

# Compiler Construction

## SMD163

### Lecture 2: Lexical analysis

Viktor Leijon & Peter Jonsson with slides by Johan Nordlander.  
Contains material generously provided by Mark P. Jones

COMPUTER SCIENCE  
AND ELECTRICAL ENGINEERING  
LULEÅ UNIVERSITY OF TECHNOLOGY

1

## Practical matters

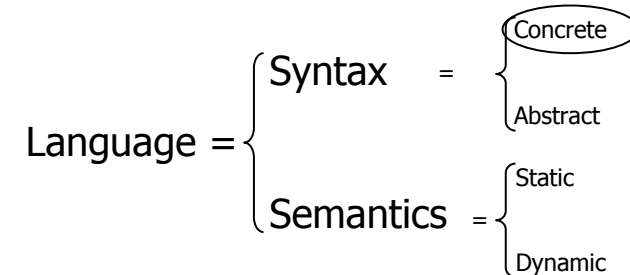
- ◆ Course registration through the student portal, deadline is **today**.
- ◆ The homepage is updated with the first half of the project.
- ◆ Deadline for homework 1 is Monday at 10 am.
  
- ◆ The importance of syntax and semantics – even in prose.

2

## Basics of Lexical Analysis:

3

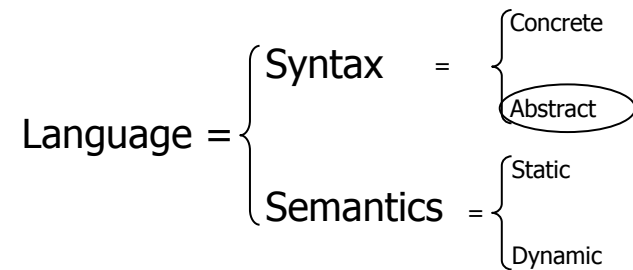
## Some definitions:



Concrete Syntax: the representation of a program text in its source form as a sequence of bits/bytes/characters/lines.

4

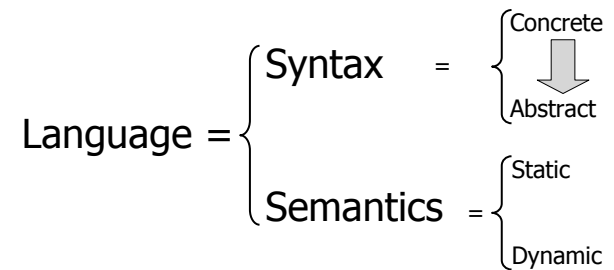
## Some definitions:



Abstract Syntax: the representation of program structure, independent of written form.

5

## Syntax Analysis:

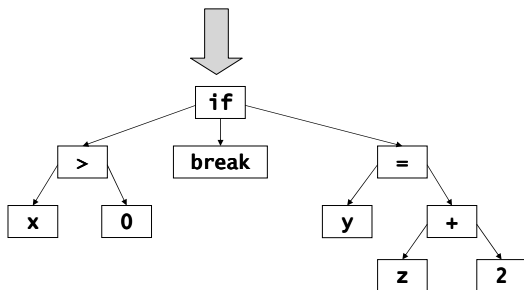


This is one of the areas where theoretical computer science has had major impact on the practice of software development.

6

## Syntax Analysis:

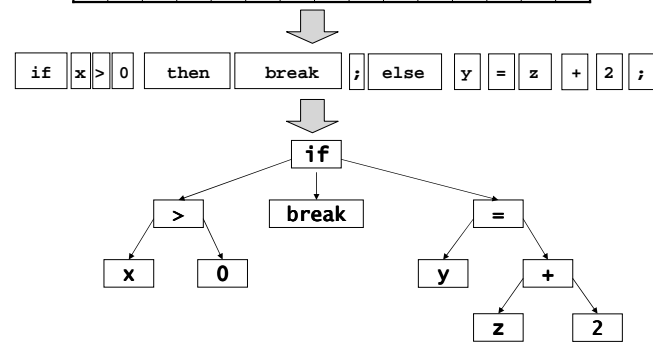
i	f		x	>	0		t	h	e	n		b	r	e
a	k	;		e	l	s	e		y	=	z	+	2	;



7

## Syntax Analysis:

i	f		x	>	0		t	h	e	n		b	r	e
a	k	;		e	l	s	e		y	=	z	+	2	;



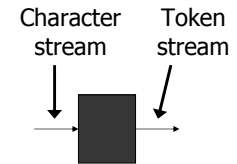
8

## Separate Lexical Analysis?

- ◆ It isn't necessary to separate lexical analysis from parsing.
- ◆ But it does have several potential benefits:
  - Simpler design (separation of concerns);
  - Potentially more efficient (parsing often uses more expensive techniques than lexical analysis);
  - Isolates machine/character set dependencies;
  - Good tool support.
- ◆ Modern language specifications often separate lexical and grammatical syntax.

