecl*** "`|`" **synAttrDecl*** "`]`"
type	::=	**conid**
	|	**codeBlock**
inhAttrDecl	::=	**varids** "`:`" **type**
varids	::=	**varid** ("`,`" **varid**)*
synAttrDecl	::=	**varids** ("`USE`" **codeBlock codeBlock**)? "`:`" **type**
dataAlt	::=	"`|`" **conid field***
field	::=	**varids** "`:`" **type**
	|	**conid**
semAlt	::=	"`|`" **conid+ semDef***
semDef	::=	(**varid** | "`lhs`") **attrDef+**
	|	"`loc`" **locDef+**
attrDef	::=	"`.`" **varid assign expr**
locDef	::=	"`.`" **pattern assign expr**
expr	::=	**codeBlock**
	|	**layoutCodeBlock**
assign	::=	"`=`"
	|	"`:=`"
pattern	::=	**conid pattern$_1$***
	|	**pattern$_1$**
pattern$_1$	::=	**varid** ("`@`" **pattern$_1$**)?
	|	"`(`" **patterns**? "`)`"
	|	"`_`"

patterns ::= **pattern** (“,” **pattern**)*

4.9 Compiler flags

short option	long option	description
`-m`	`--module[=name]`	generate module header, specify module name
`-d`	`--data`	generate data type definitions
`-f`	`--semfuns`	generate semantic functions
`-c`	`--catas`	generate catamorphisms
`-s`	`--signatures`	generate type signatures for semantic functions
	`--newtypes`	use newtypes instead of type synonyms
`-p`	`--pretty`	generate pretty printed list of attributes
`-w`	`--wrappers`	generate wappers for semantic domains
`-r`	`--rename`	prefix data constructors with the name of corresponding type
	`--nest`	use nested pairs, instead of large tuples
`-o file`	`--output=file`	specify output file
`-v`	`--verbose`	verbose error message format
`-h,-?`	`--help`	get usage information
`-a`	`--all`	do everything (-dcfsprm)
	`--prefix=prefix`	set prefix for semantic functions, default is `sem_`
	`--self`	generate self attribute for all nonterminals
	`--cycle`	check for cyclic attribute definitions
	`--version`	get version information

4.10 Exercises

4-1 **Bitstring conversion revisited**. This exercise is the continuation of exercise 3-1. Read that exercise before proceeding with this one.

1. Give the `DATA` definitions so that in the corresponding generated `Haskell` code the abstract syntax trees as given in part 3 of exercise 3-1 can be handled. Explain the differences between the two.
2. Now add the definitions of the attributes to be computed (`ATTR` section).
3. Add semantic functions that compute the values of the attributes just defined (`SEM` section).
4. Which rules could have been omitted because of the copy/chain rules in the AG system, or which rules should be added if you did not have these copy-chain rules?
5. Do the AG specific part (starting at part 2) of this exercise for the abstract datatype

```
DATA Bin
    | Empty
    | Zero
    | One
    | Split left, right: Bin
```

You may have to reconsider the attributes to be computed. (You should not write a parser to yield abstract syntax trees of this kind.) Verify that the AG system works for the abstract syntax tree corresponding to

(Split (Split One Zero) (Split Empty (Split One One)))

4-2 **Table formatting**. This exercise is similar to the *repmin problem*, so you might look at that problem and its solution again if you are in need of inspiration.

The task is to program a so-called *table-formatter*. The following example demonstrates what the input language is supposed to look like:

```
<table>
<row> <elem> aap noot </elem> <elem> langestring </elem> </row>
<row> <elem> langestring </elem> <elem> kort </elem> </row>
</table>
```

We now want to align the table. Each column should have its own fixed width. Thus, for our example, we would get something like this:

aap noot	langestring
langestring	kort

We will approach this task in steps. An abstract grammar that is suitable for the problem is:

```
data Table = Table Rows
data Rows  = NoRows
           | ConsRows Row Rows
data Row   = Row Elems
data Elems = NoElems
           | ConsElems Elem Elems
data Elem  = Elem Lines
data Lines = NoLines
           | ConsLines Line Lines
type Line  = String
```

Now proceed according to the following plan:

1. Why have we presented a more complex abstract grammar than necessary by introducing "too many" nonterminals?
2. Add an attribute that computes the minimal height necessary for each `Elem`.
3. Add an attribute that computes the minimal height necessary for each `Row`.
4. Add an attribute that computes the list of minimal widths necessary (the list should have one entry per column) for each `Row`. (Compare this computation with the calculation of the minimum in the *repmin* problem.)
5. Add an attribute that merges these lists of minimal widths into one list of minimal widths for the whole table.
6. Now you can compute each *Elem*'s width and height. Make sure that this information about its width and height is available at each *Elem*.
7. Compute now, in a new attribute, for each `Elem` the *formatted table entry*, i.e. a list of lines. (Compare this with the distribution of the minimum in the *repmin* problem.)
8. Merge these into a list of lines for each row.
9. Merge these into a list of lines for the whole table.
10. Extend the syntax of the language so that in the place of each `Elem` there can be a (nested) `Table`. Notice how simple this is!
11. Answer the very first question in this list once more.
12. Check whether you can simplify your solution because some of the rules you have given could be automatically generated by the so-called *copy-rules*.

4-3 **Attribute grammars (from Exam 20010104)**.

1. Design a grammar for expressions with operators `+`, `-`, `*`, and `/`. The usual precedence and associativity properties of these operators should be reflected in the grammar. The operands may only be integers.
2. Write code that defines a synthesised attribute `val` in which the result value of the expression is computed.
3. Assume that we have a machine with an unlimited number of registers R_i, and with instructions of the form *Op* R_i R_j R_k, where *Op* represents one of *Add*, *Substract*, *Mul*, and *Divide*. Furthermore *Add* R_i R_j R_k should mean $R_i := R_j + R_k$ etc. Define now attributes such that the root of the expression has an attribute `code` which consists of a sequence of instructions that, if executed, would compute the value of the expression in register R_0. Just to make sure: it is not the intention that you compute the value of the expressions yourself (such as in the previous part of the exercise) and use that; only register instructions should be used for computation.
4. Can you adapt your previous solution so that it makes use of a minimal number of different registers? If you think that this is already the case for your solution to the previous part of the exercise, then please explain why your solution is optimal in this sense.

4-4 **Gebruik van het AG systeem (from Exam 20010504)**. In this exercise we will transform a list of strings into a nice listing where the way the strings are arranged is affected by the total space available. The listing consists of columns. All columns have the same width. The number of lines that is needed by the listing is minimal with repect to the condition that the width of the listing is less than the given maximum width and that the columns are wide enough for the longest word in the input list. An exception is the case where the length of the longest word exceeds the given width. Then there should be one column which is just as wide as needed to place all the words in it.

We use the list `["aap", "noot", "mies", "foo", "bar", "p", "q"]` as an example. For a given maximum width of 25 characters, this list is arranged as follows:

```
|aap |noot|mies|foo |
|bar |p   |q   |    |
```

For a maximum width of 15 we get:

```
|aap |noot|
|mies|foo |
|bar |p   |
|q   |    |
```

And for a maximum width of 5 the result is:

```
|aap |
|noot|
|mies|
|foo |
|bar |
|p   |
|q   |
```

Use the following incomplete solution as a starting point:

```
{
makeAbstrSynTree l = sem_Root_Root (foldr sem_Words_Cons sem_Words_Nil l)

test1list = ["aap", "noot", "mies", "foo", "bar", "p", "q"]
```

```
test1tree = makeAbstrSynTree test1list

main = putStr test1tree

spaces n     = replicate n ' '
}

DATA Root
   | Root    words: Words

DATA Words
   | Nil
   | Cons    word: String  words: Words

SEM Root [ || txt: String ]
  | Root  lhs.txt   = @words.txt

SEM Words [ || txt: String ]
  | Nil   lhs.txt = ""
  | Cons  lhs.txt = "|" ++ @word ++
                    "|\n" ++ @words.txt
```

This solution generates the following output (if you call `main`):

```
|aap|
|noot|
|mies|
|foo|
|bar|
|p|
|q|
```

Your task is to complete the AG code so that the problem described above is solved by the program. For this, the AG system is to be used as much as possible. Haskell may be used for supporting code where neccesary. Moreover, you should not place the complete calculation in `Root`. This means, for instance, that the production of the piece of text for a single word takes place in the `Cons` alternative of `SEM Words` (as it is the case in the given starting version).

Complete the AG code so that:

1. the vertical bars end up aligned. (Hint: for this you first have to compute the breadth of the columns.)
2. the form of the listing is influenced by the given total width. The width can be passed to the program that is generated by the AG as follows:

   ```
   main = putStr (test1tree 15)
   ...
   SEM Root [ totalwidth: Int || txt: String ]
   ...
   ```

3. if there are less words than there are cells in the table, then empty cell are filled with spaces. This is – for instance – the case in the example above where 7 words are arranged in a 2×4 and in a 4×2 table.

Chapter 5

A Barebones Compiler

This chapter contains the simplest version of the SL compiler.

5.1 Attribute grammar for the barebones SL compiler (SLbb)

The 'not doing much useful' version of the SL compiler, named *SLbb*.

5.1.1 Main

The compiler is invoked by executing *main* which asks for a filename. The characters of the file are passed through the scanner. The function *parseIO* uses a (here, *pRoot* ()) parser to produce a result and errormessages.

```
import SLParser
import UU.Pretty
import UU.Parsing

slVersion
  = "SL Interpreter"
slDate
  = "20031106"
slIntroducedFeatures
  = "Introduction to AG & Silly SL Interpreter"
slSignOn =
  " ____    _" ++ "\n" ++
  "/ ___| | |" ++ "\tSL (Simple Language) Compiler.\n" ++
  "\\___ \\" ++ " | | " ++ "\tVersion " ++ slVersion ++ "/" ++ slDate ++ "\n" ++
  " ___) || |__  " ++ "\tIntroducedFeatures: " ++ "\n" ++
  "|____/ |_____| " ++ "\t\t" ++ slIntroducedFeatures ++ "\n"

compile filename = do
  tokens ← slScan filename
  (exprpp, res) ← parseIO (pRoot ()) tokens
  putStrLn ("The expression:\n" ++ (disp exprpp 100 ""))
  putStrLn "Evaluates to:"
  putStrLn (show res)
  putStrLn "-------------------------------------------------------"

main = do
  putStrLn slSignOn
  putStr "Enter filename: "
```

```
filename ← getLine
tokens ← slScan filename
compile filename
```

5.1.2 Parsing

```
module SLParser where
import UU.Parsing
import UU.Scanner
import SLAttributes
import SLTypes
import Char
```

Scanning In order to use the scanner module we have to parameterise it with the reserved identifiers, the reserved operators, the special characters that are always recognised and always stand for themselves, and the characters that may be used to construct operator symbols.

```
slScan name  = scanFile keywordsText keywordsOps specialChars opChars name
keywordsText = ["if", "then", "else", "fi",
                "let", "in", "ni",
                "True", "False",
                "Int", "Bool", "Unit"
               ]
keywordsOps  = ["=", "\\", "->", "::", ":=",
                "&&", "||",
                "==", "/=", "<", ">", "<=", ">=",
                "+", "-", "*", "/"
               ]
specialChars = "();,[]"
opChars      = "!#$%&*+/<=>?@\\^|-:"
```

Parsing

```
pRoot _ = sem_Root_Root <$> pExprSeq
  where
  pExprSeq       = (λexprs → case exprs of
                               [e] → e
                               es → sem_Expr_Seq (foldr sem_Exprs_Cons sem_Exprs_Nil es))
                   <$> pSemics pAssignOrExpr
  pAssignOrExpr  = sem_Expr_Assign <$> pVarid <* pKey ":=" <*> pExpr
                   <|> pExpr
  pExpr          = pAndPrioExpr
                   <|> pLambda
  pLambda        = sem_Expr_Lam <$
                     pKey "\\" <*> pFoldr1 (sem_Vars_Cons, sem_Vars_Nil) pVarid
                                   <* pKey "->" <*> pExpr
  pAndPrioExpr   = pChainl (sem_Expr_Op <$> pOps ["&&", "||"])
                           pCmpPrioExpr
  pCmpPrioExpr   = pChainl (sem_Expr_Op <$> pOps ["==", "/=", "<", ">", "<=", ">="])
                           pAddPrioExpr
  pAddPrioExpr   = pChainl (sem_Expr_Op <$> pOps ["+", "-"])
```

```
                        pMulPrioExpr
pMulPrioExpr  = pChainl (sem_Expr_Op <$> pOps ["*", "/"])
                        pLamApply
pOps ops      = pAny pKey ops
pLamApply     = (λfs → case fs of                                          70
                  [fe] → fe
                  _    → sem_Expr_Lamcall (foldl1 sem_Expr_Apply fs)
                ) <$> pList1 pFactor
pFactor       = sem_Expr_Boolexpr True <$ pKey "True"
                  <|> sem_Expr_Boolexpr False <$ pKey "False"              75
                  <|> sem_Expr_Unit <$ pKey "Unit"
                  <|> sem_Expr_Ident <$> pVarid
                  <|> (sem_Expr_Intexpr ∘ string2int) <$> pInteger
                  <|> pParens pExpr
                  <|> sem_Expr_If                                          80
                    <$ pKey "if"   <*> pExpr
                    <* pKey "then" <*> pExpr
                    <* pKey "else" <*> pExpr
                    <* pKey "fi"
                  <|> sem_Expr_Let                                         85
                    <$ pKey "let" <*> pDecls
                    <* pKey "in"  <*> pExprSeq
                    <* pKey "ni"
pDecls        = foldr sem_Decls_Cons sem_Decls_Nil <$> pSemics pDecl
pDecl         = sem_Decl_Decl <$> pVarid <*> pTypeDecl <* pKey "=" <*> pExpr  90
pTypeDecl     = pKey "::" *> pType <|> pSucceed anytype
pType         = pChainr (makeFnType <$ pKey "->") pNonFuncType
pNonFuncType  = inttype <$ pKey "Int"
                  <|> booltype <$ pKey "Bool"
                  <|> unittype <$ pKey "()"                                95
                  <|> pParens pType
string2int = foldl (λval dig → (10 * val + ord dig − ord '0')) 0
```

5.1.3 Type structure

```
module SLTypes where

data Type = TAny
  | TBool                                                                  100
  | TInt
  | TUnit
  | TFunc Type Type

instance Show Type where
  show (TAny)      = "ANYTYPE"                                             105
  show (TBool)     = "Bool"
  show (TInt)      = "Int"
  show (TUnit)     = "()"
  show (TFunc f a) = "(" ++ show f ++ ")" ++ " -> " ++ show a

type Types = [Type]                                                       110

  -- basic types

booltype         = TBool
inttype          = TInt
```

```
unittype          = TUnit
anytype           = TAny
makeFnType f x = TFunc f x
isAnnotated TAny = False
isAnnotated _     = True
```

5.1.4 Top level AG (combining all aspects)

```
imports
{
import SLTypes
}
DATA Root
  | Root     Expr
DATA Expr
  | Unit
  | Intexpr  Int
  | Boolexpr Bool
  | Ident    var : String
  | Op       op : String le, re : Expr
  | If       cond, thenExpr, elseExpr : Expr
  | Let      decls : Decls expr : Expr
  | Assign   var : String expr : Expr
  | Apply    func, arg : Expr
  | Lamcall  call : Expr
  | Lam      vars : Vars expr : Expr
  | Seq      exprs : Exprs
DATA Decl
  | Decl     var : String type : Type expr : Expr
TYPE Decls = [Decl]
TYPE Exprs = [Expr]
TYPE Vars  = [String]
INCLUDE "SLPrettyprint.ag"
INCLUDE "SLEvaluate.ag"
INCLUDE "SLStaticErrors.ag"
```

5.1.5 Expression evaluation

In principle, evaluating an expression returns a *Value*. For simple expressions like *Intexpr* it is easy to compute such a result, namely the value of *Intexpr* itself. Computation of a value is also straightforward for operators *Op* because the operator only requires the value's of its operands. However, computation involving identifiers is bit more involved as the value associated with an identifiers is computed elsewhere, in the *Decls* of a *Let*. The solution employed here is to return a *Computation* instead of a *Value*. A *Computation* still expects an *Env* holding the *Value*'s for identifiers. The *Env* is used in an *Ident* and extended with additional bindings between identifiers and values in a *Let* and a *Lam*.

```
{
data Value          = UnitVal
                    | BoolVal Bool
```

```
                        | IntVal Int
                        | Function (Value → Value)
type Env         = [(String, Value)]
type EnvTransformer = Env → Env
type Computation = Env → (Value, Env)
instance Show Value where
  show UnitVal     =  "()"
  show (BoolVal b) =  show b
  show (IntVal i)  =  show i
  show (Function _) = "<! Function value !>"
}
ATTR Root [ | | val : Value]
SEM Root
  | Root lhs.val          = computeVal @expr.comp []
ATTR Expr [ | | comp : Computation]
SEM Expr
  | Unit     lhs.comp     = λenv → (UnitVal, env)
  | Intexpr  lhs.comp     = λenv → (IntVal @int, env)
  | Boolexpr lhs.comp     = λenv → (BoolVal @bool, env)
  | Ident    lhs.comp     = λenv → (lookupValue @var env, env)
  | Op       lhs.comp     = λenv → let lVal =  computeVal @le.comp env
                                      rVal =  computeVal @re.comp env
                              in (evalOp @op (lVal, rVal), env)
  | If       lhs.comp     = λenv → let condVal = computeVal @cond.comp env
                                       thenVal = computeVal @thenExpr.comp env
                                       elseVal = computeVal @elseExpr.comp env
                              in (evalIf condVal thenVal elseVal, env)
  | Let      lhs.comp     = λenv → @expr.comp (@decls.envTransformer env)
  | Assign   lhs.comp     = λenv → let (val, _) = @expr.comp env
                              in (UnitVal, (@var, val) : env)
  | Apply    lhs.comp     = λenv → let funcVal = computeVal @func.comp env
                                      argVal  = computeVal @arg.comp env
                              in (evalApply funcVal argVal, env)
  | Lam      vars.bodyComp =       @expr.comp
             lhs.comp     =       @vars.comp
  | Seq      lhs.comp     =       @exprs.comp
ATTR Exprs [ | | comp : Computation null : Bool]
SEM Exprs
  | Nil  lhs.comp         = λenv → (UnitVal, env)
         lhs.null         = True
  | Cons lhs.comp         = λenv → let (val, env') = @hd.comp env
                              in if @tl.null
                                 then (val, env')
                                 else @tl.comp env'
         lhs.null         = False
ATTR Vars [bodyComp : Computation | | comp : Computation]
SEM Vars
  | Nil  lhs.comp         =       @lhs.bodyComp
  | Cons lhs.comp         = λenv → (Function (λv → computeVal @tl.comp ((@hd, v) : env)), env)
ATTR Decls [ | | envTransformer : EnvTransformer]
SEM Decls
  | Nil  lhs.envTransformer = id
```

```
   | Cons lhs.envTransformer = λenv → @tl.envTransformer (@hd.envTransformer env)
ATTR Decl [ | | envTransformer : EnvTransformer ]
SEM Decl
   | Decl  lhs.envTransformer = λenv → (@var, computeVal @expr.comp env) : env
{                                                                                  205
computeVal :: (Env → (Value, Env)) → Env → Value
computeVal comp env = fst (comp env)
lookupValue :: String → Env → Value
lookupValue v env = maybe (lookUpError v) id (lookup v env)
evalOp :: String → (Value, Value) → Value                                          210
evalOp op vals =
  case op of
    "&&" → evalLogicalOp (∧) "&&" vals
    "||" → evalLogicalOp (∨) "||" vals
    "==" → evalRelOp (≡) "==" vals                                                 215
    "/=" → evalRelOp (≢) "/=" vals
    ">=" → evalRelOp (⩾) ">=" vals
    "<=" → evalRelOp (⩽) "<=" vals
    "<"  → evalRelOp (<) "<" vals
    ">"  → evalRelOp (>) ">" vals                                                  220
    "+"  → evalIntOp (+) "+" vals
    "-"  → evalIntOp (−) "-" vals
    "*"  → evalIntOp (×) "*" vals
    "/"  → evalIntOp div "/" vals
evalLogicalOp :: (Bool → Bool → Bool) → String → (Value, Value) → Value            225
evalLogicalOp op opString (lVal, rVal) =
  case (lVal, rVal) of
    (BoolVal b1, BoolVal b2) → BoolVal (b1 `op` b2)
    _                        → opError opString
evalRelOp :: (Int → Int → Bool) → String → (Value, Value) → Value                  230
evalRelOp op opString (lVal, rVal) =
  case (lVal, rVal) of
    (IntVal n, IntVal m)  → BoolVal (n `op` m)
    _                     → opError opString
evalIntOp :: (Int → Int → Int) → String → (Value, Value) → Value                   235
evalIntOp op opString (lVal, rVal) =
  case (lVal, rVal) of
    (IntVal n, IntVal m)  → IntVal (n `op` m)
    _                     → opError opString
evalIf :: Value → Value → Value → Value                                            240
evalIf cond thenVal elseVal =
  case cond of
    (BoolVal True)   → thenVal
    (BoolVal False)  → elseVal
    _                → ifCondError                                                 245
evalApply :: Value → Value → Value
evalApply funcVal argVal =
  case funcVal of
    Function f → f argVal
    _          → applyError                                                        250
lookUpError v = error $
```

```
                    "The variable " ++ v ++ " is unbound."
opError x       = error $
                    "The operator " ++ x ++ " is applied to arguments of the wrong type."
ifCondError     = error $                                                          255
                    "A non-boolean value occurred in the condition of an if statement."
applyError      = error $
                    "A non-function value occurs on the left hand side of an application."
}
```

5.1.6 Type checking

```
SEM Expr                                                                           260
  | Ident   loc.errmsg      = empty
  | Op      loc.lerr       = empty
            loc.rerr       = empty
  | Apply   loc.funcErr    = empty
            loc.argErr     = empty                                                 265
  | If      loc.condErr    = empty
            loc.ifErr      = empty
  | Assign  loc.varErr     = empty
            loc.exprErr    = empty
  | Lam     loc.argErr     = empty                                                 270
            loc.bodyErr    = empty
SEM Decl
  | Decl    loc.declErr    = empty
            loc.inferredType = text "<ERROR: Missing type annotation>"
```

5.1.7 Pretty printing

```
imports                                                                            275
{
import UU.Pretty
}
ATTR Root [ | | ppExpr : PP_Doc ]
SEM Root                                                                           280
  | Root lhs.ppExpr    = @expr.pp
SEM Expr  [ | | pp : PP_Doc ]
  | Unit     lhs.pp    = text "()"
  | Intexpr  lhs.pp    = text $ show @int
  | Boolexpr lhs.pp    = text $ show @bool                                         285
  | Ident    lhs.pp    = text @var >|< @errmsg
  | Op       lhs.pp    = @le.pp >#< @lerr >#< text @op >#< @rerr >#< @re.pp
  | Apply    lhs.pp    = pp_parens (@func.pp >|< @funcErr)
                           >#< pp_parens (@arg.pp >|< @argErr)
  | Let      lhs.pp    = text "let" >|< pp_block "" "" "; " @decls.pps             290
                           >-< " in" >#< @expr.pp
                           >-< " ni"
  | If       lhs.pp    = pp_ite "if " " then " " else " " fi"
                           (@cond.pp >|< @condErr) @thenExpr.pp (@elseExpr.pp >|< @ifErr)
  | Assign   lhs.pp    = text @var >|< @varErr >#< text ":=" >#< @expr.pp >|< @exprErr
  | Seq      lhs.pp    = pp_block "" "" "; " (@exprs.pps)
```

```
    | Lam     lhs.pp     = text "\\" >#< @vars.pp >|< @argErr >#< text "->"
                            >#< @expr.pp >|< @bodyErr
SEM Decls  [ | | pps : {[PP_Doc]} ]
    | Nil     lhs.pps    = []                                                    300
    | Cons    lhs.pps    = @hd.pp : @tl.pps
SEM Decl   [ | | pp : PP_Doc ]
    | Decl    loc.ppTp = if isAnnotated @type
                           then text ":" >#< show @type
                           else text ":" >#< show @inferredType                  305
              lhs.pp     = text @var >#< @ppTp >|< @declErr >#< text "=" >#< @expr.pp
SEM Exprs  [ | | pps : {[PP_Doc]} ]
    | Nil     lhs.pps    = []
    | Cons    lhs.pps    = @hd.pp : @tl.pps
SEM Vars   [ | | pp : PP_Doc ]                                                   310
    | Nil     lhs.pp     = empty
    | Cons    lhs.pp     = text @hd >#< @tl.pp
```

Chapter 6

Code Generation for a Stack Machine

In this chapter we will explore the way language features are mapped to a machine. First we will take a look at the internals of such a (simple) machine (Section 6.1). Then we will discuss how to generate code for expressions (Section 6.2), followed by a discussion of more complex language constructs such as functions and the use of variables (Section 6.3). Since not all details will be covered, the chapter is concluded by a small code generating compiler for SL using an attribute grammar (Section 6.4).

6.1 Stack machine model

All software executes on some hardware platform. A hardware platform involves many aspects like handling input and output, managing memory, and so on. The most central aspect however of some platform concerns the execution of instructions, done by a processor. Such a processor behaves according to some model and even though many different processors do exist they share many aspects. In this chapter the focus lies on a minimal set of most commonly available features necessary for mapping languages features to. For completeness a summary of leftout features is included in section 6.1.5.

6.1.1 Execution model

The basic execution model of a machine consists of *memory* (denoted by M) and a *central processing unit* (or *CPU*, *processor*), see Figure 6.1. CPU and memory are physically separated, though in practice this boundary is not so precise. The CPU reads and writes data from and to memory, possibly modifying the data in between. The way this is done is encoded in *instruction*s. A CPU basically does the following steps (described in pseudo Java code):

```
while ( ... )
{
  fetch instruction from memory ;
  execute instruction ;
}
```

Memory and registers Memory is the part of the model where data is stored. Instructions themselves are also stored as data. Memory is represented by a row of *memory unit*s. A memory unit is a row of *bit*s (*bitrow*). A bit can take 2 values: true or false, often written as 1 and 0. The

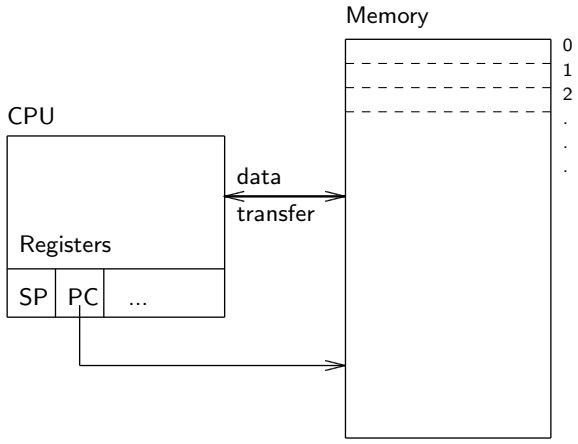


Figure 6.1: CPU with memory and registers

number of bits in a memory unit depends on the machine, often its size is 8 bits (*byte*). To make things simpler later on the size of a memory unit used here is 32 bits (*word*).

```
MemoryUnit memory[ size of memory ] ;
```

Memory behaves just like arrays in programming languages (like Java). Its value is retrieved using an index into the array. This index is called an *address*.

Memory is not the only storage available to a CPU. A CPU needs a bit of extra data in order to do its job. This bit of extra data is not stored in memory but in special locations inside a CPU, called *register*s. Access to registers is a lot faster (e.g. 10 times) than access to memory, being an important reason to use registers for data which is often needed and essential to a CPU. Though only a few values are essential to a CPU, often more registers can be found inside a CPU. These are called *general purpose register*s as they may be used freely in contrast to *special purpose register*s which are (by definition) used for some specific goal by the CPU. Two of these special purpose registers are the programcounter and the stackpointer.

Programcounter The CPU has to know where to fetch its next instruction from, it has to know the address of the next instruction to load. This address is stored in a register called *programcounter* (or *PC*). The CPU uses the program counter like this:

```
while ( ... )
{
  instruction = M[ PC ] ;
  PC += size of instruction ;
  execute instruction ;
}
```

Which instruction will execute is thus easily influenced by changing the value of the program counter.

Stack and Heap A CPU executes instructions. Instructions can read from and write to any memory location. This freedom easily creates chaos: how does a program know which location is free to be used or is already in use for other purposes? The most widely used and easiest solution to restrict this problem is to use a stack.

A *stack* conceptually stores values. A value is stored for later retrieval by *push*ing a value onto a stack. Retrieval is done by a reverse operation called *pop*:

```
Stack push( Stack, Value )
( Stack, Value ) pop( Stack )
```

An essential characteristic of a stack is that the most recent value pushed is the first to be popped. This behavior is called *last in first out* (or *LIFO*) and is expressed by the equation:

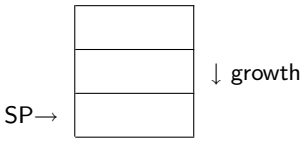
```
pop( push( stack1, value ) ) == ( stack1, value )
```

A stack is easily implemented in Java using an array and an index into that array pointing to the current top of the stack:

```
public class IntStack
{
    private int memory[] ;
    private int stackPointer = -1 ;

    public IntStack( int size )
    {
        memory = new int[ size ] ;
    }

    public void push( int v )
    {
        memory[ ++stackPointer ] = v ;
    }

    public int pop()
    {
        return memory[ stackPointer-- ] ;
    }
}
```

As a Java library this implementation lacks grossly in safety and robustness, for example because a check on under- or overflow is missing. It is however what most processors offer: an index into memory indicating the current top of the stack, called the *stackpointer* (or *SP*). Available instructions use and manipulate the stack by accessing values relative to the top of the stack and changing the stackpointer. The value of the stackpointer is stored in another special purpose register, also called the stackpointer (or SP).

Though the stack grows from lower to higher addresses this might just as well have been the other way around. The direction in which a stack grows does not make much difference as long as a convention is used consistently. Here we adhere to growth from lower to higher addresses while graphically displaying stacks in pictures with the higher addresses at the bottom, that is, a stack grows in the same direction as text is read:



At the right side general comment is included whereas on the left side the registers pointing to stacklocations are displayed. In this case the stackpointer is pointing to the top of the stack. As this is the case by definition the stackpointer will sometimes be left out of a stacklayout.

Though a stack was introduced in this paragraph via the operations `push` and `pop` acting on the the top of a stack, the actual use of a stack does not restrict itself to the top of a stack. This will become clear when instructions for manipulating the stack are introduced.

6.1.2 Instructions

A processor executes instructions. An instruction does something such as copying values or performing arithmetic. Since all data is stored as bitrows (see Section 6.1.3) this is also true for instructions themselves. An instruction is just a number which is interpreted by a processor as "do some specific instruction XXX". For example, the instruction `MUL` is represented by the number `0x8`. Since this is rather inconvenient a special purpose language is used, called *assembly language*.

Assembly language The format and syntax of an assembly language is usually very simple and straightforward. Each instruction is placed on a line of its own together with its operands. This may optionally be prefixed by a label or suffixed by comment. A label consists of a string followed by a colon ':', a comment starts with a semicolon ';' and is ended by the end of the line:

```
label:      Instr      Opnd's              ; Comment
```

An operand positioned after the instruction is called an *inline operand* in contrast to operands taken from other locations, like the stack. This separation is made since inline operands are constants, that is, assembled instructions with their inline operands do not change after code has started executing. Operands taken from a stack however may be different each time the same code with the same inline operands is executed.

Assembly language frees us from the necessity to remember the numbers representing instructions by allowing the programmer to use mnemonics instead. An assembly language usually offers additional services, these are discussed elsewhere in Section 6.2.2.

6.1.3 Values

Meaning of values The data stored in memory consists of bits without meaning. Values retrieved from memory on an address, or from a register, are simply rows of bits. Its meaning depends on the way these bits are used by instructions, that is, how these bits are interpreted. Common *interpretation*s include interpretation as integer or as address values. Instructions themselves are also stored in memory so the processor has a builtin interpretation of these bitrows.

Value encoding A value is encoded using a row of bits $b_{n-1} \dots b_0$, where n depends on the type of value. Bit b_0 is called the *least significant bit*, b_{n-1} is called the *most significant bit*. The size n of this bitrow usually also is a multiple of the number of bits used for a memory unit. In most current machines a memory unit consists of 8 bits, called a *byte*, 4 bytes often called a *word*. Here, the size of a memory unit equals 32 bits. An integer value is also encoded with a bitrow of 32 bits. This choice makes codegeneration as discussed here somewhat simpler. All other values (address, boolean) used here also fit into a memory word.

Integers are the most basic interpretation of a bitrow, we therefore need to specify the relationship between the encoding of an integer (the bitrow) and its value (the mathematical meaning):

$$\begin{aligned} b &= b_{n-1} \dots b_0 \\ value_{Int}(b) &= \sum_{i=0}^{n-1} b_i 2^i \end{aligned}$$

As addresses also are integers the same interpretation is used for addresses as well.

The value of $value_{Int}(b)$ will never be negative, for this reason integers encoded by this relationship are called *unsigned integers*. The lowest value which can be encoded is 0 (all 0 bits), the highest depends on the number of bits used, for $n = 32$ this is $4294967295 = 2^{n-1}$ (all 1 bits)

Negative values cannot be encoded this way so for *signed integer*s the most significant bit is used to encode the sign (positive or negative) in the following way:

$$value_{signed}(b) = \sum_{i=0}^{n-2} b_i 2^i - b_{n-1} 2^{n-1}$$

For $n = 32$, its range varies from $-2147483648 = -2^{n-2}$ (most significant bit 1, rest 0) to $2147483647 = 2^{n-2} - 1$ (most significant bit 0, rest 1). This encoding is the default one used for integers in most languages, including the simple language used in this chapter.

Boolean values often are encode by the integer value 0 for `False`, other integer valuese interpreted as `True`, see Section 6.2.2.

Other types Most languages and machines also use other kinds of values, such as floating point values and characters. Bitrows of different sizes and structure are used to encode these values. A discussion of these types is beyond the scope of these lecture notes.

