CSc 453

Compilers and Systems Software

3 : Lexical Analysis I

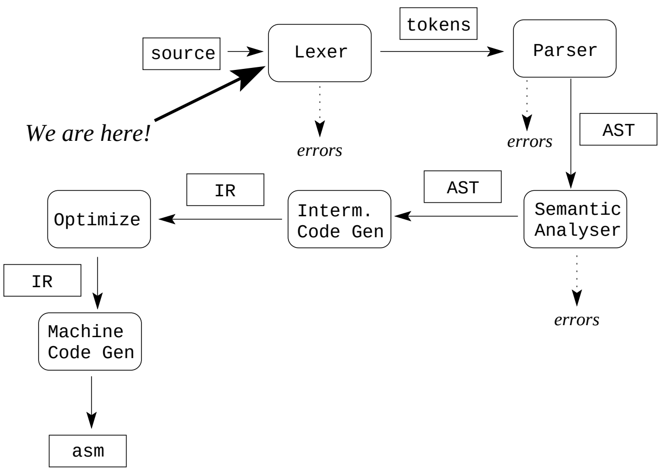
Department of Computer Science
University of Arizona

collberg@gmail.com

Christian Collberg

September 27, 2009

Compiler Phases



Compiler Phases – Lexical analysis

- The lexer reads the source file and divides the text into lexical units (tokens), such as:

  Reserved words **BEGIN**, **IF**,...
  identifiers `x`, `StringTokenizer`,...
  special characters $+, *, -$, `^`,...
  numbers 30, 3.14,...
  comments `(* text *)`,
  strings `"text"`.

- Lexical errors (such as 'illegal character', 'undelimited character string', 'comment without end') are reported.

Lexical Analysis of English

- The sentence

```
"The boy's cowbell won't play."
```

would be translated to the list of tokens

```
the, boy+possessive, cowbell, will, not, play
```

Lexical Analysis of Java

- The sentence

```
"x = 3.14 * (9.0+y);"
```

would be translated to the list of tokens

```
<ID,x>, EQ, <FLOAT,3.14>, STAR, LPAREN,
<FLOAT,9.0>, PLUS, <ID,y>, RPAREN, SEMICOLON
```

Example – Lexical Analysis

- Break up the source code (a text file) and into tokens.

Source Code	Stream of Tokens
PROCEDURE Foo ();	PROCEDURE, <id,Foo>, LPAR, RPAR, SC,
VAR i : INTEGER;	VAR, <id,i>, COLON, <id,INTEGER>,SC,
BEGIN	BEGIN, <id,i>,CEQ,<int,1>,SC,
i := 1;	WHILE, <id,i>, LT, <int,20>,DO,
WHILE i < 20 DO	PRINT, <id,i>, MUL, <int,2>, SC,
PRINT i * 2;	<id,i>, CEQ, <id,i>, MUL, <int,2>, PLUS,
i := i * 2 + 1;	<int,1>, SC, ENDDO, SC, END, <id,Foo>, SC
ENDDO;	
END Foo;	

Problems

Free vs. Fixed Format

- Most languages are free format, i.e. it does not matter where on a line of text a certain token occurs.
- FORTRAN (at least early versions) uses a fixed format where the first 6 characters on the input line is a label, and the last characters (columns 72-80) a comment.
- A "C" in the first column indicates a comment line.
- Any character in column 6 indicates a continuation line.

```
C     Compute the determinant:
       det = a(1,1) * a(2,2) * a(3,3) + a(1,2) * a(2,3) * a(3,1)
      &    + a(2,1) * a(3,2) * a(1,3) - a(3,1) * a(2,2) * a(1,3)
      &    - a(2,1) * a(1,2) * a(3,3) - a(1,1) * a(3,2) * a(2,3)
```

Free vs. Fixed Format. . .

- Python, Occam, and some functional languages use indentation to indicate nesting:

```
def quicksort(list, start, end):
    if start < end:
        split = partition(list, start, end)
        quicksort(list, start, split-1)
        quicksort(list, split+1, end)
    else:
        return
```

Whitespace

In most modern languages whitespace (blanks and tabs) are significant. FORTRAN and Algol-68 are different: whitespace may be added anywhere to improve readability. The FORTRAN statement

```
DO 5 I = 1.25
```

is an assignment statement, meaning the same as:

```
DO5I = 1.25
```

This statement, on the other hand, is a loop statement:

```
DO 5 I = 1,25
   ...
5 CONTINUE
```

Whitespace...

