# CS 321: Languages and Compiler Design I

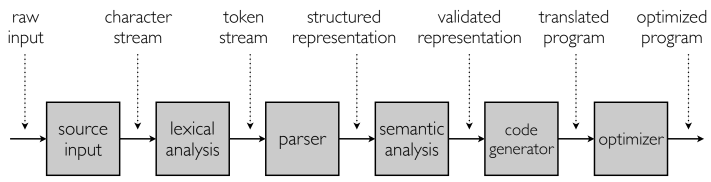
Mark P Jones, Portland State University

Winter 2014

Week 2: Lexical Analysis

## Rough plan for the rest of the term ...

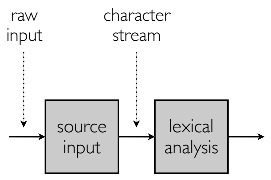
## Through the phases ...



- Compilers are often structured as a pipeline of separate phases
- During the next few weeks, we'll follow the same structure, taking a more detailed look at each of the phases in turn
- Starting with ...

## Source input

## Implementing source input



Source input issues include:

- Getting characters into memory (your OS can help!)
- Dealing with character set encodings: ASCII, EBCDIC, ISO 8859_1, UTF, etc.
- In Java, ignore a trailing ^Z at the end of a file
- In Haskell, process "literate source files"
- Expand tabs if indentation is significant
- Process Unicode escapes. For example: \u2297 ⇒ ⊗

## Introduction to syntax analysis

$$\text{language} = \begin{cases} \text{syntax} = \begin{cases} \text{concrete} \\ \text{abstract} \end{cases} \\ \text{semantics} = \begin{cases} \text{static} \\ \text{dynamic} \end{cases} \end{cases}$$

concrete syntax: the representation of a program text in its source form as a sequence of bits/bytes/characters/lines

$$\text{language} = \begin{cases} \text{syntax} = \begin{cases} \text{concrete} \\ \text{abstract} \end{cases} \\ \text{semantics} = \begin{cases} \text{static} \\ \text{dynamic} \end{cases} \end{cases}$$

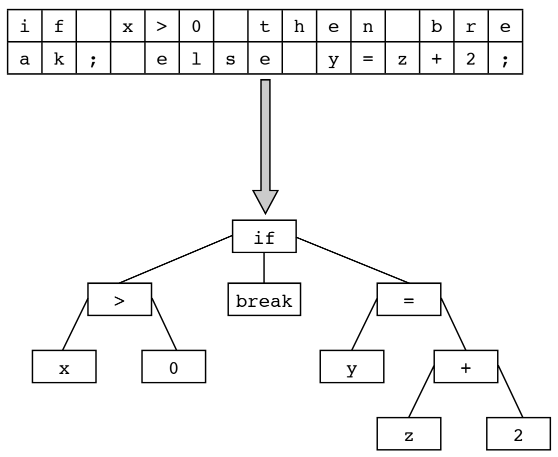
abstract syntax: the representation of a program structure, independent of written form

## Syntax analysis

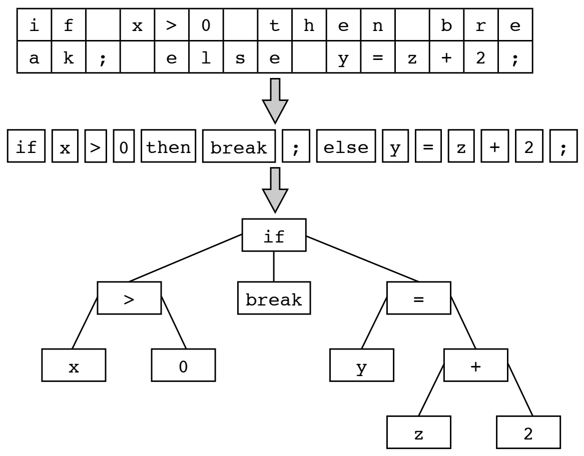
$$\text{language} = \begin{cases} \text{syntax} = \begin{cases} \text{concrete} \\ \Downarrow \\ \text{abstract} \end{cases} \\ \text{semantics} = \begin{cases} \text{static} \\ \text{dynamic} \end{cases} \end{cases}$$

