Universiteit Utrecht

[Faculty of Science
Information and Computing Sciences]

Compiler Construction

WWW: `http://www.cs.uu.nl/wiki/Cco`

Edition 2011/2012

Agenda

- The structure of a compiler
  - Trees
  - Running a compiler
  - Trees and ATerms
  - Pretty printing
  - Trees and ATerms

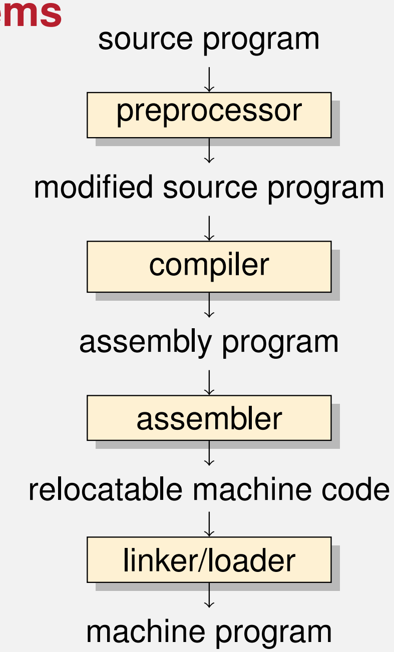
2. The structure of a compiler

Language-processing systems

§2

Typically, compilers are parts of larger systems.

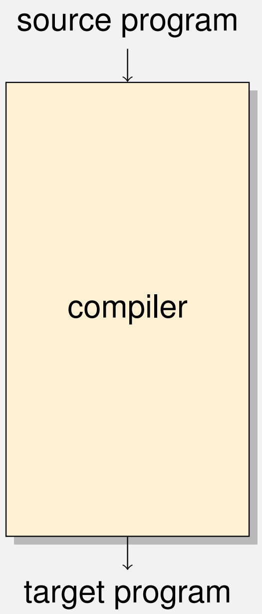
For example, for the generation of machine-executable code from a source program written in a high-level programming langauge, besides a compiler, several other programs may be involved: e.g., a preprocessor, an assembler, and/or a linker.



Black-box view

As a whole, a compiler performs a meaning-preserving translation from its source language into its target language.

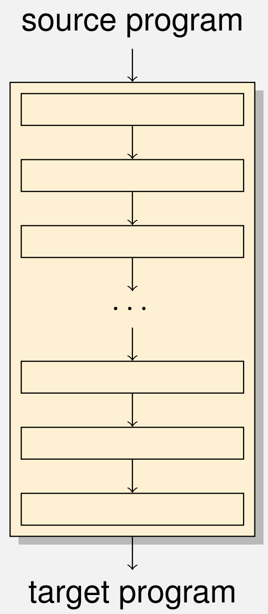
Optionally, it also validates the source program, reporting (possible) errors to the user, and/or produces a log of its activities.



White-box view

Internally, a compiler is typically implemented as a pipeline of components.

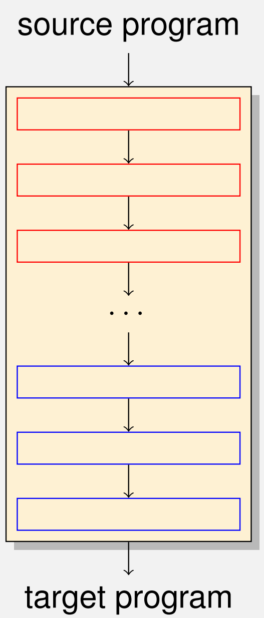
Each component in the pipeline takes as input the output of its predecessor and produces as output the input for its successor.



Analysis and synthesis

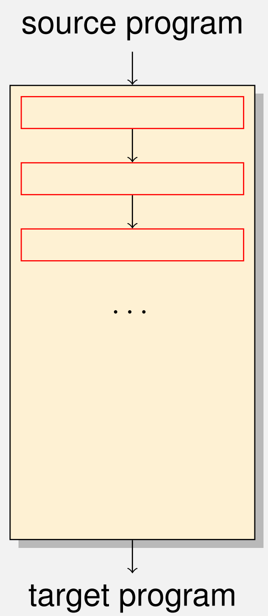
The first components in the pipeline form the front end of the compiler and perform the analysis of the source program.

