2 Scanners

Contents of this module

1. Recognizing words
2. Regular expressions
3. From regular expression to scanner
    - Thompson's construction
    - Subset construction
    - Hopcroft's minimization algorithm
4. Implementing scanners
    - Table-driven scanners
    - Direct-coded scanners
    - Hand-coded scanners

Scanner

The scanner, or lexical analyzer, reads a stream of characters and produces a stream of words or tokens

- applies rules that describe the **lexical structure** of the input programming language
- assigns each valid word a **syntactic category**, or “part of speech”

Scanners can be implemented very efficiently

- **scanner generator**: tool that processes mathematical description of language’s lexical syntax to produce a fast recognizer
- **hand-crafted scanner**: can be faster by avoiding a portion of the overhead that cannot be avoided in a generated scanner

Both types of scanners can be implemented to require just $O(1)$ time per character, *i.e.*, they run in time proportional to the number of characters in the input stream.

Microsyntax

The lexical structure of a language is also called the **microsyntax** of a programming language

- specifies how to **group** characters into words
- specifies how to **separate** words that run together

The set of valid words is typically specified by rules rather than by enumeration in a dictionary.

Keywords

- a word that is reserved for a particular syntactic purpose
- cannot be used as an identifier

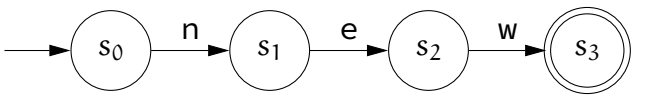
To recognize keywords, the scanner can either use dictionary lookup or encode the keywords directly into its microsyntax rules.

Recognizing Words

Consider the problem of recognizing the keyword `new`. We can formulate an algorithm that processes the input stream character by character.

```
1  c ← NextChar()
2  if c = 'n' then
3      c ← NextChar()
4      if c = 'e' then
5          c ← NextChar()
6          if c = 'w' then
7              return success
8          else
9              try something
                else
10     else
11         try something else
12 else
13     try something else
```

This code fragment can be represented using a simple **transition diagram**



The transition diagram represents a **recognizer**

- circles represent abstract **states** in the computation
- s_0 is the **initial state**, or start state
- s_3 is an **accepting state**, drawn with double circles

Recognizing Words

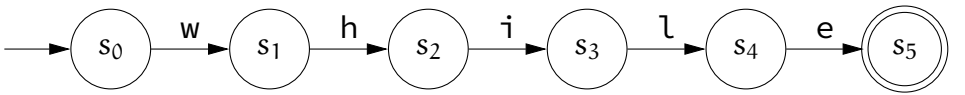
What happens to any other input that is **not** the keyword `new`, *e.g.*, `not`?

- the `n` takes the recognizer into state s_1, but the `o` does not match the edge leaving s_1
- in the code, `if` cases that do not match `new` use an `else` branch to *try something else*
- in the recognizer, this action can be represented as a transition to an **error state**.

Note When we draw the transition diagram of a recognizer, we usually omit transitions to the error state. Each state has a transition to the error state on each unspecified input.

Recognizing Words

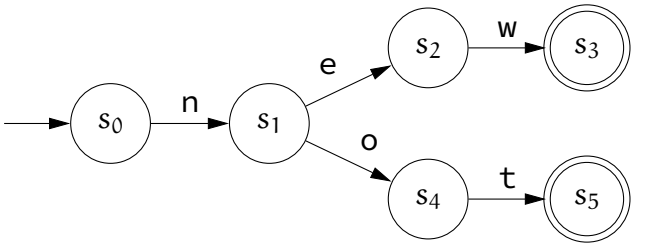
Using the same approach to build a recognizer for the keyword `while` would produce the following transition diagram.



Translating this recognizer to code would involve **five** nested `if-then-else` statements.

Recognizing Words

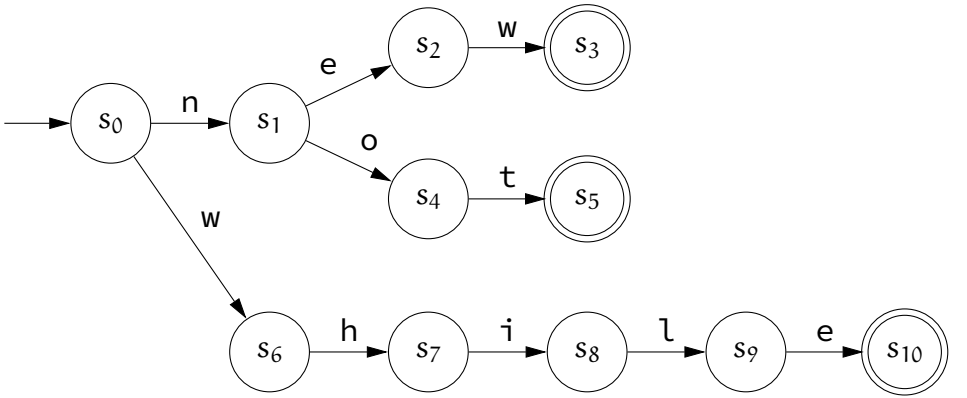
To recognize multiple words such as `new` and `not`, we can create **multiple edges** that leave a given state.



