```
// ExpandRules.cpp - EXPAND ROBOT RESPONSE RULES INTO A REPLY LOOKUP TABLE
//
// MODULE INDEX
// NAME                    CONTENTS
// GetChar                 Read a character
// GetToken                Read a token
// GetComponent            Get component
// GetPhrase               Get phrase
// PermuteKeySet           Permute a key phrase set
// GetAlternative          Get an alternative
// GetAlternativeList      Get alternative list
// EmitAnswer              Emit an answer
// main                    Main line
//
// MAINTENANCE HISTORY
// DATE        PROGRAMMER AND DETAILS
// 28-09-15    JS   Original
// 29-10-15    JS   Throw rejection on invalid component
//
//-----------------------------------------------------------------------------

#include <cstring>              // C-style string manipulation functions
#include <cctype>               // C-style character typing functions
#include <string>               // C++ string declarations
#include <iostream>             // C++ I/O stream declarations
#include <vector>               // C++ vector declarations
#include <map>                  // C++ map declarations
using namespace std;            // Expand the standard namespace

//-----------------------------------------------------------------------------

// ERROR EXCEPTION

struct Error_c {
    int        errorLineNo;     // Error line number
    string     errorMessage;    // Error message
    Error_c (int lineNo, const char *message) // Constructor
        : errorLineNo(lineNo), errorMessage(message) {}
};

//-----------------------------------------------------------------------------

// GLOBAL DATA

enum {WORD, COLON, SEMI, BAR, AMPERSAND, OPEN_BRACKET, CLOSE_BRACKET,
    ASTERISK, INVALID_CHAR, END_OF_FILE} tokenType; // Token type code
string        tokenVal;         // Token value
int           tokenLineNo;      // Token line number
istream::int_type   nextCh;     // Next character
int           inputLineNo;      // Current input line number
map<string,string>  outRuleMap; // Output rule map

//-----------------------------------------------------------------------------

// READ A CHARACTER

void
```

```
GetChar ()
{
    if (nextCh == '\n') inputLineNo ++ ;
    nextCh = cin.get();
}

//---------------------------------------------------------------------------

// READ A TOKEN

void
GetToken ()
{
    while (isspace(nextCh)) GetChar ();
    tokenLineNo = inputLineNo;
    tokenVal = "";
    if (isalpha(nextCh)) {
        tokenType = WORD;
        do {
            tokenVal += static_cast<char>(nextCh);
            GetChar ();
        } while (isalpha(nextCh));
    } else {
        if (nextCh == ':') tokenType = COLON;
        else if (nextCh == ';') tokenType = SEMI;
        else if (nextCh == '|') tokenType = BAR;
        else if (nextCh == '&') tokenType = AMPERSAND;
        else if (nextCh == '(') tokenType = OPEN_BRACKET;
        else if (nextCh == ')') tokenType = CLOSE_BRACKET;
        else if (nextCh == '*') tokenType = ASTERISK;
        else if (nextCh == istream::traits_type::eof()) tokenType = END_OF_FILE;
        else {
            tokenType = INVALID_CHAR;
            throw Error_c (tokenLineNo, "invalid input character");
        }
        GetChar ();
    }
}

//---------------------------------------------------------------------------

// GET COMPONENT

void
GetComponent (
    vector<string>  *compVec)   // Returned alternative components
{
    void GetAlternativeList (vector<string> *questionVec);
    if (tokenType == WORD) {
        compVec->clear ();
        compVec->push_back (tokenVal);
        GetToken ();
    } else if (tokenType == ASTERISK) {
        compVec->clear ();
        compVec->push_back ("*");
        GetToken ();
    } else if (tokenType == OPEN_BRACKET) {
```

```
        GetToken ();
        GetAlternativeList (compVec);
        if (tokenType != CLOSE_BRACKET)
            throw Error_c (tokenLineNo, "mismatched parenthesis");
        GetToken ();
    } else
          throw Error_c (tokenLineNo, "invalid or missing component");
}

//---------------------------------------------------------------------------

// GET PHRASE

void
GetPhrase (
    vector<string>  *phraseVec) // Returned alternative phrases
{
    vector<string>  prevPhraseVec; // Previous phrase vector
    vector<string>  compVec;    // Component vector
    GetComponent (phraseVec);
    while (
        tokenType == WORD || tokenType == ASTERISK ||
        tokenType == OPEN_BRACKET
    ) {
        GetComponent (&compVec);
        prevPhraseVec = *phraseVec;
        phraseVec->clear ();
        for (size_t i = 0; i < prevPhraseVec.size(); i++)
            for (size_t j = 0; j < compVec.size(); j++)
                phraseVec->push_back (prevPhraseVec[i] + ' ' + compVec[j]);
    }
}

//---------------------------------------------------------------------------

// PERMUTE A KEY PHRASE SET

void
PermuteKeySet (
    vector<string>  *alternateVec,  // Returned alternate vector
    const vector<vector<string> > *keySet) // Key phrase set
{
    vector<string>  subAltVec;  // Sub-key alternate vector
