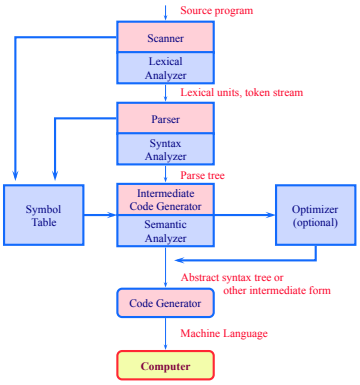
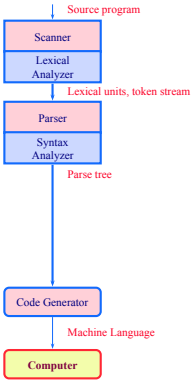
CSCI: 4500/6500 Programming Languages

Lex & Yacc

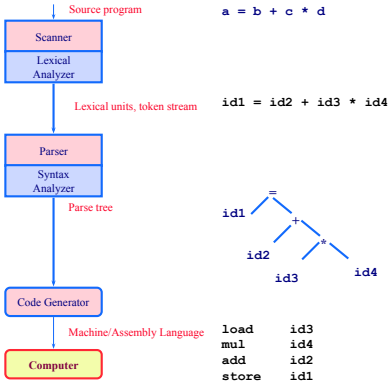
Big Picture: Compilation Process



Big Picture: Compilation Process



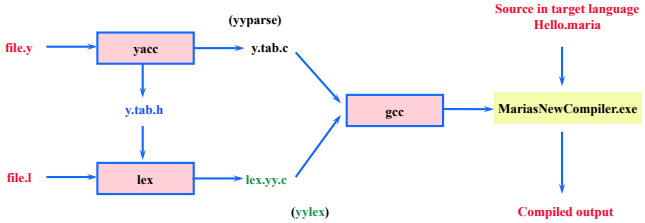
Big Picture: Compilation Process



General Process

- **1975 Lex & YACC automated the compilation process**
  - **The GNU version of these are called flex and bison and are free.**
- **Lex take patterns and generate code for a lexical analyzer or scanner.**
  - **Converts strings of input using user defined patterns to tokens.**
- **Yacc reads user specified grammars to generate code for a syntax analyzer or a parser, then the parser 'compiles' your program in your language**
  - **The syntax analyzer uses grammar rules that allow it to analyze tokens from the lexical analyzer and create a syntax tree (a hierarchical data structure).**
  - **CC.**
- **Final step - code generation, does a depth-first walk of the syntax tree to generate code (e.g., in machine code).**

Overview



- **Pattern matching rules for tokens in file.l (defines the vocabulary of the language)**
  - **Tells `lex` what the strings/symbols of the language look like and so it can convert them to tokens (enters them into the symbol table, with attributes, such as data type, e.g., integer) which yacc understands**
- **Grammar rules for language in file.y**
  - **Tells yack what the grammar rules are so it can analyze the tokens that it got from lex and creates a syntax tree.**

Lex and Yacc

- Lex and Yacc are tools for generating language parsers
- Each do a "single function"
- This is what they do:
  - » "Get a token from the input stream (stream of characters)" and
  - » "Parse a sequence of tokens to see if it matches a grammar"
- Yacc generates a parser (another program).
  - » Calls Lex generated "tokenizer function" (yylex()) each time it wants a token.
  - » You can define actions for particular grammar rules. For example, it can:
    - – print the string "match" if the input matched the grammar it expected (or something more complex)

What is Lex ?

- Lex is a lexical analyzer generator (or tokenizer). It enables you to define the 'vocabulary' of the language.
  - » How?: It automatically "generates" a lexer or scanner given a lex specification file as input (.l file)
- Purpose: Breaks up the input stream into tokens and it sees a group of characters that match a key, takes a certain action.
  - » For example consider breaking up a file containing the story "Moby Dick" into individual words
    - – Ignore white space
    - – Ignore punctuation
- Generates a C source file, e.g., maria.c, example1.c
  - » contains a function called yylex() that obtains the next valid token in the input stream.
  - » this source file in turn can then be compiled by the C compiler (e.g., gcc) to machine/assembly code.