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

## Syntax analysis



## Syntax analysis



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

## Combined lexical analysis and parsing?

- It isn't technically necessary to separate lexical analysis and 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 in the lexer

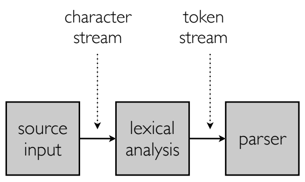
  Good tool support (e.g., lex, yacc)
- Modern language specifications often separate lexical and grammatical syntax

## Basics of lexical analysis

## Lexical analysis

- Lexical analysis is carried out by a

  lexical analyzer
  lexer
  scanner
  } synonyms



- Goal: to recognize and identify the sequence of tokens represented by a the characters in a program text
- The definition of tokens (or "lexical structure") is an important part of many language specifications

## Basic terminology

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

  "`0.0`", "`3.14`", "`1e-97d`"
  "`true`", "`false`"
  "`if`", "`then`", "`int`"
  "`String`", "`main`"

## Basic terminology

- Token: a name for a set of lexemes (a description of what each lexeme represents)
- Examples in Java include:

| | | |
|---|---|---|
| "`0.0`", "`3.14`", "`1e-97d`" | are double literals | tokens |
| "`true`", "`false`" | are boolean literals | |
| "`if`", "`then`", "`int`" | are keywords | |
| "`String`", "`main`" | are identifiers | |

- Tokens/lexemes are normally chosen so that each lexeme has just one token type

## Basic terminology

- Pattern: a description of the way that lexemes are written
- In the Java language specification:

  "an identifier is an unlimited-length sequence of Java letters and Java digits. An identifier cannot have the same spelling as a keyword ..."

- We'll see how to make this kind of thing more precise soon by using regular expressions as patterns

## Common token types

- Keywords, symbols, punctuation

  `for`, `if`, `then`, `<=`, `+`, `(`, `;`, `.`, ...

- Literals and constants

  integers
  floating point numbers
  characters
  strings
  etc.

- Identifiers

  `String`, `main`, `awt`, `handler`, `Exception`, `i`, ...

## Token attributes

- Some tokens may have associated attributes
- In some cases, the lexeme itself might be used as the attribute (e.g., the text of an identifier name)
- The value represented by a literal/constant might be treated as an attribute
- For error reporting, we might include positional information
  - name of source file
  - line/column number
  - etc...
- Attributes capture properties of the lexeme by itself
  - e.g., the initial value of a variable is *not* a token attribute

## Other input elements

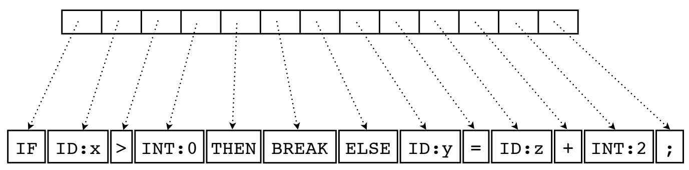
- Other elements that may appear in the input stream (but are not tokens) include:
  - Whitespace: the space, tab, newline character, etc., which typically have no significance in the language other than to separate tokens
  - Comments, in various forms
  - Illegal characters
- These are filtered out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

## Common comment types

- Single line
```
// C++, Java
-- occam, Haskell
;  Scheme, Lisp
#  csh, bash, sh, make
C  fortran
```
- Non-nesting brackets
```
/* Prolog, C, C++, Java */
```
- Nesting brackets
```
(* Pascal, Modula-2, ML *)
{- Haskell -}
```

## Representing token streams

- In principle, we could construct an array containing a separate data object for each lexeme in the input stream



- 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, but instead expect tokens to be read <u>in pieces</u>, and <u>on demand</u>

## Lexical analysis for mini

```
/** Read the next token and return the
 *  corresponding integer code.
 */
int nextToken();

/** Returns the code for the current token.
 */
int getToken() {
    return token;
}

/** Returns the text (if any) for the current lexeme.
 */
String getLexeme() {
    return lexemeText;
}

/** Return a position describing where the current token
 *  was found.
 */
Position getPos();
```

advance to the next lexeme and return a code for the token type

read current token code

read current lexeme text

read current position

## Recognizing identifiers

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

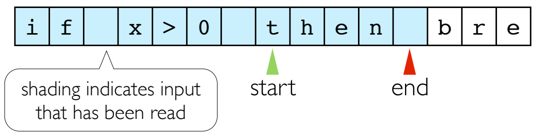
```
if (isIdentifierStart(c)) {

    do {
        c = readNextInputChar();
    } while (c!=EOF &&
             isIdentifierPart(c));

    return token=IDENT;
}
```

a symbolic constant, defined elsewhere, to represent identifier tokens

## Buffering

input characters must be stored in a buffer ...

