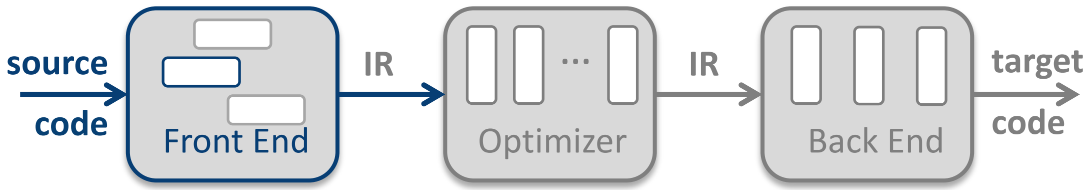
COMP 412
FALL 2018

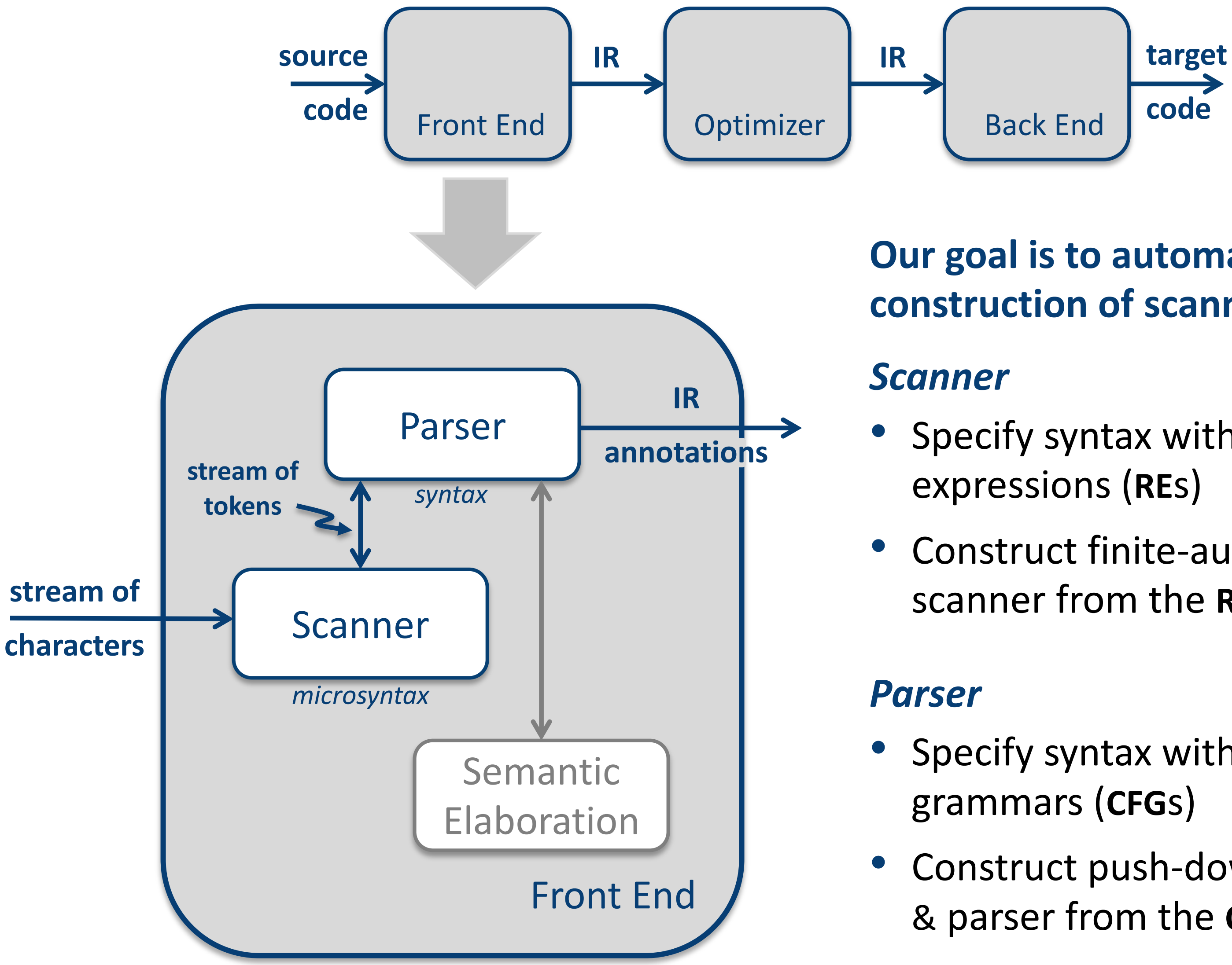
Lexical Analysis, I

Comp 412



Chapter 2 in EaC2e

Implementation Strategies



Our goal is to automate the construction of scanners & parsers

Scanner

- Specify syntax with regular expressions (**RE**s)
- Construct finite-automaton & scanner from the **RE**