Lex's Input File Syntax

- Lex input file consist of up to three sections
  - » Lex and C definition section that can be used in the middle section. C definitions are wrapped in %{ and %}
  - » Pattern action pairs, where the pattern is a regular expression and the action is in C syntax
  - » Supplementary C-routines (later)

```
… definitions ….
%%
… patterns    rules …
%%
… subroutines …
```

example1.l

```
%{
#include <stdio>
%}
%%
Stop            printf("Stop command received");
Start           printf("Start command received");
%%
```


Lex Syntax

- Lex input file consist of up to three sections ('%%')

```
%{
%}
```
- .c is generated after running
  - This part will be embedded into *.c
  - Substitutions, code and start states; will be copied into *.c

```
%%
```
- Define how to scan (pattern) and what action to take for each lexeme that translates to a token

```
%%
```
- Any user code for example main() calls yylex() one is provided by default.

http://www.cs.uga.edu/~maria/classes/4500-Spring-2012/lexyacc.zip

example1.l

```
%{
#include <stdio.h>
%}
%%
Stop            printf("Stop command received");
Start           printf("Start command received");
%%
```

Maximize compatibility to ATT's lex implementation

Write to standard output instead of lex.yy.c

```
{atlas:maria:255} flex -l -t example1.l > example1.c
{atlas:maria:257} gcc example1.c -o example1 -lfl
{atlas:maria:261} example1
hello
hello
Start
Start command received
^D
```

You may wonder how the program runs, as we didn't define a main() function. This function is defined for you in libl (liblex) which we compiled in with the -lfl command.

Regular expressions

example2.l

```
%{
#include <stdio.h>
%}
%%
[01234567890]+                  printf("NUMBER\n");
[a-zA-Z][a-zA-Z0-9]*            printf("WORD\n");
%%
```

```
{atlas:maria:422} flex -l -t example2.l > example2.c
{atlas:maria:423} gcc example2.c -o example2 -lfl
{atlas:maria:424} example2
hello
WORD

5lkfsj
NUMBER
WORD

lkjklj3245
WORD
```

Make sure that you do not create zero length matches like '[0-9]*' - your lexer might get confused and start matching empty strings repeatedly.

More Complicated (C-like) Syntax

- Supposing we know what we want the Syntax to look more C like…


```
logging
        {
        category lame-servers { null; };
        category cname { null; };
        };
zone "."
        {
        type hint;
        file "/etc/bind/db.root";
        };
```

input3.txt

```
logging
        {
        category lame-servers { null; };
        category cname { null; };
        };
zone "."
        {
        type hint;
        file "/etc/bind/db.root";
        };
```

example3.l

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

%%
[a-zA-Z][a-zA-Z0-9]*    printf("WORD ");
[a-zA-Z0-9\/.-]+        printf("FILENAME ");
\"                      printf("QUOTE ");
\{                      printf("OBRACE ");
\}                      printf("EBRACE ");
;                       printf("SEMICOLON ");
\n                      printf("\n");
[ \t]+                  /* ignore whitespace */;
%%
```

input3.txt

```
logging
        {
        category lame-servers { null; };
        category cname { null; };
        };
zone "."
        {
        type hint;
        file "/etc/bind/db.root";
        };
```

```
{atlas:maria:470}  example3  < input3.txt
WORD OBRACE
WORD FILENAME OBRACE WORD SEMICOLON EBRACE SEMICOLON
WORD WORD OBRACE WORD SEMICOLON EBRACE SEMICOLON
EBRACE SEMICOLON

WORD QUOTE FILENAME QUOTE OBRACE
WORD WORD SEMICOLON
WORD QUOTE FILENAME QUOTE SEMICOLON
EBRACE SEMICOLON
```

example3.l

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

%%
[a-zA-Z][a-zA-Z0-9]*    printf("WORD ");
[a-zA-Z0-9\/.-]+        printf("FILENAME ");
\"                      printf("QUOTE ");
\{                      printf("OBRACE ");
\}                      printf("EBRACE ");
;                       printf("SEMICOLON ");
\n                      printf("\n");
[ \t]+                  /* ignore whitespace */;
%%
```

Some Rules of the Rules

- The rules section of the Lex/Flex input contains a series of rules of the form:
  - » pattern action
- Patterns (more next slide):
  - » Un-indented (starts in the first column) and the action starts on the same line
  - » Ends or terminates at first non-escaped white space
- Action:
  - » If action is empty and a pattern match
    - – then input token is discarded (ignored)
  - » If the action is enclosed in braces {} then the action may cross multiple lines

