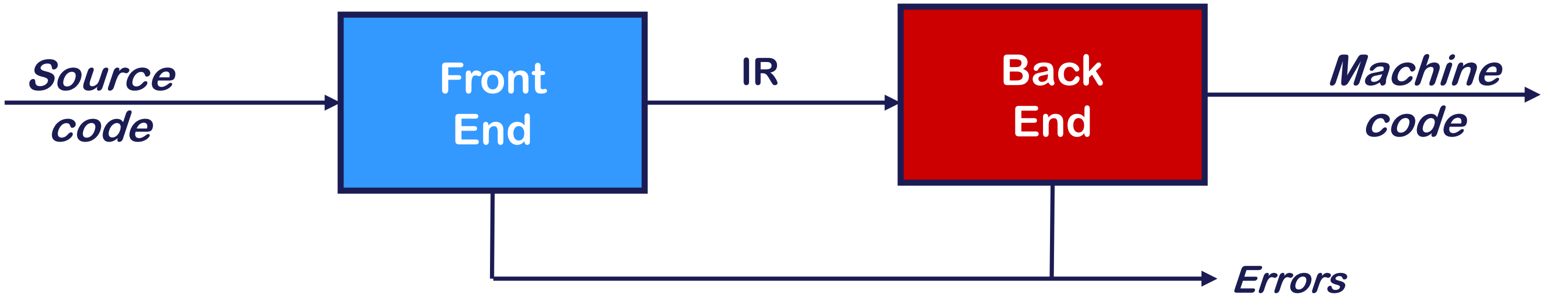
Scanner

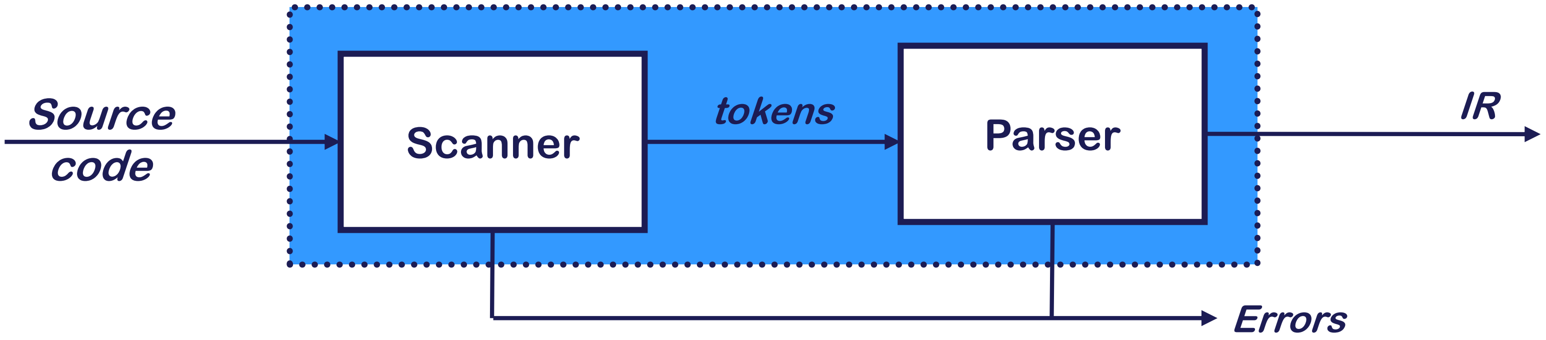
Hwansoo Han

Traditional Two-pass Compiler



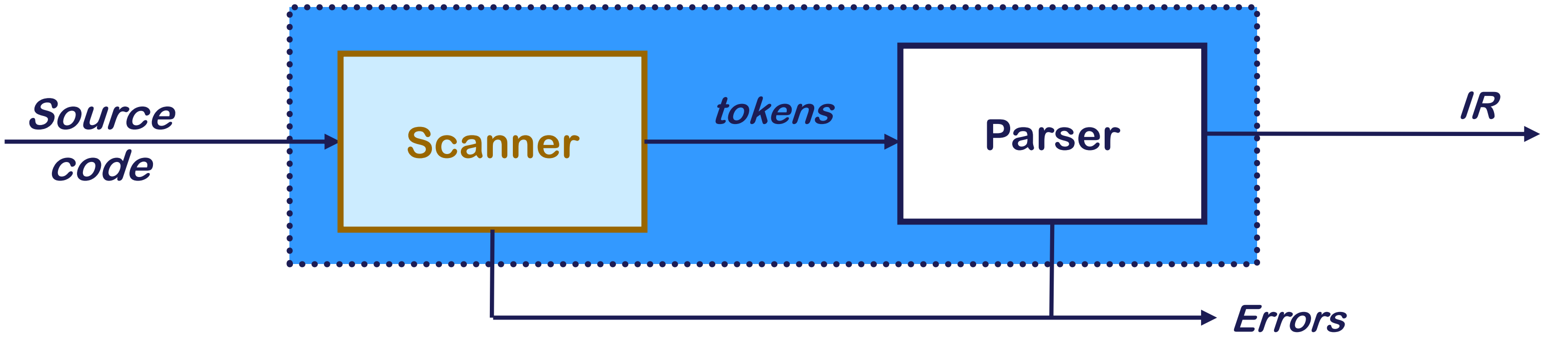
- **High level functions**
  - Recognize legal program, generate correct code (OS & linker can accept)
  - Manage the storage of all variables and code
- **Two passes**
  - Use an intermediate representation (IR)
  - Front end maps legal source code into IR - O(n) or O(n log n)
  - Back end maps IR into target machine code - typically NP-complete
