Compiler Construction

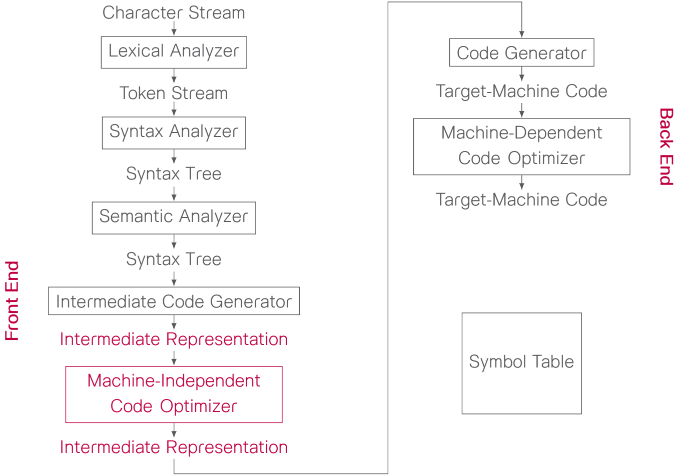
Chapter 2: Scanners

Dittaya Wanvarie

Department of Mathematics and Computer Science
Chulalongkorn University

Second semester, 2024

Overview of compilation



Scanner

- Scan through every character of the input program
  - Input: stream of characters
- Group characters together to form words and punctuation in the source language
  - Output: stream of words (lexemes), each labeled with its syntactic category
  - Rules that describe each of the category is called a microsyntax or lexical grammar
- Then, the output from the scanning process should be `<lexeme,category>`

Syntactic categories

In a typical programming language

- Identifier
  - A single alphabetic character followed by a zero or more alphanumeric characters
- Each of the keywords/reserved words, e.g. **`if`**, **`while`**
- Each of the operators, e.g. `+ - ; >=`

Efficient scanners can scan the input in $O(1)$ time per character.

Scanner's time requirement

- Design time e.g. writing language specifications
- Build time e.g. generating scanner codes
- Compile time i.e. runtime of a scanning process

Outline

Recognizing words

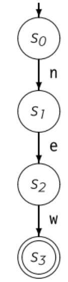
Since the lexical grammar is a regular grammar,

- we can recognize words in the grammar using a finite automaton.
- We can also write a regular grammar using a regular expression
- We can construct an NFA (and convert it to a DFA) from a regular expression
  - Hence, we can **generate** a scanner for any lexical grammar using regular expressions

Recognizing a word

```
c ← NextChar();
if (c = 'n') then
    c ← NextChar();
    if (c = 'e') then
        c ← NextChar();
        if (c = 'w') then
            report success;
        else
            try something else;
    else
        try something else;
else
    try something else;
```

(a) Code



(b) Recognizer

■ **FIGURE 2.1** Code Fragment to Recognize the Word "**new**".

Figure: Recognizing a word[1]

[1] Fig 2.1 from Keith D. Cooper and Linda Torczon. **Engineering a Compiler (Third Edition)**, Morgan Kaufmann, 2022.

Recognizing multiple words

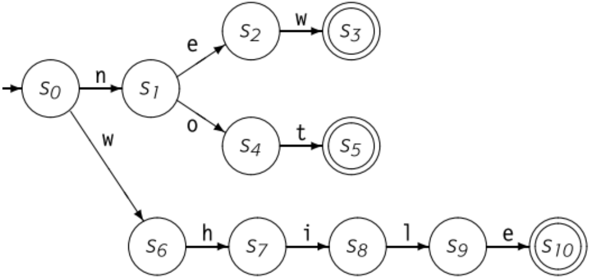


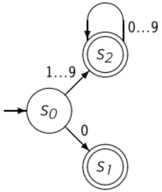
Figure: Recognizing multiple words[2]

[2] Keith D. Cooper and Linda Torczon. **Engineering a Compiler (Third Edition)**, Morgan Kaufmann, 2022. Page 32.

