Preparatory Course – Supplemental Test for Grammars and Parsing
Friday, January 5, 2007 (9:00-12:00)

This exam consists of 6 multiple-choice questions (Part I) and 4 open questions (Part II). The maximum number of points for each question is given (100 points in total). Give short and precise answers. Try to find elegant solutions, and provide type signatures for your auxiliary functions. You may use all functions from Haskell's standard libraries and the ParseLib module. It is not allowed to consult the lecture notes during the test. Good luck!

Part I: Multiple Choice Questions (30 points)

1 Ambiguity (5 points)

Which of the following statements is the *most* correct answer?

a) A language is ambiguous if every sentence has two or more parse trees

b) A language is unambiguous if every sentence has a unique left-most derivation

c) A context-free grammar is ambiguous if the right-hand side of one of its production rules starts with the non-terminal that appears on the left-hand side.

d) All three statements above are valid.

2 Regular grammars (5 points)

Which of the following grammars is regular and generates the same language as the regular expression $a*(\epsilon + b + cb*a)$?

a)
$S \rightarrow aS \mid \epsilon \mid A$
$A \rightarrow bB \mid cC$
$B \rightarrow \epsilon$
$C \rightarrow bC \mid a$

b)
$S \rightarrow aS \mid bA \mid cB$
$A \rightarrow a$
$B \rightarrow bB \mid a$

c)
$S \rightarrow aS \mid a \mid b \mid cA$
$A \rightarrow bA \mid a$

d)
$S \rightarrow AB$
$A \rightarrow aA \mid \epsilon$
$B \rightarrow \epsilon \mid b \mid cC$
$C \rightarrow bC \mid a$

3 Parser combinators (5 POINTS)

The function *listOf* combines two parsers: the new parser recognizes one or more items (first argument) that are separated (second argument). The type of this combinator is:

$$listOf :: Parser\ s\ a \rightarrow Parser\ s\ b \rightarrow Parser\ s\ [a]$$

Let *list* be defined as follows:

$$list\ x\ xs = x : xs$$

Which of the following definitions correctly implements *listOf*?

a) $listOf\ p\ s = list\ <\$>\ p\ <*>\ s\ <*>\ listOf\ p\ s\ <|>\ succeed\ [\,]$

b) $listOf\ p\ s = list\ <\$>\ p\ <*>\ many1\ ((\lambda x\ y \rightarrow y)\ <\$>\ s\ <*>\ p)$

c) $listOf\ p\ s = list\ <\$>\ p\ <*>\ option\ ((\lambda x\ y \rightarrow y)\ <\$>\ s\ <*>\ listOf\ p\ s)\ [\,]$

d) $listOf\ p\ s = list\ <\$>\ p\ <*>\ option\ ((\lambda x\ y \rightarrow y)\ <\$>\ s\ <*>\ p)\ [\,]$

4 Regular languages (5 POINTS)

Which of the following statements is the *most* correct answer?

a) Every finite language is regular.

b) For some regular grammar G, it holds that $L(G) = L(G)^*$.

c) Syntactically different regular expressions can describe the same language.

d) All statements above are valid.

5 Removing left-recursion (5 POINTS)

The following grammar can be used to describe traveling schemes:

$$\begin{array}{lcl} TS & \rightarrow & TS\ Time\ Time\ TS \mid Station \\ Station & \rightarrow & Identifier \\ Time & \rightarrow & Nat : Nat \end{array}$$

Which of the following grammars is equivalent but no longer left-recursive?