The last components form the back end and take care of the synthesis of the target program.



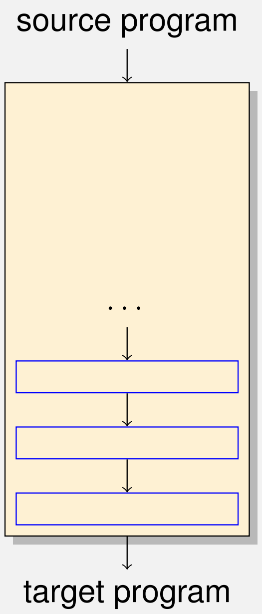
Front end

- Breaks up the source program into its constituent pieces.
- Imposes a grammatical structure on it.
- Checks for syntactic and semantic errors.
- Produces informative warnings and error messages.
- Constructs a symbol table.
- Creates an intermediate representation.



Back end

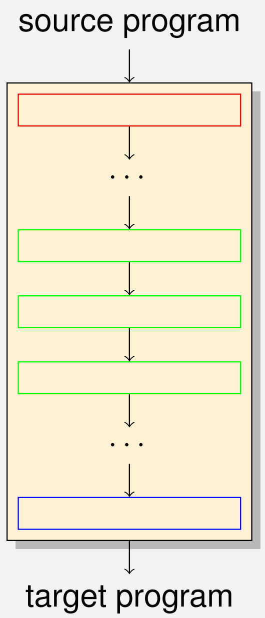
- Constructs the target program from the intermediate representation and the information stored in the symbol table.



Middle end

Sometimes we separate out a middle end in which optimisations on intermediate representations of the program have place.

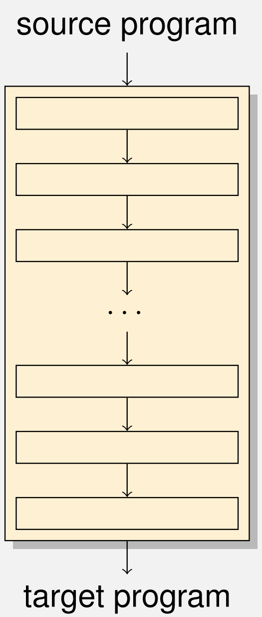
Ideally, the middle end operates on an intermediate representation that is completely independent from the source and target languages. This way, we can easily combine different front and back ends.



Phases

Logically, the compilation pipeline consists of a sequence of phases. In each phase, one representation of the source program is transformed into another.

The logical division of the compilation process into phases is not necessarily matched by the actual implementation. In practice, several phases are grouped together or split into subphases. Furthermore, some intermediate representations may not constructed explicitly.



Phases: example

Typical phases in the compilation of a high-level program into machine-executable code are:

- Lexical analysis.
- Syntactic analysis.
- Semantic analysis.
- Intermediate-code generation.
- Intermediate-code optimisation.
- Code generation.

Symbol tables

Symbol tables are data structures that contain incrementally obtained information about the source program. This information is typically produced in the front end of the compiler and consumed in the back end.

For example, the symbol table may contain information about identifiers such as their position in the source program and their type.

The symbol table may be shared by the different components that make up a compiler. Alternatively, it can be passed around by the compiler components.

In practice, the symbol table is not really a single table, but a compound data structure consisting of multiple tables, that may each contain subtables and so on.

Loose coupling

From a software engineer's point of view it may be advantageous to construct the components of a compiler as loosely coupled as possible.

Such modularity keeps the overall design comprehensible and makes it possible to debug and test components in isolation.

Ultimately, each component can exist as a stand-alone executable program, taking its inputs and producing its outputs from and to the command line or a file. Components can then be composed at the command line to form (sub)compilers.

Trees and strings

Internally, the intermediate representations of the source program typically take the form of trees.