An error in a single FORTRAN statement resulted in the loss of the first American probe to Venus (the Mariner I).

```
    ....
    DO 5 K = 1. 3
    T(K) = W0
    Z = 1.0/(X**2)*B1**2+3.0977E-4*B0**2
    D(K) = 3.076E-2*2.0*(1.0/X*B0*B1+3.0977E-4*
   *(B0**2-X*B0*B1))/Z
    E(K) = H**2*93.2943*W0/SIN(W0)*Z
    H = D(K)-E(K)
  5 CONTINUE
```

This is now considered an urban legend.

Buffering

- If done incorrectly, lexical analysis can be an expensive phase of the compiler – It is the only phase which actually considers each and every character of the program.
- It is, for example, crucial not to read one character at a time from the input file. Rather, a large block of the input text file must be read and but into a buffer. This buffer is then used to provide the lexer with character.
- Sometimes the lexer may also need to look ahead at characters to come before deciding on what token appears next in the text. The buffer is useful in such circumstances also.

Keywords

- Most languages have reserved keywords, which means that these words may not be redefined by the user.
- PL/I does not reserve keywords which makes it difficult for the lexer to distinguish between user-defined identifiers and keywords:

```
IF THEN THEN THEN = ELSE; ELSE ELSE = THEN;
```

Error handling

- What do we do when an error is encountered during lexical analysis?

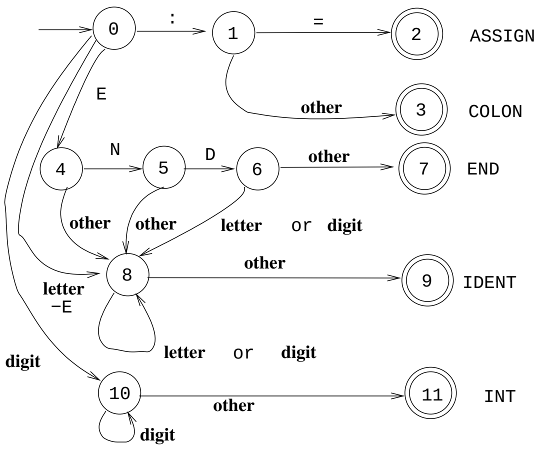
Panic	Skip characters until a well-formed token is found.
Replace	Replace an incorrect character.
Delete	Delete an incorrect character.
Insert	Insert a missing character.
Transpose	Switch two characters.

Communication

- The Lexer may communicate with the parser in many different ways.
- Lexical analysis might, for example, run as a special pass writing the tokens on a temporary file which is read by the parser.
- Or – and this is probably the most common situation – the parser makes a procedure call to the lexer whenever a token is needed.
- The Lexer and the Parser could also run as two concurrent processes communicating over a pipe.

Transition Diagrams

Transition Diagrams



```
TYPE TokenType = (Assign, End, ...);
VAR  s         : (State0, State1, ...);
     c         : CHAR;
PROCEDURE GetToken () : TokenType;
   CASE s OF
     State0 : c := NextChar();
              CASE c OF
                 ":"        : s := State1|
                 "E"        : s := State4|
                 "0" .. "9" : s := State10|
                 ELSE       : s := State8
              END|
     State1 : c := NextChar();
              IF c="=" THEN s := State2
              ELSE          s := State3 END|
```

```
State2 : RETURN Assign|
State3 : PutChar(c); RETURN Colon|
State4 : c := NextChar();
         IF c = "N" THEN s := State5
         ELSE            s := State8 END|
State5 : c := NextChar();
         IF c = "D" THEN s := State6
         ELSE            s := State8 END|
State6 : c := NextChar();
         IF IsLetterOrDigit(c)
         THEN s := State8
         ELSE s := State7 END;|
State7 : PutChar(c); RETURN End|
```

```
    State8 : c := NextChar();
             IF NOT IsLetterOrDigit(c)
             THEN s := State9 END|
    State9 : PutChar(c); RETURN Ident|
    State10 : c := NextChar();
              IF NOT IsDigit(c)
              THEN s := State11 END|
    State11 : PutChar(c); RETURN Int|
  END;
END GetToken;
```

Regular Grammars and Lexical Analysis

Regular Grammars

- A grammar is regular if all rules are of the form

$$
\begin{aligned}
A &\rightarrow aB \\
A &\rightarrow a
\end{aligned}
$$

- By convention, the symbols $A, B, C, \ldots$ are non-terminals, $a, b, c, \ldots$ are terminals, and $\alpha, \beta, \gamma, \ldots$ are strings of symbols.
- Regular grammars are used to describe the lexical structure of programs, i.e. what tokens look like.

