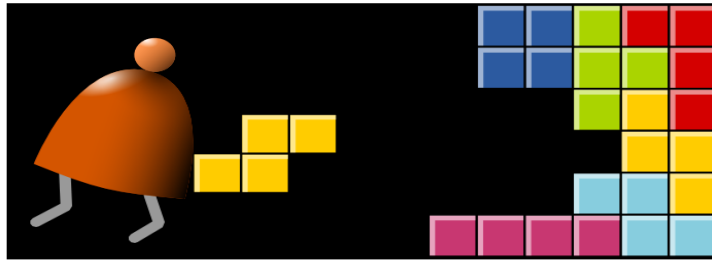


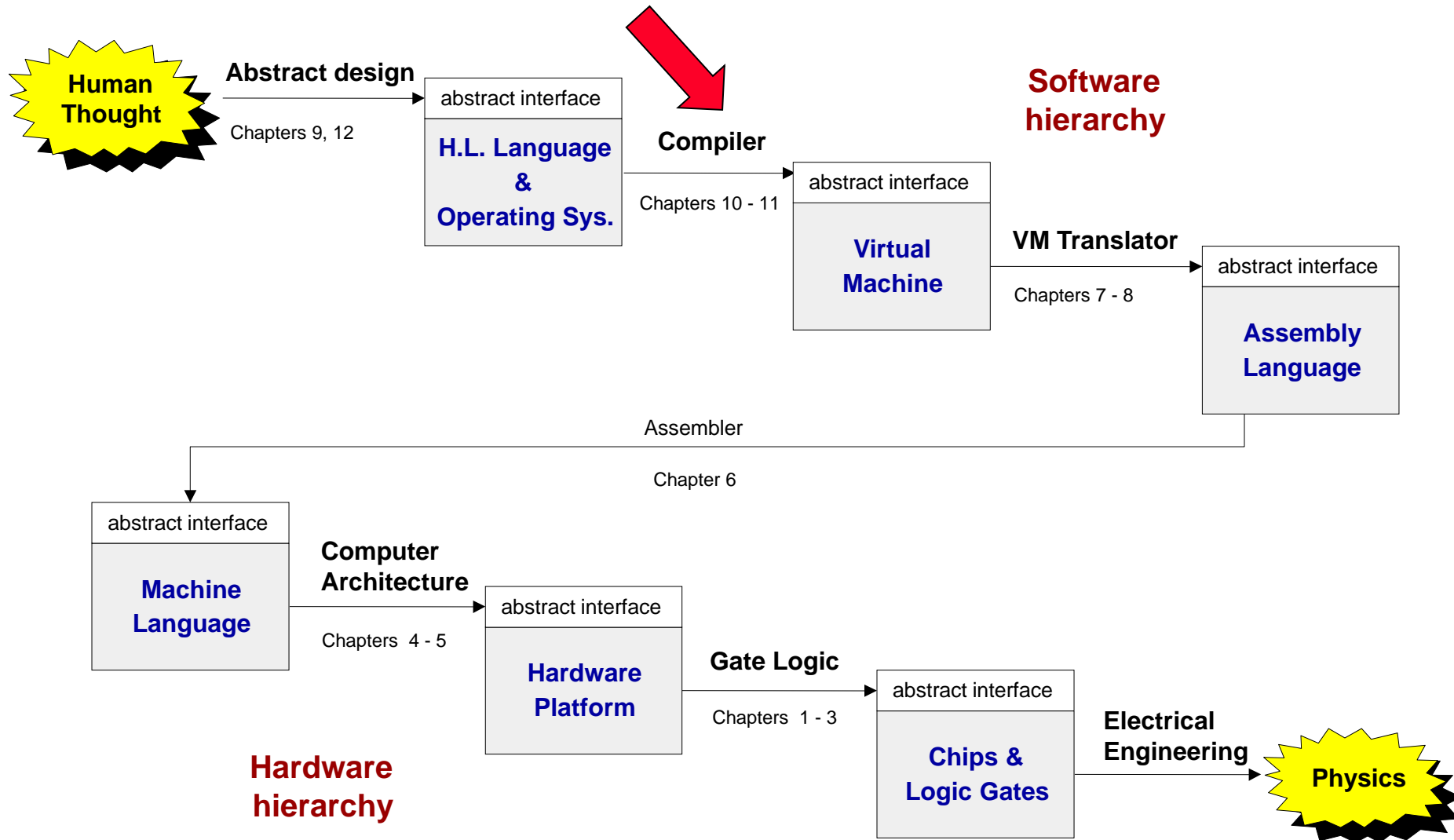
# Compiler I: Syntax Analysis



*Building a Modern Computer From First Principles*

[www.nand2tetris.org](http://www.nand2tetris.org)

# Course map



## Motivation: Why study about compilers?

---

The first compiler is FORTRAN compiler developed by an IBM team led by John Backus (Turing Award, 1977) in 1957. It took 18 man-month.

### Because Compilers ...

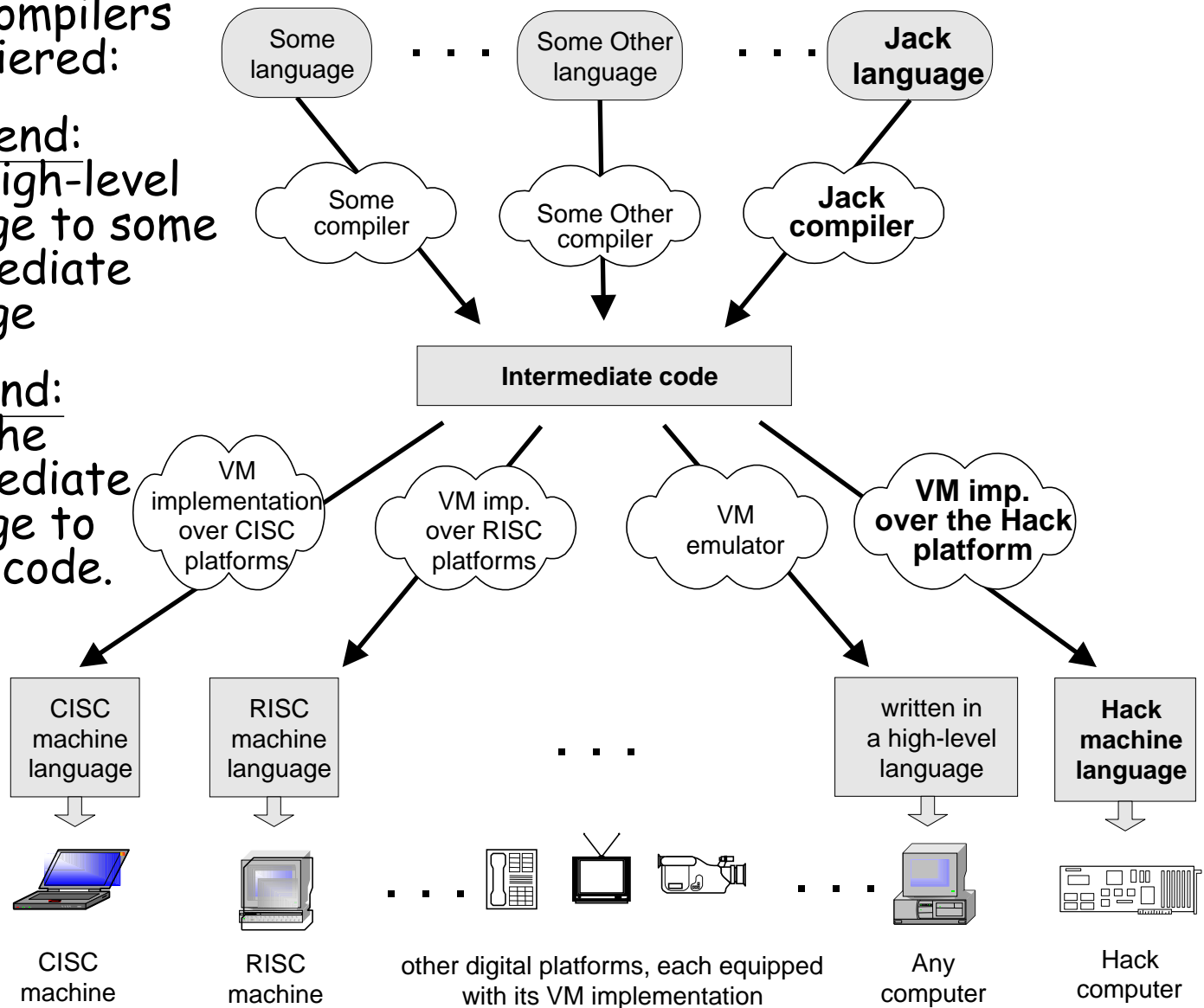
- Are an essential part of applied computer science
- Are very relevant to computational linguistics
- Are implemented using classical programming techniques
- Employ important software engineering principles
- Train you in developing software for transforming one structure to another (programs, files, transactions, ...)
- Train you to think in terms of "description languages".
- Parsing files of some complex syntax is very common in many applications.