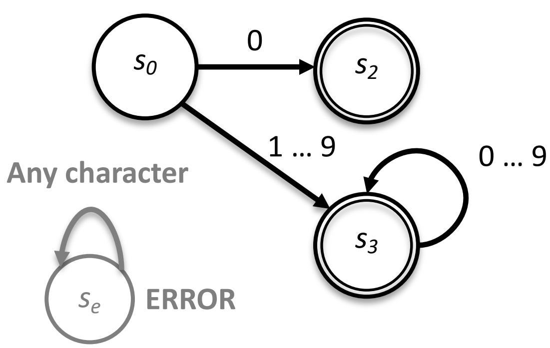
Parser

- Specify syntax with context-free grammars (**CFG**s)
- Construct push-down automaton & parser from the **CFG**

Big Picture

In Lecture 2, we saw some ambiguity in defining "positive integer"

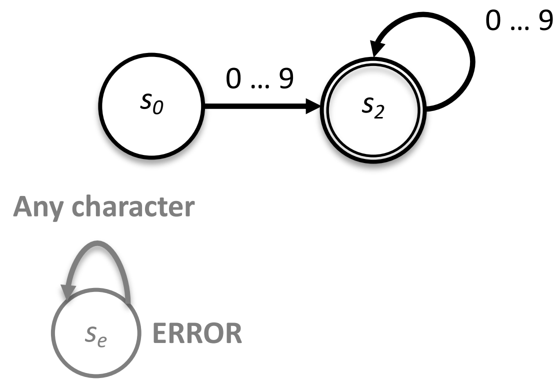
- Is 001 a positive integer? What about 00?
- The automata are precise specifications, but the words are not



Transitions to s_e are implicit from every state

Tasteful Positive Integer

(forbids 001)



Transitions to s_e are implicit from every state

Tasteless Positive Integer

(allows 001)

We need a better notation for specifying microsyntax than these transition diagrams.

Regular Expressions

We need a better notation for specifying microsyntax

"**better**" ⇒ both formal and constructive

Regular Expressions over an Alphabet Σ

- If $\underline{x} \in \Sigma$, then $\underline{x}$ is an **RE** denoting the set $\{ \underline{x} \}$ or the language $L = \{ \underline{x} \}$
- If $\underline{x}$ and $\underline{y}$ are **RE**s then
  - $\underline{xy}$ is an **RE** denoting $L(\underline{x})L(\underline{y}) = \{ pq \mid p \in L(\underline{x}) \text{ and } q \in L(\underline{y}) \}$
  - $\underline{x} \mid \underline{y}$ is an **RE** denoting $L(\underline{x}) \cup L(\underline{y})$
  - $\underline{x}^*$ is an **RE** denoting $L(\underline{x})^* = \cup_{0 \le k < \infty} L(\underline{x})^k$ (*Kleene Closure*)
    → *Set of all strings that are zero or more concatenations of* $\underline{x}$
  - $\underline{x}^+$ is an **RE** denoting $L(\underline{x})^+ = \cup_{1 \le k < \infty} L(\underline{x})^k$ (*Positive Closure*)
    → *Set of all strings that are one or more concatenations of* $\underline{x}$ (or $\underline{xx}^*$)
- ε is an **RE** denoting the empty set

Many **RE**-based systems support additional notation and operators. Those added features build on alternation, concatenation, and closure — plus, perhaps logical complement or negation. Complement is easy and efficient, if we think of the underlying **DFA**. (We will revisit this issue.)

Regular Expressions

How do these operators help?

The operators are *concatenation*, *alternation*, and *closure*.

Regular Expressions over an Alphabet Σ

- If $\underline{x}$ is in Σ, then $\underline{x}$ is an **RE** denoting the set $\{ \underline{x} \}$ or the language $L = \{ \underline{x} \}$

  → *The spelling of any letter in the alphabet is an* ***RE***
- If $\underline{x}$ and $\underline{y}$ are **RE**s then
  - $\underline{xy}$ is an **RE** denoting $L(\underline{x})L(\underline{y}) = \{ pq \mid p \in L(\underline{x}) \text{ and } q \in L(\underline{y}) \}$

    → ***If we concatenate letters, the result is an RE, so we can spell words***
  - $\underline{x} \mid \underline{y}$ is an **RE** denoting $L(\underline{x}) \cup L(\underline{y})$

    → ***Any finite list of words can be written as an RE,*** $(w_0 \mid w_1 \mid w_2 \mid \ldots \mid w_n)$
  - $\underline{x}^*$ is an **RE** denoting $L(\underline{x})^* = \cup_{\mathbf{0} \leq k < \infty} L(\underline{x})^k$
  - $\underline{x}^+$ is an **RE** denoting $L(\underline{x})^+ = \cup_{\mathbf{1} \leq k < \infty} L(\underline{x})^k$

    → ***We can use closure to write finite descriptions of infinite, but countable, sets***
- ε is an **RE** denoting the empty set

