Solutions EE2-15

LANGUAGE PROCESSORS

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.

1. a) Define the terms AST and IR, and briefly explain the role of each in a compiler. [3]

Answer :
AST = Abstract Syntax Tree, IR = Intermediate Representation. The AST is built as a language-specific representation of the input code in order to analyse and check the semantics of the code. The IR is a language-independent representation built from the AST, used to perform optimisations at a lower level without needing to know which language the code was expressed in.

b) What are the four levels of Chomsky's hierarchy? [3]

Answer :

- *0 : Unrestricted or recursively enumerable*
- *1 : Context-sensitive*
- *2 : Context-free*
- *3 : Regular*

c) A grammar can be defined as a tuple (V_T, V_N, P, s). Give a name for each element of the tuple which describes its role. [3]

Answer :

- *V_T : The set of terminal symbols, which defines the tokens (or characters) which appear in the input language.*
- *V_N : The set of terminal symbols, which gives the abstract symbols constructed out of other terminals and non-terminals.*
- *P : The set of production rules, which explain how symbols can be rewritten in terms of other symbols.*
- *s : A start symbol.*

d) What are the conditions for a grammar to be left-linear? [3]

Answer :
All production forms must be of the form $A \rightarrow a$ or $A \rightarrow b\ C$, where A and C are non-terminals and a and b are terminals.

e) Define synthesised versus inherited attributes. [4]

Answer :
A synthesised attribute is determined at the leaves (terminals) of an AST, and each non-terminal will synthesise the attribute as a function of its child nodes attributes. An inherited attribute is defined by a parent node, and propagated downwards (inherited) to the children of the node.

f) Describe an ε transition, and explain why it is desirable to eliminate them from grammars before constructing a digital state machine. [4]

Answer :
An epsilon transition allows a state machine to change state without consuming an input symbol. Eliminating them is used to ensure that the state machine can only to be in one state at a time.

g) What is an LL(1) grammar, and why are they considered efficient? [4]

Answer :
LL(1) grammars require at most one token of lookahead to parse a sentence. This is efficient because the parser can be implemented as a top-down parser without backtracking, taking time linear in the number of symbols.

h) Explain the worst-case behaviour of a top-down versus bottom-up parser when processing sentences from this grammar:

```
A ::= 'x' A 'y'
    | 'x' A 'z'
    | 'c'
```

[4]

Answer :
The top-down parser will need to use backtracking, so for a sentence of the form "xxxxxxxxxczzzzzzzzz" it would require space linear in the input length, and exponential time. The bottom-up parser would also require linear space, but only linear time.

i) Describe the role and signature (input and output types) of the transition function in a PDA. [4]

Answer :
The transition function takes in a state, the symbol popped from the top of the stack, and a character; it produces the next state and zero or more stack symbols to put onto the stack.

j) How can register allocation be modelled as graph colouring? [4]

Answer :
Represent variables as nodes. An edge exists between two nodes if they are live

simultaneously. If we can colour the graph with k colours, such that no nodes with the same colour are connected by an edge, then all variables can be be stored in k registers.

k) Define a basic block. [4]

Answer :
A basic block is a sequence of instructions that contains no internal control flow. The block can only be entered at the first instruction, and control may only leave at the last instruction.

2. A configuration language is needed for describing the dependencies between different software packages. A brain-storming session has resulted in the following language example, which needs to be turned into a grammar and parser.

```
package libc {
  description : [C standard library];
  version     : 3.7.2 ;
  source      : http://source.gnu.org/gcc/3.7.2/src.tar.gz ;
};
package gcc {
  description : [The GNU C compiler];
  version     : 3.7.2;
  dependsOn   : libc:3.7.2 ;
  source      : ftp://ftp.gnu.org/gcc/3-7-2/sources.tar.gz ;
  binary[x86]  : ftp://ftp.gnu.org/gcc/3-7-2/ia32.tar.gz ;
  binary[mips] : ftp://ftp.gnu.org/gcc/3-7-2/mips.tar.gz ;
};
package go {
  version     : 1.3;
  binary[x86] : http://google.com/go/Releases/FAE21A787FFE17/x86.tar.bz2;
  dependsOn   : libjpg:3.1.2;
  binary[x86_64] : http://google.com/go/Releases/FAE21A787FFE17/x86_64.tar.bz2;
  dependsOn   : libgocore:1.3 ;
description : [The Go compiler];
};
```