Interacting with their environments, however, compilers consume and produce "flat" data streams, i.e., strings of characters or bytes.

Implementing a compiler, we therefore have to be able to convert between flat data and trees, and to transform one type of trees into another.

2.1 Trees

Algebraic data types

Internally, a compiler passes around tree-shaped representations of the source program.

In Haskell, such tree-structured data is typically represented in terms of algebraic data types.

For example:

$$\begin{aligned}\textbf{data}\ \mathit{Exp} &= \mathit{Const}\ \mathit{Int}\\ &\mid \mathit{Add}\ \mathit{Exp}\ \mathit{Exp}\\ &\mid \mathit{Mul}\ \mathit{Exp}\ \mathit{Exp}\end{aligned}$$

☞ Recall: a declaration of an algebraic data type introduces both a type constructor (*Exp*) and a family of data constructors (*Const*, *Add*, and *Mul*).

A need for flat representations

But if we want to encapsulate each compiler component in its own executable program, we need to pass tree-shaped data between programs.

Hence, we need to be able to convert between tree-shaped and “flat”, textual representations of trees, so that we can read and write tree-structured data from and to files and terminals.

Read and Show

One possibility is to rely on implementations of Haskell’s *Read* and *Show* classes:

$$\begin{aligned}&\textbf{class}\ \mathit{Read}\ \alpha\ \textbf{where}\\ &\quad \mathit{read} :: \mathit{String} \to \alpha\\ &\textbf{class}\ \mathit{Show}\ \alpha\ \textbf{where}\\ &\quad \mathit{show} :: \alpha \to \mathit{String}\end{aligned}$$

☞ Type classes are groups of types that share some common functionality.

Read and Show: deriving instances

Read and *Show* are so-called derivable type classes: for a large set of programmer-defined data types, a Haskell compiler can automatically derive instances of these classes:

$$\begin{aligned}\textbf{data}\ \mathit{Exp} &= \mathit{Const}\ \mathit{Int}\\ &\mid \mathit{Add}\ \mathit{Exp}\ \mathit{Exp}\\ &\mid \mathit{Mul}\ \mathit{Exp}\ \mathit{Exp}\\ &\quad \textbf{deriving}\ (\mathit{Read}, \mathit{Show})\end{aligned}$$

Read and Show: examples


```
*Main> Const 2 ‘Add‘ Const 3
Add (Const 2) (Const 3)

*Main> show (Const 2 ‘Add‘ Const 3)
"Add (Const 2) (Const 3)"

*Main> read ("Add (Const 2) (Const 3)") :: Exp
Add (Const 2) (Const 3)
```

☞ Why do we need the explicit type annotation in *read* (`"Add (Const 2) (Const 3)"`) :: *Exp*?

Read and Show: assessment

Pro:

- **Easy implementable:** for most data types, *Read* and *Show* can be derived by the Haskell compiler.

Cons:

- **Haskell-centric:** the format on which *Read* and *Show* operate is essentially the Haskell syntax for constructor application. If we consider exchange between components with different implementation languages, there may be formats that are better supported across different programming languages.
- **Single-line output:** derived implementations of *show* produce their output on a single line and, hence, the display of large trees appears quite chaotic. Inspecting the output of a compiler component, it may be hard to recognise the tree structure.

Alternatives

As an alternative to *Read* and *Show* we may consider a more widespread format for representing tree-shaped structures, such as XML or ATerm.

XML is ubiquitous and has excellent tool support. However, the format is quite verbose.

The ATerm format is perhaps less known, but it is specifically targeted at compilers and has fairly good tool support.

The ATerm format

Annotated Term format: a structured representation of arbitrary tree-shaped data.

See:

- Mark van den Brand, Hayco de Jong, Paul Klint, and Pieter A. Olivier. Efficient annotated terms. *Software—Practice and Experience (SPE)*, 30(3):259–291, 2000.
- `http://www.meta-environment.org/Meta-Environment/ATerms`.

As a means of representing trees, ATerms are, in a sense, comparable to both XML and algebraic data types.

