**CSCI565 - Compiler Design**

**Spring 2017**

**Solution**

**Problem 1 [40 points]: Regular Expressions and Finite Automata**

In this problem, we consider the translation of strings with integer values into a comma-separated-value (csv) format. The input string consists of a sequence of one or more letters or numerical digits (forming an identifier) followed by one or more blank space characters to which we have a set of three integers separated by the ‘-‘ character and terminated by a ‘newline’ (‘\n’) character. The difficulty arises from the fact that the first integer can be a negative integer and thus begins with the ‘-‘ character but the two subsequent integers are assumed to be positive. The output string has all ‘-‘ character, except the first, replaced by an opening ‘(‘, ‘:‘ and a ‘)‘ character respectively. Below you can find a couple of examples of the intended translation scheme where the ‘\n’ stands for the new-line character:

<table>
<thead>
<tr>
<th>Input</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>F01 -1-0-1\n</td>
<td>F01, (-1:0), 1\n</td>
</tr>
<tr>
<td>DX1 1-0-3\n</td>
<td>DX1, (1:0), 3\n</td>
</tr>
</tbody>
</table>

String that do not comply with the structured described are not to be translated and the resulting translation generates an error condition. We can model this condition by having the translation program emit an error message as described in more detail in section d).

For this translation scheme, answer the following questions:

a) [05 points] Derive a regular expression RE over the ASCII character alphabet that captures valid strings and thus invalid strings.

b) [10 points] Convert the RE developed in a) into an NFA using the Thompson construction.

c) [10 points] Convert the NFA derived in section b) to a DFA using the subset construction. Show the mapping between the states in the NFA and the resulting DFA.

d) [10 points] Minimize the DFA derived in section c) (or show it is already minimal) using the iterative refinement algorithm described in class.

e) [05 points] Using the C code skeleton described in class for the implementation of DFA with a table, implement a C code function that can recognize valid strings matching the RE derived in a). Whenever an error condition is detected, your implementation should print the ‘error’ string.

**Solution:**

a) [05 points] Derive a regular expression RE over the ASCII character alphabet that captures valid strings and thus invalid strings.

A regular expression (not necessarily unique) could be described as shown below where *letter* denotes an alphabetic character and *digit* a numerical digit between 0 and 9. Also, *blank* denotes a space character, *int* denotes a string representing an integer (both positive or negative) and *posint* denotes a string exclusively representing a positive integer. Lastly, ‘.’ denotes regular expression concatenation and ‘|’ alternation.
RE = {letter}.({letter} | {digit})*.{blank}+.{int}'-{'posint}'-{'posint}.{blank}*. ‘\n’

letter = (‘a’, ‘b’, … , ‘z’)
digit = (‘0’, ‘1’, … , ‘9’)
posint = {digit}+
int = (‘*’ | ε).{digit}+

b) [10 points] Convert the RE developed in a) into an NFA using the Thompson construction.

A simplified version of the NFA resulting from the use of the Thompson construction for the RE described in a) above is shown below. For compactness, we summarized multiple identical state transitions on various characters, such as transitions on all alphabetic characters as a single state transition. The start state is the state labeled 0. All omitted transitions implicitly denote a transition to an error trap state, say Serror.

![NFA Diagram]

c) [10 points] Convert the NFA derived in section b) to a DFA using the subset construction. Show the mapping between the states in the NFA and the resulting DFA.

The figure below depicts the DFA resulting from the use of the subset construction. Each DFA state is labelled by the corresponding subset of states of the original NFA.

![DFA Diagram]

