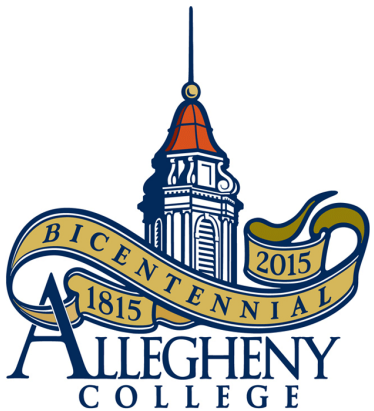
CS201 - PL'S

Lexical & Syntactic Analysis

Aravind Mohan

Allegheny College

September 6, 2021



Most Important Steps in Compilation

- Optional Preprocessing
- Lexical analysis (scanning)
- Syntax analysis (parsing)
- Semantic analysis
- Intermediate code generation
- Optimization (usually machine-independent)
- Final code generation
- Optional final optimization

Lexical Analysis

0	<unsigned int>
1	'('
2	')'
3	'+'
4	'-'
5	"for"
6	"while"
7	<identifier> (not reserved)

8	';'
9	'='
10	"=="
11	"<="
12	">="
13	<string literal>
14	<
...	... etc. ...

A token is any component of a program that is generally treated as an indivisible piece, e.g., a variable name, an operator such as <=, a punctuation mark such as a semicolon, a string constant, etc.

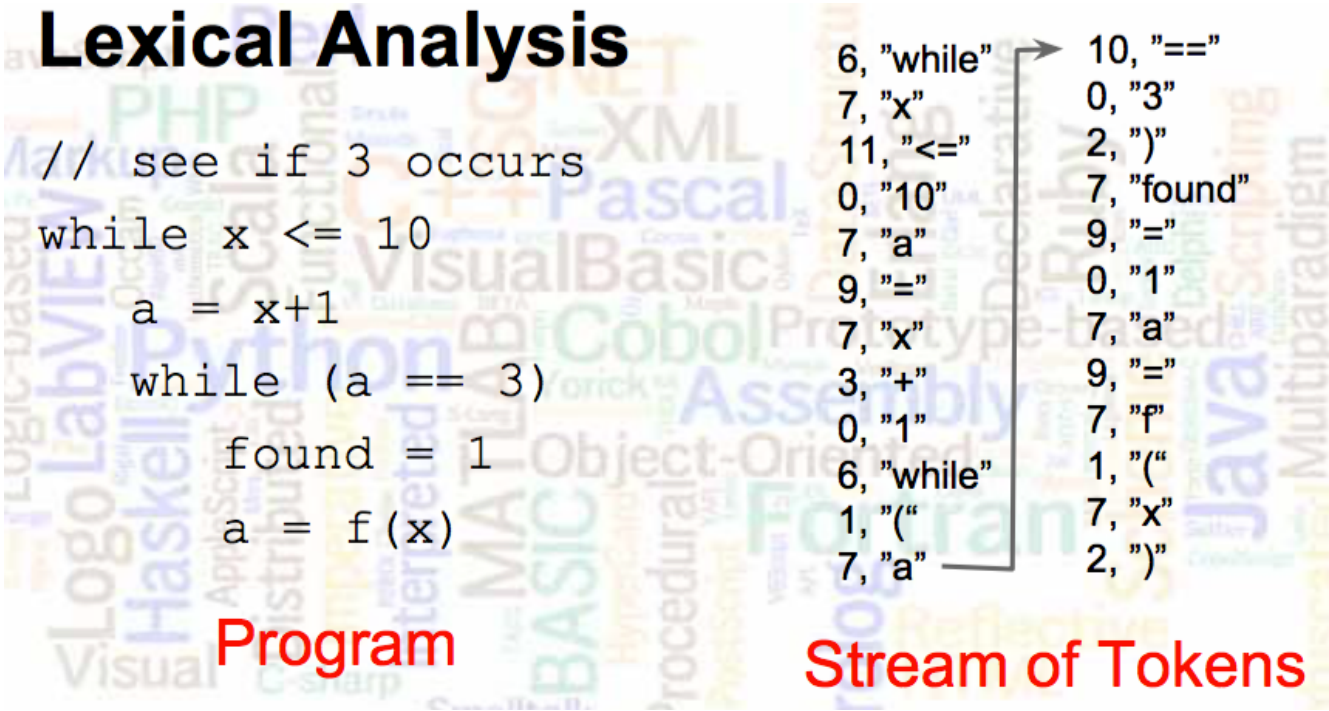
Lexical Analysis

For each token type, give a description:

- either a literal string (e.g., "$\leq$" or "while" to describe an operator or reserved word),
- or a $< rule >$ (e.g., the rule $< unsigned\ int >$ might stand for "a sequence of one or more digits"; the rule $< identifier >$ might stand for "a letter followed by a sequence of zero or more letters or digits."

Lexical Analysis

Lexical analysis produces a “token stream” in which the progam is reduced to a sequence of token types, each with its identifying number and the actual string (in the program) corresponding to it.


Lexical Analysis
// see if 3 occurs
while x <= 10
a = x+1
while (a == 3)
found = 1
a = f(x)
Program
6, "while"
7, "x"
11, "<="
0, "10"
7, "a"
9, "="
7, "x"
3, "+"
0, "1"
6, "while"
1, "("
7, "a"
10, "=="
0, "3"
2, ")"
7, "found"
9, "="
0, "1"
7, "a"
9, "="
7, "f"
1, "("
7, "x"
2, ")"
Stream of Tokens

Syntactic Analysis

- The syntax of a language is described by a **grammar** that specifies the legal combinations of tokens.
- Grammars are often specified in BNF notation ("Backus Naur Form"):

```
<item1> ::= valid replacements for
<item1>
<item2> ::= valid replacements for
<item2>
```

Syntactic Analysis

Example: an expression can be either a simple variable identifier; an integer; or an expression, followed by an operator, followed by another expression:

CLassic BNF notation

```
<expr> ::= <id> | <int> | <expr> <op> <expr>
```

Alternative notations:

The book uses this notation (but as three separate rules)

```
expr ⟶ id | int | expr op expr
```

```
expr ::= id | int | expr {op expr}*
```

The symbol "|" means "or"

The "{...}*" means "zero or more repetitions of the items in {...}"

Grammars (Context-free Grammars)

- Collection of VARIABLES (things that can be replaced by other things), also called NON-TERMINALS.
- Collection of TERMINALS ("constants", strings that can't be replaced)
- One special variable called the START SYMBOL.
- Collection of RULES, also called PRODUCTIONS.

`variable` → `rule1 | rule2 | rule3 | ...`

You can also write each rule on a separate line (as in the book)

Grammars (Context-free Gramars)

Grammar

A, B, and C are non-terminals.
0, 1, and 2 are terminals.
The start symbol is A.
The rules are:

- $A \rightarrow 0A|1C|2B|0$
- $B \rightarrow 0B|1A|2C|1$
- $C \rightarrow 0C|1B|2A|2$

Can this be parsed?

2011020

Grammars (Context-free Gramars)

Grammar

A, B, and C are non- terminals.
0, 1, and 2 are terminals.
The start symbol is A, the rules are:

- $A \rightarrow 0A|1C|2B|0$
- $B \rightarrow 0B|1A|2C|1$
- $C \rightarrow 0C|1B|2A|2$

Can this be parsed?

1112202

00102

2120

Syntactic Analysis

The "{...}+" means "one or more repetitions of the items in {...}"