The ATerm format: structure

An ATerm $\mathcal{A}$ can be:

- An **integer** constant: $2, 3, \ldots$
- A **floating-point** constant: $3.14, \ldots$
- A **string** constant: "x", "abc", "", ...
- A **constructor application** $C(\mathcal{A}_1, \cdots, \mathcal{A}_n)$: $Const(2)$, $Ident(\texttt{"x"})$, $Pos(1,1)$, EOF, ...
- A **tuple** $(\mathcal{A}_1, \cdots, \mathcal{A}_n)$: $(Ident(\texttt{"x"}), Op(\texttt{"*"}))$, ...
- A **list** $[\mathcal{A}_1, \cdots, \mathcal{A}_n]$: $[Const(2), Op(\texttt{"+"})]$, ...
- An **annotated term** $\mathcal{A}\,\{\mathcal{A}_1, \cdots, \mathcal{A}_n\}$: $Ident(\texttt{"x"})\,\{Pos(1,9)\}$, ...

The ATerm format: examples

ATerm for $Const\ 2$:

$Const(2)$

ATerm for $Const\ 2$ ‘Add‘ $Const\ 3$:

$Add(Const(2), Const(3))$

ATerm for $Const\ 2$ ‘Add‘ $(Const\ 3$ ‘Mul‘ $Const\ 5)$:

$Add(Const(2)$
$\quad, Mul(Const(3), Const(5)$
$\quad)\,\{size(5), depth(3), value(17)\}$

Representing ATerms in Haskell

The CCO library exposes a module *CCO.Tree* that contains an algebraic data type for representing ATerms:

```
data ATerm = Integer Integer
           | Float Double
           | String String
           | App Con [ATerm]
           | Tuple [ATerm]
           | List [ATerm]
           | Ann ATerm [ATerm]
           deriving (Eq, Read, Show)
type Con = String
```


A class for tree types

The class *Tree* contains types that can be represented as ATerms:

```
class Tree α where
  fromTree :: α → ATerm
  toTree   :: ATerm → α
```

The methods *fromTree* and *toTree* convert between trees and ATerms.

A class for tree types: example

To make *Exp* a member of the *Tree* class, we need to provide an instance declaration:

$$\textbf{instance}\ \mathit{Tree}\ \mathit{Exp}\ \textbf{where}$$
$$\quad \mathit{fromTree} = \cdots$$
$$\quad \mathit{toTree} \;\;\;= \cdots$$

A class for tree types: example (cont'd)

$$\begin{aligned}
&\mathit{fromTree}\ (\mathit{Const}\ n) \\
&\quad = \mathit{App}\ \texttt{"Const"}\ [\mathit{Integer}\ (\mathit{toInteger}\ n)] \\
&\mathit{fromTree}\ (\mathit{Add}\ e_1\ e_2) \\
&\quad = \mathit{App}\ \texttt{"Add"}\ [\mathit{fromTree}\ e_1, \mathit{fromTree}\ e_2] \\
&\mathit{fromTree}\ (\mathit{Mul}\ e_1\ e_2) \\
&\quad = \mathit{App}\ \texttt{"Mul"}\ [\mathit{fromTree}\ e_1, \mathit{fromTree}\ e_2]
\end{aligned}$$

Haskell integers are converted to ATerm integers.

Haskell constructor applications are converted to ATerm constructor applications.

A class for tree types: example (cont'd)

$$\begin{aligned}
&\mathit{toTree}\ (\mathit{App}\ \texttt{"Const"}\ [\mathit{Integer}\ n]) &&= \mathit{Const}\ (\mathit{fromInteger}\ n) \\
&\mathit{toTree}\ (\mathit{App}\ \texttt{"Add"}\ [a_1, a_2]) &&= \mathit{Add}\ (\mathit{toTree}\ a_1)\ (\mathit{toTree}\ a_2) \\
&\mathit{toTree}\ (\mathit{App}\ \texttt{"Mul"}\ [a_1, a_2]) &&= \mathit{Mul}\ (\mathit{toTree}\ a_1)\ (\mathit{toTree}\ a_2) \\
&\mathit{toTree}\ (\mathit{Ann}\ a\ _) &&= \mathit{toTree}\ a \\
&\mathit{toTree}\ _ &&= \mathit{error}\ \texttt{"toTree: ..."}
\end{aligned}$$

Converting from ATerms to tree types constitutes a partial function.

