Languages: lexical analysis

Glenn Bruns
CSUMB

A more realistic grammar

```
expr ::= NUM | ID | expr + expr
```

Derive `ID + NUM`:

```
expr →
expr + expr →
ID + expr →
ID + NUM
```

Derive `x + 22`:

?

Learning outcomes

After this lecture, you should be able to:

- use a lexical analyzer

Defining terminal symbols

```
expr ::= NUM | ID | expr + expr

NUM is one or more digits
ID is a letter followed by zero or more digits or letters
```

Derive x + 22:

expr →
expr + expr →
ID + expr →
ID + NUM

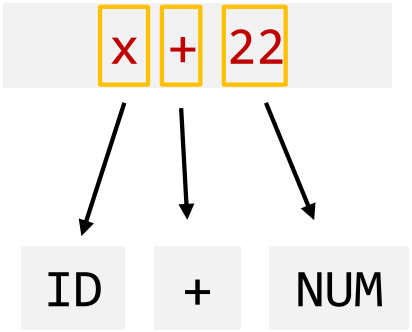
Lexical analysis

```
expr ::= NUM | ID | expr + expr

NUM: [0-9][0-9]*              // one or more digits
ID:  [a-zA-Z][a-zA-Z0-9]*     // an identifier
```

1. Break the input into "tokens"

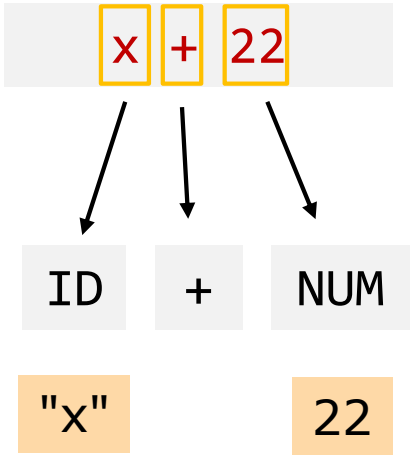
2. Parse the tokens




```
ID + NUM
expr →
expr + expr →
ID + expr →
ID + NUM
```

white space is thrown away

Token values



When we "tokenize", we store the values of some tokens for later use.

API for a lexical analyzer

```
Lexer:
   // return ID, NUM, a single character, or NONE
   int lexan()

   // return value of last NUM token
   int num_val():

   // return value of last ID token
   char *id_val():
```

Call lexan() to get the next token (from standard input).

If it's a NUM token you can call get_val() to get the value.

The kinds of tokens supported will depend on the BNF.

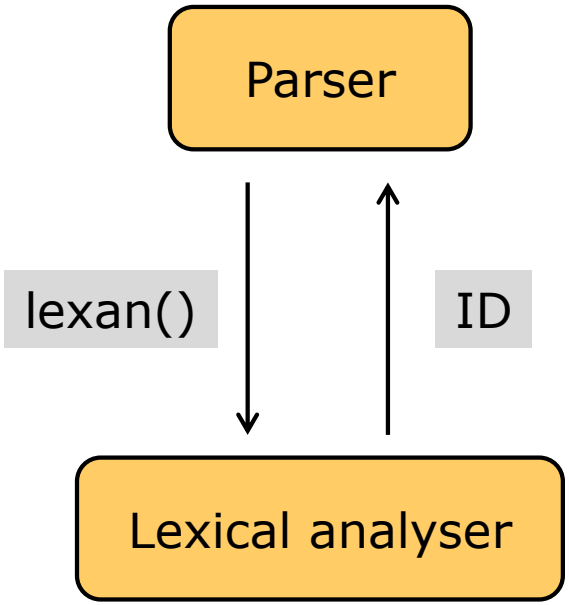
Pseudocode for lexan()

```
int val;
char buf[BUF_SIZE];

int lexan() {
   while (true) {
      get t from standard input
      if t is space or tab
         skip
      else if t is a digit {
         val = integer formed from t and following digits
         return NUM
      else if t is a letter {
         buf = string formed from t and following letters
               and digits
         return ID
      else if end of file
         return DONE
      else
         return t
```

Using the lexical analyzer

Parser calls lexical analyzer ("lexer") to get a token



Example

```
expr ::= NUM + NUM
```

Pseudocode to parse an expr:

```
lookahead = lexan()           # get a token
if (lookahead == NUM) {
   lookahead = lexan() {
      if (lookahead == '+') {
         lookahead = lexan()
         if (lookahead == NUM) {
            print "success"
         }
      }
   }
}
print "syntax error"
```

Actual code

```
#include "lexer.h"
void main() {
   lexer_init();

   // get lookahead token
   int lookahead;
   lookahead = lexan();
   // parse NUM + NUM
   if (lookahead  == NUM) {
      lookahead = lexan();
      if (lookahead == '+') {
         lookahead = lexan();
         if (lookahead == NUM) {
            printf("parsed NUM + NUM\n");
            exit(EXIT_SUCCESS);
         }
      }
   }
   printf("syntax error\n");
   exit(EXIT_FAILURE);
}
```

The lookahead token is the next token to be processed.