6.1.4 SSM interpreter

In this chapter the code generation for a simple stack machine (SSM) is described. Though this machine does not exist in physical hardware (being too simple) a small interpreter for it exists. The interpreter allows the user to load assembly files and execute them. See [13] for more details about its use and Appendix B for a specification of SSM instructions.

6.1.5 Simplifications w.r.t. other machines

In order to be able to focus on the mapping from language features to machine features many aspects of other machines have deliberately been left out. This is a list of such features:

- Instructions generally do not restrict themselves to taking arguments from a stack only. Variations exist for getting them out of all kind of locations.
- Instructions are encoded more compactly
- Instructions for manipulating values other than integers exist: e.g. floating points, values occupying either more or less space of a memory unit, etc.
- A result of an instruction may not fit into some location. This is called overflow, the hardware of a processor automatically detects this.
- Errors (exceptions) may happen, e.g. division by zero, page faults. A processor has a handling mechanism for these situations.
- A processor often has mechanisms to support an Operating System: protection, parallellism and memory usage, interfacing with hardware.
- A processor may include performance increase measures which affect the way instructions behave.

6.2 Expression evaluation

Expressions are easily evaluated using a stack. The basic idea is that each expression consists of an operator acting on operands, for example, `*` acting on `6` and `5` in the expression `6*5`. The stack is then used to pick the operands from (by popping them). These operands are then used by an operator which leaves the result on the stack (by pushing). This will give the following sequence of stacklayouts for the expression `6*5`:

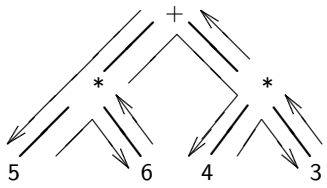
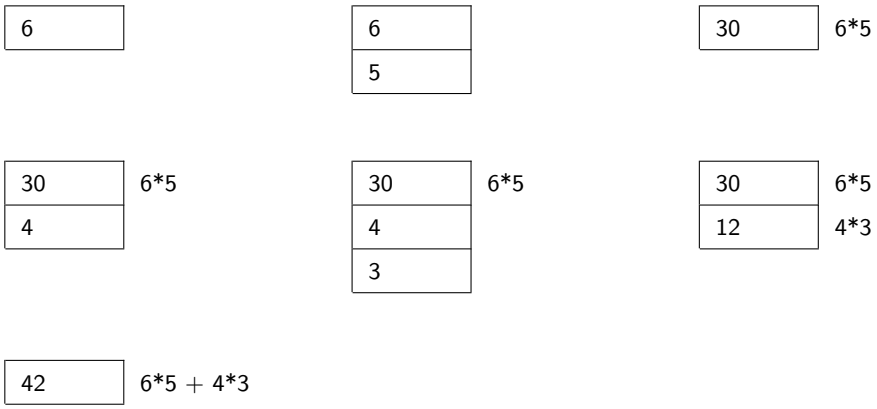


Figure 6.2: Tree representation and postfix walk of an expression

The result left on the stack can then be used for further processing, for example as the left operand of `+` in `(6*5) + (4*3)`. The complete evaluation order for this expression corresponds to a postfix treewalk of the tree representation of the expression, displayed in Figure 6.2. In this treewalk a visit to a constant means pushing the value onto the stack and the visit of an operator means popping the necessary arguments and replacing them by the result of the operation. The postfix treewalk `6 5 * 4 3 * +` gives the following sequence of stacklayouts:



6.2.1 Compilation scheme for operator and operands

Though it may be clear how the stack of a machinemodel may be used to evaluate expressions the actual job of doing this is the responsibility of instructions. For example, the evaluation of the expression `(6*5) + (4*3)` is accomplished by the following sequence of instructions:

```
LDC     6               ; push 6
LDC     5               ; push 5
MUL                     ; multiply
LDC     4               ; push 4
LDC     3               ; push 3
MUL                     ; multiply
ADD                     ; add
```

The instruction `LDC` loads (pushes) a constant on the stack, instructions `MUL` and `ADD` do what their names suggest: multiplication and addition by replacing the top two values of the stack with the result. `MUL` is the machine level counterpart of `*` and `ADD` is the machine level counterpart of `+`.

What still remains to be specified is the precise relationship between language constructs and instructions giving the result belonging to such a language construct. This is done using compilation schemes.

A *compilation scheme* gives the relation between language constructs and compiled instruction sequences by writing the compiled instructions as a function of a language construct:

$$
\begin{aligned}
&Scheme[[Construct]] \\
&\quad \equiv \quad CompilationPart_1; \\
&\qquad\quad\; CompilationPart_2; \\
&\qquad\quad\; \vdots \\
&\qquad\quad\; CompilationPart_n;
\end{aligned}
$$

Here, *Scheme* applied to a *Construct* gives a *Compilation* composed of $CompilationPart_i$'s, each $CompilationPart_i$ separated by a semicolon '`;`'. Each part is concatenated (textually) in the order it appears. If a compilation scheme needs extra information, this information is given to a scheme in the form of parameters, in the same way as written down in a functional language (like Haskell): $Scheme[[Construct]]\ p_1\ p_2 \ldots p_n$.

Within a compilation scheme the `verbatim or teletype` font is used to denote text which will be output of the codegeneration. Text written down in *italicmath* font also generates output but consists of expressions evaluated at compile time. The syntax for these expressions is not defined formally and assumed to be readable and understandable for anyone familiar with a functional programming language.

Many different schemes may exist depending on a specific language construct or compilation techniques. Here, for expressions, the compilation scheme is called $\mathcal{E}$. We need rules to write down the compilation scheme of an expression with operators and operands:

$$
\begin{aligned}
&\mathcal{E}[[Constant]] \\
&\quad \equiv \quad \texttt{LDC} \quad Constant
\end{aligned}
$$

$$
\begin{aligned}
&\mathcal{E}[[o_1\ op\ o_2]] \\
&\quad \equiv \quad \mathcal{E}[[o_1]]; \\
&\qquad\quad\; \mathcal{E}[[o_2]]; \\
&\qquad\quad\; \mathcal{O}[[op]];
\end{aligned}
$$

$$
\begin{aligned}
&\mathcal{O}[[+]] \\
&\quad \equiv \quad \texttt{ADD}
\end{aligned}
$$

$$
\begin{aligned}
&\mathcal{O}[[*]] \\
&\quad \equiv \quad \texttt{MUL}
\end{aligned}
$$

Since an operator itself is not an expression (at least in the SL language definition) we need a different compilation scheme for operators: $\mathcal{O}$. The compilation scheme for operators compiles operators to their corresponding instruction. It is used in the compilation scheme for expressions. Other operators like `-` and `/` (with the corresponding instructions `SUB` and `DIV`) are defined in a similar way.

As we will encounter other forms of expressions, the compilation schemes will have to be extended or new ones will have to be introduced.

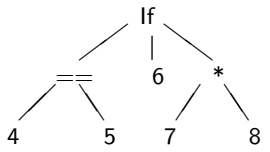


Figure 6.3: Tree representation of a conditional expression

Use of Abstract Syntax and Attribute Grammar A compilation scheme follows the (recursive) structure of the abstract syntax for a language, not its concrete syntax, even though the concrete form appears in the argument place of a compilation scheme. The abstract syntax of SL is used without introduction. The full definition of the abstract syntax for (most of) the language constructs discussed in this chapter can be found in Section 6.4. The definitions found there are written using an attribute grammar. Though textually written down in a different form the definitions resemble the compilations schemes used here.

As an example of the close relationship between a definition written using an attribrute grammar and compilation schemes the equivalent for operators operating on operands is given:

DATA *Expr*
  | *Op op* : *String le*, *re* : *Expr*
SEM *Expr* [∨ *code* : *Code*]
  | *Op* **lhs**.*code* = (@*le*.*code*
  ++ @*re*.*code*
  ++ **case**@*op* **of**
  "+" → "ADD"
  "*" → "MUL"
)

The result of a compilation scheme corresponds to the synthesized attribute `code`. Any additional parameters needed by a compilation scheme correspond to inherited attributes. Additional parameters are first necessary in Section 6.2.3.

The focus of this chapter however will be on the relationship between language constructs and its equivalent written down in machine level instructions. For this, compilation schemes will be the main notation since the corresponding attribute grammar definition contains implementation details which are necessary to make it work but are distractive at the same time.

Use of machine level instructions Machine level instructions are introduced where necessary without going into the details of their specific behavior. For example, `MUL` was introduced with a short remark about its workings and it was left up to the context to clarify this further. Other instructions will also be introduced with the minimal amount of explanation necessary. More detailed information about the used instructions can be found in [13] or Appendix B.

6.2.2 Conditionals

A *conditional expression* looks like:

```
if 4 == 5 then 6 else 7*8 fi
```

Its corresponding tree representation is displayed in Figure 6.3. The abstract syntax for a conditional expression looks like:

DATA *Expr*
  | *Op ce*, *te*, *ee* : *Expr*

As was the case with the abstract syntax for operators and operands, the subtrees of the tree in Figure 6.3 correspond to the three subexpressions `ce`, `te` and `ee`.

On the machine level no `if` instruction exists. The instruction set however offers instructions to jump to a different location, that is, to modify the contents of the programcounter. These instructions are called *branch instructions* (or *jump instructions*). Branch instructions come in two variations: *conditional branch instruction* and *unconditional branch instruction*. The unconditional branch instruction `BRA` (branch always) always jumps to a destination:

```
dest:       ...
            ...
            BRA       dest              ; branch to destination
```

Here, the code between `dest` and `BRA` is executed in an infinite loop (unless it contains a branch to a destination outside the loop).

On its own, the unconditional branch instruction is quite useless since it does not allow making a choice between jumping destinations. A conditional expression however pops the top of the stack first and jumps or jumps not depending on the value found on top of the stack. The `BRF` (branch on false) branches if it finds `False` on top if the stack, otherwise it does not jump and continues with the instruction following the `BRF`. A value is considered to be `False` if it equals zero, all other values are interpreted as `True`.

Finally, for a conditional expression, we need instructions producing a `True` or `False` value on top of the stack. This is done by *compare instructions*, for example `EQ` which, as the machine level counterpart of the operator `==` produces `True` on top of the stack if the two topmost values on the stack are equal.

The example conditional expression now compiles to:

```
            LDC       4
            LDC       5
            EQ                          ; if 4 == 5
            BRF       elselabel
            LDC       6                 ; then
            BRA       filabel
elselabel:  LDC       7                 ; else
            LDC       8
            MUL
filabel:    ...                         ; fi
```

These instructions calculate a boolean value. If this value is `False` the then part is skipped and the else part is executed. If the value is `True` the then part is executed and the else part is skipped.

Hidden assembly details The code presented here actually hides some machine level details because it is written in assembly language. Branch instructions have to know where to jump to. This information is encoded as the difference between the location of the branch destination and the location immediately after the branch instruction itself. For example, the preceding code actually should be:

```
            LDC       4
            LDC       5
            EQ                          ; if 4 == 5
            BRF       4
            LDC       6                 ; then
            BRA       5
            LDC       7                 ; else
            LDC       8
            MUL
            ...                         ; fi
```

Or even worse, because unreadable, and produced by the assembler (without the comments):

```
0x84        4
0x84        5
0xE                             ; if 4 == 5
0x6C        4
0x84        6                   ; then
0x68        5
0x84        7                   ; else
0x84        8
0x8
...                             ; fi
```

Compilation scheme In the following compilation scheme only the compilation for `==` is included.

$$
\begin{aligned}
&\mathcal{E}[[\texttt{if } ce \texttt{ then } te \texttt{ else } ee \texttt{ fi}]] \\
&\quad \equiv \quad \mathcal{E}[[ce]]; \\
&\qquad\quad \texttt{BRF} \quad elselabel; \\
&\qquad\quad \mathcal{E}[[te]]; \\
&\qquad\quad \texttt{BRA} \quad filabel; \\
&\qquad\quad elselabel : \mathcal{E}[[ee]]; \\
&\qquad\quad filabel :
\end{aligned}
$$

$$
\begin{aligned}
&\mathcal{O}[[==]] \\
&\quad \equiv \quad \texttt{EQ}
\end{aligned}
$$

The other five standard compare operators `<` (less then), `>` (greater then), `/=` (not equal), `<=` (less or equal) and `>=` (greater or equal) also have machine level counterparts: `LT`, `GT`, `NE`, `LE` and `GE`.

6.2.3 Let expressions

A *let expression* introduces a name for an expression:

```
let i :: Int = 5 * 6
 in i + 4 * i
 ni
```

In functional languages this is equivalent to the expression between `in` and `ni` with all occurrences of the introduced name replaced by its associated value:

```
(5 * 6) + 4 * (5 * 6)
```

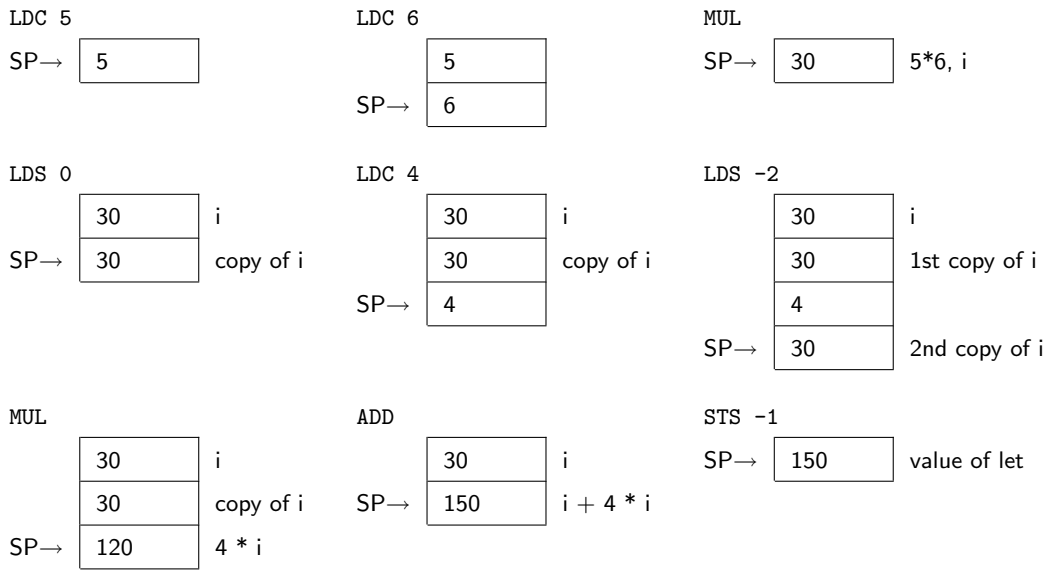
In an imperative languages this is generally not the case as it is not guaranteed that the value associated with a name is the same at all places where the name occurs. Somewhere between two such occurrences the value of a name may be changed by assigning a new value to it. We will consider assignments later on (Section 6.2.4) but the implementation discussed in this chapter is based on the premise that assignments have to be implemented.

Allowing the update of a value associated with an identifier means that we have to know where to put a newer value. This can be done by associating a memory location with a name instead of associating a value with the name. The memory location is then used to store the value for later retrieval. If a new expression is evaluated for the name, this location is then updated with the value resulting from the evaluation of the new expression.

Stackpointer relative addressing At which memory location is the value for a name stored? Since expressions are evaluated on a stack, and its result is left on the top of the stack, the simplest approach is just to leave it there and remember the distance (or *offset*, *displacement*) between its location and the stackpointer. The following code implements the example:

```
LDC     5
LDC     6
MUL                         ; 5*6, hereafter referred to as i
LDS     0                   ; i, 0 relative to SP
LDC     4
LDS     -2                  ; i, -2 relative to SP
MUL
ADD
STS     -1                  ; result on top, overwrite i
```

Its effect can be seen from the corresponding sequence of stacklayouts:



A couple of extra instructions are introduced with this example: `LDS` (load from stack) and `STS` (store on stack), used for copying the value of the memory location associated with a name and placing the result of the let expression at the right place. `LDS` copies (loads) a value relative to the top of the stack. `LDS` has one inline argument, the offset from the top of the stack. The sum of this offset and the value of the stackpointer (just before the execution of the instruction) is used as the address of the memory location from which the value is copied. `STS` does the reverse, it copies (stores) the value on top of the stack to a memory location relative to the top of the stack. The address of the memory location is calculated in the same way.

A few words about words: the combination of a name and its memory location is also called a *variable*, as the value of the name may vary. The variable is said to be *allocate*d on the stack. Giving the variable a value its first time is called the *initialization*, which, in this case, is combined with its allocation. The variable is accessed via *stackpointer relative addressing*.

Compilation scheme using stackpointer The compilation scheme now looks like:

$$
\begin{array}{l}
\mathcal{E}[[\texttt{let}\ id\ \texttt{::}\ type\ \texttt{=}\ e_1\ \texttt{in}\ e_2\ \texttt{ni}]]\ env\ depth \\
\quad \equiv \quad \mathcal{E}[[e_1]]\ env\ depth; \\
\qquad\quad \mathcal{E}[[e_2]]\ (\{id \overset{depth}{\longmapsto} depth\} \oplus env)\ (depth+1); \\
\qquad\quad \texttt{STS}\quad -1
\end{array}
$$

$$
\begin{array}{l}
\mathcal{E}[[o_1\ op\ o_2]]\ env\ depth \\
\quad \equiv \quad \mathcal{E}[[o_1]]\ env\ depth; \\
\qquad\quad \mathcal{E}[[o_2]]\ env\ (depth+1); \\
\qquad\quad \mathcal{O}[[op]];
\end{array}
$$

$$
\begin{array}{l}
\mathcal{E}[[id]]\ env\ depth \\
\quad \equiv \quad \mathcal{V}_l[[id]]\ env\ depth
\end{array}
$$

$$
\begin{array}{l}
\mathcal{V}_l[[id]]\ env\ depth \\
\quad \equiv \quad \texttt{LDS}\quad (env[id]_{depth} - depth)
\end{array}
$$

$$
\begin{array}{l}
\mathcal{P}[[expr]] \\
\quad \equiv \quad \mathcal{E}[[expr]]\ \{\}\ 0
\end{array}
$$

For this compilation scheme extra information needs to be passed around, a current depth *depth* of the stack, and an environment *env* remembering at which depth (measured from the initial depth 0) a variable can be found.

The operator $\oplus$ returns a new environment combining an additional mapping $id \overset{depth}{\longmapsto} depth$ (from identifier to a depth, the mapping is named 'depth') with another environment, and $env[id]_{depth}$ extracts the depth information for *id* out of the environment *env*.

Note that the compilation scheme for operators also had to be adapted in order to pass the correct values of *env* and *depth*. We also need an compilation scheme $\mathcal{P}$ to indicate what a program consists of and what the initial values of *env* and *depth* should be.

Markpointer relative addressing Even though an implementation using displacements relative to the stackpointer works, it is often not the preferred one. The reason for this is that in more complex situations the displacement may be unknown. For example, after the allocation and initialization of a variable, data with a variable size may have been pushed on the stack, that is, each time the code is executed the amount pushed may be different. Its size is only dynamically known at runtime instead of already being statically known during compilation. Only in the latter case the displacement value is known and can be put in the compiled code as an inline parameter of instructions.

In order to avoid such a restrictions often an extra register is used, known as *markpointer* (or *framepointer*), abbreviated by *MP*. Initially, the markpointer and stackpointer have the same value, they point to the same memory location. As the stackpointer changes, the markpointer does not change in our current compilation scheme, thus providing a stable point of reference for addressing variables, independent of the stackpointer.

In order to use locations relative to the markpointer two instructions, similar to `LDS` and `STS`, are used: `LDL` (load local) and `STL` (store local). `LDL` copies a value in the same way as `LDS` does but relative to the markpointer instead of to the stackpointer. This can be seen from the compilation of the let expression example using the markpointer (lines beginning with * have changed):

```
                LDC         5
                LDC         6
                MUL                         ; 5*6, hereafter referred to as i
    *           LDL         1               ; i, 1 relative to MP
                LDC         4
    *           LDL         1               ; i, (still) 1 relative to MP
                MUL
                ADD
                STS         -1              ; result on top, remove i
```

The administration for the compilation scheme only needs a few small changes for the use of variable:

$$
\begin{array}{l}
\mathcal{E}[[\texttt{let}\ id\ \texttt{::}\ type\ \texttt{=}\ e_1\ \texttt{in}\ e_2\ \texttt{ni}]]\ env\ depth \\
\quad \equiv \quad \mathcal{E}[[e_1]]\ env\ depth; \\
\qquad\quad \mathcal{E}[[e_2]]\ (\{id \overset{displ}{\longmapsto} (depth+1)\} \oplus env)\ (depth+1); \\
\qquad\quad \texttt{STS}\quad -1
\end{array}
$$

$$
\begin{array}{l}
\mathcal{E}[[id]]\ env\ depth \\
\quad \equiv \quad \mathcal{V}_l[[id]]\ env
\end{array}
$$

$$
\begin{array}{l}
\mathcal{V}_l[[id]]\ env \\
\quad \equiv \quad \texttt{LDL}\quad env[id]_{displ}
\end{array}
$$

Note that the use of *depth* has been replaced by *displ* as the depth/displ value is now used to administer the displacement relative to the markpointer.

Multiple declarations In the given example for let expressions only a single identifier was declared and initialized with a value. This is only to make clear how a let expression is evaluated on a machine level, most languages allow multiple declarations, for example like:

```
let i :: Int = 4
  ; j :: Int = i + 5
 in i + j
 ni
```

This expression can easily be rewritten to

```
let i :: Int = 4
 in let j :: Int = i + 5
     in i + j
     ni
 ni
```

which fits into the given compilation scheme.

See Section 6.4 for the way this has been solved using an attribute gramar. See Exercise 6-11 and Exercise 7-1 for issues involving mutually recursive declarations.

6.2.4 Assignments and sequences

An assignment allows modification of the value associated with an identifier, after it has been given its initial value:

```
let i :: Int = 5
 in i := i + 1
 ;  i * i
 ni
```

The result of this expression is 36.

The *assignment* operator `:=` looks like a normal expression but in many languages does not simply evaluate to a value as other expressions do. Its main purpose is to have a side effect on a specific memory location, the location associated with the identifier to which the expression is assigning to. In many languages (e.g. Java) an assignment evaluates to a value as well as modifies a location.

Here, an assignment only changes the memory location of the identifier, and is only allowed as part of a sequence of expressions between `in` and `ni` of a let expression. The last expression of this sequence is the result of the let expression. On a syntactical level the sequence of expressions are separated by semicolon's '`;`'. Both assignments and normal expressions may occur in the sequence.

The semicolon '`;`' can be considered to be a special kind of left associative operator which throws away its left operand and returns its right operand. For an implementation this means that the '`;`' operator has to remove from the stack what the evaluation of the left operand has left there. In order to be able to do this we need to know the size of what is left on the stack. This may vary however, depending on an operand of the '`;`' being either an assignment having $size == 0$ or another expression generally having $size \geqslant 0$. For simplicity, we assume that a compilation scheme can use a function `size :: Expr -> Int` returning the amount of memory units needed for storage of an expression. Again, the attribute grammar in Section 6.4 shows how to extract this information using a rudimentary type system.

The code for the example looks like:

```
LDC     5               ; i
LDL     1               ; copy of i
LDC     1
ADD                     ; i+1
STL     1               ; i := i+1
LDL     1               ; copy of i
LDL     1               ; copy of i
MUL
STS     -1              ; result on top, remove i
```

The instruction `STL` behaves like `STS` with the same difference as between `LDL` and `LDS`: using the markpointer instead of the stackpointer as the point of reference for adding the displacement. `STL` removes the top value of the stack and stores it in the specified location, leaving the stack on the same level as before the evaluation of the assignment.

Throwing away expression values A bit of extra work has to be done in the following situation:

```
let i :: Int = 5
 in 6 * 7
 ;  i * i
 ni
```

The expression `6*7` is evaluated and left on the top of the stack. It then has to be removed. The instruction `AJS` (adjust stack) is used for this purpose, it just adds the value of its inline parameter to the stackpointer:

```
...
LDC     6
LDC     7
MUL                         ; 6*7
AJS     -1                  ; and remove it
...
```

The given solution to this problem immediately raises the question why the problem exists in the first place. Why not forbid this situation since the expression is evaluated and never used? The answer lies for a part in the use of functions (see Section 6.3) in imperative languages which may have a side effect (via an assignment) and may be invoked purely for this side effect. A function may return a result but it will be thrown away. This is the choice made in current imperative languages as it makes life for the programmer a bit easier since a function can be written to return a result without having to take into account the fact that this result will or will not be used.

Compilation scheme The compilation scheme for assignments now looks like:

$$\begin{array}{l}\mathcal{E}[[id\ \texttt{:=}\ e]]\ env\ displ \\ \quad \equiv \quad \mathcal{E}[[e]]\ env\ displ; \\ \qquad\quad \mathcal{V}_s[[id]]\ env\end{array}$$

$$\begin{array}{l}\mathcal{V}_s[[id]]\ env \\ \quad \equiv \quad \texttt{STL} \quad env[id]_{displ}\end{array}$$

And for sequences of expressions:

$$\begin{array}{l}\mathcal{E}[[e_1\ \texttt{;}\ e_2]]\ env\ displ \\ \quad \equiv \quad \mathcal{S}[[e_1]]\ env\ displ; \\ \qquad\quad \mathcal{E}[[e_2]]\ env\ displ\end{array}$$

$$\begin{array}{l}\mathcal{S}[[e]]\ env\ displ \\ \quad \equiv \quad \mathcal{E}[[e]]\ env\ displ; \\ \qquad\quad \text{if } size(e) > 0 \\ \qquad\quad \text{then } \texttt{AJS} \quad -size(e) \\ \qquad\quad \text{fi}\end{array}$$

6.3 Functions

Functions as well as function invocations are expressions returning a value:

```
let id :: Int -> Int = \x -> x
  ; f :: Int -> Int = id
 in f (id 3)
 ni
```

The variable `id` is given as its initial value the *function expression* (or *lambda expression*) `\x->x`, the identity function. This value is also associated with the variable `f`. Both `id` and `f` invoke the same function, making the expression `f (id 3)` return `3`.

6.3.1 Invocation conventions

Let us first look at a simpler example and the way to fit this into existing compilation schemes:

```
let double :: Int -> Int = \x -> x + x
 in double 3
 ni
```

First, what information is needed to be able to call the function `double` in the expression `double 3`? A function encapsulates a calculation, it does something. This "doing something" is represented on the machine level by instructions. These instructions are stored in memory, so, if the starting address of the first instruction would be known a jump could be made to that location. When the function would be ready a second jump back to the location from where the first jump was made should be taken in order to continue with the expression where the function invocation occurs.

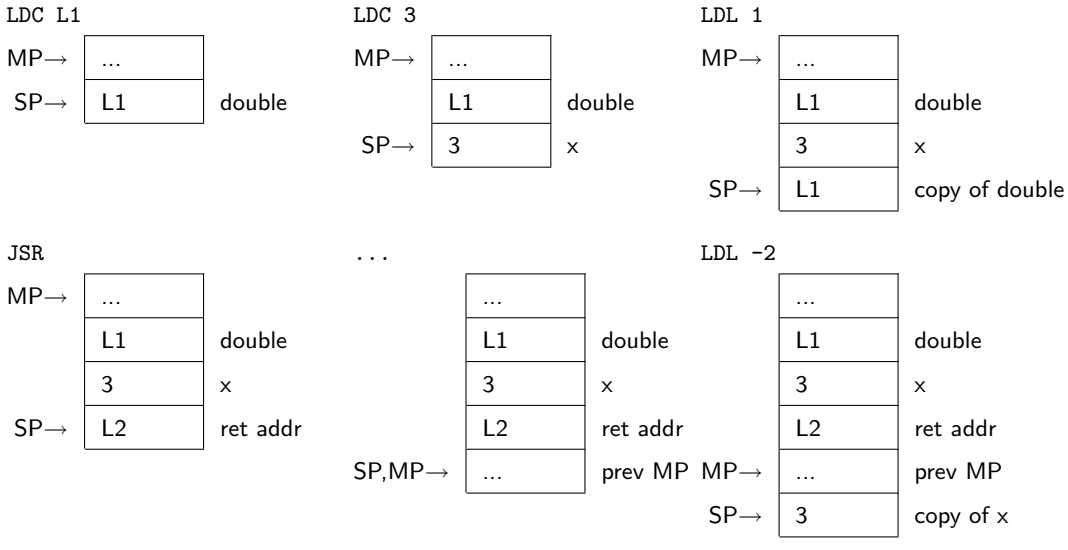
Second, the convention adopted so far is that operands of an operator are replaced by its result. In order to keep things simple functions should behave as much as possible the same way since a function conceptually does not differ from an operator: it also accepts operands and produces a result.

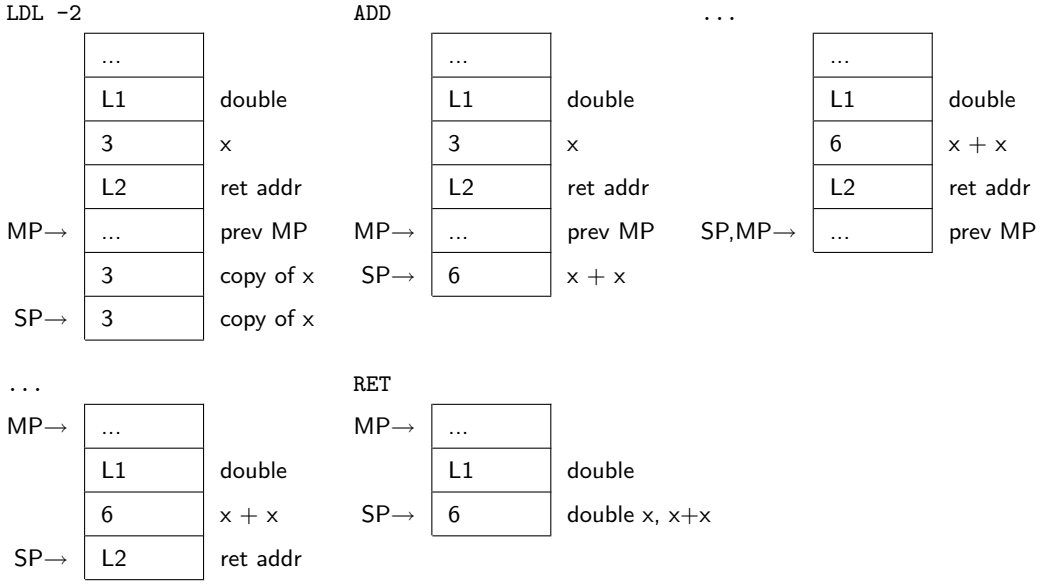
The following code reflects these two issues. The execution starts at the label `main`.

```
L1:         ...
            LDL     ...             ; copy of parameter x
            LDL     ...             ; copy of parameter x
            ADD
            ...
            RET
main:       LDC     L1              ; double
            LDC     3               ; argument for double
            LDL     1               ; copy of double
            JSR                     ; invoke double
L2:         ...
```

Two instructions, `JSR` and `RET` are needed to make this work. `JSR` (jump to subroutine) removes the top of the stack and jumps to that value. This is similar to a `BRA` instruction except that the destination location is taken from the stack instead of being derived from the programcounter and an inline parameter. Another essential difference is that the address of the location immediately after the `JSR` instruction is pushed on the stack just before the jump. This address – here labeled with `L2` – is given to the function as an extra argument since the function has to know where to return to when it is ready. The `RET` (return) instruction does the work of this return, it removes the so called *return address* from the stack and jumps to this address.

Other issues (indicated by the `...`) come to the surface when we try to follow the changes in the stacklayout:





The sequence of stacklayouts suggests that something has to be done with the markpointer.

6.3.2 Access to local variables and parameters

The body of a lambda expression may well be a let expression. The compilation scheme for a let expression assumes that the markpointer is set up so that it can be used to access local variables. This setting up has to be done just before the body of a lambda expression and undone just after it. The convention adopted here is to let the markpointer point to the same location as the stackpointer, which is just below the area used for the stack.

Another reason for adjusting the markpointer is that parameters now can be treated as local variables. They just happen to be pushed on the stack before the body of a lambda expression is evaluated. The only difference is that parameters are accessed using a negative displacement instead of a positive displacement with respect to the markpointer.

The missing part in the code presented so far takes care of setting up the markpointer correctly at the beginning of a function and restoring the original situation at the end of the function. Normally, another complication arises because the result of the function has to be put at the right location in the stack. Here it is not really a problem because the size of the arguments is the same as the size of the result, that is, 1 memory unit. The use of a `STL` is sufficient:

```
L1:         LDR     MP              ; save the previous MP
            LDRR    MP SP           ; copy the SP to MP
            LDL     -2              ; copy of parameter x
            LDL     -2              ; copy of parameter x
            ADD
            STL     -2              ; overwrite x with result
            LDRR    SP MP           ; restore SP
            STR     MP              ; restore MP
            RET
main:       LDC     L1              ; double
            LDC     3               ; argument for double
            LDL     1               ; copy of double
            JSR                     ; invoke double
L2:         ...
```

The code presented here is not the code which will do the job. We have not taken into account the way variables can be used, locally as well globally.

6.3.3 Local vs global

Locality of a variable is measured against a function or let expression.

```
let i :: Int = 5                (a)
  ; j :: Int = 6                (b)
  ; k :: Int = 7                (c)
  ; l :: Int = i + j + k        (d)
  ; f :: Int -> Int             (e)
       = \k -> let i = 8        (f)
               in i + j + k     (g)
               ni
 in let j :: Int = 9            (h)
    in f (i + j + k)            (i)
    ni
 ni
```

Which `i`, `j` and `k` are used at the various places in this example? The following table shows which identifier at a location corresponds to which variable, indicated by the location in the example program.

	variable		
location	i	j	k
(d)	(a)	(b)	(c)
(g)	(f)	(b)	(f)
(i)	(a)	(h)	(c)

Variables (or other definitions) are called *visible* at locations of a program if they can be used at those locations. The part of the program where variables are visible is called the *scope* of a variable. In SL two kinds of expressions allow the introduction of a variable: let expressions and lambda expressions. Let expressions define local variables and lambda expressions parameters. The definitions within let expressions (and lambda expressions) are said to be on a *lexical level*. Each new let expression (and lambda expression) introduces a new lexical level, one higher than the level of the let (or lambda) expression it is contained in. The lexical level of the program itself equals 0. The *lexical level* of a variable is equal to the lexical level of the let (or lambda) expression it is defined in. A variable defined in a let (or lambda) expressions is not visible in other let (or lambda) expressions on the same or lower lexical level. A variable defined in a let (or lambda) expression is visible in let (or lambda) expressions defined in the let (lambda expression) of the variable, with a higher lexical level. A variable may be made invisible by a definition on a higher lexical level of the same identifier of the variable.