Patterns: *Regular Expressions* in Lex

- Operators:
  - » `\ [ ] ^ - ? . * + | ( ) $ / { } % < >`
  - » Use 'escape' to use an operator as a character, the escape character is "\"
  - » Examples:
    - `\$` = "$"
    - `\\` = "\"
- [ ]: Defines 'character classes' - matches the string once.
  - » `[ab]`
    - a or b
  - » `[a-z]`
    - a or b or c or … z
  - » `[^a-zA-Z]`
    - ^ negates - any character that is NOT a letter - not the circumflex is in a character class[], outside a character class ^ means beginning of line.
- ".":
  - » Matches all characters except newline (\n)
- A? A* A+:

Regular Expressions in Lex

- More Examples:
  - ab?c
    - ac or abc
  - [a-z]+
    - Lowercase strings
  - [a-zA-Z][a-zA-Z0-9]
    - Alphanumeric strings, maria1, maria2

Regular Expressions

- Order of precedence
  - Kleene *, ?, +
  - Concatenation
  - Alternation (|)
  - All operators are left associative
  - Example:
    - a*b|cd*
      - = ((a*)b)|(c(d*))
- [\t\n]
  - if no action defined for this match, lex just ignores that input.

Pattern Matching Primitives

Metacharacter	Matches
.	any character except newline
\n	newline
*	zero or more copies of the preceding expression
+	one or more copies of the preceding expression
?	zero or one copy of the preceding expression
^	beginning of line / complement (outside the [ ] character class).
$	end of line
a|b	a or b
(ab)+	one or more copies of ab (grouping)
[ab]	a or b
a{3}	3 instances of a
"a+b"	literal "a+b"

Lex *predefined* variables

yytext	String containing the lexeme
yyleng	Length of the string
yyin	Input stream pointer FILE *
yyout	Output stream pointer FILE *

Example: Counting number of words in a file and their total size:

```
[a-zA-Z]+   {words++; chars += yyleng;}
```


Printing out a Summary

exampleX.l

```
%{
#include <stdio.h>
int num_words = 0, num_chars = 0;
%}

%%
[a-zA-Z]+          {num_words++; num_chars += yyleng;}
%%

int main()
{
yylex();
printf("%d words %d chars\n", num_words, num_chars );
};
```

```
{atlas:maria:422} flex -l -t exampleX.l > exampleX.c
{atlas:maria:423} gcc exampleX.c -o exampleX -lfl
{atlas:maria:424} exampleX
helkje helkhweoj lkwjerflkwj maria is great

Skljdf

^D
7 words 44 chars
```


Lex *Library* Routines

yylex()	Default main() calls yylex()
yymore()	Returns next token
yyless(n)	Retain the first n characters in yytext
yywarp()	Called at end of file EOF, returns 1 is default

More Lex

- A regular expression in Lex finishes with a space, tab or newline
- The choices of Lex:
  - » Lex always matches the longest (number of characters) lexeme possible
  - » If two or more "lexemes" are of the same length, then Lex choose the lexem corresponding to the first regular expression.
  - » Example: matchlongest.l

Summary LEX

- Lex is a scanner generator
  - » generates C code (or C++ code) to scan inputs for lexemes (string of the language) and convert them to tokens
- Lex defines the tokens by using an input file (specification file) that specifies the lexemes as regular expressions
- Regular expressions in the specification file terminates with
  - » space, newline or tab
- Rules:
  - » Longest possible match first,
  - » Same length, picks the expression that is specified first

- Last time: Big picture and tutorial on LEX
- Today: Tutorial on YACC
- Thursday – Parse Trees / Ambiguity, and Conclusion on Parsing.

What is *YACC*: Yet Another Compiler-Compiler?

- A tool that automatically generates a parser according to given specifications.
  - » YACC is higher-level that lex:
    - – "it deals with sentence instead of words."
  - » YACC specification is given in a specification file, typically post fixed with a y, so a .y file.
  - » Parses input streams coming from lex that now contains "tokens" (these tokens may have values associated with them, more on this shortly).
- YACC uses "grammar rules" that allows it to analyze if the stream of tokens from lex is legal.
  - » We will use a free version of YACC called bison (http://www.gnu.org/software/bison/manual/pdf/bison.pdf)

YACC File Format (looks a lot like a lex configuration file)

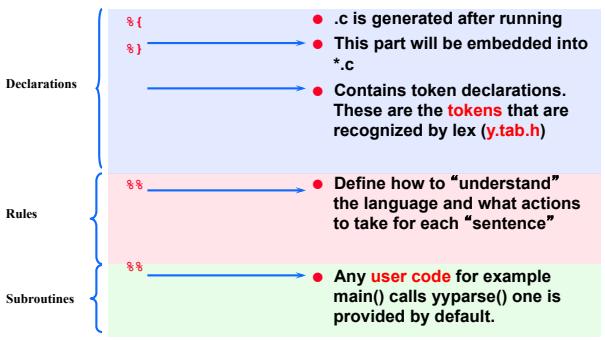
```
%{
        . . . C declarations . . .
%}
        . . . yacc declarations . . .
%%

        . . . rules . . .

%%
        . . . Subroutines . . .
```