Formal definition

- $S = \{s_0, s_1, s_2, s_3, s_4, s_5, s_6, s_7, s_8, s_9, s_{10}, s_e\}$
- $\Sigma = \{\text{e, h, i, l, n, o, t, w}\}$
- $\delta = \left\{\begin{array}{ccccc} s_0 \xrightarrow{\text{n}} s_1, & s_0 \xrightarrow{\text{w}} s_6, & s_1 \xrightarrow{\text{e}} s_2, & s_1 \xrightarrow{\text{o}} s_4, & s_2 \xrightarrow{\text{w}} s_3, \\ s_4 \xrightarrow{\text{t}} s_5, & s_6 \xrightarrow{\text{h}} s_7, & s_7 \xrightarrow{\text{i}} s_8, & s_8 \xrightarrow{\text{l}} s_9, & s_9 \xrightarrow{\text{e}} s_{10} \end{array}\right\}$
- $s_0 = s_0$
- $S_A = \{s_3, s_5, s_{10}\}$

Recognizing unsigned integers



FA for Unsigned Integers

(a) FA[3]

```
state ← s0;
char ← NextChar();
while (state ≠ se and char ≠ eof) do
    state ← δ(state,char);
    char ← NextChar();
end;
if (state ∈ SA) then
    report acceptance;
else report failure;
```

(a) Code to Interpret State Table

$$S = \{s_0, s_1, s_2, s_e\}$$

$$\Sigma = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$$

$$\delta = \left\{ \begin{array}{ll} s_0 \xrightarrow{0} s_1, & s_0 \xrightarrow{1 \ldots 9} s_2 \\ s_2 \xrightarrow{0 \ldots 9} s_2 & \end{array} \right\}$$

$$S_A = \{s_1, s_2\}$$

(b) Formal Definition of the FA

■ **FIGURE 2.2** A Recognizer for Unsigned Integers.

(b) Formal definition[4]

Figure: Recognizing words

[3] Keith D. Cooper and Linda Torczon. **Engineering a Compiler (Third Edition)**, Morgan Kaufmann, 2022. Page 37.

[4] Fig 2.2 from Keith D. Cooper and Linda Torczon. **Engineering a Compiler (Third Edition)**, Morgan Kaufmann, 2022.

Regular expressions

- FAs can be viewed as specifications for a recognizer (of a c category)
- Transitions in an FA is the spelling of all words recognized in the language
- Regular expression (RE) is the format to describe spelling

RE has 3 basic operations

1. Alternation, $r_a | r_b$
2. Concatenation $r_a r_b$
3. Closure, r^*: zero or more copies of a pattern
   - Finite closure, r^i
   - Positive closure, r^+

Definition of RE

Recursive definition

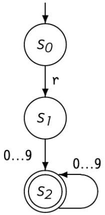
1. If a $\in \Sigma$, then a is an RE
2. If r_a and r_b are REs, then
   - $r_a | r_b$ is an RE
   - $r_a r_b$ is an RE
   - r_a^* is an RE
3. ϵ is an RE and denotes an empty string

Precedence: parentheses, closure, concatenation, alternation

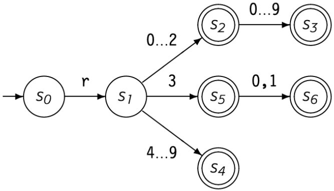
Examples of REs

- Identifier: $([A..Z]|[a..z]|\_)([A..Z][a..z][0..9])^*$
- Unsigned integer: $(0|[1..9][0..9]^*)$
- Unsigned decimals: $(0|[1..9][0..9]^*).(\epsilon|[0..9]^*)$
- Scientific notation of real numbers: $(0|[1...9][0...9]^*)(\epsilon|.[0...9]^*)(E|e)(|+|-)(0|[1...9][0...9]^*)$
- Quoted strings: we should not allow ïn the spelling
  - Complement, ^
  - $(\wedge(\texttt{"}|\texttt{\textbackslash n}))^*$
  - Comments: $/*(\wedge*|\wedge^+\wedge/)^**/$ in C/Java or $\wedge\#$ in Python

Which FA is better?



