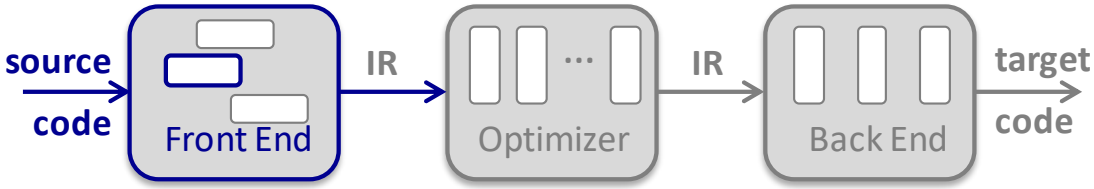
COMP 506
Rice University
Spring 2019

Building a Scanner, Part I



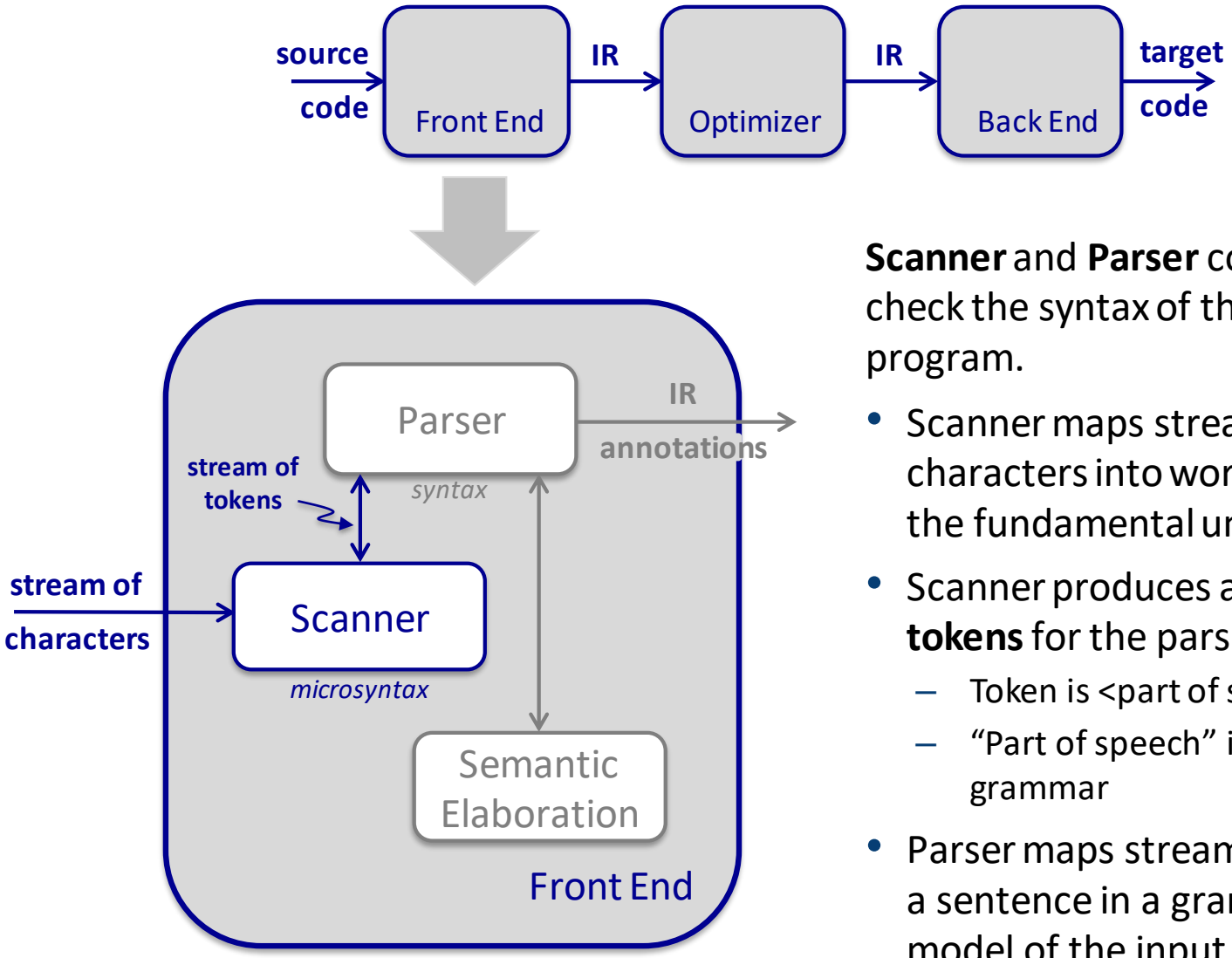
Copyright 2019, Keith D. Cooper, Linda Torczon & Zoran Budimlić, all rights reserved.

Students enrolled in Comp 506 at Rice University have explicit permission to make copies of these materials for their personal use.

Faculty from other educational institutions may use these materials for nonprofit educational purposes, provided this copyright notice is preserved