YACC Syntax

- `Yacc` input file consist of up to three sections



The YACC Specification File (.y)

- **Definitions**
  - » Declarations of tokens
  - » Type of values used on parser stack (if different from default type (INTEGER)).
    - – These definitions are then defined in a header file, y.tab.h that lex includes (processed separately –d)
- **Rules**
  - » Lists grammar rules with corresponding routines
- **User Code**

The Rule Section: BNF like

```
%%
production :  symbol1 symbol2 …      { action }
          |  symbol3 symbol4 … { action }
          |  …
          ;

Production :  symbol1 symbol2        { action }
          ;
%%
```


Example (*Preview*)

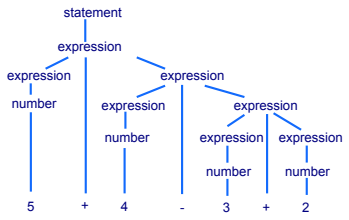
- **Give me a rule in a grammar that includes**
  - » expressions that
  - » add or subtract two NUMBERS.
- **Assume 'NUMBERS' is already defined.**
- **USES stacks to keep track of 'values' and 'current symbols' (current parsing state) in parsing a grammar.**
  - » Parse stack (current parsing state)
  - » Value stack (type and value)

Diving In: Example Rule

```
%%
statement : expression                  { printf ("= %g\n", $1); }
expression : expression '+' expression          { $$ = $1 + $3; }
     | expression '-' expression                { $$ = $1 - $3; }
     | number                                        { $$ = $1; }
     ;
%%
```

According these two productions,

5 + 4 – 3 + 2 is parsed into:



BUT First: Lets do a *simpler* example to see how YACC works with LEX

- **Create a simple language to control a thermostat (red)**
- **Need:**
  - » 2 states that are set to on or off
  - » Target
  - » Temperature
  - » Number
- **First create a scanner (lexer) to define the tokens of the language:**
  - » `example4.l` next…

Session:
heat on
Heater on!
heat off
Heater off!
target temperature 22
New temperature set!

Tokens:
heat
on
off
target
temperature
number

Controlling a Thermostat: `Lex` input

Tokens:
heat
on
off
target
temperature
number

heat on
Heater on!
heat off
Heater off!
target temperature 22
New temperature set!

example4.l

```
%{
#include <stdio.h>
#include "example4.tab.h"  /* generated from yacc */
extern YYSTYPE yylval;     /* need for lex/yacc */
%}
%%
[0-9]+            return NUMBER;
heat              return TOKHEAT;
on|off            return STATE;
target            return TOKTARGET;
temperature       return TOKTEMPERATURE;
\n                /* ignore end of line */;
[ \t]+            /* ignore whitespace */;
%%
```

- Tokens are fed (`returned`) to **yacc**
- **y.tab.h** defines the tokens : generated from yacc.

Controlling a Thermostat: YACC input

```
heat on
      Heater on!
heat off
      Heater off!
target temperature 22
      New temperature set!
```

Tokens:
heat
on
off
target
temperature
number

example4.y

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

heat_switch:
        TOKHEAT STATE
        {
        printf("\tHeat turned on or off\n");
        } ;

target_set:
        TOKTARGET TOKTEMPERATURE NUMBER
        {
        printf("\tTemperature set\n");
        } ;
```


Details: More later

- If components in a rule is ***empty***, it means that the result can match the empty string (λ). For example, how would you define a comma-separated sequence of zero or more letters?
  - » A, B, C, D, …. E

```
startlist:    /* empty */
        | letterlist
        ;
letterlist: letter
        | letterlist ',' letter
        ;
```

- Left recursion better on Bison because it can then use a bounded stack (we look at this at more detail next week or maybe this Thursday)

Controlling a Thermostat: The rest of the YACC specifications

```
heat on
      Heater on!
heat off
      Heater off!
target temperature 22
      New temperature set!
```

Tokens:
heat
on
off
target
temperature
number

example4.y

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

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

int yywrap() /* called at EOF can open another
                then return 0 */
{
        return 1; /*  YES! I am really done */
}

main()
{
       yyparse(); /* get things started */
}

%}

%token NUMBER TOKHEAT STATE TOKTARGET TOKTEMPERATURE
%%
```