  → ***ε is sometimes useful for writing more concise REs***

Regular Expressions

Let the notation [a…z] be shorthand for the RE

(a | b | c | d | e | f | g | h | i | j | k | l | m | n | o | p | q | r | s | t | u | v | w | x | y | z)

Examples

Tasteless positive integer [0…9] [0…9]*

or [0…9]+

Tasteful positive integer 0 | [1…9] [0…9]*

Identifier (Algol-like lang) ([a…z] | [A…Z]) ([a…z] | [A…Z] | [0…9])*

Decimal number 0 | [1…9] [0…9]* . [0…9]*

Real number ((0 | [1…9] [0…9]*) | (0 | [1…9] [0…9]* . [0…9]*) E [0…9] [0…9]*

Each of these **RE**s corresponds to an automaton. More precisely, they correspond to a ***deterministic finite automaton***, *or* **DFA.**

- **Deterministic:** at each point, it makes a consistent, predictable decision
- **Finite:** a bounded number of states in the automaton

What Is The Point?

RE-derived scanners require **O**(1) time per character with tiny constant overhead.

Why do we care about regular expressions in the context of a compiler?

- We use **RE**s to specify *microsyntax* —- the mapping of spelling to parts of speech
  - An identifier is **([a…z] | [A…Z]) ([a…z] | [A…Z] | [0…9])***
  - Keywords are specified by their spellings, e.g., **if**, **then**, **else**
  - Those spellings are, in turn, **RE**s

We typically add some special characters, e.g., _ # $ @

- We use tools derived from automata theory to derive a **DFA** from the **RE**s and then convert the **RE** to code that implements a scanner
  - Automatic construction reduces the time & cost of scanner construction
  - Derivation from a formal notation eliminates implementation errors
  - Resulting scanners are both ***efficient*** (**O**(*n*)) and ***fast*** (*low constant overhead*)
- **RE**-derived scanners are widely used
  - Compilers, text editors, input checking on web pages, software to filter **URL**s

Digression # 1

How Does Class Relate to Regex Libraries?

Regular expressions (called REs, or regexes, or regex patterns) are essentially a tiny, highly specialized programming language embedded inside Python and made available through the re module. …

Regular expression patterns are **compiled into a series of bytecodes which are then executed by a matching engine** written in C. For advanced use, **it may be necessary to pay careful attention to how the engine will execute a given RE, and write the RE in a certain way in order to produce bytecode that runs faster. Optimization** isn't covered in this document, because it **requires that you have a good understanding of the matching engine's internals**.

The regular expression language is relatively small and restricted, so not all possible string processing tasks can be done using regular expressions. There are also tasks that can be done with regular expressions, but the expressions turn out to be very complicated. In these cases, you may be better off writing Python code to do the processing; while Python code will be slower than an elaborate regular expression, it will also probably be more understandable.

From Python 2.7.10 documentation, emphasis added

- You will learn how to "compile" **RE**s to a **DFA** & implement a **DFA**
  - Execution cost is guaranteed **O**(1) per input character, *independent* of the expression
- You will have deeper understanding of their power & their use

Digression # 2

A Digression on Time

(the "meta" issue)

In COMP 412, we will talk about a lot of "times"

- Design time, build time, compile time, run time, …
- In practice, the issue of when something happens is one that causes a great deal of confusion among students of compiler construction

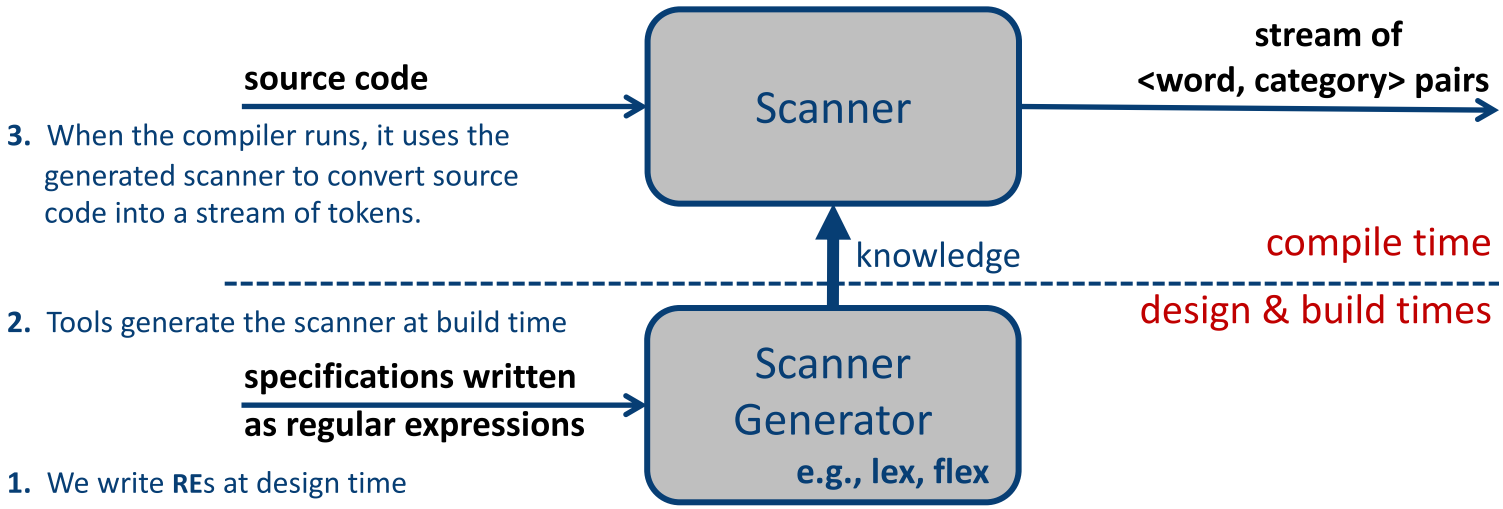
Small # of builds
  - Design time and build time happen long before compiler runs
    - → *Costs incurred at design or implementation time do not increase compile time*