In this particular example, annotations are ignored.

2.2 Running a compiler

A feedback monad

For dealing with feedback it is relevant to virtually all parts of a compilation pipeline, we introduce a monad for feedback management:

```
data Feedback α   -- abstract, instance of Monad
```

☞ A monad is a data type μ for representing and constructing effectful computations, supporting the operations $return :: \alpha \to \mu\ \alpha$ and $(\gg\!=) :: \mu\ \alpha \to (\alpha \to \mu\ \beta) \to \mu\ \beta$ ("bind").

Issuing messages

Feedback is constructed from three types of messages: log messages, warning messages, and error messages.

Log and warning messages have, respectively, a verbosity and a severity level associated, that allows control over the amount of feedback that is generated.

```
trace :: Int → String → Feedback ()   -- for log messages
warn  :: Int → String → Feedback ()   -- for warning messages
fail  ::       String → Feedback α    -- for error messages
```

☞ Actually, $fail$ is just $fail :: Monad\ \mu \Rightarrow String \to \mu\ \alpha$ from the $Monad$ class.

Running a feedback computation

We can "run" a feedback computation and, as a side effect, have its messages written to some file-system object:

```
runFeedback :: Feedback α →   -- the computation
               Int        →   -- verbosity level
               Int        →   -- severity level
               Handle     →   -- object to write messages to
               IO (Maybe α)
```

Feedback computations fail if an error message has been issued.

☞ $Maybe$ is the $Prelude$ type of computations that may fail:

```
data Maybe α = Nothing   -- for failure
             | Just α    -- for succes
```


Example of a feedback computation


```
divider :: Int → Int → Feedback Int
divider m n = do trace 2 "Start divider ..."
                 if n ≡ 0 then fail "Error: division by zero!"
                          else do trace 1 "Dividing ..."
                                  return (m `div` n)

runDivider :: Int → Int → Int → IO ()
runDivider v m n = do let d = divider m n
                      result ← runFeedback d v 1 stderr
                      case result of
                        Nothing → return ()
                        Just k  → print k
```

Example of a feedback computation (cont'd)


```
*Divider> runDivider 2 8 4
Start divider ...
Dividing ...
2

*Divider> runDivider 1 8 4
Dividing ...
2

*Divider> runDivider 2 8 0
Start divider ...
Error: division by zero!
```


Components

Recall that a compiler is typically implemented as a pipeline of components.

Components inside the pipeline take as input the output of their predecessor and produce as output the input to their successor. The component at the beginning of a pipeline reads its input from a file or the command line, while the component at the end of a pipeline writes its output to a file or the command line.

To represent components that take inputs of type α and produce outputs of type β, the CCO library provides an abstract type constructor $Component$:

$$\textbf{data}\ Component\ \alpha\ \beta \quad \text{-- abstract, instance of Arrow}$$

Arrows

Arrows are a generalisation of monads and provide a common interface to effectful computations:

$$\begin{aligned}
&\textbf{class}\ Arrow\ \varphi\ \textbf{where}\\
&\quad pure \quad :: (\alpha \to \beta) \to \varphi\ \alpha\ \beta\\
&\quad (\ggg) \quad :: \varphi\ \alpha\ \beta \to \varphi\ \beta\ \gamma \to \varphi\ \alpha\ \gamma\\
&\quad first \quad :: \varphi\ \alpha\ \beta \to \varphi\ (\alpha,\gamma)\ (\beta,\gamma)\\
&\quad second :: \varphi\ \alpha\ \beta \to \varphi\ (\gamma,\alpha)\ (\gamma,\beta)
\end{aligned}$$

See: John Hughes. Generalising monads to arrows. *Science of Computer Programming*, 37(1–3):67–111, 2000.