9

## Lexical Analysis:



- ◆ Lexer: carries out lexical analysis.
- ◆ Goal: to recognize and identify the sequence of tokens represented by the characters in a program text.
- ◆ The definition of tokens, i.e., lexical structure, is an important part of a language specification.

10

## Basic Terminology:

- ◆ Lexeme: A particular sequence of characters that might appear together in an input stream as the representation for a single entity.
- ◆ Some lexemes in Java.

```
0.0    3.14    1e-97d    true
false  "false"  "hello world!" if
then  ;    {    }    ';'    'a'
'\n' class  String  main  temp
+    &&    0xC0B0L    2000
```
- ◆ Token: A lexeme treated as a data atom

11

## Basic Terminology:

- ◆ Token type: A name for a group of lexemes. (A description of what each lexeme represents.)  
In Java:
  - 0.0 3.14 1e-97d are double literals
  - true false are boolean literals
  - "false" "hello world!" are string literals
  - if then static are keywords
  - + && are operators
  - ; { } are separators
  - ';' 'a' '\n' are character literals
  - String main temp are identifiers
  - 0xC0B0L 2000 are integer literals
- ◆ Tokens/lexemes are normally chosen so that each lexeme has just one token type.

12

## Basic Terminology:

- ◆ Pattern: A description of the way that lexemes are written.
- ◆ In Java, *"an identifier is an unlimited-length sequence of Java letters and Java digits, the first of which must be a Java letter. An identifier cannot have the same spelling as a keyword ..."*
- ◆ We'll see how to make this kind of thing more precise soon...

13

## Token Attributes:

Many token types have associated attributes:

- ◆ In many cases, the lexeme itself might be used as an attribute;
- ◆ Literals and constants: the corresponding value might be treated as an attribute;
- ◆ For error reporting, we might include positional information:
  - Name of source file;
  - Line/column number;
  - Etc.

14

## Other Input Elements:

- ◆ Other elements that may appear in the input stream include:
  - Whitespace: the space, tab, newline character, etc., which typically have no significance in the language (other than as token separators);
  - Illegal characters, which should not appear in any input;
  - Comments, in various flavors.
- ◆ These are filtered out during lexical analysis, and not passed as tokens to the parser.

15

## Common Comment Types:

- ◆ Nesting brackets:
  - (\* Pascal, Modula-2, ML \*)
  - { Pascal }
  - {- Haskell -}
- ◆ Non-nesting brackets:
  - /\* Prolog, Java, C++, C \*/
- ◆ Single Line:
  - // C++, Java
  - -- occam, Haskell
  - ; Scheme, Lisp
  - % Prolog
  - # csh, bash, sh, make
  - C Fortran
  - REM Basic

16

## Representing Token Streams:

- ◆ We could construct a new data object for each lexeme that we find, and build an array that contains all of the tokens for a given source in order.
- ◆ Different types of token object are needed for different types of token when the number and type of attributes vary.
- ◆ In practice, many compilers do not build token objects, and instead expect tokens to be read *in pieces* and *on demand*.

17

## A simple lexer for MiniJava:

```
// Read the next token.  
int nextToken();
```

Advance to the next lexeme and return a code for the token type.

```
// Returns the token code for the current lexeme.  
int getToken();
```

```
// Returns the text (if any) for the current lexeme.  
String getLexeme();
```

```
// Return the position of the current lexeme.  
Position getPos();
```

18

## Recognizing Identifiers:

Suppose that *c* contains the character at the front of the input stream:

```
if (isIdentifierStart(c)) {  
    do {  
        c = readNextInputChar();  
    } while (c!=EOF &&  
            isIdentifierPart((char)c));  
    return IDENT;  
}
```

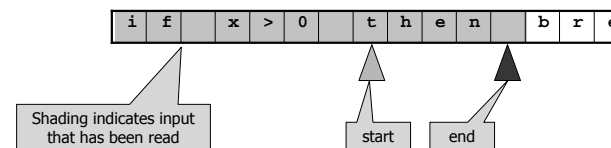
A symbolic constant, defined elsewhere, to represent identifier tokens.