(a) Simple RE[5]



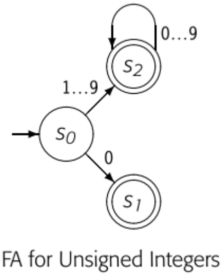
(b) More specific RE[6]

Figure: More states need more spaces, not more time

[5]Keith D. Cooper and Linda Torczon. **Engineering a Compiler (Third Edition)**, Morgan Kaufmann, 2022. Page 41.

[6]Keith D. Cooper and Linda Torczon. **Engineering a Compiler (Third Edition)**, Morgan Kaufmann, 2022. Page 41.

A set of languages generated from regular expressions are regular, and there are equivalent finite automata that accept/recognize those languages.



(a) FA for unsigned integers[7]

$$0|([1...9])([0...9])^*$$

(b) RE

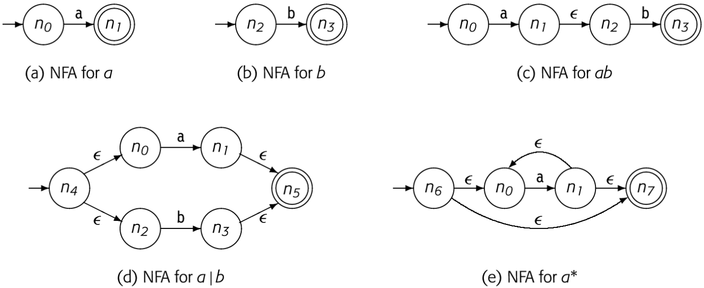
[7]Keith D. Cooper and Linda Torczon. **Engineering a Compiler (Third Edition)**, Morgan Kaufmann, 2022. Page 37.

From RE to DFA

Steps

1. Regular expression
2. Nondeterministic FA
3. Deterministic FA
4. Minimal DFA $\longrightarrow$ save memory

RE to NFA

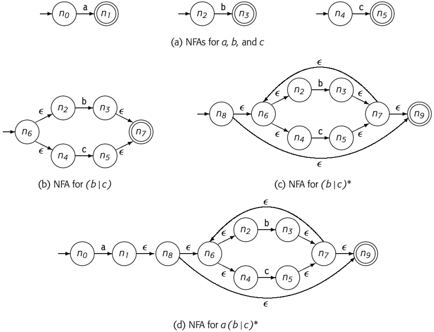


■ **FIGURE 2.4** Trivial NFAs for Regular Expression Operators.

Figure: NFA Construction from RE[8]

[8] Fig 2.4 from Keith D. Cooper and Linda Torczon. **Engineering a Compiler (Third Edition)**, Morgan Kaufmann, 2022.

RE to NFA: Thompson's construction



■ FIGURE 2.5 Applying Thompson's Construction to a (b | c)*.

Figure: Thompson's construction[9]

[9]Fig 2.5 from Keith D. Cooper and Linda Torczon. **Engineering a Compiler (Third Edition)**, Morgan Kaufmann, 2022.

NFA to DFA: Subset construction

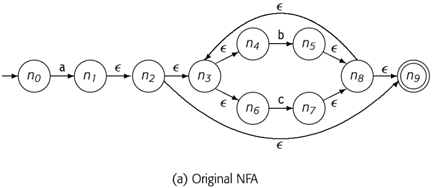

```
q0 ← FollowEpsilon({n0})
Q ← q0
WorkList ← {q0}

while (WorkList ≠ Ø) do
    remove q from WorkList

    for each character c ∈ Σ do
        temp ← FollowEpsilon(Delta(q, c))
        if temp ≠ Ø then
            if temp ∉ Q then
                add temp to both Q and WorkList
            T[q, c] ← temp
```

FIGURE 2.6 The Subset Construction.

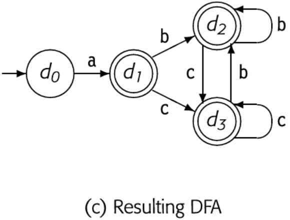
Figure: Subset construction[10]

[10] Fig 2.6 from Keith D. Cooper and Linda Torczon. **Engineering a Compiler (Third Edition)**, Morgan Kaufmann, 2022.

Subset construction example



(a) Original NFA[11]



(b) Final DFA[12]

[11] Fig 2.7(a) from Keith D. Cooper and Linda Torczon. **Engineering a Compiler (Third Edition)**, Morgan Kaufmann, 2022.