These rules give the following table for the various variables:

variable	at	lexical level	visible at
i	(a)	1	(d), (i)
	(f)	3	(g)
j	(b)	1	(d), (g)
	(h)	2	(i)
k	(c)	1	(d), (i)
	(f)	2	(g)

A visible variable having the same lexical level as an expression is said to be a *local variable*, all other visible variables are said to be *global variable*s.

6.3.4 Let vs Lambda

As defined by the given level counting, the variable `j` at location `(i)` is at level 2, just as `i` at location `(g)`. However, because a nested let expression

```
let p :: .. = ..
 in let q :: .. = ..
     in
     ...
```

is equivalent to

```
let p :: .. = ..
 ;  q :: .. = ..
 in
 ...
```

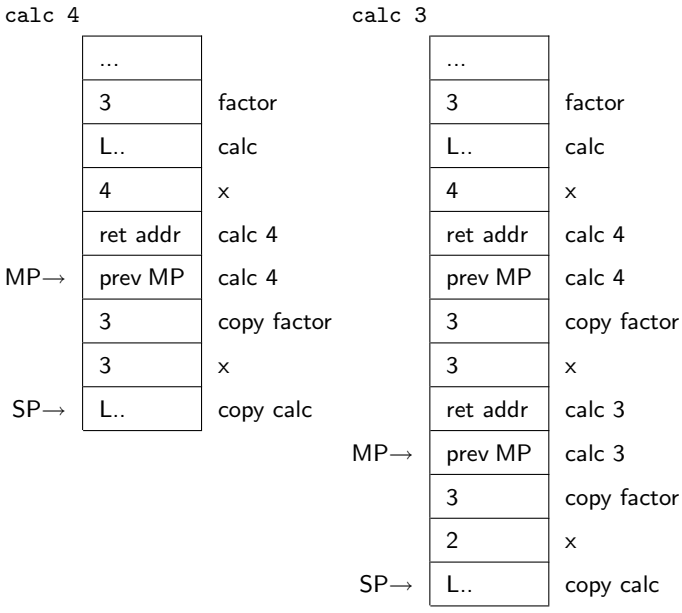
the increment with 1 for the level of a let expression (with respect to the surrounding expression) is not necessary. A let expression may just as well have the same level as the surrounding expression. Therefore, in the rest of this chapter variables introducted in a let expression have the same lexical level as the variables from the surrounding expression.

6.3.5 Access to global variables

Local variables as well as parameters are accessed via the markpointer, with a displacement known at compile time. Is this also possible for global variables? The answer is no, as shown by a counterexample recursively calculating a value:

```
let factor :: Int = 3
  ; calc   :: Int -> Int =
               \x -> if x > 0 then factor * calc (x-1) else 1 fi
 in calc 4
 ni
```

Assuming access to global variables via the markpointer can be done, the stacklayout between the invocation of `calc 4` and `calc 3` changes as follows:



The two stacklayouts show the situation just before the invocation of `calc 3` and `calc 2`. The displacement from the markpointer to (for example) the variable `factor` is -4 in the first stacklayout and -8 in the second and will vary even more as for recursive calls the depth of the recursion is unknown at compile time.

If we look a bit more closely at the stacklayouts we can see that the markpointer value used for accessing `factor` (and `calc`) cannot be found anymore in the markpointer register. Its value has been saved on the stack to make place for newer markpointer values. However, it is this previous markpointer we do need. An obvious, but still wrong, solution might be to follow the `prev MP` values until we arrive at the one needed for accessing `factor`. We only need to know how many times we need to follow the `prev MP` value, but this is, again, not possible to determine as the number of times to follow the `prev MP` depends on the depth of the recursion. As we already observed, the depth of recursion is unknown and consequently this solution does not work too.

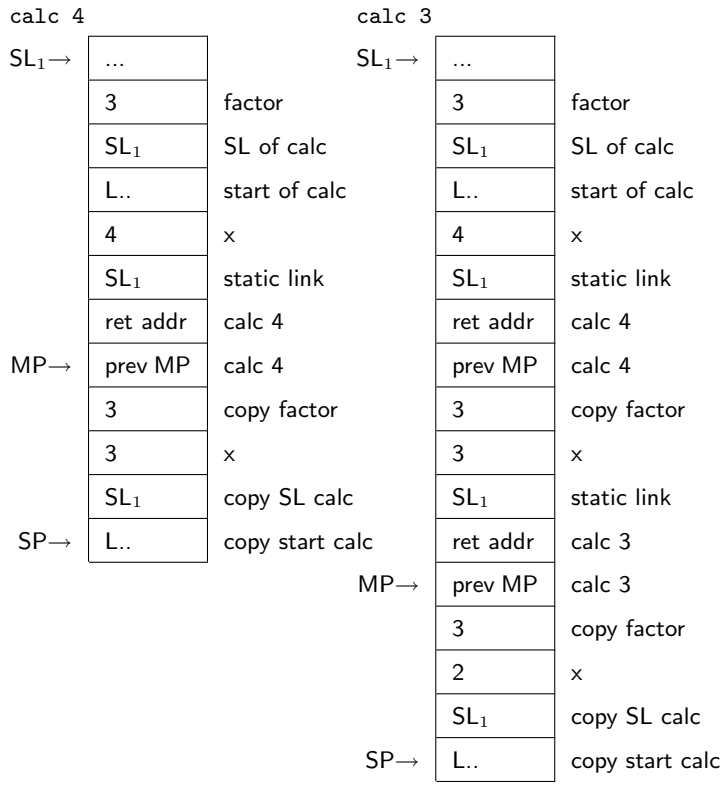
Another solution might be to pass all the extra information needed by a function as parameters to the function. For our example, the function `calc` needs to access `factor` as well as `calc` itself:

```
let factor :: Int = 3
  ; calc   :: .. -> .. -> Int -> Int =
                \c f x -> if x > 0 then f * c f (x-1) else 1 fi
 in calc calc factor 4
 ni
```

This solution encounters two problems. First, a copy of `factor` has been given. In the presence of assignments this will not work as the original may be changed while the copy has still the previous value of the original. This could be solved by passing the address of the variable instead of a copy of the value. However, a second problem still remains, the solution is quite inefficient: if many global variables exist many pointers to them are passed as extra parameters.

Static link The main conclusion of the preceding discussion is that extra information has to be passed to a function for accessing global variables. This then has to be done as efficiently as possible. In our example the lambda expression associated with `calc` has lexical level 2 and needs to access `factor` (and `calc`) on level 1. So we give this level 2 lambda expression what it needs: the markpointer value of the level 1 expression it is defined in so it can access `factor` and `calc` via this markpointer value.

This markpointer value always points to the variables of the textually enclosing let (or lambda) expression, one lexical level lower. Since the nesting structure of a program follows from the textual nesting of a program it is statical information. For this (also somewhat historical) reason the here introduced markpointer value is called *static link*, or *SL* (not to be confused with the abbreviated name 'SL' of "Simple Language"). By convention, the static link is always found between the first argument and the previous markpointer:



As the stacklayouts indicate the static link is the same for all recursive invocations of `calc`. The following code (slightly adapted from the output of the compiler from Section 6.4) uses and initializes the static link:

```
                LDC     3                   ; factor
                LDR     MP                  ; static link of calc
                LDC     L0                  ; start of code of calc
                BRA     L1                  ; jump over the code of calc
L0:             LDR     MP
                LDRR    MP SP
                LDL     -3                  ; copy of x
                LDC     0
                GT                          ; if x > 0
                BRF     L2                  ; then
                LDL     -2                  ; copy of static link
*1              LDA     1                   ; copy of factor (via static link)
                LDL     -3                  ; copy of x
                LDC     1
                SUB                         ; x-1
                LDL     -2                  ; copy of static link
*2              LDMA    2 2                 ; copy of calc (via static link)
                JSR                         ; invoke calc
                MUL
                BRA     L3
L2:             LDC     1                   ; else
L3:             STL     -3
                LDRR    SP MP
                STR     MP
                STS     -1
                RET
L1:             LDC     4
                LDML    2 2                 ; copy of calc
                JSR                         ; invoke calc
                STS     -3                  ; finalization of let expr
                AJS     -2
```

For each lambda expression now two memory units are used, one to store the static link and one to store the start address of the code (see also Exercise 6-8). The static link is easily calculated since the code for the initialization of a declaration of lexical level n always is executed within an expression of lexical level $n-1$. The static link for level n simply is the markpointer of level $n-1$.

This example also introduces some extra instructions. For example, loading `factor` on the stack (line marked with `*1`) now cannot be done with the `LDL` instruction because displacement is to be taken with respect to the static link loaded by the preceding instruction, instead of via the MP register. This is done with the `LDA` (load via address) instruction, which works similar to `LDS` and `LDL` but adds the displacement to the value popped from the top of the stack instead.

For loading values which consist of more than one memory unit yet another variant exists: `LDMA` (load multiple via address), having one extra inline parameter indicating the number of memory units to load. This instruction is used to load a copy of `calc` (line marked with `*2`), which now consists of two memory units. Similar variants do exist for the other load (and store) instructions.

6.3.6 Function as value

Functions also can be passed around as values themselves:

```
let apply :: (Int -> Int) -> Int -> Int =
             \f x -> f x
  ; incBy1 :: Int -> Int = \x -> x+1
 in apply incBy1 2
 ni
```

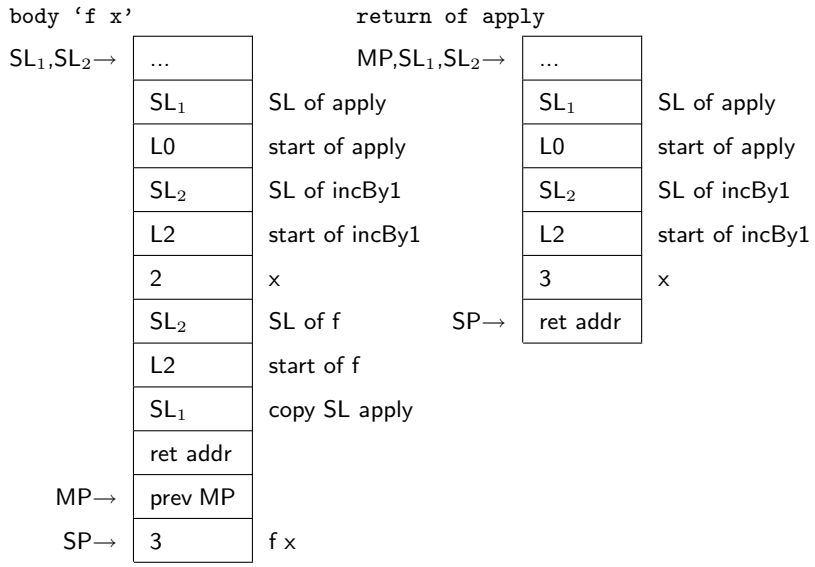
This gives the following piece of code:

```
                LDR     MP                  ; static link of apply
                LDC     L0                  ; start of code of apply
                BRA     L1
L0:             LDR     MP
                LDRR    MP SP
                LDL     -5                  ; copy of x
                LDML    -4 2                ; copy of f
                JSR                         ; f x
*1              STL     -5                  ; copy result over argument
                LDRR    SP MP
                STR     MP
                STS     -3                  ; move return address above result
*2              AJS     -2                  ; correct stack to point to return addr
                RET
L1:             LDR     MP                  ; static link of incBy1
                LDC     L2                  ; start of code of incBy1
                BRA     L3
L2:             LDR     MP
                LDRR    MP SP
                LDL     -3
                LDC     1
                ADD                         ; x+1
                STL     -3
                LDRR    SP MP
                STR     MP
                STS     -1
                RET
L3:             LDC     2
                LDML    3 2                 ; copy of incBy1
                LDML    1 2                 ; copy of apply
                JSR                         ; apply incBy1 2
                STS     -4
                AJS     -3
```

On the machine level function values are treated like any other value. On the machine level a value is just a bunch of memory units. The only difference between values is the number of memory units used to store the value.

6.3.7 Size of parameters and result

The example of the previous section also shows the solution to some complications arising when the size of the parameters and the size of the result are no longer equal. The function `apply` is passed 4 memory units (2 for the lambda value and 1 for the integer, 1 for the static link) while it returns 1 memory unit. This gives rise to two stacklayouts, the first being the situation after the body `f x` of `apply` has been evaluated, the second the situation just before returning.



The code between `*1` and `*2` now takes care of putting the result at the location of the parameters and moving the return address just above the result so that after the return instruction only the result is left on the stack.

A slightly different complication arises when the result is larger than the available space. In that case the result will not only overwrite the parameters but also the return address. In this case the return address must be saved somewhere before the result can be written to the right location. This is not worked out further here, but see Exercise 6-7.

Both of these problems arise because the "last in first out" behavior of a stack not really fits the order in which values become available and can be disposed of. Parameters, return address and result become available in the following order

1. Parameters available, pushed onto the stack
2. Return address available, pushed onto the stack
3. Result available, pushed onto the stack

but only can be disposed of, and are used, in the same order

1. Parameters no longer needed
2. Return address used for returning, then no longer needed
3. Result used by caller of function, then no longer needed

instead of the other way around as is normally the case with "last in first out". This problem only can be solved by cheating in some way. Here the cheat consists of replacing the parameters with the result without destroying the value of the return address.

6.3.8 Compilation schemes

Function invocation The compilation scheme for function invocation now looks like:

$$
\begin{array}{l}
\mathcal{E}[[f\ e_1 \ldots e_n]]\ env\ displ\ level \\
\quad \equiv \quad \mathcal{E}[[e_n]]\ env\ displ\ level; \\
\qquad\qquad \vdots \\
\qquad\qquad \mathcal{E}[[e_1]]\ env\ displ\ level; \\
\qquad\qquad \mathcal{E}[[f]]\ env\ displ\ level; \\
\qquad\qquad \texttt{JSR}
\end{array}
$$

Lambda expression The compilation scheme for lambda expressions:

$$
\begin{array}{l}
\mathcal{E}[[\backslash p_1 \ldots p_n \texttt{ -> } e]]\ env\ displ\ level \\
\quad \equiv \quad \texttt{LDR} \quad \texttt{MP}; \\
\qquad\qquad \texttt{LDC} \quad startlabel; \\
\qquad\qquad \texttt{BRA} \quad afterlabel; \\
\qquad\qquad startlabel: \\
\qquad\qquad \texttt{LDR} \quad \texttt{MP}; \\
\qquad\qquad \texttt{LDRR} \quad \texttt{MP SP}; \\
\qquad\qquad \mathcal{E}[[e]] \left(\begin{array}{ll} & \{p_i \overset{displ}{\longmapsto} -(2 + \sum_{j=1}^{i} size(p_j))\} \\ \oplus & \{p_i \overset{level}{\longmapsto} (level + 1)\} \\ \oplus & \{p_i \overset{size}{\longmapsto} (size(p_i))\} \\ \oplus & env \end{array} \right) 1\ (level + 1); \\
\qquad\qquad \mathcal{V}_i[[]]\ \texttt{STL STML}\ \left(-(2 + \sum_{j=1}^{n} size(p_j))\right)\ size(e); \\
\qquad\qquad \texttt{LDRR} \quad \texttt{SP MP}; \\
\qquad\qquad \texttt{STR} \quad \texttt{MP}; \\
\qquad\qquad \text{if } \sum size(p_i) \geqslant size(e) \\
\qquad\qquad \text{then } \texttt{STS} \quad -(1 + \sum size(p_i) - size(e)) \\
\qquad\qquad \text{fi}; \\
\qquad\qquad \text{if } \sum size(p_i) > size(e) \\
\qquad\qquad \text{then } \texttt{AJS} \quad -(\sum size(p_i) - size(e)) \\
\qquad\qquad \text{fi}; \\
\qquad\qquad \texttt{RET}; \\
\qquad\qquad afterlabel:
\end{array}
$$

Expression The compilation scheme for let and other expressions involving identifiers, now taking into account lexical levels:

$$
\begin{array}{l}
\mathcal{E}[[\texttt{let}\ id\ \texttt{::}\ type\ \texttt{=}\ e_1\ \texttt{in}\ e_2\ \texttt{ni}]]\ env\ displ\ level \\
\quad \equiv \quad \mathcal{E}[[e_1]]\ env\ displ\ level; \\
\qquad\quad \mathcal{E}[[e_2]] \left(\begin{array}{ll} & \{id \overset{displ}{\longmapsto} displ\} \\ \oplus & \{id \overset{level}{\longmapsto} level\} \\ \oplus & \{id \overset{size}{\longmapsto} size(e_1)\} \\ \oplus & env \end{array} \right) (displ + size(e_1))\ level; \\
\qquad\quad \mathcal{V}_i[[]]\ \texttt{STS STMS}\ (-(size(e_1) + size(e_2) - 1))\ size(e_2); \\
\qquad\quad \texttt{AJS} \quad (\textstyle\sum size(e_2) - size(e_1))
\end{array}
$$

$$
\begin{array}{l}
\mathcal{E}[[id]]\ env\ level \\
\quad \equiv \quad \mathcal{V}_l[[id]]\ env\ level
\end{array}
$$

$$
\begin{array}{l}
\mathcal{E}[[id\ \texttt{:=}\ e]]\ env\ displ\ level \\
\quad \equiv \quad \mathcal{E}[[e]]\ env\ displ\ level; \\
\qquad\quad \mathcal{V}_s[[id]]\ env\ level
\end{array}
$$

Variables and static link usage The compilation scheme for identifiers, now taking into account lexical levels and use of the static link:

$\mathcal{V}_l[[id]]\ env\ level$
$\equiv$ if $env[id]_{level} = level$
then $\mathcal{V}_i[[]]$ `LDL LDML` $env[id]_{displ}\ env[id]_{size}$
else
`LDL -2;`
$\mathcal{V}_l^{sl}[[id]]\ env\ (level - 1)$
fi

$\mathcal{V}_l^{sl}[[id]]\ env\ level$
$\equiv$ if $env[id]_{level} = level$
then $\mathcal{V}_i[[]]$ `LDA LDMA` $env[id]_{displ}\ env[id]_{size}$
else
`LDA -2;`
$\mathcal{V}_l^{sl}[[id]]\ env\ (level - 1)$
fi

$\mathcal{V}_s[[id]]\ env\ level$
$\equiv$ if $env[id]_{level} = level$
then $\mathcal{V}_i[[]]$ `STL STML` $env[id]_{displ}\ env[id]_{size}$
else
`LDL -2;`
$\mathcal{V}_s^{sl}[[id]]\ env\ (level - 1)$
fi

$\mathcal{V}_s^{sl}[[id]]\ env\ level$
$\equiv$ if $env[id]_{level} = level$
then $\mathcal{V}_i[[]]$ `STA STMA` $env[id]_{displ}\ env[id]_{size}$
else
`LDA -2;`
$\mathcal{V}_s^{sl}[[id]]\ env\ (level - 1)$
fi

$\mathcal{V}_i[[]]\ instr1\ instrm\ displ\ size$
$\equiv$ if $size = 1$
then $instr1\ displ$
else $instrm\ displ\ size$
fi

$\mathcal{P}[[expr]]$
$\equiv$ $\mathcal{E}[[expr]]\ \{\}\ 1\ 0$

The use of variables now has to incorporate varying sizes as well as the use of a static link for global variables. This results in a bit more complex compilation scheme where some case analysis is done. The resulting compilation schemes become more cluttered as this and other (future) information is used to write more optimized code. The meaning (semantics) of these schemes is precise enough to convey the ideas without dwelling to much on the implementation. This is where the attribute grammar system (used in Section 6.4) is preferred because the various issues can be separated and defined more exactly (as it will produce a working compiler).

6.4 Attribute grammar for code generating SL

The code generating version of the SL compiler, named *SLcg*. Parsing and pretty printing is the same as in the SLbb version, Chapter 5.

6.4.1 Main

```
import UU.Pretty
import UU.Parsing
import SLParser                                                          3
import SLTypes
import SLAttributes

slVersion
  = "Code Generation"
slDate                                                                   8
  = "20031106"
slIntroducedFeatures
  = "Code Generation & Type Checking"
slSignOn =
  "  ____    _" ++ "\n" ++                                              13
  "/ ___| | |" ++ "\tSL (Simple Language) Compiler.\n" ++
  "\\___ \\" ++ " | | " ++ "\tVersion " ++ slVersion ++ "/" ++ slDate ++ "\n" ++
  " ___) || |__  " ++ "\tIntroducedFeatures: " ++ "\n" ++
  "|____/ |_____| " ++ "\t\t" ++ slIntroducedFeatures ++ "\n"

compile filename = do                                                    18
  tokens ← slScan filename
  (codepp, exprpp, tp) ← parseIO (pRoot ()) tokens
  putStrLn ("Code generated for the expression:\n" ++ (disp exprpp 300 ""))
  let basename = takeWhile ((≠) '.') filename
  writeFile (basename ++ ".ssm") (disp codepp 80 "")                     23
  putStrLn ("Generated code in file " ++ basename ++ ".ssm")
  putStrLn "-------------------------------------------------------"

main = do
  putStrLn slSignOn
  putStr "Enter filename: "                                              28
  filename ← getLine
  compile filename
```

6.4.2 Type structure

```
module SLTypes where

import UU.Pretty

data Type = TAny                                                         33
  | TBool
  | TInt
  | TUnit
  | TFunc Type Type deriving Eq

instance Show Type where                                                 38
  show (TAny)      = "ANY"
  show (TBool)     = "Bool"
```

```
  show (TInt)       = "Int"
  show (TUnit)      = "()"
  show (TFunc f a) = "(" ++ show f ++ ")" ++ " -> " ++ show a                    43
type Types = [Type]
type Ids   = [String]
booltype = TBool
inttype  = TInt
unittype = TUnit                                                               48
anytype  = TAny
ppErr msg = text "<!ERROR:" > # < msg > | < text "!>"
makeFnType f x = TFunc f x
matchTypes TAny t   = (t, empty)
matchTypes t TAny   = (t, empty)                                               53
matchTypes t expType = if t ≡ expType
                       then (t, empty)
                       else (anytype, ppErr ("Expected " ++ show expType))
checkIfLambda tp = case tp of
  (TFunc f x)       → (f, x, empty)                                            58
  TAny              → (anytype,
                       foldr makeFnType anytype (repeat anytype),
                       empty)
  _                 → (anytype,
                       foldr makeFnType anytype (repeat anytype),               63
                       ppErr "Is not a function")
getFirstArgType []     = (anytype, repeat anytype)
getFirstArgType (a : as) = (a, as)
codeSize :: Type → Int
codeSize (TUnit)      = 0                                                      68
codeSize (TInt)       = 1
codeSize (TBool)      = 1
codeSize (TFunc f x)  = 2
codeSize _            = 0
extractResType (TFunc _ (TFunc a b)) = extractResType (TFunc a b)              73
extractResType (TFunc a b)           = b
extractResType _                     = anytype
extractArgTypes (TFunc a (TFunc b c)) = a : extractArgTypes (TFunc b c)
extractArgTypes (TFunc a b)           = [a]
extractArgTypes _                     = replicate 30 anytype                   78
makeFunctionType argTypes bodyTp = foldr makeFnType bodyTp argTypes
ifLambda tp = case tp of
  (TFunc _ _) → True
  _ → False
isAnnotated TAny = False                                                       83
isAnnotated _    = True
```

6.4.3 Top level AG (combining all aspects)

```
imports
{
```

```
import Prelude hiding (Ordering (..))
import SLTypes                                                          88
import SCode
}
DATA Root
  | Root     Expr
DATA Expr                                                               93
  | Unit
  | Intexpr  Int
  | Boolexpr Bool
  | Ident    var : String
  | Op       op : String le, re : Expr                                  98
  | If       cond, thenExpr, elseExpr : Expr
  | Let      decls : Decls expr : Expr
  | Assign   var : String expr : Expr
  | Apply    func, arg : Expr
  | Lamcall  call : Expr                                                103
  | Lam      vars : Vars expr : Expr
  | Seq      exprs : Exprs
DATA Decl
  | Decl     var : String type : Type expr : Expr
TYPE Decls = [Decl]                                                     108
TYPE Exprs = [Expr]
TYPE Vars  = [String]
INCLUDE "SLPrettyprint.ag"
INCLUDE "SLTypecheck.ag"
INCLUDE "SLCodeGen.ag"                                                  113
INCLUDE "SLLabel.ag"
```

6.4.4 Type checking

```
ATTR Root [ | | type : Type]
SEM Root
  | Root     lhs .type          = @expr.resType
             expr.typeEnv       = []                                    118
             expr.expType       = anytype
ATTR Expr [expType : Type || resType : Type]
SEM Expr
  | Unit     lhs .resType       = unittype
  | Intexpr  lhs .resType       = inttype                               123
  | Boolexpr lhs .resType       = booltype
  | Ident    loc .(restp, errmsg) = lookupType @var @lhs.typeEnv
             lhs .resType       = @restp
  | Op       loc .(restp, lerr, rerr)
                                = typeCheckOp @op @le.resType @re.resType 128
             lhs .resType       = @restp
  | If       loc .(restp, ifErr)  = matchTypes @thenExpr.resType @elseExpr.resType
             loc .(_, condErr)    = matchTypes @cond.resType booltype
             lhs .resType       = @restp
  | Let      lhs .resType       = @expr.resType                         133
  | Assign   loc .(vartp, varErr) = lookupType @var @lhs.typeEnv
```

```
                  loc .(_, exprErr)     = matchTypes @expr.resType @vartp
                  lhs .resType          = unittype
    | Apply       loc .(argtp, restp, funcErr)
                                        = checkIfLambda @func.resType                    138
                  loc .(_, argErr)      = matchTypes @arg.resType @argtp
                  arg .expType          = @argtp
                  lhs .resType          = @restp
    | Lam         loc .bodyType         = extractResType @lhs.expType
                  loc .(_, bodyErr)     = matchTypes @expr.resType @bodyType           143
                  loc .argTypes         = extractArgTypes @lhs.expType
                  loc .argErr           = if length @argTypes ≡ length @vars.resTypes
                                            then empty
                                            else ppErr "Wrong number of arguments"
                  vars.expTypes         = @argTypes                                    148
                  expr.expType          = @bodyType
                  lhs .resType          = makeFunctionType @vars.resTypes @expr.resType
    | Seq         lhs .resType          = if null @exprs.resTypes
                                            then anytype
                                            else last @exprs.resTypes                  153
ATTR Expr [typeEnv : TypeEnv | | ]
SEM Expr
    | Lam         expr.typeEnv          = addVarsToEnv @vars.variables @vars.resTypes @lhs.typeEnv
    | Let         expr.typeEnv          = @decls.typeEnv
ATTR Exprs [typeEnv : TypeEnv || resTypes : Types]                                     158
SEM Exprs
    | Cons        lhs .resTypes         = @hd.resType : @tl.resTypes
                  hd  .expType          = anytype
                  hd  .typeEnv          = @lhs.typeEnv
    | Nil         lhs .resTypes         = []                                           163
ATTR Vars [expTypes : Types || resTypes : Types variables : {[String]}]
SEM Vars
    | Cons        loc .(argtp, argtps)  = getFirstArgType @lhs.expTypes
                  tl  .expTypes         = @argtps
                  lhs .resTypes         = @argtp : @tl.resTypes                        168
                  lhs .variables        = @hd : @tl.variables
    | Nil         lhs .resTypes         = []
                  lhs .variables        = []
ATTR Decl [| typeEnv : TypeEnv |]
SEM Decl                                                                               173
    | Decl        expr.expType          = @type
                  loc .typeEnv          = addToEnv (@var, @type ) @lhs.typeEnv
                  expr.typeEnv          = @typeEnv
                  lhs .typeEnv          = @typeEnv
                  loc .declErr          = empty                                        178
                  loc .inferredType     = text "<ERROR: Missing type annotation>"
ATTR Decls [| typeEnv : TypeEnv |]
SEM Decls
    | Nil         lhs .typeEnv          = []
    | Cons        lhs .typeEnv          = @hd.typeEnv ++ @tl.typeEnv                   183
                  hd  .typeEnv          = @lhs.typeEnv

{
type TypeEnv = [(String, Type)]
```

```
typeCheckOp op leftType rightType =
  let check ltp rtp restp = (restp, snd (matchTypes leftType ltp), snd (matchTypes rightType rtp))
  in case op of
       "+"  → check inttype inttype inttype
       "-"  → check inttype inttype inttype
       "*"  → check inttype inttype inttype
       "/"  → check inttype inttype inttype
       "==" → check inttype inttype booltype
       "/=" → check inttype inttype booltype
       "<"  → check inttype inttype booltype
       ">"  → check inttype inttype booltype
       "<=" → check inttype inttype booltype
       ">=" → check inttype inttype booltype
       "&&" → check booltype booltype booltype
       "||" → check booltype booltype booltype
lookupType var env = maybe (anytype, ppErr "Undeclared variable")
                           (λtp → (tp, empty))
                           (lookup var env)
addToEnv v env           = v : env
addVarsToEnv vs tps env = (zip vs tps) ++ env
}
```

6.4.5 Instruction set

```
module SCode where
import Prelude hiding (Ordering (..))

newtype Label = Label Int
instance Show Label where
  show (Label i) = 'L' : (show i)

data S =
    HALT
  | TRAP    Int
  | AJS     Int
  | SWP
  | LDC     Int
  | LDLABEL Label
  | LDS     Int
  | ADD
  | SUB
  | MUL
  | DIV
  | EQ
  | NE
  | LT
  | GT
  | LE
  | GE
  | AND
  | OR
  | BRA     Label
  | BRF     Label
```

```
  | DEFLABEL Label
  | ANNOTE   String Int Int String String
  | LDMS     Int Int
  | LDL      Int                                                                    238
  | LDML     Int Int
  | LDR      String
  | LDRR     String String
  | LDA      Int
  | LDMA     Int Int                                                                243
  | STS      Int
  | STMS     Int Int
  | STL      Int
  | STML     Int Int
  | STR      String                                                                 248
  | STA      Int
  | STMA     Int Int
  | JSR
  | RET
instance Show S where                                                             253
  show s = case s of
    HALT              → "  HALT"
    TRAP x            → "  TRAP " ++ show x
    AJS l             → "  AJS  " ++ show l
    SWP               → "  SWP"                                                   258
    LDC c             → "  LDC  " ++ show c
    LDLABEL l         → "  LDC  " ++ show l
    LDS d             → "  LDS  " ++ show d
    ADD               → "  ADD"
    SUB               → "  SUB"                                                   263
    MUL               → "  MUL"
    DIV               → "  DIV"
    EQ                → "  EQ"
    NE                → "  NE"
    LT                → "  LT"                                                    268
    GT                → "  GT"
    LE                → "  LE"
    GE                → "  GE"
    AND               → "  AND"
    OR                → "  OR"                                                    273
    BRA l             → "  BRA " ++ show l
    BRF l             → "  BRF " ++ show l
    DEFLABEL l        → show l ++ ": "
    ANNOTE r l h cl t → "  annote " ++ r ++ "  " ++ show l ++ "  " ++ show h ++ "  " ++ cl ++ "  " ++ show t
    LDMS d s          → "  LDMS " ++ show d ++ "  " ++ show s                     278
    LDL d             → "  LDL  " ++ show d
    LDML d s          → "  LDML " ++ show d ++ "  " ++ show s
    LDR d             → "  LDR  " ++ d
    LDRR r1 r2        → "  LDRR " ++ r1 ++ "  " ++ r2
    LDA d             → "  LDA  " ++ show d                                       283
    LDMA d s          → "  LDMA " ++ show d ++ "  " ++ show s
    STS d             → "  STS  " ++ show d
    STMS d s          → "  STMS " ++ show d ++ "  " ++ show s
    STL d             → "  STL  " ++ show d
```

```
    STML d s        → "  STML " ++ show d ++ "  " ++ show s
    STR r           → "  STR  " ++ r
    STA d           → "  STA  " ++ show d
    STMA d s        → "  STMA " ++ show d ++ "  " ++ show s
    JSR             → "  JSR"
    RET             → "  RET"
opCode :: String → S
opCode op = case op of
  "&&" → AND
  "||" → OR
  "==" → EQ
  "/=" → NE
  ">=" → GE
  "<=" → LE
  "<"  → LT
  ">"  → GT
  "+"  → ADD
  "-"  → SUB
  "*"  → MUL
  "/"  → DIV
```

6.4.6 Label generation

Labels have to be unique. In a language like Java this is done by maintaining a global counter. When a new unique value is needed, the value of the counter is taken and the counter itself is incremented. In functional languages and attribute grammars this behavior is mimicked by passing this counter around as an attribute. Here, the counter is hidden inside a *Label*.

```
{
incrementLabel (Label i)        = Label (i + 1)
}
ATTR Expr Exprs Decl Decls [ | label : Label | ]
SEM Root
  | Root expr.label = Label 0
SEM Expr
  | If   loc.afterThenLabel = @lhs.label
         loc.afterElseLabel  = incrementLabel @lhs.label
         cond.label          = incrementLabel @afterElseLabel
  | Lam  loc.lamLabel        = @lhs.label
         loc.afterLamLabel   = incrementLabel @lhs.label
         expr.label          = incrementLabel @afterLamLabel
```

6.4.7 Code generation

Code generating means generation of text written in another language, assembly language. The generation of this text is done compositionally as indicated by the compilation schemes. The code for a syntactical construct is composed of the smaller parts of which the syntactical construct is built of, together with some gluecode to put the pieces together. Composition itself is just concatenation of these pieces.

However, because the concatenation ++ itself can be quite expensive, especially when working with

large lists, another (often used) solution has been used here. Instead of immediately concatenating, function composition is used. The function being composed will do the concatenation when eventually invoked but in a more efficient manner, by using the Haskell cons operator `:`.