- `yyerror()` called when yacc finds an error
- `yywrap()` used if reading from multiple files, `yywrap()` called at `EOF`, so you can open up another file and return 0. OR you return 1 to indicate nope you are "really" done.

example4.l example4.y

Flex, YACC and Run

```
{atlas:maria:194} flex -l -t example4.l > example4L.c
{atlas:maria:195} bison -d example4.y; rm example4.tab.c // declaration
{atlas:maria:196} bison -v example4.y -o example4Y.c
{atlas:maria:197} gcc example4L.c example4Y.c -o example4
```

Notice: No link command to gcc, i.e. no -lfl

Do you know why?

```
{atlas:maria:202} example4
heat on
        Heat turned on or off
heat off
        Heat turned on or off
target temperature 10
        Temperature set
target temperature 10
error: parse error
```



```
{atlas:maria:202} example4
heat on
        Heat turned on or off
heat off
        Heat turned on or off
target temperature 10
        Temperature set
target temperature 10
```

Wanted this …

```
heat on
      Heater on!
heat off
      Heater off!
target temperature 22
      New temperature set!
```

```
target temperature 22
      Temperature set to 22
```

Need access to the value of parameters

Add parameters to YACC

- When `lex` matches a target:
  - » The "matched string" is in "`yytext`"
  - » How is it communicated to YACC?
    - – To communicate a **value** to yacc you can use the variable "`yylval`" (so far we have not seen how to do this, but we will now).

Controlling a Thermostat: `Lex` input

```
heat on
    Heater on!
heat off
    Heater off!
target temperature 22
    Temperature set to 22
```

Tokens:
heat
on
off
target
temperature
number

example5.l

```
%{
#include <stdio.h>
#include "example5.tab.h"     /* generated from yacc */
extern YYSTYPE yylval;
%}
%%
[0-9]+          yylval=atoi(yytext); return NUMBER;
heat            return TOKHEAT;
on|off          yylval=!strcmp(yytext,"on");
                 return STATE;
target          return TOKTARGET;
temperature     return TOKTEMPERATURE;
\n              /* ignore end of line */;
[ \t]+          /* ignore whitespace */;
%%
```

- Heater on! Heater off!
- Tokens are fed (`returned`) to **yacc**
- **y.tab.h** defines the tokens


```
heat on
    Heater on!
heat off
    Heater off!
target temperature 22
    Temperature set to 22
```

example5.l

```
%{
#include <stdio.h>
#include "example4.tab.h"/* generated from yacc */
extern YYSTYPE yylval;
%}
%%
[0-9]+          yylval=atoi(yytext); return NUMBER;
heat            return TOKHEAT;
on|off          yylval=!strcmp(yytext,"on");
                 return STATE;
target          return TOKTARGET;
temperature     return TOKTEMPERATURE;
\n              /* ignore end of line */;
[ \t]+          /* ignore whitespace */;
%%
```

example5.y

```
heat_switch:
        TOKHEAT STATE
        {
         if($2)
                        printf("\tHeat turned on\n");
                else
                        printf("\tHeat turned off\n");
        }
        ;
```

```
heat on
    Heater on!
heat off
    Heater off!
target temperature 22
    Temperature set to 22
```

example5.l

```
%{
#include <stdio.h>
#include "example4.tab.h"  /* generated from yacc */
extern YYSTYPE yylval;
%}
%%
[0-9]+          yylval=atoi(yytext); return NUMBER;
heat            return TOKHEAT;
on|off          yylval=!strcmp(yytext,"on");
                return STATE;
target          return TOKTARGET;
temperature     return TOKTEMPERATURE;
\n              /* ignore end of line */;
[ \t]+          /* ignore whitespace */;
%%
```

example5.y

```
target_set:
    TOKTARGET TOKTEMPERATURE NUMBER
        {
            printf("\tTemperature set to %d\n", $3)

        }
        ;
```

example5.y

```
heat_switch:
        TOKHEAT STATE
        {
         if($2)
                        printf("\tHeat turned on\n");
                else
                        printf("\tHeat turned off\n");
        }
        ;
```

```
target_set:
        TOKTARGET TOKTEMPERATURE NUMBER
        {
                printf("\tTemperature set to %d\n",$3);
        }
        ;
```

```
{atlas:maria:248} flex -l -t example5.l > example5L.c
{atlas:maria:249} bison -v example5.y -o example5Y.c
{atlas:maria:250} bison -d example5.y; rm example5.tab.c
rm: remove example5.tab.c (yes/no)? yes
{atlas:maria:251} gcc example5L.c example5Y.c -o example5
{atlas:maria:253} example5
heat on
        Heat turned on
target temperature 10
        Temperature set to 10
```

