Lecture 11

Stacks and Compilers

Stack

- Is a collection with one accessible element - the most recent element "pushed" onto the stack. – top
- Supports Last-In, First-Out (LIFO) operations. – push, pop

- Is used often in computer systems
  - compiler matches openers { [(and closers)] }
  - evaluating expressions in computer languages
  - maintain order of method calls.

Applications of Stacks in Compilers

- Examining programs to see if *symbols balance* properly. (Bracket matching)

  { a = (3 + 2); b = (12-4)/8 + (33-2/5; }

- Performing "*postfix*" calculations

  12 4 - 8 / ☛ 1

- Performing "*infix*" to "*postfix*" conversions

  (12-4)/8 ☛ 12 4 - 8 /

Balancing Symbols

- Compilers check your programs for syntax errors, but frequently a lack of one symbol (such as a missing { or }) may cause the compiler to spill out numerous lines of diagnostics without identifying the real error.

- A useful tools in this situation is a program that checks whether the symbols (),[], {} are balanced.

- A stack can be used to verify whether a program contains balanced parentheses, braces, and brackets. Push opening symbols onto a stack and see if closing symbols make sense.

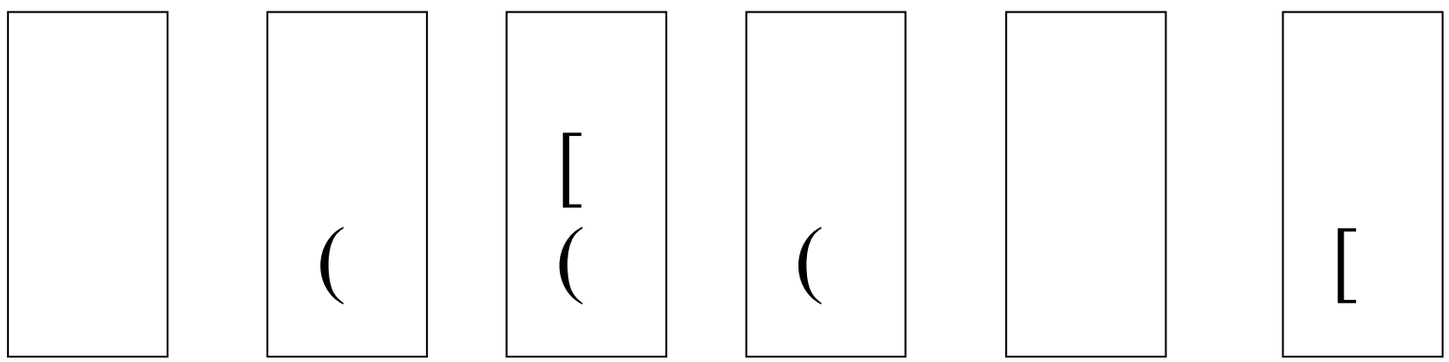
Basic Algorithm

1. Make an empty stack
2. Read the symbols one by one until the end of the file (eof)
   - if the symbol read is an opening symbol, *push* it onto the stack
   - if the symbol read is a closing symbol then
     - if the stack is empty, report an error
     - otherwise, *pop* the stack. If the symbol *popped* is not the corresponding opening symbol, report an error
3. At the end of the file, if the stack is not *empty*, report an error

Example:

([ ] }*)* [eof *

- the stack at different stages of the algorithm



Errors are indicated by *

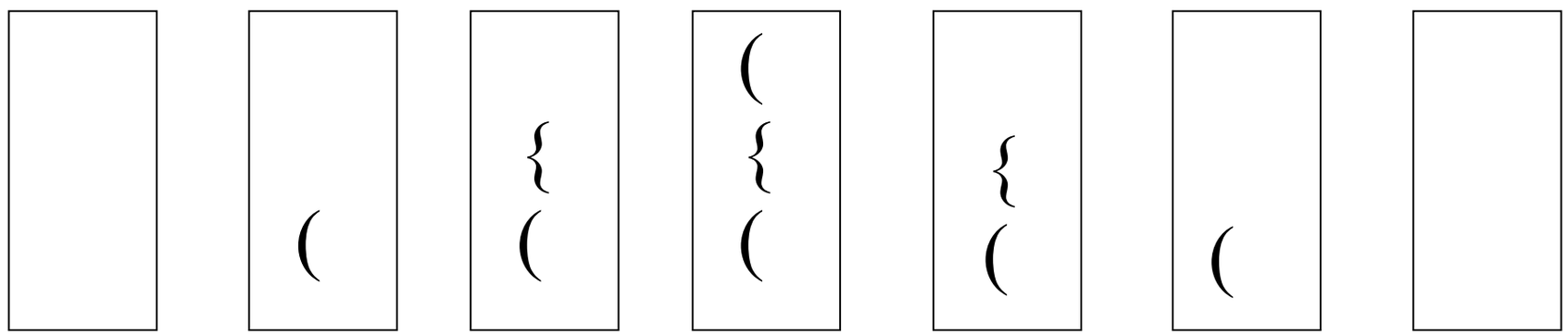
}* because) was popped when the corresonding symbol is {

)* because there was no matching opening symbol

eof* because the stack contains [unmatched at end of input

Example:

({ ( ) }))* eof

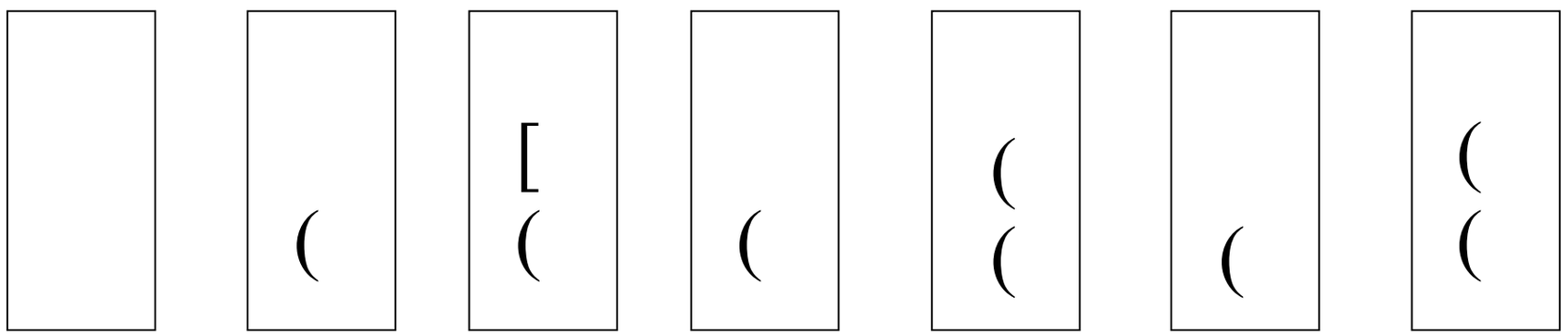


Errors are indicated by *

)* because there was no matching opening symbol

Example:

([ ] ( ) (eof *

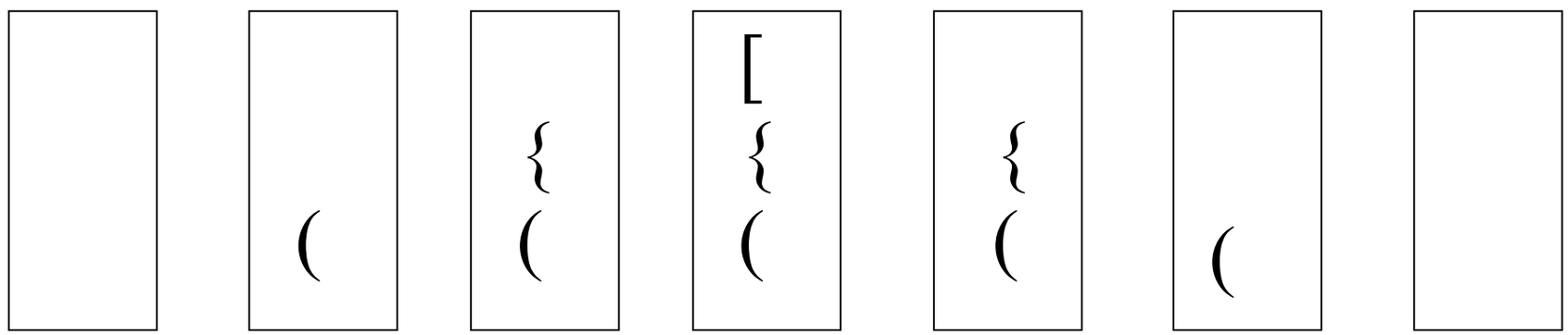


Errors are indicated by *

eof* because (and (are unmatched at end of input

Example:

({ [ ] }) eof



in.get(ch), in.putback(ch)

- istream in(cin);
- istream in.get(ch);
- istream in.putback(ch);

x=in.get()	Read one character from in and return its integer value; return EOF for end-of-file
in.get(c)	Read a character from in into c
in.get(p,n,t)	Read at most n characters from in into [p:...); consider t a terminator
in.get(p,n)	in.get(p,n,'\n')
in.putback(c)	Put c back into in's stream buffer
in.unget()	Back up in's stream buffer by one, so that the next character read is the same as the previous character

Chart is modified from Stroustrup, Bjarne (2013-07-10). The C++ Programming Language (4th Edition) (Kindle Location 39040). Pearson Education. Kindle Edition.

Enumerated types

using symbols instead of numbers for constant values improves the readability of your code

- Simplest way to create your *own* type
  - you declare an enumerated type by using the **enum** keyword
  - you list all the values (the values are called *enumerators*) the type can hold:

```
enum Seasons { Winter, Spring, Summer, Fall };
```

  - every enumerator is assigned an integer value, either explicitly or by default.

0 1 2 3

A collection of named integer constants

```
#define EOL 0
#define VALUE 1
#define OPAREN 2
```

enum:

```
enum TokenType { EOL, VALUE, OPAREN, CPAREN, EXP,
MULT, DIV, PLUS, MINUS };
```

It is possible to create explicit values:

```
enum seasons_t {spring = 10, summer = 100, fall = 50, winter = 5};
enum months_t {January = 1, February, March, April};
```

An alternative to if for multi-way branching if the condition being tested is equality for an integral type

```
enum Months { January = 1, February, March, April, May};
Months month = January;

switch (month)//expression must evaluate to an integral type
{
 case January:
   cout << "First month of the year!\n:";
 case February:
 case March:
   cout << "It is cold this month!\n";
   break;

 case April:
   cout << "Spring\n";
 default:
   cout << "One third of the year is over.\n";
   break;
}
```