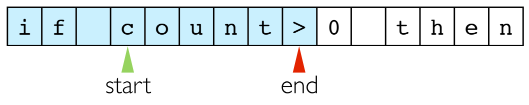
- because we often need to store the characters that constitute a lexeme until we find the end:



## 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 read the following character:

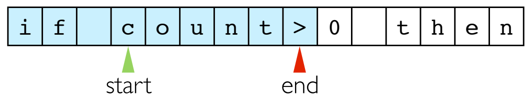


## Buffering

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` $\Longrightarrow$ $\boxed{\texttt{do}}\boxed{\texttt{10}}\boxed{\texttt{c}}\boxed{\texttt{=}}\boxed{\texttt{1}}\boxed{\texttt{,}}\boxed{\texttt{10}}$

`do 10 c = 1.10` $\Longrightarrow$ $\boxed{\texttt{do10c}}\boxed{\texttt{=}}\boxed{\texttt{1.10}}$

## 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.

  Why? Because these put a bound on the size of the buffer that is needed

- In modern language designs, typically only one or two characters of lookahead are required for lexical analysis

- In modern language designs, whitespace is not typically allowed in the middle of a lexeme

## Buffering in the mini compiler's lexer

```
Source source;
String line;
int col = (-1);
int c;

void nextLine() {
    line = source.readLine();
    col  = (-1);
    nextChar();
}

int nextChar() {
    if (line==null) {
        c   = EOF;
        col = 0;  // EOF is always at column 0
    } else if (++col>=line.length()) {
        c   = EOL;
    } else {
        c   = line.charAt(col);
    }
    return c;
}
```

a buffer containing a single line of text

current position in the buffer

read the next buffer from the source

read the next character from the current line buffer

## Buffering in the mini compiler's lexer

```
/** Read the next token and return the corresponding integer code.
 */
public int nextToken() {
    for (;;) {
        skipWhitespace();
        lexemeText = null;
        switch (c) {
            case EOF   : return token=ENDINPUT;

            // Separators:
            case '('   : nextChar();
                         return token='(';
            case ')'   : nextChar();
                         return token=')';
            case '{'   : nextChar();
                         return token='{';
            case '}'   : nextChar();
                         return token='}';
            case ';'   : nextChar();
                         return token=';';
            case ','   : nextChar();
                         return token=',';
            ...
```

variable to hold text for current lexeme

simple, single character tokens

## Continued ...

```
...
case '='  : nextChar();
            if (c=='=') {
                nextChar();
                return token=EQEQ;
            } else {
                return token='=';
            }
...
case '&'  : nextChar();
            if (c=='&') {
                nextChar();
                return token=CAND;
            } else {
                return token='&';
            }
...
```

could be either "=" or "==" ... which is it?

could be either "&" or "&&" ... which is it?

## Continued ...

```
...
case '/'  : nextChar();
            if (c=='/') {
                skipOneLineComment();
            } else if (c=='*') {
                skipBracketComment();
            } else {
                return token = '/';
            }
            break;

...

default   : if (Character.isJavaIdentifierStart((char)c)) {
                return identifier();
            } else if (Character.digit((char)c, 10)>=0) {
                return number();
            } else {
                illegalCharacter();
                nextChar();
            }
        }
    }
}
```

could be either :
"//..." (single line comment)
"/*...*/" (bracketed comment)
"/" (division)
which is it?

variable length tokens: identifiers and integer literals

## Recognizing identifiers (reprise)

If we only want to detect that an identifier was detected ...

```
if (isIdentifierStart(c)) {

    do {
        c = getChar();
    } while (c!= EOF &&
               isIdentifierPart(c));

    return token=IDENT;
}
```

How do we modify this to capture *which* identifier it was?