To translate this recognizer to code, we need to elaborate the *try something else* paths.

Recognizing Words

We can combine the recognizer for `new` or `not` with the one for `while` by **merging** their initial states and **relabeling** all the states.



The recognizer has three accepting states, s_3, s_5, and s_{10}. If any state encounters an input character that does not match one of its transitions, the recognizer moves to an error state.

A Formalism for Recognizers

Towards formalizing recognizers…

- transition diagrams are a concise **abstraction** of the code that would be required to implement the corresponding recognizer
- they can also be described as formal mathematical objects, called **finite automata**

In the following, we will use finite automata as a **formalism** to specify recognizers.

A Formalism for Recognizers

A **Finite Automaton** (FA) is a five-tuple $(S, \Sigma, \delta, s_0, S_A)$, where

- S is the finite set of states in the recognizer, along with an error state s_e.
- Σ is the finite alphabet used by the recognizer. Typically, Σ is the union of the edge labels in the transition diagram.
- $\delta(s, c)$ is the recognizer's transition function. It maps each state $s \in S$ and each character $c \in \Sigma$ into some next state. In state s_i with input character c, the FA takes the transition $s_i \xrightarrow{c} \delta(s_i, c)$.
- $s_0 \in S$ is the designated start state.
- S_A is the set of accepting states, $S_A \subseteq S$. Each state in S_A appears as a double circle in the transition diagram.

A Formalism for Recognizers

Example

The transition diagram of the recognizer for `new`, `not`, and `while` can be formalized as the following finite automaton.

$$S = \{s_0, s_1, s_2, s_3, s_4, s_5, s_6, s_7, s_8, s_9, s_{10}, s_e\}$$

$$\Sigma = \{\mathtt{e}, \mathtt{h}, \mathtt{i}, \mathtt{l}, \mathtt{n}, \mathtt{o}, \mathtt{t}, \mathtt{w}\}$$

$$\delta = \left\{ \begin{array}{l} s_0 \xrightarrow{\mathtt{n}} s_1, s_0 \xrightarrow{\mathtt{w}} s_6, s_1 \xrightarrow{\mathtt{e}} s_2, s_1 \xrightarrow{\mathtt{o}} s_4, s_2 \xrightarrow{\mathtt{w}} s_3, \\ s_4 \xrightarrow{\mathtt{t}} s_5, s_6 \xrightarrow{\mathtt{h}} s_7, s_7 \xrightarrow{\mathtt{i}} s_8, s_8 \xrightarrow{\mathtt{l}} s_9, s_9 \xrightarrow{\mathtt{e}} s_{10} \end{array} \right\}$$

$$s_0 = s_0$$

$$S_A = \{s_3, s_5, s_{10}\}$$

For all other combinations of state s_i and input character c, we define $\delta(s_i, c) = s_e$, where s_e is the designated error state.

A Formalism for Recognizers

A FA accepts a string x if and only if, starting in s_0, the sequence of characters takes the FA through a transition sequence that leaves it in an accepting state when the entire string has been consumed.

Examples

`new` $s_0 \xrightarrow{n} s_1, s_1 \xrightarrow{e} s_2, s_2 \xrightarrow{w} s_3$

$\longrightarrow$ the FA **accepts** the word since $s_3 \in S_A$

`nut` $s_0 \xrightarrow{n} s_1, s_1 \xrightarrow{u} s_e$

$\longrightarrow$ once the FA enters s_e, it stays in s_e until it exhausts the input stream

A Formalism for Recognizers

More formally, if the string x is composed of characters $x_1x_2x_3 \ldots x_n$, then the FA $(S, \Sigma, \delta, s_0, S_A)$ accepts x if and only if

$$\delta(\delta(\ldots\delta(\delta(\delta(s_0, x_1), x_2), x_3)\ldots, x_{n-1}), x_n) \in S_A.$$

Apart from ending up in an accepting state, two other cases are possible.