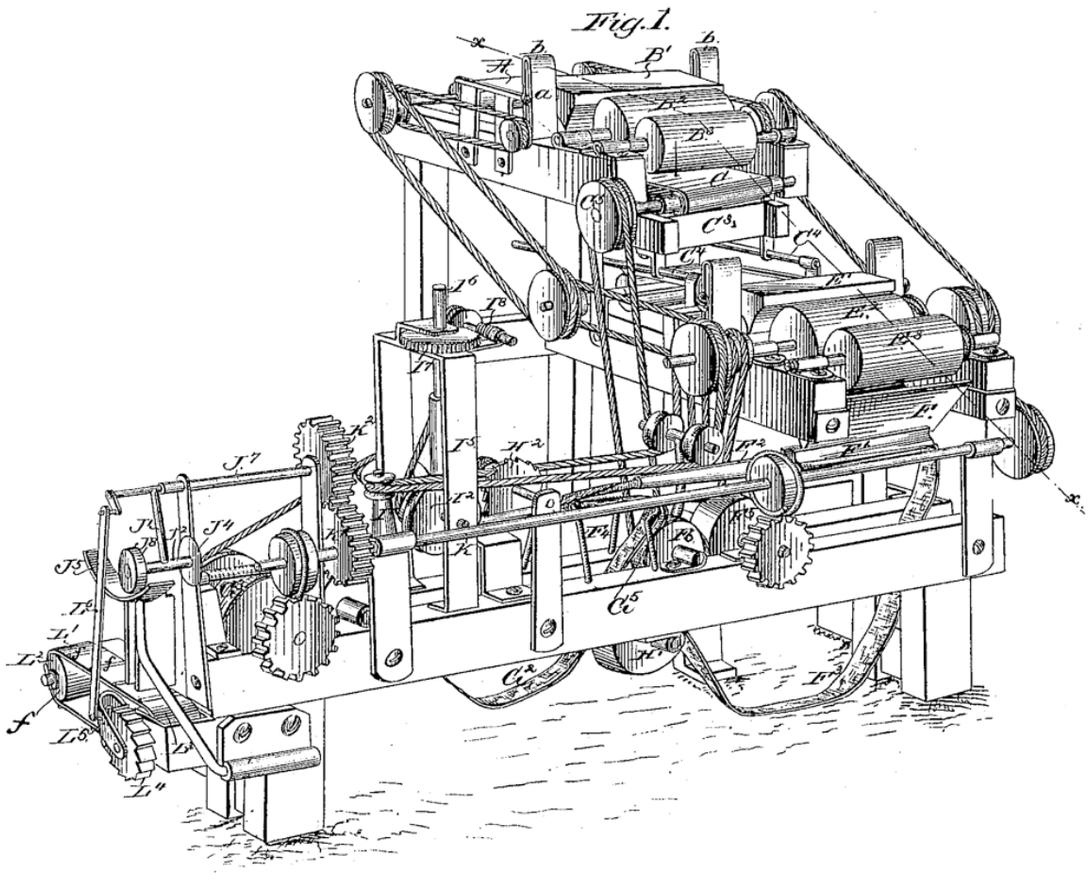
```
int main( )
{
    // print (“hello world!”)
    cout << “hello world!” << endl;

    cout << “Did you know that (5 + 3/(4 - 2) = ”
         << (5 + 3)/(4 - 2) << “?” << endl;
   /* Next time show that (3+2)*(4 = 20
      or 4 + (3040- 30)/2 = 1514 */

    return 0;
{
```

➡

```
(){()(){
```



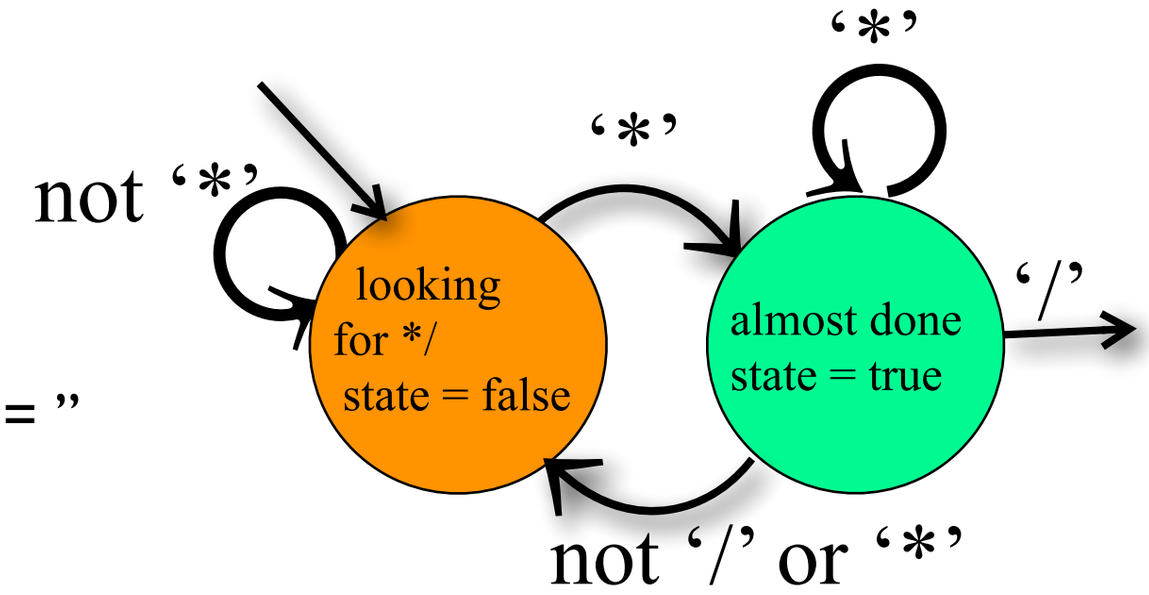
Machine picture from: http://upload.wikimedia.org/wikipedia/commons/c/ce/Bonsack_machine.png