For $Component$, we have:

$$\begin{aligned}
&pure \quad :: (\alpha \to \beta) \to Component\ \alpha\ \beta\\
&(\ggg) \quad :: Component\ \alpha\ \beta \to Component\ \beta\ \gamma \to Component\ \alpha\ \gamma\\
&first \quad :: Component\ \alpha\ \beta \to Component\ (\alpha,\gamma)\ (\beta,\gamma)\\
&second :: Component\ \alpha\ \beta \to Component\ (\gamma,\alpha)\ (\gamma,\beta)
\end{aligned}$$

Creating components

Primitive components can be created either from pure computations,

$$pure :: (\alpha \to \beta) \to Component\ \alpha\ \beta$$

or from computations that involve $Feedback$:

$$component :: (\alpha \to Feedback\ \beta) \to Component\ \alpha\ \beta$$

Creating components: example

Assume we have to construct a compilation pipeline that

- Reads a floating-point value from the standard input.
- Checks that the value is nonnegative.
- Calculates the square root of the value.
- Writes the square root to the standard output.

Creating components: example (cont'd)

To read, we create a *Component* that consumes a *String* and produces a *Double*. If the *String* cannot be parsed into a *Double*, we issue an error message.

```
parser :: Component String Double
parser = component $ λinput → do
           trace_ "Parsing ..."
           case [r | (r, _) ← reads input] of
             r : _ → return r
             _     → fail "Parse error!"
```

☞ The function $trace\_$ is defined as $trace\_ = trace\ 1$.

☞ For the actual parsing, we instantiate the list-of-successes parser $reads :: Read\ \alpha \Rightarrow String \to [(\alpha, String)]$ for *Double*.

Creating components: example (cont'd)

Next, we define a *Component* that checks whether its *Double*-input is nonnegative. If the check fails, a warning message is emitted. The component always returns its input unmodified.

```
validator :: Component Double Double
validator = component $ λr → do
              trace_ "Validating ..."
              when (r < 0) (warn_ "Warning: negative input!")
              return r
```

☞ The function $warn\_$ is defined as $warn\_ = warn\ 1$.

Creating components: example (cont'd)

The actual calculation is performed by a *Component* that simply applies the *Prelude* function $sqrt$ to its input:

```
calculator :: Component Double Double
calculator = component $ λr → do
               trace_ "Calculating ..."
               return (sqrt r)
```

Creating component: example (cont'd)

The final *Component* in the pipeline uses the function *show* to turn a *Double* into a *String*.

$$
\begin{aligned}
&\mathit{printer} :: \mathit{Component}\ \mathit{Double}\ \mathit{String} \\
&\mathit{printer} = \mathit{component}\ \$\ \lambda r \to \mathbf{do} \\
&\qquad \mathit{trace\_}\ \texttt{"Printing ..."} \\
&\qquad \mathit{return}\ (\mathit{show}\ r)
\end{aligned}
$$

Composing components: example

To assemble a pipeline, we compose individual *Component*-values by means of the *Arrow*-combinator ($\ggg$):

$$
\begin{aligned}
&\mathit{pipeline} :: \mathit{Component}\ \mathit{String}\ \mathit{String} \\
&\mathit{pipeline} = \mathit{parser} \ggg \mathit{validator} \ggg \mathit{calculator} \ggg \mathit{printer}
\end{aligned}
$$

The pipeline itself constitutes a *Component* that takes *String*-inputs to *String*-outputs.

Wrapping components

Recall that a compilation pipeline, at its source and sink sides, is supposed to exchange flat data with file-system objects.

Indeed, if a *Component* consumes and produces flat data, i.e., *String*-values, it can be turned into a stand-alone program that reads from the standard input channel and writes to the standard output channel:

$$\mathit{ioWrap} :: \mathit{Component}\ \mathit{String}\ \mathit{String} \to \mathit{IO}\ ()$$

Wrapping components: example

For our square-root calculator with *pipeline* :: *Component String String*, we have:

$$
\begin{aligned}
&\mathit{main} :: \mathit{IO}\ () \\
&\mathit{main} = \mathit{ioWrap}\ \mathit{pipeline}
\end{aligned}
$$

☞ The *main* function is where a Haskell program starts its execution.