Regular Grammars...

- The following grammar describes C identifiers:

$$\begin{array}{rcl} \text{id} & \rightarrow & \underline{\texttt{letter}} \mid \underline{\texttt{letter}}\ \text{S} \\ \text{S} & \rightarrow & \underline{\texttt{letter}} \mid \underline{\texttt{letter}}\ \text{S} \\ \text{S} & \rightarrow & \underline{\texttt{digit}} \mid \underline{\texttt{digit}}\ \text{S} \end{array}$$

$$\begin{array}{rcl} \underline{\texttt{digit}} & \rightarrow & 0 \mid 1 \mid \cdots \mid 9 \\ \underline{\texttt{letter}} & \rightarrow & A \mid \cdots \mid Z \mid \\ & & a \mid \cdots \mid z \end{array}$$

- Here's a derivation of the identifier `cow5`:

$$\text{id} \Rightarrow \underline{\texttt{letter}}\ \text{S} \Rightarrow c\ \text{S} \Rightarrow c\ \underline{\texttt{letter}}\ \text{S} \Rightarrow c\ o\ \text{S} \Rightarrow$$

$$c\ o\ \underline{\texttt{letter}}\ \text{S} \Rightarrow c\ o\ w\ \text{S} \Rightarrow c\ o\ w\ \underline{\texttt{digit}} \Rightarrow \underline{cow5}$$

Regular Grammars. . .

- This is a grammar for floating point numbers. As written, it is not quite regular: We treat digit as a terminal.

$$
\begin{aligned}
\text{float} &\longrightarrow \underline{+}\ \text{float1} \mid \underline{-}\ \text{float1} \mid \text{float1} \\
\text{float1} &\longrightarrow \underline{\texttt{digit}}\ \text{float1} \mid \text{float2} \\
\text{float2} &\longrightarrow \underline{.}\ \text{float3} \\
\text{float3} &\longrightarrow \underline{\texttt{digit}}\ \text{float4} \mid \underline{\texttt{digit}} \\
\text{float4} &\longrightarrow \underline{\texttt{digit}}\ \text{float4} \mid \text{float5} \\
\text{float5} &\longrightarrow \underline{\texttt{E}}\ \text{float6} \\
\text{float6} &\longrightarrow \underline{+}\ \text{float7} \mid \underline{-}\ \text{float7} \mid \text{float7} \\
\text{float7} &\longrightarrow \underline{\texttt{digit}}\ \text{float7} \mid \underline{\texttt{digit}}
\end{aligned}
$$

Use the grammar on the previous slide to derive `0.5E+7`.

$$
\begin{aligned}
\textsf{float} &\Rightarrow \textsf{float1} \Rightarrow \underline{\texttt{digit}}\ \textsf{float1} \\
&\Rightarrow 0\ \textsf{float1} \\
&\Rightarrow 0\ \textsf{float2} \\
&\Rightarrow 0\ .\ \textsf{float3} \\
&\Rightarrow 0\ .\ \underline{\texttt{digit}}\ \textsf{float4} \\
&\Rightarrow 0\ .\ 5\ \textsf{float4} \\
&\Rightarrow 0\ .\ 5\ \textsf{float5} \\
&\Rightarrow 0\ .\ 5\ \underline{\texttt{E}}\ \textsf{float6} \\
&\Rightarrow 0\ .\ 5\ \underline{\texttt{E}}\ \underline{+}\ \textsf{float7} \Rightarrow 0\ .\ 5\ \underline{\texttt{E}}\ \underline{+}\ 7
\end{aligned}
$$

Regular Expressions

Regular Expressions

Regular expressions (REs) have the same expressive power as regular grammars. An RE for FP numbers:

$$(\backslash + \mid \backslash -)?\texttt{digit} * \backslash.\texttt{digit} + (\texttt{E}(\backslash + \mid \backslash -)?\texttt{digit}+)?$$

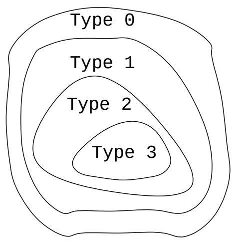
RE	Matches
`character`	The character.
$e_1 \mid e_2$	S, if S is matched by e_1 or e_2.
$e_1\ e_2$	S_1S_2, if e_1 matches S_1 and e_2 matches S_2.
$e+$	One or more S if S is matched by e.
$e*$	Zero or more S if S is matched by e.
$e?$	Zero or one S if S is matched by e.
(e)	S, if S is matched by e.
$\backslash e$	S, if S is matched by e.