Billions of compiles
  - Compile time happens every time the user invokes the compiler
    - → *Users are, appropriately, sensitive to compile time*
    - → *Costs incurred at compile time do not increase run time*

many per compile
  - Run-time costs affect actual application performance
    - → *One critical goal for compilation is to keep run time to a minimum, which means reducing the overhead introduced by translation*

As we look at strategies for *generating scanners* & *parsers*, keep in mind that generation costs are incurred at implementation time

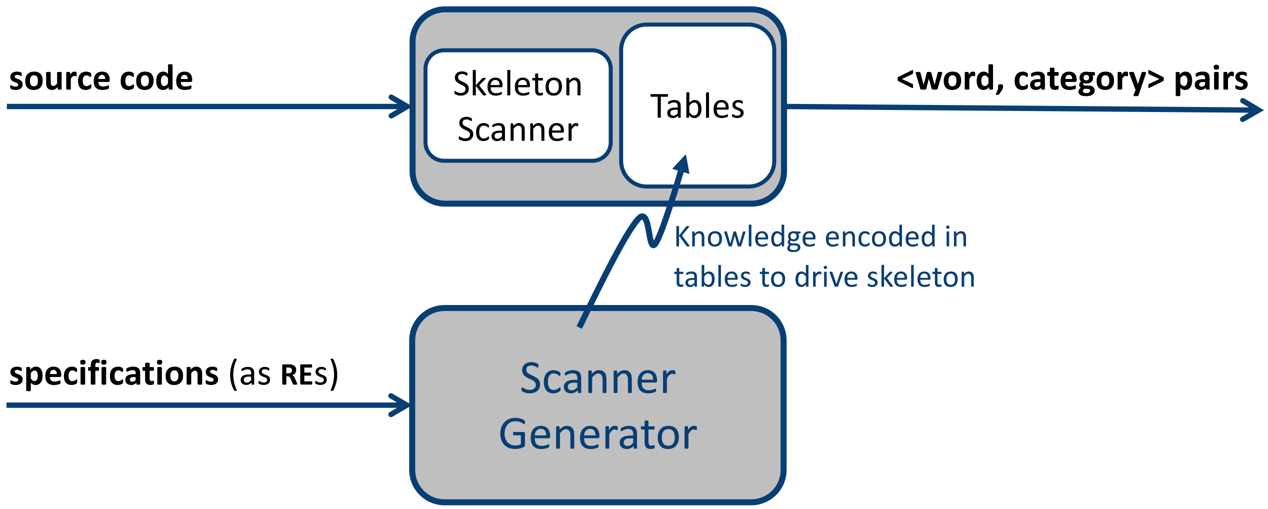
Automatic Scanner Construction: Meta Issues



Goals

- Simplify the construction of robust, efficient scanners
- Develop techniques that have widespread applicability
- Understand the underlying theory & practice