The rest of the Haskell code here implements the details of the compilation schemes found elsewhere in this chapter.

```
INCLUDE "SLCodeGenEnv.ag"
{type CodeGen = [S] → [S]}
ATTR Root [ | | ppCode : PP_Doc]                                                    323
SEM Root
  | Root     lhs.ppCode      = vlist . map (text.show) . (@expr.code) . (TRAP 0:) . (HALT:) $ []
ATTR Expr [ | | code : CodeGen]
SEM Expr
  | Unit     lhs.code        = id                                                  328
  | Intexpr  lhs.code        = (LDC@int:)
  | Boolexpr lhs.code        = if @bool then (LDC 1:) else (LDC 0:)
  | Ident    lhs.code        = (loadLocal @lhs.level @idLevel @idDispl (codeSize @restp))
                               .(annoteCode ("copy of " ++ @var) (codeSize @restp))
             loc.(idLevel, idDispl)                                                333
                             = lookupVar @var @lhs.env
  | Op       lhs.code        = @le.code. @re.code.(opCode @op:)
  | If       lhs.code        = @cond.code
                               .(BRF @afterThenLabel:)
                               .@thenExpr.code                                     338
                               .annoteCode "then value" (codeSize @thenExpr.resType)
                               .(BRA @afterElseLabel:)
                               .(DEFLABEL @afterThenLabel:)
                               .@elseExpr.code
                               .annoteCode "else value" (codeSize @elseExpr.resType) 343
                               .(DEFLABEL @afterElseLabel:)
  | Let      loc.exprSize    = codeSize @expr.resType
             lhs.code        = @decls.code
                               .@expr.code
                               .store STS STMS (−(@decls.localVarsSize + @exprSize − 1))
                                     (codeSize @expr.resType)                      348
                               .(AJS (−(@decls.localVarsSize − codeSize @expr.resType)):)
  | Assign   lhs.code        = @expr.code
                               .storeLocal @lhs.level @varLevel
                                           @varDispl (codeSize @expr.resType)      353
             loc.(varLevel, varDispl)
                             = lookupVar@var@lhs.env
  | Apply    lhs.code        = @arg.code. @func.code
  | Lamcall  lhs.code        = @call.code.(JSR:)
  | Lam      lhs.code        = lamDefCode @lamLabel @afterLamLabel @vars.paramsSize 358
                                          @expr.code (codeSize @expr.resType)
  | Seq      lhs.code        = @exprs.code
ATTR Exprs [ | | code : CodeGen]
SEM Exprs
  | Nil      lhs.code        = id                                                  363
  | Cons     loc.codesize    = codeSize @hd.resType
             lhs.code        = let adjustSP = if null @tl.resTypes ∨ @codesize ≡ 0
                                              then id
                                              else (AJS (−(@codesize)):)
```

```
                                  in @hd.code.adjustSP. @tl.code                                  368
SEM Decl [ | | code : CodeGen localVarSize : Int]
  | Decl    lhs.code          = @expr.code
                                .annoteCode ("def of " ++ @var) (codeSize @type)
            lhs.localVarSize  = codeSize @type
SEM Decls [ | | code : CodeGen localVarsSize : Int]                                              373
  | Nil     lhs.code          = id
            lhs.localVarsSize = 0
  | Cons    lhs.code          = @hd.code. @tl.code
            lhs.localVarsSize = @hd.localVarSize + @tl.localVarsSize
SEM Vars [ | | paramsSize : Int]                                                                 378
  | Nil     lhs.paramsSize    = 0
  | Cons    lhs.paramsSize    = @tl.paramsSize + codeSize @argtp

{
lookupVar _ []                 = error "Unbound variable!"
lookupVar v ((v', x, y) : env) = if v ≡ v' then (x, y) else lookupVar v env                      383
store singleStore mulStore displ size =
  if size ≡ 1
  then (singleStore displ:)
  else (mulStore displ size:)
load singleLoad mulLoad displ size =                                                            388
  if size ≡ 1
  then (singleLoad displ:)
  else (mulLoad displ size:)
storeLocal fromLevel toLevel displ size =
  if fromLevel ≡ toLevel                                                                        393
  then store STL STML displ size
  else ((LDL (−(paramDispl)):)
       .(followStaticLink toLevel (fromLevel − 1))
       .(store STA STMA displ size)
       )                                                                                        398
loadLocal fromLevel toLevel displ size =
  if fromLevel ≡ toLevel
  then load LDL LDML displ size
  else ((LDL (−(paramDispl)):)
       .(followStaticLink toLevel (fromLevel − 1))                                              403
       .(load LDA LDMA displ size))
followStaticLink toLevel fromLevel =
  if toLevel ≡ fromLevel
  then id
  else ((LDA (−(paramDispl)):)                                                                  408
       .(annoteCode ("SL lev " ++ show fromLevel) 1)
       .(followStaticLink toLevel (fromLevel − 1)))
lamDefCode entryLabel afterLabel szParams bodyCode szResult =
  let disOfRET = szParams + sizeOfLamAdm − szResult − 1
      szOfAJS  = disOfRET − 1                                                                   413
  in (LDR "MP":)
     .(annoteCode "Static Link" 1)
     .(LDLABEL entryLabel:)
     .(annoteCode "Start PC" 1)
     .(BRA afterLabel:)                                                                         418
```

```
      .(DEFLABEL entryLabel:)
      .(LDR "MP":)
      .(ANNOTE "SP" 0 0 "blue" "Prev MP":)
      .(LDRR "MP" "SP":)
      .bodyCode                                                          423
      .(storeLocal 0 0 (−(paramDispl + szParams)) szResult)
      .(LDRR "SP" "MP":).(STR "MP":)
      .(if disOfRET > 0 then (STS (−(disOfRET)):) else id)
      .(if szOfAJS > 0 then (AJS (−(szOfAJS)):) else id)
      .(RET:)                                                            428
      .(DEFLABEL afterLabel:)
  annoteCode msg size = (ANNOTE "SP" (1 − size) 0 "green" msg:)
  sizeOfLamAdm = 2 :: Int
  }
```

6.4.8 Environment

Environment The environment remembers information for identifiers. For an identifier we need its type and its location in memory (level + displacement). The environment itself is implemented as a list.

Scope behavior is implemented straightforward. Any newly declared identifier is put in front of the environment together with its associated information. The search function (lookup) always stops at the first found identifier, which will by construction be from the most innermost level possible.

```
  {                                                                      433
  type Level = Int
  type Displ = Int
  type Env   = [(String, Level, Displ)]
  displOfLocalVars      = 1 :: Int
  sizeOfLambdaAdmin = 2 :: Int                                           438
  paramDispl            = 2 :: Int
  }
  ATTR Expr [level : Level env : Env | displ : Displ | ]
  SEM Root
    | Root expr.level = 0                                                443
      expr.env        = [ ]
      expr.displ      = displOfLocalVars
  SEM Expr
    | Intexpr  lhs.displ       = @lhs.displ + codeSize inttype
    | Boolexpr lhs.displ       = @lhs.displ + codeSize booltype          448
    | Ident    lhs.displ       = @lhs.displ + codeSize @restp
    | Op       lhs.displ       = @lhs.displ + codeSize @restp
    | If       thenExpr.displ  = @lhs.displ
               elseExpr.displ  = @lhs.displ
    | Let      lhs.displ       = @lhs.displ + codeSize (@expr.resType)   453
    | Apply    arg.displ       = @lhs.displ
               func.displ      = @arg.displ
               lhs.displ       = @func.displ
    | Lamcall  lhs.displ       = @lhs.displ + codeSize @call.resType
    | Lam      loc.lamLevel    = @lhs.level + 1                          458
```

$$
\begin{array}{lll}
expr.level & = @lamLevel & \\
expr.displ & = displOfLocalVars & \\
expr.env & = @vars.env & \\
vars.displ & = -(sizeOfLambdaAdmin) & \\
vars.level & = @lamLevel & 463 \\
vars.env & = @\mathbf{lhs}.env &
\end{array}
$$

SEM *Decl* [*level* : *Level* | *env* : *Env* *displ* : *Displ* |]
| *Decl* $\mathbf{loc}.declEnv = (@var,\ @\mathbf{lhs}.level,\ @\mathbf{lhs}.displ) : @\mathbf{lhs}.env$

$$
\begin{array}{lll}
expr.env & = @declEnv & \\
\mathbf{lhs}.env & = @declEnv & 468 \\
\mathbf{lhs}.displ & = @\mathbf{lhs}.displ + codeSize\ @type &
\end{array}
$$

SEM *Vars* [*level* : *Level* | *env* : *Env* *displ* : *Displ* |]
| *Cons* $\mathbf{loc}.varDispl = @\mathbf{lhs}.displ - codeSize\ @argtp$

$$
\begin{array}{lll}
tl.env & = (@hd,\ @\mathbf{lhs}.level,\ @varDispl) :\ @\mathbf{lhs}.env & \\
tl.displ & = @varDispl & 473
\end{array}
$$

ATTR *Decls* [*level* : *Level* | *env* : *Env* *displ* : *Displ* |]
ATTR *Exprs* [*level* : *Level* *env* : *Env* *displ* : *Displ* | |]

6.5 Exercises

6-1 **Pretty printing an expression with minimal parentheses**. When the SL compiler prints an expression, parentheses are included around all applications of operators, thereby making the precedences of the operators explicit. For example, the input

```
3 + 4 * 5 + 6
```

is printed as

```
((3 + (4 * 5)) + 6)
```

Modify the AG of the SL compiler in such a way that only the parentheses are printed that are really necessary to modify the natural precedence of the operators. For example, in `(3 + 4) * 5` the parentheses are necessary, but in `3 + (4 * 5)` they are not.

6-2 **SSM expression evaluation**. Translate the following expressions to SSM code:

1. `5+3`
2. `5+3-2`
3. `3-2+5`
4. `3-(-2)`
5. `3*2+5/6`
6. `(3*2+5)/6`

6-3 **Adding operators**. In the SL compiler, the operators `*` and `/` are defined. Add the missing operator `%`. The corresponding SSM instruction is `MOD`. Add also the unary operator `-`.

6-4 **Assigning assignment results**. In the language that is accepted by the compiler it is not possible to write expressions such as `a := (b := 3 + 4)` or `a := 3 + (b := 4)`. Which problems do occur if one tries to add this functionality, and how can these problems be solved?

The parentheses may be omitted. They are included in the examples to indicate that "`:=`" is right-associative. The idea of this extension is that an assignment itself results in a value – namely the value that is assigned – and that this value can thus be used in an expression again.

6-5 **Intermediate results of sequenced expressions**. If there is a sequence of expressions in the body of a **let** statement (between the **in** and ni keywords), the compiler expects that each expression that precedes a "`;`" does not leave anything on the stack. This is true for assignments ("`:=`"), but not for other expressions. The SL compiler therefore has a builtin correction that removes values that remain on the stack by means of a `AJS` instruction, when it encounters a "`;`" in such a situation. For example, in

```
let a :: Int = 2
 in a
 ;  a
 ni
```

the value produced by the first "`a`" is removed. The value produced by the second "`a`" remains, though, because it is the result of the sequence.

Now, if we allow expressions such as `a := (b := 3 + 4)` or `a := 3 + (b := 4)` (cf. Exercise 6-4), then assignments also produce values on the stack.

1. Adapt the grammar (the parser combinators) for SL so that these constructions will be accepted.
2. Modify the code generation so that an assignment *does* produce a value on the stack.
3. Optimize the code generation so that an assignment immediately before a "`;`" does not (unnecessarily) produce a value on the stack, which would just be removed again.

6-6 **While expression**. Add a "`while` construct" to the SL language so that we can write:

```
let a :: Int = 3
 ;  b :: Int = 1
 in while a > 0
    do b := b + a
    ;  a := a - 1
 ;  od
 ;  b
 ni
```

A `while` expression does not have a return value (here) and can (just as an assignment) only be used to change variables.

6-7 **Functions without arguments**. Functions in SL always have at least one argument. However, this is not a necessity: functions could also have side-effects only (by performing assignments) or compute a value without using arguments for the computation. Extend the SL compiler in such a way that it is no longer required for functions to have arguments. Watch out for the following caveats:

1. A function without arguments cannot be distinguished from the use of an identifier. When is a function a function, when is it a function call? Is a different syntax necessary? If yes, design one.
2. Is there space on the stack to store a result? If not, which modifications are required in code generation?

6-8 **Order of administration for functions**. The function administration consists of the static link and the start of the code for a function (see Section 6.3.5, page 91), put on the stack in this order. Investigate the consequences of putting the two values in reverse order. Adapt the code of the example containing `calc`.

6-9 **Stack layouts**. Draw the stacklayout and generate (by hand) the code for the following programs. Clearly indicate the values of (previous) markpointer(s), stackpointer and static link(s).

1.
```
let add :: Int -> Int -> Int = \x y -> x + y
 in add 3 4
 ni
```

2.
```
let fac :: Int -> Int =
            \n -> if n > 0 then n * fac (n-1) else 1 fi
 in fac 4
 ni
```

at the point where `fac 1` is called.

3.
```
let a :: Int = 5
 ;  f :: Int -> Int =
          \x -> let g :: Int -> Int =
                          \y -> let h :: Int -> Int = \z -> a
                                 in h 3
                                 ni
                in g 3
                ni
 in f 5
 ni
```

at the point where the body of `h` is executed.

4.
```
let a     :: Int               = 5
 ;  id    :: Int -> Int        = \x -> x
 ;  const :: Int -> Int -> Int = \x y -> x
 ;  f     :: Int -> Int        =
             \b -> let g :: Int -> Int -> Int =
                                \p q -> id a * const q p + b
                   in g 3 4
                   ni
 in f 7
 ni
```

at the point where `const q p` is called.

6-10 **Functions as result of function**. Look at the following program:

```
let apply :: (Int -> Int) -> Int -> Int =
              \f x -> f x
 ;  incBy :: Int -> (Int -> Int) =
              \x -> let f :: Int -> Int = \y -> x + y
                     in f
                     ni
 in apply (incBy 1) 3
 ni
```

The SL compiler results in the error `Wrong number of parameters` for `incBy`. The call `incBy 1` results in the error `Expected Int`. Assume that this program should be accepted (such as is common in functional languages). The result of the program is – according to the code generated by the SL compiler – *not* (as one would expect) `4`. Why not? What would have to be done to remedy the situation and get the expected result?

6-11 **Mutually recursive definitions**. In an example such as

```
let a :: ... = b
 ;  b :: ... = a
 in ...
 ni
```

the SL compiler reports an error at the definition of `a` claiming that `b` is not (yet) defined. This behaviour is not necessarily undesireable: if one looks at the stack implementation of SL, values are not computed lazily, and cyclic references in definitions would lead to non-terminating computations. However, in the case of function definitions mutual recursion can be useful an does not lead to implementation problems: for example,

```
let f :: ... -> ... = \x -> if ... then g x else ... fi
 ;  g :: ... -> ... = \x -> if ... then f x else ... fi
 in ...
 ni
```

is perfectly acceptable. Adapt the SL compiler so that it accepts programs such as the one just given, and translates them correctly.

6-12 **Partially parameterized function calls**. In SL it is possible to write functions that are only partially applied, such as in the following program:

```
let apply  :: (Int -> Int) -> Int -> Int = \f x -> f x
  ; sum    :: Int -> Int -> Int          = \x y -> x + y
  ; incBy1 :: Int -> Int                 = sum 1
 in apply incBy1 3
 ni
```

The effect is that in the generated program "`sum`" will be called (during the definition of "`incBy1`") without passing all necessary arguments.

1. Since the compiler obviously produces incorrect code in this situation, change the compiler to report an error if a function is partially applied.
2. An alternative solution would be to allow programs with partially applied functions, but to fix the code generation. Would that be possible with the given implementation of a stack machine? If yes, explain how – if not, explain why!

6-13 **Update operators (from Exam 20001023)**. The example language used in the practicals has a number of builtin operators. In some languages an infix operator such as "`+`" automatically also introduces an operator "`:=+`". The meaning of `a :=+ expr` is then that the value of `expr` is added to the value of `a`, and the result of the addition is also the result of the expression.

We assume that the abstract grammar for expressions has been extended to be able to represent the following "`InfixUpdate`" case:

```
DATA Expr |  InfixUpdate op: String var:  String right: Expr
```

1. Give a schema for the SSM code that has to be generated for such "update" expressions, i.e. explain how code for expressions such as `x :=+ y+3` has to be generated.
2. Some languages require that such update expressions are computed before the computation of a surrounding expression begins. For example, the expression `f x (x :=+ 3)` is then shorthand notation for

   ```
   x := x + 3; f x x
   ```

   As a consequence, code from within the expression has to end up before the expression. How has the attribute grammar of our example language to be modified in order to generate correct code in such situations? Which extra attributes are needed? Are extra nonterminals needed, and how do the attribute rules look like? Caveat: it is allowed that update expressions appear nested, i.e. the right hand side of an update expressions contains more updates; verify that your schema works in this case as well!

6-14 **Stackmachine code generatie (from Exam 20010504).** Look at the following SL program:

```
let y  :: Int = 1
 ;  f1 :: Int -> Int = \x -> x + y
 ;  f2 :: (Int->Int) -> Int -> Int
       = \f y -> let ff2 :: Int -> Int = \y -> f1 y
                 in if y > 0
                    then f2 ff2 (y-1)
                    else f 1
                    fi
                 ni
 in f2 f1 3
 ni
```

For this exercise we assume the *SLcg* compiler version.

1. What is the result of this program?
2. Draw the stack layout as it will have been created by the SSM at the moment that "`f2`" is called for the second time, just before the body of `f2` will be executed. Clearly indicate the locations of saved program counters (or return addresses), saved mark pointers, static links and variables in the stack layout. Also indicate with arrows where the used registers (SP, MP) and the values that are referenced point to. Finally, indicate clearly what the displacement of all the variables is.
3. Give the stack machine code for `f2` (the definition of `ff2` can be omitted).
4. The function `f1` could alternatively be defined as in the following program:

```
let y  :: Int = 1
 ;  addTo :: Int -> (Int -> Int)
          = \z -> let add :: Int -> Int
                             = \x -> x + z
                  in add
                  ni
 ;  f1 :: Int -> Int = addTo y
 ;  f2 :: (Int->Int) -> Int -> Int
       = \f y -> let ff2 :: Int -> Int = \y -> f1 y
                 in if y > 0
                    then f2 ff2 (y-1)
                    else f 1
                    fi
                 ni
 in f2 f1 3
 ni
```

   The SLcg compiler will produce error messages for this input code to warn about the contruction. However, we will ignore the errors for this subpart of the exercise: the generated code still works and produces, when executed, a result, albeit an incorrect one (we would expect the same result as in the first part of the exercise). Why the incorrect result?

Chapter 7

Type Systems

This chapter concerns itself with type inferencing, in the specific case of the language `SL`. The chapter is meant as an introduction to the attribute grammar implementation for the type inferencer given later on.

The main source of information for the first two sections is Cardelli [11]. The section on type rules is strongly related to the way type rules are treated in Section 7.6 and is meant to treat the subject in a more implementation independent manner. Section 7.6 itself has its roots in Damas and Milner [12].

The chapter is organized as follows. We start with the motivation for types and introduce various concepts that will help set the scene. We proceed then with a section on the difference between type checking and type inferencing.

We move to the more technical details by introducing the type language for the language `SL` and give the type rules of the type system of `SL`. In this, we start with the easier non-polymorphic parts of `SL` and conclude with a general introduction to polymorphism, a description of the process of unification and then finally the type rules that involve polymorphism in `SL`.

7.1 Why typing?

7.1.1 Avoiding run-time errors

The most important goal of type checking is to prevent run-time errors during the execution of a program. Examples of such errors are: multiplying an array by a character, calling a function with too many or too few parameters, calling a function with parameters of the wrong kind, dividing by zero, illegal memory referencing or the execution of an illegal instruction. The first of these errors you probably have never seen during the execution of your programs. The reason is that most compilers for programming languages use type checking to prevent these errors from occurring during execution.

A *type* of an identifier is the range of values that this identifier can take. In an *untyped language* anything goes: an identifier can take on any value, so during compile time there is nothing wrong with writing expressions such as `3(4)` or `"abc"/5`. In contrast a *typed language* is a language where identifiers can be given a (nontrivial) type. The rules for types in a given programming language are formalized in a *type system*, which tracks all the types of identifiers and expressions in a program. The goal of the type checker and type inferencer is to enforce a given type system. The difference between typed and untyped languages does not lie in the fact that one has types and the other does not, the difference lies in the fact that for one the correctness of the typing is enforced at compile-time and for the untyped case, at run-time. An untyped language such as

`Scheme` ([2, 14, 22]) does have types. However, errors such as using a list where an integer was expected will be discovered only during run-time.

In the same way that there are differences between the structure of programming languages, there are also differences between their respective type systems. For instance, if you call a function in Java with n parameters, then there has to be at least one function with the same name (defined within the scope) with exactly that number of parameters. In Haskell this is not the case, because by currying we might call a function with less parameters and obtain a function instead.

Another example of differences between type systems is that in a language such as `C` [23] or `C++` [29] there is really no difference between a boolean and an integer. In `Java` [6] and `Haskell` [27] it is an error to use an integer where a boolean was expected and vice versa. Usually, people will not want a character `'a'` to be added to an integer, but `C` (and to be honest, `Java` as well) will not make note of this fact. (Implicitly, it knows how to convert/cast characters to integers.) In this way, many possible sources of errors are not made noticeable by the compiler.

If a run-time error is caught by the operating system, then we call it a *trapped error*. Examples of these errors are division by zero, illegal memory referencing or the execution of an illegal instruction. In most cases, the operating system will terminate the program with, hopefully, a suitable error message.

The other kind of error is the *untrapped error*, the occurrence of which goes unnoticed, although its effects may be noticeable in a wrong outcome of the program. For instance the following Array Summing program

```
int j[] = new int[3];
int sum = 0;
for (int i = 0; i <= 3; i++)
   sum += j[i];
```

will generally yield an incorrect value in the variable `sum` when executed in `C++`, but generally no execution error. `Java` incorporates run-time array boundaries checking, so this progeam would result in a `java.lang.ArrayIndexOutOfBoundsException` exception being thrown. The disadvantage of untrapped errors should be evident: we have no way of knowing whether the outcome of a program can be trusted.

The goal of type checking is to prevent errors from occurring, in other words, to make programs well-behaved. A programming language is called *safe* if there are no untrapped errors, and this, one might argue, is the least we would want. Running the example above in `C` shows that not all programming languages are safe.

Languages such as Scheme or Lisp ([28, 1, 15]) are untyped, but perfectly safe. The safety is obtained by incorporating run-time checks in the program. In other words, it is possible to write and "compile" code in which you apply a list as a function, to eventually find out at run-time that this is not allowed.

Typed safe languages try to avoid doing run-time checks by statically determining where a program will go wrong and pointing this out to the programmer. As we will see later this is not always possible and most programming languages are a mix of static checks (as many as possible) and doing run-time checks for the parts that can not be statically checked. An example of the latter is addressing outside the boundaries of arrays, which in `Java`, as we saw earlier, gives rise to an exception being thrown, and not in a compilation error.

7.1.2 Efficiency of code

Another important reason for type checking is efficiency. If we know that a certain identifier may have values in a certain restricted interval then we know how much memory we have to allocate for that identifier. If we consider a programming language where no untrapped errors may occur,

then type checking can remove a large number of possibilities for errors and as a consequence we need not add special error checking code to the generated code to ensure that errors are trapped.

For example, take the Array Summing code defined earlier. A compiler might generate code here to the effect that what is actually compiled is:

```
int j[] = new int[3];
int sum = 0;
for (int i = 0; i <= 3; i++)
  if ((i >= 0) && (i < j.length))
      sum += j[i];
  else
      throw( new ArrayIndexOutOfBoundsException() );
```

From this code it should be evident that an exception will always be thrown. However, it can be proven that in general it is undecidable whether a certain position in or outside an array will be accessed. In other words: a program that discovers these errors beforehand in *all* cases does not exist. The problem is strongly linked to the well-known Halting Problem, which can be proven to be undecidable.

As a consequence, we have no choice but to include special code to check whether indexing outside array boundaries occurs at some point. Although you might come up with a clever program to detect indexing errors, this will never be a bulletproof program. In some implementations of programming languages run-time checks are optional, which may save execution time, but at the cost of the possible occurrence of untrapped errors.

A compiler that inserts code as illustrated in Array Summing example *does* make sure that the checks are done. The programmer is not anymore responsible for making these checks, and this will make programs easier to write and less cluttered by detail. However, in most professional programs the programmer will want to catch mistakes such as these himself, so that he can handle them according to his wishes. The throwing of exceptions is well-suited for this. A similar line of reasoning applies to division by zero.

As a final illustration of code efficiency: in `Haskell` it is possible to define new data types using constructors. Now, it is often the case that an identifier of such a datatype is passed to a function and matched against a number of possible patterns:

```
data IntTree = Leaf Int | Bin IntTree IntTree
data IntList = Null | Cons Int IntList
flatten :: IntTree -> IntList
flatten (Leaf x) = [x]
flatten (Bin x y) = flatten x ++ flatten y
```

If we call this function as in `flatten (Cons 1 (Cons 2 Null))` we are certain to get complaints from the Haskell compiler. The reason is of course that the type checker is able to determine that `(Cons 1 (Cons 2 Null))` is not a valid `IntTree`.

Note that because of the pattern match, we must be able to tell in the body of the function, whether `flatten` was passed a leaf, or a tree with two subtrees. This means that in actual generated code, we have to code in some way, what constructors are used in the construction of the parameter. However, as a result of having type checked the program, we do not have to verify during run-time whether the parameter is really of type `IntTree`. As a result, we do not need to code in the values of the identifiers what the type of the identifier is. As a consequence, the length of the coding is only determined by the number of constructors for a given data type and not by the amount of other data types we might have defined. In fact, identifiers in `Scheme` all bear such a coding to be able to tell functions, lists and integers apart at run-time. In this light, it is not surprising that you can not define new data types in `Scheme`, a serious drawback if you want to write large readable programs.

7.1.3 Efficiency of Development

Another very simple reason for having types in your program in the presence of a good type checker, is that, like the syntax checker, a type checker finds many errors during the writing of a program (instead of during the debug/testing phase).

There are interpreted languages where a line of code is checked when that specific line is executed. For large programs, this is unfeasible, because the number of possible executions paths is enormous: to be sure everything is alright, we have to check *all* of them. Worse yet, having made a change to the program, we might have to start all over again.

Another aspect is that in a functional programming language such as `Haskell`, which has a very powerful type system, it is often said by experienced programmers that if a program compiles, then it is correct. The reason for this is that most of the program errors are not in writing `x + y` instead of `x * y`, but because the programmer uses the wrong number of arguments, uses incompatible formal and actual parameters, uses undefined identifiers and so on. These are rather "small" errors, that might go easily undetected if it were not for the type checker.

Summarizing, the type checker helps you find many inadvertent mistakes, without having to go through the trouble of deducing their existence and then finding them yourself.

7.2 Type Checking versus Type Inferencing

In this chapter it is important to make a distinction between a type checker and a type inferencer. A *type checker* as used in Chapter 6 (Section 6.4) can only verify that a given typing of a program is consistent. Essentially, a type system (as given in this chapter) is a logic, and a type checker can check proofs in such a system. You have already seen a small type checker earlier in the course, because some type information was needed in the phase of code generation (Section 6.4).

A *type inferencer* is then comparable to a proof generator. In many logics finding a proof for a given theorem is undecidable. The type system of a programming language is chosen such that inferring the types of the expressions and identifiers can be done efficiently, because ultimately the goal of typing is a pragmatic one: we want to automatically detect type errors, and preferably without having the user supply all the typing information, as is the case in most imperative languages such as `Java` and `C`. This then is the advantage of type inferencing: the programmer need not explicitly write down the types of his identifiers, because they can be inferred by the compiler. In `Haskell` most types can be inferred, but there are extensions that can only be checked, not inferred.

It is possible that certain programs that are intuitively typable are forbidden by a type system. An example is $(\backslash x \rightarrow x\ \ x)\ \ (\backslash i \rightarrow i)$ where the reader might infer the type $(a \rightarrow a)$ for the result, but Haskell will not infer a type (Hugs gives the error "`unification would give infinite type`"). In other words, the type inferencer does not know how to handle it. The reason is that the expression $(\backslash x \rightarrow x\ \ x)$ has a recursive/infinite type and type checking is done in a compositional way: the type of a function application is determined from the type of the function and the type of its argument, and the latter two are determined *independently*. In this case the argument is such that a type could be inferred for the complete expression, but an algorithm usually does not take that fact into account. It is probably good that this is so, because it would very likely make the algorithm inefficient and arbitrary.

7.3 Type Language

For reasons of abstraction it is necessary to be able to describe in a concise and precise manner what exactly is the type system for which we want to build an inferencer. Before we go into the details of the type system used here, we shall first take a look at the types we shall encounter and see how to write them down.

7.3.1 Basic Types

In principle, a type system is anything you want it to be, but in most programming languages you find many of the same types.

A *basic type* is a type which can not be decomposed into other types. Examples of these from `Haskell` are `Int, Integer, Float, Char` and `Bool`.

In other languages, you might encounter various other basic datatypes (or the same ones, but under different names). In `Java` for instance, there are many basic datatypes that are similar to `Int` in `Haskell`, each with their own range of values. The reason for their existence is efficiency; if you know that a variable only has values in the range 2 up to 120, then a `byte` is sufficient to store it. Usually, this fact is only exploited when the need arises, because it makes programs more difficult to maintain. For `Java` basic types such as `byte, long, short, double`, but also `void` are present beside the equivalents of the basic types of `Haskell`.

For the remainder of this chapter, basic types are `Int` and `Bool`. For didactic reason, we keep the number of basic types small, because if you know how to handle one, you know how to handle them all.

For each basic type T, we have to decide what are the constants of type T. For type `Bool` these values will be `True` and `False`, and for the type `Int` we use the well-known values $\ldots, -2, -1, 0, 1, 2, \ldots$.

7.3.2 Composite Types

For *composite types*, types formed from other types, we again see a lot of the same types: language designers all want to define their own particular language, but the components from which types are built and the kind of composition used are rarely original.

In functional languages one often sees predefined lists and functions as first-class citizens: an identifier can as easily stand for a function as for an integer. In many imperative languages, it is possible to pass functions to functions and such, but often this is not very easy to use and not much advertised. This has a lot to do with the fact that programmers and designers of imperative languages have a different view of what computation is. The basic operation for them, it seems, is the assignment, and functions are only meant to structure sequences of assigments. Some datatypes often found in imperative languages are arrays, structs/records, classes, pointers and references. You will not see them here.

In languages such as Haskell it is also possible to define user defined data types such as the type `Tree` and `List` given earlier. In the language `SL` data types which involve constructors are not present (but, for example, see Exercise 7-7).

We will not treat the subject of polymorphism in our language until Section 7.5.

For the moment, we consider the following ways of constructing types from types. First, given two types T_1 and T_2 we can construct the type of functions from T_1 to T_2 denoted by $T_1 \to T_2$. The type T_1 is the argument type of the function, the type T_2 the result type. As is usually the case with functional programming languages we use currying to model functions of higher arity: a function with parameters of type T_1 and T_2 respectively and return type T is written $T_1 \to (T_2 \to T)$. As usual we assume right associativity for $\to$ and we may write $T_1 \to T_2 \to T$ instead.

The second kind of composite type is that of the (cartesian) product of two types T_1 and T_2. Often this type is referred to as a tuple. Such a type is rather similar to the records and structs of imperative languages, except that we refer to the separate fields by their position, not by a name. For the type language then: given two types T_1 and T_2, we can construct the type (T_1, T_2) of pairs of values of type T_1 and T_2 (in that order).

It goes without saying that arbitrary nestings of composite types are allowed. Although we shall extend the definition later to comprise more types, the basic type language for the language `SL` is

given by the following inductive definition.
A *type* is

- a basic type, i.e., `Int` or `Bool`, or
- a composite type in which case it is either
  - a functional type $T_1 \to T_2$ for types T_1, T_2, or
  - a product type (T_1, T_2) for types T_1, T_2.

A pair consisting of a boolean and an integer can be written as $(\texttt{Bool}, \texttt{Int})$. Note that the order matters, so this type is different from $(\texttt{Int}, \texttt{Bool})$. Examples of values with the former type are `(True,4)`, of the latter `(0,False)`.
The function exchanging the boolean and integer component of a pair has type

$$T = (\texttt{Bool}, \texttt{Int}) \to (\texttt{Int}, \texttt{Bool}) \ .$$

A question that should always be asked in any type system is when two given types are equal. There are two extremes: the first one says that types that have the same structure are equal (*structural equivalence*), the latter says that types with the same names are equal (*by-name equivalence*).
For example, look at the following `Haskell` program where we use *type aliasing* to define two "different" types from the same type.

```
type Name = String
type Address = String
```

Now what happens if we compare an identifier of type `Name` to an identifier of type `Address`. Is that a type error or not? Some people argue that it is, because otherwise you would not have wanted to make the distinction between the type of "names" and the type of "addresses". These people advocate the by-name equivalence of types. Other people consider that the underlying type of addresses and names is the same so it should be possible to compare identifiers of those types; this is called structural equivalence. Again it is a matter of choice and in many languages a compromise is reached. The important thing to remember is that a language designer has to make a choice and make it explicit in the manuals for the language and the code of the compiler.
In `Haskell` the type equivalence is structural. The following would happen in the example given earlier: the types Name and Address are unfolded into String, and from then on an identifier of type Name and an identifier of type Address automatically have the same type. In our language `SL`, and unfortunately also in `Java`, the question of type equality is void, because we do not have type definitions.
Another consideration in any type system is what is exactly the *scope* of an identifier. Essentially, the scope of an identifier `id` is that part of the code where we can refer to `id`. In most programming language you are allowed to redefine identifiers, but these different occurrences should be considered different identifiers: they just happen to have the same name. In our language and in most other languages, an identifier defined at a lower level *shadows* the definition on a higher level (see also Section 6.3.3). Although scope seems like a rather simple concept, its definition in languages such as `C` is often not well understood.