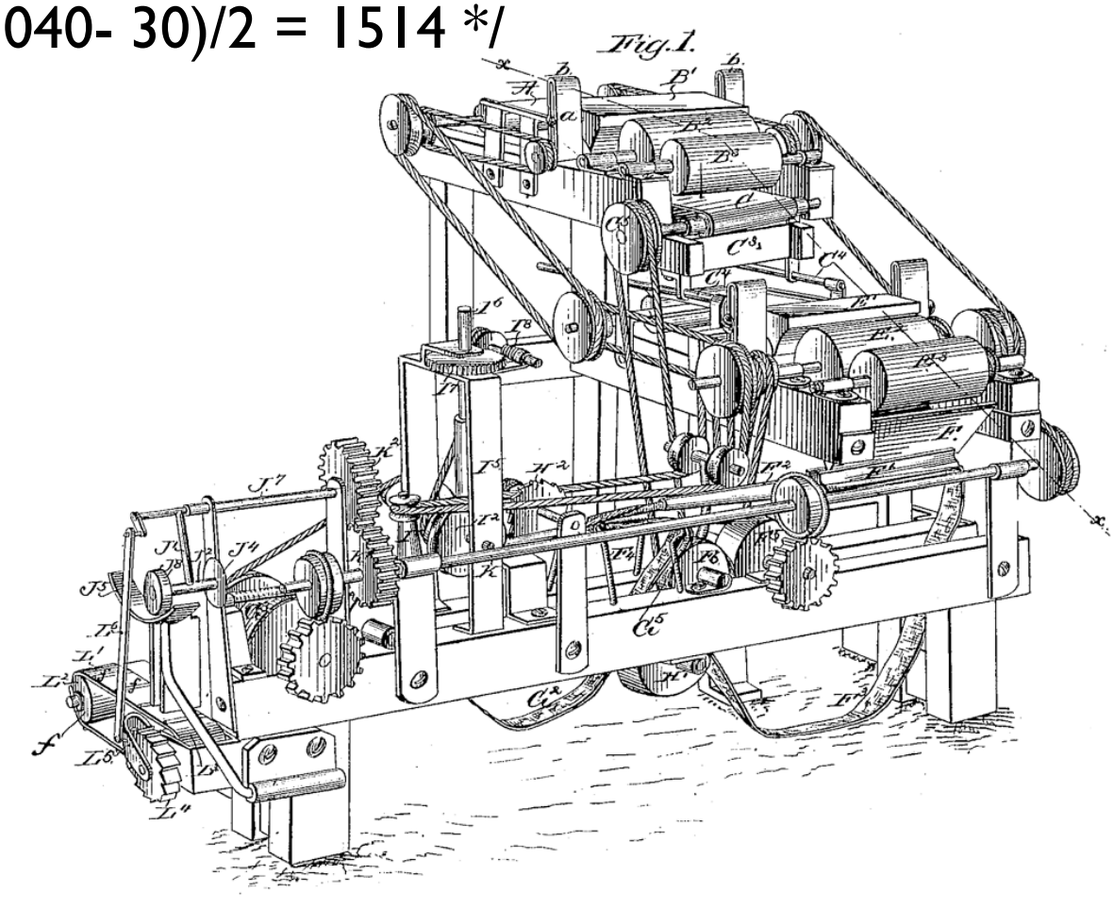
Tokenizer

```
int main( )
{
    // print (“hello world!”)
    cout << “hello world!” << endl;

    cout << “Did you know that (5 + 3/(4 - 2) = ”
         << (5 + 3)/(4 - 2) << “?” << endl;
   /* Next time show that (3+2)*4 = 20
      or 4 + (3040- 30)/2 = 1514 */

    return 0;
{
```