d) [05 points] Minimize the DFA derived in section c) (or show it is already minimal) using the iterative refinement algorithm described in class.
e) [10 points] Using the C code skeleton described in class for the implementation of DFA with a table, implement a C code function that can recognize valid strings matching the RE derived in a). Whenever an error condition is detected, your implementation should print the ‘error’ string.
```c
#include <stdio.h>
#include <stdlib.h>
#include <strings.h>
#include <ctype.h>
#define error_state -1
int current_state;

int delta(char c){
    switch(current_state){
    case 0:
        if(isalpha(c) != 0){
            printf("%c",c);
            return 1;
        }
        return error_state;
        break;
    case 1:
        if(isalpha(c) != 0) || (isdigit(c) != 0)){
            printf("%c",c);
            return 1;
        }
        if(c == ' '){
            printf("%c", ,c);
            return 2;
        }
        return error_state;
        break;
    case 2:
        if(c == ' '){
            return 2;
        }
        if(c == '-'){
            printf("(-");
            return 3;
        }
        if(isdigit(c) != 0){
            printf("(%c",c);
            return 4;
        }
        return error_state;
        break;
    case 3:
        if(isdigit(c) != 0){
            printf("%c",c);
            return 4;
        }
        return error_state;
        break;
    case 4:
        if(isdigit(c) != 0){
            printf("%c",c);
            return 4;
        }
        if(c == '-'){
            printf(";");
            return 5;
        }
        return error_state;
        break;
    case 5:
        if(isdigit(c) != 0){
            printf("%c",c);
            return 5;
        }
        return error_state;
        break;
    case 6:
        if(isdigit(c) != 0){
            printf("%c",c);
            return 6;
        }
        if(c == '-'){
            printf("), ");
            return 7;
        }
        return error_state;
        break;
    case 7:
        if(isdigit(c) != 0){
            printf("%c",c);
            return 8;
        }
        if(c == ' '){
            printf(" ");
            return 9;
        }
        return error_state;
        break;
    case 8:
        if(isdigit(c) != 0){
            printf("%c",c);
            return 8;
        }
        if(c == ' '){
            return 9;
        }
        return error_state;
        break;
    case 9:
        if(c == ' '){
            return 9;
        }
        return error_state;
        break;
    case 10:
        break;
    default:
    return error_state;
    }
}

int main (int argc, char ** argv){
    char c;
    current_state = 0;
    while(1){
        c = getchar();
        if(c == '\n')
            break;
        current_state = delta(c);
        if(current_state == error_state){
            break;
        }
        if((current_state == 8) || (current_state == 9)){
            printf(" Valid string\n");
        } else {
            printf(" error \n");
        }
        return 0;
    }
```
Problem 2 [30 points]: Predictive Top-Down Parsing

Consider the CFG grammar $G = (N=\{P, SL\}, T = \{‘b’, ’e’, ‘s’, ’;’\}, P, E)$ with set of productions $P$:

$$
P \rightarrow ‘b’ SL ‘e’$

$$
SL \rightarrow SL ‘;’ ‘s’ | ‘s’$

Questions:

a) [10 points] Eliminate left-recursion in this grammar by deriving an equivalent grammar that is non-left recursive. If the resulting grammar needs to be left-factored, derive an equivalent grammar that is also left-factored and thus has the LL(1) property.

b) [10 points] Compute the FIRST and FOLLOW sets for each production’s RHS and the non-terminal symbol respectively. Use these to show that the grammar has in fact the LL(1) property.

c) [10 points] Derive the LL(1) parsing table as described in the lectures and show that in fact the grammar is parseable using the LL(1) parsing algorithm. Show the sequence of parsing actions for the input “b s ; s e”.

Solution:

a) The original grammar is left-recursive on the SL non-terminal symbol. Instead, we can rewrite this grammar as shown below which is both non-left recursive and left-factored.

$$
P \rightarrow ‘b’ SL ‘e’$

$$
SL \rightarrow ‘s’ T$

$$
T \rightarrow ‘;’ SL | ε$

b) The FIRST and FOLLOW sets are as shown below. In this particular case the FIRST sets are identical to the FIRST sets for each of the productions as none of the productions generate the empty string.

FIRST(‘b’ SL ‘e’) = \{ ‘b’ \} 
FIRST(‘s’ T) = \{ ‘s’ \} 
FIRST(‘;’ SL) = \{ ‘;’ \} 
FIRST(e) = \{ ε \}

FOLLOW(P) = \{ $ \}
FOLLOW(SL) = \{ ‘e’ \}
FOLLOW(T) = \{ ‘e’ \}

c) From the item c) above it is clear that the intersection of all FIRST(RHS) for all productions is the empty set, so the LL table will have no multiply-defined entries. The LL parsing table is as shown below.

