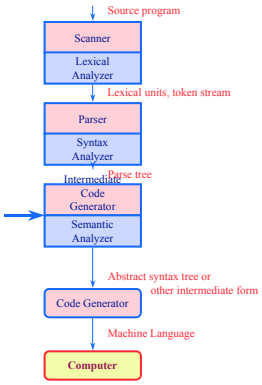
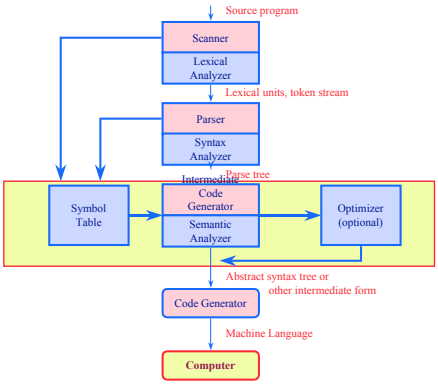
CSCI: 4500/6500 Programming Languages

Lex & Yacc

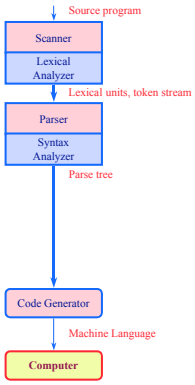
Big Picture: Compilation Process



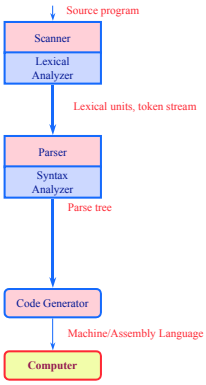
Big Picture: Compilation Process



Big Picture: Compilation Process

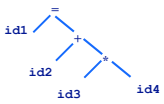


Big Picture: Compilation Process



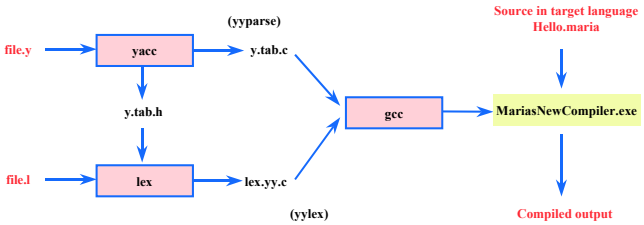

```
a = b + c * d

id1 = id2 + id3 * id4
```




```
load    id3
mul     id4
add     id2
store   id1
```

Overview



- **Pattern matching rules for tokens in file.l**
  - 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 yacc 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
- The each do a "single function"
- This is what they do:
  - » "Get a token from the input stream" and
  - » "Parse a sequence of tokens to see if it matches a grammar"
- Yacc generated parser
  - » 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 (tokenizer). It automatically "generates" a lexer or scanner given a lex specification file as input (.l file)
- Breaks up the input stream into tokens
  - » 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
  - » contains a function called yylex() that obtains the next valid token in the input stream.
  - » this source file can then be compiled by the C compiler to machine or assembly code.

Lex 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 …
```

```
%{
#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 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.

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

```
{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
```

```
%{
#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
```

```
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 */;
%%
```

```
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
```

```
%{
#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 */;
%%
```

Regular Expressions in Lex

- **Operators:**
  - \ [ ] ^ - ? . * + | ( ) $ / { } % < >
  - Use escape to use an operator as a character, the escape character is "\"
  - Examples:
    - \$ = "$"
    - \\ = "\"
- **[ ]: Defines character classes**
  - [ab] a or b
  - [a-z] a or b or c or … z
  - [^a-zA-Z] any character that is NOT a letter
- ".":
  - Matches all characters except newline (\n)
- A? A* A+:
  - 0 or one instance of A, 0 or more, 1 or more instances of A

Regular Expressions in Lex

- **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] ;
  - no action defined for this matched, so 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
$	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 *

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

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 chooses the lexeme corresponding to the first regular expression.

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

What is Yacc?

- A tool that automatically generates a parser according to given specifications.
  - » Yacc is higher-level that lex, e.g. deals with sentence instead of words.
  - » Yacc specification given in a specification file, typically post fixed with a y, so a .y file.
- Parses input streams coming from lex that now consisting of "tokens" (these tokens may have values associated with them, more on this shortly).
- Yacc uses "grammar rules" that allow it to analyze if the stream of tokens from lex is legal.
  - » Yacc creates a syntax tree

