SOLUTIONS

LANGUAGE PROCESSORS EE2-15

Ensure throughout that your written characters are unambiguous, especially in terms of '*' versus '+' and white-space. If necessary, use a square under-bracket to indicate space characters.

Bison and C++ will be interpreted by a human, so some syntax errors can be tolerated as long as the intended solution is clear.

1. a) How is a right-linear grammar classified using Chomsky's hierarchy? [2]

Answer :
A right-linear grammar is a regular grammar.

b) Is Bison a top-down or bottom-up parser? Use a feature or capability of Bison to support your answer. [3]

Answer :
It is a bottom-up parser, as it uses a shift-reduce approach. It also can support left recursive grammars, which is a feature of bottom-up parsers.

c) Where can branches appear within a basic block? [2]

Answer :
Control can only leave a basic block at the end, so a branch can only appear as the final instruction.

d) Given an n character input string, what is the worst-case size needed for the stack of an LR(1) parser? [2]

Answer :
Each character could cause a push up until the last character, so it is O(n).

e) Describe the following sets: symbols, terminals, and non-terminals. [3]

Answer :
Non-terminals are symbols that exist in the input language. Non-terminals do not appear in the input language, and are defined in terms of other symbols. The set of symbols consists of the union of terminals and non-terminals.

f) Left-factor the following grammar: [2]
`X ::= 'c' 'a' 't' | 'c' 'a' 'r'`

Answer :
We'll introduce a new auxiliary symbol Xt:

```
X  ::= 'c' 'a' Xt
Xt ::= 't' | 'r'
```

g) Give two advantages of interpreters over compilers. [3]

Answer :

- *There is much less delay before the program starts executing.*
- *They are much simpler to write.*
- *They are much easier to port between platforms.*

h) Given the following grammar:
`E ::= E '+' E | E '*' E | Num`
use the input string 6+7*10 to show that the grammar is ambiguous. [5]

Answer :
One derivation is:

```
E[6+7*10]
E[E[6] + E[7*10]]
E[E[Num[6]] + E[ E[7] * E[10]]]
E[E[Num[6]] + E[ E[Num[7]] * E[Num[10]]]]
```

another is:

```
E[6+7*10]
E[E[6+7] * E[10]]
E[E[6+7] * E[Num[10]]]
E[E[E[6]+E[7]] * E[Num[10]]]
E[E[E[6]+E[Num[7]]] * E[Num[10]]]
E[E[E[Num[6]]+E[Num[7]]] * E[Num[10]]]
```

i) Give the First set for the production "α b", where α is a non-empty sequence of symbols, and b is a terminal. [4]

Answer :
If $\varepsilon \in First(\alpha)$ *then* $First(\alpha\ b) = (First(\alpha) - \{\varepsilon\}) \cup First(b)$.
Else $First(\alpha\ b) = First(\alpha)$.

j) Give pseudo-code for a general-purpose DFA. [6]

Answer :

```
function dfa(table, start, src):
  state=start
  while state!=None:
    c=src()
    state=table[(state,c)]
  return state
```

k) Consider the following chain of reasoning:

- Fact: Context-free grammars are defined over a finite set of tokens.
- Fact: The set of identifiers in C is infinite.
- Inference: C does not have a context-free grammar.

This appears to lead to a contradiction with:

- Fact: context-free grammars *do* exist for C.

i) Identify the faulty reasoning that leads to the contradiction. [4]

Answer :
While the set of possible identifiers is infinite, they all map to a single terminal/token class within the C grammar. So in the grammar, there are still only a finite set of tokens.
(Note: there is a deeper problem with C because of potential conflicts between typedef names and identifiers. However, this can be handled for the same reason, and is not relevant to the contradiction seen here.)

ii) Describe the technique used to resolve this problem in compilers. [4]

Answer :
The separation into lexers and parsers means that the infinite set of identifiers is grouped into a single token class using regular expressions in the lexer. The parser then receives just an identifier token, with the value of the identifier attached as an attribute.
(Note: this is also where the typedef-name identifier clash can be handled, by effectively making the lexer context-sensitive (maintaining a symbol table), and allowing the parser to be context-free.)