Regular Expression Examples

Expression	Matches
a	‘a‘.
$[a-z]$	‘a‘, ‘b‘, . . ., ‘z‘.
$[a-zA-Z0-9]$	‘a‘, ‘b‘, . . ., ‘z‘, ‘A‘, ‘B‘, . . ., ‘Z‘, ‘0‘, ‘1‘, . . ., ‘9‘, .
$[a-zA-Z0-9]*$	Zero or more letters or digits.
$(a \mid b+)?$	‘‘, ‘a‘, ‘b‘, ‘bb‘, ‘bbb‘, ...
$(a \mid b+)?(cd)*$	‘‘, ‘a‘, ‘b‘, ‘bb‘, ‘bbb‘, . . ., ‘acd‘, ‘bcd‘, ‘cdcd‘, ...

The Chomsky Hierarchy

TYPE	GRAMMAR	PSR
0	Unrestricted	$\alpha \longrightarrow \beta$
1	Context Sensitive	$\alpha \longrightarrow \beta$, $\mid \alpha \mid \leq \mid \beta \mid$
2	Context Free	$A \longrightarrow \beta$
3	Regular	$A \longrightarrow a\beta$ $A \longrightarrow a$



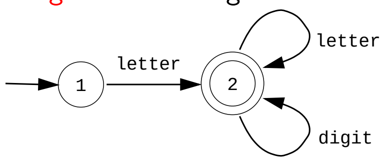
The Chomsky Hierarchy...

- Regular languages are less powerful than context free languages.
- Languages are organized in the Chomsky Hierarchy according to their generative power.
- Type 3 languages are more restrictive (can describe simpler languages than) type 2 languages.
- Type 3 languages can be parsed in linear time, type 2 languages in cubic time.
- Programming languages are in between type 2 and 3.
- Two natural languages (Swiss German and Bambara) are known not to be context free.

DFA

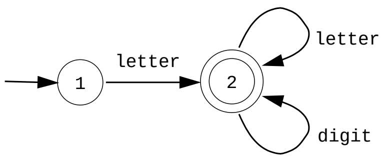
Finite Automata

- Here's a transition diagram describing Pascal identifiers:



- Circles represent states. They represent how much of the input string we have processed.
- Arrows represent transitions from one state to the next, when the character labeling the arrow is matched.
- State 1 is the start state.
- Accepting states are represented by double circles.

Finite Automata...



- Parsing a string of characters using this transition diagram can be indicated by listing the states and transitions used:

$$\longrightarrow 1 \xrightarrow{t} 2 \xrightarrow{m} 2 \xrightarrow{p} 2 \xrightarrow{8} 2$$

- This shows that the string of characters "`tmp8`" form a legal Pascal identifier.

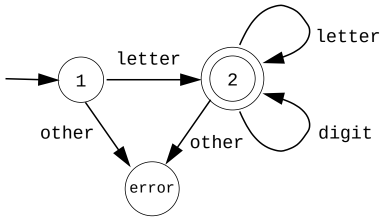
DFA

- A Deterministic Finite Automaton M consists of
  - An alphabet Σ,
  - A set of states S,
  - A transition function $T : S \times \Sigma \longrightarrow S$,
  - A start state $s_0 \in S$,
  - A set of accepting states $A \subset S$.
- T records the transitions between states, depending on input:

$$x \xrightarrow{c} y \qquad T(x, c) = y$$

DFA Error States

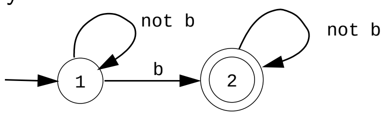
- The transition function $T : S \times \Sigma \to S$ is a function. Hence $T(s, c)$ must be defined for every state s and character c.
- But we have ignored any erroneous input. We should have said



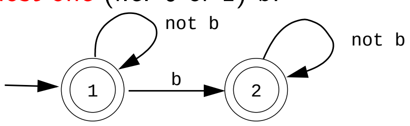
but this would be tedious. Instead, we normally assume that these error transitions always exist.

