Program Development Tools

- lex
- makefiles
- vi and gvim
- ctags
- source level debugging
- diff and cmp
- svn
- gprof

Lexical Analyzers

- A lexical analyzer reads in a stream of characters as input and produces a sequence of symbols called tokens as output.
- Useful for a variety of tasks.
- Tools exist to automatically create a lexical analyzer from a specification.

Lexical Analysis Terms

- A token is a group of characters having a collective meaning (e.g. id).
- A lexeme is an actual character sequence forming a specific instance of a token (e.g. **`num`**).
- A pattern is the rule describing how a particular token can be formed (e.g. [A-Za-z_][A-Za-z_0-9]*).
- Characters between tokens are called whitespace (e.g.blanks, tabs, newlines, comments).

Attributes for Tokens

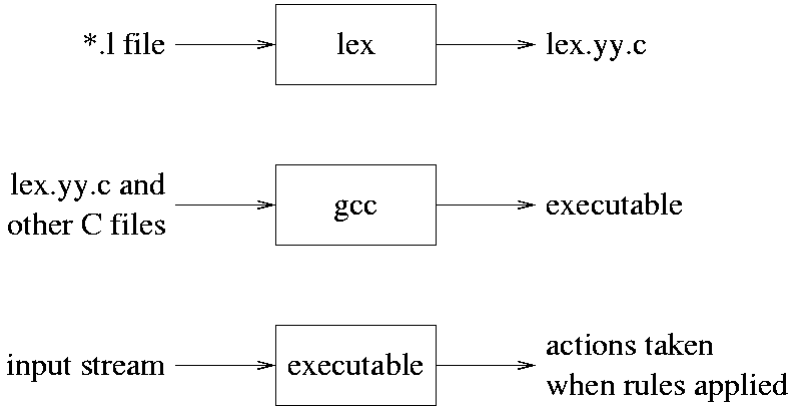
- Tokens can have attributes that can be passed back to the function calling the lexical analyzer.
  - Constants
    - value of the constant
  - Identifiers
    - pointer to a location where information is kept about the identifier

General Approaches to Implementing Lexical Analyzers

- Use a lexical-analyzer generator, such as Lex.
- Write the lexical analyzer in a conventional programming language.
- Write the lexical analyzer in assembly language.

Lex - A Lexical Analyzer Generator

- Can link with a lex library to get a main routine.
- Can use as a function called yylex().
- Easy to interface with yacc.



Lex Specifications

Lex Source

{ definitions }
%%
{ rules }
%%
{ user subroutines }

Definitions

declarations of variables, constants, and regular definitions

Rules

regular expression action

Regular Expressions

operators "\ [ ] ^ -? . * + | ( ) $ / { }
actions C code

Lex Regular Expression Operators

- “s” string s literally
- \c character c literally (used when c would normally be used as a lex operator)
- [s] for defining s as a character class
- ^ to indicate the beginning of a line
- [^s] means to match characters not in the s character class
- [a-b] used for defining a range of characters (a to b) in a character class
- r? means that r is optional

Lex Regular Expression Operators (cont.)

- . means any character but a newline
- r* means zero or more occurances of r
- r+ means one or more occurances of r
- r1|r2 r1 or r2
- (r) r (used for grouping)
- $ means the end of the line
- r1/r2 means r1 when followed by r2
- r{m,n} means m to n occurences of r

Example Regular Expressions in Lex

a*	zero or more a's
a+	one or more a's
[abc]	a, b, or c
[a-z]	lower case letter
[a-zA-Z]	any letter
[^a-zA-Z]	any character that is not a letter
a.b	a followed by any character followed by b
ab|cd	ab or cd
a(b|c)d	abd or acd
^B	B at the beginning of line
E$	E at the end of line

Lex Specifications (cont.)

Actions

Actions are C source fragments. If it is compound or takes more than one line, then it should be enclosed in braces.