19

## Buffering:

Input characters must be stored in a buffer...

- ◆ Because we often need to store the characters that constitute a lexeme until we find the end:

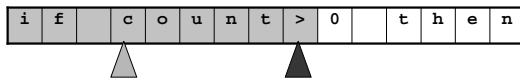


20

## Buffering:

Input characters must be stored in a buffer...

- ◆ Because we might not know that we've reached the end of a token until we've read the following character:



21

## Buffering:

Input characters must be stored in a buffer...

- ◆ Because we might need to look ahead to see what tokens are coming:

Fortran 77:



do 10 c = 1, 10 ⇒ do 10 c = 1, 10

do 10 c = 1.10 ⇒ do10c = 1.10

Now generally considered an example of really bad design!

22

## Impact on Language Design:

- ◆ In some languages, only the first 32 characters of an identifier are significant; string literals cannot have more than 256 characters; etc...  
⇒ Puts an upper bound on the size of a buffer.
- ◆ In most languages, only one or two characters of lookahead are required for lexical analysis.
- ◆ In most languages, whitespace cannot appear in the middle of a token.

23

## Buffering in the MiniJava lexer:

```
BufferedReader src;
int row = 0;
int col = 0;
String line;
int c;

void nextLine() {
    line = src.readLine();
    col = (-1);
    row++;
    nextChar();
}

int nextChar() {
    if (line == null) {
        c = EOF;
    } else if (++col >=
line.length()) {
        c = EOL;
    } else {
        c = line.charAt(col);
    }
    return c;
}
```

24

## Basic nextToken() function:

```
int nextToken() {
    for (;;) {
        lexemeText = null;
        switch (c) {
            case EOF      :      token=ENDINPUT;
            return token;
            case EOL      :      nextLine(); break;
            case ' '      :      nextChar(); break;
            case '/'      :      skipComment(); break;
            case ';'      :      nextChar();
            token=SEMI; return token;
            ...
        }
    }
}
```

25

## Identifying Identifiers:

The current input line also serves as a buffer for the lexeme text:

```
c = line.charAt(col);
if (isIdentifierStart(c)) {
    int start = col;
    do {
        c = line.charAt(++col);
    } while (col < line.length &&
            isIdentifierPart((char)c));
    lexemeText = line.substring(start, col);
    return IDENT;
}
```

If there are lexemes that span multiple lines, then additional buffering must be provided.

26

## Recognizing Identifiers:

### ◆ In Java:

“an identifier is an unlimited-length sequence of Java letters and Java digits, the first of which must be a Java letter. An identifier cannot have the same spelling as a keyword ...”

### ◆ How can we make this pattern more precise?

27

## Recognizing Identifiers:

### ◆ C.f. the concrete implementation:

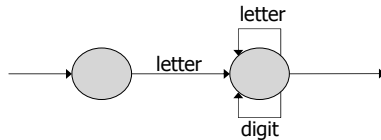
```
c = line.charAt(col);
if (isIdentifierStart(c)) {
    int start = col;
    do {
        c = line.charAt(++col);
    } while (col < line.length &&
            isIdentifierPart((char)c));
    lexemeText = line.substring(start, col);
    return IDENT;
}
```

### ◆ How can we avoid unnecessary details?

28

## Recognizing Identifiers:

- ◆ Idea: describe the implementation using a state machine notation:



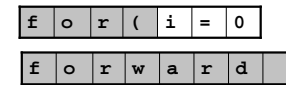
- ◆ If the token for every pattern were described as a state machine, implementation would be much easier
- ◆ But there are some problems lurking...

29

## Maximal Munch:

- ◆ A widely used convention:
  - The rule of the longest lexeme: if there are two different lexemes at the beginning of the input stream, always choose the longest alternative.

- ◆ For example:



- ◆ Another classic:



30

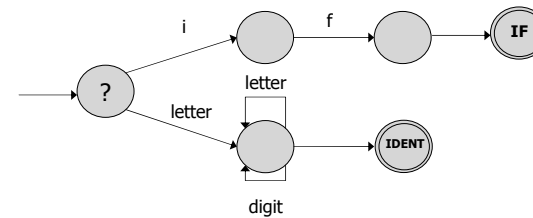
## Implementation:

- ◆ Remember the last state and input position where a valid lexeme had been detected.
- ◆ Continue reading ahead to look for a longer lexeme with the same prefix.
- ◆ If you find a longer one, update the state and position, and continue.
- ◆ If you don't find a longer one, go back to the last valid lexeme.
- ◆ (N.B. Buffers play an important role here).