  - Admits multiple front ends & multiple passes (*better code*)

Front End



- **Responsibilities**
  - Recognize legal (& illegal) programs
  - Report errors in a useful way
  - Produce IR & preliminary storage map
  - Shape the code for the back end
  - Much of front end construction *can be automated*

Front End - Scanner



- **Scanner**
  - Maps character stream into words (basic units of syntax)
  - Produces tokens — a word & its part of speech

    x = x + y ;

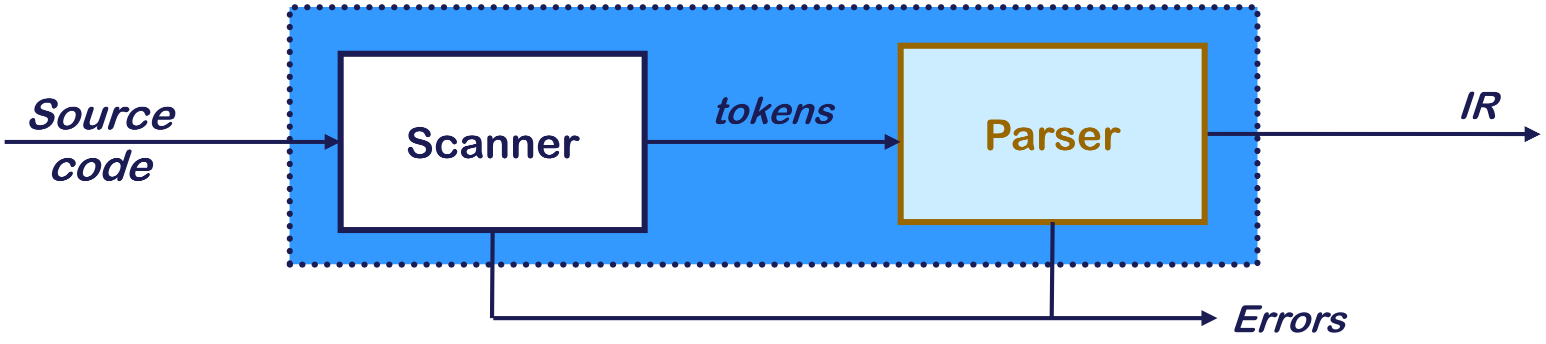
    becomes <id, x> <EQ, => <id, x> <OP, +> <id, y> <EoE, ;>