7.4 Type Inferencing

7.4.1 Type Rules

We will now introduce the type rules one by one, starting with the easiest ones. For the expressions `5` and `False` we will want to assign types to these constants, so that later we can use these types

to type the expressions of which these constants are a part. For instance if we encounter the expression `5 + False` we can then check whether the two arguments have the correct type and if so what the result type is.

Basic types and constants For the constants of our language it is sufficient to have the following rules:

$$\textsc{True} : \Gamma \vdash \mathsf{True} : \mathtt{Bool}$$

$$\textsc{False} : \Gamma \vdash \mathsf{False} : \mathtt{Bool}$$

$$\textsc{Integer} : \Gamma \vdash \mathsf{i} : \mathtt{Int}, \text{ if } i \in \{\ldots, -1, 0, 1, \ldots\}$$

Consider the first of these rules, which tells us what the type of the constant `True` is: it is `Bool`. The rule is given a name TRUE so that we can refer to it later. For the integers, we have an infinite amount of rules, for each integer there is exactly one, stating that an integer "5" or "2001" is of type `Int`; in this case the rules $\textsc{Integer}_5$ and $\textsc{Integer}_{2001}$ would be applied. The symbol $\vdash$ in these rules is usually pronounced "entails" , and we shall explain its use by the following rule, where we introduce identifiers.

Identifiers If we want to type check expressions such as `5 + x`, then we have to know at that point what the inferred type of x is. The definition or binding of x is somewhere else and the need exists for a structure in which we can collect the types for the identifiers that have been introduced. This structure is called an *environment* and shall be denoted by Γ. An environment may be used to store other, compiler and implementation dependent, information, but here this is not done. The environment Γ is simply a set of pairs $(\mathtt{id}, \text{type})$, which we can inspect and modify when necessary. With this piece of information we can introduce the following rule for introducing identifiers. The format of these rules is the same as in logic: the conclusion under the line holds if the conditions above the line are satisfied.

$$\textsc{Ident} : \frac{(\mathsf{id} : t) \in \Gamma}{\Gamma \vdash \mathsf{id} : t}$$

The previous derivation rule IDENT tells us the following: *If* in our environment Γ we have recorded the information that the identifier `id` is of type t, *then* we may conclude that the expression `id` has type t. The notation $\Gamma \vdash \ldots$ makes explicit that the conclusion after the $\vdash$ can depend on what is contained in Γ.

Application We can now type all the atomic elements in our programming language, but obviously we also need a rule for combining expressions and applying functions. This can be done by the following rule.

$$\textsc{Apply} : \frac{\Gamma \vdash \mathsf{expr_1} : a \to b, \Gamma \vdash \mathtt{expr_2} : a}{\Gamma \vdash \mathsf{expr_1}\ \mathsf{expr_2} : b}$$

Intuitively, if we know that a certain expression has a functional type $a \to b$ and we follow it by an argument of type a, then the result of the application is a value of type b. For example, if $\text{expr}_1 = \mathtt{fac}$ with type $\mathtt{Int} \to \mathtt{Int}$ and $\text{expr}_2 = 5$, then the result type b of `fac 5` is `Int`.

The environment Γ can also contain at the outset a number of functions such as `+` of type $\mathtt{Int} \to \mathtt{Int} \to \mathtt{Int}$ and `not` of type $\mathtt{Bool} \to \mathtt{Bool}$ and so forth. In this way we can mimick the fact that certain functions in our language are predefined and hence can be used in expressions.

Lambda expressions We shall also want to be able to introduce our own functions and for this we use lambda abstractions. The inference rule reads:

$$\textsc{Lambda}: \frac{(\Gamma - \texttt{id}) \cup \{(\texttt{id} : a)\} \vdash \texttt{expr} : b}{\Gamma \vdash \backslash\texttt{id} \rightarrow \texttt{expr} : a \rightarrow b}$$

where $\Gamma - \texttt{id}$ is the environment Γ from which we have removed all entries for the identifier `id`: $\Gamma - \texttt{id} = \{(a : t) \mid (a : t) \in \Gamma, a \neq \texttt{id}\}$.
This rule should be read as follows: if we can derive the type b for the expression `expr` given an environment Γ to which we add an identifier `id` of type a, then the lambda abstraction `\id -> expr` has a functional type $a \rightarrow b$. This is one of the places where we add something to the environment: the identifier to be abstracted. Because we add the identifier only to the environment that is used to infer the type for the body of the lambda abstraction, it implies that the *scope* of the identifier `id` is limited to the body of the lambda abstraction. Before passing the environment to the body of the lambda abstraction, we remove from Γ all previous references to identifiers of the same name, because they are now shadowed by the new binding. In practical situations (Section 7.6) this is solved less explicit and more efficiently, but also less neatly as seen from the viewpoint of the underlying logic as used here.

Conditional expressions The type inferencing rule for conditional expressions reads:

$$\textsc{Cond}: \frac{\Gamma \vdash \texttt{expr}_1 : \texttt{Bool}, \Gamma \vdash \texttt{expr}_2 : a, \Gamma \vdash \texttt{expr}_3 : a}{\Gamma \vdash \textbf{ if } \texttt{expr}_1 \textbf{ then } \texttt{expr}_2 \textbf{ else } \texttt{expr}_3 \textbf{ fi } : a}$$

This rule is not so difficult: given a boolean expression and two expressions that must have the same type a, we can derive that the conditional expression has the type of the alternatives. We have to insist that the types of the `then` and `else` part are the same to be able to derive a unique type for the conditional.

Products (or Tuples) As a final rule, for the moment, we introduce products:

$$\textsc{Product}: \frac{\Gamma \vdash \texttt{expr}_1 : a, \Gamma \vdash \texttt{expr}_2 : b}{\Gamma \vdash (\texttt{expr}_1, \texttt{expr}_2) : (a, b)}$$

7.4.2 Type derivation

We now look at a more complicated example to see how it should be typed. We shall build a *derivation tree* for a small piece of code, similar to derivation trees for proofs in logic (in fact what we do here is an instance of this process, where the rules given earlier are the deduction rules of the logic). We could give the derivation tree just like that and ask you to verify it is correct, but because we will later have to use the rules above to drive the process of type inferencing we shall show how the derivation tree might be found. We assume an initial environment

$$\begin{aligned}\Gamma_b = \{ \quad & (*, \texttt{Int} \rightarrow \texttt{Int} \rightarrow \texttt{Int}),\\ & (\texttt{fac}, \texttt{Int} \rightarrow \texttt{Int}),\\ & (\texttt{not}, \texttt{Bool} \rightarrow \texttt{Bool})\}\end{aligned}$$

Consider the following piece of code (where `(*) 299 x` is equivalent to `299 * x`)

```
( 10, if (not False)
      then (\x -> (*) 299 x)
      else fac
      fi
)
```

$$\dfrac{\dfrac{\dfrac{(* : Int \to Int \to Int) \in \Gamma_2}{\Gamma_2 \vdash * : Int \to Int \to Int}\ \textsc{Ident} \quad \dfrac{}{\Gamma_2 \vdash \mathsf{299} : Int}\ \textsc{Integer}}{\Gamma_2 \vdash (*)\ \mathsf{299} : Int \to Int}\ \textsc{Apply} \quad \dfrac{(\mathsf{x} : Int) \in \Gamma_2}{\Gamma_2 \vdash \mathsf{x} : Int}\ \textsc{Ident}}{\Gamma_2 \vdash (*)\ \mathsf{299}\ \mathsf{x} : Int}\ \textsc{Apply}$$
$$\dfrac{\Gamma_2 \vdash (*)\ \mathsf{299}\ \mathsf{x} : Int}{\Gamma_1 \vdash \lambda \mathsf{x} \to (*)\ \mathsf{299}\ \mathsf{x} : Int \to Int}\ \textsc{Lambda}$$

where $\Gamma_1 = \Gamma_b$ and $\Gamma_2 = \Gamma_b \cup \{(x : \texttt{Int})\}$

Figure 7.1: The derivation subtree for $(\backslash\ x\ \to\ *\ 299\ x)$

$$\dfrac{\dfrac{}{\Gamma_1 \vdash 10 : Int}\ \textsc{Integer} \quad \dfrac{\dfrac{\dfrac{(\mathsf{not} : Bool \to Bool) \in \Gamma_1}{\Gamma_1 \vdash \mathsf{not} : Bool \to Bool}\ \textsc{Ident} \quad \dfrac{}{\Gamma_1 \vdash \mathsf{False} : Bool}\ \textsc{False}}{\Gamma_1 \vdash \mathsf{not\ False} : Bool}\ \textsc{Apply} \quad \boxed{\text{Figure 7.1}} \quad \dfrac{(\mathsf{fac} : Int \to Int) \in \Gamma_2}{\Gamma_2 \vdash \mathsf{fac} : Int \to Int}\ \textsc{Ident}}{\Gamma_1 \vdash \mathbf{if}\ (\mathsf{not\ False})\ \mathbf{then}\ (\lambda \mathsf{x} \to (*)\ \mathsf{299}\ \mathsf{x})\ \mathbf{else}\ \mathsf{fac}\ \mathbf{fi} : Int \to Int}\ \textsc{Cond}}{\Gamma_1 \vdash (10, \mathbf{if}\ (\mathsf{not\ False})\ \mathbf{then}\ (\lambda \mathsf{x} \to (*)\ \mathsf{299}\ \mathsf{x})\ \mathbf{else}\ \mathsf{fac}\ \mathbf{fi}) : (Int, Int \to Int)}\ \textsc{Product}$$

where $\Gamma_1 = \Gamma_b$ and $\Gamma_2 = \Gamma_b \cup \{(x : \texttt{Int})\}$

Figure 7.2: A first derivation tree

The expression is a pair at top level, so the rule to apply at top level will be PRODUCT (see Figure 7.1 and Figure 7.2). To obtain the types from which this product is built, we shall have to type the two subexpressions. The first of these is `10`, the type of which can be found by applying the rule $\textsc{Integer}_{10}$. Hence the type of the first element of the product is `Int`.

To complete the rule for PRODUCT we continue with the second expression. This expression is a conditional at top-level, which means that we have to identify and type the condition, the then-expression and else-expression, verify that the types of the latter two are the same and the type of the condition is a boolean, and we are done.

The boolean expression is a function application and hence we apply the rule APPLY. In that case `not` must be a function and fortunately the environment tells us that this is the case. For the argument we apply the rule FALSE to discover that the argument to `not` is of the correct type `Bool`. The result of the application is again of type `Bool`, which is precisely what we need for the condition of the conditional.

The then-part is a lambda abstraction and we can proceed as follows. The function `*` is a binary operation on integers and we find that the type of `x` should be `Int` in order to be compatible. As a result, the type of the lambda abstraction is `Int` → `Int`.

The type of the function `fac`, as recorded in the environment, is `Int` → `Int` and the type of the conditional, which is also `Int` → `Int`, is well-defined. We are now done with the conditional and may conclude that the complete expression has type (`Int`, `Int` → `Int`).

The complete derivation tree can be constructed from Figure 7.1 and Figure 7.2. Note that expressions with an infix operator are transformed into expressions where the operator is written prefixed between parentheses. For example, $299\ *\ x$ is written $(*)\ 299\ x$.

Note that in most places we worked top-down through the expression, but in the case of the lambda abstraction we seem to reverse this. The reason is as follows: the type of `x` is not yet known when we start on the LAMBDA rule. During the actual process of inferencing as it will be programmed later on, we put `x` into the environment, but leave its type "open". This is actually what $\Gamma \cup \{(x : a)\}$ is supposed to indicate. When typing the body we can discover all kinds of information about `x` and record that information in the environment. This solution nicely corresponds to the way we shall handle polymorphism in our language.

7.5 Polymorphism

The type system of `SL` also allows polymorphism. In many modern languages and manuals, you will find references to polymorphism. It seems however that they do not always refer to the same concept.

In `Java` it is allowed to define the same function more than once as long as the signatures of the two functions differ. (The signature is the number, order and types of the arguments of the function. It does not include the return-type: two functions with the same arguments, but different return types are not allowed.) This kind of polymorphism was described by Strachey (see [26]) as *ad-hoc polymorphism* and currently often goes under the name of *overloading*: a function name is allowed to stand for a number of things, all of them potentially very differently coded.

An example of overloading that you have encountered many times is the kind where you may use an operator such as + for both the addition of integers as well as floating point values. Although the behaviour is conceptually the same, the code executed for adding integers is generally quite different from the code executed to add two floating point values. In most programming languages, if not all, this is allowed for the pragmatic reason that programmers are used to using these operators for both types.

Inheritance as present in `Java` is also sometimes referred to as polymorphism. In principle, one can define a function in a class and use it for any subclass as well. In that case, the behaviour of the function is identical to the behaviour for the original class. However, you are allowed to redefine the body of these inherited functions, making it possible to associate a totally new functionality with the same method. This is comparable to overloading using type classes in Haskell: although two functions for types of the same class have the same name, their workings may be very different.

With inheritance in `Java` it becomes possible to define functions that work for objects of more than a single class, but this is not a case of polymorphism, only one of *subtyping*, the idea being that if an operation works for a class $\mathcal{C}$, then it will also works for all subclasses, which should model special cases of $\mathcal{C}$.

From the course on Functioneel Programmeren and Grammatica's en Ontleden [18] you know that a different kind of polymorphism is present in `Haskell`. The crucial aspect here is that some code works for more than just a single type.

A particularly well-known example is the `map` function which does not need know much about the particular list it works on, or the function that it applies to each of the elements, as long as the function takes elements of the list as its argument. As a consequence the type of this function is written

$$(a \to b) \to [a] \to [b] \ .$$

Intuitively, this should be read as follows: for all types a and all types b, `map` is a function that takes a function from a to b and a list of as and returns a list of bs. In a way the type variable a stands for "all possible types". In `Haskell` you can qualify this further using *type classes*. In `SL` we shall not consider type classes.

In view of the above, we also want to allow type variables in our types. Hence we extend the definition of *type* to

- a basic type, i.e., `Int` or `Bool`, or
- a type variable $a \in \mathcal{V}$, or
- a composite type in which case it is either
  - a functional type $T_1 \to T_2$ for types T_1, T_2, or
  - a product type (T_1, T_2) for types T_1, T_2.

The set $\mathcal{V}$ denotes an infinite set of type variables, including $a, b, \ldots$.

For example, consider the type $T = (\texttt{Bool}, a) \to (a, \texttt{Bool})$, that intuitively maps a pair consisting of a boolean and something of a yet unknown type to a pair where these elements have been interchanged. The fact that we use a twice here indicates an equality of type between the second element of the argument and the first element of the result. Note that we could also have written $T = (\texttt{Bool}, b) \to (b, \texttt{Bool})$ to designate the same type, much in the same way that the name of a local variable in a function does not influence the outcome. The type T *is* different from $(\texttt{Bool}, b) \to (a, \texttt{Bool})$, because the equality of the type variables is now not enforced.

7.5.1 Substitution

Given the type $(a \to b) \to [a] \to [b]$ of `map` in `Haskell`, what is the type of `map fac`? Because `fac` has type $\texttt{Int} \to \texttt{Int}$, `map fac` does not map lists of as to lists of bs anymore, but lists of integers to lists of integers. In other words, the fact that we apply `map` to a parameter of a certain type may have consequences for the result type as well. To be able to cope with this we need substitutions and the process of unification.

For a type T we shall want to instantiate certain type variables with other types. For instance, a type variable a in T might be instantiated with `Int` or with a function such as $b \to b$.

A *substitution* σ is a (partial) function from the set of type variables $\mathcal{V}$ to the set of all types. The *domain* of σ, i.e., the values for which it is defined, we denote $\mathrm{dom}(\sigma)$.

Applying a substitution to a type T, yields a type $\langle T \rangle \sigma$ as follows:

$$\langle T \rangle \sigma = \begin{cases} T & \text{if } T \in \{\texttt{Bool}, \texttt{Int}\} \\ \sigma(T) & \text{if } (T \in \mathcal{V} \text{ and}) \ T \in \mathrm{dom}(\sigma) \\ T & \text{if } T \in \mathcal{V} \text{ and } T \notin \mathrm{dom}(\sigma) \\ \langle T_1 \rangle \sigma \to \langle T_2 \rangle \sigma & \text{if } T = T_1 \to T_2 \\ (\langle T_1 \rangle \sigma, \langle T_2 \rangle \sigma) & \text{if } T = (T_1, T_2) \end{cases}$$

In other words: the basic types are left unchanged, for a composite type we apply the substitution to the types of which it is built, and in case of a variable we apply the substitution. In the latter case there are two possibilities: either the variable a is in the domain of the substitution, and we return the image of the type variable $\sigma(a)$, or it is not, and then we simply return a. Note that the application of a substitution is defined in such a way that for variables a for which σ is undefined, we assume that $\sigma(a) = a$.

An example then: let σ be the substitution that maps a to $\texttt{Int} \to c$. For the type $T = (\texttt{Bool}, a) \to (b, \texttt{Bool}) \to a$, the result is $\langle T \rangle \sigma = (\texttt{Bool}, \texttt{Int} \to c) \to (b, \texttt{Bool}) \to \texttt{Int} \to c$. Note that b is left unchanged.

Given a substitution σ, the substitution $\sigma' = \sigma[b \leftarrow T]$ with $b \in \mathcal{V}$ and T a type, is such that

$$\sigma'(a) = \begin{cases} T & \text{if } a = b \\ \sigma(a) & \text{otherwise} \end{cases}$$

The idea is that σ' is very much like σ, except, possibly, for the image of b.

Note that you have to take care performing substitutions. For instance, if the substitution in the previous examples would map a to $\texttt{Int} \to b$, then the result would be $T = (\texttt{Bool}, \texttt{Int} \to b) \to (b, \texttt{Bool}) \to \texttt{Int} \to b$, which is different from $(\texttt{Bool}, \texttt{Int} \to c) \to (b, \texttt{Bool}) \to \texttt{Int} \to c$, because of the implicit equality of the bs in the former. In case you do not want these implicit bindings (and in most cases you do not), always make sure that you use "fresh" type variables in the types that form the image of the substitution.

For two types T and T', if there is a substitution σ such that $\langle T \rangle \sigma = T'$, then T' is an *instance* of T. For example, $(\texttt{Bool}, a)$ is an instance of b, by applying the substitution that simply maps b to $(\texttt{Bool}, a)$. The type $(\texttt{Bool}, \texttt{Int})$ is an instance of $(\texttt{Bool}, a)$, but $(\texttt{Int}, \texttt{Int})$ is not. Because we can combine substitutions by function composition, $(\texttt{Bool}, \texttt{Int})$ is also an instance of b.

7.5.2 Unification

In the process of type inferencing we might arrive at the following two facts: we have evidence that a function f has type $(\texttt{Bool}, a) \to (a, \texttt{Bool})$ and another piece of evidence tells us that this same f has type $b \to (\texttt{Int}, \texttt{Bool})$ Essentially, what we are after is the type that satisfies both these constraints. In other words, the type of f should be an instance of both the former and the latter type. The process of finding this type is called *unification*. There can be many such types however so which one do we want? Because we can always instantiate a more general type to a less general one, we always want the most general unifying type. The type system of our language is such that there is always a unique most general type.

In the example you can see that in both cases f is a function. The argument to the function has type $(\texttt{Bool}, a)$ and in the second case b. We can arrive from b at $(\texttt{Bool}, a)$ by performing a substitution $[b \leftarrow (\texttt{Bool}, a)]$. Now the arguments to the functions correspond, but we still have to see about the result type. These types are $(a, \texttt{Bool})$ and $(\texttt{Int}, \texttt{Bool})$, and the substitution is $[a \leftarrow \texttt{Int}]$. Combining these two we obtain $[b \leftarrow (\texttt{Bool}, a)][a \leftarrow \texttt{Int}]$. However, things are not so simple, because if we apply this substitution blindly, we will find $(\texttt{Bool}, \texttt{Int}) \to (\texttt{Int}, \texttt{Bool})$ and $(\texttt{Bool}, a) \to (\texttt{Int}, \texttt{Bool})$ and these types are not equal. What we want is that the a in $[b \leftarrow (\texttt{Bool}, a)]$ is also substituted by $\texttt{Int}$. Hence we need to unify the two substitutions obtaining the substition $[b \leftarrow (\texttt{Bool}, \texttt{Int})][a \leftarrow \texttt{Int}]$ and applying these would result in $(\texttt{Bool}, \texttt{Int}) \to (\texttt{Int}, \texttt{Bool})$ in both cases.

In the algorithm for unification we need the set of *free variables* in a type T. For now, this is simply the set of all variables in T and can be defined recursively as follows:

$$\mathrm{FV}(T) = \begin{cases} \{T\} & \text{if } T \in \mathcal{V} \\ \emptyset & \text{if } T \in \{\texttt{Int}, \texttt{Bool}\} \\ \mathrm{FV}(T_1) \cup \mathrm{FV}(T_2) & \text{if } T = T_1 \to T_2 \\ \mathrm{FV}(T_1) \cup \mathrm{FV}(T_2) & \text{if } T = (T_1, T_2) \end{cases}$$

We now describe the recursive process of unification. The parameters to the process are two types T_1 and T_2, the result is a substitution such that $\langle T_1 \rangle \sigma = \langle T_2 \rangle \sigma$, and if this substitution does not exist, a special symbol $\bot$ (called *bottom*) is returned. In the algorithm below $a, b \in \mathcal{V}$. The rules can be applied in any order, except for the last rule which should only be applied if no other rule applies.

$$\begin{array}{lcl} \mathrm{U}(a, a) & = & [] \\ \mathrm{U}(a, T) & = & [a \leftarrow T] \text{ if } a \notin \mathrm{FV}(T) \\ \mathrm{U}(T, b) & = & [b \leftarrow T] \text{ if } b \notin \mathrm{FV}(T) \\ \mathrm{U}(T_1 \to T_2, T_3 \to T_4) & = & \sigma_2 \cdot \sigma_1 \text{ where } \sigma_1 = \mathrm{U}(T_1, T_3),\ \sigma_2 = \mathrm{U}(\langle T_2 \rangle \sigma_1, \langle T_4 \rangle \sigma_1) \\ \mathrm{U}((T_1, T_2), (T_3, T_4)) & = & \sigma_2 \cdot \sigma_1 \text{ where } \sigma_1 = \mathrm{U}(T_1, T_3),\ \sigma_2 = \mathrm{U}(\langle T_2 \rangle \sigma_1, \langle T_4 \rangle \sigma_1) \\ \mathrm{U}(\texttt{Int}, \texttt{Int}) & = & [] \\ \mathrm{U}(\texttt{Bool}, \texttt{Bool}) & = & [] \\ \mathrm{U}(\_, \_) & = & \bot. \end{array}$$

Note that in the above $\cdot$ is ordinary function composition. The substitution $\bot$ indicates the impossibility to unify the two types. If, for example, we find that for a functional type, the unification of the arguments yields $\bot$, then we want the unification of the type as a whole to fail as well. To avoid cluttering up the definition of U, we define $\bot \cdot \sigma = \sigma \cdot \bot = \bot$ and $\langle T \rangle \bot = \bot$. This makes sure that after we encounter an error in the unification, whatever has happened or will happen, the end result will be $\bot$.

In the example leading up to the definition of unification, it was said that we sometimes need to unify substitutions. The definition of unification however handles this differently: in the rules for the composite types (and product types as well), we apply the substitution σ_1 to T_2 and T_4 and unify the outcomes. The reason is that in σ_1 we might have found a type to be substituted for a type variable a, and if a is also present in T_2 or T_4, then this has to be taken into account. The

```
let i = \x -> x
 in i i
 ni
```

Figure 7.3: A case for quantifying type variables

advantage of the current definition is that we do not need to unify the substitutions anymore. It does imply a left to right order in the process of unification.

In the rule for $\mathrm{U}(a, T)$ there is the condition that $a \notin \mathrm{FV}(T)$, i.e., a is a type variable not present in T. For example if $T = (a, b)$, then it should not be surprising that unification is not possible. To unify a with (a, b) we would have to substitute a pair for a, say (c, d), but then this also has to be done to a in (a, b), yielding a pair $((c, d), b)$. But then we have to substitute a pair, say (e, f) for c in (c, d), and similarly we should do this for $((c, d), b)$. This process can continue forever. In these cases, `Haskell` would give an error message stating that in this way you obtain an infinite type. There are type systems where this situation can be handled, but in our language this is not possible.

An example where unification fails is $\mathrm{U}(\texttt{Int} \to a, a \to \texttt{Bool})$. As a final example, it is not always necessary that the result of unification is a type without type variables: the unification of $(a, (a, b))$ and $(c, (d, c))$ is as you may verify $(d, (d, d))$.

7.5.3 Type Rules For Polymorphism

Consider Figure 7.3. As we shall later show, the expression in the figure has type $a \to a$, but what exactly is the type of i? Looking at the definition of i in the let-expression, a first guess is that it is $a \to a$. But notice then what happens to the type of the body. The first i takes an argument of type $a \to a$ and hence should have type $(a \to a) \to (a \to a)$. Now some might argue that we have an infinite type here: because the i which is applied has type $(a \to a) \to (a \to a)$, the argument also must have that type, which will complicate the type for the argument i. However, this is not the case. For this we have to examine what a let really means: the code given above can be unfolded into $(\backslash x \to x)\ (\backslash x \to x)$.

It can be seen that we may type both function and argument separately, obtaining $b \to b$ and $a \to a$ for the two applications, respectively. Unification then tells us to substitute $a \to a$ for b in the former and the result type of the expression is $a \to a$, as indicated earlier.

Instantiation The crucial realization here is that the two is are really different identifiers, although when seen in the definition part of a let-expression they may have seemed the same variable.

In other words, the body of the let has two different *instantiations* of the i defined in the definition part of the let. In both cases i must be a function mapping a to a, but the instances substituted for a may differ.

The fact that a type variable a may be instantiated later is administrated by adding $\forall a$ in front of the type expression. In our language, the $\forall$ is used purely for administration. The programmer should never see any types like that in the error message he obtains or, even if he explicitly writes down types in his program, in his program.

Before continuing with the necessary type rules, we first incorporate $\forall$ into our type language.

- a basic type, i.e., `Int` or `Bool`, or
- a type variable $a \in \mathcal{V}$, or
- a composite type in which case it is either

```
\f -> let x = f 3
      in (x, f)
      ni
```

Figure 7.4: A case for not always quantifying type variables

- a functional type $T_1 \to T_2$ for types T_1, T_2, or
- a product type (T_1, T_2) for types T_1, T_2, or
- a quantified type $\forall a.T_1$ for a type T_1.

Please note that in our types we shall only see $\forall$ occurring at top level, even if we have defined type more generally.

The definition of *free variables* should now be refined to cope with the extra case:

$$\mathrm{FV}(T) = \begin{cases} \{T\} & \text{if } T \in \mathcal{V} \\ \emptyset & \text{if } T \in \{\texttt{Int}, \texttt{Bool}\} \\ \mathrm{FV}(T_1) \cup \mathrm{FV}(T_2) & \text{if } T = T_1 \to T_2 \\ \mathrm{FV}(T_1) \cup \mathrm{FV}(T_2) & \text{if } T = (T_1, T_2) \\ \mathrm{FV}(T_1) - \{a\} & \text{if } T = (\forall a.T_1) \end{cases}$$

The only difference is that quantified type variables are not free.

The definition of substitution and unification can remain the same, because we shall never unify quantified types: we shall instantiate the quantified type variables beforehand.

Generalization The introduction of $\forall$ is handled by the following derivation rule:

$$\textsc{Gen} : \frac{\Gamma \vdash \texttt{expr} : t}{\Gamma \vdash \texttt{expr} : \forall a.t} a \notin \mathrm{FV}(\Gamma)$$

where $\mathrm{FV}(\Gamma) = \{\mathrm{FV}(t) \mid (x : t) \in \Gamma\}$ is the set of type variables occurring free in Γ. For example, for the environment

$$\Gamma = \{(x : \forall a.a \to a), (y : b), (z : (\texttt{Int}, c))\} \ ,$$

we have $\mathrm{FV}(\Gamma) = \{b, c\}$. The type variable a is not included, because it occurs bound in the type for x.

For the code of Figure 7.3, the result would be that i as defined in the let expression has type $\forall a.a \to a$. This is allowed, because there is no (restricting) information about x in Γ. The function i in the definition of the `let` is said to be *polymorphic* in the type variable a.

The restriction $a \notin \mathrm{FV}(\Gamma)$ shows that we can not blindly add $\forall$s, because not all type variables are polymorphic; some are *monomorphic*.

Generalization in the context of a let expression Consider the code fragment in Figure 7.4. If we unfold the let in this case, then the f in the body and the f in the definition for x become bound to the same f in the lambda abstraction. So in this case they must have the same type. This explains why we can not simply go around quantifying every type variable that we find in the definition part of a let expression. You might wonder how we can discover which type variable we have to quantify and which we do not. Intuitively, type variables in types that are already in the environment before we start on the let may not be quantified. As an aside, note that the code `x = f 3` also tells us something about f: it has type $\texttt{Int} \to b$ (or an instance thereof). This information will be recorded in the substitution returned by the unification process.

The rule for a let expression (that may be recursive) now reads:

$$\textsc{Let} : \frac{(\Gamma - \texttt{id}) \cup (\texttt{id} : t) \vdash \texttt{expr} : t, (\Gamma - \texttt{id}) \cup (\texttt{id} : \texttt{gen}_\Gamma(t)) \vdash \texttt{body} : b}{\Gamma \vdash \textbf{let } \texttt{id } = \texttt{ expr } \textbf{in } \texttt{body } \textbf{ni} : b}$$

$$\dfrac{\dfrac{\dfrac{(\mathsf{x} : v_2) \in \Gamma_3}{\Gamma_3 \vdash \mathsf{x} : v_2}\ \textsc{Ident}}{\Gamma_2 \vdash \lambda \mathsf{x} \to \mathsf{x} : v_2 \to v_2}\ \textsc{Lambda} \qquad \dfrac{\dfrac{\dfrac{(\mathsf{i} : \forall a\ .\ a \to a) \in \Gamma_4}{\Gamma_4 \vdash \mathsf{i} : \forall a\ .\ a \to a}\ \textsc{Ident}}{\Gamma_4 \vdash \mathsf{i} : (v_5 \to v_5) \to v_5 \to v_5}\ \textsc{Spec} \qquad \dfrac{\dfrac{(\mathsf{i} : \forall a\ .\ a \to a) \in \Gamma_4}{\Gamma_4 \vdash \mathsf{i} : \forall a\ .\ a \to a}\ \textsc{Ident}}{\Gamma_4 \vdash \mathsf{i} : v_5 \to v_5}\ \textsc{Spec}}{\Gamma_4 \vdash \mathsf{i}\ \mathsf{i} : v_5 \to v_5}\ \textsc{Apply}}{\Gamma_1 \vdash \textbf{let}\ \mathsf{i} = \lambda \mathsf{x} \to \mathsf{x}\ \textbf{in}\ \mathsf{i}\ \mathsf{i}\ \textbf{ni} : v_5 \to v_5}\ \textsc{Let}$$

$$\Gamma_1 \vdash \textbf{let}\ \mathsf{i} = \lambda \mathsf{x} \to \mathsf{x}\ \textbf{in}\ \mathsf{i}\ \mathsf{i}\ \textbf{ni} : \forall a\ .\ a \to a\ \textsc{Gen}$$

where $\Gamma_1 = \emptyset$, $\Gamma_2 = \{(i : v_2 \to v_2)\}$ $\Gamma_3 = \{(x : v_2),\ (i : v_2 \to v_2)\}$ and $\Gamma_4 = \{(i : \forall a\ .\ a \to a)\}$.

Figure 7.5: The derivation tree for the code of Figure 7.3.

where $\texttt{gen}_\Gamma(t)$ is the type that results by quantifying over all type variables in t that do not occur free in Γ.

Note that like in the case of the lambda rule we have to remove all references to `id` that happen to occur in Γ already, because these are now shadowed by the new `id`.

Another scope rule can be deduced from the definition of LET. The scope of the identifier defined in a let expression is both the definition itself (to allow recursive definitions) and the body of the let. This is made evident by the fact that $(\texttt{id} : t)$ is added to Γ for the type checking of both the expression `expr` and the body `body`.

There is also an important difference between the case of the expression and the case of the body when adding `id` to the environment Γ. In the case of the body, as the example of Figure 7.3 shows, we want `id` to have possibly different instantiations. In the case of the expression, we do want to allow that `id` is used recursively in the expression, but not with possibly different instantiations. This can be explained again by unfolding the let.

A type which has a number of quantified type variables has to be instantiated to a type which does not have the quantifier, before we can apply unification and the like.

$$\textsc{Spec} : \frac{\Gamma \vdash \mathsf{expr} : \forall a.t}{\Gamma \vdash \texttt{expr} : \langle t \rangle [a \leftarrow b]} b \notin \mathrm{FV}(\Gamma)$$

The only thing we have to take into account is that the introduction of the new type variable does not clash with other type variables, which is why we demand that b is not a free type variable.

In the derivation tree in Figure 7.5, you see the process of generalizing and instantiating at work for the code in Figure 7.3. In the let definition part we add $\forall a$ to the type found for the body of i, and in the body we instantiate the quantified type twice.

Note the differences with the derivation tree for the code of Figure 7.4 as can be seen in Figure 7.6. The crucial difference lies in the fact that f in the environment Γ_4 (passed to the body of the let) is not quantified over v_3.

Also note that the derivation tree in Figure 7.6 as well as Figure 7.5 contains a GEN step at the root (bottom of figure). The necessity of this step depends on the context the derivation is placed in. If the derivation tree belongs to a definition in a let expression, the generalization step is done as part of this definition. However, if the expression is -for example- defined at the toplevel of a Haskell program without a surrounding let expression, the generalization has to be done explicitly by using a GEN step as done in Figure 7.6 and Figure 7.5.