# The big picture

Modern compilers are two-tiered:

- Front-end: from high-level language to some intermediate language

- Back-end: from the intermediate language to binary code.



Compiler lectures

(Projects 10,11)

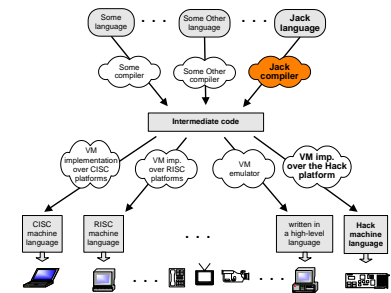
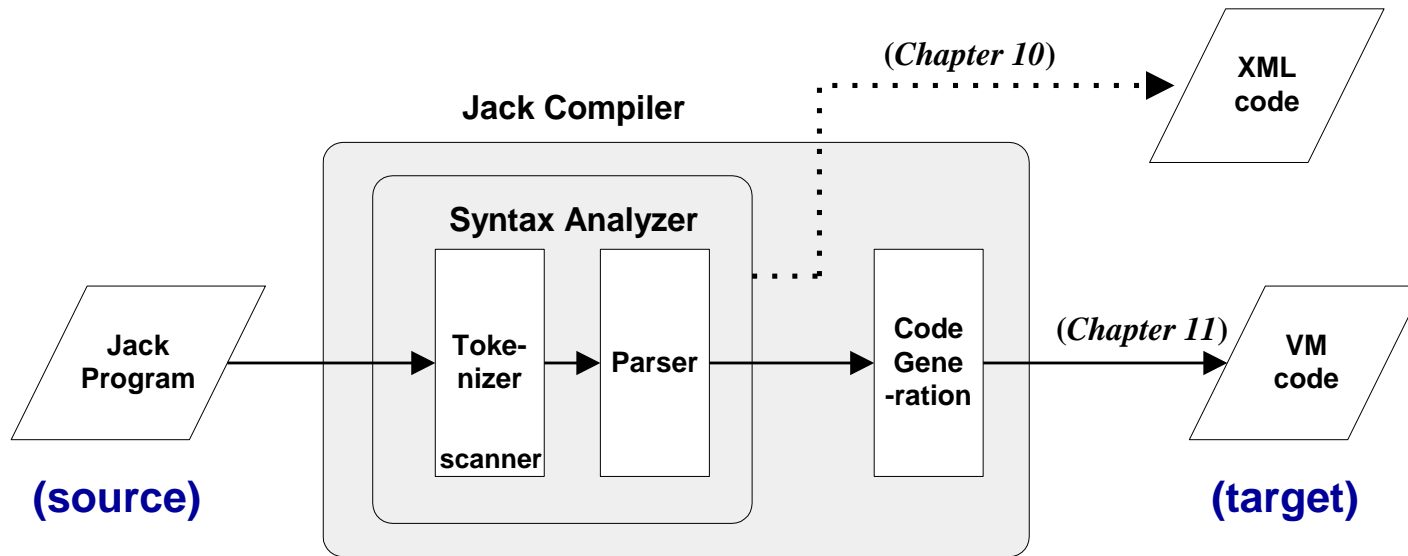
VM lectures

(Projects 7-8)

HW lectures

(Projects 1-6)

# Compiler architecture (front end)



- **Syntax analysis**: understanding the structure of the source code
  - **Tokenizing**: creating a stream of “atoms”
  - **Parsing**: matching the atom stream with the language grammar
- XML output = one way to demonstrate that the syntax analyzer works
- **Code generation**: reconstructing the **semantics** using the syntax of the target code.

# Tokenizing / Lexical analysis / scanning

## *C code*

```
while (count <= 100) { /** some loop */  
    count++;  
    // Body of while continues  
    ...
```

tokenizing

## *Tokens*

```
while  
(  
count  
<=  
100  
)  
{  
count  
++  
;  
...
```

- Remove white space
- Construct a token list (language atoms)
- Things to worry about:
  - Language specific rules: e.g. how to treat "++"
  - Language-specific classifications:  
keyword, symbol, identifier, integerConstant,  
stringConstant,...
- While we are at it, we can have the tokenizer record not only the token, but also its lexical classification (as defined by the source language grammar).

# C function to split a string into tokens

■ `char* strtok(char* str, const char* delimiters);`

- `str`: string to be broken into tokens
- `delimiters`: string containing the delimiter characters

```
1 /* strtok example */
2 #include <stdio.h>
3 #include <string.h>
4
5 int main ()
6 {
7     char str[] = "- This, a sample string.";
8     char * pch;
9     printf ("Splitting string \"%s\" into tokens:\n",str);
10    pch = strtok (str, " ,.-");
11    while (pch != NULL)
12    {
13        printf ("%s\n",pch);
14        pch = strtok (NULL, " ,.-");
15    }
16    return 0;
17 }
```

Output:

```
Splitting string "- This, a sample string." into tokens:
This
a
sample
string
```

# Jack Tokenizer

```
if (x < 153) {let city = "Paris";}
```

Source code

## Tokenizer's output

```
<tokens>
  <keyword> if </keyword>
  <symbol> ( </symbol>
  <identifier> x </identifier>
  <symbol> < </symbol>
  <integerConstant> 153 </integerConstant>
  <symbol> ) </symbol>
  <symbol> { </symbol>
  <keyword> let </keyword>
  <identifier> city </identifier>
  <symbol> = </symbol>
  <stringConstant> Paris </stringConstant>
  <symbol> ; </symbol>
  <symbol> } </symbol>
</tokens>
```



Tokenizer



# Parsing

---

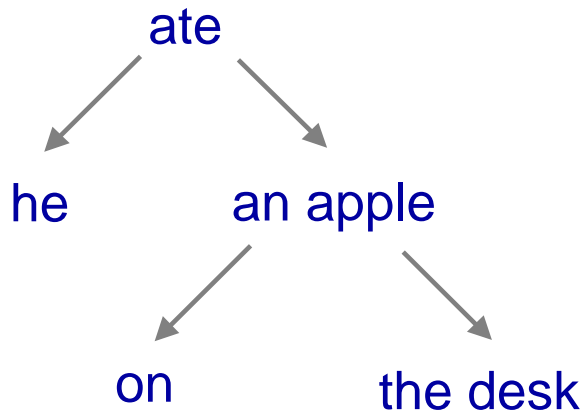
- The tokenizer discussed thus far is part of a larger program called *parser*
- Each language is characterized by a *grammar*.  
The parser is implemented to recognize this grammar in given texts
- The parsing process:
  - A text is given and tokenized
  - The parser determines whether or not the text can be generated from the grammar
  - In the process, the parser performs a complete structural analysis of the text
- The text can be in an expression in a :
  - Natural language (English, ...)
  - Programming language (Jack, ...).

