PURDUE UNIVERSITY®

CS 25200: Systems Programming

Lecture 12: Lexical Analysis and Parsing

Prof. Turkstra

Feasting with faculty

- Tuesdays at 12:00pm, Earhart Private Dining Room A
- Always welcome – no invitation needed

- Will still send out weekly batches of invitations
  - You don’t have to wait to get one!

Office Hours

- Wednesday office hours are canceled this week!

Lecture 12

- lex
- yacc
- Our shell

Lexical analysis

- Process of converting a sequence of characters into tokens
  - Token: string that has “meaning”
- Tokens are described using regular expressions
- Many uses
  - First phase of a compiler frontend
  - Configuration file parsing
  - Shell command parsing

lex

- Reads a stream of characters, recognizes tokens, and takes an action

```
%{
#include <stdio.h>
%}

%%
hello    printf("Howdy!\n");
bye      printf("Please don't leave me!\n");
%%
```

lex

- Sections are separated by %%
- First section is included directly in the output (verbatim)
- Format is
  `pattern action`
  - When pattern is matched, action is executed

```
$ lex example.l
$ gcc -lfl -o example lex.yy.c
```

Another example

```
%{
#include <stdio.h>
%}

%%
[0-9]+              printf("NUMBER\n");
[A-z][A-z0-9]*      printf("WORD\n");
%%
```

shell.l

- Current version defines a handful of input tokens
- You will have to add additional tokens to support the full syntax

```
“>>”    { return APPEND_STDOUT; }
”|”     { return PIPE; }
”&”     { return BACKGROUND; }
```

- …etc

Parsing

- The process of analyzing symbols (tokens) conforming to the rules of a formal grammar
- Often results in a parse tree

yacc

- YACC parses tokens with certain values
- YACC does not know anything about input streams
  - That's why we have lex
- YACC does not like ambiguity
  - shift/reduce conflicts
  - reduce/reduce conflicts

Example

- Suppose we have a vehicle
- engine on
  engine off
  set speed 50
  brake

lex

```
%{
#include <stdio.h>
#include “y.tab.h”
%}

%%
[0-9]+    return NUMBER;
engine    return TOKENGINE;
on|off    return STATE;
set       return TOKSET;
speed     return TOKSPEED;
brake     return BRAKE;
\n        return NEWLINE;
[ \t]+    /* ignore whitespace */
%%
```

y.tab.h?

- Generated from our grammar by yacc

```
$ bison -y -d car.y
```

```
commands: /* empty */
        | commands command
        ;

command:
        engine_switch
        | speed_set
        | brake
        ;

engine_switch:
        TOKENGINE STATE NEWLINE
        { printf("\tEngine state inverted\n"); }
        ;

speed_set:
        TOKSET TOKSPEED NUMBER NEWLINE
        { printf("\tSpeed set\n"); }

brake:
        BRAKE NEWLINE
        { printf("\tBreaking!\n"); }
        ;
```

Compiling

```
$ lex car.l
$ bison -y -d car.y
$ gcc lex.yy.c y.tab.c -o car
```

Header

```
%{
#include <stdio.h>
#include <string.h>

extern int yyparse();
extern int yylex();

void yyerror(const char *str)
{
        fprintf(stderr,"error: %s\n",str);
}

int yywrap()
{
        return 1;
}

int main()
{
        yyparse();
}

%}

%token NUMBER TOKENGINE STATE TOKSET TOKSPEED BRAKE NEWLINE ERR

%%
```

%token

- Be sure to include your token names in shell.y!

```
%token NOTOKEN, GREAT, NEWLINE, WORD,
GREATGREAT, PIPE, AMPERSAND, etc
```

Improved

```
%{
#include <stdio.h>
#include "y.tab.h"
%}

%%
[0-9]+    yylval=atoi(yytext); return NUMBER;
engine    return TOKENGINE;
on|off    yylval=!strcmp(yytext, "on"); return STATE;
set       return TOKSET;
speed     return TOKSPEED;
brake     return BRAKE;
\n        return NEWLINE;
[ \t]+    /* ignore whitespace */
.         return ERR;
%%
```

```
commands: /* empty */
        | commands command
        ;

command:
        engine_switch
        | speed_set
        | brake
        ;

engine_switch:
        TOKENGINE STATE NEWLINE
        {
          if ($2)
            printf("\tEngine is on\n");
          else
            printf("\tEngine is off\n");
        }
        ;

speed_set:
        TOKSET TOKSPEED NUMBER NEWLINE
        { printf("\tSpeed set to %d\n", $3); }

brake:
        BRAKE NEWLINE
        { printf("\tBreaking!\n"); }
        ;
```

shell.y

- Let’s take a look at shell.y
- %union

Our shell grammar

- Our shell will recognize statements of the following format:

```
cmd [arg]* [| cmd [arg]*]*[[> filename][< filename][>& filename][>> filename][>>& filename]]* [&]
```

- Some examples to consider...

```
$ ls -al
$ ls -al > out
$ ls -al | sort >& out
$ awk -f x.awk | sort -u < infile > outfile &
```