As a final example we give a piece of code where an identifier `fac` is defined recursively.

```
let fac = \x -> if x == 0 then 1 else x * fac (x - 1)
 in fac 6
 ni
```

In Figure 7.7 and Figure 7.8 the derivation tree is given.

$$\dfrac{\dfrac{\dfrac{\dfrac{(\mathsf{f} : Int \to v_3) \in \Gamma_3}{\Gamma_3 \vdash \mathsf{f} : Int \to v_3}\ \textsc{Ident} \quad \dfrac{}{\Gamma_3 \vdash 3 : Int}\ \textsc{Integer}}{\Gamma_3 \vdash \mathsf{f}\ 3 : v_3}\ \textsc{Apply} \quad \dfrac{\dfrac{(\mathsf{x} : v_3) \in \Gamma_4}{\Gamma_4 \vdash \mathsf{x} : v_3}\ \textsc{Ident} \quad \dfrac{(\mathsf{f} : Int \to v_3) \in \Gamma_4}{\Gamma_4 \vdash \mathsf{f} : Int \to v_3}\ \textsc{Ident}}{\Gamma_4 \vdash (\mathsf{x}, \mathsf{f}) : (v_3, Int \to v_3)}\ \textsc{Product}}{\Gamma_2 \vdash \textbf{let}\ \mathsf{x} = \mathsf{f}\ 3\ \textbf{in}\ (\mathsf{x}, \mathsf{f})\ \textbf{ni} : (v_3, Int \to v_3)}\ \textsc{Let}}{\Gamma_1 \vdash \lambda \mathsf{f} \to \textbf{let}\ \mathsf{x} = \mathsf{f}\ 3\ \textbf{in}\ (\mathsf{x}, \mathsf{f})\ \textbf{ni} : (Int \to v_3) \to (v_3, Int \to v_3)}\ \textsc{Lambda}}{\Gamma_1 \vdash \lambda \mathsf{f} \to \textbf{let}\ \mathsf{x} = \mathsf{f}\ 3\ \textbf{in}\ (\mathsf{x}, \mathsf{f})\ \textbf{ni} : \forall a\ .\ (Int \to a) \to (a, Int \to a)}\ \textsc{Gen}$$

where $\Gamma_1 = \emptyset$, $\Gamma_2 = \{(f : \texttt{Int} \to v_3)\}$ and $\Gamma_3 = \Gamma_4 = \{(x : v_3),\ (f : \texttt{Int} \to v_3)\}$.

Figure 7.6: The derivation tree for the code of Figure 7.4.

$$\dfrac{\dfrac{\dfrac{(* : Int \to Int \to Int) \in \Gamma_3}{\Gamma_3 \vdash * : Int \to Int \to Int}\ \textsc{Ident} \quad \dfrac{(\mathsf{x} : Int) \in \Gamma_3}{\Gamma_3 \vdash \mathsf{x} : Int}\ \textsc{Ident}}{\Gamma_3 \vdash (*)\ \mathsf{x} : Int \to Int}\ \textsc{Apply} \quad \dfrac{(\mathsf{fac} : Int \to Int) \in \Gamma_3}{\Gamma_3 \vdash \mathsf{fac} : Int \to Int}\ \textsc{Ident} \quad \dfrac{\dfrac{\dfrac{(- : Int \to Int \to Int) \in \Gamma_3}{\Gamma_3 \vdash - : Int \to Int \to Int}\ \textsc{Ident} \quad \dfrac{(\mathsf{x} : Int) \in \Gamma_3}{\Gamma_3 \vdash \mathsf{x} : Int}\ \textsc{Ident}}{\Gamma_3 \vdash (-)\ \mathsf{x} : Int \to Int}\ \textsc{Apply} \quad \dfrac{}{\Gamma_3 \vdash 1 : Int}\ \textsc{Integer}}{\Gamma_3 \vdash (-)\ \mathsf{x}\ 1 : Int}\ \textsc{Apply}}{\Gamma_3 \vdash \mathsf{fac}\ ((-)\ \mathsf{x}\ 1) : Int}\ \textsc{Apply}}{\Gamma_3 \vdash (*)\ \mathsf{x}\ \mathsf{fac}\ ((-)\ \mathsf{x}\ 1) : Int}\ \textsc{Apply}$$

where $\Gamma_3 = \{(x : \texttt{Int}), (\texttt{fac} : \texttt{Int} \to \texttt{Int}), (==: \texttt{Int} \to \texttt{Int} \to \texttt{Bool}), (- : \texttt{Int} \to \texttt{Int} \to \texttt{Int}), (* : \texttt{Int} \to \texttt{Int} \to \texttt{Int})\}$.

Figure 7.7: The derivation tree for the expression $x * \texttt{fac}(x - 1)$.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{(== : Int \to Int \to Bool) \in \Gamma_3}{\Gamma_3 \vdash == : Int \to Int \to Bool}\ \textsc{Ident} \quad \dfrac{(\mathsf{x} : Int) \in \Gamma_3}{\Gamma_3 \vdash \mathsf{x} : Int}\ \textsc{Ident}}{\Gamma_3 \vdash (==)\ \mathsf{x} : Int \to Bool}\ \textsc{Apply} \quad \dfrac{}{\Gamma_3 \vdash 0 : Int}\ \textsc{Integer}}{\Gamma_3 \vdash (==)\ \mathsf{x}\ 0 : Bool}\ \textsc{Apply} \quad \dfrac{}{\Gamma_3 \vdash 1 : Int}\ \textsc{Integer} \quad \boxed{\text{Figure 7.7}}}{\Gamma_3 \vdash \textbf{if}\ (==)\ \mathsf{x}\ 0\ \textbf{then}\ 1\ \textbf{else}\ (*)\ \mathsf{x}\ \mathsf{fac}\ ((-)\ \mathsf{x}\ 1)\ \textbf{fi} : Int}\ \textsc{Cond} \quad \dfrac{(\mathsf{fac} : Int \to Int) \in \Gamma_4}{\Gamma_4 \vdash \mathsf{fac} : Int \to Int}\ \textsc{Ident} \quad \dfrac{}{\Gamma_4 \vdash 6 : Int}\ \textsc{Integer}}{\Gamma_2 \vdash \lambda x \to \textbf{if}\ (==)\ \mathsf{x}\ 0\ \textbf{then}\ 1\ \textbf{else}\ (*)\ \mathsf{x}\ \mathsf{fac}\ ((-)\ \mathsf{x}\ 1)\ \textbf{fi} : Int \to Int}\ \textsc{Lambda} \quad \dfrac{}{\Gamma_4 \vdash \mathsf{fac}\ 6 : Int}\ \textsc{Apply}}{\Gamma_1 \vdash \textbf{let}\ \mathsf{fac} = \lambda x \to \textbf{if}\ (==)\ \mathsf{x}\ 0\ \textbf{then}\ 1\ \textbf{else}\ (*)\ \mathsf{x}\ \mathsf{fac}\ ((-)\ \mathsf{x}\ 1)\ \textbf{fi}\ \textbf{in}\ \mathsf{fac}\ 6\ \textbf{ni} : Int}\ \textsc{Let}$$

where $\Gamma_1 = \{(==: \texttt{Int} \to \texttt{Int} \to \texttt{Bool}), (- : \texttt{Int} \to \texttt{Int} \to \texttt{Int}), (* : \texttt{Int} \to \texttt{Int} \to \texttt{Int})\}$,
$\Gamma_2 = \Gamma_4 = \Gamma_1 \cup \{(\texttt{fac} : \texttt{Int} \to \texttt{Int})\}$ and $\Gamma_3 = \Gamma_2 \cup \{(x : \texttt{Int})\}$.

Figure 7.8: A recursive let-definition

7.6 Attribute grammar for type inferencing SL compiler (SLti)

The type inferencing version of the SL compiler, named *SLti*. Parsing and pretty printing is the same as in the SLbb version, Chapter 5.

7.6.1 Main

```
import SLParser
import UU.Pretty
import UU.Parsing
import SLTypes
import SLAttributes

slVersion
  = "Type Inferencing"
slDate
  = "20031106"
slIntroducedFeatures
  = "Type Inferencing"
slSignOn =
  " ____    _" ++ "\n" ++
  "/ ___| | |" ++ "\tSL (Simple Language) Compiler.\n" ++
  "\\___ \\" ++ " | | " ++ "\tVersion " ++ slVersion ++ "/" ++ slDate ++ "\n" ++
  " ___) || |__  " ++ "\tIntroducedFeatures: " ++ "\n" ++
  "|____/ |_____| " ++ "\t\t" ++ slIntroducedFeatures ++ "\n"

compile filename = do
  tokens ← slScan filename
  (exprpp, tp) ← parseIO (pRoot ()) tokens
  putStrLn ("The expression:\n" ++ (disp exprpp 300 ""))
  putStrLn ("Is inferred to have type " ++ show tp)
  putStrLn "-------------------------------------------------------"

main = do
  putStr slSignOn
  putStr "Enter filename: "
  filename ← getLine
  compile filename
```

7.6.2 Type structure

```
module SLTypes where
import UU.Pretty
import Char
data Type =
  TAny
  | TBool
  | TInt
  | TUnit
  | TFunc Type Type
  | TForall Int Type
  | TBound VarName
  | TVar VarName
```

```
  | TError String deriving Eq
type VarName = Int
instance Show Type where                                                    43
  show (TAny)       = "ANYTYPE"
  show (TBool)      = "Bool"
  show (TInt)       = "Int"
  show (TUnit)      = "()"
  show (TFunc f a)  = "(" ++ show f ++ ")" ++ " -> " ++ show a              48
  show (TForall i ts) = show ts
  show (TBound i)   = [chr $ i + (ord 'a')]
  show (TVar v)     = "v_" ++ show v
  show (TError e)   = "<!ERROR: " ++ e ++ " !>"
type Types = [Type]                                                         53
type Ids = [String]
  -- basic types
booltype = TBool
inttype  = TInt
unittype = TUnit                                                            58
anytype  = TAny
arrow    = TFunc
  -- types for in the initial gamma
bool2bool2bool = booltype `arrow` (booltype `arrow` booltype)
int2int2bool   = inttype `arrow` (inttype `arrow` booltype)                 63
int2int2int    = inttype `arrow` (inttype `arrow` inttype)
makeFnType f x = TFunc f x
getErrorMsg :: Type -> (Type, PP_Doc)
getErrorMsg (TError s) = (TAny, ppErr (text s))
getErrorMsg t          = (t, empty)                                         68
ppErr msg = text "<!ERROR:" > # < msg > | < text "!>"
getResultType (TFunc a b)  = b
getResultType (TError msg) = TError msg
getResultType (TAny)       = TAny
getResultType _            = TError "not a function"                        73
getArgType (TFunc a b)  = a
getArgType (TError msg) = TError msg
getArgType (TAny)       = TAny
getArgType _            = TError "not a function"
getOpErrors (TError s, TError t) = (ppErr (text s), ppErr (text t), True)   78
getOpErrors (TError s, _)        = (ppErr (text s), empty, True)
getOpErrors (_, TError s)        = (empty, ppErr (text s), True)
getOpErrors (TAny, _)            = (empty, empty, True)
getOpErrors (_, TAny)            = (empty, empty, True)
getOpErrors _                    = (empty, empty, False)                    83
isAnnotated (TAny) = False
isAnnotated _      = True
```

7.6.3 Unification

```
module SLUnification where
```

```
import SLTypes
import UU.Pretty                                                          88

type Subst = [(VarName, Type)]
emptysubst = []
(|⇒) :: Subst → Type → Type
(|⇒) subst t = case t of
  (TAny)          → TAny                                                  93
  (TInt)          → TInt
  (TBool)         → TBool
  (TUnit)         → TUnit
  (TFunc a b)     → TFunc (subst |⇒ a) (subst |⇒ b)
  (TForall i t')  → TForall i (subst |⇒ t')                                98
  (TBound _)      → t
  (TVar tvar)     → case lookup tvar subst of
                      Just t'   → t'
                      Nothing → t
  (TError s)      → TError s                                              103
(|||) :: Subst → Subst → Subst
(|||) s1 s2 = s1 ++ [(v, s1 |⇒ t) | (v, t) ← s2]
unify :: Type → Type → (Subst, Type)
unify TAny       _          = ([], TAny)
unify _          TAny       = ([], TAny)                                 108
unify (TError s) _          = ([], TError s)
unify _          (TError s) = ([], TError s)
unify (TVar l)   (TVar r)   = if l ≡ r
                                then ([], TVar r)
                                else ([(l, TVar r)], TVar r)             113
unify (TVar l)   t          = varBind l t
unify t          (TVar r)   = varBind r t
unify TBool      TBool      = ([], TBool)
unify TBool      t          = ([], TError ("Bool /= " ++ show t))
unify t          TBool      = ([], TError ("Bool /= " ++ show t))        118
unify TInt       TInt       = ([], TInt)
unify t          TInt       = ([], TError ("Int /= " ++ show t))
unify TInt       t          = ([], TError ("Int /= " ++ show t))
unify TUnit      TUnit      = ([], TUnit)
unify t          TUnit      = ([], TError ("() /= " ++ show t))          123
unify TUnit      t          = ([], TError ("() /= " ++ show t))
unify (TFunc a b) (TFunc a' b') = let (aSubst, t1) = unify a a'
                                      (bSubst, t2) = unify (aSubst |⇒ b) (aSubst |⇒ b')
                                  in (bSubst ||| aSubst, TFunc t1 t2)
unify (TFunc a b) t         = ([], TError (show (TFunc a b) ++ " /= " ++ show t))  128
unify t          (TFunc a b) = ([], TError (show (TFunc a b) ++ " /= " ++ show t))
unify t1         t2         = sysError ("Invalid types " ++ show t1
                                ++ " and " ++ show t2 ++ " in unification.")

unifyMsg t1 t2 = let (subst, ttp) = unify t1 t2
                     (tp, msg)    = getErrorMsg ttp                      133
                 in (subst, tp, msg)

varBind :: VarName → Type → (Subst, Type)
varBind v t = if v `occurs` t
  then ([], TError "Occurence check failed")
  else ([(v, t)], t)                                                     138
```

```
occurs v t = case t of
  TVar v'    → v ≡ v'
  TBound _   → sysError "Occurence check on universally quantified variable."
  TForall i t → occurs v t
  TAny       → False                                                    143
  TInt       → False
  TBool      → False
  TUnit      → False
  TFunc a b  → occurs v a ∨ occurs v b
unifyPossiblyAnnoted (TAny) t  = (emptysubst, t, empty)                 148
unifyPossiblyAnnoted t      t' = unifyMsg t t'
sysError s = error ("System error:" ++ s)
```

7.6.4 Top level AG (combining all aspects)

```
imports
{
import SLTypes                                                          153
import Prelude
}
DATA Root
  | Root   Expr
DATA Expr                                                               158
  | Unit
  | Intexpr  Int
  | Boolexpr Bool
  | Ident    var : String
  | Op       op : String le, re : Expr                                  163
  | If       cond, thenExpr, elseExpr : Expr
  | Let      decls : Decls expr : Expr
  | Assign   var : String expr : Expr
  | Apply    func, arg : Expr
  | Lamcall  call : Expr                                                168
  | Lam      vars : Vars expr : Expr
  | Seq      exprs : Exprs
DATA Decl
  | Decl     var : String type : Type expr : Expr
TYPE Decls = [Decl]                                                     173
TYPE Exprs = [Expr]
TYPE Vars = [String]
INCLUDE "SLPrettyprint.ag"
INCLUDE "SLTypeInference.ag"
```

7.6.5 Type inferencing

```
imports                                                                 178
{
import SLUnification
import SLTypes
```

```
}
{                                                                          183
type Assumptions = [(String, Type)]
}
ATTR Root [ | | tp : Type]
SEM Root
  | Root  expr.gamma  = initgamma                                          188
          expr.unique = 1
          expr.subst  = emptysubst
          lhs.tp      = abstract initgamma@expr.tp
ATTR Expr [gamma : Assumptions | unique : Int subst : Subst | tp : Type]
SEM Expr                                                                   193
  | Unit     lhs.tp                          = unittype
  | Intexpr  lhs.tp                          = inttype
  | Boolexpr lhs.tp                          = booltype
  | Ident    loc.(tp, unq)                   = instantiate @lhs.unique
                                                 (@lhs.subst |⇒                198
                                                   (lookupVar @var @lhs.gamma))
             loc.(tp', errmsg)               = getErrorMsg @tp
             lhs.tp                          = @tp'
             lhs.unique                      = @unq
  | Op       loc.opType                      = lookupVar @op @lhs.gamma        203
             lhs.unique                      = @lhs.unique + 1
             loc.(lSubs, leType)             = unify (getArgType @opType) (@le.tp)
             loc.(rSubs, reType)             = unify (getArgType (getResultType @opType))
                                                 (@re.tp)
             loc.(lerr, rerr, areErrs)       = getOpErrors (@leType, @reType)  208
             lhs.tp                          = if @areErrs
                                               then anytype
                                               else getResultType (getResultType @opType)
             lhs.subst                       = @lSubs ||| @rSubs ||| @re.subst
  | If       loc.(condSubst, condType, condErr) = unifyMsg booltype @cond.tp  213
             thenExpr.subst                  = @condSubst ||| @cond.subst
             loc.(resSubst, resType, ifErr)  = unifyMsg (@elseExpr.subst |⇒ @thenExpr.tp)
                                                 @elseExpr.tp
             lhs.subst                       = @resSubst ||| @elseExpr.subst
             lhs.tp                          = @resType                        218
  | Let      lhs.tp                          = @expr.tp
  | Assign   lhs.tp                          = unittype
             loc.(exprTp, exprErr)           = getErrorMsg @expr.tp
             loc.(newSubs, _, varErr)        = unifyMsg (lookupVar @var @lhs.gamma)
                                                 (@expr.tp)                    223
             lhs.subst                       = @newSubs ||| @expr.subst
  | Apply    func.unique                     = @lhs.unique + 1
             loc.(newSubs, funcType, argErr) = unifyMsg (@arg.subst|⇒ @func.tp)
                                                 (@arg.tp `arrow` (TVar @lhs.unique))
             loc.(tp, funcErr)               = getErrorMsg (getResultType @funcType) 228
             lhs.tp                          = @tp
             lhs.subst                       = @newSubs ||| @arg.subst
  | Lam      loc.argTypes                    = map (@expr.subst|⇒) @vars.tps
             lhs.tp                          = foldr arrow @expr.tp @argTypes
             loc.bodyErr                     = empty                           233
             loc.argErr                      = empty
```

```
  | Seq     lhs.tp                                = if null @exprs.tps
                                                    then anytype
                                                    else last @exprs.tps
ATTR Exprs [gamma : Assumptions | subst : Subst unique : Int | tps : Types]
SEM Exprs
  | Cons    lhs.tps                               = @hd.tp : @tl.tps
  | Nil     lhs.tps                               = []
ATTR Decls Decl [ | gamma : Assumptions subst : Subst unique : Int | ]
SEM Decl
  | Decl    expr.unique                           = @lhs.unique + 1
            loc.varTp                             = TVar @lhs.unique
            expr.gamma                            = (@var, @varTp) : @lhs.gamma
            loc.(sub, declTp, derr)               = unifyMsg @varTp @expr.tp
            loc.(annoteSub, annoteTp, declErr)    = unifyPossiblyAnnoted @type @declTp
            loc.subs                              = @annoteSub||(@sub|| @expr.subst)
            loc.declType                          = abstract
                                                    (mapSnd (@subs |⇒) @lhs.gamma)
                                                    @annoteTp
            lhs.subst                             = @subs
            lhs.gamma                             = (@var, @declType) : @lhs.gamma
            loc.inferredType                      = (text.show) @declType
ATTR Vars [ | gamma : Assumptions unique : Int | tps : Types]
SEM Vars
  | Cons tl.gamma                                 = (@hd, TVar@lhs.unique) : @lhs.gamma
         tl.unique                                = @lhs.unique + 1
         lhs.tps                                  = TVar @lhs.unique : @tl.tps
  | Nil  lhs.tps                                  = []

{
initgamma =
  [("&&", bool2bool2bool),
   ("||", bool2bool2bool),
   ("==", int2int2bool),
   ("/=", int2int2bool),
   ("<", int2int2bool),
   (">", int2int2bool),
   ("<=", int2int2bool),
   (">=", int2int2bool),
   ("+", int2int2int),
   ("-", int2int2int),
   ("*", int2int2int),
   ("/", int2int2int)]
lookupVar x [] = TError "Variable not introduced"
lookupVar x ((v, t) : vs) = if x ≡ v then t else lookupVar x vs
instantiate unique (TForall i t) =
  let recurse t = case t of
    TBound x   → TVar (unique + x)
    TFunc a b  → TFunc (recurse a) (recurse b)
    TForall _ _ → sysError "Non-flat type in instantiation."
    _          → t
  in (recurse t, unique + i)
instantiate unique t = (t, unique)
abstract env tp =
```

```
  let (names, resType) = abs tp []
      abs t ns  = case t of                                                    288
                    TVar v      -> case lookup v ns of
                                     Just i    -> (ns, TBound i)
                                     Nothing -> if v `notFreeIn` env
                                                  then (ns, TVar v)
                                                  else  let newPos = length ns   293
                                                        in (ns ++ [(v, newPos)], TBound newPos)
                    TFunc a b   -> let (nsA, tpA) = abs a ns
                                       (nsB, tpB) = abs b nsA
                                   in (nsB, TFunc tpA tpB)
                    TInt        -> (ns, t)                                   298
                    TBool       -> (ns, t)
                    TUnit       -> (ns, t)
                    TAny        -> (ns, t)
                    TError _    -> (ns, t)
                    TForall _ _ -> sysError "Non-flat type in type abstraction."   303
                    TBound _    -> sysError "In type abstraction."
  in if null names then tp else TForall (length names) resType
notFreeIn var env = or (map (notFreeInType.snd) env)
  where notFreeInType t = case t of
    TVar v      -> v ≡ var                                                   308
    TFunc a b   -> notFreeInType a ∨ notFreeInType b
    TForall i tp -> notFreeInType tp
    _           -> False
mapSnd f xys = [(x, f y) | (x, y) ← xys]
}                                                                            313
```

7.7 Exercises

7-1 **Mutually recursive definitions**. In the language "SLti" it is possible to write recursive functions, but not mutually recursive functions. Change the compiler so that this becomes possible as well.

7-2 **Unification**. The result of the unification of two types is a substitution. For the substitution, the following property holds: if the substitution is applied to the two types, the resulting two types are equal. In other words:

$$\text{if } S = \mathit{unify}(x, y) \text{ then } S(a) = S(b)$$

Compute the unifying substitution $\mathit{unify}(x, y)$ for the following combinations of types x and y:

	x		y
1.	$\mathit{Int} \to \mathit{Int}$	and	$a \to b$
2.	$\mathit{Int} \to \mathit{Int} \to \mathit{Int}$	and	$a \to b$
3.	$a \to a$	and	$(\mathit{Bool} \to b) \to c$
4.	$a \to a$	and	$\mathit{Bool} \to b \to c$
5.	$a \to b$	and	$\mathit{Int} \to a$
6.	a	and	$a \to a$
7.	(a, Int)	and	(Bool, b)
8.	$(a, a \to \mathit{Bool})$	and	$(b \to \mathit{Bool}, \mathit{Int} \to \mathit{Bool} \to \mathit{Bool})$

Remark: The application of a substitution to a type $S(a)$ is denoted by `S |=> a` in the attribute grammar for SL.

7-3 **Type Inferencing**. Apply the type inference algorithm from the lectures to the following expressions. Use the following assumptions in Γ:

- $(*) :: Int \to Int \to Int$
- $(-) :: Int \to Int \to Int$
- $(==) :: Int \to Int \to Bool$

1.
```
\f -> \x -> f x
```

2.
```
\f -> \x -> if f 3
            then True
            else x
            fi
```

3.
```
let id = \x -> x
in id id 3
ni
```

4.
```
let fac = \n -> if n == 0
                  then 1
                  else n * fac (n - 1)
in fac
ni
```

5. Type inference is not possible for the following expression. Why not?
```
\x -> let (a,b) = x
      in a * b
      ni
```

7-4 **Pretty Printing Polymorphic Types**. There is a convention for languages with polymorphic type systems that quantifiers do not have to be written explicitly. The compiler infers the position where the quantifiers have to be inserted (if you ever work with the language Prolog, you will recognize this convention there as well). For instance, the Haskell (or ML) program fragment `a -> b -> (a,b)` stands in fact for the type $\forall ab.a \to b \to (a, b)$. The reason for this choice is that in most programming languages the types are inferred in a way that it is not allowed for types to contain nested quantifiers or even quantifiers that are not on the outside. If we just typecheck our programs (and do not perform type inference), there is nothing to say against allowing quantifiers everywhere. In this case the following interpretation of the program fragment above might be more adequate: $\forall a.a \to (\forall b.b \to (a, b))$. Perform the following steps:

1. Design a suitable datatype in which such types can be represented.
2. Write a parser that recognizes program fragments such as the one given above.
3. Make sure that the result of this parser is a type in which all quantifiers are made explicit, and in such a way that all quantifiers are pushed as far to the inside as possible.
4. Extend your system so that cartesian products can be recognized as well. If a type variable occurs in both components of a cartesian product, but not anywhere on the outside, then the correct type-theoretic interpretation is that this type variable is to be *existentially* quantified immediately before the cartesian product. Thus, extend the datatype so that you can represent existential quantifiers as well. The type `c -> (a,(a -> b -> (b,c)))` must be read as $\forall c.c \to (\exists a.(a, a \to \forall b.b \to (b, c)))$.
5. Give a function that is of this example type.

7-5 **Type rules (from Exam 20001023)**.

1. Give a type proof for

```
let compose = \ f -> \g -> \ x -> (f (g x))
 in compose (+3)
 ni
```

   Assume an environment Γ in which `+` has type `Int -> Int -> Int`.

2. Give a type proof for the program above, where the expression `compose (+3)` has been replaced by `(compose (+3) id)`. Assume that `id` has type $\forall a.a \to a$.

7-6 **Type rules (from Exam 20010104).** Give a type proof for the program

```
let double  = \ f -> \ x -> (f (f x))
 ;  flip    = \ f -> \ x  -> \ y -> f y x
 in  double (flip (+) 3)
 ni
```

7-7 **Type inference implementation (from Exam 20001023).** From Haskell we know the datatype `List`. It is defined in the prelude as follows:

```
data List a = Nil | Cons a (List a)
```

The language `SL` that you have worked with in the practicals knows no datatype definitions. Therefore, a programmer unfortunately cannot define such a datatype himself. However, we can add `Nil` and `Cons` as standard functions to the language, and we can extend the syntax of expressions by a new form of case distinction which allows us to access the components of a `List`:

```
Listcase e of Nil        -> ................
              Cons x xs  -> ... x ... xs ...
           fo
```

1. How must the abstract grammar of the language SL be extended in order to represent `Listcase` constructs. You may assume that both alternatives in a `Listcase` are always present.
2. Give a representation of the type "list of integers" in the following datatype:

```
data Type = TVar     VarName
          | TFree    VarName
          | TTerm    String   [Type]
          | TForall  Int       Type
          | TAny
          | TError   String
  deriving Show
```

3. Give the Haskell types of the functions `Nil` and `Cons`.
4. How can these type be represented in the datatype `Type` given above? How does the new initial environment look like?
5. What are the type rules for the new `Listcase` statement?

7-8 **Type inference implementation (from Exam 20010104).** From Haskell we know the datatype `Maybe`. It is defined in the prelude as follows:

```
data Maybe a = Nothing | Just a
```

The language `SL` that you have worked with in the practicals knows no datatype definitions. Therefore, a programmer unfortunately cannot define such a datatype himself. However, we can add `Nothing` and `Maybe` as standard functions to the language, and we can extend the syntax of expressions by a new form of case distinction which allows us to access the components of a `Maybe`:

```
Maybecase e of Nothing        -> .........
               Just x         -> ... x ...
           fo
```

1. How must the abstract grammar of the language SL be extended in order to represent `Maybecase` constructs. You may assume that both alternatives in a `Maybecase` are always present.
2. Give a representation of the type "`Maybe (Maybe Int)`" in the following datatype:

```
data Type = TVar     VarName
          | TFree    VarName
          | TTerm    String   [Type]
          | TForall  Int       Type
          | TAny
          | TError   String
  deriving Show
```

3. First give the Haskell types of the functions `Nothing` and `Just`. How can these type be represented in the datatype `Type` given above?
4. What are the type rules for the new `Maybecase` statement?
5. Give the AG code for the new parts of the expression syntax that takes care of type inference.

7-9 **Type systems (from Exam 20010504).** Where relevant, we will assume compiler version *SLti* in this exercise.

1. Give $\mathrm{U}(T_1, T_2)$, the unification of the type T_1 and T_2, where
   (a) $T_1 = a \to a \to b$ and $T_2 = \texttt{Int} \to c$,
   (b) $T_1 = (a \to a) \to a$ and $T_2 = a \to a \to a$.

   Explain in each case the arguments that lead to your answer.
2. Give the type inference rule for lambda abstraction and explain how it works. In particular, explain how to deal with the fact the we do not know the type of the parameter from the beginning.
3. Give a type inference tree of the following `SL` program:

```
let k = \x -> \y -> x
 in k (True, k 4)
 ni
```

   Hint: To keep the tree small, explain the contents of the different environments as well as the relations between the type variables that are introduced separately.
4. Look at the following situation in an inference tree where the COND rule is applied:

$$\textsc{Cond}: \frac{\Gamma \vdash \texttt{e}_1 : v_2 \qquad \Gamma \vdash \texttt{e}_2 : v_2 \to v_1 \to \texttt{Int} \qquad \Gamma \vdash \texttt{e}_3 : v_4 \to (v_4, v_5) \to v_5}{\Gamma \vdash \textbf{ if } \texttt{e}_1 \textbf{ then } \texttt{e}_2 \textbf{ else } \texttt{e}_3 \textbf{ fi } : a}$$

   Explain precisely how the type a of **if** $\texttt{e}_1$ **then** $\texttt{e}_2$ **else** $\texttt{e}_3$ **fi** can be determined on the basis of the given types of $\texttt{e}_1$, $\texttt{e}_2$, and $\texttt{e}_3$ and show the rule given above with the final substitution applied to all types.
5. The following fragment of AG-code is an incorrect implementation of the COND rule.

```
SEM Expr
  | If LOC.(csubst, cetp, cemsg) = unifymsg booltype ce_tp
       LOC.(rsubst, tp,   msg)   = unifymsg (ee_subst |=> te_tp) ee_tp
       LHS.subst                 = rsubst ||| ee_subst
       LHS.tp                    = tp
```

Explain (using the previous subpart of this exercise) why this is wrong, and correct the AG code.

6. The following type rule is a variant of the LET rule from the lecture notes:

$$\textsc{Let}: \frac{\Gamma \vdash \texttt{expr} : t \qquad (\Gamma - \texttt{id}) \cup (\texttt{id} : t) \vdash \texttt{body} : b}{\Gamma \vdash \textbf{let } \texttt{id} \ = \ \texttt{expr } \textbf{in } \texttt{body } \textbf{ni} : b}$$

Explain what differences there are compared to the rule from the lecture notes and what effect these differences have. To illustrate the differences, give for instance programs that cannot be typed with the rule above, but can be with the rule from the lecture notes.

7. BONUS: Are there programs that can be typed with the LET rule above, but not with the original one. If yes, which programs? If not, explain why!

Appendix A

Laboratory exercises

A.1 Parsing BibTeX files

A.1.1 Introduction

The first exercise in this compiler construction course consists of writing a parser for BibTEX using the latest implementation of parser combinators. The purpose of this exercise is to refresh your memory of parsing; parsers being basic ingredients of compilers.

BibTEX is a data format for describing bibliographic data, i.e., data about books, scientific articles in journals, theses, etc. A bib database can be queried to get entries that are cited in a document and the entries obtained can be typeset in a variety of ways. This makes the bibliographic data portable and independent from a specific document or document style. An example entry is

```
@PhdThesis{Vis97.thesis,
  author =     {Visser, Eelco},
  title =      {Syntax Definition for Language Prototyping},
  year =       {1997},
  month =      {September},
  school =     {University of Amsterdam}
}
```

The `bibtex` tool reads the citations from a document, retrieves the corresponding entries from a bibtex database (file) and outputs the entries formatted in the document preparation language LATEX. For example, the entry above is formatted as

```
\bibitem{Vis97.thesis}
Eelco Visser.
\newblock {\em Syntax Definition for Language Prototyping}.
\newblock PhD thesis, University of Amsterdam, September 1997.
```

The Assignment The goal of this assignment is to produce a Haskell program that defines

- a data type description for BibTEX in Haskell, and
- a parser for BibTEX using the parser combinators described in Chapter 2. The parser should produce an element of the BibTEX data type definition.

Tips

- The complete syntax of BibTEX as accepted by `bibtex` is described in Section A.1.3 using the syntax definition formalism EBNF. This definition can be used as guideline in writing the parser and is considered to define the syntax of BibTEX for this assignment.
- Several aspects of the BibTEX syntax are rather tricky. In the next section five levels of BibTEX are defined, ranging from easy to difficult. Try making a parser for level n before you try level $n+1$. If you don't succeed in making a parser for full BibTEX, hand in a parser for the highest level that you can deal with. Your grade will be 2*the highest level that you hand in correctly.
- Several example input files that can be used for testing your parser are provided at the web page mentioned on the previous page.

A.1.2 BibTEX - Level 1 to 5

Level 1: Regular Entries, String Values

At this level the parser can deal with regular entries (with nonterminal `EEntry` in the EBNF definition), i.e., no comment, preamble or string entries. The field values of these regular entries are numbers or are strings delimited by double quotes that do not contain double quotes. Entry names and keys are just simple identifiers like in Java and Haskell. The fields are enclosed in curly braces only. A level 1 parser can parse the file `level1.bib`. An example of such an entry is:

```
@INBOOK{chicago,
   title = "The Chicago Manual of Style",
   publisher = "University of Chicago Press",
   edition = "Thirteenth",
   year = 1982,
   pages = "400--401",
   key = "Chicago"
}
```

Level 2: Strings and Preambles

At level 2 two other types of entries, i.e., preamble (EPreamble) and string (EString), are added. Field values can now also be names, but otherwise the restriction on field values remains the same. The keywords "preamble" and "string" should be in lowercase only. A level 2 parser can parse the file `level2.bib`. Examples of non-regular entries and the use of names are:

```
@preamble{"% Bibliography generated from level2.bib"}
@string{tosem =
        "ACM Transactions on Software Engineering and Methodology"}

@Article{BV96,
  author =  "Van den Brand, Mark G. J. and Visser, Eelco",
  title =   "Generation of Formatters for Context-free Languages",
  journal = tosem,
  year =    1996,
  month =   "January",
  volume =  5,
  number =  1,
  pages =   "1--41"
}
```

Level 3: Parentheses, Commas and Hash

A few more peculiarities are added: (1) Instead of `{` and `}` to delimit entries, `(` and `)` can also be used for all types of entries:

```
@string (SCRIBE-NOTE = "Chapter 12 deals with bibliographies")
```

(2) The last field of an entry can be followed by a comma. (3) Field value strings can be concatenated by the concatenation operator `#`. For example,

```
@string{trans = "ACM Transactions on "}
@string{toplas = trans # "Programming Languages and Systems"}
```

A level 3 parser can parse the file `level3.bib`.