Examples

- Strings with exactly one `b`:

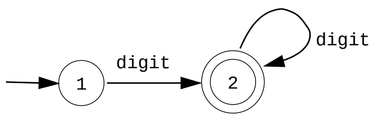


- Strings with at most one (i.e. 0 or 1) `b`:

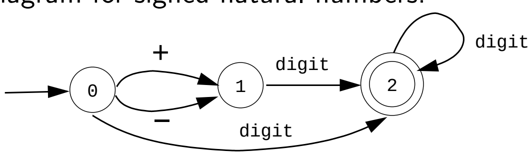


Floating Point Literals

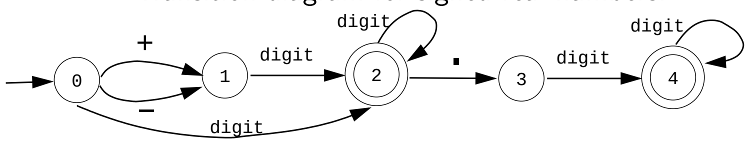
- Transition diagram for natural numbers:



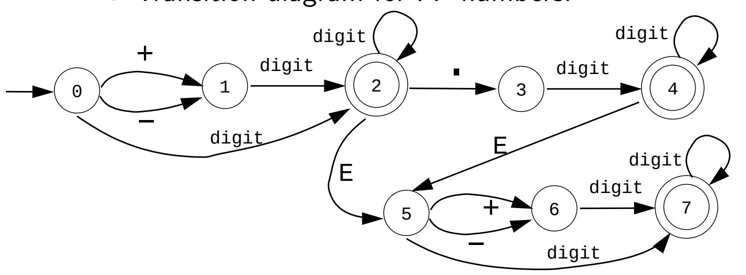
- Transition diagram for signed natural numbers:



- Transition diagram for signed real numbers:



- Transition diagram for FP numbers:

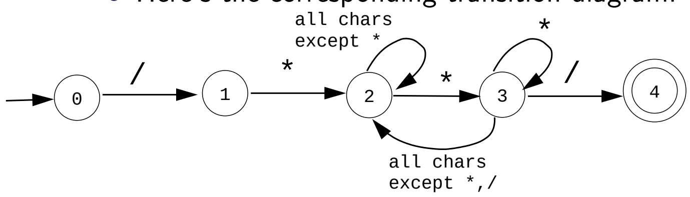


C Comments

- C comments are of the form

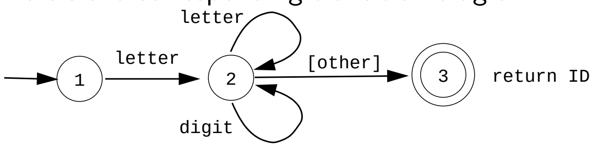
  `/* ... (no */s) ... */`

- Here's the corresponding transition diagram:



Lookahead

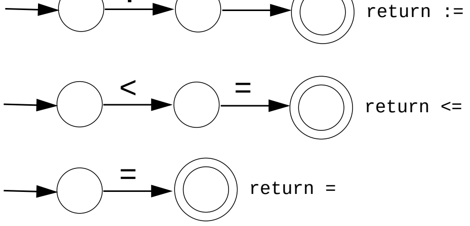
- The end of an identifier is reached when the next character is not a letter or digit.
- The string `"tmp8*hi;"` has two identifiers, terminated by `"*"` and `";"`, respectively.
- Here's the corresponding transition diagram:



- `[other]` means that we're expecting some other character (not letter or digit) as lookahead.