Yacc File Format

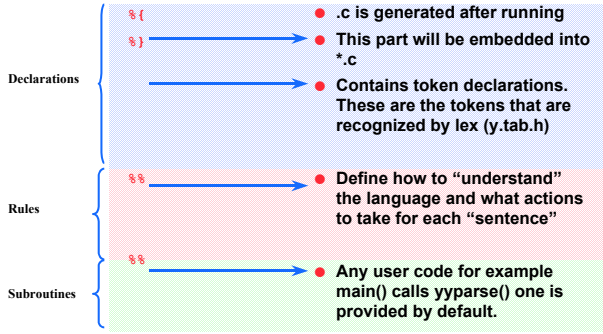
```
%{
        . . . C declarations . . .
%}
        . . . yacc declarations . . .
%%

        . . . rules . . .

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

Yacc Syntax

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



The Yacc Specification File (.y)

- **Definitions**
  - » Declarations of tokens
    - – `%token name`
  - » 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.
- **Rules**
  - » Lists grammar rules with corresponding routines
- **User Code**

The Rule Section

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

production:  symbol1 symbol2         { action }
%%
```

Example

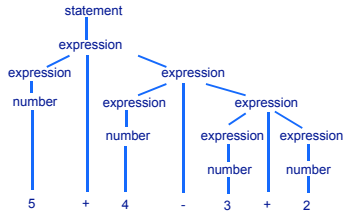
- Give me a rule in a grammar that include expressions that add or subract 2 NUMBERS.
- Assume NUMBERS are already defined.

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:



Lets do a Simpler Example

- Create a simple language to control a thermostat
- Need:
  - » 2 states that are set to on or off
  - » Target
  - » Temperature
  - » Number
- Look at a lexer code, `example4.l` next…

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

Yacc Rule Section

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");
        } ;
```

- **If components in a rule is *empty*, it means that the result can match the empty string. 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 will see why shortly)**

Controlling a Thermostat: The rest of 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;
}

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
{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 temparature 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
```

Want this …

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


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

Add parameters to yacc

- When `lex` matches a target:
  - » The "matched string" is in "`yytext`" and
  - » 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 "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 */;
%%
```

- 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

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

        }
        ;
```

```
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");
        }
        ;
```

```
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

- Generic statements to catch statements within the zone block

example6.y

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

```
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**

Yacc

- **Uses bottom up shift/reduce parsing**
  - » **Gets a token**
  - » **Pushes token onto stack**
- **Next time…**

Yacc Theory

- **Recall grammars for yacc are a variant of BNF**
  - » **can be used to express context free languages**
    X → p
  - » **X is non terminal, p is a string of non-terminals and/or terminals)**
  - » **Context free because X can be replaced by p regardless of the context X is in.**
- **Example: context free grammar of different number of a and b's**

Yacc Theory

- **Example: context free grammar of different number of a and b's (M for mixed, A for more a's)**
  - » **S → A | B**
  - » **A → MaA | MaM**
  - » **B → MbB | MbM**
  - » **M → aMbM | bMaM | ε**
- **T generates all the string of the same number of a and B's, A generates more a's and b's B more b's.**

- **Example: Grammar that multiply and adds numbers:**

```
E → E + E      (rule 1)
E → E * E      (rule 2)
E → id         (rule 3)
```

- **id is returned by lex (terminals) and only appears on right hand side.**

x + y * z is generated by:

```
E →  E  *  E        (rule 2)
  →  E  *  z        (rule 3)
  →  E  +  E * z    (rule 3)
  →  E  +  y * z    (rule 3)
  →  x  +  y * z    (rule 3)
```

```
E → E + E      (rule 1)
E → E * E      (rule 2)
E → id         (rule 3)
```

- To parse the expression we go in reverse, reduce an expression to a single non terminal, We do this by shift-reduce parsing and use a stack for storing terms

```
1. . x + y * z          shift (stack on left of dot)
2. x . + y * z          reduce (rule 3)
3. E . + y * z          shift
4. E + . y * z          shift
5. E +  y. * z          reduce (rule 3)
6. E + E. * z           shift
7. E + E * . z          shift
8. E + E * z .          reduce (rule 3)       emit multiply
9. E + E * E .          reduce (rule 2)       emit add
10.E + E .              reduce (rule 1)
11.E .                  accept
```

» When we have a match on the stack to one of right hand side of productions
  » replace the match with the left hand side of token


```
E → E + E      (rule 1)
E → E * E      (rule 2)
E → id         (rule 3)
```

- To parse the expression we go in reverse, reduce an expression to a single non terminal, We do this by shift-reduce parsing and use a stack for storing terms

```
1. . x + y * z          shift (stack on left of dot)
2. x . + y * z          reduce (rule 3)
3. E . + y * z          shift
4. E + . y * z          shift
5. E +  y. * z          reduce (rule 3)
6. E + E. * z           shift
7. E + E * . z          shift
8. E + E * z .          reduce (rule 3)       emit multiply
9. E + E * E .          reduce (rule 2)       emit add
10.E + E .              reduce (rule 1)
11.E .                  accept
```


- shift reduce conflict at step 6
  ambiguous grammar

Ambiguity

- Ambiguity can be resolved by
  - » rewriting the grammar or
  - » indicate which operator has precedence (yacc)
- Reduce Reduce conflict

```
E → T
E → id
T → id
```

  - » Either reduces to E or to T
    - – yacc will always use the first rule listed.
- For shift-reduce conflicts yacc will always shift