# Parsing examples

## English

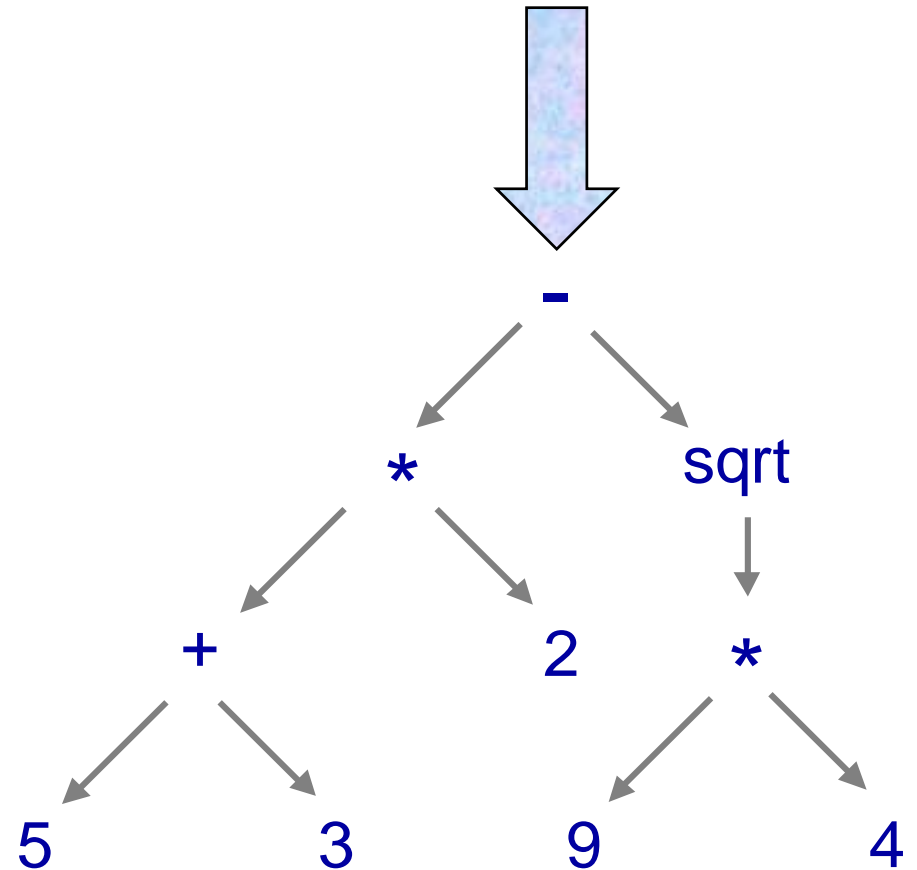
He ate an apple on the desk.

parse



## Jack

(5+3)\*2 - sqrt(9\*4)



# Regular expressions

---

■  $a|b^*$

$\{\epsilon, "a", "b", "bb", "bbb", \dots\}$

■  $(a|b)^*$

$\{\epsilon, "a", "b", "aa", "ab", "ba", "bb", "aaa", \dots\}$

■  $ab^*(c|\epsilon)$

$\{a, "ac", "ab", "abc", "abb", "abbc", \dots\}$

- A computer program that generates lexical analyzers (scanners or lexers)
- Commonly used with the yacc parser generator.
- Structure of a Lex file

**Definition section**

**%%**

**Rules section**

**%%**

**C code section**

## Example of a Lex file

```
/***/ Definition section ***/
%{
/* C code to be copied verbatim */
#include <stdio.h>
%}

/* This tells flex to read only one input file */
%option noyywrap

/***/ Rules section ***/
%%

[0-9]+  {
        /* yytext is a string containing the
           matched text. */
        printf("Saw an integer: %s\n", yytext);
    }
.|\n    { /* Ignore all other characters. */ }
```

## Example of a Lex file

```
%%  
/*** C Code section ***/  
  
int main(void)  
{  
    /* Call the lexer, then quit. */  
    yylex();  
    return 0;  
}
```

## Example of a Lex file

```
> flex test.lex
  (a file lex.yy.c with 1,763 lines is generated)

> gcc lex.yy.c
  (an executable file a.out is generated)

> ./a.out < test.txt
Saw an integer: 123
Saw an integer: 2
Saw an integer: 6
```

test.txt    abc123z. !&\*2gj6

## Another Lex example

```
%{  
int num_lines = 0, num_chars = 0;  
%}  
  
%option noyywrap  
  
%%  
\n      ++num_lines; ++num_chars;  
.      ++num_chars;  
  
%%  
main() {  
    yylex();  
    printf( "# of lines = %d, # of chars = %d\n",  
            num_lines, num_chars );  
}
```



# A more complex Lex example

```
%{
/* need this for the call to atof() below */
#include <math.h>
%}
%option noyywrap

DIGIT      [0-9]
ID          [a-z][a-z0-9]*

%%

{DIGIT}+    {
              printf( "An integer: %s (%d)\n", yytext,
                      atoi( yytext ) );
            }

{DIGIT}+"."{DIGIT}* {
              printf( "A float: %s (%g)\n", yytext,
                      atof( yytext ) );
            }
```

## A more complex Lex example

```
if|then|begin|end|procedure|function  {
    printf( "A keyword: %s\n", yytext );
}

{ID}      printf( "An identifier: %s\n", yytext );

"+"|"-"|"="|" ("|")"  printf( "Symbol: %s\n", yytext );

[ \t\n]+   /* eat up whitespace */

.          printf("Unrecognized char: %s\n", yytext );

%%

void main(int argc, char **argv ) {
    if ( argc > 1 ) yyin = fopen( argv[1], "r" );
    else yyin = stdin;

    yylex();
}
```

# A more complex Lex example

pascal.txt

```
if (a+b) then
    foo=3.1416
else
    foo=12
```

output

```
A keyword: if
Symbol: (
An identifier: a
Symbol: +
An identifier: b
Symbol: )
A keyword: then
An identifier: foo
Symbol: =
A float: 3.1416 (3.1416)
An identifier: else
An identifier: foo
Symbol: =
An integer: 12 (12)
```

# Context-free grammar

---

- Terminals: 0, 1, #
- Non-terminals: A, B
- Start symbol: A
- Rules:
  - $A \rightarrow 0A1$
  - $A \rightarrow B$
  - $B \rightarrow \#$

- Simple (terminal) forms / complex (non-terminal) forms
- Grammar = set of rules on how to construct complex forms from simpler forms
- Highly recursive.