Currently the only legal architectures are x86, x86_64, mips, and arm, though it is intended that more can be supported as needed.

a) Given that only ftp and http are allowed, and that all sources must be located at a .com or .org address, give a regular expression for matching legal URLs used in this language. Make sure your written characters are easily readable and unambiguous. [5]

Answer :

```
(http|ftp):[/][/]([a-z]+\.)+(org|com)([/]([a-zA-Z0-9\-\._])+
```

b) Define terminal symbols for this language, giving the names of each terminal and the associated regular expression. Keywords such as "package" and "source" do not need to be defined. You can refer to the previous answer where needed, rather than writing out the regex again. [5]

Answer :

```
NAME [a-z]+
STRING \[[^\[\]]*\]
VERSION [0-9]+\.[0-9]+\.[0-9]+
ARCH x86|x86_64|mips|arm
URL (see previous question)
```

c) Give a grammar for the language, with a start symbol called REPOSITORY. This grammar does not need to be efficient, but it should be unambiguous and capture the language. If necessary, describe any assumptions or restrictions you made. [6]

Answer :

```
REPOSITORY ::= PACKAGE*
PACKAGE ::= 'package' NAME '{' ( PACKAGE_PART ';' ) * '}' ';'
PACKAGE_PART ::= 'description' STRING
               | 'version' VERSION
               | 'dependsOn' NAME ':' VERSION
               | 'binary'' '[' ARCH ']' ':' URL
```

d) Design an AST for this language using C or C++. The AST should be designed to be constructed from a parser, then manipulated and queried. [6]

Answer :

```
using namespace std;

struct Version{
    int major;
    int minor;
    int revision;
};

struct Dependency{
    string name;
    Version version;
};

struct BinarySource{
    string binary;
    string url;
};

struct Package{
    string name;
    string description;
    string sourcesUrl;
    Version revision;
    set<Dependency*> dependencies;
    map<string,string> binaryUrls;

    void AddBinarySource(BinarySource *source)
    { binaryUrls[name]=url; }

    void AddDependency(Dependency *dep)
    { dependencies.insert(dep); }
};

struct Repository
{
    std::vector<Package*> packages;

    void AddPackage(Package *package)
    { packages.push_back(packages); }
};
```

e) Describe the language using grammar rules compatible with the bison parser generator, including semantic actions needed to build the AST. This does not need to use the same grammar you gave in part c). [8]

Answer :

```
REPOSITORY ::= REPOSITORY PACKAGE \{ $1->AddPackage($2); $$=$1; \}
             | PACKAGE { $$=new Repository(); $$->AddPackage($1); }

PACKAGE_HEADER ::= 'package' NAME '{' PACKAGE_BODY { $$=$3; $$->name=$2; }

PACKAGE_BODY ::= PACKAGE_BODY PACKAGE_DEPENDS_ON     { $1->AddDependency($2); $$=$1; }
               | PACKAGE_BODY PACKAGE_VERSION        { $1->version=$2; $\$=$1; }
               | PACKAGE_BODY PACKAGE_DESCRIPTION    { $1->description=$2; $$=$1; }
               | PACKAGE_BODY PACKAGE_BINARY         { $1->AddBinarySource($2); $$=$1; }
```

```
                | PACKAGE_BODY PACKAGE_SOURCE          { $1->source=$2; $$=$1; }
                | PACKAGE_FOOTER                       { $1=new Package(); }

PACKAGE_DEPENDS_ON ::= 'dependsOn' NAME ':' VERSION ';' { $$=new Dependency{$2,$4}; }

PACKAGE_VERSION ::= 'version' VERSION ';' { $$=$2; }

PACKAGE_DESCRIPTION ::= 'description' ':' STRING ';' { $$=$3; }

PACKAGE_SOURCE ::= 'source' ':' URL ';' { $$=$3; }

PACKAGE_BINARY ::= 'binary' '[' ARCH ']' '=' URL ';' { $$=new BinarySource{$3,6}; }
```

3. a) A simple digital circuit is connected to the world via a UART (character device) and processes read and write requests encoded as streams of characters. Addresses precede data in the stream and both are encoded as variable length decimal numbers.