Running the parser:

```
$ ./parser
1 + 2
parsed NUM + NUM
$ ./parser
1+2
parsed NUM + NUM
$ ./parser
1 2 +
syntax error
$
```

Other functions in the API

```
#include "lexer.h"
void main() {
   char *id;
   int lookahead, i;
   lexer_init();

   // get lookahead token
   lookahead = lexan();

   // parse ID NUM
   if (lookahead  == ID) {
      id = lexer_id_val();
      lookahead = lexan();
      if (lookahead == NUM) {
         i = lexer_num_val();
         printf("%s %d\n", id, i);
         exit(EXIT_SUCCESS);
      }
   }
   printf("syntax error\n");
   exit(EXIT_FAILURE);
}
```

`int lexer_num_val()`

returns the value of the current NUM token

`char *lexer_id_val()`

returns the value of the current ID token

Running the parser:

```
$ ./parser
x 10
x 10
$ ./parser
x10
syntax error
$ ./parser
x10 10
x10 10
$
```

Lexer API we'll use in class

`int lexer_init()`	initialize the lexical analyzer
`int lexan()`	returns type of the next token
`int lexer_num_val()`	returns the value of the current NUM token
`char *lexer_id_val()`	returns the value of the current ID token
`int lexer_lineno()`	returns the current line number

The handy helper function match():

```
void match(int token) {
    if (lookahead == token)
        lookahead = lexan();
    else error("syntax error");
}
```

Use match() when you know what the next token must be.

Summary

- lexical analysis means breaking up the input into "tokens"
- a parser gets tokens from a lexical analyzer

What's in a compiler?

What are some of the things that compilers do?

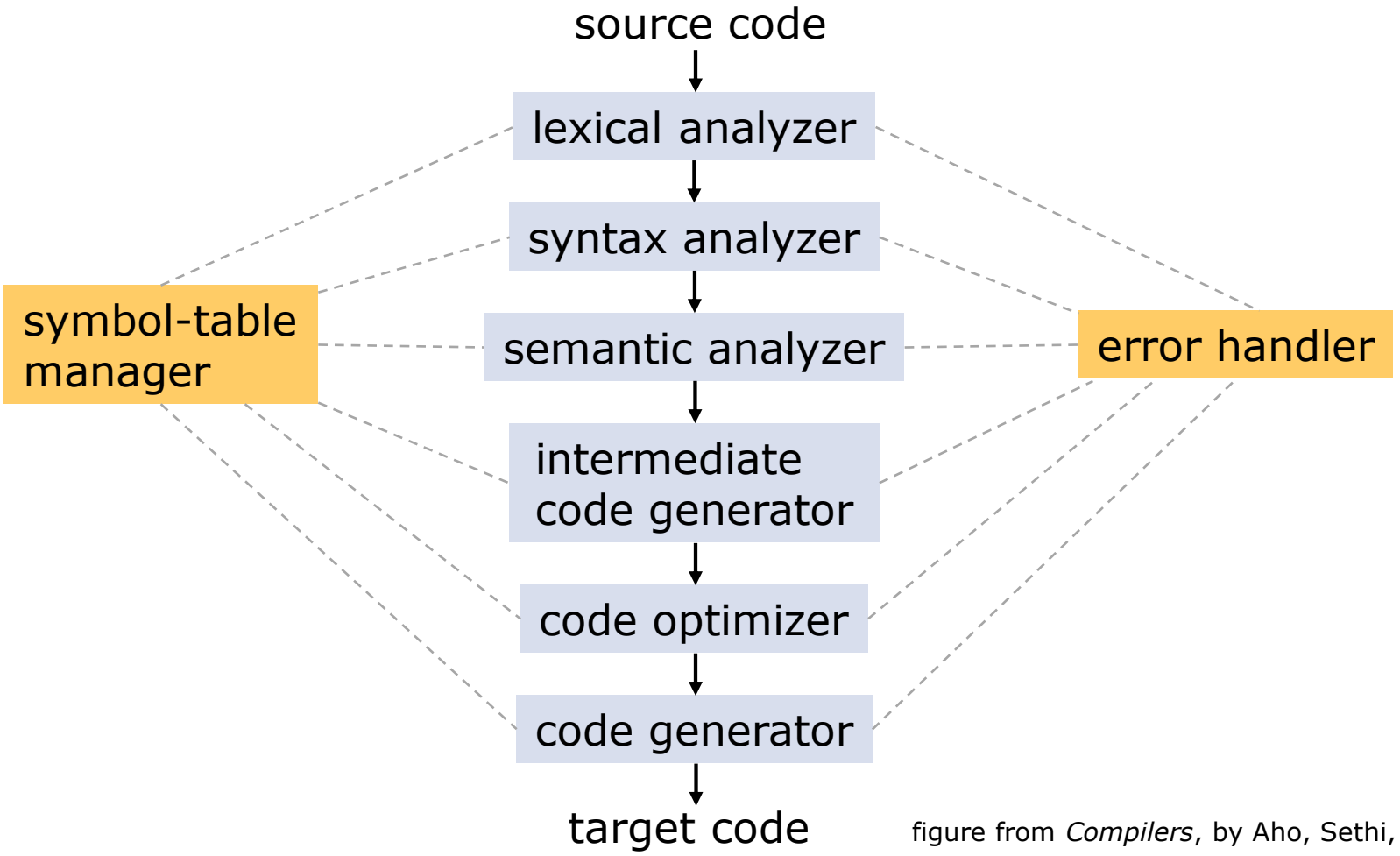


figure from *Compilers*, by Aho, Sethi, Ullman

Writing a lexical analyzer

1. Write it by hand
2. Use a lexical analyzer generator:
   - specify the tokens using regular expressions
   - run generator to get lexical analyzer code
   - examples: Lex, Flex, Quex, …

Part of a lex spec

```
L                   [A-Za-z]
D                   [0-9]
{L}({L}|{D})*       {count();
                      yylval.sym = str_create(yytext);
                      return(VAR);  }
{D}*                {count();
                      yylval.sym = str_create(yytext);
                      return(NUM);  }
```