# Examples of context-free grammar

---

■  $S \rightarrow ()$

$$S \rightarrow (S)$$

$$S \rightarrow SS$$

■  $S \rightarrow a|aS|bS$

strings ending with 'a'

■  $S \rightarrow x$

$$S \rightarrow y$$

$$S \rightarrow S+S$$

$$S \rightarrow S-S$$

$$S \rightarrow S^*S$$

$$S \rightarrow S/S$$

$$S \rightarrow (S)$$

$$(x+y)^*x-x^*y/(x+x)$$

# Examples of context-free grammar

---

■ non-terminals:  $S, E, \text{Elist}$

■ terminals:  $\text{ID}, \text{NUM}, \text{PRINT}, +, :=, (, ), ;$

■ rules:

$S \rightarrow S; S$

$S \rightarrow \text{ID} := E$

$S \rightarrow \text{PRINT} ( \text{Elist} )$

$E \rightarrow \text{ID}$

$E \rightarrow \text{NUM}$

$E \rightarrow E + E$

$E \rightarrow ( S , \text{Elist} )$

$\text{Elist} \rightarrow E$

$\text{Elist} \rightarrow \text{Elist} , E$

Try to derive:  $\text{ID} = \text{NUM} ; \text{PRINT} ( \text{NUM} )$

slide credit: David Walker

# Examples of context-free grammar

■ non-terminals:  $S, E, \text{Elist}$

■ terminals:  $\text{ID}, \text{NUM}, \text{PRINT}, +, :=, (, ), ;$

■ rules:

$S \rightarrow S; S$

$S \rightarrow \text{ID} := E$

$S \rightarrow \text{PRINT} ( \text{Elist} )$

$E \rightarrow \text{ID}$

$E \rightarrow \text{NUM}$

$E \rightarrow E + E$

$E \rightarrow ( S , \text{Elist} )$

$\text{Elist} \rightarrow E$

$\text{Elist} \rightarrow \text{Elist} , E$

left-most derivation

```
S
S ; S
ID = E ; S
ID = NUM ; S
ID = NUM ; PRINT ( Elist )
ID = NUM ; PRINT ( E )
ID = NUM ; PRINT ( NUM )
```

right-most derivation

```
S
S ; S
S ; PRINT ( Elist )
S ; PRINT ( E )
S ; PRINT ( NUM )
ID = E ; PRINT ( NUM )
ID = NUM ; PRINT ( NUM )
```

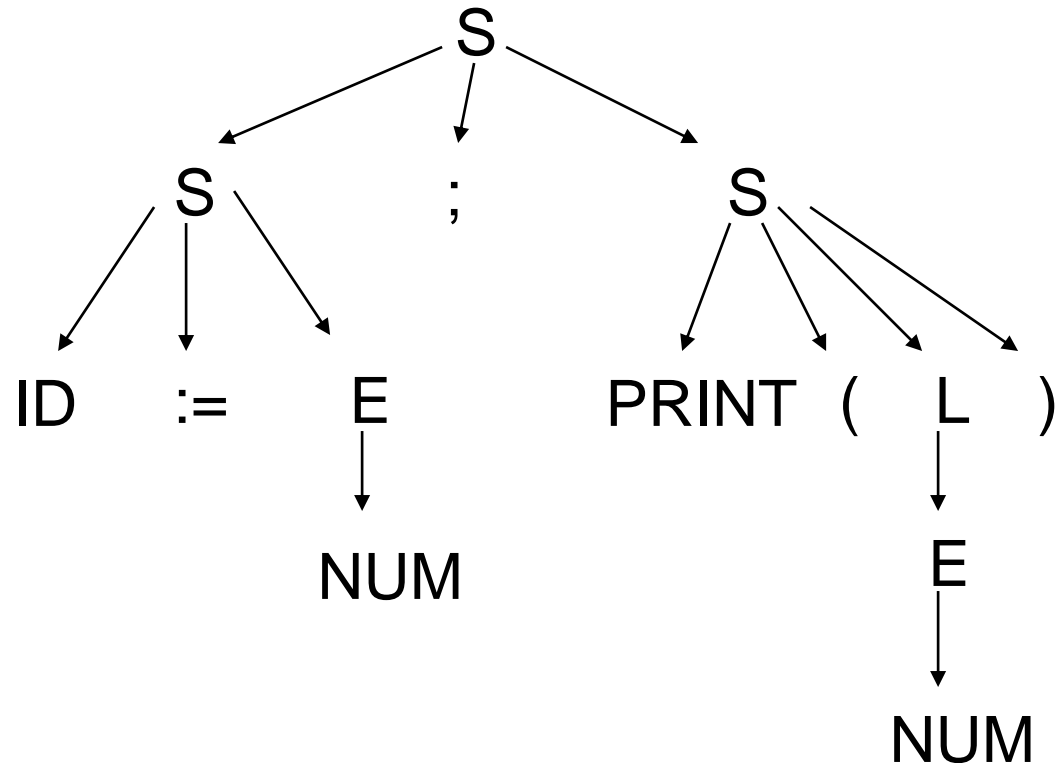
slide credit: David Walker

# Parse tree

## ■ Two derivations, but 1 tree

```
S
S ; S
ID = E ; S
ID = NUM ; S
ID = NUM ; PRINT ( Elist )
ID = NUM ; PRINT ( E )
ID = NUM ; PRINT ( NUM )
```

```
S
S ; S
S ; PRINT ( Elist )
S ; PRINT ( E )
S ; PRINT ( NUM )
ID = E ; PRINT ( NUM )
ID = NUM ; PRINT ( NUM )
```



slide credit: David Walker



# Ambiguous Grammars

---

- a grammar is ambiguous if the same sequence of tokens can give rise to two or more parse trees

- non-terminals:  $E$
- terminals:  $ID, NUM, PLUS, MUL$
- rules:
  - $E \rightarrow ID$
  - $E \rightarrow NUM$
  - $E \rightarrow E + E$
  - $E \rightarrow E * E$