2. In the following, assume we are working with regular expressions with the following constructs: sequence; alternation; one-or-more; zero-or-more; groups; character ranges; and anchors (start and end of string).

Many regular expression engines also support *capture groups*, which allows the user to indicate parts of the match that should be remembered (captured), and made available under a label. The labels can then be referred to from a substitution string. For our purposes, we will state that all bracketed groups define a capture group, and we can refer to them using the symbol $n, where n is a decimal integer. $1 then defines the first capture group, $2 the second, and so on.

Some examples of using capture groups are:

	Regex	Substitution	Input	Output
1	`[a-z]([0-9])`	`$1`	c4	4
2	`([a-z]+)=([0-9]+)`	`$1:$2`	debug=3	debug:3
3	`[a-z]+@([a-z]+([.][a-z]+)+)`	`X@$1`	bib@bob.co.uk	X@bob.co.uk
4			gpg.tar.gz	gpg

a) Write a regular expression and substitution pattern for taking a file name and extracting just the base filename, excluding any filename extensions. An example input and output is shown in line 4 of the table. [3]

Answer :
Regex: `([^.]+).*`
Substitution: `$1`

b) What is the order of precedence for the regular expression constructs, from highest to lowest? [3]

Answer :
From highest to lowest, the precedence is:

- *Brackets and character classes: () and []*
- *Quantifiers: * and +*
- *Sequences*
- *Alternation*

c) Give a Bison-like definition of a symbol "CharRange", which recognises a regular expression character range (e.g. `[a]`, `[01]`, `[0-9a-z]`). Terminals can be defined as literals or using regular expressions. You can define intermediate helper symbols if necessary. [6]

Answer :

```
CharRange   ::= '[' CharDetails+ ']'

CharDetails ::= TCharNotSub
              | TCharNotSub '-' TCharNotSub

TCharNotSub ::= ([\\].)|[^-]
```

d) Give the remaining Bison-like grammar for recognising regular expressions.

[7]

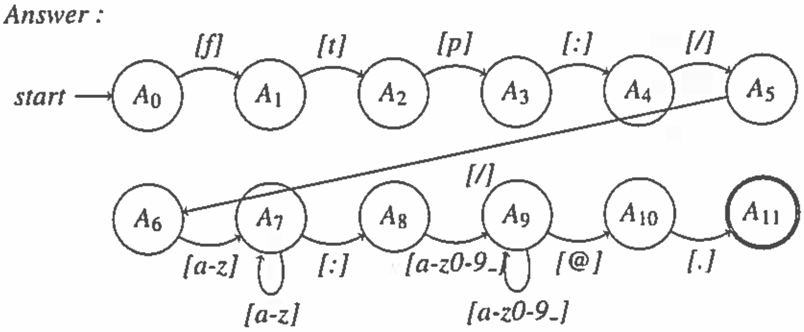
Answer :

```
Regex ::= Alternation

Alternation ::= Sequence | Sequence '|' Alternation
Sequence    ::= Quantified +
Quantified  ::= Primitive ( '+' | '*' )
Primitive   ::= '(' Alternation ')' | CharRange | TChar
```

e) The regular expression `^ftp://([a-z]+):([a-z0-9_]+)@[.]+$` is designed to match URLs containing a user name and password. Draw a DFA for recognising this pattern. [6]

Answer :



f) The URL regex contains two capture groups. Assume there is an additional DFA annotation called `append[n]`, which pushes the current input character onto the end of capture group n. Where should the annotations be added to your DFA in order to capture the groups? [5]

Answer :
The annotation `append[1]` *should be used on the arc from A6 to A7 and the self-loop on A7. The annotation* `append[2]` *should be used on the arc from A8 to A9, and on the self-loop on A9.*

3. The following AST models a simple language where all statements are also expressions:

```
struct Expr {};

struct Num      : Expr{ int value; };

struct Add      : Expr{ Expr *left; Expr *right;  };

struct VarRef   : Expr{ string id; };

struct VarDecl  : Expr{ string id; Expr *init; Expr *body; };

struct Assign   : Expr{ string target; Expr *source;  };

struct Sequence : Expr{ vector<Expr*> body;  };

struct While    : Expr{ Expr *cond; Expr *body; };

struct Func     : { string name;  vector<string> args;  Expr *body; };
```

All expressions return a value. Statement-like expressions (VarDecl, Assign, Sequence, While) return the value of the last evaluated sub-expression. While loops execute while the condition evaluates to a non-zero value.