Wrapping components: example (cont'd)

At the command line:

```
% ls
Sqrt.hs

% ghc --make -o sqrt Sqrt.hs
[1 of 1] Compiling Main             ( Sqrt.hs, Sqrt.o )
Linking sqrt ...

% ls
Sqrt.hi Sqrt.hs Sqrt.o sqrt

% echo 25.0 | ./sqrt
Parsing ...
Validating ...
Calculating ...
Printing ...
5.0
```


Wrapping components: example (cont'd)


```
% echo xyz | ./sqrt
Parsing ...
Parse error!

% echo -3.14 | ./sqrt
Parsing ...
Validating ...
Warning: negative input!
Calculating ...
Printing ...
NaN
```


Wrapping components: example (cont'd)


```
% cat > in.dbl
49.0
^D

% ls
Sqrt.hi Sqrt.hs Sqrt.o in.dbl sqrt

% cat in.dbl | ./sqrt > out.dbl
Parsing ...
Validating ...
Calculating ...
Printing ...

% ls
Sqrt.hi Sqrt.hs Sqrt.o in.dbl out.dbl sqrt

% cat out.dbl
7.0
```


2.3 Trees and ATerms

ATerms

Recall that the ATerm format provides a generic representation for the tree-like structures that typically appear in the internals of the compilation pipeline.

In Haskell, we represented ATerms by

```
data ATerm = Integer Integer
           | Float Double
           | String String
           | App Con [ATerm]
           | Tuple [ATerm]
           | List [ATerm]
           | Ann ATerm [ATerm]
           deriving (Eq, Read, Show)
```


ATerms: example

As an example of an *ATerm*, consider

```
Mul(
  Add(Const(2), Mul(Const(3), Const(5)))
, Mul(Add(Const(7), Const(9)), Const(11))
)
```

represented in Haskell by

```
App "Mul" [
  App "Add" [
    App "Const" [Integer 2]
  , App "Mul" [App "Const" [Integer 3], App "Const" [Integer 5]]
  ]
, App "Mul" [
    App "Add" [App "Const" [Integer 7], App "Const" [Integer 9]]
  , App "Const" [Integer 11]
  ]
]
```


Converting between trees and ATerms

The class *Tree* contains types that can be represented as ATerms:

```
class Tree α where
  fromTree :: α → ATerm
  toTree   :: ATerm → α
```

toTree is a partial function: not every ATerm can be converted into a tree of the appropriate type.

Hence, we let *toTree* produce its result in the *Feedback* monad:

```
class Tree α where
  fromTree :: α → ATerm
  toTree   :: ATerm → Feedback α
```


2.4 Pretty printing

Displaying trees

Often it is necessary to display trees in a human-readable format; for instance, for testing or debugging purposes.

Rather than relying on a derived *Show* instance,

```
App "Mul" [App "Add" [App "Const" [Integer 2],Ap\
p "Mul" [App "Const" [Integer 3],App "Const" [In\
teger 5]]],App "Mul" [App "Add" [App "Const" [In\
teger 7],App "Const" [Integer 9]],App "Const" [I\
nteger 11]]]
```

we typically want to present the user with a representation in concrete syntax:

```
Mul(Add(Const(2), Mul(Const(3), Const(5))), Mul(\
Add(Const(7), Const(9)), Const(11)))
```


Displaying trees (cont'd)

Ideally, the concrete-syntax representation makes the tree structure explicit:

```
Mul(
  Add(Const(2), Mul(Const(3), Const(5)))
, Mul(Add(Const(7), Const(9)), Const(11))
)
```

To display a tree in a human-readable form so that the structure of the tree is easily perceivable, we employ a so-called pretty printer.

Pretty-printer combinators

The CCO library exposes a module *CCO.Printing* that provides a family of pretty-printer combinators.