characters:  $4 + 5 * 6$

tokens:  $NUM(4) \quad PLUS \quad NUM(5) \quad MUL \quad NUM(6)$

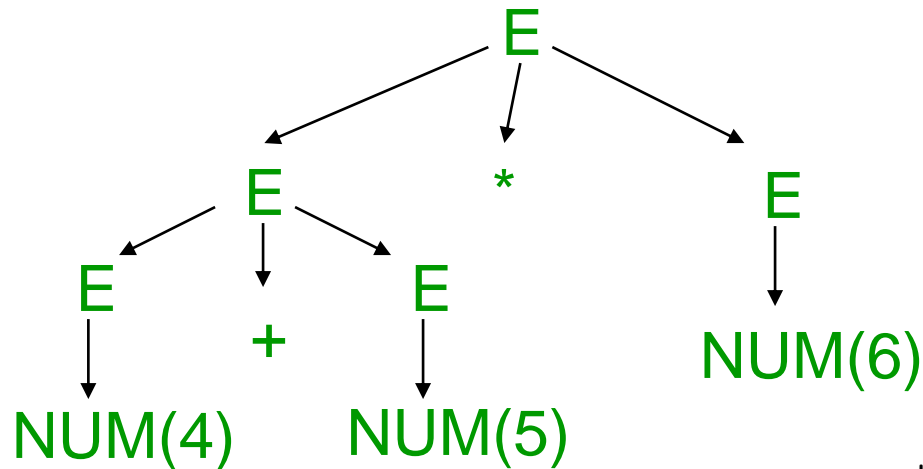
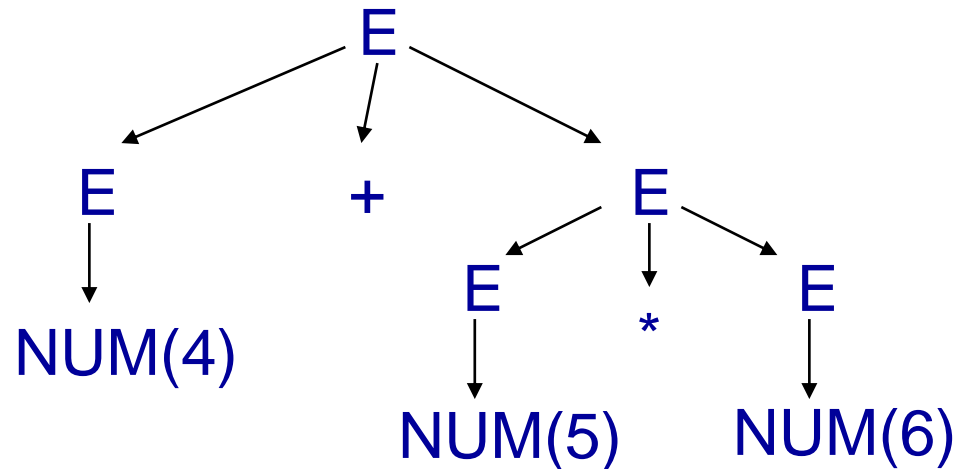
slide credit: David Walker

# Ambiguous Grammars

characters: 4 + 5 \* 6

tokens: NUM(4) PLUS NUM(5) MUL NUM(6)

$E \rightarrow ID$   
 $E \rightarrow NUM$   
 $E \rightarrow E + E$   
 $E \rightarrow E * E$



slide credit: David Walker

# Ambiguous Grammars

---

- problem: compilers use parse trees to interpret the meaning of parsed expressions
  - different parse trees have different meanings
  - eg:  $(4 + 5) * 6$  is not  $4 + (5 * 6)$
  - languages with ambiguous grammars are **DISASTROUS**; The meaning of programs isn't well-defined! You can't tell what your program might do!
- solution: rewrite grammar to eliminate ambiguity
  - fold **precedence** rules into grammar to disambiguate
  - fold **associativity** rules into grammar to disambiguate
  - other tricks as well

slide credit: David Walker

# Recursive descent parser

---

## ■ Recursive Descent Parsing

- aka: predictive parsing; top-down parsing
- simple, efficient
- can be coded by hand in ML quickly
- parses many, but not all CFGs
  - parses LL(1) grammars
    - Left-to-right parse; Leftmost-derivation; 1 symbol lookahead
- key ideas:
  - one recursive function for each non terminal
  - each production becomes one clause in the function

slide credit: David Walker

# Recursive descent parser

---

- Non-terminals:  $S, E, L$
- Terminals: NUM, IF, THEN, ELSE, BEGIN, END, PRINT, =, ;
- Rules:
  1.  $S \rightarrow \text{IF } E \text{ THEN } S \text{ ELSE } S$
  2.      $| \text{ BEGIN } S \text{ L}$
  3.      $| \text{ PRINT } E$
  4.  $L \rightarrow \text{END}$
  5.      $| ; S \text{ L}$
  6.  $E \rightarrow \text{NUM} = \text{NUM}$

slide credit: David Walker

# Recursive descent parser

- Non-terminals: S, E, L
- Terminals: NUM, IF, THEN, ELSE, BEGIN, END, PRINT, =, , ;
- Rules:
  1. S → IF E THEN S ELSE S
  2.   | BEGIN S L
  3.   | PRINT E
  4. L → END
  5.   | ; S L
  6. E → NUM = NUM

```
S()
{
    switch (next()) {
        case IF:
            eat(IF); E(); eat(THEN);
            S(); eat(ELSE); S();
            break;
        case BEGIN:
            eat(BEGIN); S(); L();
            break;
        case PRINT:
            eat(PRINT); E();
            break;
    }
}
```

slide credit: David Walker

# Recursive descent parser

- Non-terminals: S, E, L
- Terminals: NUM, IF, THEN, ELSE, BEGIN, END, PRINT, EQ(=), SEMI(;)
- Rules:
  1. S → IF E THEN S ELSE S
  2.   | BEGIN S L
  3.   | PRINT E
  4. L → END
  5.   | ; S L
  6. E → NUM = NUM

```
L()
{
    switch (next()) {
        case END:
            eat(END);
            break;
        case SEMI:
            eat(SEMI); S(); L();
            break;
        default:
            error();
    }
}
```

slide credit: David Walker

# Recursive descent parser

- Non-terminals: S, E, L
- Terminals: NUM, IF, THEN, ELSE, BEGIN, END, PRINT, EQ(=), SEMI(;)
- Rules:
  1. S → IF E THEN S ELSE S
  2.   | BEGIN S L
  3.   | PRINT E
  4. L → END
  5.   | ; S L
  6. E → NUM = NUM

```
E()  
{  
    eat(NUM);  
    eat(EQ);  
    eat(NUM);  
}
```



# Recursive descent parser

---

- Non-terminals:  $S, A, E, L$
- Terminals: EOF, ID, NUM, ASSIGN(:=), PRINT, LPAREN(() , RPAREN())
- Rules:
  1.  $S \rightarrow A \text{ EOF}$
  2.  $A \rightarrow \text{ID} := E$
  3.      $| \text{PRINT}(L)$
  4.  $E \rightarrow \text{ID}$
  5.      $| \text{NUM}$
  6.  $L \rightarrow E$
  7.      $| L, E$

slide credit: David Walker

# Recursive descent parser

---

- Non-terminals:  $S, A, E, L$
- Terminals: EOF, ID, NUM, ASSIGN( $:=$ ), PRINT, LPAREN( $($ ), RPAREN( $)$ )
- Rules:

1.  $S \rightarrow A \text{ EOF}$
2.  $A \rightarrow \text{ID} := E$
3.     $| \text{PRINT}(L)$
4.  $E \rightarrow \text{ID}$
5.     $| \text{NUM}$
6.  $L \rightarrow E$
7.     $| L, E$

```
S()  
{  
    A();  
    eat(EOF);  
}
```

slide credit: David Walker

# Recursive descent parser

- Non-terminals: S, A, E, L
- Terminals: EOF, ID, NUM, ASSIGN(:=), PRINT, LPAREN(() , RPAREN())

- Rules:

1.  $S \rightarrow A \text{ EOF}$
2.  $A \rightarrow \text{ID} := E$
3.      $| \text{PRINT}(L)$
4.  $E \rightarrow \text{ID}$
5.      $| \text{NUM}$
6.  $L \rightarrow E$
7.      $| L, E$

```
A()
{
    switch (next()) {
        case ID:
            eat(ID); eat(ASSIGN);
            E();
            break;
        case PRINT:
            eat(PRINT); eat(LPAREN);
            L(); eat(RPAREN);
            break;
    }
}
```

slide credit: David Walker

# Recursive descent parser

- Non-terminals: S, A, E, L
- Terminals: EOF, ID, NUM, ASSIGN(:=), PRINT, LPAREN(() , RPAREN())