    vector<vector<string> > subKeySet;  // Sub-key phrase set

    alternateVec->clear ();
    for (size_t i = 0; i < keySet->size(); i++) {
        subKeySet = *keySet;
        subKeySet.erase (subKeySet.begin() + i);
        if (subKeySet.size() != 0)
            PermuteKeySet (&subAltVec, &subKeySet);
        else
            subAltVec.push_back ("*");
        for (size_t j = 0; j < (*keySet)[i].size(); j++)
            for (size_t k = 0; k < subAltVec.size(); k++)
                alternateVec->push_back (
                    "* " + (*keySet)[i][j] + ' ' + subAltVec[k]);
```

```
    }
}

//---------------------------------------------------------------------------

// GET AN ALTERNATIVE

void
GetAlternative (
    vector<string>  *alternateVec)  // Returned alternate vector
{
    vector<vector<string> > keySet; // Key phrase set
    GetPhrase (alternateVec);
    if (tokenType == AMPERSAND) {
        keySet.push_back (*alternateVec);
        do {
            GetToken ();
            GetPhrase (alternateVec);
            keySet.push_back (*alternateVec);
        } while (tokenType == AMPERSAND);
        PermuteKeySet (alternateVec, &keySet);
    }
}

//---------------------------------------------------------------------------

// GET ALTERNATIVE LIST

void
GetAlternativeList (
    vector<string>  *questionVec)   // Returned question vector
{
    vector<string>  alternateVec;   // Alternatives vector
    GetAlternative (questionVec);
    while (tokenType == BAR) {
        GetToken ();
        GetAlternative (&alternateVec);
        for (size_t i = 0; i < alternateVec.size(); i++)
            questionVec->push_back (alternateVec[i]);
    }
}

//---------------------------------------------------------------------------

// EMIT AN ANSWER

void
EmitAnswer (
    const string    *question,      // Question string
    const string    *answer)        // Answer string
{
    string          normQuest;      // Normalised question
    string          upQuest;        // Up-shifted question
    string          upAnswer;       // Up-shifted answer
    enum {START, IN_WORD, END_WORD, STAR} state;

    // Normalise the question string
```

```
    state = START;
    for (size_t i = 0; i < question->length(); i++) {
        if (isalpha((*question)[i])) {
            if (state == END_WORD || state == STAR) normQuest += ' ';
            normQuest += (*question)[i];
            state = IN_WORD;
        } else if (isspace((*question)[i])) {
            if (state == IN_WORD) state = END_WORD;
        } else if ((*question)[i] == '*') {
            if (state == IN_WORD || state == END_WORD) normQuest += ' ';
            if (state != STAR) normQuest += '*';
            state = STAR;
        } else
            throw Error_c (tokenLineNo, "unexpected output char");
    }

    // Up-shift the question and answer

    for (size_t i = 0; i < normQuest.length(); i++)
        upQuest += normQuest[i] + (islower(normQuest[i]) ? 'A' - 'a' : 0);
    for (size_t i = 0; i < answer->length(); i++)
        upAnswer += (*answer)[i] + (islower((*answer)[i]) ? 'A' - 'a' : 0);

    // Process duplicate questions

    if (outRuleMap.count(upQuest) != 0) {
        if (outRuleMap[upQuest] != upAnswer)
            throw Error_c (tokenLineNo, "question has multiple answers");
    }

    // Emit the rule and save the answer

    else {
        cout << normQuest << '\t' << *answer << '\n';
        outRuleMap[upQuest] = upAnswer;
    }
}

//------------------------------------------------------------------------------

// MAIN LINE

int
main ()
{
    vector<string>  questionVec;    // Question vector
    string          answer;         // Answer to the question
    int             exitStatus;     // Exit status code

    // Initialise the token stream

    nextCh = 0;
    inputLineNo = 1;
    GetChar ();
    exitStatus = 0;
```

```
    // Process a rule list

    for (;;) {
        try {
            GetToken ();
            if (tokenType == END_OF_FILE) break;
            GetAlternativeList (&questionVec);
            if (tokenType != COLON)
                throw Error_c (tokenLineNo, "missing colon");
            GetToken ();
            if (tokenType != WORD)
                throw Error_c (tokenLineNo, "invalid answer");
            answer = tokenVal;
            GetToken ();
            while (tokenType == WORD) {
                answer += ' ' + tokenVal;
                GetToken ();
            }
            if (tokenType != SEMI)
                throw Error_c (tokenLineNo, "invalid answer");
            for (size_t i = 0; i < questionVec.size(); i++)
                EmitAnswer (&questionVec[i], &answer);
        } catch (Error_c error) {
            cerr << "Line " << error.errorLineNo << ": "
                << error.errorMessage << '\n';
            exitStatus = 1;
            if (tokenType != SEMI) {
                while (nextCh != ';' && nextCh != istream::traits_type::eof())
                    GetChar ();
                if (nextCh == ';') GetChar ();
            }
        }
    }
    return exitStatus;
}
```