[12] Fig 2.7(c) from Keith D. Cooper and Linda Torczon. **Engineering a Compiler (Third Edition)**, Morgan Kaufmann, 2022.

Subset construction example (cont.)

Set Name	DFA State	NFA States	*FollowEpsilon(Delta(q,x))*		
			a	*b*	*c*
q_0	d_0	$\{n_0\}$	$\{n_1, n_2, n_3, n_4, n_6, n_9\}$	– *none* –	– *none* –
q_1	d_1	$\{n_1, n_2, n_3, n_4, n_6, n_9\}$	– *none* –	$\{n_5, n_8, n_9, n_3, n_4, n_6\}$	$\{n_7, n_8, n_9, n_3, n_4, n_6\}$
q_2	d_2	$\{n_5, n_8, n_9, n_3, n_4, n_6\}$	– *none* –	q_2	q_3
q_3	d_3	$\{n_7, n_8, n_9, n_3, n_4, n_6\}$	– *none* –	q_2	q_3

(b) Iterations of the Subset Construction

Figure: Iterations in subset construction[13]

[13] Fig 2.7(b) from Keith D. Cooper and Linda Torczon. **Engineering a Compiler (Third Edition)**, Morgan Kaufmann, 2022.

Fixed-point computations

- Iterating application of a monotone function
- Terminate when they reach a state where further iterations produces the same answer, a **fixed point**

Minimizing DFAs

Although the size of DFA does not affect the computation time, it does affect the memory requirement

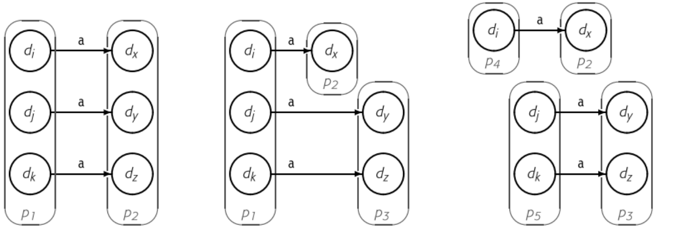
- Two states are **equivalent** when they produce the same behavior on **any** input string
- Then, we will partition a set into a **set of equivalent states**

Set partition

We construct a set partition $P = \{p_1, p_2, \ldots, p_m\}$ of the original DFA states with the following rules

1. $\forall c \in \Sigma$, if $d_i, d_j \in p_s$;
    - $d_i \xrightarrow{c} d_x$ and $d_j \xrightarrow{c} d_y$ and $d_x \in p_t$
    - then, $d_y \in p_t$
2. If $d_i, d_j \in p_k$ and $d_i \in D_A$
    - then $d_j \in D_A$

Set partitioning



(a) a Does Not Split p_1 (b) a Splits p_1 (c) Sets from Panel b After the Split

FIGURE 2.8 Splitting a Set Around a.

Figure: Splitting a set around a[14]

[14] Fig 2.8 from Keith D. Cooper and Linda Torczon. **Engineering a Compiler (Third Edition)**, Morgan Kaufmann, 2022.

Set partitioning algorithm

```
Partition ← {D_A, {D − D_A}}
Worklist ← {D_A, {D − D_A}}

while ( Worklist ≠ ∅ ) do
    select a set s from Worklist and remove it
    for each character c ∈ Σ do
        Image ← {x | δ(x,c) ∈ s}
        for each set q ∈ Partition that has a state in Image do
            q1 ← q ∩ Image
            q2 ← q − q1

            if q2 ≠ ∅ then              // split q around s and c
                remove q from Partition
                Partition ← Partition ∪ q1 ∪ q2

                if q ∈ Worklist then    // and update the Worklist
                    remove q from Worklist
                    WorkList ← WorkList ∪ q1 ∪ q2
                else if |q1| ≤ |q2| then
                    WorkList ← Worklist ∪ q1
                else WorkList ← WorkList ∪ q2

                if s = q then           // need another s
                    break
```

■ **FIGURE 2.9** DFA Minimization Algorithm.