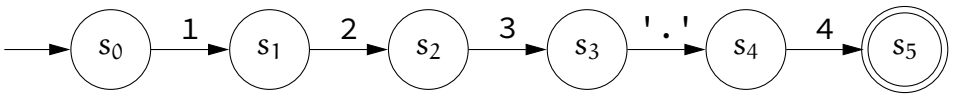
- a character x_j takes the FA into the error state s_e: this indicates a **lexical error**, *i.e.*, $x_1x_2x_3 \ldots x_j$ is not a valid prefix for any word in the accepted language
- the FA terminates in a non-accepting state other than s_e after exhausting its input: this also indicates an **error**, even though the input is a valid prefix of some accepted word

Note In any case, the FA takes one transition for each input character. Implemented efficiently, we expect the recognizer to run in time proportional to the length of the input.

Recognizing More Complex Words

The character-by-character model used so far can easily be extended to handle arbitrary collections of fully specified words. Can we use the same approach to recognize numbers?

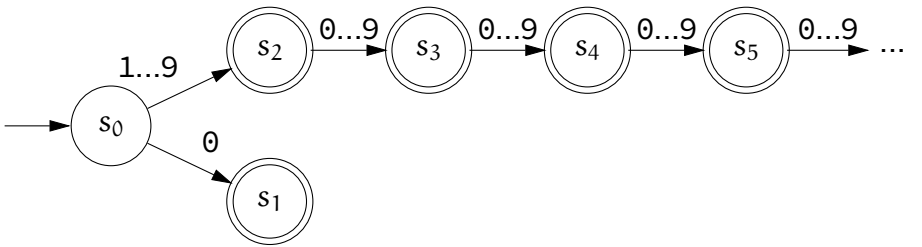
A **specific number**, such as 123.4, is straightforward.



But how would we draw a transition diagram (and design the corresponding code fragment) that can recognize **any number**, for example, unsigned integers?

Recognizing More Complex Words

An **unsigned integer** is either zero, or it is a series of one or more digits, where the first digit is from one to nine, and the subsequent digits are from zero to nine.



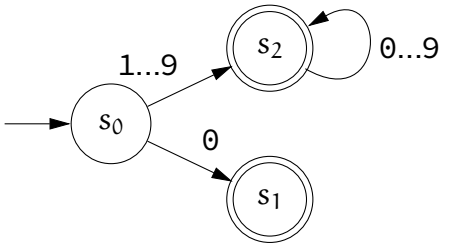
Problems

- path from s_0 to s_2, to s_3, and so on does not end, *i.e.*, this is not a **finite** automaton
- states $s_2, s_3, \ldots$ are all **equivalent**, *i.e.*, accepting states with same outgoing labels

Recognizing More Complex Words

We can fix both of these problems by allowing the transition diagram to have **cycles**

→ replace entire chain of states beginning at s_2 with a single transition from s_2 back to itself



Note This FA recognizes a class of strings with a common property, *i.e.*, unsigned integer

- class “unsigned integer” is a **syntactic category**
- text of a specific unsigned integer is its **lexeme**

$$S = \{s_0, s_1, s_2\}$$
$$\Sigma = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$$
$$\delta = \left\{ \begin{array}{l} s_0 \xrightarrow{0} s_1, s_0 \xrightarrow{1-9} s_2, \\ s_2 \xrightarrow{0-9} s_2, s_1 \xrightarrow{0-9} s_e \end{array} \right\}$$
$$s_0 = s_0$$
$$S_A = \{s_1, s_2\}$$

Recognizing More Complex Words

The introduction of cycles in the transition graph creates the need for **cyclic control flow**

- can be implemented with a `while`-loop
- δ can be specified efficiently as a table

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

```
1 c ← NextChar()
2 s ← s_0
3 while c ≠ eof ∧ s ≠ s_e do
4 |   s ← δ(s, c)
5 |   c ← NextChar()
6 if s ∈ S_A then
7 |   report acceptance
8 else
9 |   report failure
```

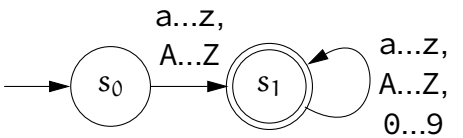
Note The last row of the δ-table can be omitted, since failure is reported as soon as the FA enters the error state s_e.

Recognizing More Complex Words

Using the same technique, we can develop FAs to recognize strings from other syntactic categories, such a signed integers, real numbers, and complex numbers.