Chapter 2 in EaC3e

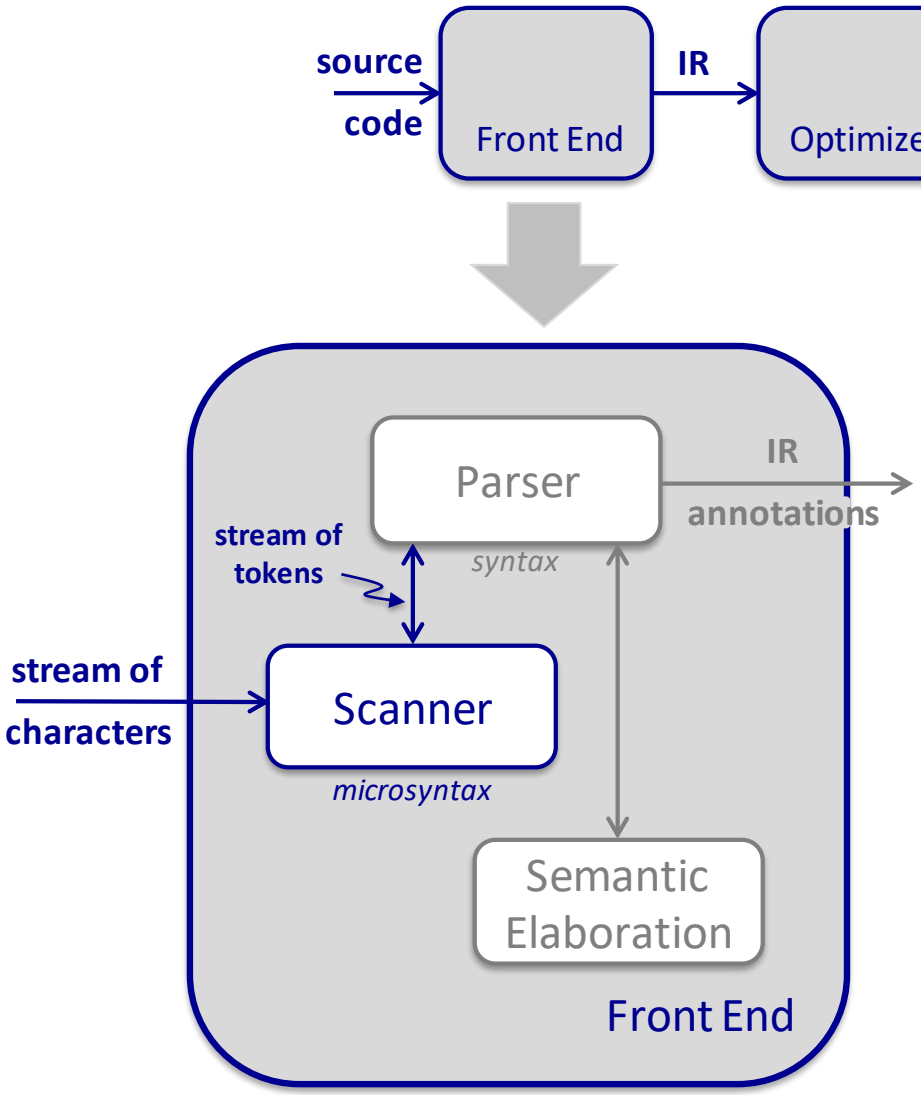
The Compiler's Front End



Scanner and **Parser** collaborate to check the syntax of the input program.

- Scanner maps stream of characters into words (words are the fundamental unit of syntax)
- Scanner produces a stream of **tokens** for the parser
  - Token is <part of speech, word>
  - "Part of speech" is a unit in the grammar
- Parser maps stream of words into a sentence in a grammatical model of the input language

The Compiler's Front End



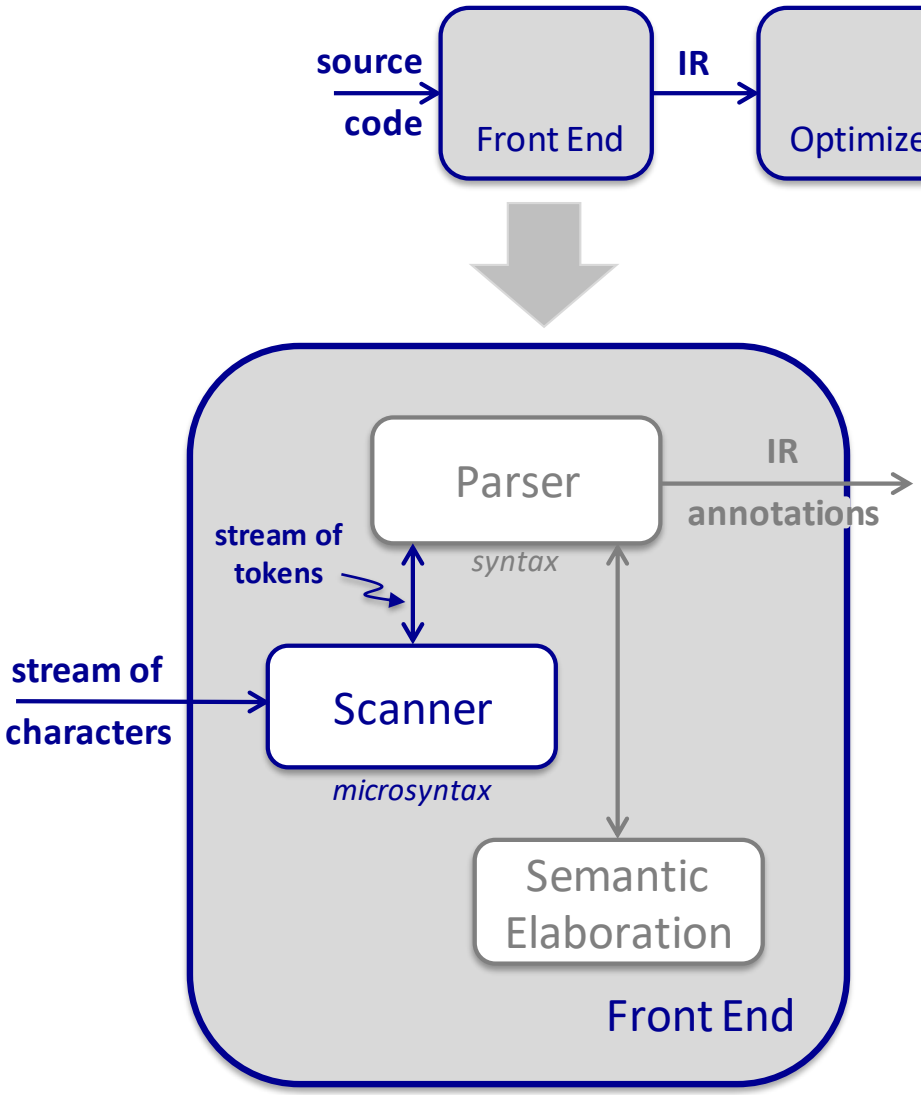
Scanner looks at every character

- Converts stream of characters to stream of tokens, or classified words:
  - <part of speech, word>
- Efficiency & scalability matter
  - Scanner is the only part of the compiler that looks at every character

Parser looks at every token

- Determines if the stream of tokens forms a sentence in the source language
- Fits tokens to some syntactic model, or grammar, for the source language