Example Rules

```
[a-z]+          printf("found word\n");
[A-Z][a-z]*     {  printf("found capitalized word\n");
                   printf{"  %s\n", yytext);
                }
```

Definitions

```
name            translation
```

Example Definition

```
digits          [0-9]
```

Example Lex Program

```
digits [0-9]
ltr    [a-zA-Z]
alpha  [a-zA-Z0-9]
%%
[-+]{digits}+        |
{digits}+            printf("number: %s\n", yytext);
{ltr}(_|{alpha})*    printf("identifier: %s\n", yytext);
"'"."'"              printf("character: %s\n", yytext);
.                    printf("?: %s\n", yytext);
```

Prefers longest match and earlier of equals.

Another Example Lex Program

```
%%
BEGIN              { return (1); }
END                { return (2); }
IF                 { return (3); }
THEN               { return (4); }
ELSE               { return (5); }
letter(letter|digit)*  { return (6); }
digit+             { return (7); }
<                  { return (8); }
<=                 { return (9); }
=                  { return (10); }
<>                 { return (11); }
>                  { return (12); }
>=                 { return (13); }
```

Make and Makefiles

- The make utility is used to:
  - Automate the execution of commands for file generation.
  - Minimize the number of commands needed for rebuilding a target.
- A makefile describes
  - a hierarchy of dependencies between individual files
  - commands to generate each file

Make and Makefiles (cont.)

- There can be one or more target entries in a makefile. Each target entry in a makefile has the following format:

```
target : dependency_file ...
         command
         ...
```

- The target is a file. There can be one or more dependency files on which the target depends. There can be one or more commands each preceded by a tab that comprise a rule for the target. These commands are used to create or regenerate the target file.

Make and Makefiles (cont.)

- A target is remade when the target file either does not exist or has an older date/time than one or more of the dependency files.
- The targets and dependency files comprise a DAG structure representing the dependencies between the different components.
- The make utility recursively checks each target against its dependencies, starting with the first target entry in the makefile.

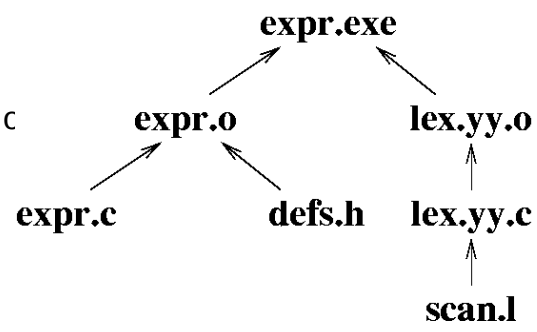
Example Makefile

```
expr.exe : expr.o lex.yy.o
     gcc -g -o expr.exe expr.o lex.yy.o

expr.o : expr.c defs.h
     gcc -g -c expr.c

lex.yy.o : lex.yy.c
     gcc -g -c lex.yy.c

lex.yy.c : scan.l
     lex scan.l
```



Invoking Make

- General form.

  make [-f *makefilename*] [*target*]

- If no *makefilename* is given, then make looks for a file called *makefile* or *Makefile* in that order. Make uses one of these files if found in the current directory.
- By default make attempts to create the first target in the file. Alternatively, a user can specify a specific *target* within the makefile to make.

Example Invocations of Make

```
# Make the first target in makefile or Makefile.
make

# Make the lex.yy.o target in makefile or Makefile.
make lex.yy.o

# Make the first target in the makeexpr makefile.
make -f makeexpr

# Make the expr.o target in the makeexpr makefile.
make -f makeexpr expr.o
```

Defining Symbols in a Makefile

- You can define a symbol in a makefile and reference the symbol in multiple places.
- General form of the definition.

  symbol = definition

- General form of the reference.

  $(*symbol)*

Example Makefile with Symbols

```
CC = gcc
CFLAGS = -g -c

expr.exe : expr.o lex.yy.o
    $(CC) -g -o expr.exe expr.o lex.yy.o

expr.o : expr.c defs.h
    $(CC) $(CFLAGS) expr.c

lex.yy.o : lex.yy.c
    $(CC) $(CFLAGS) lex.yy.o

lex.yy.c : scan.l
    lex scan.l
```

The Vi Editor (17.1)

- Vi stands for the VIsual editor.
- Why use the *vi* editor?
  - The *vi* editor is standard on every Unix system.
  - The *vi* editor allows you to use ex line commands.
  - The *vi* editor has many special features that are very useful.
  - The *vi* editor is efficient compared to emacs.