Level 4: Fields with Quotes and Curly Braces

At level 4 the parser can deal with field values that contain double quotes and nested pairs of curly braces. Note that a double quote can occur in a field value only enclosed in `{` and `}`. Also note that field values can contain newlines. A level 4 parser can parse the file `level4.bib`. An example entry with these phenomena is:

```
@PhdThesis{Aas92,
  author = {Aasa, Annika},
  title = {User Defined Syntax},
  year = {1992},
  school = {Department of Computer Sciences,
            Chalmers University of Technology
            and University of {G\"oteborg}},
  address = "S-412 96 {G\"oteborg}, Sweden"
}
```

Level 5: Comment (Full BibTEX)

To deal with full BibTEX we have to make four final modifications to the parser: (1) Keywords should not be case sensitive, i.e., `String`, `sTrIng`, and `STRING` are all valid and equivalent keywords to indicate a string entry. (2) Entry keys have a much more liberal syntax than just identifier syntax. (3) Comment can be included using `comment` entries. (4) But actually these are not needed (although they should be supported by the parser) since any text between entries not containing `@` is considered comment in BibTEX.

A level 5 parser can parse the file `btxdoc.bib`, which documents all weird things allowed by `bibtex`. An example fragment of that file with comment is:

```
Copyright (C) 1988, all rights reserved.

@COMMENT{You may put a comment in a ‘comment’ command,
        the way you would with SCRIBE.}

Or you may dispense with the command and simply give the comment,
as long as it’s not within an entry.

@MISC{prime-number-theorem,
 author = "Charles Louis Xavier Joseph de la Vall{\’e}e Poussin",
 note = "A strong form of the prime number theorem, 19th century"
}
```

A.1.3 BibTEX in EBNF

Notes:
1. The syntax is not complete. For example, the nonterminal "EName" can also produce the word `string`, but as this is a keyword it should be excluded
2. LAYOUT is allowed between 'lexical' elements. Lexical elements are elements that form a unit, e.g. numbers, identifiers, strings between " " or { }, or characters that stand for themselves like @ ( ) outside of strings.
3. `~[...]` means all characters except those between the `[ ]`.
4. `\t` = tab, `\n` = newline.
5. The notation {`Field ","`}* means a (possibly empty) sequence of Fields separated by comma's. This is a bit awkward as the * is already used to denote sequences in BNF. It should be read here as if defined by one of the pChain parser combinators.

```
Bibtex:
    C {Entry C}* C

Entry:
    EComment | EPreamble | EString | EEntry

EComment:
    "@" Comment "{" ~[\}]+ "}" |
    "@" Comment "(" ~[\)]+ ")"

EPreamble:
    "@" Preamble "{" Value "}" |
    "@" Preamble "(" Value ")"

EString:
    "@" String "{" Field "}" |
    "@" String "(" Field ")"

EEntry:
    "@" EName "{" Key "," {Field ","}* ","? "}" |
    "@" EName "(" Key "," {Field ","}* ","? ")"

String:    [Ss][Tt][Rr][Ii][Nn][Gg]
Preamble:  [Pp][Rr][Ee][Aa][Mm][Bb][Ll][Ee]
Comment:   [Cc][Oo][Mm][Mm][Ee][Nn][Tt]

LAYOUT:    [\ \t\n]*
C:        ~[\@]*

Field:  Name "=" Value

Value:
    Name                 |
    "{" ValWords    "}"  |
    "\"" ValWordsDQ "\"" |
    Value "#" Value      |

ValWord:      ~[\{\}]+
ValWordDQ:    ~[\{\}"]+
ValWords:     (ValWord   | ("{" ValWords "}"))*
ValWordsDQ:   (ValWordDQ | ("{" ValWords "}"))*

Name:    [A-Za-z0-9\-\_\/]+
EName:   Name
Key:     ~[\ \t\n\,\=\{\}\@]+
```

A.1.4 Template for Parsing with Combinators

See `http://www.cs.uu.nl/docs/vakken/ipt`.

A.2 Generating OZIS files from BibTeX files

A.2.1 Introduction

Scientific articles often are written using LaTeX. Part of LaTeX is the BibTex bibliography database subsystem, a tool which uses a text based format for storing bibliography entries in such a way that they are easily incorporated in publications written with LaTeX.

However, the BibTex format is not the only existing format for storing bibliographies. In particular employees of the Institute of Information and Computing Sciences at Utrecht are required to submit their publications in a format specifically used by the University for its publications, the format used by OZIS (universitaire OnderZoek InformatieSysteem). The purpose of this exercise is to write a tool which translates bibliography entries written in BibTex to the format required by OZIS. The following is an example of what is accepted by this tool (i.e. BibTex format):

```
@book{dijk01ipt-lecturenotes
, title = {{Implementation of Programming Languages, Lecture Notes}}
, author = {Dijkstra, Atze and Swierstra, Doaitse}
, publisher = {Utrecht University, Institute of Information and Computing Sciences}
, year = {2001}}
```

with the following output (input for manual entry into OZIS).

```
A. 6.1.4 Software technology
B. -
C. -
D. Wetenschappelijke publicatie (internationaal)
E. Boek
   Titel: Implementation of Programming Languages, Lecture Notes
   Auteur: Dijkstra, Atze and Swierstra, Doaitse
   Uitgever: Utrecht University, Institute of Information and Computing Sciences
   Jaar: 2001
```

The learning goal of this exercise is refreshing parser combinator (Chapter 2) usage and getting acquainted with the attribute grammar system (Chapter 4).

A.2.2 Specification

The following text was given to the employees as the specification of what had to be produced for OZIS (in Dutch).

> Welke informatie moet vermeld worden en hoe moet de informatie aangeleverd worden?
>
> Voor elke publicatie moeten wij weten:
>
> **A.** De leerstoelgroep cq onderzoekscode (In het kader waarvan de publicatie is geschreven):
>
> **6.1.1** Algorithm design
> **6.1.2** Geometry, Imaging and virtual environments
> **6.1.3** Intelligent systems
> **6.1.4** Software technology
> **6.1.5** Decision support system

6.1.6 Large distributed databases

6.2.1 Information science

B. Is de publicatie geschreven in het kader van een onderzoeksschool. Zo ja, welke?

C. De werkrelaties van auteurs die werkzaam zijn buiten het Instituut voor Informatica en Informatiekunde. Vermeld hiervan behalve volledige naam met titelatuur, de Universiteit (of indien dat het geval is de bedrijfsnaam), faculteit , functie, werkplaats en email.

D. De classificatie:

1. Een wetenschappelijke publicatie, gericht op collega-onderzoekers, meestal met als doel de uitbreiding van kennis.
2. Een wetenschappelijke publicatie internationaal.
3. Vakpublicatie, gericht op breder publiek meestal met als doel de verspreiding van kennis.
4. Populair, publicatie gericht op ontwikkelde leken.
5. Overig, overige producten van wetenschappelijke activiteiten zoals cd-coms etc. Aangeven.

E. Het type (en de bij elk type vermeldde gegevens):

Annonatie rechtscollege, datum uitspraak, tijdschrift, nr. annonatie, titel, auteur, class

Artikel (letter to editor) titel, auteur, class, tijdschrift, informatiedrager (indien anders dan papier), volume nr, reeks nr, issn, pagina's

Artikel in bundel (proceedings) titel, auteur, titel bundel, editor, class, pagina's, uitgever, plaats uitgave, volume nr, isbn.

Bijdrage week of dagblad titel, auteur, class, week/dagblad, datum, pagina's,

Boek monografie; (editorial book) titel, auteur, class, uitgever, plaats uitgave, isbn nr., pagina"s, reeks nr.

Boekbespreking titel(s) boek(en), uitgever(s), redacteur(en), tijdschrift, volume nr.,pagina"s, class.

Boekdeel, hoofdstuk (chapter) titel, auteur, boektitel, editor, class, uitgever, plaats uitg., isbn nr., pagina's, reeks nr.

Boekredactie titel boekredactie, auteur, class, uitgever, plaats uitgave, isbn nr., pagina's, reeks nr.

Dissertatie (phdthesis) titel, promovendus, promotor, instelling promotie, instelling onderzoek, uitgever, plaats uitg., isbn, class, datum promotie, pagina's

Extern rapport titel. auteur, instituut, class, uitgever, plaats uitgave, isbn nummer, pagina's, opdrachtgever, nummer

Intern rapport titel. auteur, instituut, class, uitgever, plaats uitgave, isbn nummer, pagina's, opdrachtgever, nummer

Octrooi titel, auteur, aangevr/verlengd, octrooi nr.

Overig (b.v. software product) titel, auteur, soort resultaat en wat verder van toepassing is.

Voordracht, lezing titel, auteur, gebeurtenis, plaats, datum, pagina's

Tijdschriftredactie; referee-schap tijdschrift, volume nr., reeks nr., redacteur, class

Not all required information can be extracted from BibTex entries, simply because it's not there. This exercise therefore is restricted to the production of as much as possible information from a BibTex file. The rest may be filled in arbitrarily.

BibTex files also contain more information then is needed for OZIS. Any 'too much' information may be copied to the OZIS output.

A.2.3 What to do

This section describes which issues have to be resolved. These issues are described as a sequence of steps which can be followed in that order whilst making the exercise.

1. As a starting point a working version of a BibTex pretty printer is given. However, this implementation does not accept all BibTex features and it does not produce the text required by OZIS. The first step is to extend the given pretty printer so that all BibTex language constructs are supported.

   The grammar for BibTex is given in Section A.1.3, page 142, which is part of another BibTex exercise which should be read for extra information. The given BibTex pretty printer implements levels 1, 3 (except Hash) and 4 from this exercise; it consists of a parser coupled with an attribute grammar specification for the pretty printing. Only entries (`EEntry`) are accepted; comments (`EComment`), preambles (`EPreamble`) and strings (`EString`) are not. Extend the parser and the attribute code for pretty printing so that the following input is accepted and pretty printed:

   ```
   @COMMENT(You may put a comment in a 'comment' command,
           the way you would with SCRIBE.)

   Free comment may also be between other entries.

   @preamble("% Bibliography generated from level2.bib")

   @string{trans = "ACM Transactions on "}
   ```

   Any text (like `Free comment may...`) may appear between entries. However, this text is neither pretty printed nor used in any other way.

2. The next step adds the creation of text for OZIS and writing this information to a file with extension `.ozis`. First make a copy of the attribution for pretty printing and rename the `pp` (and `pps`) attribute to a new attribute name, say `ppoz` (and `ppozs`). The AG system will create a program where the root parser (related to `DATA Root`) returns a tuple. Consequently the invocation of the parser has to be changed from

   ```
   main      = do ...
                  let (pp, errors) = parse pRoot tokens
                  ...
   ```

   to

   ```
   main      = do ...
                  let ((pp,ppoz), errors) = parse pRoot tokens
                  ...
                  writeFile (basename ++ ".ozis") (disp ppoz 300 "")
                  putStr ("\nGenerated OZIS data in file " ++ (basename ++ ".ozis"))
   ```

   or something equivalent[1].

   Adapt the copied pretty printing code so that the entries appear in the required OZIS format in the `.ozis` file. BibTex specifics like quotes (`"`) and curly braces (`{`, `}`) should be left out. It should resemble the suggested format from the introduction (Section A.2.1) as closely as possible. Improvements on the layout are (of course) allowed.

3. BibTex string entries are used to define strings which can subsequently be used at more than one place in a BibTex file. For example, a `publisher` field will often be the same for different BibTex entries. The BibTex tool from LaTex automatically expands these strings to their defined value. Therefore, the following entry

   ```
   @BOOK{van-leunen,
      title = "A Handbook for Scholars",
   ```

[1] The order in which the AG system puts results in the tuple depends on the implementation/version of the system. The worst which can happen is that the two pretty printed outputs are swapped.

```
   author = "Mary-Claire van Leunen",
   publisher = sv,
   year = 1979 }
@string{sv    = "Springer-Verlag"}
```

is equivalent with

```
@BOOK{van-leunen,
   title = "A Handbook for Scholars",
   author = "Mary-Claire van Leunen",
   publisher = "Springer-Verlag",
   year = 1979 }
```

The order in which the entries appear is irrelevant.

This behavior also has to be implemented. This can be done by gathering all string definitions, passing it to the root of the abstract syntax using a synthesized attribute and then distributing it to all entries via an inherited attribute. All field definitions with a value which is neither quoted nor surrounded by curly braces should then be replaced for the OZIS format. If no value is defined, an error message like

```
@PhdThesis{ ...
          , publisher = uu<!ERROR: no replacement for uu!>
          ...}
```

should be produced in the pretty printed BibTex output.

Be aware that numbers may also appear without quotes or curly braces, but should neither be replaced nor give an error message.

4. The BibTex tool also performs the string concatenation (`#` operator). This also has to be implemented for the OZIS output.

5. The fields in the BibTex file have labels written in English. These should be replaced by Dutch OZIS counterparts as suggested in the specification (Section A.2.2). See the introduction for an example where (e.g.) `publisher` is replaced by `Uitgever`.

6. The OZIS entry labeled with `C.` requires information about some of the people involved with the publication to be specified. This can be approximated by printing the value of the `author` field for an entry, if an `author` field is specified.

7. Bonus. Not all fields of an entry have to be copied to the OZIS output. For example, the `year` is not required as these entries have to be submitted for OZIS on a year by year basis. Therefore, the year is already known.

   Adapt the implementation so that fields can be filtered out.

A.3 Condition Shortcut Evaluation

In the SL fragment

```
 a := 30
; if a < 10 && a >= 0 then a / 2 else a * 2 fi
```

it is clear that the application of `&&` will have result "False" even before `a >= 0` is evaluated. Still, code is generated that evaluates both operands of `&&`. Adapt the compiler so that the second operand is only evaluated if really necessary. The `||` operator has the potential for a very similar optimization. Implement this optimization as well.

Perform the following steps to accomplish your goal:

1. The problem can be solved by adding two labels as inherited attribute to every boolean expression. One for the case that it becomes clear during evaluation that the result will be "False" in any case, the other for the case that the result is already known to be "True". The code for `&&` can then use one of the labels as jump destination for the case that the left operand evaluates to "False".

   This implies that a boolean expression does not leave a value on the stack anymore, but always results in a jump. Therefore, the AG has to be adapted in such a way that boolean expressions are allowed only as conditions in "if"-expressions. Modify the parser and the AG data structures accordingly by splitting parsers and data structures for integer and boolean expressions.

2. Add the two previously mentioned labels as attributes to boolean expressions. Initialize these (among other positions) in the conditional of an "if"-expression with the appropriate destinations in the code. The "True"-label, for instance, should point to the location directly after the "then"-part of the "if"-expression.

3. Use the keyword `Bool` such as in

   ```
   Bool a < 10 && a >= 0
   ```

   to (re-)allow the use of a boolean value as a value on top of the stack.

A.4 Extending SL with Tuples

The simple language SL of these lecture notes does not support tuples, that is a value consisting of other values. In Haskell tuples are constructed and used as follows

```
let t = (3,(True,4))
 in fst (snd t)
```

The purpose of this exercise is to add a similar construct to SL. This can be done in all compilers of these lecture notes (Section 5.1, Section 6.4 and Section 7.6), each giving rise to variations in the issues involved. This exercise is concerned only with extending the simplest version (SLbb, Section 5.1) of the compilers.

A.4.1 What to do

Add the following language features to the compiler

basic feature Creation of a tuple, e.g.

```
... (3,(True,4))
 ...
```

A tuple contains 2 elements only.

basic feature Extraction of an element, e.g.

```
... fst (snd t)
```

The function `fst` extracts the first element, the function `snd` the second.

extra feature Creation of a tuple of arbitrary size, e.g.

```
... (3,True,4)
 ...
```

A tuple contains as many elements as specified by its construction.

A.4.2 Issues

- Irrespective of the compiler involved, the parser, the abstract syntax and type structure have to be adapted. The semantics (that is, attributes) have to be adapted too but this depends on the compiler.
- The implementation of `fst` and `snd` can be done in different ways.
  - These functions can be added as syntactic constructs, that is, they are keywords and have their variant in the abstract syntax for expressions.
  - Alternatively, the functions can be added as builtin functions (lambda expressions).

  In case of the first alternative, `fst` (and `snd`) is and must be treated specially with respect to attribute definitions. For the second alternative this is not necessary. Which one to choose depends on the added features, in particular if > 2 elements are allowed in a tuple. In this context ask yourself the following questions to make a choice:
  - Does `fst` (and `snd`) work for tuples of all sizes?
  - How are the remaining (that is, 3rd, 4th, ...) accessed?

A.5 Optimizing Static Link usage

In the code generation variant of the SL compiler (Section 6.4) local variables are referenced via the mark pointer. Non-local variables (i.e. variables that are not from the current/innermost block) are referenced by first following the static link and then using the mark pointer thereby found. The following example (from Exercise 6-9)

```
let a     :: Int               = 5
 ;  id    :: Int -> Int        = \x -> x
 ;  const :: Int -> Int -> Int = \x y -> x
 ;  f     :: Int -> Int        =
             \b -> let g :: Int -> Int -> Int =
                             \p q -> id a * const q p + b
                   in g 3 4
                   ni
 in f 7
 ni
```

results in the following code for placing "`a`" on the stack in "`g`":

```
 LDL  -2
 LDA  -2
 LDA  1
```

But not just for `a`, but also for `id` and `const` the static link is needed. Each time, the static link is followed again. One can optimize here by dereferencing the static links once before the evaluation of the lambda-expression, and by "caching" elements that might be needed later as local variables. Under the assumption that the precomputed static link (of 2 levels above) has been stored in the local `1`, the new code will look like :

```
 LDL  1
 LDA  1
```

Extend the SL compiler so that all possibly necessary static links of a lambda expression will be computed before the body of the lambda is entered. Store these static links in (hidden) local variables and use the precomputed values as demonstrated in the example given above. However,

don't compute any more static links than really needed. The static link of 1 level above never has to be computed because it is already passed as a parameter (displacement `-2`).

Necessary steps:

1. Compute the references to the surrounding lexical levels without paying attention to the fact if they are really needed. For example (this is also from Exercise 6-9), in

```
let a :: Int = 5
 ;  f :: Int -> Int =
          \x -> let g :: Int -> Int =
                          \y -> let h :: Int -> Int = \z -> a
                                in h 3
                                ni
                in g 3
                ni
 in f 5
 ni
```

   only the static link for the level of "`a`" is really needed in "`h`"; not those for the intermediate levels ("`f`" and "`g`"). Try first without this further optimization.

2. Now include only those static links that are really necessary, as described in the previous step.

3. If a static link is used only once in a block, there lies no win in precomputing it. Adapt your solution accordingly.

A.6 Checking Stack usage

The SL program

```
let apply = \f x -> f x
 ;  incBy = \x -> let g = \y -> x + y
                  in g
                  ni
 in apply (incBy 1) 3
 ni
```

does not evaluate to `4`, as would be expected. The reason for this is the use of the paramter "`x`" in "`g`". In itself, this would not constitute a problem, but "`g`" is called via the expression `(incBy 1)` in "`apply`", after "`x`" has already been removed from the stack. On a stack, the value that is last created is the first to be removed again (Last In First Out). This property causes the problem here: in the example, the environment that is required by "`g`" (containing "`x`") has already been removed in the moment that the resulting "`g`" is called.

For the problem just sketched we have to add a runtime check (i.e. during the execution of the code). The static link of a function is always known. If it points to a higher position on the stack than the mark pointer, we know that the function might refer to already freed locations. We also can distinguish if a value on the stack is a function or not: the second word of an integer is 0, for a function it contains the start address (which is never 0).

Adapt the SL compiler so that code for this check is generated. The test must take place close to the positions where a result in a function is returned; the result must then be checked. The program should stop if the described error situation is detected.

A.7 Addition of Maybe datatype

This exercise is based on Exercise 7-8; make that exercise before starting with this one.

The language SL (SLti variant) does not allow the definition of new datatypes as can be done with the keyword `data` in Haskell. Inclusion of such a general mechanism requires many modifications to SL. Much simpler is the addition of one specific datatype, in the case of this exercise the type `Maybe`, defined in Haskell as follows:

```
data Maybe a  = Nothing | Just a
```

This definition introduces constructor functions (`Nothing` and `Just`) for creating new instances of this data type as well as a new type (`Maybe`) in the type system.

The introduction of a new datatype normally also introduces the possibility to pattern match on an instance of a datatype (written in Haskell):

```
x = Just 4
case x of
  Nothing -> True
  Just a  -> False
```

However, for SL, a much simpler way of making a choice based upon the structure of a `Maybe` instance is defined:

```
Maybecase e of Nothing -> .........
                  Just x  -> ... x ...
               fo
```

It is much simpler because it only works for instances of `Maybe`. Its syntactical structure is hardcoded in the grammar of SL and follows the structure of the datatype `Maybe`. That is, the structure of the pattern is hardcoded.

The goal of this exercise is to add the datatype Maybe, its constructor functions and the `Maybecase` case expression. In order to do this, implement the following steps (not necessarily in this order):

1. Add the datatype for `Maybe`. This means the addition of a new basic type and the addition of appropriate constructor functions.
2. Add the syntax (that is, the parsers) for `Maybecase`.
3. Add the modifications for the type inferencer.

A.8 Generating Type Proofs

A.8.1 Introduction

Chapter 7 discusses the importance of type checking and type inferencing. The most important goal of type checking is detection of ill-typed expressions at compile-time, to prevent these errors from occurring during execution.

A set of type rules is given for the SL-language. A type rule contains judgements of the form $(\Gamma \vdash expr : t)$, expressing that $expr$ can be assigned type t in environment Γ. The type rules are syntax-directed since there is exactly one rule for each language construct. Furthermore, there are two type rules, GEN and SPEC, that allow generalization and specialization of types. These rules are necessary to handle the polymorphism in the language. Summarising, we say that type rules are a tool to prove that a type can be assigned to an expression in a given environment. Unfortunately, the construction of such a proof is not always straightforward.

Another aspect of type systems is a type inference algorithm, that assigns a type to an expression, if possible. An implementation of such a type inference algorithm is presented in an attribute grammar (Section 7.6). This algorithm is based on the unification of types. If a unification of two types fails (returns $\perp$) a type error message is constructed, and the expression is ill-typed. On the other hand, an expression is well-typed if all unifications succeed; the most general type (also called the *principal type-scheme*) is returned.

What is the correspondence between the type rules and the type inference algorithm? The type inference algorithm should only return types that are correct with respect to the given type rules. The algorithm should be *sound* and *complete* with respect to the type rules, but in this assignment we will only consider the soundness of the type inference algorithm. In other words: if the algorithm assigns a type to an expression, we (automatically) want to construct a proof with the type rules that the assigned type is correct. This is the subject of this exercise.

A.8.2 Assignment

The goal of this exercise is to generate type proofs for expressions of the SL-language and in that way learn how type proofs are constructed theoretically as well as in practice. We will use the type rules explained in chapter 7 to construct such a proof. The compiler has to be extended such that a type proof is generated automatically for each well typed expression. A good understanding of the type inference algorithm is required to complete this exercise. As a starting point an already modified version of the SLti compiler can (and should) be used. This version already contains a great deal of the administration necessary for this exercise. Using this version as a starting point the major work consists of extracting the right information out of the type inference attributes and putting this information in the right place in the produced proof.

Step 1: Getting started

The SLti compiler needs some changes in order to make the following example

```
let i = \x -> x
 in i i
 ni
```

produce the following output

$$\dfrac{\dfrac{\dfrac{\dfrac{(\mathsf{x} : \ldots) \in \ldots}{\ldots \vdash \mathsf{x} : \ldots}\ \textsc{Ident}}{\ldots \vdash \lambda \mathsf{x} \to \mathsf{x} : \ldots}\ \textsc{Lambda} \qquad \dfrac{\dfrac{\dfrac{(\mathsf{i} : \ldots) \in \ldots}{\ldots \vdash \mathsf{i} : \ldots}\ \textsc{Ident}}{\ldots \vdash \mathsf{i} : \ldots}\ \textsc{Spec} \qquad \dfrac{\dfrac{(\mathsf{i} : \ldots) \in \ldots}{\ldots \vdash \mathsf{i} : \ldots}\ \textsc{Ident}}{\ldots \vdash \mathsf{i} : \ldots}\ \textsc{Spec}}{\ldots \vdash \mathsf{i}\ \mathsf{i} : \ldots}\ \textsc{Apply}}{\ldots \vdash \mathbf{let}\ \mathsf{i} = \lambda \mathsf{x} \to \mathsf{x}\ \mathbf{in}\ \mathsf{i}\ \mathsf{i}\ \mathbf{ni} : \ldots}\ \textsc{Let}}{\ldots \vdash \mathbf{let}\ \mathsf{i} = \lambda \mathsf{x} \to \mathsf{x}\ \mathbf{in}\ \mathsf{i}\ \mathsf{i}\ \mathbf{ni} : \forall a.a \to a}\ \textsc{Gen}$$

This output is the result of processing the `.tex` file generated by the (adapted) compiler with LaTeX.

You will have to modify the compiler to make the compiler produce useful and meaningful information in place of the dots. Before doing so, try to manually fill in the empty spots (as indicated by ...) with the appropriate values (don't immediately look at the solution a bit further). Do this also for other examples. Also, try to understand the reason why the tree looks this way. For example can you explain why the rules Gen and Spec are already inserted in the proof.

Step 2: Types

The following type synonym represents a judgement, and is used throughout the (modified) compiler:

```
type Judgement = (Gamma,PP_Doc,Type)
```

A judgement (of type `Judgement`) contains a set of assumptions, a pretty-printed expression, and a type. Since the type rules are syntax-directed, the shape of the proof corresponds with the syntax tree. In the attribute grammar, we define the judgements for the proof. For instance, the judgement for an identifier is:

```
SEM Expr
  | Ident    LOC . judge     = (unknownGamma,pptex,unknownType)
```

The problem now is what the `unknownType` (and the `unknownGamma`) should be. The required information is available in the environment (gamma) as the type of an expression is inferred by the inference algorithm. First, replace the occurrences of `unknownType` by the correct type as inferred by the inference algorithm. The LaTeX output now will print the types correctly.

Step 3: Assumptions

Now the `unknownGamma` in `(unknownGamma,pptex,unknownType)` has to be replaced. However, this gamma can be quite large so we refer to it by a number instead of putting it in the generated LaTeX derivation tree directly. Such a number refers to an entry of a table of sets of assumptions. This table of sets of assumptions looks like:

$$\Gamma_1 = \{ \ \texttt{...} \ : \ \ldots$$

$$\Gamma_2 = \{ \ \texttt{i} \ : \ v_2 \rightarrow v_2$$

$$\Gamma_3 = \left\{ \begin{array}{lcl} \texttt{x} & : & v_2 \\ \texttt{i} & : & v_2 \rightarrow v_2 \end{array} \right.$$

$$\Gamma_4 = \{ \ \texttt{i} \ : \ \forall a.a \rightarrow a$$

We introduce two type synonyms to represent such a table:

```
type Gammas    = [(Gamma,Assumptions)]
type Gamma     = Int
```

Each assumption set has a unique number, now used as a reference to an assumption set. Construct such a list of type `Gammas`, containing each set of assumptions that is used in the type proof. Include this list in the proof, and adjust the judgements so that they refer to the appropriate set of assumptions. It's recommended to start with an empty initial environment (see `initgamma`), because this will result in smaller assumption sets. Furthermore, care has to taken not to create identical assumption sets (which language constructs change a set of assumptions?) as this will clutter the generated output.

Step 4: Substitution

The generated tree of the SL example used will probably be similar to the following derivation tree:

$$\dfrac{\dfrac{\dfrac{(\mathsf{x} : v_2) \in \Gamma_3}{\Gamma_3 \vdash \mathsf{x} : v_2}\ \textsc{Ident}}{\Gamma_2 \vdash \lambda \mathsf{x} \to \mathsf{x} : v_2 \to v_2}\ \textsc{Lambda} \qquad \dfrac{\dfrac{\dfrac{(\mathsf{i} : \forall a.a \to a) \in \Gamma_4}{\Gamma_4 \vdash \mathsf{i} : \forall a.a \to a}\ \textsc{Ident}}{\Gamma_4 \vdash \mathsf{i} : (v_5 \to v_5) \to v_5 \to v_5}\ \textsc{Spec} \qquad \dfrac{\dfrac{(\mathsf{i} : \forall a.a \to a) \in \Gamma_4}{\Gamma_4 \vdash \mathsf{i} : \forall a.a \to a}\ \textsc{Ident}}{\Gamma_4 \vdash \mathsf{i} : v_5 \to v_5}\ \textsc{Spec}}{\Gamma_4 \vdash \mathsf{i}\ \mathsf{i} : v_5 \to v_5}\ \textsc{Apply}}{\dfrac{\Gamma_1 \vdash \mathbf{let}\ \mathsf{i} = \lambda \mathsf{x} \to \mathsf{x}\ \mathbf{in}\ \mathsf{i}\ \mathsf{i}\ \mathbf{ni} : v_5 \to v_5}{\Gamma_1 \vdash \mathbf{let}\ \mathsf{i} = \lambda \mathsf{x} \to \mathsf{x}\ \mathbf{in}\ \mathsf{i}\ \mathsf{i}\ \mathbf{ni} : \forall a.a \to a}\ \textsc{Gen}}\ \textsc{Let}$$

with the same Γ's as earlier:

$$\Gamma_1 = \{$$

$$\Gamma_2 = \left\{ \begin{array}{lcl} \mathtt{i} & : & v_2 \to v_2 \end{array} \right.$$

$$\Gamma_3 = \left\{ \begin{array}{lcl} \mathtt{x} & : & v_2 \\ \mathtt{i} & : & v_2 \to v_2 \end{array} \right.$$

$$\Gamma_4 = \left\{ \begin{array}{lcl} \mathtt{i} & : & \forall a.a \to a \end{array} \right.$$

However, one has to make sure the types printed in step 1 are indeed the types as inferred by the inference algorithm. This may not always be the case as the printed types are intermediate inferred types, not the final ones. This can be repaired by the appropriate use of an attribute representing the final substitution.

Appendix B

SSM instructions (and other topics)

The content of this appendix is automatically generated from the SSM interpreter.
Generated by SSM 1.2.2 at Wed Feb 13 15:56:59 CET 2002

B.1 Topics

add(B.2), ajs(B.3), and(B.4), annote(B.5), bra(B.6), brf(B.7), brt(B.8), bsr(B.9), code(B.10), div(B.11), eq(B.12), False(B.13), ge(B.14), gt(B.15), halt(B.16), help(B.17), instruction(B.18), jsr(B.19), labels(B.20), lda(B.21), ldaa(B.22), ldc(B.23), ldl(B.24), ldla(B.25), ldma(B.26), ldml(B.27), ldms(B.28), ldr(B.29), ldrr(B.30), lds(B.31), ldsa(B.32), le(B.33), link(B.34), lt(B.35), markpointer(B.36), memory(B.37), mod(B.38), MP(B.39), mul(B.40), ne(B.41), neg(B.42), nop(B.43), not(B.44), or(B.45), PC(B.46), programcounter(B.47), registers(B.48), ret(B.49), return(B.50), RR(B.51), SP(B.52), sta(B.53), stack(B.54), stackpointer(B.55), stl(B.56), stma(B.57), stml(B.58), stms(B.59), str(B.60), sts(B.61), sub(B.62), swp(B.63), swpr(B.64), swprr(B.65), syntax(B.66), trap(B.67), True(B.68), unlink(B.69), xor(B.70)

B.2 Semantics: add

Instruction	**Nr of inline Opnds**	**Nr of stack Opnds**	**Nr of stack Results**	**Instr code (hex)**
add	0	2	1	0x1

Description Addition. Replaces 2 top stack values with the addition of those values.

Pre and Post State
$SP_{post} = SP_{pre} - 1$
$M_{post}[SP_{post}] = M_{pre}[SP_{pre} - 1] + M_{pre}[SP_{pre}]$

Example

```
ldl 2 ; increment local var
ldc 1
add
stl 2
```

B.3 Semantics: ajs

Instruction	**Nr of inline Opnds**	**Nr of stack Opnds**	**Nr of stack Results**	**Instr code (hex)**
ajs	1	instruction dependent	0	0x64

Description	Adjust Stack. Adjusts the stackpointer with fixed amount.
Pre and Post State	$SP_{post} = SP_{pre} + M_{post}[PC_{pre} + 1]$
Example	`ajs -2 ;lower stack by 2`

B.4 Semantics: and

Instruction	Nr of inline Opnds	Nr of stack Opnds	Nr of stack Results	Instr code (hex)
and	0	2	1	0x2

Description	And. Replaces 2 top stack values with the bitwise and of those values.
Pre and Post State	$SP_{post} = SP_{pre} - 1$ $M_{post}[SP_{post}] = M_{pre}[SP_{pre} - 1] \& M_{pre}[SP_{pre}]$

Example	`ldc 0xFF00 ; variant of ldc 0xF000`
	`ldc 0xF0F0`
	`and`

B.5 Semantics: annote

Instruction	Nr of inline Opnds	Nr of stack Opnds	Nr of stack Results	Instr code (hex)
annote	5	0	0	0xff

Description	Annotate. A meta instruction (not producing code), annotating the stack display in the user interface with text and color. Annote takes 5 arguments, (1) a register name, (2) a low offset w.r.t. the register (used as starting point for annotating), (3) a high offset, (4) a color, (5) text. Color can be one of black, blue, cyan, darkGray, gray, green, lightGray, magenta, orange, pink, red, yellow. Text including spaces need to be enclosed in double quotes. The annote instruction is tied to the preceding (non-meta) instruction and will be performed immediately after the execution of that instruction.
Pre and Post State	
Example	`annote SP -1 0 red "Pushed constants" ; annote top 2 stack values`

B.6 Semantics: bra

Instruction	Nr of inline Opnds	Nr of stack Opnds	Nr of stack Results	Instr code (hex)
bra	1	0	0	0x68

Description Branch Allways. Jumps to the destination. Replaces the PC with the destination address.