The Compiler's Front End



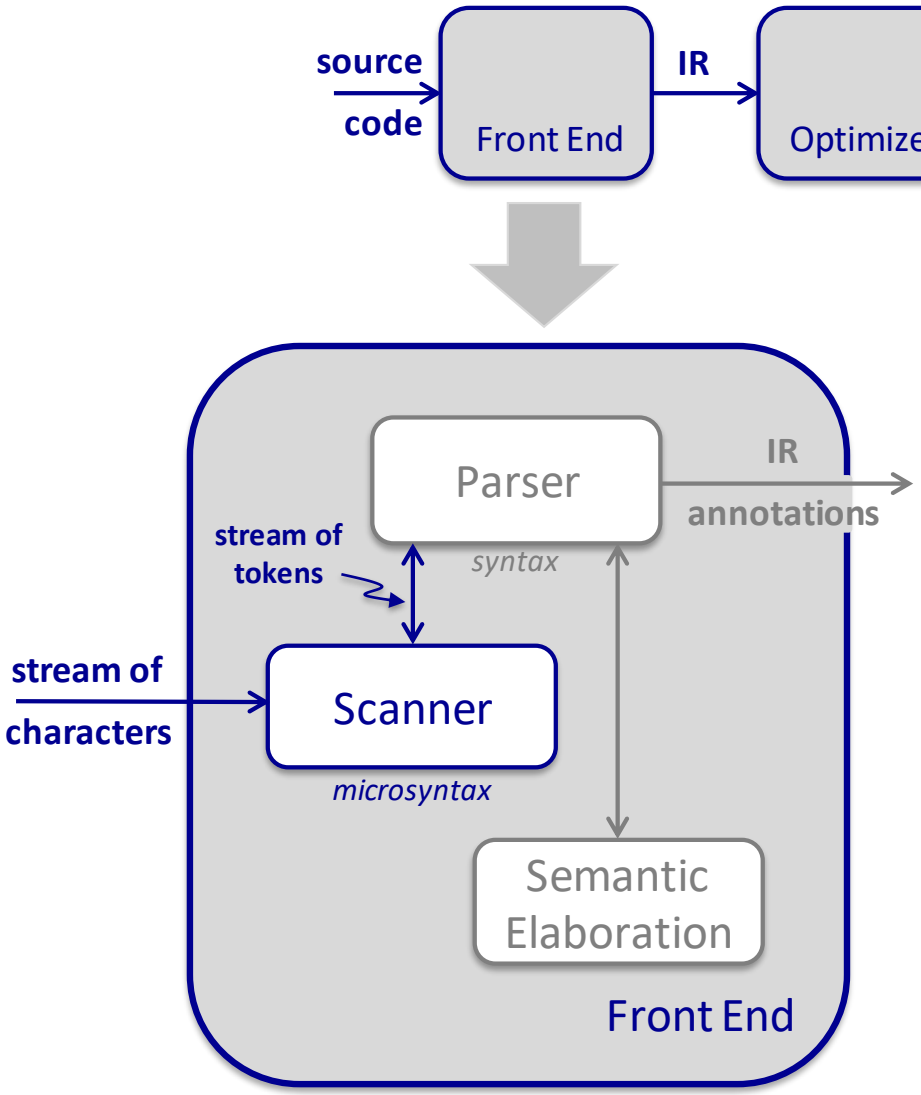
Why separate scanning & parsing?

- Primary rationale is efficiency
- Scanner identifies & classifies words by spelling (microsyntax)
- Parser constructs derivations in a grammar (syntax)
- Parsing is harder than scanning

Modern view (*less widely held*)

- Scanner-less parsers are gaining popularity, because they eliminate one more set of tools
  - Maybe we can afford the overhead
  - A little more involved (**SGLR** parsers)

Implementation Strategies



Relevant Question: How can we 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**

Why Can't We Just Use A Regex Library

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

Scanner generators use the same principles as regex libraries

- Automaton costs **O**(1) per recognized character, independent of the expression
- Scanner generators have a clean interface to generated parsers (*e.g.*, **flex→bison**)
- Read § 2.4 in EaC2e if you want the details
  - → We will skim the surface lightly in lecture

Why Can't We Just Use A Reg-

Wait a minute! What did that slide say?

> ...gexes, or regex patterns) are essentially a tiny, highly ...ming language embedded inside Python and made available through the ...odule. ...
>
> 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***

We will come back to this point next lecture

Scanner generators use the same principles as regex libraries

- Automaton costs **O**(1) per recognized character, independent of the expression
- Scanner generators have a clean interface to generated parsers (*e.g.,* **flex**→**bison**)
- Read § 2.4 in EaC2e if you want the details
  - → We will skim the surface lightly in lecture

Example

Suppose that we need to recognize the keyword "not" :

- How would we look for it?
  - ♦ All of your training, to date, tells you to call some fancy routine,
    - → such as fscanf() in **C**, the Scanner class in Java, or a regex library (Python, Java, **C++**)
  - ♦ **COMP 506** is a class about implementing languages, not just about using them, so we will look a little deeper
- In a compiler, we would build a recognizer

How about some code?

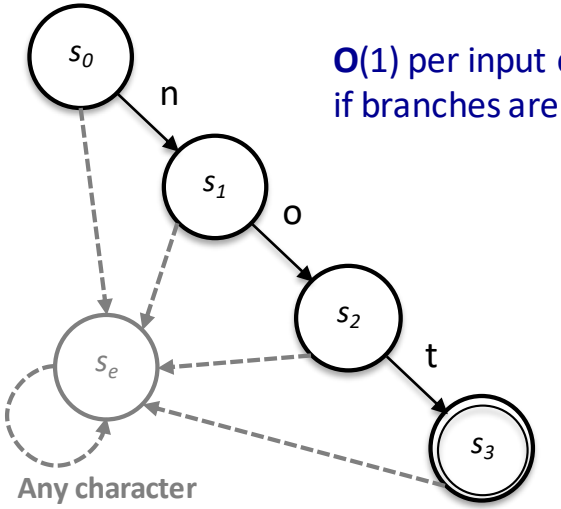
- The spelling is 'n' followed by 'o' followed by 't'
- The code, shown to the right, is as simple as the description
  - ♦ Cost is **O**(1) per character
  - ♦ Structure allows for precise error messages

```
c ← next character
If c = 'n' then {
   c ← next character
   if c = 'o' then {
      c ← next character
      if c = 't'
           then return <NOT,"not">
           else report error
      }
      else report error
   }
   else report error
```

Automata

We can represent this code as an automaton

```
c ← next character
If c = 'n' then {
   c ← next character
   if c = 'o' then {
      c ← next character
      if c = 't'
           then return <NOT,"not">
           else report error
      }
      else report error
   }
   else report error
```



O(1) per input character, if branches are **O**(1)

Transition Diagram for "not"

- Execution begins in the start state, s_0
- On an input of 'n', it follows the edge labeled 'n', and so on, …
- Transition to an error state, s_e, on any unexpected character
- Halts when out of input
- States drawn with double lines indicate success (a "final" state)