prog ⟶ {*statement*}$^{+}$

statement ⟶ *assignment* | *loop* | *io*

assignment ⟶ *id* = *expression*

loop ⟶ **while** (*expression*) *prog*

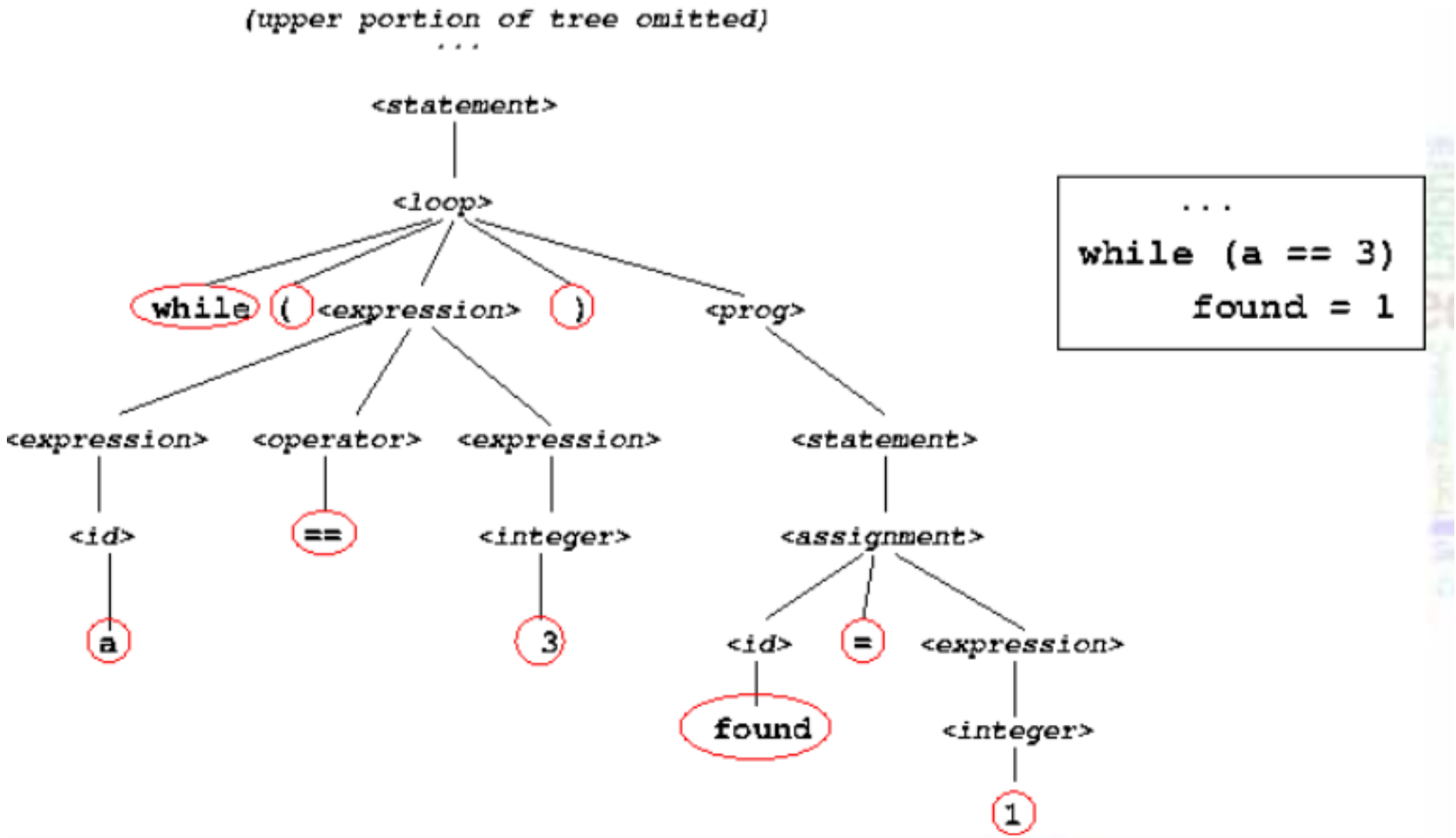
In this example, "=", "while", "(", and ")" are terminals

"A program is one or more statements."