Pre and Post State $PC_{post} = PC_{pre} + M_{pre}[PC_{pre} + 1] + 2$

Example

```
bra main
subroutine ldc 1
ldc 2
add
str RR
ret
main: bsr subroutine
ldr RR
...
```

B.7 Semantics: brf

Instruction	Nr of inline Opnds	Nr of stack Opnds	Nr of stack Results	Instr code (hex)
brf	1	1	0	0x6c

Description Branch on False. If a False value is on top of the stack, jump to the destination.

Pre and Post State $SP_{post} = SP_{pre} - 1$
$PC_{post} = PC_{pre} + M_{pre}[PC_{pre} + 1] + 2(if\ false\ on\ top\ of\ the\ stack)$

Example

```
ldc 2
ldc 3
eq
brf FalseAction
```

B.8 Semantics: brt

Instruction	Nr of inline Opnds	Nr of stack Opnds	Nr of stack Results	Instr code (hex)
brt	1	1	0	0x6d

Description Branch on True. If a True value is on top of the stack, jump to the destination.

Pre and Post State $SP_{post} = SP_{pre} - 1$
$PC_{post} = PC_{pre} + M_{pre}[PC_{pre} + 1] + 2(if\ true\ on\ top\ of\ the\ stack)$

Example

B.9 Semantics: bsr

Instruction	Nr of inline Opnds	Nr of stack Opnds	Nr of stack Results	Instr code (hex)
bsr	1	0	1	0x70

Description	Branch to subroutine. Pushes the PC on the stack and jumps to the subroutine.
Pre and Post State	$SP_{post} = SP_{pre} + 1$ $M_{post}[SP_{post}] = PC_{pre} + 2$ $PC_{post} = PC_{pre} + M_{pre}[PC_{pre} + 1] + 2$

Example	`bra main`
	`subroutine ldc 1`
	`ldc 2`
	`add`
	`str RR`
	`ret`
	`main: bsr subroutine`
	`ldr RR`
	`...`

B.10 General: code

Code is the part of memory used to store instructions. It starts at address 0.

B.11 Semantics: div

Instruction	Nr of inline Opnds	Nr of stack Opnds	Nr of stack Results	Instr code (hex)
div	0	2	1	0x4

Description	Division. Replaces 2 top stack values with the division of those values.
Pre and Post State	$SP_{post} = SP_{pre} - 1$ $M_{post}[SP_{post}] = M_{pre}[SP_{pre} - 1]/M_{pre}[SP_{pre}]$

Example	`ldl -1 ; divide and leave on stack`
	`ldc 3`
	`div`

B.12 Semantics: eq

Instruction	Nr of inline Opnds	Nr of stack Opnds	Nr of stack Results	Instr code (hex)
eq	0	2	1	0xe

Description	Test for equal. Replaces 2 top stack values with boolean result of the test. False is encoded as 0, True as 1. Used in combination with brf. This is a variant of cmp combined with beq.
Pre and Post State	$SP_{post} = SP_{pre} - 1$ $M_{post}[SP_{post}] = M_{pre}[SP_{pre} - 1] == M_{pre}[SP_{pre}]$

Example	`ldc 2`
	`ldc 3`
	`eq`
	`brf FalseAction`

B.13 General: False

Value False is encoded by a 0.

B.14 Semantics: ge

Instruction	Nr of inline Opnds	Nr of stack Opnds	Nr of stack Results	Instr code (hex)
ge	0	2	1	0x13

Description Test for greater or equal. Replaces 2 top stack values with boolean result of the test. False is encoded as 0, True as 1. Used in combination with brf. This is a variant of cmp combined with bge.

Pre and Post State $SP_{post} = SP_{pre} - 1$
$M_{post}[SP_{post}] = M_{pre}[SP_{pre} - 1] >= M_{pre}[SP_{pre}]$

Example

```
ldc 2
ldc 3
ge
brf FalseAction
```

B.15 Semantics: gt

Instruction	Nr of inline Opnds	Nr of stack Opnds	Nr of stack Results	Instr code (hex)
gt	0	2	1	0x11

Description Test for greater then. Replaces 2 top stack values with boolean result of the test. False is encoded as 0, True as 1. Used in combination with brf. This is a variant of cmp combined with bgt.

Pre and Post State $SP_{post} = SP_{pre} - 1$
$M_{post}[SP_{post}] = M_{pre}[SP_{pre} - 1] > M_{pre}[SP_{pre}]$

Example

```
ldc 2
ldc 3
gt
brf FalseAction
```

B.16 Semantics: halt

Instruction	Nr of inline Opnds	Nr of stack Opnds	Nr of stack Results	Instr code (hex)
halt	0	0	0	0x74

Description Halt execution. Machine stops executing instructions.

Pre and Post State

Example

```
halt
```

B.17 General: help

The Simple Stack Runner executes instructions for a hypothetical (and thus simple) machine. See memory, registers, syntax, instruction as starting points for help.

B.18 General: instruction

An instruction is an encoding of some operation, executed in the machine. A set of instructions stored in memory is called the code. Some instructions have inline operands, that is, after their location in the code

an extra operand is stored, a constant, e.g. in "ldc 1". In pre/post conditions this location is indicated by M[PCpre+1] since it can be found on that location. The behavior of an instruction is both informally described as well as using pre/postcondifitions.

B.19 Semantics: jsr

Instruction	Nr of inline Opnds	Nr of stack Opnds	Nr of stack Results	Instr code (hex)
jsr	0	1	1	0x78

Description Jump to subroutine. Pops a destination from the stack, pushes the PC on the stack and jumps to the destination.

Pre and Post State
$SP_{post} = SP_{pre}$
$PC_{post} = M_{pre}[SP_{pre}]$
$M_{post}[SP_{post}] = PC_{pre} + 1$

Example

```
bra main
subroutine ldc 1
ldc 2
add
str RR
ret
main: ldc subroutine
jsr
ldr RR
...
```

B.20 General: labels

A label is an identifier indicating a position in the code. When loading, the code location of a label is calculated (called resolution). This is done in the user interface of the program and after loading labels are not kept consistent (when adding new instructions for example).

B.21 Semantics: lda

Instruction	Nr of inline Opnds	Nr of stack Opnds	Nr of stack Results	Instr code (hex)
lda	1	1	1	0x7c

Description Load via Address. Dereferencing. Pushes the value pointed to by the value at the top of the stack. The pointer value is offset by a constant offset.

Pre and Post State
$SP_{post} = SP_{pre}$
$M_{post}[SP_{post}] = M_{pre}[M_{pre}[SP_{pre}] + M_{pre}[PC_{pre} + 1]]$

Example

```
ldla -2 ; a different way of doing ldl -2
lda 0
```

B.22 Semantics: ldaa

Instruction	Nr of inline Opnds	Nr of stack Opnds	Nr of stack Results	Instr code (hex)
ldaa	1	1	1	0x80

Description Load Address of Address. Pushes the address of a value relative to the address on top of the stack. This instruction effectively adds a constant to the top of the stack.

Pre and Post State

$SP_{post} = SP_{pre} + 1$
$M_{post}[SP_{post}] = M_{pre}[SP_{pre}] + M_{pre}[PC_{pre} + 1]$

Example

```
ldaa -2
```

B.23 Semantics: ldc

Instruction	Nr of inline Opnds	Nr of stack Opnds	Nr of stack Results	Instr code (hex)
ldc	1	0	1	0x84

Description Load Constant. Pushes the inline constant on the stack.

Pre and Post State

$SP_{post} = SP_{pre} + 1$
$M_{post}[SP_{post}] = M_{pre}[PC_{pre} + 1]$

Example

```
ldl 2 ; increment local var
ldc 1
add
stl 2
```

B.24 Semantics: ldl

Instruction	Nr of inline Opnds	Nr of stack Opnds	Nr of stack Results	Instr code (hex)
ldl	1	0	1	0x88

Description Load Local. Pushes a value relative to the markpointer.

Pre and Post State

$SP_{post} = SP_{pre} + 1$
$M_{post}[SP_{post}] = M_{pre}[MP_{pre} + M_{pre}[PC_{pre} + 1]]$

Example

```
ldl -1 ; divide and leave on stack
ldc 3
div
```

B.25 Semantics: ldla

Instruction	Nr of inline Opnds	Nr of stack Opnds	Nr of stack Results	Instr code (hex)
ldla	1	0	1	0x8c

Description Load Local Address. Pushes the address of a value relative to the markpointer.

Pre and Post State

$SP_{post} = SP_{pre} + 1$
$M_{post}[SP_{post}] = MP_{pre} + M_{pre}[PC_{pre} + 1]$

Example

```
ldla -2 ; update local using its address
ldc 5
sta 0
```

B.26 Semantics: ldma

Instruction	Nr of inline Opnds	Nr of stack Opnds	Nr of stack Results	Instr code (hex)
ldma	2	1	instruction dependent	0x7e

Description Load Multiple via Address. Pushes values relative to by the value at the top of the stack. Same as single load variant but second inline parameter is size.

Pre and Post State

$displ = M_{pre}[PC_{pre} + 1]$
$size = M_{pre}[PC_{pre} + 2]$
$SP_{post} = SP_{pre} + size - 1$
$M_{post}[SP_{post} - size + 1..SP_{post}] = M_{pre}[M_{pre}[SP_{pre}] + displ..M_{pre}[SP_{pre}] + displ + size - 1]$

Example

```
none
```

B.27 Semantics: ldml

Instruction	Nr of inline Opnds	Nr of stack Opnds	Nr of stack Results	Instr code (hex)
ldml	2	0	instruction dependent	0x8a

Description Load Multiple Local. Pushes values relative to the markpointer. Same as single load variant but second inline parameter is size.

Pre and Post State

$displ = M_{pre}[PC_{pre} + 1]$
$size = M_{pre}[PC_{pre} + 2]$
$SP_{post} = SP_{pre} + size$
$M_{post}[SP_{post} - size + 1..SP_{post}] = M_{pre}[MP_{pre} + displ..MP_{pre} + displ + size - 1]$

Example

```
ldml -1 2 ; divide and leave on stack
ldc 3
div
```

B.28 Semantics: ldms

Instruction	Nr of inline Opnds	Nr of stack Opnds	Nr of stack Results	Instr code (hex)
ldms	2	0	instruction dependent	0x9a

Description Load Multiple from Stack. Pushes values relative to the top of the stack. Same as single load variant but second inline parameter is size.

Pre and Post State

$displ = M_{pre}[PC_{pre} + 1]$
$size = M_{pre}[PC_{pre} + 2]$
$SP_{post} = SP_{pre} + size$
$M_{post}[SP_{post} - size + 1..SP_{post}] = M_{pre}[SP_{pre} + displ..SP_{pre} + displ + size - 1]$

Example

```
ldms -1 2; multiply and leave on stack
mul
```

B.29 Semantics: ldr

Instruction	Nr of inline Opnds	Nr of stack Opnds	Nr of stack Results	Instr code (hex)
ldr	1	0	1	0x90

Description Load Register. Pushes a value from a register. Registers 0, 1, 2 and 3 are called PC (program counter), SP (stack pointer), MP (mark pointer) and RR (return register) respectively.

Pre and Post State
$SP_{post} = SP_{pre} + 1$
$M_{post}[SP_{post}] = REG_{pre}[M_{pre}[PC_{pre} + 1]]$

Example
```
ldr RR ; decrement register
ldc 1
sub
str RR
```

B.30 Semantics: ldrr

Instruction	Nr of inline Opnds	Nr of stack Opnds	Nr of stack Results	Instr code (hex)
ldrr	2	0	0	0x94

Description Load Register from Register. Copy the content of the second register to the first. Does not affect the stack.

Pre and Post State
$REG_{post}[M_{pre}[PC_{pre} + 1]] = REG_{pre}[M_{pre}[PC_{pre} + 2]]$

Example
```
ldrr SP MP ; SP <- MP
```

B.31 Semantics: lds

Instruction	Nr of inline Opnds	Nr of stack Opnds	Nr of stack Results	Instr code (hex)
lds	1	0	1	0x98

Description Load from Stack. Pushes a value relative to the top of the stack.

Pre and Post State
$SP_{post} = SP_{pre} + 1$
$M_{post}[SP_{post}] = M_{pre}[SP_{pre} + M_{pre}[PC_{pre} + 1]]$

Example
```
lds -1 ; multiply and leave on stack
ldc 2
mul
```

B.32 Semantics: ldsa

Instruction	Nr of inline Opnds	Nr of stack Opnds	Nr of stack Results	Instr code (hex)
ldsa	1	0	1	0x9c

Description Load Stack Address. Pushes the address of a value relative to the stackpointer.

Pre and Post State
$SP_{post} = SP_{pre} + 1$
$M_{post}[SP_{post}] = SP_{pre} + M_{pre}[PC_{pre} + 1]$

Example
```
ldsa -2 ; update value on stack using its address
ldc 5
sta 0
```

B.33 Semantics: le

Instruction	Nr of inline Opnds	Nr of stack Opnds	Nr of stack Results	Instr code (hex)
le	0	2	1	0x12

Description Test for less or equal. Replaces 2 top stack values with boolean result of the test. False is encoded as 0, True as 1. Used in combination with brf. This is a variant of cmp combined with ble.

Pre and Post State
$SP_{post} = SP_{pre} - 1$
$M_{post}[SP_{post}] = M_{pre}[SP_{pre} - 1] <= M_{pre}[SP_{pre}]$

Example

```
ldc 2
ldc 3
lr
brf FalseAction
```

B.34 Semantics: link

Instruction	Nr of inline Opnds	Nr of stack Opnds	Nr of stack Results	Instr code (hex)
link	1	0	instruction dependent	0xa0

Description Reserve memory for locals. Convenience instruction combining the push of MP and the adjustment of the SP.

Pre and Post State
$MP_{post} = SP_{pre} + 1$
$M_{post}[MP_{post}] = MP_{pre}$
$SP_{post} = MP_{post} + M_{pre}[PC_{pre} + 1]$

Example

```
bra main
subroutine link 2 ; reserve for 2 locals
ldc 1
ldc 2
add
stl 1 ; store in 2nd local
ldl 1
str RR
unlink
ret
main: bsr subroutine
ldr RR
...
```

B.35 Semantics: lt

Instruction	Nr of inline Opnds	Nr of stack Opnds	Nr of stack Results	Instr code (hex)
lt	0	2	1	0x10

Description Test for less then. Replaces 2 top stack values with boolean result of the test. False is encoded as 0, True as 1. Used in combination with brf. This is a variant of cmp combined with blt.

Pre and Post State
$SP_{post} = SP_{pre} - 1$
$M_{post}[SP_{post}] = M_{pre}[SP_{pre} - 1] < M_{pre}[SP_{pre}]$

Example

```
ldc 2
ldc 3
lt
brf FalseAction
```

B.36 General: markpointer

The Mark Pointer (MP) is used to access local variables, allocated on the stack. Each variable is accessed using a displacement relative to the MP.

B.37 General: memory

Memory stores words. A word is an 32 bits integer. Currently only a limited amount of memory words is reserver (2000), this is rather arbitrary, in the future memory size will adapt automatically to the amount needed.

B.38 Semantics: mod

Instruction	Nr of inline Opnds	Nr of stack Opnds	Nr of stack Results	Instr code (hex)
mod	0	2	1	0x7

Description Division. Replaces 2 top stack values with the modulo of those values.

Pre and Post State
$SP_{post} = SP_{pre} - 1$
$M_{post}[SP_{post}] = M_{pre}[SP_{pre} - 1] \% M_{pre}[SP_{pre}]$

Example

```
ldl -2 ; x = x % y
ldl -3
mod
stl -2
```

B.39 General: MP

The Mark Pointer (MP) is used to access local variables, allocated on the stack. Each variable is accessed using a displacement relative to the MP.

B.40 Semantics: mul

Instruction	Nr of inline Opnds	Nr of stack Opnds	Nr of stack Results	Instr code (hex)
mul	0	2	1	0x8

Description Multiplication. Replaces 2 top stack values with the multiplication of those values.

Pre and Post State $SP_{post} = SP_{pre} - 1$
$M_{post}[SP_{post}] = M_{pre}[SP_{pre} - 1] * M_{pre}[SP_{pre}]$

Example
```
No info
```

B.41 Semantics: ne

Instruction	Nr of inline Opnds	Nr of stack Opnds	Nr of stack Results	Instr code (hex)
ne	0	2	1	0xf

Description Test for not equal. Replaces 2 top stack values with boolean result of the test. False is encoded as 0, True as 1. Used in combination with brf. This is a variant of cmp combined with bne.

Pre and Post State $SP_{post} = SP_{pre} - 1$
$M_{post}[SP_{post}] = M_{pre}[SP_{pre} - 1]! = M_{pre}[SP_{pre}]$

Example
```
ldc 2
ldc 3
ne
brf FalseAction
```

B.42 Semantics: neg

Instruction	Nr of inline Opnds	Nr of stack Opnds	Nr of stack Results	Instr code (hex)
neg	0	1	1	0x20

Description Negation. Replaces top stack values with the (integer) negative of the value.

Pre and Post State $SP_{post} = SP_{pre}$
$M_{post}[SP_{post}] = -M_{pre}[SP_{pre}]$

Example
```
ldc 1 ; variant of ldc -1
neg
```

B.43 Semantics: nop

Instruction	Nr of inline Opnds	Nr of stack Opnds	Nr of stack Results	Instr code (hex)
nop	0	0	0	0xa4

Description No operation. Well, guess what...

Pre and Post State

Example
```
nop
```

B.44 Semantics: not

Instruction	Nr of inline Opnds	Nr of stack Opnds	Nr of stack Results	Instr code (hex)
not	0	1	1	0x21

Description Not. Replaces top stack values with the bitwise complement of the value.

Pre and Post State $SP_{post} = SP_{pre}$
$M_{post}[SP_{post}] = M_{pre}[SP_{pre}]$

Example

```
ldc 0x0000FFFF ; variant of ldc 0xFFFF0000
not
```

B.45 Semantics: or

Instruction	Nr of inline Opnds	Nr of stack Opnds	Nr of stack Results	Instr code (hex)
or	0	2	1	0x9

Description Or. Replaces 2 top stack values with the bitwise or of those values.

Pre and Post State $SP_{post} = SP_{pre} - 1$
$M_{post}[SP_{post}] = M_{pre}[SP_{pre} - 1] | M_{pre}[SP_{pre}]$

Example

```
ldc 0xFF00 ; variant of ldc 0xFFFF
ldc 0xF0F0
or
```

B.46 General: PC

The Program Counter (PC) is used to remember what the current next instruction is. It contains the address (i.e. points to) the location of the next instruction. The machine fetches an instruction from the location pointed to by the PC. After each fetch it is automatically updated to point to the next instruction.

B.47 General: programcounter

The Program Counter (PC) is used to remember what the current next instruction is. It contains the address (i.e. points to) the location of the next instruction. The machine fetches an instruction from the location pointed to by the PC. After each fetch it is automatically updated to point to the next instruction.

B.48 General: registers

Eight registers are available, some of which have a specific purpose. A register is private location in a processor, often faster accessible then external memory. Currently the Program Counter (PC), Stack Pointer (SP), Mark Pointer (MP) and Return Register (RR) as well as freely usable scratch registers are available, respectively identified by numbers 0..7. Registers are identified by the name R¡x¿, where ¡x¿ is the register number. Register with a specific purpose are also named with the name indicating their purpose.

B.49 Semantics: ret

Instruction	Nr of inline Opnds	Nr of stack Opnds	Nr of stack Results	Instr code (hex)
ret	0	1	0	0xa8

Description Return from subroutine. Pops a previously pushed PC from the stack and jumps to it.

Pre and Post State $SP_{post} = SP_{pre} - 1$
$PC_{post} = M_{pre}[SP_{pre}]$

Example

```
bra main
subroutine ldc 1
ldc 2
add
str RR
ret
main: bsr subroutine
ldr RR
...
```

B.50 General: return

register=@RR

B.51 General: RR

The Return Register (RR) is used to return a value without placing it on a stack. Strictly seen this is not necessary but a convenience, since values also can be passed via the stack.

B.52 General: SP

The Stack Pointer (SP) is used to push and pop values for usage in expression evaluation. The Stack is also used to store variables. These are often accessed via the MP.

B.53 Semantics: sta

Instruction	Nr of inline Opnds	Nr of stack Opnds	Nr of stack Results	Instr code (hex)
sta	1	2	0	0xac

Description Store via Address. Pops 2 values from the stack and stores the second popped value in the location pointed to by the first. The pointer value is offset by a constant offset.

Pre and Post State $SP_{post} = SP_{pre} - 2$
$M_{post}[M_{pre}[SP_{pre}] + M_{pre}[PC_{pre} + 1]] = M_{pre}[SP_{pre} - 1]$

Example

```
ldla -2 ; update local using its address
ldc 5
sta 0
```

B.54 General: stack

Stack is the part of memory used to store values needed for evaluating expressions. The stack is located after the code and grows from lower addresses to higher ones.

B.55 General: stackpointer

The Stack Pointer (SP) is used to push and pop values for usage in expression evaluation. The Stack is also used to store variables. These are often accessed via the MP.

B.56 Semantics: stl

Instruction	Nr of inline Opnds	Nr of stack Opnds	Nr of stack Results	Instr code (hex)
stl	1	1	0	0xb0

Description Store Local. Pops a value from the stack and stores it in a location relative to the markpointer.

Pre and Post State

$SP_{post} = SP_{pre} - 1$
$M_{post}[MP_{pre} + M_{pre}[PC_{pre} + 1]] = M_{pre}[SP_{pre}]$

Example

```
ldl 2 ; increment local var
ldc 1
add
stl 2
```

B.57 Semantics: stma

Instruction	Nr of inline Opnds	Nr of stack Opnds	Nr of stack Results	Instr code (hex)
stma	2	instruction dependent	0	0xae

Description Store Multiple via Address. Pops values from the stack and stores it in a location relative to the value at the top of the stack. Same as single store variant but second inline parameter is size.

Pre and Post State

$displ = M_{pre}[PC_{pre} + 1]$
$size = M_{pre}[PC_{pre} + 2]$
$SP_{post} = SP_{pre} - size - 1$
$M_{post}[M_{pre}[SP_{pre}] + displ..M_{pre}[SP_{pre}] + displ + size - 1] = M_{pre}[SP_{post} + 1..SP_{post} + size]$

Example

```
none
```

B.58 Semantics: stml

Instruction	Nr of inline Opnds	Nr of stack Opnds	Nr of stack Results	Instr code (hex)
stml	2	instruction dependent	0	0xb2

Description Store Multiple Local. Pops values from the stack and stores it in a location relative to the markpointer. Same as single store variant but second inline parameter is size.

Pre and Post State
$displ = M_{pre}[PC_{pre} + 1]$
$size = M_{pre}[PC_{pre} + 2]$
$SP_{post} = SP_{pre} - size$
$M_{post}[MP_{pre} + displ..MP_{pre} + displ + size - 1] = M_{pre}[SP_{post} + 1..SP_{post} + size]$

Example

```
ldl 2 ; increment local var
ldc 1
add
stml 2 1 ; equivalent to stl 2
```

B.59 Semantics: stms

Instruction	Nr of inline Opnds	Nr of stack Opnds	Nr of stack Results	Instr code (hex)
stms	2	instruction dependent	0	0xba

Description Store Multiple into Stack. Pops values from the stack and stores it in a location relative to the top of the stack. Same as single store variant but second inline parameter is size.

Pre and Post State
$displ = M_{pre}[PC_{pre} + 1]$
$size = M_{pre}[PC_{pre} + 2]$
$SP_{post} = SP_{pre} - size$
$M_{post}[SP_{pre} + displ..SP_{pre} + displ + size - 1] = M_{pre}[SP_{post} + 1..SP_{post} + size]$

Example

```
lds -1 ; substract and store in stack
ldc 2
sub
stms -2 1 ; equivalent to sts -2
```

B.60 Semantics: str

Instruction	Nr of inline Opnds	Nr of stack Opnds	Nr of stack Results	Instr code (hex)
str	1	1	0	0xb4

Description Store Register. Pops a value from the stack and stores it in a location relative to the markpointer. See also ldr.

Pre and Post State
$SP_{post} = SP_{pre} - 1$
$REG_{post}[M_{pre}[PC_{pre} + 1]] = M_{pre}[SP_{pre}]$

Example

```
ldr RR ; decrement register
ldc 1
sub
str RR
```

B.61 Semantics: sts

Instruction	Nr of inline Opnds	Nr of stack Opnds	Nr of stack Results	Instr code (hex)
sts	1	1	0	0xb8

Description Store into Stack. Pops a value from the stack and stores it in a location relative to the top of the stack.

Pre and Post State
$SP_{post} = SP_{pre} - 1$
$M_{post}[SP_{pre} + M_{pre}[PC_{pre} + 1]] = M_{pre}[SP_{pre}]$

Example

```
lds -1 ; substract and store in stack
ldc 2
sub
sts -2
```

B.62 Semantics: sub

Instruction	Nr of inline Opnds	Nr of stack Opnds	Nr of stack Results	Instr code (hex)
sub	0	2	1	0xc

Description Substraction. Replaces 2 top stack values with the subtraction of those values.

Pre and Post State
$SP_{post} = SP_{pre} - 1$
$M_{post}[SP_{post}] = M_{pre}[SP_{pre} - 1] - M_{pre}[SP_{pre}]$

Example

```
No info
```

B.63 Semantics: swp

Instruction	Nr of inline Opnds	Nr of stack Opnds	Nr of stack Results	Instr code (hex)
swp	0	2	2	0xbc

Description Swap values. Swaps the 2 topmost values on the stack.

Pre and Post State
$SP_{post} = SP_{pre}$
$M_{post}[SP_{post}] = M_{pre}[SP_{pre} - 1]$
$M_{post}[SP_{post} - 1] = M_{pre}[SP_{pre}]$

Example

```
ldc 1 ; variant for ldc 2 followed by ldc 1
ldc 2
swp
```

B.64 Semantics: swpr

Instruction	Nr of inline Opnds	Nr of stack Opnds	Nr of stack Results	Instr code (hex)
swpr	1	1	1	0xc0

Description Swap Register. Swaps the content of a register with the top of the stack.

Pre and Post State
$SP_{post} = SP_{pre}$
$M_{post}[SP_{post}] = REG_{pre}[M_{pre}[PC_{pre} + 1]]$
$REG_{post}[M_{pre}[PC_{pre} + 1]] = M_{pre}[SP_{pre}]$

Example

B.65 Semantics: swprr

Instruction	Nr of inline Opnds	Nr of stack Opnds	Nr of stack Results	Instr code (hex)
swprr	2	0	0	0xc4

Description Swap 2 Registers. Swaps the content of a register with another register.

Pre and Post State

$$REG_{post}[M_{pre}[PC_{pre}+1]] = REG_{pre}[M_{pre}[PC_{pre}+2]]$$
$$REG_{post}[M_{pre}[PC_{pre}+2]] = REG_{pre}[M_{pre}[PC_{pre}+1]]$$

Example

```
swprr MP R7 ; swap MP with scratch register
```

B.66 General: syntax

Syntax of instructions (as loaded from file) is: (label:)? (instr arg*)?. In other words, an (optional) instruction preceded by an (optional) label and followed by an argument if required. Comment may start with ";" or "//" (Java/C++ style) and ends at the end of the line. This characters are interpreted as start of comment. A label may be used as an argument. Example: "l1: beq l1 ; comment".

B.67 Semantics: trap

Instruction	Nr of inline Opnds	Nr of stack Opnds	Nr of stack Results	Instr code (hex)
trap	1	instruction dependent	instruction dependent	0xc8

Description Trap to environment function. Trap invokes a systemcall, which one is determined by its argument. Currently just 1 call exists, print the topmost element on the stack as an integer in the output window.

Pre and Post State

Example

```
ldc 5
trap 0 ; print 5 on output
```

B.68 General: True

Value True is encoded by a -1 (all 1 bit pattern 0xFFFFFFFF). However, when testing in the context of a BRF instruction takes place, anything else than 0 is considered to be True.

B.69 Semantics: unlink

Instruction	Nr of inline Opnds	Nr of stack Opnds	Nr of stack Results	Instr code (hex)
unlink	0	instruction dependent	0	0xcc

Description Free memory for locals. Convenience instruction combining the push of MP and the adjustment of the SP.

Pre and Post State

$MP_{post} = M_{pre}[MP_{pre}]$
$SP_{post} = MP_{pre} - 1$

Example

```
bra main
subroutine link 2 ; reserve for 2 locals
ldc 1
ldc 2
add
stl 1 ; store in 2nd local
ldl 1
str RR
unlink
ret
main: bsr subroutine
ldr RR
...
```

B.70 Semantics: xor

Instruction	Nr of inline Opnds	Nr of stack Opnds	Nr of stack Results	Instr code (hex)
xor	0	2	1	0xd

Description Exclusive Or. Replaces 2 top stack values with the bitwise exclusive or of those values.

Pre and Post State

$SP_{post} = SP_{pre} - 1$
$M_{post}[SP_{post}] = M_{pre}[SP_{pre} - 1]^{M}_{pre}[SP_{pre}]$

Example

```
ldc 0xFF00 ; variant of ldc 0x0FF0
ldc 0xF0F0
xor
```

Bibliography

[1] American National Standard for Programming Language Common LISP. ANSI/NCITS X3.226-1994, 1994.

[2] Scheme. `http://www.schemers.org/`, 2001.

[3] Martin Abadi and Luca Cardelli. *A Theory of Objects*. Springer, 1996.

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

[5] Andrew W. Appel. *Modern Compiler Implementation in Java, basic techniques*. Cambridge Univ. Press, 1997.

[6] Ken Arnold and James Gosling. *The Java Programming Language*. Addison-Wesley, 1996.

[7] F.L. Bauer (editor) and J. Eickel. *Compiler Construction, An Advanced Course*. Springer, 1974.

[8] Richard Bird. *Introduction to Functional Programming using Haskell*. Prentice Hall, 1998.

[9] Ras Bodik. CS 536: Introduction to Programming Languages and Compilers - Spring 2001. `http://www.cs.wisc.edu/~bodik/cs536.html`, 2001.

[10] Luca Cardelli. Typeful programming. In *Formal Description of Programming Concepts*, IFIP State of the Art Reports Series. Springer-Verlag, 1989.

[11] Luca Cardelli. *Type systems*, pages 2208–2236. CRC Press, 1997.

[12] Luis Damas and Robin Milner. Principal type-schemes for functional programs. In *Proceedings of Principles of Programming Languages (POPL)*, pages 207–212. ACM, ACM, 1982.

[13] Atze Dijkstra. Simple Stack Machine (SSM). `http://www.cs.uu.nl/~atze/SSM/index.html`, 2001.

[14] R. Kent Dybvig. *The Scheme Programming Language*. Prentice Hall, 1996.

[15] R. Matthew Emerson. The Association of Lisp Users. `http://www.lisp.org/`, 2001.

[16] Dick Grune, Henri E. Bal, J.H. Jacobs, and Koen G. Langendoen. *Modern Compiler Design*. Wiley, 2000.

[17] Samuel P. Harbison and Guy L. Steele Jr. *C, A Reference Manual*. Prentice Hall, 1991.

[18] Johan T. Jeuring and S. Doaitse Swierstra. *Grammars and parsing. Lecture Notes.* Utrecht University, Institute of Information and Computing Sciences, 2000.

[19] Thomas Johnsson. Attribute Grammars as a Functional Programming Paradigm. In *Functional Programming Languages and Computer Architecture*, LNCS, pages 154–173, sep 1987.

[20] Mark P. Jones. Typing Haskell in Haskell. In *Haskell Workshop*, 1999.

[21] Richard Jones and Rafael Lins. *Garbage Collection, Algorithms for Automatic Dynamic Memory Management*. Wiley, 1996.

[22] R. Kelsey (eds), W. Clinger, and J. Rees. Revised 5th Report on the Algorithmic Language Scheme. *ACM SIGPLAN Notices*, 33, Sep 1998.

[23] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language*. Prentice Hall, 1978.

[24] M.F. Kuiper and S.D. Swierstra. Using Attribute Grammars to Derive Efficient Functional Programs. In *Computing Science in the Netherlands CSN'87*, nov 1987.

[25] Cathy May (editor) and et. al. *The PowerPC Architecture: A Specification for a New Family of RISC processors*. Morgan Kaufman Publishers, 1994.

[26] Simon L. Peyton Jones. *The Implementation of Functional Programming Languages*. Prentice Hall, 1987.

[27] Simon Peyton Jones (editor) and John Hughes (editor). Report on the Programming Language Haskell 98. A Non-strict, Purely Functional Language, 1999.

[28] Guy L. Steele, Jr. *Common LISP : The Language*. Digital Press, 1990.

[29] Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley, 1993.

[30] S.D. Swierstra. Parser Combinators: from Toys to Tools. In *Haskell Workshop*, 2000.

[31] S.D. Swierstra and P.R. Azero Alcocer. Fast, error correcting parser combinators: A short tutorial. In *SOFSEM'99, 26th Seminar on Current Trends in Theory and Practice of Informatics*, pages 111–129, November 1999.

[32] Simon Thompson. *Type Theory and Functional Programming*. Addison-Wesley, 1991.

[33] William M. Waite and Gerhard Goos. *Compiler Construction*. Springer-Verlag, 1984.

[34] Reinhard Wilhelm and Dieter Maurer. *Compiler Design*. Addison-Wesley, 1995.

[35] H. Zantema and P.W.H Lemmens. *Beschrijven en Bewijzen*. Delft University Press, 1999.

Index