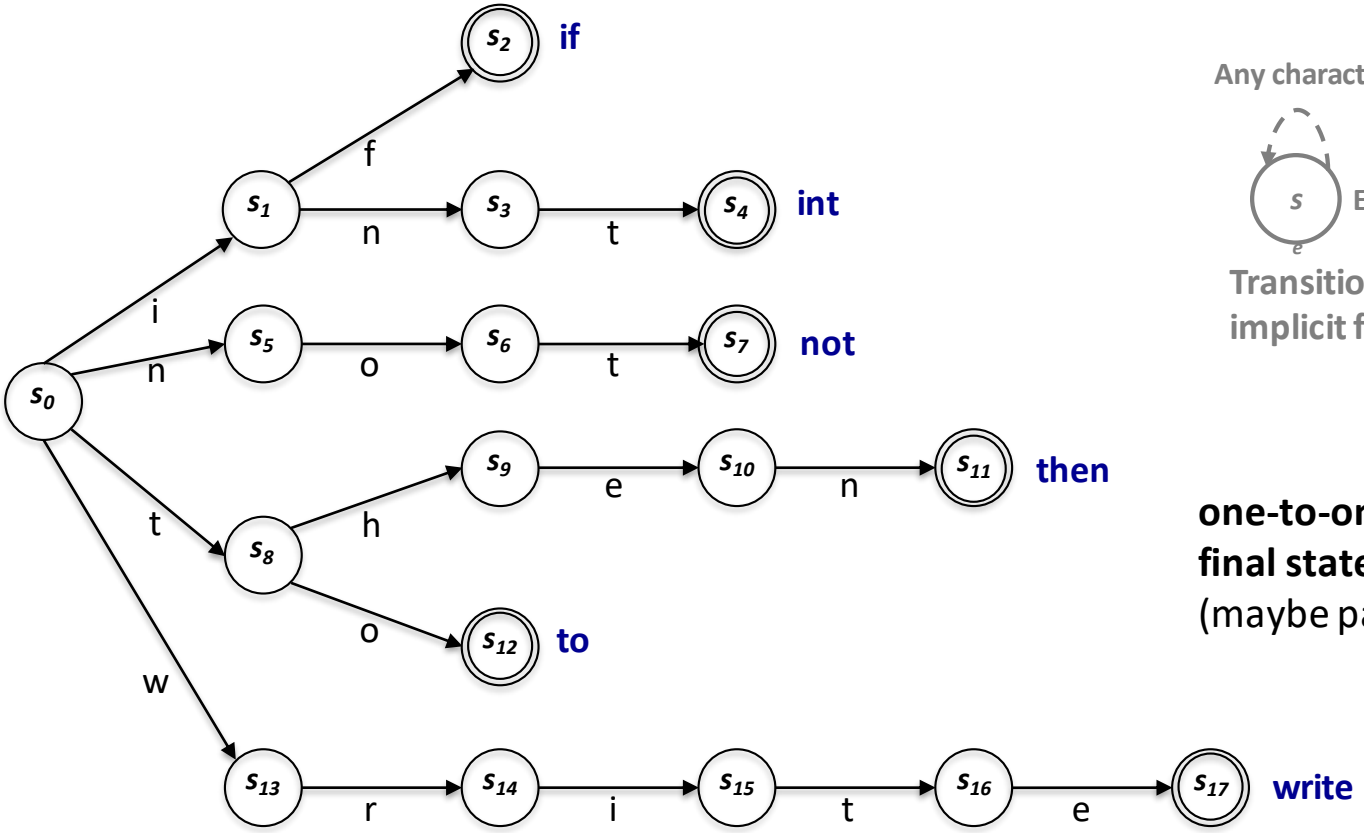
Cost is O(1) per character

See § 2.2 in EaC2e

Automata

To recognize a larger set of keywords is (relatively) easy

- Say, for example, the set of words **{ if, int, not, then, to, write }**



Transitions to s_e are implicit from every state

one-to-one map from final states to words
(maybe parts of speech)

Cost is still O(1) per character

Specifying An Automaton

There is some subtlety here. We introduced notation for concatenation and choice (or alternation).

Concatenation:
ab is **a** followed by **b**

Alternation:
a | b is either **a** or **b**

Specifying a collection of words is simple

- We list the words
  - ♦ Each word's spelling is unique and concise
  - ♦ We can separate them with the | symbol, read as "or"
- The specification

  if | int | not | then | to | write

  is a simple ***regular expression*** for the language accepted by our automaton

Every regular expression corresponds to an automaton

- We can construct, automatically, an automaton from the **RE**
- We can construct, automatically, an **RE** from any automaton
- We can convert an automaton easily and directly into code
- The automaton and the code both have **O**(1) cost per input character

So, an **RE** specification leads directly to an efficient scanner

Unsigned Integers

Notice the cyclic transition edges in the automata

In the example, each word had one spelling.

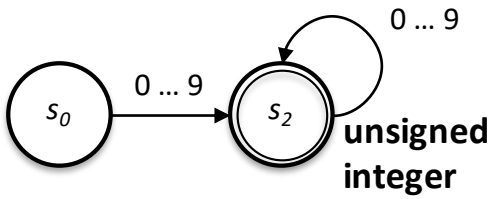
What about a single part of speech with many spellings?

Consider specifying an unsigned integer

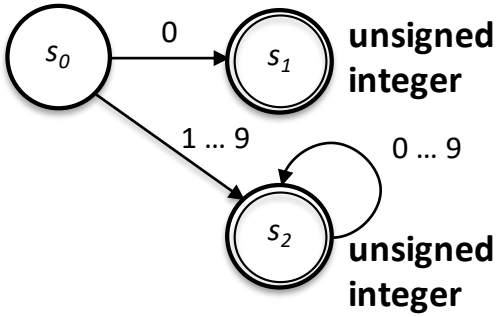
- An unsigned integer is any string of digits from the set [0...9]
- We might want to specify that no leading zeros are allowed

Is "0001" allowed?
Or "00"?

The Automata

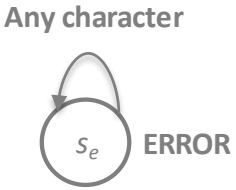


With Leading Zeros



Without Leading Zeros

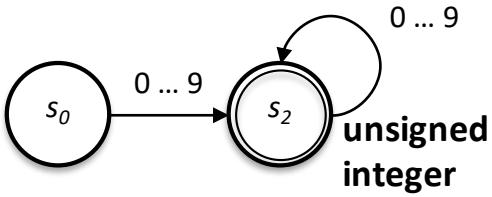
How do we write the corresponding RE?



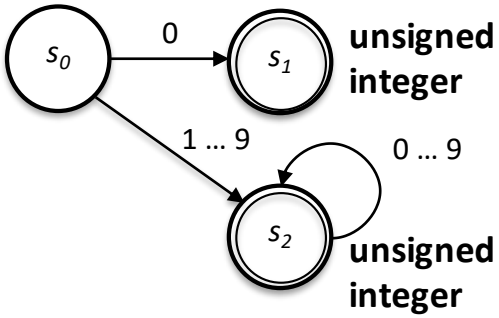
0 … 9 means any character from 0 to 9, inclusive; sometimes we will write it as 0-9

Unsigned Integers

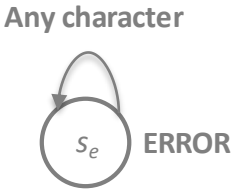
The Automata



With Leading Zeros



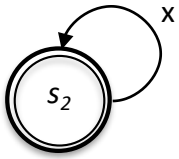
Without Leading Zeros



The Regular Expressions

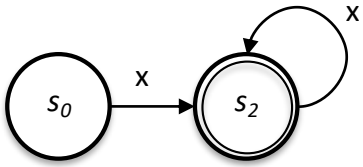
- We need a notation to represent that cyclic edge in the automaton
- The Kleene Closure

  x^* represents *zero or more instances of* 'x'



- The Positive Closure

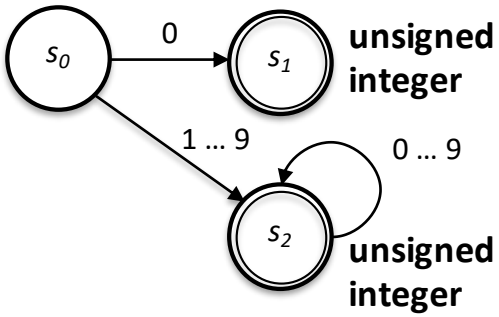
  x^+ represents *one or more instances of* 'x'