Invoking Vi

- The vi editor is invoked by issuing the command in the following form. The -r option is for recovering a file where the system crashed during a previous editing session. The -t option is to indicate the position within a file the editing should start. The use of tags will be discussed more later.

  vi [-t *tag*] [-r] *filename*

Vi Modes

- The *vi* editor has three main modes:
  - character input mode: where text can be entered
    - insert, append, replace, add lines
  - window mode: where regular commands can be issued
    - basic cursor motions
    - screen control
    - word commands
    - deletions
    - control commands
    - miscellaneous commands
  - line mode: where *ex* commands can be issued

Vi Character Input Mode

- After invoking *vi*, the user is in the window command mode. There are a few different commands to enter character input mode. At that point a user types in the desired text. A user selects the ESC key to return back to command mode.

Vi Commands to Go into Character Input Mode

- The following commands are used to go into character input mode. All but the *r* command require the user to hit the ESC key to go back into window command mode.

a	append text after the cursor position
A	append text at the end of line
i	insert text before the cursor position
I	insert text before the first nonblank character in the line
o	add text after the current line
O	add text before the current line
r*chr*	replace the current character with *chr*
R	replace text starting at the cursor position

Vi Basic Cursor Motions

- The basic cursor motions allow you to move around in the file. The letter options allow movement when the arrow keys are not defined.

h ←	go back one character
j ↓	go down one line
k ↑	go up one line
l →	go forward one character (space also works)
+ CR	go down one line to first nonblank character
–	go up one line to first nonblank character
0	go to the beginning of the line
$	go to the end of the line
H	go to the top line on the screen
L	go to the last line on the screen

Vi Word Movements

- The following commands can be used to position the cursor.

w	position the cursor at the beginning of the next word
b	position the cursor at the beginning of the previous word
e	position the cursor at the end of the current word

Vi Screen Control

- The following commands can be used to control the screen being displayed.

^U	scroll up one half page
^D	scroll down one half page
^B	scroll up one page
^F	scroll down one page
^L	redisplay the page

Deletions in Vi

- The following *vi* commands can be used to delete text.

dd	delete the current line
D	delete text from the cursor to the end of the line
x	delete character at the cursor
X	delete character preceding the cursor
dw	delete characters from the cursor to the end of the word

Searching in Vi

- The following *vi* commands can be used to search for text patterns, where each pattern is a regular expression.

/*pattern*	search forward for the specified pattern
/	search forward for the last specified pattern
?*pattern*	search backward for the specified pattern
?	search backward for the last specified pattern
n	perform the last / or ? command

Miscellaneous Vi Commands

- Below are some miscellaneous *vi* commands. Commands that delete text also place the deleted text into a buffer.

u	undo previous command
U	restore entire line
Y	save current line into buffer
p	put saved buffer after cursor position
P	put saved buffer before cursor position
J	join current line with following line
%	position cursor over matching (,), {, or }
ZZ	save file and exit (same as :wq)

Repeating Vi Commands

- You can indicate how many times a particular command is to be performed.
- Examples:

3dd	delete 3 lines
4w	advance 4 words
7x	delete 7 characters
5n	perform last search 5 times

Vi Line Command Mode

- You can enter line command mode by typing a colon. At that point you can enter *ex* commands.
- Examples:

:150	# go to line 150
:+50	# go forward 50 lines
:1,.d	# delete lines from line 1 to current line
:1,$ s/old/new/g	# perform substitutions on the entire file
:w [*filename*]	# write file [to specified *filename*]
:q	# quit editing session
:wq	# write file and quit
:q!	# quit and don't save changes

Ctags: Create a Tags File for Use with Vi

- *ctags* is a Unix utility that takes in a set of source files as input and creates a *tags* file as output.
- The *tags* file contains for each function and macro:
  - Object name.
  - File in which the object is defined.
  - Pattern describing the location of the object.
- General form:

  ctags *list_of_files*

Using the Tags File

- You can use the tags file with the *vi* (*vim*, *gvim*) editors.
- Use the -t option when invoking *vi*. The editor will look in the current directory for the *tags* file and attempt to find the location of the function or macro in the appropriate file. If it finds it, then the file is opened and the cursor is positioned to that location.