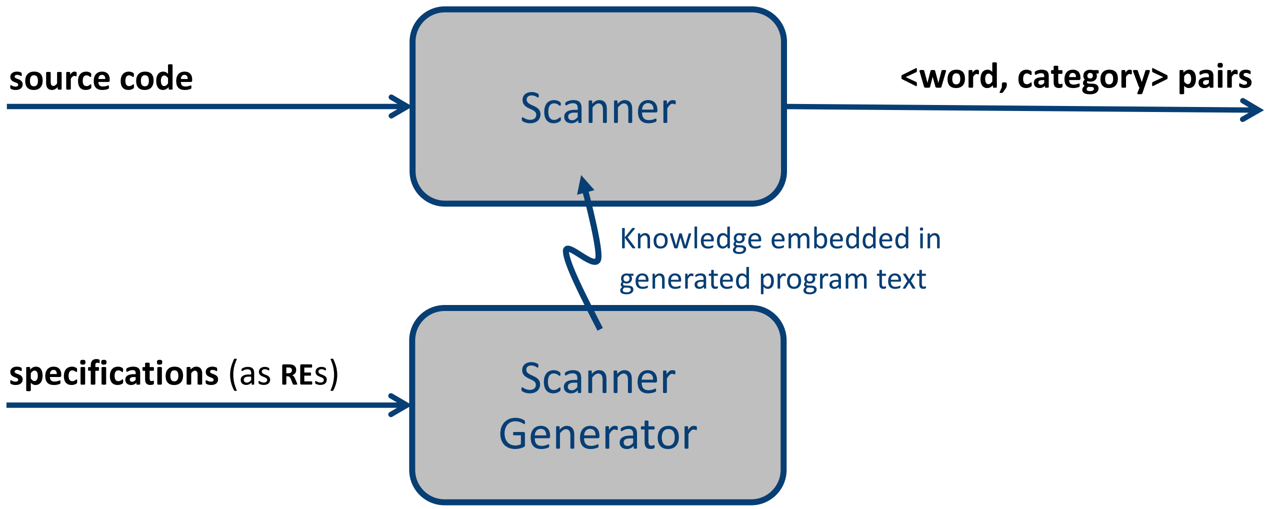
Automatic Scanner Construction



Scanner Generator

- May encode its knowledge in tables that drive a “skeleton scanner”
  - Skeleton scanner interprets the tables to simulate the **DFA** See § 2.5.1
- Every scanner uses the same skeleton, independent of **RE**
- Scanner generator builds the **DFA** from the **RE**, & converts it to a table

Automatic Scanner Construction



Scanner Generator

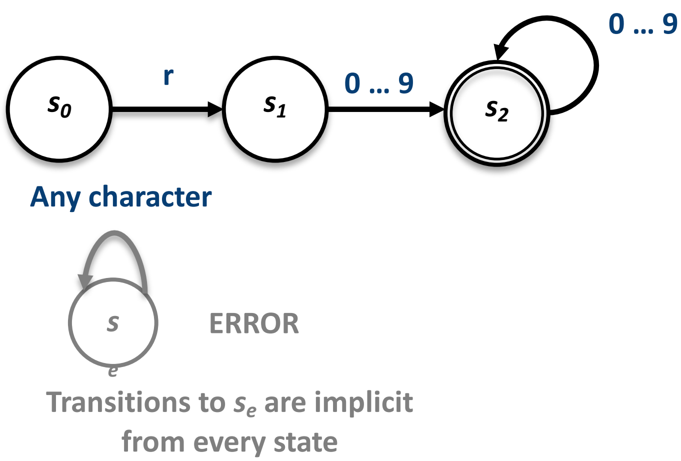
- May encode its knowledge of the recognizer directly into code
  - Transitions are compiled into conditional logic

See § 2.5.2

- Scanners for different **RE**s are different
- Produces a scanner that has very low overhead per character
- Scanner generator builds the **DFA** from the **RE**, & emits code for it

Example from Lecture 2

Recognizer for an ILOC register name (allow redundant zeros)



Recognizer for r [0…9] [0…9]*

Rules for DFA Operation

- Start in state s_0 & make transitions on each input character
- **DFA** accepts a word x <u>if and only if</u> x leaves the **DFA** in an accepting s_i or final state
- If the **DFA** encounters a character with no specified transition, it moves to s_e & stays in that state
- <u>r17</u> takes it through s_0, s_1, s_2, s_2 and it accepts
- <u>r</u> takes it through s_0, s_1 and it fails
- <u>ra</u> takes it through s_0, s_1, and s_e, so it fails

We will use the **RE** for a register name as a continuing example.

Example

Skeleton Scanner | Tables

To be useful, the DFA must be executable

```
char ← next character
state ← s0
while (char ≠ EOF) {
    state ← δ[state,char]
    char ← next character
}

if (state is a final state)
    then report success
    else report failure
```

O(1) per character

Skeleton Scanner

δ	r	0,1,2,3,4, 5,6,7,8,9	Any 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

Character classifier maps any character into one of the 3 classes: {r}, {0…9},{ all others}

Transition Table (δ)

For each character, the skeleton scanner does a table lookup and reads the next character — both of which should be **O**(1) operations

This skeleton scanner is simplified. See Figure 2.14 in § 2.5.1 of EaC2e and the accompanying text.