$[0\text{-}9]^+$ for leading zeros; 0 | [1-9] $[0\text{-}9]^*$ for without leading zeros

Unsigned Integers

Again, the RE corresponds directly to an automaton and an implementation




```
c ← next character
n ← 0
if c = '0'
   then return <CONSTANT,n>
   else if ('1' ≤ c ≤ '9') then {
      n ← atoi(c)
      c ← next character
      while ('0' ≤ c ≤ '9') {
        t ← atoi(c)
        n ← n * 10 + t
        c ← next character
      }
      return <CONSTANT,n>
   }
   else report error
```

The automaton and the code can be generated automatically

- The details of the constructions are given in § 2.4 of **EaC2e** or the **412** notes

Cost is still O(1) per character

Regular Expressions

We need a more formal definition for a regular expression

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 $L(\underline{x})^* = \cup_{\mathbf{0} \le k < \infty} L(\underline{x})^k$ (*Kleene Closure*)
    - → *Set of all strings that are zero or more concatenations of* $\underline{x}$
  - $\underline{x}^+$ is an **RE** denoting $L(\underline{x})^+ = \cup_{\mathbf{1} \le k < \infty} L(\underline{x})^k$ (*Positive Closure*)
    - → *Set of all strings that are one or more concatenations of* $\underline{x}$
- ε is an **RE** denoting the empty set

Many **RE**-based systems support more complex operators. Those operators are built on top of alternation, concatenation, and closure — plus, perhaps logical complement or negation. Complement is easy and efficient; reverse the final and non-final states.

Regular Expressions

How do these operators help?

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** (spelling of 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 $L(\underline{x})^* = \cup_{\mathbf{0} \le k < \infty} L(\underline{x})^k$
  - $\underline{x}^+$ is an **RE** denoting $L(\underline{x})^+ = \cup_{\mathbf{1} \le k < \infty} L(\underline{x})^k$
    → *We can use closure to write finite descriptions of infinite, but countable, sets*
- ε is an **RE** denoting the empty set
    → *In practice, the empty string is often useful*

Regular Expressions

Let the notation **[0-9]** be shorthand for **(0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9)**

Examples

Positive integer	**[0-9] [0-9]***
or	**[0-9]$^+$**
No leading zeros	**0 | [1-9] [0-9]***
Algol-style Identifier	**([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]$^+$**

Each of these **RE**s corresponds to an automaton and an implementation.

From RE to Scanner

We can use results from automata theory to construct scanners directly from REs

- There are several ways to perform this construction
- Classic approach is a two-step method.
  1. Build automata for each piece of the **RE** using a simple template-driven method
     → *Build a specific variation on an automaton that has transitions on ε and non-deterministic choice (multiple transitions from a state on the same symbol)* — An **NFA**
     → *This construction is called "Thompson's construction"*
  2. Convert the newly built automaton into a deterministic automaton
     → *Deterministic automaton has no ε-transitions and all choices are single-valued* — A **DFA**
     → *This construction is called the "subset construction"*
- Given the deterministic automaton, we can run a minimization algorithm on it to reduce the number of states
  → *Minimization is a **space optimization**. Both the original automaton and the minimal one take **O**(1) time per character*

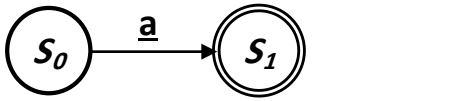
Ken Thompson, CACM, 1968

Thompson's Construction (in 2 slides)

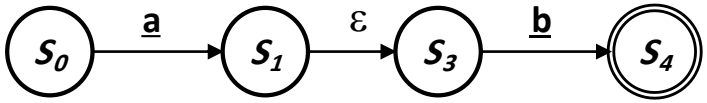
(see § 2.4.2)

The Key Idea

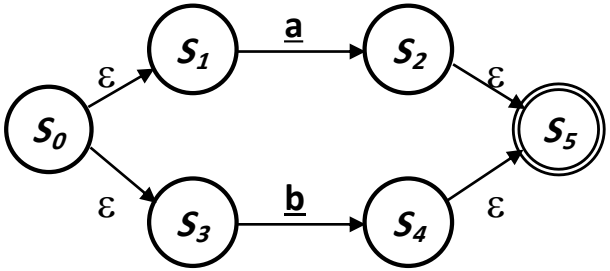
- For each **RE** symbols and operator, we have a small template
- Build them, in precedence order, and join them with ε-transitions



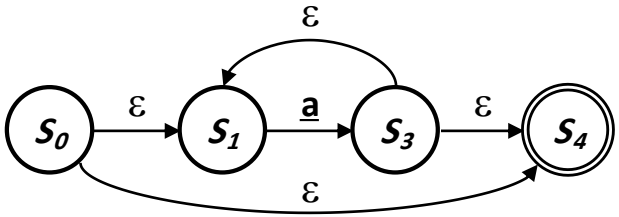
NFA for a



NFA for ab



NFA for a | b



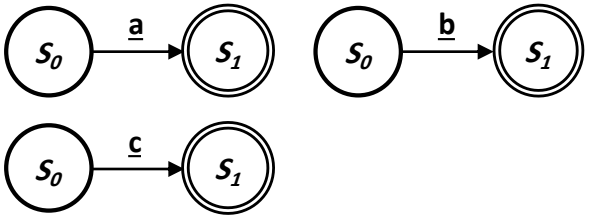
NFA for a^*

Precedence in REs: Parentheses, then closure, then concatenation, then alternation

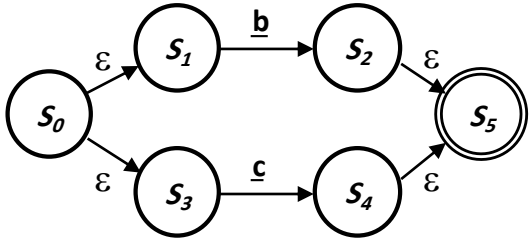
Thompson's Construction (in 2 slides)