a)
$$\begin{array}{ll} TS & \rightarrow TS\ (Time\ Time\ Station)* \\ Station & \rightarrow Identifier \\ Time & \rightarrow Nat : Nat \end{array}$$

b)
$$\begin{array}{ll} TS & \rightarrow Station \mid Station\ Z \\ Z & \rightarrow Time\ Time\ TS \mid Time\ Time\ TS\ Z \\ Station & \rightarrow Identifier \\ Time & \rightarrow Nat : Nat \end{array}$$

c)
$$\begin{array}{ll} TS & \rightarrow Z\ Time\ Time\ TS \mid Station \\ Z & \rightarrow TS \mid \epsilon \\ Station & \rightarrow Identifier \\ Time & \rightarrow Nat : Nat \end{array}$$

d)
$$\begin{array}{ll} TS & \rightarrow Station\ Time\ Time\ Z \\ Z & \rightarrow Station \mid TS \\ Station & \rightarrow Identifier \\ Time & \rightarrow Nat : Nat \end{array}$$

6 LL parsing (5 points)

Have a look at the following two grammars:

$$G_1: \quad \begin{aligned} S &\rightarrow cA \mid b \\ A &\rightarrow cBC \mid bSA \mid a \\ B &\rightarrow cc \mid Cb \\ C &\rightarrow aS \mid ba \end{aligned} \qquad\qquad G_2: \quad \begin{aligned} S &\rightarrow AaS \mid B \\ A &\rightarrow cS \mid \epsilon \\ B &\rightarrow b \end{aligned}$$

Are these grammars LL(1)?

a) Neither of the two grammars is LL(1)

b) Only G_1 is LL(1)

c) Only G_2 is LL(1)

d) Both grammars are LL(1)

Part II: Open Questions (70 points)

7 Chomsky hierarchy (15 points)

Position each of the following five languages in the Chomsky hierarchy. For each language, choose between the following three answers (an explanation is not required):

- The language is regular
- The language is context-free, but not regular
- The language is not context-free

a) $\{\ a^n \mid n \in \mathbb{N}, n \text{ is even}\ \}$

b) $\{\ ss^R \mid s \in \{a, b, c\}^*\ \}$

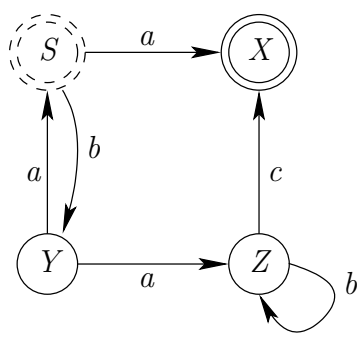
c) $\{\ a^n b^m \mid n, m \in \mathbb{N}, \mathrm{n} < \mathrm{m}\ \}$

d) $\{\epsilon, a, aab, aba, ba\}$

e) $\{\ a^{n^2} \mid n \in \mathbb{N}\ \}$

8 Regular languages (20 points)

Have a look at the following non-deterministic finite-state automaton (NFA), where S is the only start state, and $\{S, X\}$ is the collection of accepting states.



a) Give three different strings from $\{a, b, c\}^*$ that belong to the language of the NFA.

b) Give three different strings from $\{a, b, c\}^*$ that do *not* belong to the language of the NFA.

c) Construct a deterministic finite-state automaton (DFA) that describes the same language as the NFA. It suffices to draw a picture.

d) Give a regular expressions for the language of the NFA.

9 Parser combinators (15 POINTS)

We will study a language for linear equations. Let us first have a look at an example:

```
x  =  y + 6z;
y  =  10 - x + 3z;
z  =  5(3 + x);
```

Variable names consist of one or more lowercase letters. Since we do not have negation, all constants are positive. The expression `5(3 + x)` means `5` times `(3+x)`: the left operand of a product is always a constant. A context-free grammar for this language is given. All terminal symbols are quoted for clarity. The non-terminal S is the start symbol.

$$
\begin{array}{lcl}
S & \rightarrow & (\ Var\ \texttt{"="}\ E_1\ \texttt{";"}\ )^* \\
E_1 & \rightarrow & (E_1\ Op)^?\ E_2 \\
E_2 & \rightarrow & Con\ E_3 \ \mid\ Con \ \mid\ E_3 \\
E_3 & \rightarrow & Var \ \mid\ \texttt{"("}\ E_1\ \texttt{")"} \\
Op & \rightarrow & \texttt{"+"} \ \mid\ \texttt{"-"} \\
Var & \rightarrow & Letter^+ \\
Con & \rightarrow & Digit^+
\end{array}
$$