An example function for multiplication is:

```
Func[ multiply, [a,b],
  VarDecl[ res, 0,
    Sequence[
      While[ b,
        Sequence[
          Assign[ b, Add[ VarRef[ b ], Num[ -1 ] ] ],
          Assign[ res, Add[ VarRef[ res ], VarRef[ a ] ] ]
        ],
      ],
      res
    ]
  ]
]
```

a) Translate the example function to C. State any assumptions needed. [5]

Answer :
We'll assume that all data-types are integers.

```
int multiply(int a, int b)
{
  int res=0;
  while(b){
    b=b-1;
    res=res+a;
  }
  return res;
}
```

b) The AST needs to be compiled to MIPS assembly. Define a function calling convention which can support this language, and describe a function call as seen by both caller and callee. (This does *not* have to follow the GNU ABI, and generality is more important than efficiency.) [6]

Answer :

For this language we need to support a function return point, a method for passing parameters, and include support for temporary values during evaluation. The method chosen here is:

- *Return address passed in ra=$31*
- *Stack pointer passed in sp=$1*
- *n input arguments are at sp(0),sp(1),...,sp(n).*
- *Return value is placed in ret=$2*
- *All registers are preserved by callee except ret=$2*
- *Frame pointer is at fp=$3*

A function call is then:

- *Caller: Write the n arguments at sp(0),sp(1),...,sp(n).*
- *Caller: jra (jump with return address) to callee.*
- *Callee: For each register push the value: for i=0..31, sp(i)=ri; sp=sp+1*
- *Callee: Copy stack pointer to frame-pointer: fp=sp*
- *Callee: Execute function body. Stack space is allocated by incrementing sp.*
- *Callee: Write result to $2*
- *Callee: Copy the frame-pointer to stack-pointer: sp=fp*
- *Callee: Pop all registers back off stack: for i=31..0, sp=sp-1; ri=sp(i)*
- *Callee: jr to register 31*
- *Caller: Use result found in $2*

c) Give a general MIPS assembly template for the code emitted for a While loop. The template should follow your calling convention and the semantics of the language. [6]

Answer :

```
top:
<ExprCond>
# Leaves it's value in register $2
beq $2,$0 bottom
nop
<ExprBody>
jmp top
nop
bottom:
```

d) A virtual function called codeGen is going to be added to the Expr node. Give a function prototype (i.e. arguments and return type) for the `Expr::codeGen` function, and add minimal comments to explain how it works. If necessary, helper classes or function declarations can be used. [6]

Answer :

```
class Context
{
  // Reserves a new space on the stack and returns it
  int createTemporary();

  // Establish that the given name is at the given place on the stack
  void bindName(string name, int location);

  // Remove an existing binding (potentially restoring previous one)
  void unbindName(string name);

  // Lookup binding, returning -1 if it doesn't exist
  int lookupName(string name);

  // Creates a new unused label
  string createLabel();
};

// Emit code into dst, using the (modifiable) ctxt
void Expr::codeGen(std::cout &dst, Context *ctxt) const;
```

e) Give C++ code for the implementation of `While::codeGen`. [7]

Answer :

```
void While::codeGen(std::cout &dst, Context *ctxt) const
{
  string top=ctxt->createLabel(), bottom=ctxt->createLabel();
  dst<<top<<"\n";
  cond->codeGen(dst, ctxt);
  dst<<"bne $2, $0\n";
  dst<<"nop\n";
  body->codeGen(dst, ctxt);
  dst<<"jmp top;\n";
  dst<<"nop\n";
  dst<<bottom<<endl;
}
```