Our shell grammar

single_command_list

```
cmd [arg]* [| cmd [arg]*]*
```

single_command argument_list

io_modifier_list

```
[[> filename][< filename][>& filename][>> filename][>>& filename]]*
```

io_modifier

```
[&]
```

background

Shell rules

```
goal: entire_command_list;

entire_command_list:
    entire_command_list entire_command
    | entire_command
    ;

entire_command:
    single_command_list io_modifier_list background NEWLINE
    | NEWLINE
    ;

single_command_list:
     single_command_list PIPE single_command
     | single_command
     ;
```

```
single_command:
    executable argument_list
    ;

argument_list:
    argument_list argument
    | /* can be empty */
    ;

executable:
    WORD
    ;

argument:
    WORD
    ;
```

```
io_modifier_list:
    io_modifier_list io_modifier
    | /* can be empty */
    ;

io_modifier:
    STDOUT WORD
    | APPEND_STDOUT WORD
    | STDERR WORD
    | STDOUT_STDERR WORD
    | APPEND_STDOUT_STDERR WORD
    | STDIN WORD
    ;

background:
    AMPERSAND
    | /* can be empty */
    ;
```

Need to add actions

```
io_modifier:
    STDOUT WORD {
      g_current_command->out_file = $2;
    }
    | APPEND_STDOUT WORD
    | STDERR WORD
    | STDOUT_STDERR WORD
    | APPEND_STDOUT_STDERR WORD
    | STDIN WORD
    ;
```

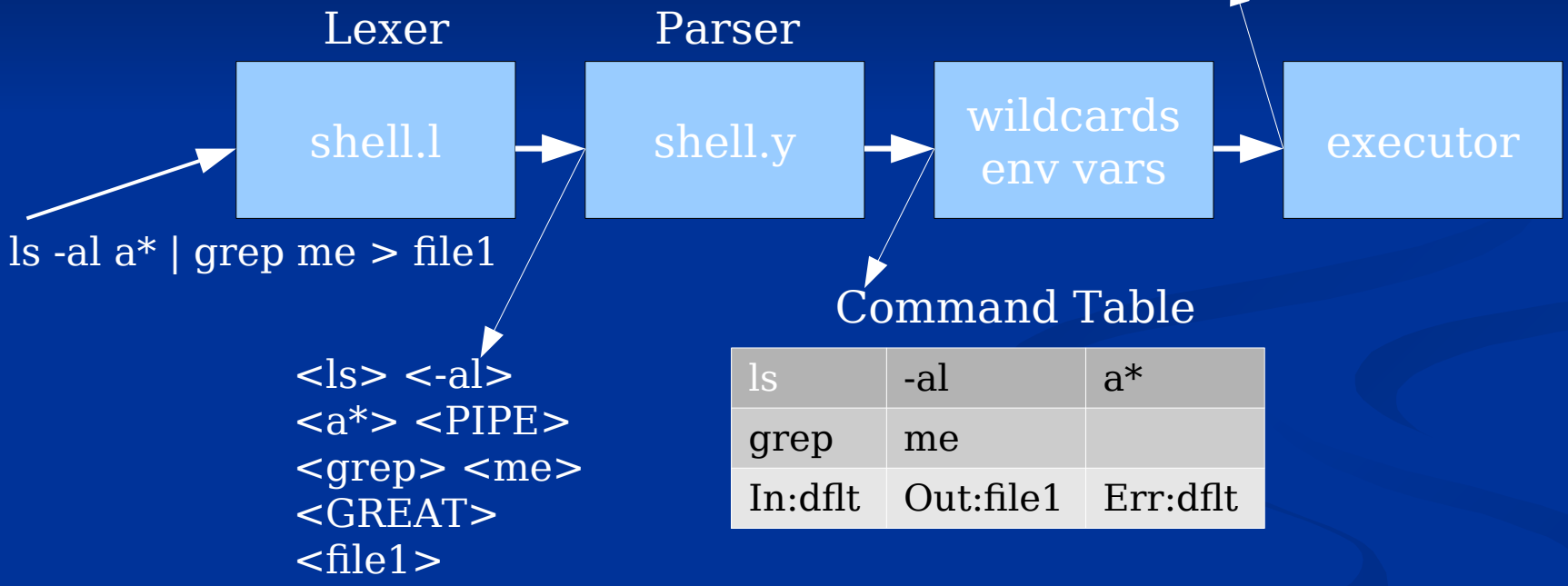
```
executable:
    WORD {
      g_current_single_command = malloc(...);
      create_single_command(...);
      insert_argument(...);
    }
    ;

single_command:
    executable argument_list {
      insert_single_command(...);
      g_current_single_command = NULL;
    }
    ;
```

```
entire_command_list:
    entire_command_list entire_command {
      execute_command(...);
      g_current_command = malloc(...);
      create_command(...);
    }
    | entire_command {
      // same
    }
    ;
```

Shell

Final Command Table

ls	-al	aab	aaa
grep	me		
In:dflt	Out:file1	Err:dflt	



Shell

- Command loop is implemented in the grammar itself
- The error token is a special token used for error recovery
  - Parses all tokens until a known token is found (NEWLINE)
  - yyerrok tells parser we recovered from the error
- Must add actions {…} in the grammar to fill the command table

```
arg_list:
    arg_list WORD { insert_argument(cur_cmd, $2); }
    | /* empty */
    ;
```

Questions?