Example

Skeleton Scanner | Tables

To capture and classify the lexeme, we add a little work to each state

```
char ← next character
state ← s0
lexeme ← null string
while (char ≠ EOF) {
    lexeme ← lexeme || char
    state  ← δ[state,char]
    char   ← next character
}

If (state is a final state) then {
      category ← f(state)
      return <lexeme,category>
}
else  report failure
```

Still
O(1)

Skeleton Scanner

δ	**r**	**0,1,2,3,4, 5,6,7,8,9**	**Any 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

Transition Table (δ)

Skeleton Scanner | Tables

Example

To capture the register number, we would need state-specific actions

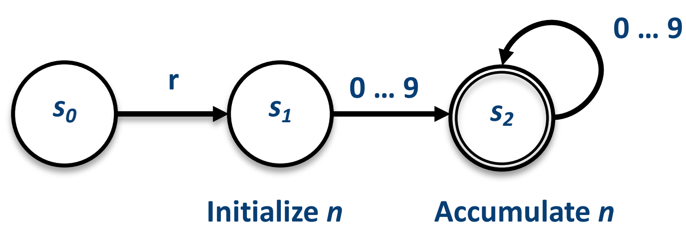
```
char ← next character
state ← s0
while (char ≠ EOF) {
   state  ← δ[state,char]
   char   ← next character
   if (state = s1)
      n ← 0
   else if (state = s2)
      n ← n *10 + char – ‘0’
}

If (state is a final state) then {
     category ← f(state)
     return <lexeme,category>
}
else  report failure
```

Still
O(1)

δ	r	0,1,2,3,4, 5,6,7,8,9	Any 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

Transition Table (δ)



More Complex **RE**s

What about a more complex language?

- r [0…9] [0…9]* allows arbitrary register numbers (*e.g.*, r000 or r999)
- What if we want to limit the register name to r0 through r31?

Write a tighter specification into the **RE**

- r ((0|1|2) ([0…9] | ε) | (4|5|6|7|8|9) | (3|30|31))
- r0|r1|r2|r3| … |r31|r00|r01|r02| … |r09

Non-standard use of …
but the meaning is clear

Each of these **RE**s can be converted to a **DFA**

- The **DFA** has the same **O**(1) cost per transition
- The **DFA** takes one transition per input character
- The **DFA** uses the same skeleton scanner

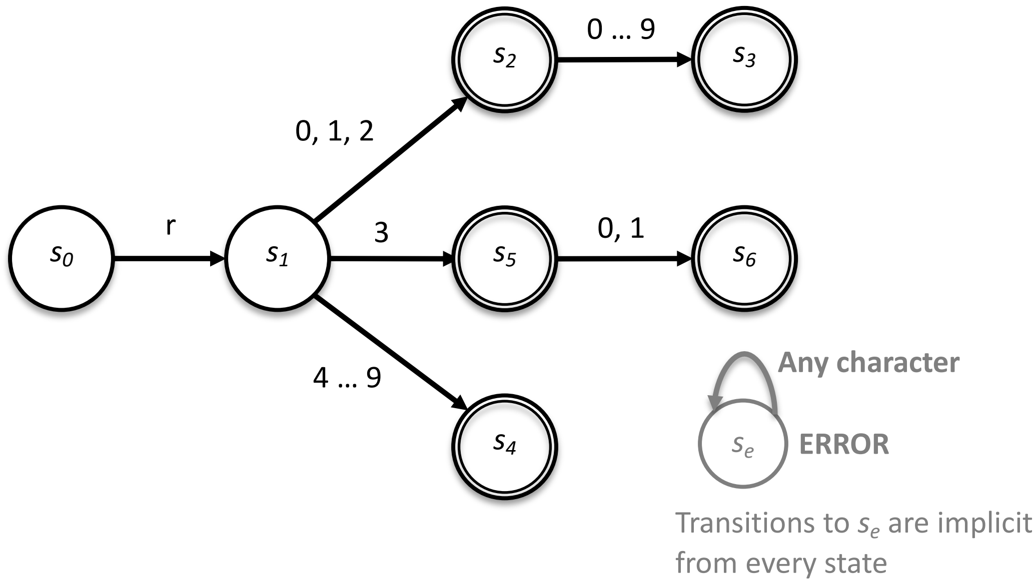
The added complexity is in the **RE**, not in the scanner†

With a scanner generated from an **RE**, using a more complex **RE** incurs no additional compile time.

† recall the Python documentation

More Complex REs

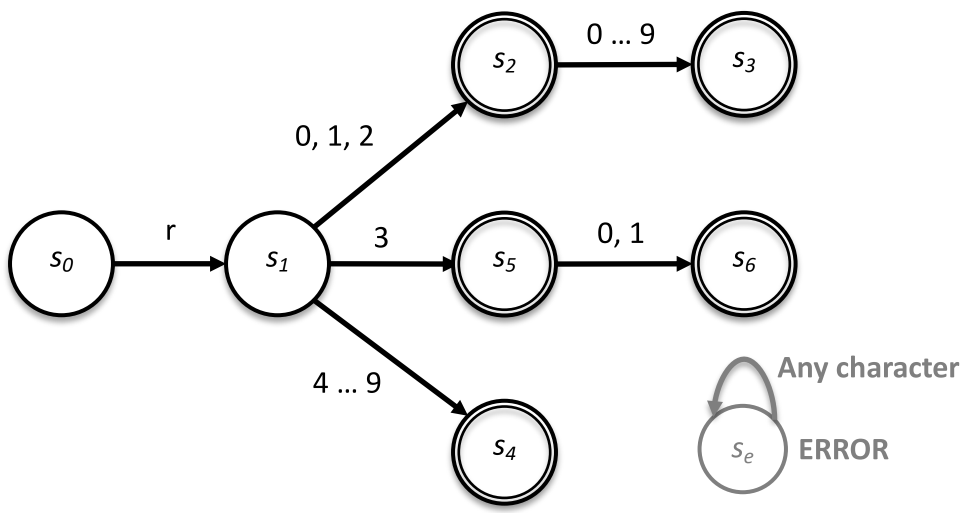
The DFA for r ((0|1|2) ([0…9] | ε) | (4|5|6|7|8|9) | (3|30|31))



