भारतीय सूचना प्रौद्योगिकी संस्थान गुवाहाटी

Indian Institute of Information Technology Guwahati

COMPILERS LAB (CS321)

End-Semester Exam, 2022

Total Points: 50 Time: 3 hours

Question:	1	2	3	Total
Points:	10	10	30	50

Banking system

An **interest calculator** performs simple operations on funds that are deposited in banks. It is also able to combine various interest calculation operations in a natural way.

Suppose x is the fund deposited for k years. The available operations are:

- **regular:** this is written $x + k$ in our expressions, though our `C` function that computes it is called regular$(\cdots)$. The resulting value of this operation is $x + (x * 2.01)/100 * k$.

  For example,

  $x = 100$
  $k = 3$
  $x + k = 106.03$

- **simple:** this is written $x * k$, and is computed by the `C` function simple$(\cdots)$. The resulting value of this function is $x + (x * 5.5 * k)/100$.
- **compound:** this is written `compound`(x, k) and is computed by the `C` function compound$(\cdots)$. The resulting value of this function is $x(1 + 5.7/k)^k$.

Let **CALC** be the language of banking system that can be generated by the following grammar

$$
\begin{aligned}
S &\rightarrow E \\
E &\rightarrow \epsilon \\
E &\rightarrow \texttt{regular(E,E)} \\
E &\rightarrow \texttt{simple(E,E)} \\
E &\rightarrow \texttt{compound(E,E)} \\
E &\rightarrow \texttt{NUM NUM} \mid \texttt{NUM} \\
\texttt{NUM} &\rightarrow 0 \mid 1 \mid 2 \mid 3 \cdots \mid 9
\end{aligned}
$$

In this grammar, the non-terminals are S, E and NUM. Treat `regular`, `simple` and `compound` as just single tokens. For `simple` and `compound`, we allow any nonempty prefix of the function name as well as the full name; e.g., s, si, sim, ..., simple are all acceptable lexemes for the token simple.

1. Construct a lex file Prob1 to build a lexical analyser for **CALC**. [10]

   Sample output:

   ```
   $ ./a.out
   regular(s(co(500,2)))
   Token:  regular; Lexeme:  regular
   Token and Lexeme:  (
   Token:  simple; Lexeme:  s
   Token and Lexeme:  (
   Token:  compound; Lexeme:  co
   Token and Lexeme:  (
   Token:  NUM; Lexeme:  500
   Token and Lexeme:  ,
   Token:  NUM; Lexeme:  2
   Token and Lexeme:  <newline>
   Control − D
   ```

2. Make a copy of prob1.l, call it prob2.l, then modify it so that it can be used with yacc. Then construct a yacc file prob2.y. Then use these lex and yacc files to build a parser for **CALC**. [10]

   The core of your task is to write the grammar rules in the Rules section, in yacc format, with associated actions (using the examples in chain.y as a guide).

   Sample output:

   ```
   $ ./a.out
   regular(s(co(500,2),2),2)
   8557.1922975
   Control − D
   ```

3. In a hypothetical programming language STUD, the type `huh`[2][3] can be read as ”two sighs of 3 huhs”. The corresponding type expression is `sigh(2,sigh(3,huh))`. The expression `sigh` takes two parameters, a number and `huh`. Write a SDD for the type `huh`. Implement your SDD with an LL(1) parser and identify whether each attribute used is synthesized or inherited. You can use your LL(1) parser from previous labs. [30]