{,11),6 (,6),6 (,6 {,2),1 (,1

Machine picture from: http://upload.wikimedia.org/wikipedia/commons/c/ce/Bonsack_machine.png

The Tokenizer Class

The Tokenizer class provide a constructor that requires an istream and then provides a set of accessors that can be used to get:

- the next token (either an opening/closing symbol)
- the current line number
- the number of errors (mismatched quotes and comments)

```
class Tokenizer
{
public:
    Tokenizer( istream & input )
     : currentLine( 1 ), errors( 0 ), inputStream( input ) { }

     // The public routines
    char getNextOpenClose( );
    int getLineNumber( ) const{return currentLine;}
    int getErrorCount( ) const{return errors;}

private:
    enum CommentType { SLASH_SLASH, SLASH_STAR };

    istream & inputStream;     // Reference to the input stream
    char ch;                   // Current character
    int currentLine;           // Current line
    int errors;                // Number of errors detected

     // A host of internal routines
    bool nextChar( );
    void putBackChar( );
    void skipComment( CommentType start );
    void skipQuote( char quoteType );
};
```

```
int main( )
{
    char ch = '(';
     // print ("hello world!"
     cout << "hello world!" << endl;

     cout << "Did you know that (5 + 3/(4 - 2) = "
          << (5 + 3)/(4 - 2) << "?" << endl;
    /* Next time show that (3+2)*4 = 20
       or 4 + (3040- 30)/2 = 1514 **/

     return 0;
{
```

in.get(c)	Read a character from in into c
in.putback(c)	Put c back into in's stream buffer

```
// nextChar sets ch based on the next character in
// inputStream and adjusts currentLine if necessary.
// It returns the result of get.

bool Tokenizer::nextChar( )
{
    if( !inputStream.get( ch ) )
        return false;
    if( ch == '\n' )
        currentLine++;
    return true;
}


// putBackChar puts the character back onto inputStream.
// Both routines adjust currentLine if necessary.
void Tokenizer::putBackChar( )
{
    inputStream.putback( ch );
    if( ch == '\n' )
        currentLine--;

}
```

```
// Return the next opening or closing symbol or '\0' (if EOF).
// Skip past comments and character and string constants.
char Tokenizer::getNextOpenClose( )
{
    while( nextChar( ) )
    {
        if( ch == '/' )
        {
            if( nextChar( ) )
            {
                if( ch == '*' )
                    skipComment( SLASH_STAR );
                else if( ch == '/' )
                    skipComment( SLASH_SLASH );
                else if( ch != '\n' )
                    putBackChar( );
            }
        }
        else if( ch == '\'' || ch == '"' )
            skipQuote( ch );

        else if( ch == '(' || ch == '[' || ch == '{' ||
                 ch == ')' || ch == ']' || ch == '}' )
            return ch;
    }
    return '\0';       // End of file
}
```