- Accepts a more constrained set of register names
- Still **O**(1) cost per input character
- More states ⇒ more rows in the transition table ⇒ more memory

More Complex **RE**s

The DFA for r ((0|1|2) ([0…9] | ε) | (4|5|6|7|8|9) | (3|30|31))



Transitions to s_e are implicit from every state

- Accepts a more constrained set of register names
- Still **O**(1) cost per input character
- More states ⇒ more rows in the transition table ⇒ more memory

Automata Theory Moment

Earlier, we said we would revisit logical complement of an **RE** or a **DFA**.

To complement a **DFA**:

- Make non-final states into final states
- Make final states into non-final states

DFA then accepts any string that the original did not accept => its complement

This result is not obvious when thinking about the **RE**.

More Complex **REs**

Notice that the character classifier has many more divisions that did the earlier one. Still, it should be implementable as a function with **O**(1) cost. (see § 2.5)

The DFA for r ((0|1|2) ([0…9] | ε) | (4|5|6|7|8|9) | (3|30|31))

δ	r	0, 1	2	3	4 …9	Any Others
s_0	s_1	s_e	s_e	s_e	s_e	s_e
s_1	s_e	s_2	s_2	s_5	s_4	s_e
s_2	s_e	s_3	s_3	s_3	s_3	s_e
s_3 , s_4	s_e	s_e	s_e	s_e	s_e	s_e
s_5	s_e	s_6	s_e	s_e	s_e	s_e
s_6	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

Compressed 2 states, as well

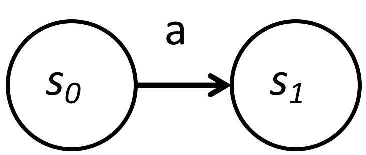
This table runs in the same skeleton scanner without changes

- To change the language, just change the table
- Still **O**(1) cost per character

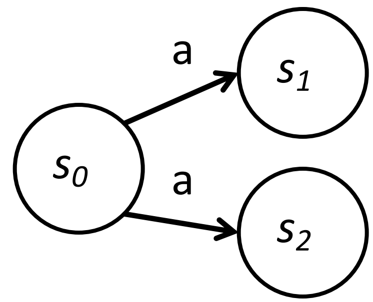
Terminology Matters

So far, we have only looked at deterministic automata, or DFAs

- **DFA** ≅ Deterministic Finite Automaton
- Deterministic means that it has only one transition out of a state on a given character



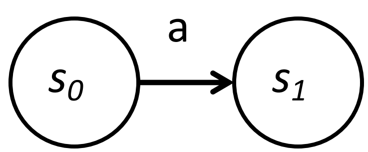
rather than



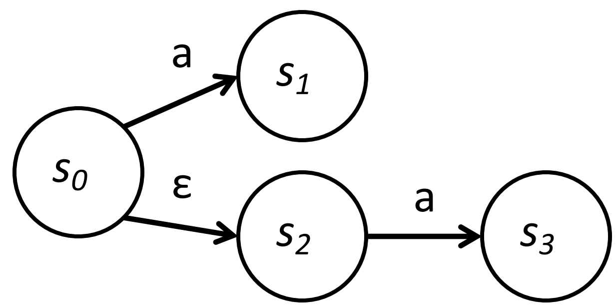
Determinism (or not)

So far, we have only looked at deterministic automata, or DFAs

- **DFA** ≅ Deterministic Finite Automaton
- Deterministic means that it has only one transition out of a state on a given character



rather than



- Can a finite automaton have multiple transitions out of a single state on the same character?
  - *Yes, we call such an* ***FA*** *a Nondeterministic Finite Automaton*
  - *And, yes, the* ***NFA*** *is one of the more odd notions in* ***CS*** *… but a useful one*
- **NFA**s and **DFA**s are equivalent
  - Sometimes, it is easier to build an **NFA** than to build a **DFA**

ε–transition does not consume an input character, which should worry us. (**O**(1) ?)

Where are we going?