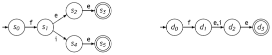
Figure: DFA minimization algorithm[15]

[15] Fig 2.9 from Keith D. Cooper and Linda Torczon. **Engineering a Compiler (Third Edition)**, Morgan Kaufmann, 2022.

Example: DFA minimization

Step	Partition on Entry	Worklist	s	c	Image	q	q_1	q_2	Action
–	p_0: $\{s_3, s_5\}$, p_1: $\{s_0, s_1, s_2, s_4\}$	p_0, p_1	—	—	—	—	—	—	—
1	p_0: $\{s_3, s_5\}$, p_1: $\{s_0, s_1, s_2, s_4\}$	p_1	p_0	e	s_2, s_4	p_1	s_2, s_4	s_0, s_1	*split* $p_1 \rightarrow p_2, p_3$
		p_2, p_3	p_0	f	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	*none*
		p_2, p_3	p_0	i	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	*none*
2	p_0: $\{s_3, s_5\}$, p_2: $\{s_2, s_4\}$, p_3: $\{s_0, s_1\}$	p_3	p_2	e	s_1	p_3	s_1	s_0	*split* $p_3 \rightarrow p_4, p_5$
		p_4, p_5	p_2	f	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	*none*
		p_4, p_5	p_2	i	s_1	$\emptyset$	$\emptyset$	$\emptyset$	*none*
3	p_0: $\{s_3, s_5\}$, p_2: $\{s_2, s_4\}$, p_4: $\{s_1\}$, p_5: $\{s_0\}$	p_5	p_4	e	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	*none*
		p_5	p_4	f	s_0	p_5	s_0	$\emptyset$	*none*
		p_5	p_4	i	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	*none*
4	p_0: $\{s_3, s_5\}$, p_2: $\{s_2, s_4\}$, p_4: $\{s_1\}$, p_5: $\{s_0\}$	$\emptyset$	p_5	e	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	*none*
		$\emptyset$	p_5	f	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	*none*
		$\emptyset$	p_5	i	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	*none*

(a) Iterations of Hopcroft's Algorithm on the Original DFA for "fee | fie"



(b) Original DFA for "fee | fie"

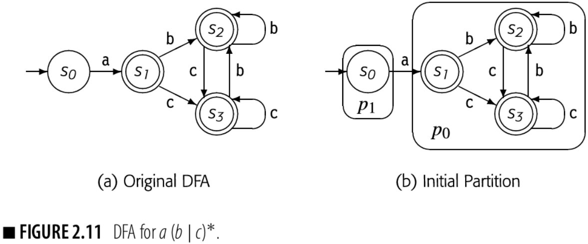
(c) Minimal DFA for "fee | fie"

FIGURE 2.10 Applying the DFA Minimization Algorithm.

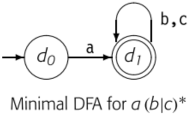
Figure: Applying DFA minimization algorithm[16]

[16] Fig 2.10 from Keith D. Cooper and Linda Torczon. **Engineering a Compiler (Third Edition)**, Morgan Kaufmann, 2022.

Example: DFA minimization (cont.)



(a) $a(b|c)^*$ [17]



(b) Final product[18]

[17] Fig 2.11 from Keith D. Cooper and Linda Torczon. **Engineering a Compiler (Third Edition)**, Morgan Kaufmann, 2022.

[18] Keith D. Cooper and Linda Torczon. **Engineering a Compiler (Third Edition)**, Morgan Kaufmann, 2022. Page 59.

Outline

Implementing scanners

- We can automate the scanner construction using language specifications; REs
- We have a scanning algorithm (FA derivations) that works for any programming language, as long as we have the REs

Difference between DFA and scanners

However,

- An FA accepts one language, but we need to recognize several languages i.e. one language for one syntactic category
- An FA either accepts or rejects the input after fully processing it. In contrast, a scanner reads just enough input to match one of the token types, then returns while maintaining its current position. It continues this process to match tokens sequentially in the input stream.