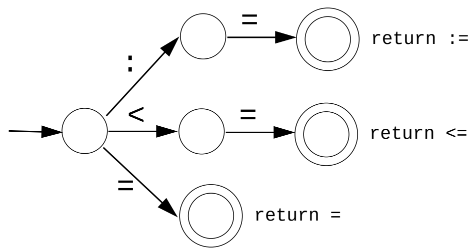
Towards an NFA

- Here are transition diagrams for recognizing :=, <=, and =:

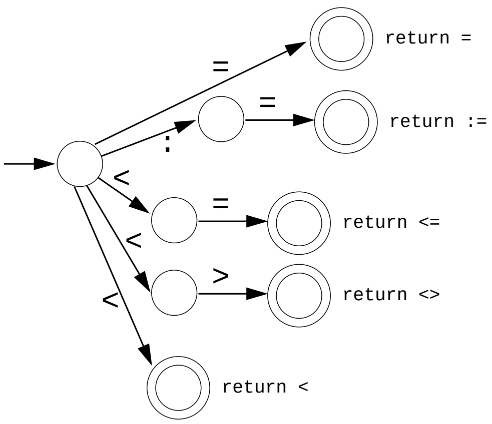


Towards an NFA...

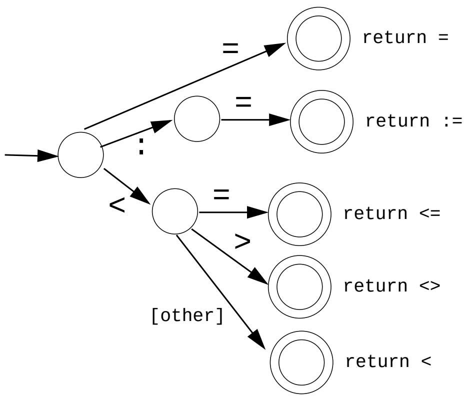
- But, we'd like just one start state, since, at any time during parsing, any token could occur:



- What if two tokens start with the same character? Note that this is not a DFA since there are three transitions on the same character:

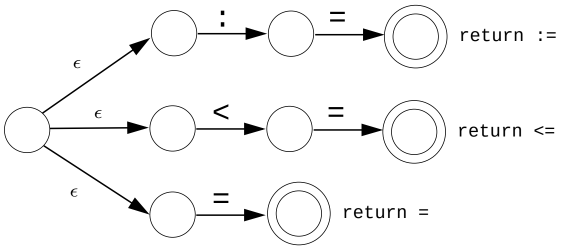


- We can break out the offending character:



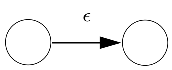
Towards an NFA...

- But, this factoring of states becomes tedious. Instead we can construct a Nondeterministic Finite Automaton (NFA), by adding ϵ-transitions:



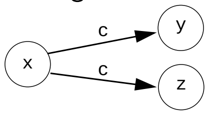
ϵ-Transitions

- An ϵ-transition occurs without consulting the input and without consuming any characters:



NFA

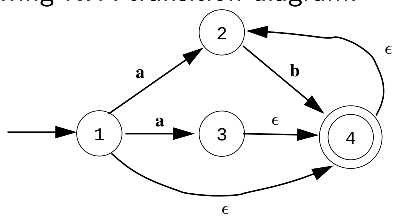
- A Nondeterministic Finite Automaton M consists of
  - An alphabet Σ,
  - A set of states S,
  - A transition function $T : S \times (\Sigma \cup \{\epsilon\}) \rightarrow \mathbb{P}(S)$,
  - A start state $s_0 \in S$,
  - A set of accepting states $A \subset S$.
- $\mathbb{P}(S)$ is the power-set of S, the set of all subsets of S.
- On any transition, we can go to a set of states:



$T(x, c) = \{y, z\}$

NFA Example

- Consider the following NFA transition diagram:



- abb is accepted by these moves: $\rightarrow 1 \xrightarrow{a} 2 \xrightarrow{b} 4 \xrightarrow{\epsilon} 2 \xrightarrow{b} 4$
- or by these moves: $\rightarrow 1 \xrightarrow{a} 3 \xrightarrow{\epsilon} 4 \xrightarrow{\epsilon} 2 \xrightarrow{b} 4 \xrightarrow{\epsilon} 2 \xrightarrow{b} 4$
- The NFA accepts `ab+|ab*|b*`, or, simpler, `(a|`ϵ`)b*`.

Summary

Readings and References

- Read Louden, pp. 31–80.
- Or, read the Dragon book, pp. 83–140.
- The Python example is taken from `http://www.hetland.org/python/quicksort.html`.
- The FORTRAN example is taken from `http://www.math.hawaii.edu/206L/197/fortran/fort4.htm`.
- The Mariner 1 example is taken from `http://wwwzenger.informatik.tu-muenchen.de/persons/huckle/bugse.html`.