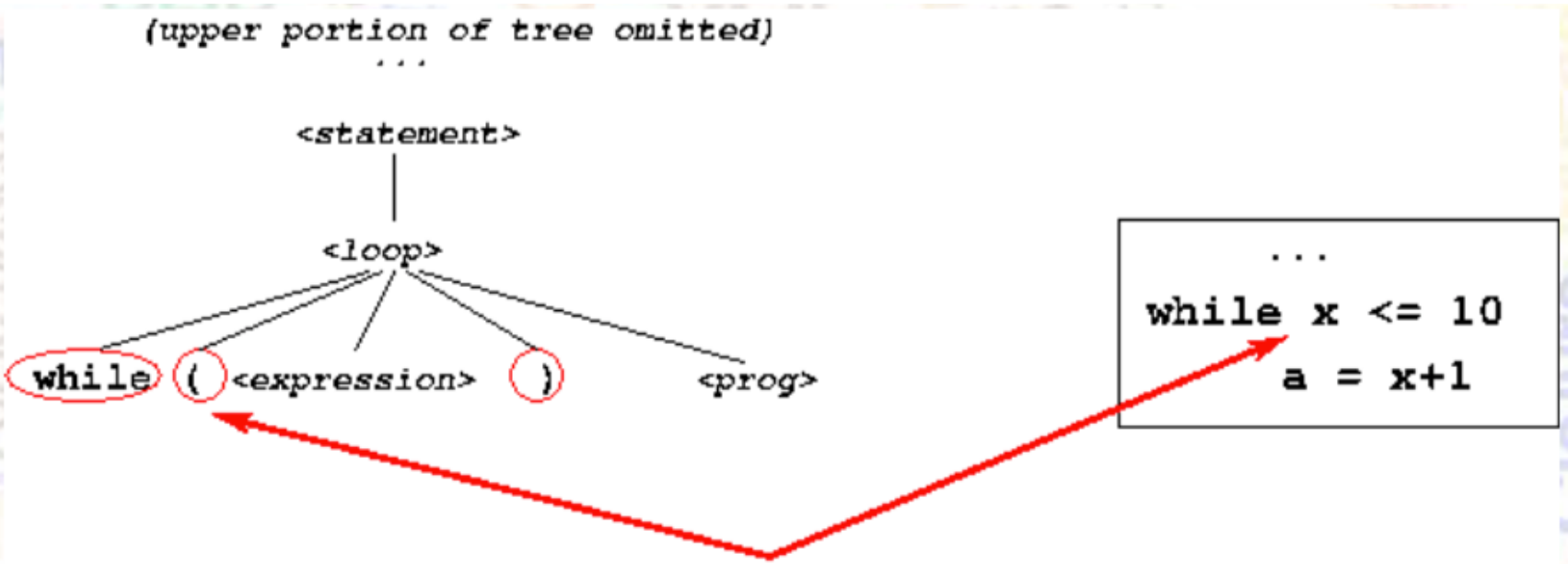
"A statement is an assignment, a loop, or an input/output command."

"An assignment is an identifier, followed by "=", followed by an expression."

Sample Parse Tree (portion)



Sample Parse Tree (failed)



Parse halts after "while" -- unable to match the "(" in the rule with a "(" in the input. An error is reported by the compiler.

Grammar for Java (version 8)

- Overview of notation used:
  `https://docs.oracle.com/javase/specs/jls/se8/html/jls-2.html`
- The full syntax grammar:
  `https://docs.oracle.com/javase/specs/jls/se8/html/jls-19.html`

Compiling

So far, we have looked at:

- the scanner (lexical analysis)–tokenizes input
- the parser (syntactic analysis)–validates structure
- Next, we do **semantic analysis**: is the program meaningful?

EXAMPLE:

In Java,

```
int i, i, i;
```

has the right structure for a declaration, but it's not legal to redeclare i within the same block of code.

Semantic Analysis

- During lexical analysis and parsing, as we process tokens we gather the user-defined names into a **symbol table**.
- Symbol table contains information such as:
  - where the symbol first appeared (usually in a declaration)
  - whether it has an initial value (parsing will tell us this)
  - what its type is (parsing tells us), etc.

Symbol Table

```
Line#    Program source

           ...
  20  int i = 10;
  21  double j = 3;
  22  int k;
           ...
```

Entry	Symbol	Declared at	Type	Init Value
...	...	...	...	...
5	int	<library>.103	--	(TYPE)
6	double	<library>.142	--	(TYPE)
7	i	<Lab3>.20	5	10
8	j	<Lab3>.21	6	3.0
9	k	<Lab3>.22	5	uninit
...	...	...	...	...