Therefore

- A scanner can (greedily) simulate a DFA until it hits an error
- If d_i is an accepting state, the scanner has found a word
- If d_i is not an accepting state, the scanner may have passed through a possible accepting states (due to it greedy manner)

Precedence

Sometimes, two lexical rules can overlap

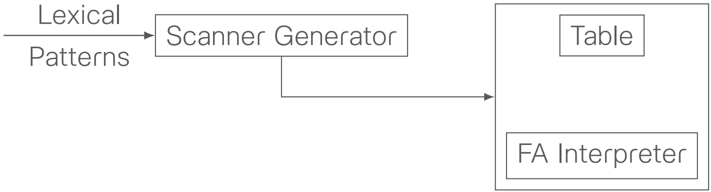
- A simple variable is also valid as a part of a string literal

To simplify the rule

- The compiler writer determines the precedence of the rules

Table-driven scanners

Structure



Pseudocode

1. Initialization of all data structures
2. Scanning the character stream
    - Simulating a DFA
3. Rollback if necessary
4. Return the result

Example: Register names

```
state ← s0;
lexeme ← "";
clear stack;
push(bad);

while (state ≠ se) do
    char ← NextChar();
    lexeme ← lexeme + char;

    if state ∈ SA then
        clear stack;
        push(bad);

    push(state);

    col ← CharClass[char];
    state ← δ[state,col];

while (state ∉ SA and state ≠ bad) do
    state ← pop();
    if state ≠ bad then
        truncate lexeme;
        RollBack();

if state ∈ SA
    then return Type[state];
    else return invalid;
```

(a) Code to Interpret the Tables

r	0...9	Other
Register	*Digit*	*Other*

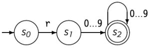
(b) The Classification Table, *CharClass*

	Register	Digit	Other
s_0	s_1	s_e	s_e
s_1	s_e	s_2	s_e
s_2	s_e	s_2	s_e
s_e	s_e	s_e	s_e

(c) The Transition Table, δ

s_0	s_1	s_2	s_e
invalid	*invalid*	*register*	*invalid*

(d) The Token Type Table, *Type*



(e) The Underlying DFA

FIGURE 2.12 A Table-Driven Scanner for Register Names.

Figure: Table-driven scanner for register names[19]

[19] Fig 2.12 from Keith D. Cooper and Linda Torczon. **Engineering a Compiler (Third Edition)**, Morgan Kaufmann, 2022.

Avoiding excess rollback

Greedy matching leads to several rollbacks

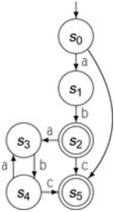


Figure: `ab | (ab)*c` and input `abababab`[20]

To avoid excess rollback,

- Track the current input position
- Record dead-end transitions

[20] Keith D. Cooper and Linda Torczon. **Engineering a Compiler (Third Edition)**, Morgan Kaufmann, 2022. Page 65.

The maximal munch scanner

```
state ← s0;
lexeme ← "";
clear stack;
push(⟨bad, -1⟩);
while (state ≠ se) do
    if Failed[state,InputPos] then
        ⟨state, InputPos⟩ ← pop();
        truncate lexeme;
        break;
    char ← Input[InputPos];
    lexeme ← lexeme + char;
    if state ∈ SA then
        clear stack;
        push(⟨bad, -1⟩);
    push(⟨state, InputPos⟩);
    col ← CharClass[char];
    state ← δ[state, col];
    InputPos ← InputPos + 1;
while (state ∉ SA and state ≠ bad) do
    if state ≠ se then
        Failed[state, InputPos] ← true;
    ⟨state, InputPos⟩ ← pop();
    if state ≠ bad then
        truncate lexeme;
if state ∈ SA
    then return TokenType[state];
    else return invalid;
```

■ **FIGURE 2.13** The Maximal Munch Scanner.

Figure: Maximum munch scanner[21]

[21] Fig 2.13 from Keith D. Cooper and Linda Torczon. **Engineering a Compiler (Third Edition)**, Morgan Kaufmann, 2022.

Direct-coded scanner: lookup

For each character, the table-driven scanner performs two table lookups

- `col = CharClass(char)`
- `state = delta[state, col]`

Although the lookup take $O(1)$ time, the actual constant-cost overheads can be avoid

- `state = delta_0 + (state * len(delta[0]) + col) * w`
- We will discuss the detail again in the optimization topic

Direct-coded scanner: while loop

Branching based on the characteristic of each state

- Each state has its own code fragment
- The state transition is a branch/jump/goto statement

The code is closely related to low-level language, which can be directly translated in to machine codes.