- Rules:

1.  $S \rightarrow A \text{ EOF}$
2.  $A \rightarrow ID := E$
3.  $\quad \mid \text{PRINT}(L)$
4.  $E \rightarrow \text{ID}$
5.  $\quad \mid \text{NUM}$
6.  $L \rightarrow E$
7.  $\quad \mid L, E$

```
E()
{
    switch (next()) {
        case ID:
            eat(ID);
            break;
        case NUM:
            eat(NUM);
            break;
    }
}
```

slide credit: David Walker

# Recursive descent parser

- Non-terminals: S, A, E, L
- Terminals: EOF, ID, NUM, ASSIGN(:=), PRINT, LPAREN(() , RPAREN())

- Rules:

1.  $S \rightarrow A \text{ EOF}$
2.  $A \rightarrow ID := E$
3.  $\quad \mid \text{PRINT}(L)$
4.  $E \rightarrow ID$
5.  $\quad \mid \text{NUM}$
6.  $L \rightarrow E$
7.  $\quad \mid L, E$

```
L()
{
    switch (next()) {
        case ID:
            ???
        case NUM:
            ???
    }
```

Problem:

E could be ID

L could be E could be ID

# Recursive descent parser

---

- Non-terminals:  $S, A, E, L$
- Terminals: EOF, ID, NUM, ASSIGN(:=), PRINT, LPAREN(() , RPAREN())
- Rules:

1.  $S \rightarrow A \text{ EOF}$

2.  $A \rightarrow \text{ID} := E$

3.  $\quad \mid \text{PRINT}(L)$

4.  $E \rightarrow \text{ID}$

5.  $\quad \mid \text{NUM}$

6.  $L \rightarrow E$

7.  $\quad \mid L, E$

Problem:

$E$  could be ID

$L$  could be  $E$  could be ID

$L \rightarrow E M$

$M \rightarrow , E M$

$\quad \mid \epsilon$

slide credit: David Walker

# A typical grammar of a typical C-like language

---

## Code samples

```
while (expression) {  
    if (expression)  
        statement;  
    while (expression) {  
        statement;  
        if (expression)  
            statement;  
    }  
    while (expression) {  
        statement;  
        statement;  
    }  
}
```

```
if (expression) {  
    statement;  
    while (expression)  
        statement;  
}  
if (expression)  
    if (expression)  
        statement;  
}
```

# A typical grammar of a typical C-like language

```
program:          statement;

statement:        whileStatement
                  | ifStatement
                  | // other statement possibilities ...
                  | '{' statementSequence '}'

whileStatement:   'while' '(' expression ')' statement

ifStatement:      simpleIf
                  | ifElse

simpleIf:          'if' '(' expression ')' statement

ifElse:           'if' '(' expression ')' statement
                  'else' statement

statementSequence: '' // null, i.e. the empty sequence
                  | statement ';' statementSequence

expression:       // definition of an expression comes here
```



# Parse tree

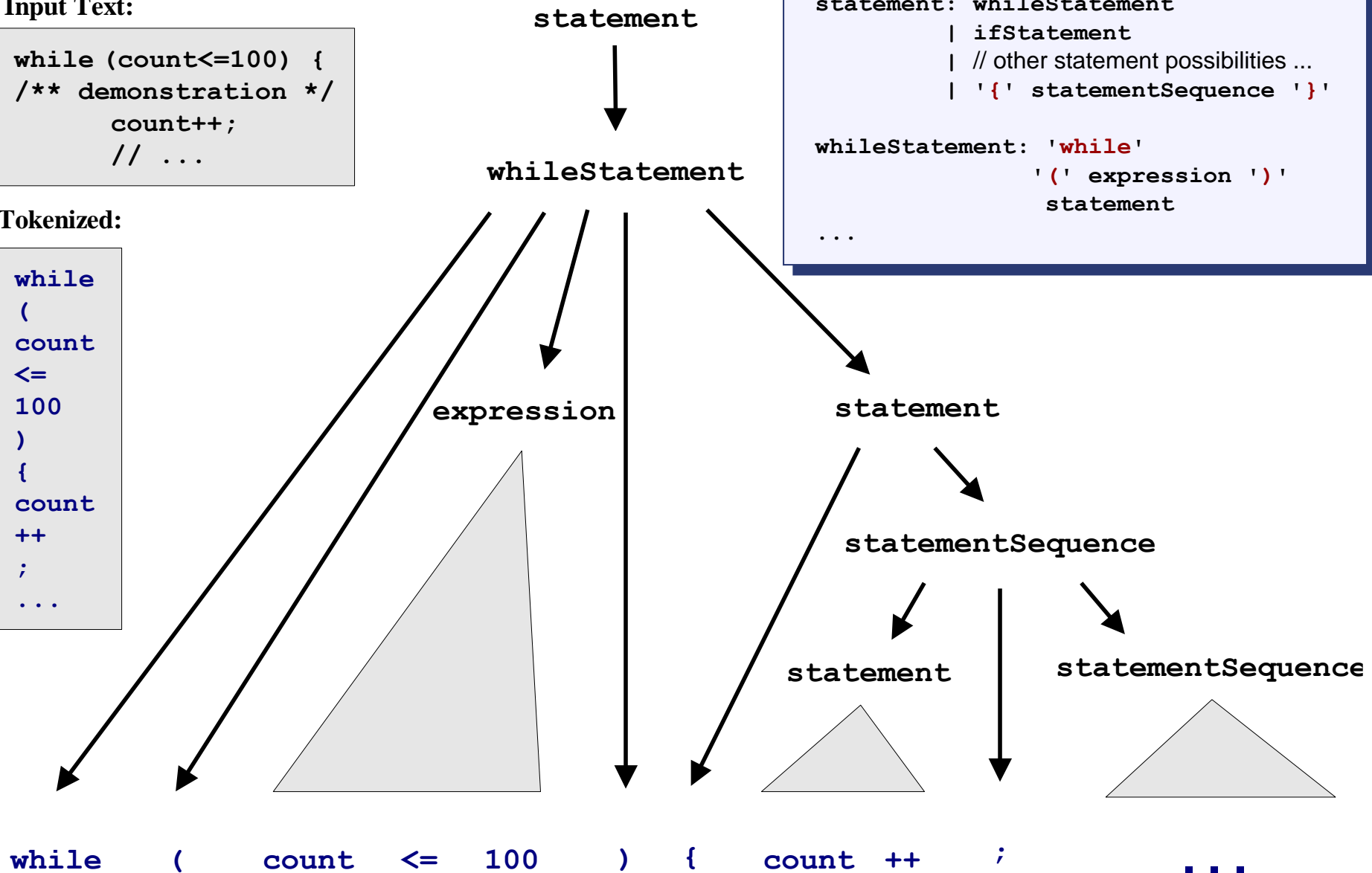
Input Text:

```
while (count<=100) {  
  /** demonstration */  
  count++;  
  // ...  
}
```

Tokenized:

```
while  
(  
count  
<=  
100  
)  
{  
count  
++  
;  
...  
}
```

```
program:  statement;  
  
statement: whileStatement  
          | ifStatement  
          | // other statement possibilities ...  
          | '{' statementSequence '}'  
  
whileStatement: 'while'  
               '(' expression ')' statement  
               ...
```



# Recursive descent parsing

```
...
statement:  whileStatement
           | ifStatement
           | ...           // other statement possibilities follow
           | '{' statementSequence '}'

whileStatement: 'while' '(' expression ')' statement

ifStatement: ...           // if definition comes here

statementSequence: ' ' // null, i.e. the empty sequence
                  | statement ';' statementSequence

expression: ... // definition of an expression comes here
...           // more definitions follow
```

## code sample

```
while (expression) {
    statement;
    statement;
    while (expression) {
        while (expression)
            statement;
            statement;
    }
}
```

- Highly recursive
- LL(0) grammars: the first token determines in which rule we are
- In other grammars you have to look ahead 1 or more tokens
- Jack is almost LL(0).