## 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(c));
    lexemeText = line.substring(start, col);
    return token=IDENT;
}
```

(If there are lexemes that can span multiple lines, then additional buffering would be required.)

## Reflections

- We've seen several simple programming patterns here
  - recognizing single characters
  - recognizing distinct operators with a common prefix
  - recognizing variable length lexemes
  - ...
- None of these is particularly difficult (or interesting)
- One conclusion: this is a straightforward way to construct a lexer; a good approach for a quick project
- Another take: somewhat ad-hoc and error prone; maybe not the best approach from a software engineering perspective

## Summary

- Lexical analysis converts character streams into token streams
- Buffering plays an important role
- But the techniques we've seen so far seem a little ad hoc ...

## Lexical analysis using finite automata

## Recognizing identifiers

- In the Java language specification:

  "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 description more precise?

## Recognizing identifiers

- A concrete implementation:

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

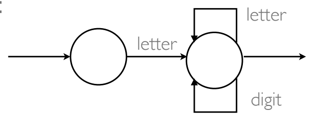
- How can we avoid unnecessary details?

## Recognizing identifiers

- In more abstract terms:

  The pattern can be described by a regular expression: letter (letter|digit)*

  The recognition of an identifier can be described by a finite automaton:



- But this doesn't specify:

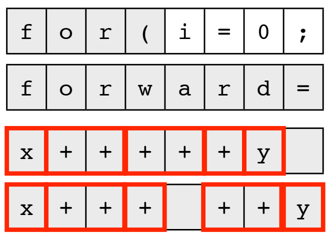
  How letter and digit are defined
  When does an identifier stop?
  What if we get stuck?

## Maximal munch / longest lexeme

- A widely used convention:

  if there are two (or more) different lexemes that can be read at a given point in the input stream, always choose the longest alternative

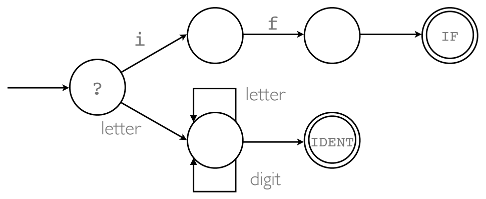
- For example:
- Another classic:



shading indicates how much of the input was read before the end of the token was found

## Non-determinism

- The lexeme "`if`" looks a lot like an identifier



- What do we do if the first input character is "`i`"?
- What if the first character is "`i`", we follow the top branch ... and then the next character is "`b`"?

## Solution 1: backtracking

- When faced with multiple alternatives:

  Explore each alternative in turn

  Pick the alternative that leads to the longest lexeme

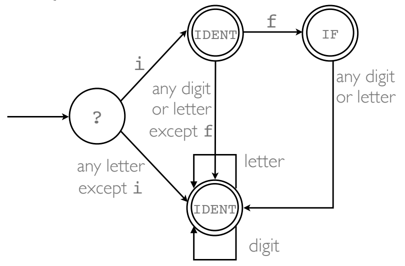
- Again, buffers play a key role, recording past input in case we need to go back
- But, in general, this approach is complex to program and potentially expensive to execute

## 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 and efficient (so long as we can look up entries in the keyword table without too much difficulty), but not always applicable

## Solution 3: delay the decision!

- Find an equivalent machine without the conflict



- Recognizes the same set of lexemes, but without the ambiguity in how they are categorized as tokens
- Fiddly, hard to get right by hand

## Another example

How do we recognize a comment?

it begins with /*

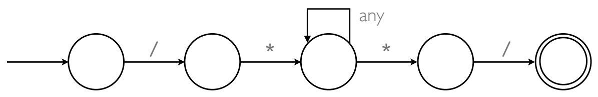
it ends with */

any characters can appear in between

... (well, almost any characters)

## Naive description

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

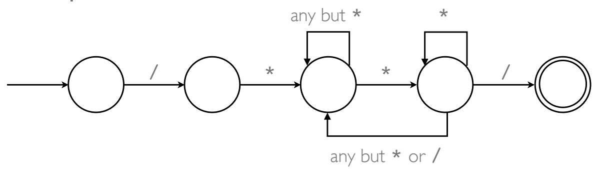


- But this machine can get stuck if we follow the second * branch too soon

```
/* y = x * z */
```

## A more careful approach

- The previous state machine is 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



## 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();
        }
    }
}
```

complication: interaction with error handling

complication: interaction with source input

## 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 ...

## 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, much like "an assembly language for lexical analysis"
- Can we build a lexical analyzer generator that will take care of all the dirty details (correctly) and let compiler developers work at a higher-level?
- If so, what would the input look like?