skip past comments

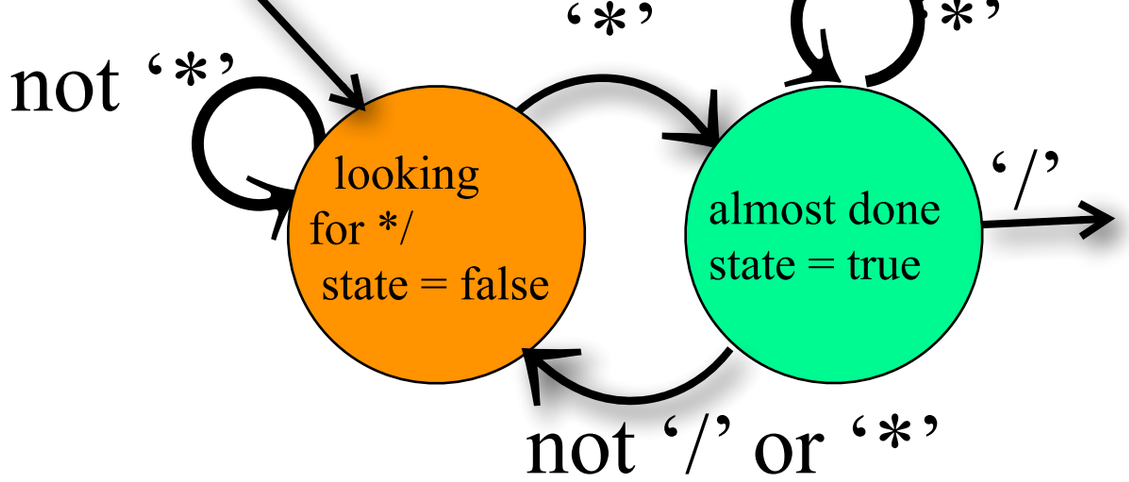
skip past quotes

return ch found paren!

```
int main( )
{
    char ch = '(';
    // print ("hello world!"
    cout << "hello world!" << endl;

    cout << "Did you know that (5 + 3/(4 - 2) = "
         << (5 + 3)/(4 - 2) << "?" << endl;
   /* Next time show that (3+2)*4 = 20
      or 4 + (3040- 30)/2 = 1514 **/

    return 0;
{
```




```
int main( )
{
    char ch = '(';
     // print ("hello world!"
     cout << "hello world!" << endl;

     cout << "Did you know that (5 + 3/(4 - 2) = "
             << (5 + 3)/(4 - 2) << "?" << endl;
    /* Next time show that (3+2)*4 = 20
       or 4 + (3040- 30)/2 = 1514 **/

     return 0;
{
```

```
// Precondition: We are about to process a comment;
//                  have already seen comment start token.
// Postcondition: Stream will be set immediately after
//                  comment ending token.
void Tokenizer::skipComment( CommentType start )
{
    if( start == SLASH_SLASH )
    {
        while( nextChar( ) && ( ch != '\n' ) )
            ;
        return;
    }

  // Look for */
    bool state = false;    // Seen first char in comment ender.

    while( nextChar( ) )
    {
        if( state && ch == '/' )
            return;
        state = ( ch == '*' );
    }

    cout << "Unterminated comment at line " << getLineNumber( ) << endl;
    errors++;
}
```

skip past comment starting with //

skip past comment starting with /* and ending with */

error

```
int main( )
{
    char ch = '(';
     // print ("hello world!"
     cout << "hello world!" << endl;

     cout << "Did you know that (5 + 3/(4 - 2) = "
          << (5 + 3)/(4 - 2) << "?" << endl;
    /* Next time show that (3+2)*4 = 20
       or 4 + (3040- 30)/2 = 1514 **/

     return 0;
{
```

```
// Precondition: We are about to process a quote; have already seen beginning quote.
// We assume all quotes are on a single line
// Postcondition: Stream will be set immediately after matching quote.
void Tokenizer::skipQuote( char quoteType )
{
    while( nextChar( ) )
    {
        if( ch == quoteType )
            return;
        if( ch == '\n' )
        {
            cout << "Missing closed quote at line "
                 << ( getLineNumber( ) - 1 ) << endl;
            errors++;
            return;
        }
        // If a backslash, skip next character.
        else if( ch == '\\' )
            nextChar( );
    }
}
```