<table>
<thead>
<tr>
<th></th>
<th>‘b’</th>
<th>‘e’</th>
<th>‘s’</th>
<th>‘;’</th>
<th>$</th>
</tr>
</thead>
<tbody>
<tr>
<td>P</td>
<td>$</td>
<td>‘e’</td>
<td>‘s’</td>
<td>‘;’</td>
<td>$</td>
</tr>
<tr>
<td>SL</td>
<td>SL → ‘b’ SL ‘e’</td>
<td>SL → ‘s’ T</td>
<td>SL → ‘;’ SL</td>
<td>T → ε</td>
<td>T → ‘;’ SL</td>
</tr>
</tbody>
</table>
d) For the input “b s ; s e” when the parser actions are as shown below:

<table>
<thead>
<tr>
<th>STACK</th>
<th>INPUT</th>
<th>ACTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>$P$</td>
<td>b s ; s ; s e $</td>
<td>----</td>
</tr>
<tr>
<td>Se SL b</td>
<td>b s ; s ; s e $</td>
<td>expand P → ‘b’ SL ‘e’</td>
</tr>
<tr>
<td>Se SL</td>
<td>s ; s ; s e $</td>
<td>matched ‘b’</td>
</tr>
<tr>
<td>Se T s</td>
<td>s ; s ; s e $</td>
<td>expand SL → ‘s’ T</td>
</tr>
<tr>
<td>Se T</td>
<td>s ; s e $</td>
<td>matched ‘s’</td>
</tr>
<tr>
<td>Se SL ;</td>
<td>s ; s e $</td>
<td>expand T → ‘;’ SL</td>
</tr>
<tr>
<td>Se SL</td>
<td>s ; e $</td>
<td>matched ‘;’</td>
</tr>
<tr>
<td>Se T s</td>
<td>s ; e $</td>
<td>expand SL → ‘s’ T</td>
</tr>
<tr>
<td>Se T</td>
<td>e $</td>
<td>matched ‘s’</td>
</tr>
<tr>
<td>Se</td>
<td>e $</td>
<td>expand T → ε</td>
</tr>
<tr>
<td>S</td>
<td>$</td>
<td>matched ‘ε’</td>
</tr>
<tr>
<td></td>
<td></td>
<td>accept</td>
</tr>
</tbody>
</table>
Problem 3 [30 points]: Bottom-Up Parsing

Consider the CFG \( G = \{ NT = \{ E, T, F \}, T = \{ a, b, +, * \}, P, E \} \) with the set of productions as follows:

1. \( E \rightarrow E + T \)
2. \( E \rightarrow T \)
3. \( T \rightarrow T \ F \)
4. \( T \rightarrow F \)
5. \( F \rightarrow F * \)
6. \( F \rightarrow a \)
7. \( F \rightarrow b \)

a) [05 points] Compute the FIRST and FOLLOW for all nonterminals in \( G \).
b) [10 points] Consider the augmented grammar \( G' = \{ NT, T, \{ (0) E' \rightarrow E$ \} + P, E' \} \). Compute the set of LR(0) items for \( G' \).
c) [10 points] Compute the LR(0) parsing table for \( G' \). If there are shift-reduce conflicts use the SLR parse table construction algorithm.
d) [05 points] Show the movements of the parser for the input \( w = "a+ab*" \).

Solution:

a) We compute the FIRST and FOLLOW for the augmented grammar \( 0) E' \rightarrow E$ 

\[
\begin{align*}
\text{FIRST}(E) & = \text{FIRST}(T) = \text{FIRST}(F) = \{ a, b \} \\
\text{FOLLOW}(E) & = \{ +, $ \} \\
\text{FOLLOW}(T) & = \text{FIRST}(F) + \text{FOLLOW}(E) = \{ a, b, +, $ \} \\
\text{FOLLOW}(F) & = \{ *, a, b, +, $ \}
\end{align*}
\]

b) Consider the augmented grammars \( E' \rightarrow E$ \) we compute the LR(0) set of items.