  vi -t *tag*
- Examples:

  vi -t main

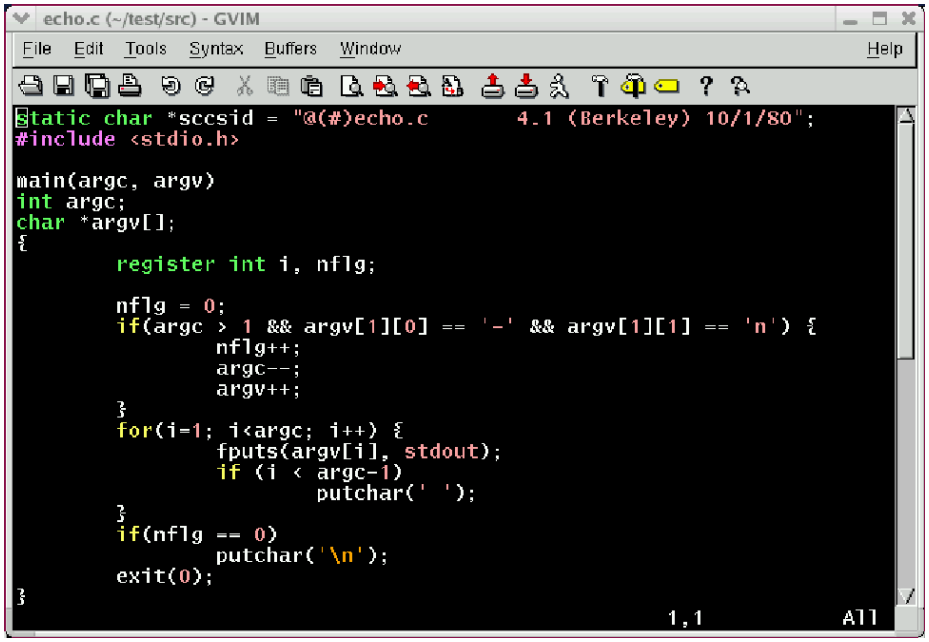
  vi -t max

Vim: Vi IMproved, a Programmer's Text Editor

- Vim is a text editor that is upwards compatible with vi.
- Enhancements over vi:
  - multi-level undo
  - syntax highlighting
  - on-line help
  - visual selection
  - tag searching support

Gvim: GUI Vim

- Below is a snapshot of using gvim.



Multi-Level Undo in Vim

- Can use the “u” command to undo multiple changes, as opposed to vi, which can only undo the last change. Each time you enter “u”, the previous change is undone.
- Can redo the last undone change by using the CTRL-R command. Each time you enter CTRL-R, the last undone change is reperformed.

Syntax Highlighting in Vim

- Syntax highlighting allows vim to show parts of the text in another font or color.
- The rules for syntax highlighting are stored in a file. So vim can always be easily extended to support syntax highlighting in new types of files. Vim currently supports syntax highlighting for files with the following extensions:

  *.c, *.cpp, *.pl, *.sh, *.csh, *.java, *.html, *.tex, *.tr

Visual Selection in Vim

- Select text portions using the left mouse button.
- Some of the commands you can perform on visually selected portions include:

d	delete the highlighted text
y	place in a buffer
U	make uppercase
u	make lowercase
gq	format lines to width of window

- Can change the position of the cursor by clicking on a specific location in the window.
- Can use the middle mouse button to insert text selected by the left mouse button.

Tag Searching Support in Vim

- Can position the cursor over a name, such as the name of a function.
- Typing CTRL-] will cause vim to search in the tags file for the name under the cursor. If found, then the file containing the function or macro name is opened and the cursor is positioned to that function.
- This command is equivalent to exiting vim and typing

  vi -t *name*

On-line Help in Vim

- You can type “:help” to get online help in Vim.
- There are several keywords or files that can be selected within the on-line help to learn about specific vim features.

The .gvimrc File

- The .gvimrc file is used to initialize various settings.
- Example commands often used in a .gvimrc file:

```
:set incsearch          # shows text where the pattern has
                        # matched so far when typing a search
                        # command
:win width height       # set the width and the height in
                        # characters of the window
:syntax on              # enable syntax highlighting
:set background=dark    # set background of window to be
                        # dark (default text will be white)
```