Let's build an NFA for a (b | c)*

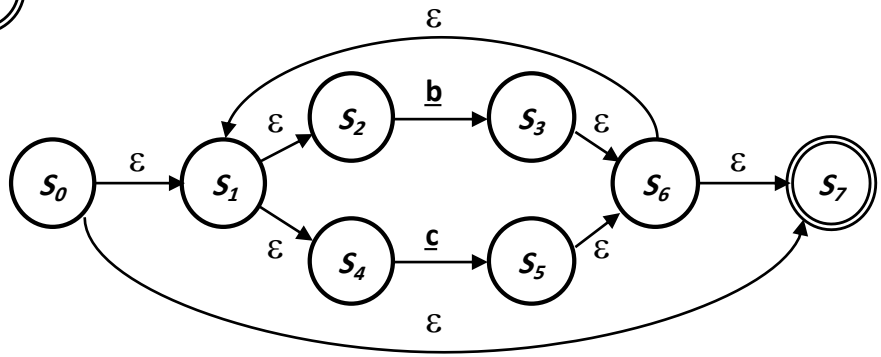
1. a, b, & c



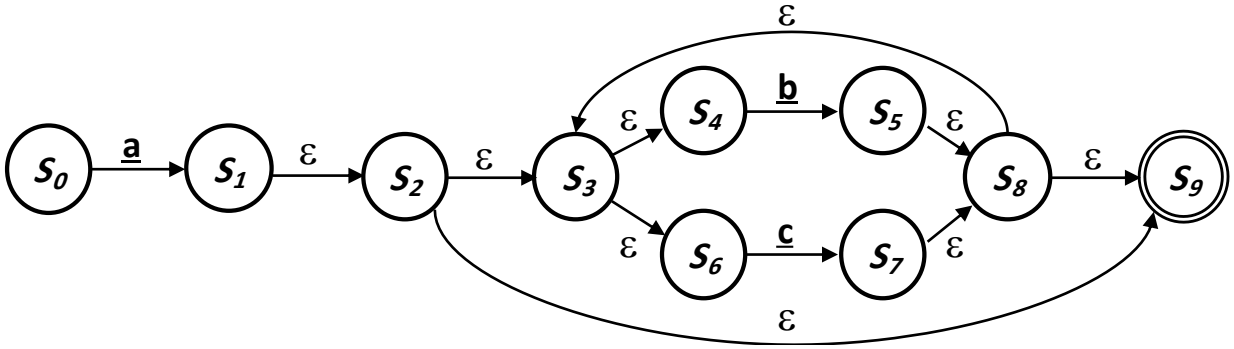
2. b | c



3. (b| c)*



4. a (b| c)*



Precedence in REs: Parentheses, then closure, then concatenation, then alternation

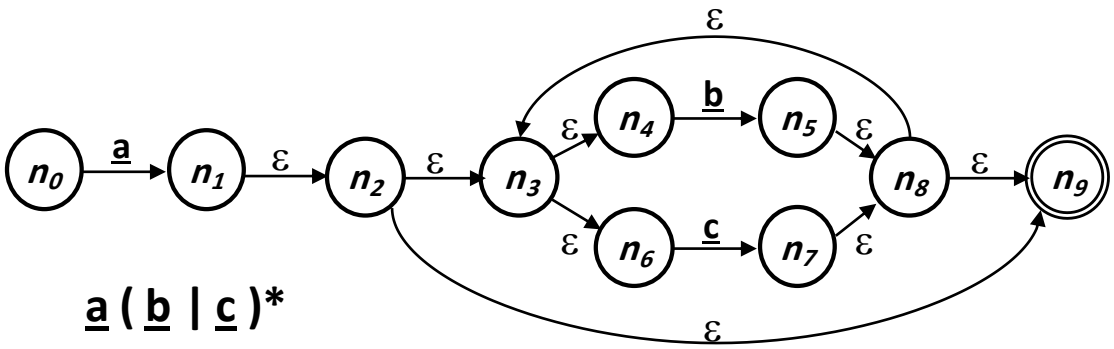
Subset Construction

(see § 2.4.3)

The Concept

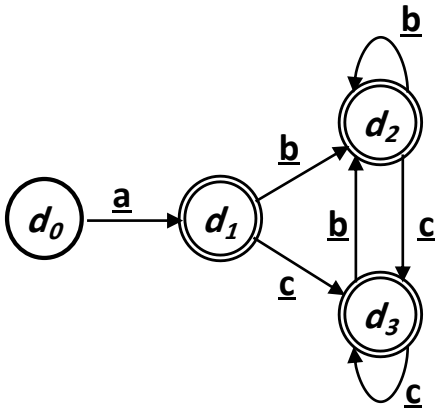
- Build a simpler automaton (no **ε**-transitions, no multi-valued transitions) that simulates the behavior of the more complex automaton
- Each state in the new automaton represents a set of states in the original

NFA



a (b | c)*

DFA



DFA	NFA
d_0	n_0
d_1	$n_1\ n_2\ n_3\ n_4\ n_6\ n_9$
d_2	$n_5\ n_8\ n_9\ n_3\ n_4\ n_6$
d_3	$n_7\ n_8\ n_9\ n_3\ n_4\ n_6$

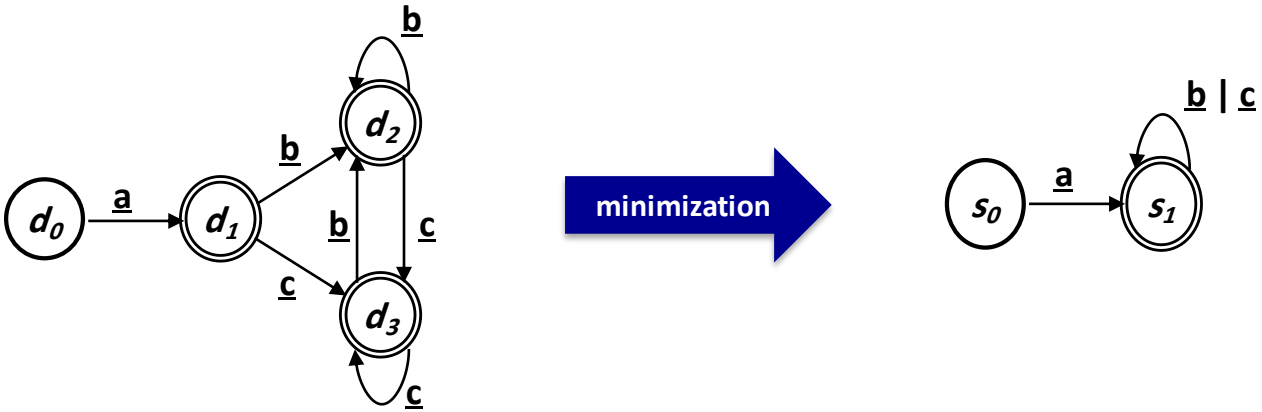
Mapping between NFA and DFA states

Minimization

(see § 2.4.4 & 2.6.3)

DFA Minimization algorithms work by discovering states that are equivalent in their contexts and replacing multiple equivalent states with a single one

- Minimization reduces the number of states, but does not change the costs



Minimal DFA State	s_0	s_1
Original DFA States	d_0	d_1 , d_2 , d_3

Implementing an Automaton

(see § 2.5)

A common strategy is to simulate the DFA's execution

- Skeleton parser + a table that encodes the automaton

```
state ← s0
char ← NextChar( )
while (char ≠ EOF) {
    state ← δ[state,char]
    char ← NextChar( )
}
if (state is a final state)
  then report success
   else report an error
```

Simple Skeleton Scanner

δ	a	b	c
s_0	s_1	s_e	s_e
s_1	s_e	s_1	s_1

Transition table for our minimal DFA

- The scanner generator constructs the table
- The skeleton parser does not change (much)

Implementing an Automaton (see § 2.5)

A common strategy is to simulate the DFA's execution

- Skeleton parser + a table that encodes the automaton

```
state ← s0
char ← NextChar( )
while (char ≠ EOF) {
    state ← δ[state,char]
    char ← NextChar( )
}
if (state is a final state)
  then report success
   else report an error
```

Simple Skeleton Scanner

Notice that the skeleton scanner uses a **while loop** and **an array access** to replace the **if-then-else** from our earlier examples.