These combinators can be used to construct and combine values of the abstract data type *Doc* of printable documents:

data *Doc* -- *abstract*

Primitive document constructors

The empty document:

empty :: *Doc*

A document containing a specified text:

text :: *String* → *Doc*

For example:

text "pretty"

yields

```
pretty
```

Stacking

The combinator (>-<) is used to place one document on top of another:

```
infixr 2 >-<
(>-<) :: Doc → Doc → Doc
```

For example:

```
text "pretty" >-< text "printing"
```

yields

```
pretty
printing
```


Identation

The function *indent* is used to indent a document by a given amount of spaces:

```
ident :: Int → Doc → Doc
```

For example:

```
text "pretty" >-< indent 2 (text "printing")
```

yields

```
pretty
  printing
```


Concatenation

The combinator (>|<) is used to concatenate two documents:

```
infixr 3 >|<
(>|<) :: Doc → Doc → Doc
```

For example:

```
text "pretty" >|< text "printing"
```

yields

```
prettyprinting
```


Dovetailing

If its first operand is a multiline document, (>|<) performs what is known as “dovetailing”.

For example:

```
(text "combinators" >-< text "for ") >|<
(text "pretty" >-< text "printing")
```

yields

```
combinators
for pretty
    printing
```

Parallelisation

The combinator (>//<) is used to introduce a choice between two alternative formattings. When a document is printed, the most space-efficient one is chosen.

```
infixr 1 >//<
(>//<) :: Doc → Doc → Doc
```

Parallelisation should only be used with care: nested uses of (>//<) can easiliy cause an explosion in the number of alternatives to consider. Therefore, a local choice can be enforced by means of the function *join*:

```
join :: Doc → Doc
```


Rendering

Documents are rendered by means of the function *render*:

```
render :: Int → Doc → Maybe String
```

render takes as its first argument the amount of horizontal space that is available for printing. It then constructs the most space-efficient rendering specified by its *Doc* argument that still fits the available space. If the document cannot be rendered within the available space, *Nothing* is returned.

As an alternative, *render_* always produces a rendering, enlarging the given amount of space as necessary:

```
render_ :: Int → Doc → String
```


2.5 Trees and ATerms

Exercise: utilities for turning ATerms into trees


```
class Tree α where
  fromTree :: α → ATerm
  toTree   :: ATerm → Feedback α
```

```
data Either α β = Left α | Right β
```

```
instance (Tree α, Tree β) ⇒ Tree (Either α β) where
  fromTree (Left x)  = App "Left"  [fromTree x]
  fromTree (Right y) = App "Right" [fromTree y]
  toTree = parseTree [app "Left"  (Left <$> arg)
                     , app "Right" (Right <$> arg)
                     ]
```

From ATerms to trees: example

$test :: String \to IO\ ()$
$test\ input = \mathbf{case}\ runFeedback\ feedback\ 1\ 1\ stderr\ \mathbf{of}$
$\quad Nothing \to return\ ()$
$\quad Just\ tree \to print\ tree$
$\mathbf{where}$
$\quad feedback :: Feedback\ (Either\ Bool\ Int)$
$\quad feedback = toTree\ (parseATerm\ input)$

From ATerms to trees: example (cont'd)


```
*Test> test "Left(False)"
Left False

*Test> test "InBetween(True)"
Error in ATerm.
*** Unexpected : InBetween
*** Expected   : Left or Right
*** In term    : InBetween(True)

*Test> test "Left(Perhaps)"
Error in ATerm.
*** Unexpected : Perhaps
*** Expected   : False or True
*** In term    : Perhaps
```


From ATerms to trees: example (cont'd)


```
*Test> test "Left()"
Error in ATerm: Left takes 1 argument, but none
were given.
*** In term : Left()

*Test> test "Left(False, True)"
Error in ATerm: Left takes 1 argument, but 2
were given.
*** In term : Left(False, True)
```


Implementation

318 lines of Haskell code:

- comment and whitespace: 161 lines
- generating error messages: 146 lines

Available from the CCO library through the *CCO.Tree.Parser* module.