The two transaction types are:

- Read request: `A[0-9]+R`
- Write request: `A[0-9]+W[0-9]+`

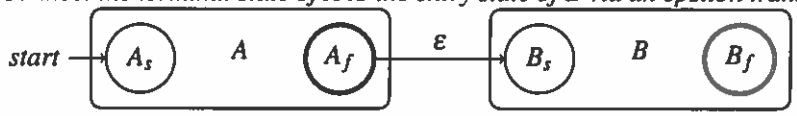
Both requests start with a base Address; R will cause data to be read from that address and sent to another channel (not seen here); while W indicates a piece of data to write at that address.

b) Give one example each of a valid read and write request. [2]

Answer :
Read: `A234R` *Write:* `A13W124`

c) Use a diagram to show how Thompson's algorithm is used to construct an NFA for the sequence AB, where A and B are arbitrary regular expressions. [3]

Answer :
Connect the terminal state of A to the entry state of B via an epsilon transition:

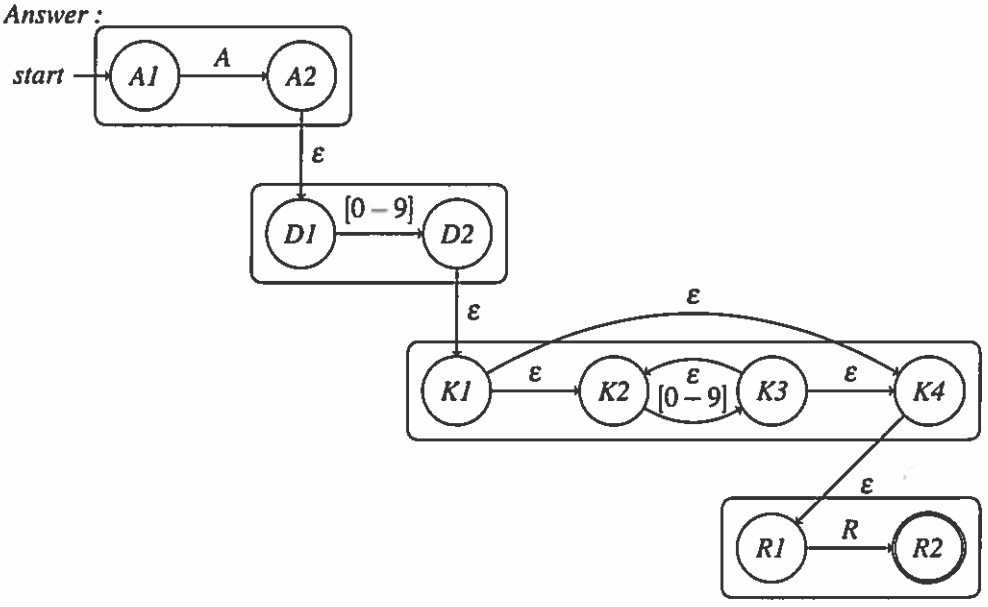


d) Rewrite the read request using the Kleene star. [3]

Answer :
`A[0-9][0-9]*R`

e) Construct the NFA for a read request using Thompson's algorithm. [6]

Answer :



f) Define the ε-closure function, and apply it to the nodes in the NFA you created in part e). [5]

Answer :
The ε-closure of a node captures all the nodes reachable via one or more ε transitions. For the NFA, we have:

- $\varepsilon - closure(A1) = A1$
- $\varepsilon - closure(A2) = A2, D1$
- $\varepsilon - closure(D1) = D1$
- $\varepsilon - closure(D2) = D2, K1, K2, K4$
- $\varepsilon - closure(K1) = K1, K2, K4$
- $\varepsilon - closure(K2) = K2$
- $\varepsilon - closure(K3) = K3, K4, R1$
- $\varepsilon - closure(R1) = R1$
- $\varepsilon - closure(R2) = R2$

g) Give the name of the general algorithm used to convert an NFA to a DFA, and describe it using pseudo-code. [7]

Answer :
The general purpose algorithm is subset (or powerset) construction.

```
1. Set the start state of the DFA as the epsilon-closure of the NFA
2. Push the start state onto a queue of new states
3. While the queue of new states is not empty:
   a. Pop a state s from the queue
   b. For each possible input x:
       i. s' = move(s,x)
      ii. s'' = \epsilon-closure(s')
     iii. Add s'' as a new state of the DFA
      iv. Push s'' onto the queue
4. Terminal NFA states are those containing any DFA terminal states
```

h) A request can be either a read request or a write request. Using the algorithm from the previous question, or any other method, generate the DFA for a full request (i.e. either a read or write request). [4]

Answer :