As the parser encounters names, it looks them up to see if they are already declared; if not, it creates a table entry. (Some names are pre-declared as part of the language.)

Symbol Table

```
Line#    Program source

          ...
20  int i = 10;
21  double j = 3;
22  int k;
23  k = i*j;
```

Entry	Symbol	Declared at	Type	Init Value
...	...	...	...	...
5	int	<library>.103	--	(TYPE)
6	double	<library>.142	--	(TYPE)
7	i	<Lab3>.20	5	10
8	j	<Lab3>.21	6	3.0
9	k	<Lab3>.22	5	uninit
...	...	...	...	...

```
test.java:23: error: possible loss of precision:
k = i*j;
     ^
    required: int, found: double
```

Symbol Table

```
Line#    Program source

            ...
20  int i = 10;
21  double j = 3;
22  int k;
23  k = i*j;
24  int j;
```

```
test.java:24: error: variable j is already defined in method main
(String[])
int j;
    ^
```

Entry	Symbol	Declared at	Type	Init Value
...	...	...	...	...
5	int	<library>.103	--	(TYPE)
6	double	<library>.142	--	(TYPE)
7	i	<Lab3>.20	5	10
8	j	<Lab3>.21	6	3.0
9	k	<Lab3>.22	5	uninit
...	...	...	...	...

Symbol Table

```
Line#   Program source
  20  int i = 10;
  21  double j = 3;
  22  int k;
  23  k = i*j;
  24  int j;
  25  m = 17;
```

```
test.java:25: error: cannot find symbol
m = 17;
^
  symbol:   variable m
```

Entry	Symbol	Declared at	Type	Init Value
...	...	...	...	...
5	int	<library>.103	--	(TYPE)
6	double	<library>.142	--	(TYPE)
7	i	<Lab3>.20	5	10
8	j	<Lab3>.21	6	3.0
9	k	<Lab3>.22	5	uninit
...	...	...	...	...

Intermediate Code Generation

- During this phase, the parsed program is converted into a simpler, step-by-step description in some **intermediate language**.
- Intermediate language may exist only as an internal representation within the compiler–it does not need be an “actual” language).
- A simple example is something called “three-address code.”

Intermediate Code Generation

In “Three-address code”, everything is reduced to simple operations on two values, with the result stored in a (possibly temporary) variable.

Source code:

```
double f = 3 + i * 7;
```

Three-address code:

temp1 = i
temp2 = 7
temp1 = temp1 * temp2
temp2 = 3
temp1 = temp2 + temp1
f = int-to-float(temp1)

Why Intermediate Code Generation?

Simple representation, easy to manipulate, close to assembly/machine language. E.g.,

```
temp1 = x
temp2 = 10
temp1 = temp1 + temp2
```

Three-address code

```
lw   $t0,x
li   $t1,10
add  $t0,$t0,$t1
```

MIPS assembly language

Many levels, from very basic:

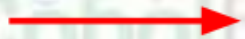
```
temp1 = temp1 + 1
temp1 = temp1 + x
temp1 = temp1 + 5
```

```
temp1 = temp1 + x
temp1 = temp1 + 6
```

"constant folding"

… to more complex:

```
temp1 = 1
temp2 = x
temp3 = temp2+9
temp2 = temp2*temp3
temp1 = 8
```



```
temp2 = x
temp3 = temp2+9
temp2 = temp2*temp3
temp1 = 8
```

"dead code removal"

Optimization

Here is a list of the many, many kinds of optimizations:
`http://www.compileroptimizations.com/`

(The examples show the effects on source code, but the optimizations are usually made on the intermediate code.)

Code Generation

- It is usually very straightforward to generate machine instructions from intermediate code, since the intermediate code is simple.
- Some further machine-specific optimizations may take place during or after this stage.

Pipelining

- The steps in compilation don't need to be done in whole phases, one after the other, but can be "**pipelined**":
- `int count = 1;`
  create tokens, pass to parser, generate some intermediate code
- `j = j + count;`
  create tokens, pass to parser, generate some intermediate code
  ... etc. ...

Reading Assignment

PLP Chapter 01, Section 1.4, 1.6