McMaster University
Department of Computing and Software
Dr. W. Kahl

SFWR ENG 3E03
Exercise Sheet 3

Design and Selection of Programming Languages

27 September 2006

Simplified FunnelWeb

FunnelWeb is a literate programming tool, i.e., it supports production of **documents** that **also contain code**, and supports presentation of code in a sequence that is more appropriate for humans, while using macro-preprocessor-like functionality to arrange the "code chunks" in the right sequence for the target system of the code. The following example input file contains two *macro definitions* signalled by the *special sequence* "@$", and one *output file definition* (which can be seen as a special kind of macro definition) signalled by the special sequence "@O".

Inside the documentation, there can be references to code material, like the "@{bin@}" here in line 9; such references are delimited by the special sequences "@{" and "@}".

```
A frequently needed functionality is adding a
new first element to a search path environment variable:

@$@<addpath@>@(@2@)@M@{if [ -d @2 ]
then export @1=@2:$@1
fi@}

In many cases, the three standard paths for executables, man pages,
and (shared) libraries are extended with subdirectories @{bin@},
@{man@}, @{lib@} of a common installation prefix:

@$@<addpaths@>@(@1@)@M@{
if [ -d @1 ]
then
  @<addpath@>@(PATH@,@1/bin@)
  @<addpath@>@(MANPATH@,@1/man@)
  @<addpath@>@(LD_LIBRARY_PATH@,@1/lib@)
fi@}

We use this to create a file that will be ‘‘sourced’’
from our @{.bash_profile@}:

@O@<.bash_addpaths@>@{
@<addpaths@>@(/usr/local@)
@<addpaths@>@(/usr/local/packages/ghc-6.5@)
@}
```

For your information, the "literate documentation output" produced from this file by the original FunnelWeb is printed on page 2 (details of the graphical representation do not matter). An invocation of FunnelWeb on this example file will produce a file named .bash_addpaths; the contents of this is printed on 2, too.

In FunnelWeb, all *special sequences* start with the character "@" and have ***exactly one*** character after that; the special sequences for delimiting macro (and file) names are "@<" and "@>". For our simplified version, macro names consist of letters, digits, ".", "_", and spaces, but no newline characters.

Literate Documentation Output of FunnelWeb Example

A frequently needed functionality is adding a new first element to a search path environment variable:

addpath[1](◇2)**M** ≡
```
{if [ -d ◇2 ]
 then export ◇1=◇2:$◇1
 fi}
```
This macro is invoked in definitions 2, 2 and 2.

In many cases, the three standard paths for executables, man pages, and (shared) libraries are extended with subdirectories `bin`, `man`, `lib` of a common installation prefix:

addpaths[2](◇1)**M** ≡
```
{
 if [ -d ◇1 ]
 then
   addpath[1]('PATH','◇1/bin')
   addpath[1]('MANPATH','◇1/man')
   addpath[1]('LD_LIBRARY_PATH','◇1/lib')
 fi}
```
This macro is invoked in definitions 3 and 3.

We use this to create a file that will be "sourced" from our `.bash_profile`:

.bash_addpaths[3] ≡
```
{
 addpaths[2]('/usr/local')
 addpaths[2]('/usr/local/packages/ghc-6.3')
 }
```
This macro is attached to an output file.

FunnelWeb output file `.bash_addpaths`

```
if [ -d /usr/local ]
then
  if [ -d /usr/local/bin ]
  then export PATH=/usr/local/bin:$PATH
  fi
  if [ -d /usr/local/man ]
  then export MANPATH=/usr/local/man:$MANPATH
  fi
  if [ -d /usr/local/lib ]
  then export LD_LIBRARY_PATH=/usr/local/lib:$LD_LIBRARY_PATH
  fi
fi

if [ -d /usr/local/packages/ghc-6.5 ]
then
  if [ -d /usr/local/packages/ghc-6.5/bin ]
  then export PATH=/usr/local/packages/ghc-6.5/bin:$PATH
  fi
  if [ -d /usr/local/packages/ghc-6.5/man ]
  then export MANPATH=/usr/local/packages/ghc-6.5/man:$MANPATH
  fi
  if [ -d /usr/local/packages/ghc-6.5/lib ]
  then export LD_LIBRARY_PATH=/usr/local/packages/ghc-6.5/lib:$LD_LIBRARY_PATH
  fi
fi
```

(a) Give three regular expressions, one for each of the following three kinds of tokens:
- delimited macro names, as for example "`@<addpath@>`",
- special sequences occurring in the example, except "`@<`" and "`@>`", and
- text that does not contain any "`@`" characters.

(b) Produce
- bison token type declarations and
- a flex lexer

for the token classes described in (a), except that **newline** characters should now be treated **as separate tokens**.

The two macros in the example file above both have arguments, signalled by the "formal parameter header" "`@(@1@)`" for a macro with one argument and "`@(@2@)`" for a macro with two arguments; inside the macro body, "`@1`" stands for a reference to the first argument.