Parser implementation: a set of parsing methods, one for each rule:

- `parseStatement()`
- `parseWhileStatement()`
- `parseIfStatement()`
- `parseStatementSequence()`
- `parseExpression()`.

# The Jack grammar

**Lexical elements:** The Jack language includes five types of terminal elements (tokens):

keyword: `'class' | 'constructor' | 'function' |`  
`'method' | 'field' | 'static' | 'var' |`  
`'int' | 'char' | 'boolean' | 'void' | 'true' |`  
`'false' | 'null' | 'this' | 'let' | 'do' |`  
`'if' | 'else' | 'while' | 'return'`

symbol: `'{' | '}' | '(' | ')' | '[' | ']' | '.' |`  
`',' | ';' | '+' | '-' | '*' | '/' | '&' |`  
`'|' | '<' | '>' | '=' | '~'`

integerConstant: A decimal number in the range 0 .. 32767.

StringConstant: `'''` A sequence of Unicode characters not including double quote or newline `'''`

identifier: A sequence of letters, digits, and underscore (`' '`) not starting with a digit.

**'x'**: x appears verbatim

**x**: x is a language construct

**x?**: x appears 0 or 1 times

**x\***: x appears 0 or more times

**x|y**: either x or y appears

**(x,y)**: x appears, then y.

# The Jack grammar

**Program structure:** A Jack program is a collection of classes, each appearing in a separate file. The compilation unit is a class. A class is a sequence of tokens structured according to the following context free syntax:

class: `'class' className '{' classVarDec* subroutineDec* '}'`

classVarDec: `('static' | 'field') type varName (',' varName)* ';'`

type: `'int' | 'char' | 'boolean' | className`

subroutineDec: `('constructor' | 'function' | 'method')  
('void' | type) subroutineName '(' parameterList ')'  
subroutineBody`

parameterList: `((type varName) (',' type varName)*)?`

subroutineBody: `'{' varDec* statements '}'`

varDec: `'var' type varName (',' varName)* ';'`

className: identifier

subroutineName: identifier

varName: identifier

**'x'**: x appears verbatim

**x**: x is a language construct

**x?**: x appears 0 or 1 times

**x\***: x appears 0 or more times

**x|y**: either x or y appears

**(x,y)**: x appears, then y.

# The Jack grammar

## Statements:

statements: statement\*

statement: letStatement | ifStatement | whileStatement |  
doStatement | returnStatement

letStatement: **'let'** varName ( **'['** expression **']'** )? **'='** expression **';'**

ifStatement: **'if'** ( expression ) **'{'** statements **'}'**  
( **'else'** **'{'** statements **'}'** )?

whileStatement: **'while'** ( expression ) **'{'** statements **'}'**

doStatement: **'do'** subroutineCall **';'**

ReturnStatement **'return'** expression? **';'**

**'x'**: x appears verbatim

**x**: x is a language construct

**x?**: x appears 0 or 1 times

**x\***: x appears 0 or more times

**x|y**: either x or y appears

**(x,y)**: x appears, then y.

# The Jack grammar

## Expressions:

expression: term (op term)\*

term: integerConstant | stringConstant | keywordConstant |  
varName | varName '[' expression ']' | subroutineCall |  
'(' expression ')' | unaryOp term

subroutineCall: subroutineName '(' expressionList ')' | (className |  
varName) '.' subroutineName '(' expressionList ')'

expressionList: (expression (',' expression)\* )?

op: '+' | '-' | '\*' | '/' | '&' | '|' | '<' | '>' | '='

unaryOp: '-' | '~'

KeywordConstant: 'true' | 'false' | 'null' | 'this'

**'x'**: x appears verbatim

**x**: x is a language construct

**x?**: x appears 0 or 1 times

**x\***: x appears 0 or more times

**x|y**: either x or y appears

**(x,y)**: x appears, then y.

# Jack syntax analyzer in action

```
Class Bar {  
    method Fraction foo(int y)  
        var int temp; // a variable  
        let temp = (xxx+12)*-63;  
        ...  
    ...  
}
```

Syntax analyzer

## Syntax analyzer

- With the grammar, we can write a syntax analyzer program (parser)
- The syntax analyzer takes a source text file and attempts to match it on the language grammar
- If successful, it can generate a parse tree in some structured format, e.g. XML.

```
<varDec>  
    <keyword> var </keyword>  
    <keyword> int </keyword>  
    <identifier> temp </identifier>  
    <symbol> ; </symbol>  
</varDec>  
<statements>  
    <letStatement>  
        <keyword> let </keyword>  
        <identifier> temp </identifier>  
        <symbol> = </symbol>  
        <expression>  
            <term>  
                <symbol> ( </symbol>  
                <expression>  
                    <term>  
                        <identifier> xxx </identifier>  
                    </term>  
                    <symbol> + </symbol>  
                    <term>  
                        <int.Const.> 12 </int.Const.>  
                    </term>  
                </expression>  
            </term>  
        </expression>  
    </letStatement>  
    ...
```

# Jack syntax analyzer in action

```
Class Bar {  
    method Fraction foo(int y)  
        var int temp; // a variable  
        let temp = (xxx+12)*-63;  
        ...  
    ...  
}
```

Syntax analyzer

- If **xxx** is non-terminal, output:

**<xxx>**

Recursive code for  
the body of **xxx**

**</xxx>**

- If **xxx** is terminal  
(keyword, symbol, constant,  
or identifier) , output:

**<xxx>**

**xxx** value

**</xxx>**

```
<varDec>  
    <keyword> var </keyword>  
    <keyword> int </keyword>  
    <identifier> temp </identifier>  
    <symbol> ; </symbol>  
</varDec>  
<statements>  
    <letStatement>  
        <keyword> let </keyword>  
        <identifier> temp </identifier>  
        <symbol> = </symbol>  
        <expression>  
            <term>  
                <symbol> ( </symbol>  
                <expression>  
                    <term>  
                        <identifier> xxx </identifier>  
                    </term>  
                    <symbol> + </symbol>  
                    <term>  
                        <int.Const.> 12 </int.Const.>  
                    </term>  
                </expression>  
            </term>  
        </expression>  
    </letStatement>  
    ...
```



# The Jack grammar

## Expressions:

expression: term (op term)\*

term: integerConstant | stringConstant | keywordConstant |  
varName | varName '[' expression ']' | subroutineCall |  
'(' expression ')' | unaryOp term

subroutineCall: subroutineName '(' expressionList ')' | (className |  
varName) '.' subroutineName '(' expressionList ')'

expressionList: (expression (',' expression)\* )?

op: '+' | '-' | '\*' | '/' | '&' | '|' | '<' | '>' | '='

unaryOp: '-' | '~'