## The lex family of lexer generators

## The lex family

- `lex` is a tool for generating C programs to implement lexical analyzers

  input is a set of (regular expression + associated action) rules

  output is C source code for a lexer
- It can also be used a a quick way to generate simple text processing utilities
- `lex` dates from the mid-seventies and has spawned a family of clones: flex, JLex, JFlex, Alex, ML lex, etc...
- `lex` is based on ideas from the theory of formal languages and automata; remember CS311? Coming up in Lab 2!

## Fragments from mini.jflex

```
"("             { return '('; }
")"             { return ')'; }
"{"             { return '{'; }
"}"             { return '}'; }
";"             { return ';'; }
...
"="             { return '='; }
"=="            { return EQL; }
">"             { return '>'; }
">="            { return GTE; }
...
"while"         { return WHILE; }
"if"            { return IF; }
"else"          { return ELSE; }
"print"         { return PRINT; }
"return"        { return RETURN; }
"new"           { return NEW; }
...
{Letter}({Letter}|{Digit})*
                { semantic = new Id(yytext());     return IDENT; }

[0-9]+          { semantic = new IntLit(yytext()); return INTLIT; }

{ \t\n}         { /* ignore whitespace */ }
```

execute the associated action when we see input matching the left hand pattern

simple things should be easy

let the lexer generator figure out how to deal with ambiguity!

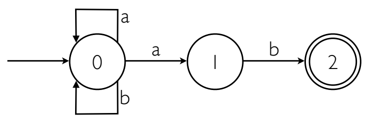
## Inside a lexer generator

The internals of a typical lexer generator typically involve multiple steps

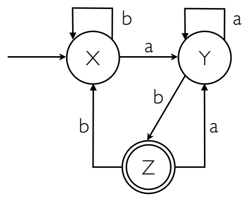
$$\begin{tikzcd}
\text{regular expressions} \arrow[d] \\
\text{non-deterministic finite automata (NFA)} \arrow[d] \\
\text{deterministic finite automata (DFA)} \arrow[d] \\
\text{executable code / tables}
\end{tikzcd}$$

## Example

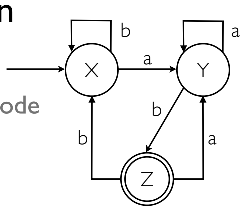
- Given a regular expression (a|b)*ab, we can build a corresponding NFA:



- And then we can derive a suitable DFA:



## From DFA to implementation



So you've built a DFA ... what next?

- Option 1: generate custom executable code

```
stateX:  switch (getchar()) {
           case 'a' : goto stateY;
           case 'b' : goto stateX;
           default  : goto error;
         }
stateY:  ...
```

- Option 2: encode the machine in a table and interpret the transitions in the table during matching

| State | 'a' | 'b' | 'c' | accept? |
|---|---|---|---|---|
| X | Y | X | err | no |
| Y | Y | Z | err | no |
| Z | Y | X | err | yes |

## Variations on the theme

- There are lots of lex-like tools, each catering to particular
  - programming languages
  - operating systems/environments
- Most of them have more features that I have described on the previous slides; read the manual!
- They vary in minor details of syntax, etc., but the lex heritage is usually very clear
- For work in C/C++, I recommend flex
- For standalone lexer work in Java, I recommend JFlex. We'll also be using JavaCC for integrated lexing/parsing.

## Handwritten or machine generated?

- A mildly controversial topic!
- Issues include:
  - How efficient is the generated code?
  - How easily does it interface to other code?
  - How natural is the input? (If the language you are compiling has some awkward features, the lexers produced by a tool might need some massaging to "do the right thing")
  - How good are the error messages?

## Summary

- Regular expressions provide a high-level language for describing lexemes
- lex and family are useful tools for writing text processing utilities ... as well as lexers
- lex works by mapping regular expressions to NFAs, which are converted into DFAs, which are used to produce executable code
  - key benefit: automates the task of eliminating non-determinism
- A handwritten or a machine generated lexer? The choice is yours!