31

## Non-determinism:

- ◆ The lexeme "if" looks a lot like an identifier:



- ◆ What do we do if the first char is 'i'?
- ◆ What if the first character is 'i', we follow the top branch ... and the next character is 'b'?

32

## Solution 1: Backtracking

- ◆ When faced with multiple alternatives:
  - Explore each alternative in turn.
  - Pick the alternative that leads to the longest lexeme.
- ◆ Again, buffers play a key role.
- ◆ Complex to program and potentially expensive to execute.

33

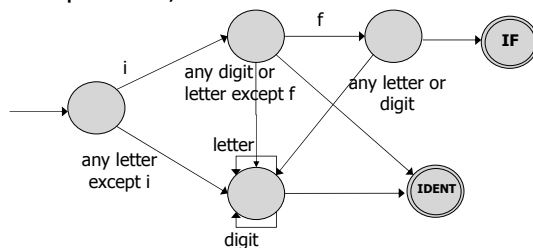
## Solution 2: Postprocessing

- ◆ To begin with, just treat and recognize "if" as a normal identifier.
- ◆ But before we return identifiers as tokens, check them against a built-in table of keywords, and return a different type of token as necessary.
- ◆ Simple, efficient (so long as we can look up entries in the keyword table without too much difficulty) ... but not always applicable.

34

## Solution 3: Delay the Decision!

- ◆ Find an equivalent, deterministic machine:



- ◆ Recognizes the same set of lexemes, without ambiguity.
- ◆ Hard to get the right machine by hand ...

35

## Another Example:

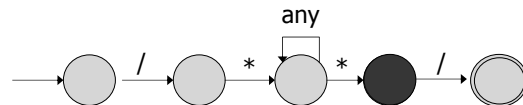
How do we recognize a comment?

- ◆ It begins with /\*
- ◆ It ends with \*/
- ◆ Any characters can appear in between ... (well, almost any characters)

36

## Naïve Description:

- ◆ A simple first attempt to recognize comments, might lead to the following state machine:



- ◆ But this machine can fail if we follow the second \* branch too soon.

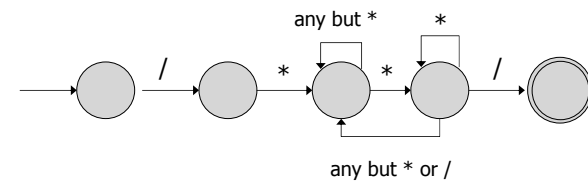
```
/* y = x * z */
          ▲
```

37

## A More Careful Approach:

- ◆ The previous state machine was non-deterministic: there is a choice of distinct successor states for the character '\*', and we can't tell which branch to take without looking ahead.

- ◆ An equivalent, deterministic machine is as follows:



38

## Code to Read a Comment:

```
if (c=='/') { // skip bracketed comment
    nextChar();
    if (c=='*') {
        nextChar();
        for (;;) {
            if (c=='*') {
                do { nextChar(); } while (c=='*');
                if (c=='/') {
                    nextChar();
                    return;
                }
            }
        }
        if (c==EOF) { ... Unterminated comment ... }
        if (c==EOL) nextLine(); else nextChar();
    }
}
```

Further complications: error handling

Further complications: interaction with source input

39

## Handwritten Lexical Analyzers:

- ◆ Doesn't require sophisticated programming.
- ◆ Often requires care to avoid non-determinism, or potentially expensive backtracking.
- ◆ Can be fine tuned for performance and for the language concerned.
- ◆ But it might also be something we would want to automate ...

40

## Can a Machine do Better?

- ◆ It can be hard to write a (correct) lexer by hand ...
- ◆ But that's not surprising: finite state machines are low level ... an 'assembly language of lexical analysis'
- ◆ Can we build a lexical analyzer generator that will take care of all the dirty details, and let the humans work at a higher level?
- ◆ If so, what would its input look like?

41

## The lex Family:

- ◆ lex is a tool for generating C programs to implement lexical analyzers.
- ◆ It can also be used as a quick way of generating simple text processing utilities.
- ◆ lex dates from the mid-seventies and has spawned a family of clones: flex, ML lex, JLex, JFlex, etc...
- ◆ Lex is based on ideas from the theory of formal languages and automata ...

42

## Next lecture

- ◆ On Monday the 23rd.
- ◆ We continue with lexers, in particular with lexer generators.
- ◆ Still on chapter 2.
- ◆ Also deadline for the first homework assignment!

43