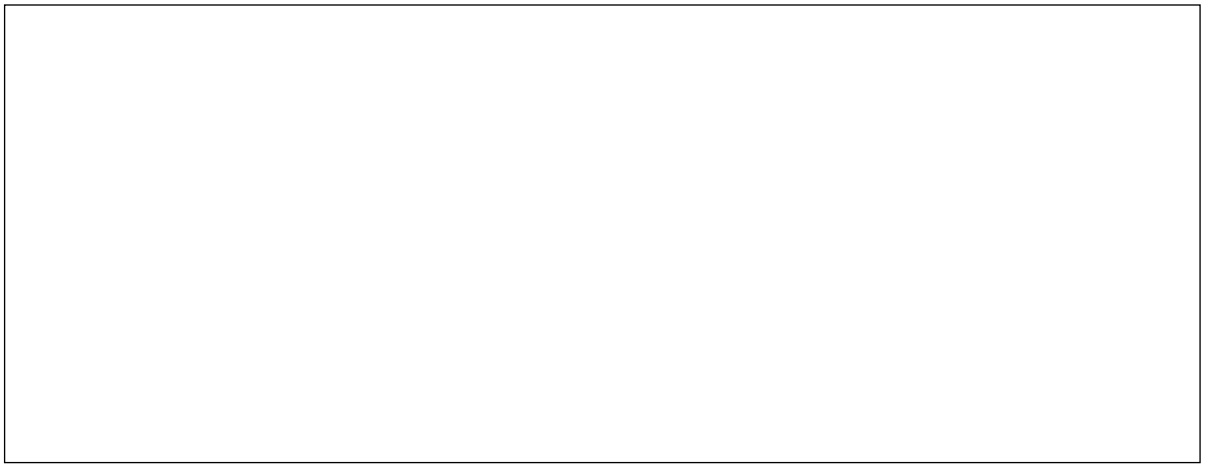
Code chunks can contain verbatim code, and also *macro invocations*, where macro names are again delimited by "`@<`" and "`@>`". Macros names in code chunks can be immediately followed by arguments in the shape of an *actual parameter list* delimited by "`@(`" and "`@)`"; if there are several arguments, these are separated by "`@,`" — all this is visible in the example. Each argument can again be an arbitrary code chunk.

Macro definition bodies are code chunks surrounded by the delimiters "`@{`" and "`@}`", just like code chunks embedded in documentation text.

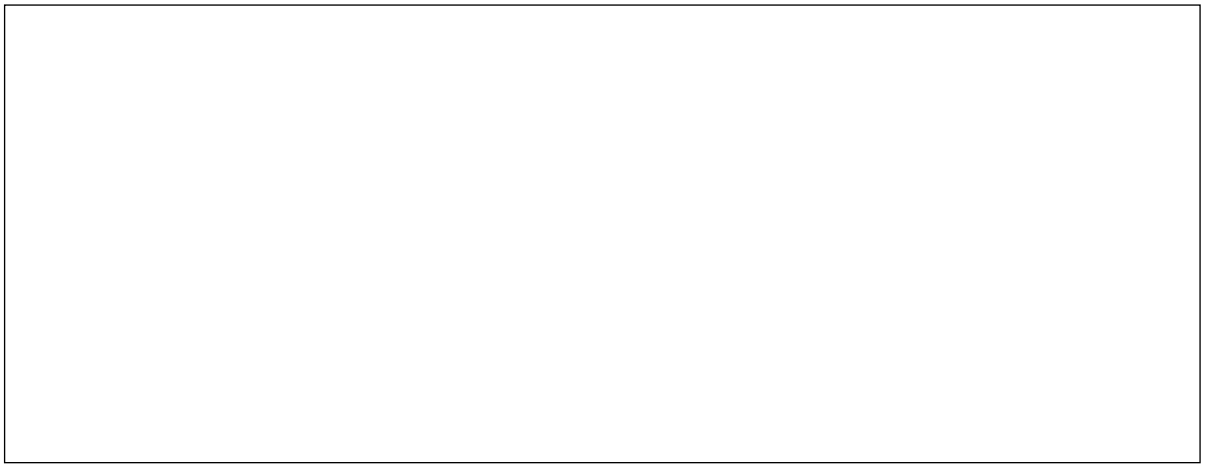
The line structure of code chunks is relevant since all the lines in each product instance of a code chunk inherit the indentation level of its macro invocation — this effect can be seen in the product example on 2.

The opening "`@{`" of a macro definition body can be preceded by the special sequence "`@M`"; this indicates that this macro can be used several times.

(c) Give a **concrete** grammar in bison notation (for the time being without actions) for the aspects of FunnelWeb explained above such that your grammar reflects these explanations and the usage in the example.



(d) Give an **abstract** grammar (for example in EBNF) for the aspects of FunnelWeb explained above.



(e) Does FunnelWeb require any consistency checks after parsing? (Think “scope” and “typing”…)

(f) In the context of the two macro definitions from the example above, produce a FunnelWeb output file definition that contains a macro invocation that can be understood to produce a **type error** — then **explain** what the type of a macro is and why your example produces a type error.

(g) Define Java classes to represent the abstract syntax of FunnelWeb files. (Use of library classes where appropriate is encouraged.)

(h) Define C datatypes to represent the abstract syntax of FunnelWeb files.

(i) Use the C datatypes to fill in the actions of the bison parser.

(j) The production of code (in this example of the file `.bash_addpaths` on 2) from literate program input is called "tangling". Produce a C function that takes the abstract syntax tree of a legal FunnelWeb file as argument and produces tangled output for all "`@O`" files mentioned in the input.

(k) Produce a FunnelWeb fragment that should allow you to test whether FunnelWeb performs static or dynamic type checking — **explain!**