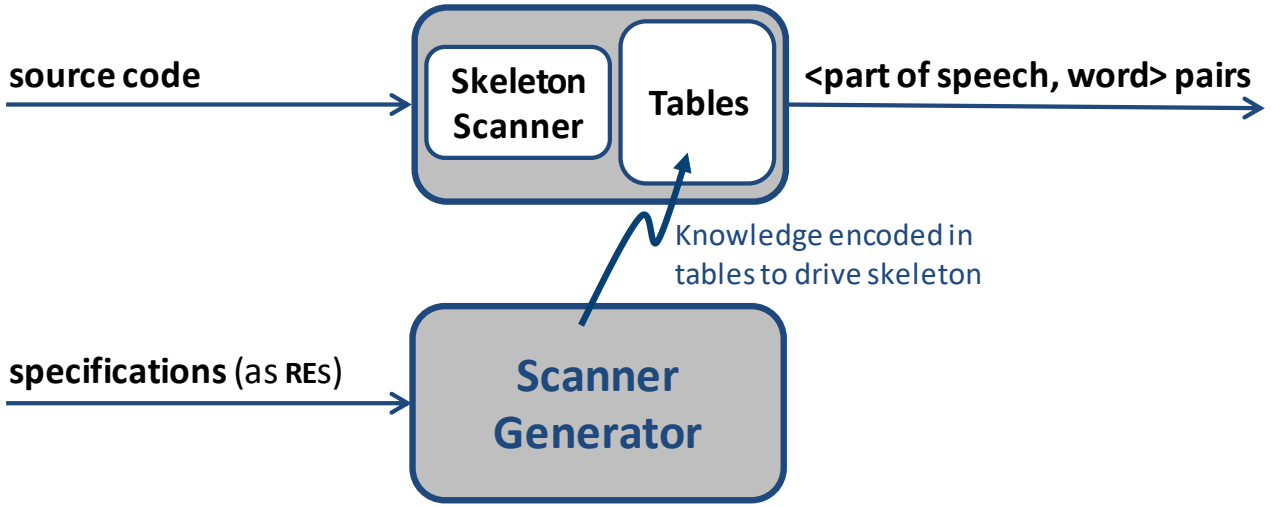
This structure translates into an **end-of-loop jump** and a **branch for the test**—an easily predictable branch. (1 cycle vs. 4 or 5 ?)

The branches in the **if-then-else** are **hard to predict**, making that code much slower than the skeleton scanner.

Students are often surprised to discover how much time scanning and parsing takes.

- The scanner generator constructs the table
- The skeleton scanner does not change (much)

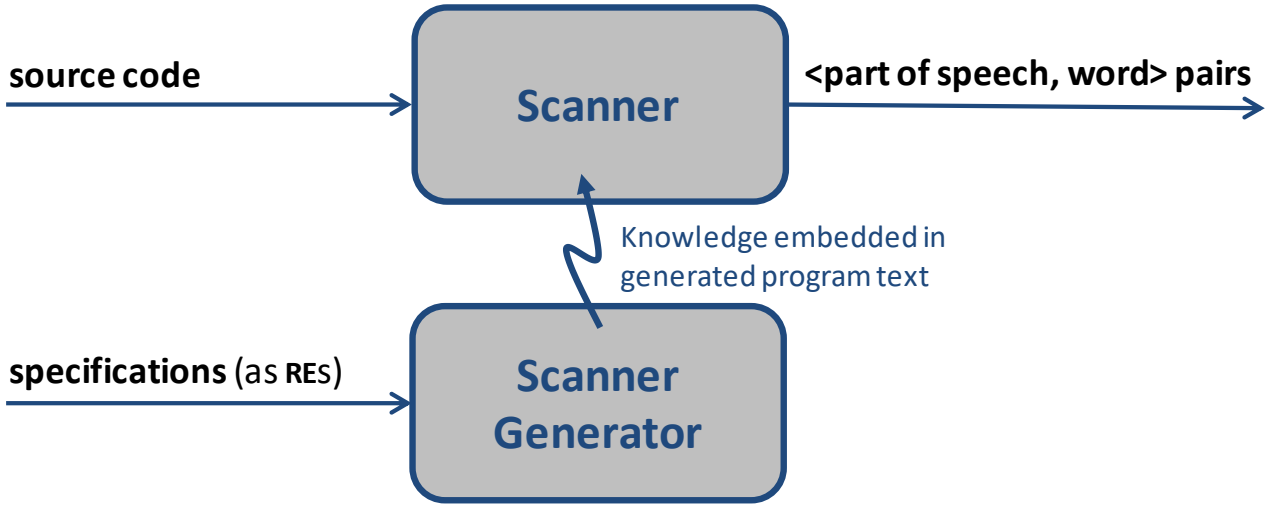
Automatic Scanner Construction



Scanner Generator

- Takes in a specification written as a collection of regular expressions
- Combines them into one **RE** using alternation ("|")
- Builds the minimal automaton (**§ 2.4, 2.6.2**)
- Emits the tables to drive a skeleton scanner (**§ 2.5**)