Example: Register names

```
s0:   clear stack;
      push(bad);
      char ← NextChar();
      lexeme ← char;
      push(s0);
      if (char = 'r')
         then goto s1;
         else goto sout;

s1:   char ← NextChar();
      lexeme ← lexeme + char;
      push(s1);
      if ('0' ≤ char ≤ '9')
         then goto s2;
         else goto sout;

s2:   char ← NextChar();
      lexeme ← lexeme + char;
      clear stack;
      push(s2);
      if '0' ≤ char ≤ '9'
         then goto s2;
         else goto sout;

sout: state ← se;
      while (state ∉ SA and state ≠ bad) do
         state ← pop();
         if state ≠ bad then
            truncate lexeme;
            RollBack();
      end;
      if state ∈ SA
         then return Type[state];
         else return invalid;
```

■ **FIGURE 2.14** A Direct-Coded Scanner for $r\ [0 \ldots 9]^+$.

Figure: Maximum munch scanner[22]

[22] Fig 2.14 from Keith D. Cooper and Linda Torczon. **Engineering a Compiler (Third Edition)**, Morgan Kaufmann, 2022.

Hand-coded scanners

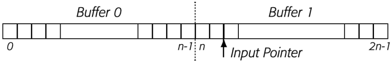
Manually written and optimized.

Integration issue

Since the scanner is only the first phase in the compilation, the ease of integration with later phase is also important

- Pipeline: read the input once and pass the matched token to the next phase
- Use input buffer

Buffer



(a) Double buffer[23]

```
NextChar() {
    Char ← Buffer[Input];
    if Char ≠ eof then
        Input ← (Input + 1) mod 2n;
        if (Input mod n = 0) then
            fill Buffer[Input : Input + n - 1];
            Fence ← (Input + n) mod 2n;
    return Char;
}

RollBack() {
    if (Input = Fence) then
        signal roll back error;
    else
        Input ← (Input - 1) mod 2n;
}

Initialize() {
    Input ← 0;
    Fence ← 0;
    fill Buffer[0 : n-1];
}
```

■ **FIGURE 2.15** Implementing *NextChar* and *RollBack*.

(b) Buffer implementation[24]

Figure: Buffer

[23] Keith D. Cooper and Linda Torczon. **Engineering a Compiler (Third Edition)**, Morgan Kaufmann, 2022. Page 71.

[24] Fig 2.15 from Keith D. Cooper and Linda Torczon. **Engineering a Compiler (Third Edition)**, Morgan Kaufmann, 2022.

Compressing the transition table

The size of alphabet and states are large and may not be fit in the first-level cache.

δ	0	1	2	3	4	5	6	7	8	9	Other
s_0	s_1	s_2	s_2	s_2	s_2	s_2	s_2	s_2	s_2	s_2	s_e
s_1	s_e	s_e	s_e	s_e	s_e	s_e	s_e	s_e	s_e	s_e	s_e
s_2	s_2	s_2	s_2	s_2	s_2	s_2	s_2	s_2	s_2	s_2	s_e
s_e	s_e	s_e	s_e	s_e	s_e	s_e	s_e	s_e	s_e	s_e	s_e

(a) The Full Transition Table for $0 \mid [1\ldots9]\,[0\ldots9]^*$

δ	0	1…9	Other
s_0	s_1	s_2	s_e
s_1	s_e	s_e	s_e
s_2	s_2	s_2	s_e
s_e	s_e	s_e	s_e

(b) The Compressed Table

FIGURE 2.16 Transition-Table Compression Example.

Figure: Compressing transition table[25]

However, we still need a map from a character to a column (character class).

[25] Fig 2.16 from Keith D. Cooper and Linda Torczon. **Engineering a Compiler (Third Edition)**, Morgan Kaufmann, 2022.

Outline

Examples

- C: lex, flex
- Java: JavaCC
- Python: PLY

And many more:
`https://en.wikipedia.org/wiki/Comparison_of_parser_generators`