\[
\begin{align*}
s_0 \ &= \text{closure}(\{(E' \rightarrow \bullet E$\}) \\
\ &= E' \rightarrow \bullet E$ \\
E \ &\rightarrow \bullet E + T \\
E \ &\rightarrow \bullet T \\
T \ &\rightarrow \bullet T \ F \\
T \ &\rightarrow \bullet F \\
F \ &\rightarrow \bullet F * \\
F \ &\rightarrow \bullet a \\
F \ &\rightarrow \bullet b
\end{align*}
\]

\[
\begin{align*}
s_2 \ &= \text{goto} (s_0, T) \\
\ &= \text{closure}(\{(E \rightarrow T), [T \rightarrow T \ F]\}) \\
\ &= E \rightarrow \bullet T \\
T \ &\rightarrow \bullet F \\
F \ &\rightarrow \bullet F * \\
F \ &\rightarrow \bullet a
\end{align*}
\]

\[
\begin{align*}
s_3 \ &= \text{goto} (s_0, F) \\
\ &= \text{closure}(\{(T \rightarrow F), [F \rightarrow F *]\}) \\
\ &= T \rightarrow \bullet F \\
F \ &\rightarrow \bullet F *
\end{align*}
\]

\[
\begin{align*}
s_4 \ &= \text{goto} (s_2, a) \\
\ &= \text{closure}(\{(F \rightarrow a)\}) \\
\ &= F \rightarrow \bullet a
\end{align*}
\]

\[
\begin{align*}
l_5 \ &= \text{goto} (s_2, b) \\
\ &= \text{closure}(\{(F \rightarrow b)\}) \\
\ &= F \rightarrow \bullet b
\end{align*}
\]
c) We cannot construct an LR(0) parsing table because states s1, s2, s3, s7 and s9 have shift-reduce conflicts. We use the FOLLOW sets to eliminate the conflicts and build the SLR parsing table below.
d) For example if input = “a+ab*$” the parsing is:

<table>
<thead>
<tr>
<th>STATE</th>
<th>'a'</th>
<th>'b'</th>
<th>'+'</th>
<th>$</th>
<th>E</th>
<th>T</th>
<th>F</th>
</tr>
</thead>
<tbody>
<tr>
<td>s0</td>
<td>shift s4</td>
<td>shift s5</td>
<td></td>
<td></td>
<td>goto s1</td>
<td>goto s2</td>
<td>goto s3</td>
</tr>
<tr>
<td>s1</td>
<td></td>
<td></td>
<td>shift s6</td>
<td></td>
<td>Accept</td>
<td></td>
<td></td>
</tr>
<tr>
<td>s2</td>
<td>shift s4</td>
<td>shift s5</td>
<td>reduce r2</td>
<td>reduce r2</td>
<td>goto s3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>s3</td>
<td>reduce r4</td>
<td>reduce r4</td>
<td>reduce r4</td>
<td>shift s8</td>
<td>reduce r4</td>
<td>goto s3</td>
<td></td>
</tr>
<tr>
<td>s4</td>
<td>reduce r6</td>
<td>reduce r6</td>
<td>reduce r6</td>
<td>reduce r6</td>
<td>reduce r6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>s5</td>
<td>reduce r7</td>
<td>reduce r7</td>
<td>reduce r7</td>
<td>reduce r7</td>
<td>reduce r7</td>
<td></td>
<td></td>
</tr>
<tr>
<td>s6</td>
<td>shift s4</td>
<td>shift s5</td>
<td></td>
<td></td>
<td>goto s9</td>
<td>goto s3</td>
<td></td>
</tr>
<tr>
<td>s7</td>
<td>reduce r3</td>
<td>reduce r3</td>
<td>reduce r3</td>
<td>shift s8</td>
<td>reduce r3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>s8</td>
<td>reduce r5</td>
<td>reduce r5</td>
<td>reduce r5</td>
<td>reduce r5</td>
<td>reduce r5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>s9</td>
<td>shift s4</td>
<td>shift s5</td>
<td>reduce r1</td>
<td>reduce r1</td>
<td>goto s7</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

STACK ACTION PROD

$0 shift s4 
$0a4 reduce r6 F → a
$0F3 reduce r4 T → F
$0T2 reduce r2 E → T
$0E1 shift s6
$0E1+6 shift s4
$0E1+6a4 reduce r6 F → a
$0E1+6F3 reduce r4 T → F
$0E1+6T9 shift s5
$0E1+6T9b5 reduce r7 F → b
$0E1+6T9F7 shift s8
$0E1+6T9F7*8 reduce r5 F → F*
$0E1+6T9F7 reduce r3 T → TF
$0E1+6T9 reduce r4 E → E+T
$0E1 accept