Automatic Scanner Construction



Scanner Generator

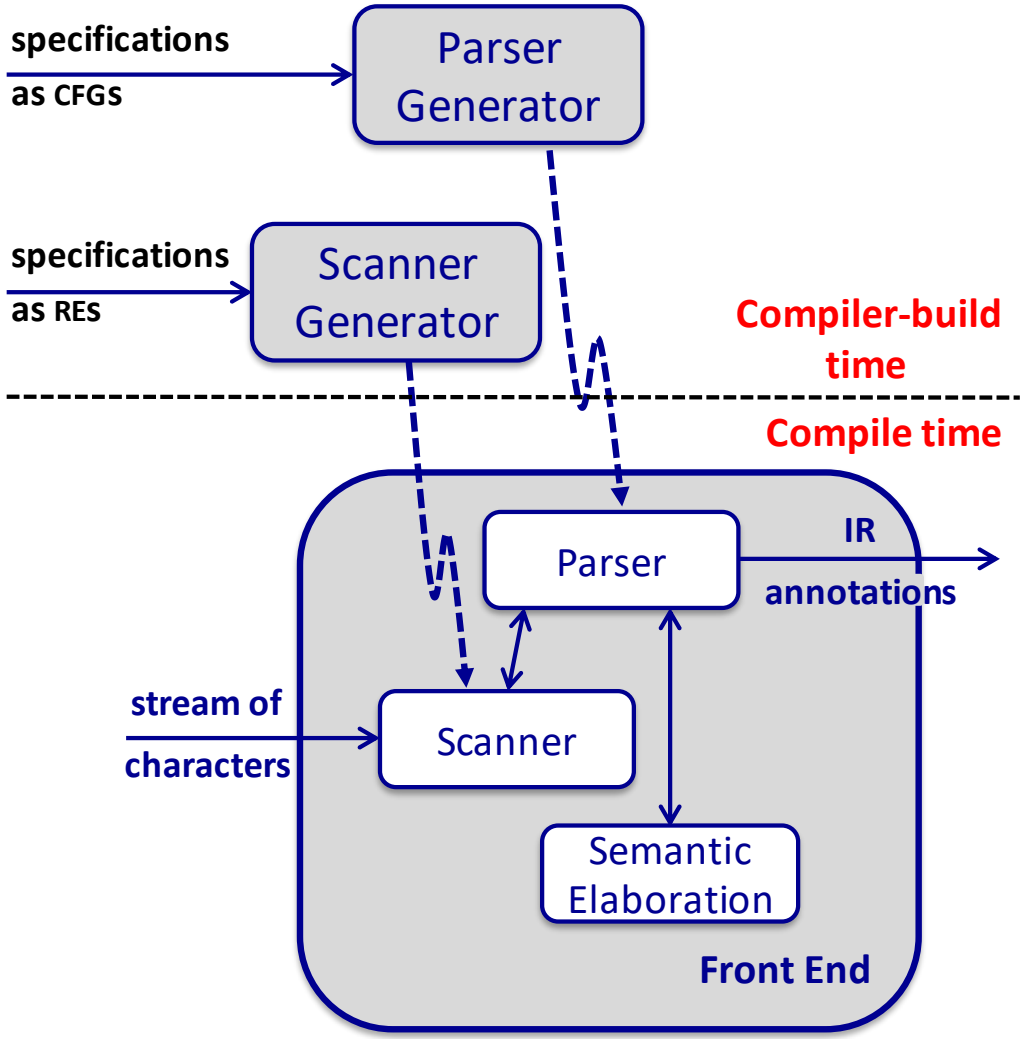
- As alternative, the generator can produce code rather than tables
- Direct-coded scanners are ugly, but often faster than table-driven scanners
- Other than speed, the two are equivalent

⇒ Use the scanner generator available to you

Automatic Generation of Scanners and Parsers

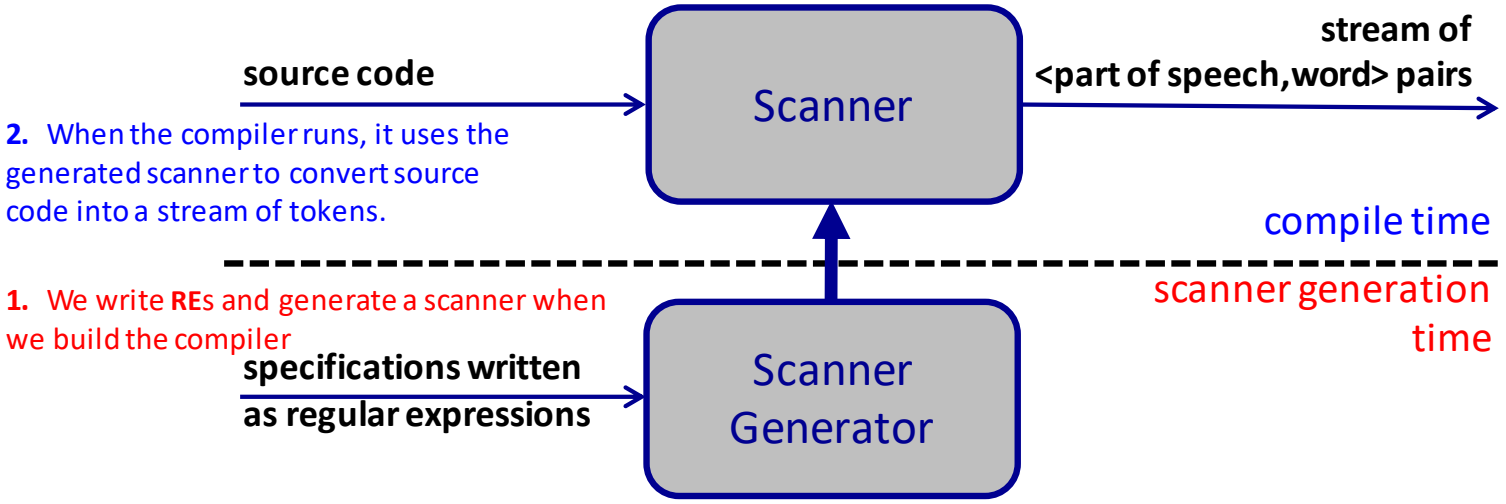
Three time frames

- At **design time**, the compiler writer writes specifications for the microsyntax (spelling) and the syntax (grammar)
- At **build time**, the tools convert specifications to code and compile that code to produce the actual compiler
- At **compile time**, the user invokes the compiler to translate an application into an executable form



In lab 1, you will use a scanner generator & a parser generator

Automatic Scanner Construction



Scanner and parser generators operate at compiler-build time

- The compiler writer creates a input file that describes the microsyntax (spelling) of the words in the source language *
- When the compiler is built, the scanner generator creates the scanner
  - ♦ Builds an automaton, converts it to a table or code *
- When the compiler runs, it invokes the generated scanner to tokenize input
  - ♦ Scanner returns a stream of <*part of speech*,*word*> pairs to the parser

Automatic Scanner Generation

The Point (for Scanners)

- The technology lets us write a set of **RE**s and generate a good scanner
- Scanner generator builds the **NFA**, the **DFA**, the minimal **DFA**, and then writes out a table-driven or direct coded scanner
- The tools reliably produce fast, robust scanners

For most modern language features, this works and works well

- You should think twice before introducing a language feature that defeats a **DFA**-based scanner
- We have seen some over time; they have not been particularly useful
  - ♦ Insignificant blanks in **FORTRAN**, non-reserved keywords in **PL/I**

Of course, not everything fits into the regular language framework

⇒ which is why we need parsers …

Summary

Automating Scanner Construction

1. Write down the **RE**s for the input language and connect them with "|"
2. Build a big, simple **NFA**
3. Build the **DFA** that simulates the **NFA**
4. Minimize the number of states in the **DFA**
5. Generate an implementation from the minimal **DFA**

Scanner Generators

- lex, flex, jflex, and such all work along these lines
- Algorithms are well-known and well-understood
- Key issues are: *finding longest match rather than first match*, and *engineering the interface to the parser*
- You could build a scanner generator in a weekend

"Find the longest match" is where **O**(1) per character can turn into **O**(n). More next lecture.