KeywordConstant: 'true' | 'false' | 'null' | 'this'

**'x'**: x appears verbatim

**x**: x is a language construct

**x?**: x appears 0 or 1 times

**x\***: x appears 0 or more times

**x|y**: either x or y appears

**(x,y)**: x appears, then y.

# Recursive descent parser (simplified expression)

---

- $EXP \rightarrow TERM (OP TERM)^*$
- $TERM \rightarrow integer \mid variable$
- $OP \rightarrow + \mid - \mid * \mid /$

# From parsing to code generation

---

- $EXP \rightarrow TERM (OP TERM)^*$
- $TERM \rightarrow \text{integer} \mid \text{variable}$
- $OP \rightarrow + \mid - \mid * \mid /$

```
EXP() :  
    TERM();  
    while (next() == OP)  
        OP();  
        TERM();
```

# From parsing to code generation

---

- $EXP \rightarrow TERM (OP TERM)^*$
- $TERM \rightarrow \text{integer} \mid \text{variable}$
- $OP \rightarrow + \mid - \mid * \mid /$

```
EXP() :  
    TERM();  
    while (next() == OP)  
        OP();  
        TERM();
```

```
TERM():  
    switch (next())  
        case INT:  
            eat(INT);  
        case VAR:  
            eat(VAR);
```

# From parsing to code generation

---

- $\text{EXP} \rightarrow \text{TERM} (\text{OP} \text{ TERM})^*$
- $\text{TERM} \rightarrow \text{integer} \mid \text{variable}$
- $\text{OP} \rightarrow + \mid - \mid * \mid /$

OP():

switch (next())

case +: eat(ADD);

case -: eat(SUB);

case \*: eat(MUL);

case /: eat(DIV);

EXP() :

TERM();

while (next() == OP)

OP();

TERM();

TERM():

switch (next())

case INT:

eat(INT);

case VAR:

eat(VAR);

# From parsing to code generation

---

- $\text{EXP} \rightarrow \text{TERM} (\text{OP} \text{ TERM})^*$
- $\text{TERM} \rightarrow \text{integer} \mid \text{variable}$
- $\text{OP} \rightarrow + \mid - \mid * \mid /$

OP():

switch (next())

case +: eat(ADD);

case -: eat(SUB);

case \*: eat(MUL);

case /: eat(DIV);

EXP() :

TERM();

while (next() == OP)

OP();

TERM();

TERM():

switch (next())

case INT:

eat(INT);

case VAR:

eat(VAR);

# From parsing to code generation

- $\text{EXP} \rightarrow \text{TERM} (\text{OP} \text{ TERM})^*$
- $\text{TERM} \rightarrow \text{integer} \mid \text{variable}$
- $\text{OP} \rightarrow + \mid - \mid * \mid /$

OP(): **print('<op>');**

switch (next())

case +: eat(ADD);

**print('<sym> + </sym>');** TERM(): **print('<term>');**

case -: eat(SUB);

**print('<sym> - </sym>');**

case \*: eat(MUL);

**print('<sym> \* </sym>');**

case /: eat(DIV);

**print('<sym> / </sym>');** **print('</term>');**

**print('</op>');**

EXP() : **print('<exp>');**

TERM();

while (next() == OP)

OP();

TERM();

**print('</exp>');**

switch (next())

case INT: **print('<int> next() </int>');**

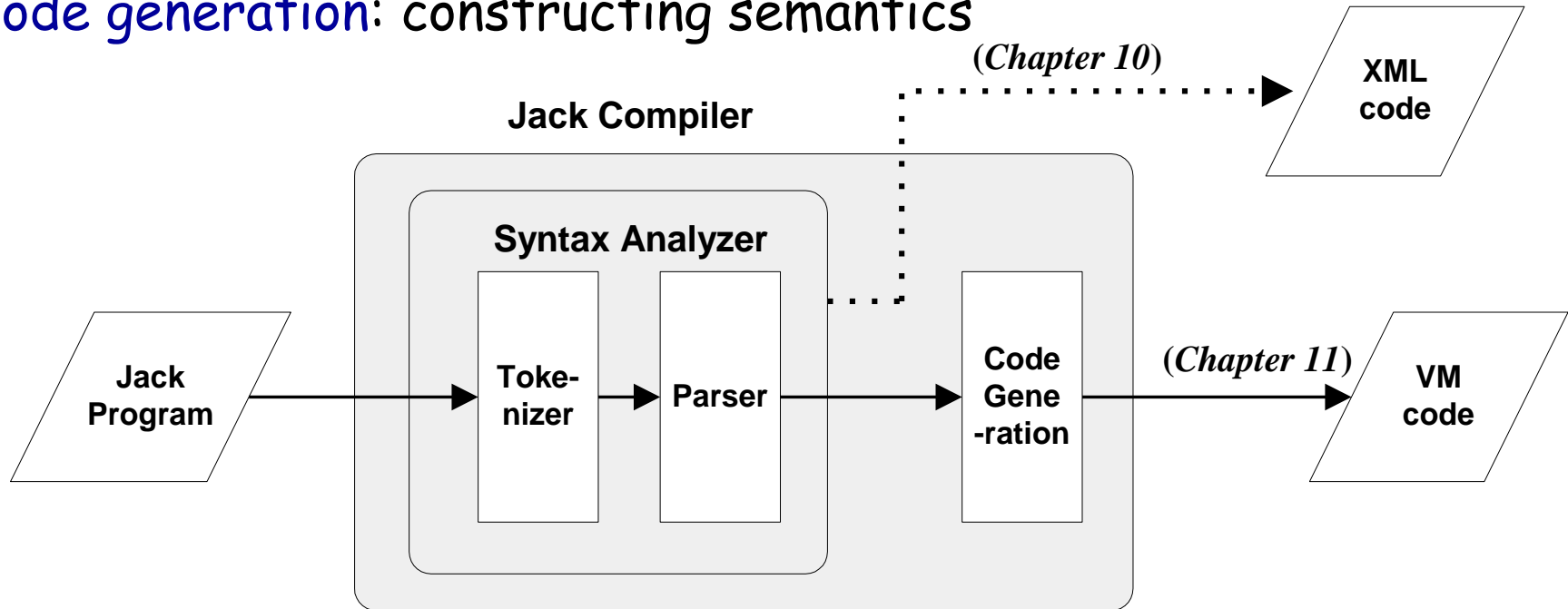
eat(INT);

case VAR: **print('<id> next() </id>');**

eat(VAR);

# Summary and next step

- **Syntax analysis:** understanding syntax
- **Code generation:** constructing semantics



## The code generation challenge:

- Extend the syntax analyzer into a full-blown compiler that, instead of passive XML code, generates executable VM code
- Two challenges: (a) handling data, and (b) handling commands.



- The parse tree can be constructed on the fly
- The Jack language is intentionally simple:
  - Statement prefixes: `let`, `do`, ...
  - No operator priority
  - No error checking
  - Basic data types, etc.
- The Jack compiler: designed to illustrate the key ideas that underlie modern compilers, leaving advanced features to more advanced courses
- Richer languages require more powerful compilers

- Syntax analyzers can be built using:
  - `Lex` tool for tokenizing (`flex`)
  - `Yacc` tool for parsing (`bison`)
  - Do everything from scratch (our approach ...)
- Industrial-strength compilers: (LLVM)
  - Have good error diagnostics
  - Generate tight and efficient code
  - Support parallel (multi-core) processors.