An important syntactic category are **identifier names**

→ an alphabetic character followed by zero or more alphanumeric characters



Note Most programming languages extend the notion of “alphabetic character” to include designated special characters, such as the underscore.

Regular Expressions

To simplify and automate scanner implementation, we need…

- a concise notation for specifying the lexical structure of words,
- a way of turning those specifications into an FA, and
- a mechanism to produce code that implements the FA.

In the following, we look at these three problems in turn, beginning with the first.

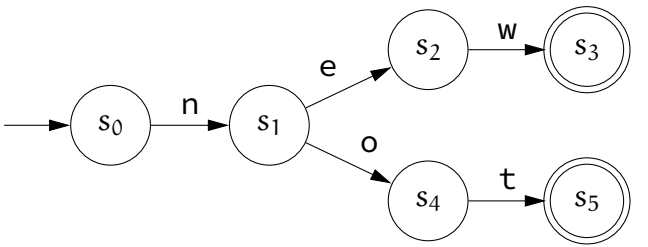
Note The set of words accepted by a FA, $\mathcal{F}$, forms a language, denoted $\mathrm{L}(\mathcal{F})$. We can also describe its language using a notation called a **regular expression** (RE). The language described by an RE is called a **regular language**.

Regular Expressions

Examples

- The language consisting of the single word `new` is described by an RE written as *new*
- The language consisting of the two words `new` or `while` is written as *new* | *while*
- The language consisting of `new` or `not` can be written as *new* | *not* or as *n (ew* | *ot)*

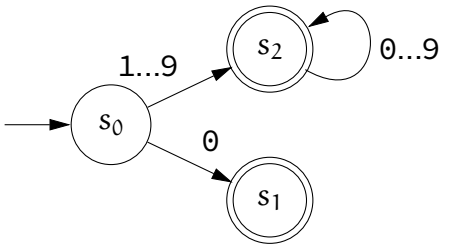
The RE *n (ew* | *ot)* suggests the structure of the FA that we drew earlier to recognize these two words.



Regular Expressions

Recall the FA that we developed for unsigned integers

- uses a cycle to recognize arbitrary integers of any length, other than `0`
- to write it as a RE, we need a notation for this notion of “zero or more occurrences”



For a RE χ, the meaning “zero or more occurrences of χ” is written a χ^*. We call the $*$ operator **Kleene closure**, or **closure** for short.

Example Using the closure operator, we can write an RE for this FA:

*0 | (1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9) (0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9)**

Regular Expressions

An **Regular Expression** (RE) describes a set of strings over the characters contained in some alphabet, Σ, augmented with a character ϵ that represents the empty string. For a given RE, r, we denote the language that it specifies as $L(r)$. A RE is built up from three basic operations:

1. The **alternation**, or union, of two sets of strings, R and S, denoted $R \,|\, S$, is $\{x \,|\, x \in R \vee x \in S\}$.
2. The **concatenation** of two sets R and S, denoted RS, contains all strings formed by prepending an element of R onto one from S, or $\{xy \,|\, x \in R \wedge y \in S\}$.
3. The **Kleene closure** of a set R, denoted R^*, is $\bigcup_{i=0}^{\infty} R^i$. This is just the union of the concatenations of R with itself, zero or more times.

Regular Expressions

For convenience, we introduce the following notation (syntactic sugar).

- **Finite Closure** R^i: one to i occurrences of R, *e.g.*, $R^3 = (R \mid RR \mid RRR)$
- **Positive Closure** R^+: one or more occurrences of R, *i.e.*, RR^*
- **Ranges** $[x_0 \ldots x_n]$: abbreviate ranges *e.g.*, $[0 \ldots 3] = (0 \mid 1 \mid 2 \mid 3)$
- **Complement** $\hat{}c$: complement of c with respect to Σ, *i.e.*, $\{\Sigma - c\}$

Note To eliminate any ambiguity, parentheses have highest precedence, followed by complement, closure, concatenation, and alternation, in that order.

Regular Expressions

Exercise

Design a RE to match Java-style multi-line comments. Can you think of a FA that would implement this RE?

Regular Expressions

Regular expressions are **closed** under many operations. These closure properties play a critical role in the use of REs to build scanners.

- Closure under **union** implies that any finite language is a regular language: we can construct an RE for any finite collection of words by listing them in a large alternation.
- Closure under **concatenation** allows us to build complex res from simpler ones by concatenating them.
- Closure under both **Kleene closure** and the finite closures: we can specify particular kinds of large, or even infinite, sets with finite patterns.

The fact that REs are closed under alternation, concatenation, and closure is critical to the constructions that we will look at in the following.