example6.l

Continue lame-servers

- **Write YACC grammar**
- **Need to translate the lexer so that it returns values to YACC (e.g., file names and zone names)**

```
logging
        {
        category lame-servers { null; };
        category cname { null; };
        };
zone "."
        {
        type hint;
        file "/etc/bind/db.root";
        };
```

example6.l

```
logging
        {
        category lame-servers { null; };
        category cname { null; };
        };
zone "."
        {
        type hint;
        file "/etc/bind/db.root";
        };
```

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

%%
[a-zA-Z][a-zA-Z0-9]*    printf("WORD ");
[a-zA-Z0-9\/.-]+        printf("FILENAME ");
\"                      printf("QUOTE ");
\{                      printf("OBRACE ");
\}                      printf("EBRACE ");
;                       printf("SEMICOLON ");
\n                      printf("\n");
[ \t]+                  /* ignore whitespace */;
%%
```

```
%{
#include <stdio.h>
#include "example5.tab.h"
extern YYSTYPE yylval;
%}

%%

zone                    return ZONETOK;
file                    return FILETOK;
[a-zA-Z][a-zA-Z0-9]*    yylval = strdup(yytext); return WORD;
[a-zA-Z0-9\/.-]+        yylval = strdup(yytext); return FILENAME;
\"                      return QUOTE;
\{                      return OBRACE;
\}                      return EBRACE;
;                       return SEMICOLON;
\n                      /* ignore EOL */;
[ \t]+                  /* ignore whitespace */;
%%
```

example6.y

```
logging
        {
        category lame-servers { null; };
        category cname { null; };
        };
zone "."
        {
        type hint;
        file "/etc/bind/db.root";
        };
```

```
#define YYSTYPE char *
  … other routines …
%%
commands:
   | commands command SEMICOLON
   ;
command:
   zone_set
   ;
zone_set:
   ZONETOK quotedname zonecontent
   {
        printf("Complete zone for '%s' found\n", $2 );
   }
   ;
```

example6.y

```
logging
        {
        category lame-servers { null; };
        category cname { null; };
        };
zone "."
        {
        type hint;
        file "/etc/bind/db.root";
        };
```

```
zonecontent:
   OBRACE zonestatements EBRACE

quotedname:
   QUOTE FILENAME QUOTE
        {
                $$=$2;
        }
```

- quotedname's value is without the quotes,
  - » $$=$2

example6.y

```
logging
        {
        category lame-servers { null; };
        category cname { null; };
        };
zone "."
        {
        type hint;
        file "/etc/bind/db.root";
        };
```

- Generic statements to catch statements within the zone block

```
zonestatements:
   |
   zonestatements zonestatement SEMICOLON
   ;
zonestatement:
   statements
   |
   FILETOK quotedname
   {
        printf("A zonefile name '%s' was encountered\n", $2 );
   }
```

- Generic statements to catch block statements too

example6.y

```
logging
        {
        category lame-servers { null; };
        category cname { null; };
        };
zone "."
        {
        type hint;
        file "/etc/bind/db.root";
        };
```

```
block:
   OBRACE zonestatements EBRACE SEMICOLON
   ;
statements:
   | statements statement
   ;
statement: WORD | block | quotedname
```

example6.l example6.y

Flex, Yacc and Run

```
{atlas:maria:194} flex -l -t example6.l > example6L.c
{atlas:maria:195} bison -d example6.y; rm example6.tab.c
{atlas:maria:196} bison -v example6.y -o example6Y.c
{atlas:maria:197} gcc example6L.c example6Y.c -o example6
```

```
zone "."
        {
        type hint;
        file "/etc/bind/db.root";
        };
```

```
{atlas:maria:202} example6 < input6.txt
A zonefile name '/etc/bind/db.root' was encountered
Complete zone for '.' found
```

Summary

- YACC file you write your own main() which calls yyparse()
  - » yyparse() is created by YACC and ends up in y.tab.c
- yyparse() reads a stream of token/value pairs from yylex()
  - » Code yylex() yourself or have lex do it for you
- yylex() returns an integer value representing a "token type" you can optionally define a value for the token in yylval (default int)
  - » tokens have numeric id's starting from 256