Observe that both operators are left-associative. We will define a parser for this language in two steps: first, we will write a lexer that returns a list of tokens. The lexer also takes care of whitespace. The first part of the lexer is given:

```
data Token = Open | Close | Is | SemiColon | Plus | Min | TokVar String | TokCon Int
  deriving (Eq, Show)

terminals :: [(Token, Char)]
terminals = [(Open, '('), (Close, ')'), (Is, '='), (SemiColon, ';'), (Plus, '+'), (Min, '-')]

lexer :: Parser Char [Token]
lexer = pack whiteSpace (listOf parseToken whiteSpace) whiteSpace
```

a) Define the parsers

```
whiteSpace :: Parser Char String
parseToken :: Parser Char Token
```

that are both used in the definition of *lexer*. You may use the table *terminals* (not mandatory). Lexing the example equations above should result in the following list of tokens:

```
[ TokVar "x", Is, TokVar "y", Plus, TokCon 6, TokVar "z", SemiColon, TokVar "y"
, Is, TokCon 10, Min, TokVar "x", Plus, TokCon 3, TokVar "z", SemiColon, TokVar "z"
, Is, TokCon 5, Open, TokCon 3, Plus, TokVar "x", Close, SemiColon]
```

b) The abstract syntax for equations is defined as follows. Note that the constructor (`:*:`) denotes products such as `5(3 + x)`.

```
type Equations = [Equation]
type Equation  = (String, Expr)
data Expr      = Var String | Con Int
               | Expr :+: Expr | Expr :-: Expr | Expr :*: Expr
```

Define a parser for equations that has type:

```
pEquations :: Parser Token Equations
```

Hint: closely follow the given grammar. If you have problems defining the semantic functions, then continue without these functions. Also make sure that your parser is not left-recursive.

10 Folds and algebras (20 POINTS)

The *Music* data type can represent complex pieces of music (an explanation follows):

```
data Music = Note Pitch Duration
           | Rest Duration
           | Music :+: Music      -- sequential composition
           | Music :=: Music      -- parallel composition
           | Tempo Float Music    -- scale tempo
           | Ann String Music     -- annotate

type Pitch    = Int
type Duration = Float
```

The basic building blocks are a single note (constructor *Note*) and a rest (constructor *Rest*). For each note, we have a pitch (the highness of a tone), which is represented by an *Int*, and a duration (how long the note lasts), which is represented by a *Float*. The exact interpretation of a pitch and a duration is not relevant. Furthermore, we have two infix constructor functions to combine two pieces of music. (m_1 :+: m_2) represents sequential composition: first play m_1, and once this is finished, play m_2. The other combinator, (m_1 :=: m_2), is for parallel composition: m_1 and m_2 are played simultaneously. This combined piece of music is finished as soon as both m_1 and m_2 are finished. The constructor *Tempo* scales the tempo in which it has to be executed. For instance, *Tempo* 2 m results in m being played twice as fast. The constructor *Ann* is used to annotate a piece of music: the text is merely intended for the musician.

a) Give a type synonym for the algebra of the *Music* data type.

b) Define the *fold* for the *Music* data type. Also write down the type signature of this function (and use the type synonym for the algebra).

In the remaining questions you have to use your *fold* function to do the recursion on the *Music* data type. Write down a type signature for each algebra you use.

c) Define the function

annotations :: *Music* → [*String*]

that collects all annotations in a piece of music.

d) Calculate the duration of a piece of music:

duration :: *Music* → *Duration*

e) Transpose a piece of music: given an integer n, increase the pitch of all *Note*s by n. Your function should have the following type:

transpose :: *Music* → *Int* → *Music*