We will show how to construct, for any RE *r*, a deterministic finite-state automaton that recognizes *r*

Introduce **NFAs**

Overview:

1. Simple and direct construction of a **nondeterministic finite automaton** (**NFA**) to recognize a given **RE**
   – Easy to build in an algorithmic way
   – Requires transitions on ε to combine regular subexpressions
2. Construct a **deterministic finite automaton** (**DFA**) that simulates the **NFA**
   – Use a set-of-states construction
3. Minimize the number of states in the **DFA**
   – We will look at 2 different algorithms: Brzozowski & Hopcroft
4. Generate the scanner code
   – Additional specifications needed for the actions

Optional, but worthwhile; reduces **DFA** size

The Plan for Scanner Construction

RE $\rightarrow$ NFA *(Thompson's construction)*

- Build a *nondeterministic finite automaton* (**NFA**) for each term in the **RE**
- Combine them in patterns that model the operators

NFA $\rightarrow$ DFA *(Subset construction)*

- Build a **DFA** that simulates the **NFA**

DFA $\rightarrow$ Minimal DFA

- Brzozowski's algorithm
- Hopcroft's algorithm

DFA $\rightarrow$ RE

- All pairs, all paths problem
- Union together paths from s_0 to a final state

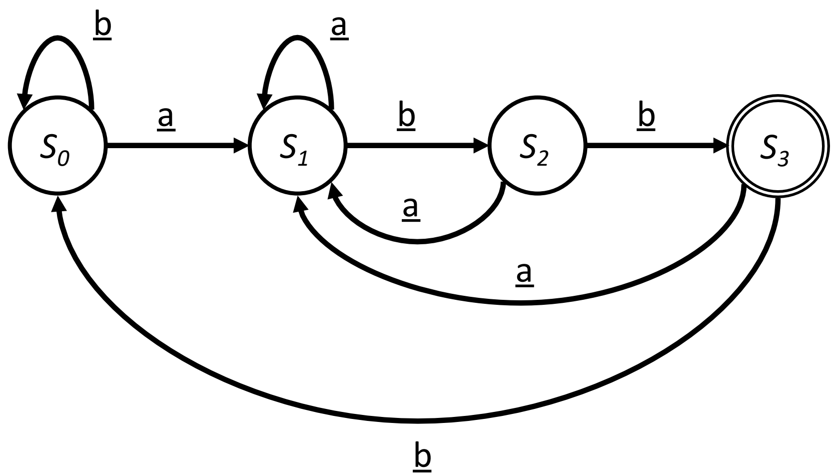
The Cycle of Constructions

Taken together, these constructions prove that DFAs, NFAs and REs are equivalent.

STOP

Example

What about a DFA for (a | b)* abb ?



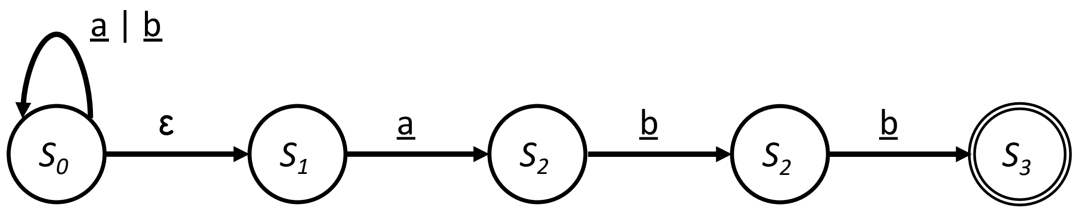
This DFA is not particularly obvious from the RE.

Each RE corresponds to one or more *deterministic finite automatons* (DFAs)

- We know a **DFA** exists for each **RE**
- The **DFA** may be hard to build directly
- Automatic techniques will build it for us …

Example as an **NFA**

Here is a simpler FA for (a | b)* abb — an NFA



(a | b)* abb

Here is an NFA for the same language

- The relationship between the **RE** and the **NFA** is more obvious
- The ε-transition pastes together two **DFA**s to form a single **NFA**
- We can rewrite this **NFA** to eliminate the ε-transition
  - ε-transitions are an odd and convenient quirk of **NFA**s
  - Eliminating this one makes it obvious that it has 2 transitions on a from s_0

Non-deterministic Finite Automata

An NFA accepts a string *x* *iff* ∃ a path though the transition graph from s_0 to a final state such that the edge labels spell *x*, ignoring ε's

- Transitions on ε consume no input
- Two models for **NFA** execution
  1. To "run" the **NFA**, start in s_0 and ***guess*** the right transition at each step †
  2. To "run" the **NFA**, start in s_0 and, at each non-deterministic choice, clone the **NFA** to purse all possible paths. If any of the clones succeeds, *accept*

Why study NFAs?

- They are an interesting and powerful abstraction
- They are the key to automating the **RE**→**DFA** construction

† See page 44 in EaC2e.

Relationship between **NFA**s and **DFA**s

DFA is a special case of an NFA

- **DFA** has no ε transitions
- **DFA**'s transition function is single-valued
- Same rules will work

DFA can be simulated with an NFA

– *Obviously*

NFA can be simulated with a DFA

(less obvious, but still true)

- Simulate sets of possible states
- Possible exponential blowup in the state space
- Still, one state per character in the input stream

⇒ **NFA & DFA are equivalent in ability to recognize languages**

Rabin & Scott, 1959