    <part of speech, word> ≅ <token type, lexeme>

  - Typical tokens include *number*, *identifier*, +, –, *new*, *while*, *if*

    Scanner eliminates white space
  - Produced by automatic scanner generator

Front End - Parser



- **Parser**
  - Recognizes context-free syntax
  - Guides context-sensitive ("semantic") analysis
    E.g. type checking
  - Builds IR for source program
  - Produced by automatic parser generators

Front End - example (1)

- **Context-free syntax can be put to better use**

1.	*goal*	$\rightarrow$	*expr*
2.	*expr*	$\rightarrow$	*expr op term*
3.		|	*term*
4.	*term*	$\rightarrow$	number
5.		|	id
6.	*op*	$\rightarrow$	+
7.		|	-

S = *goal*

T = { number, id, +, - }

N = { *goal*, *expr*, *term*, *op* }

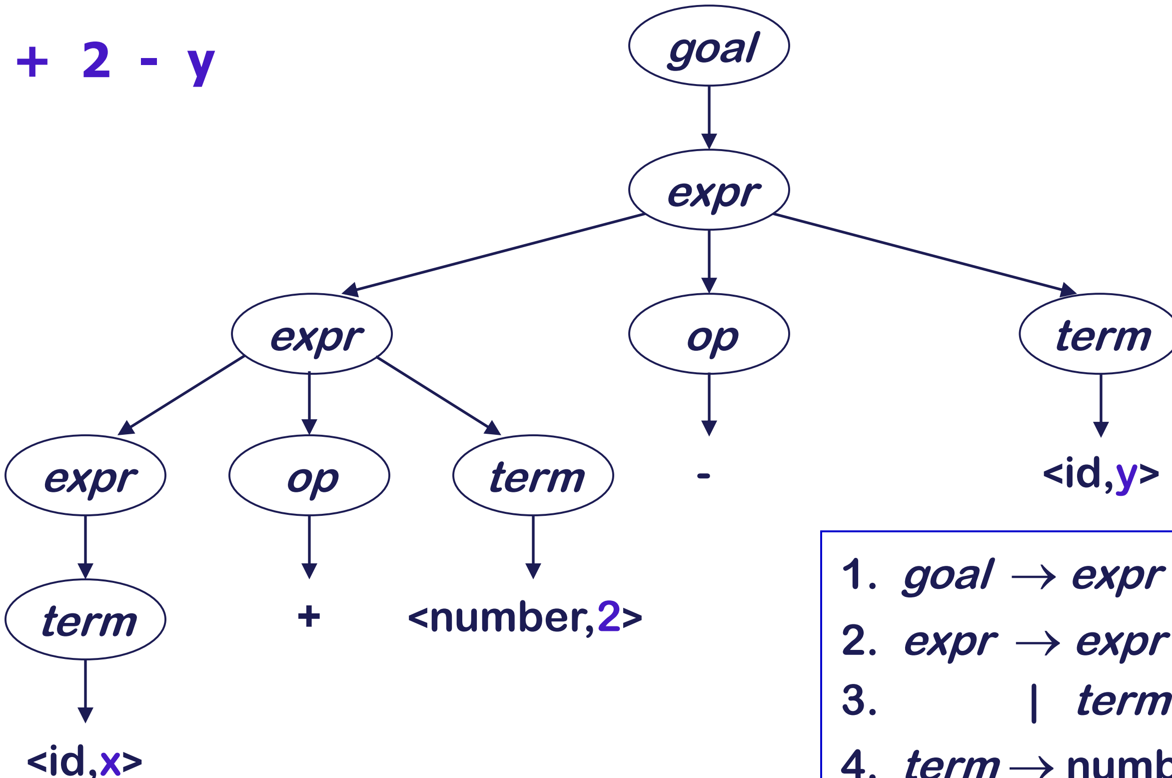
P = { 1, 2, 3, 4, 5, 6, 7}

- This grammar defines simple expressions with addition & subtraction over "number" and "id"
- This grammar, like many, falls in a class called "context-free grammars", abbreviated CFG

Front End - example (2)

- **A parse can be represented by a tree (*parse tree* or *syntax tree*)**

x + 2 - y

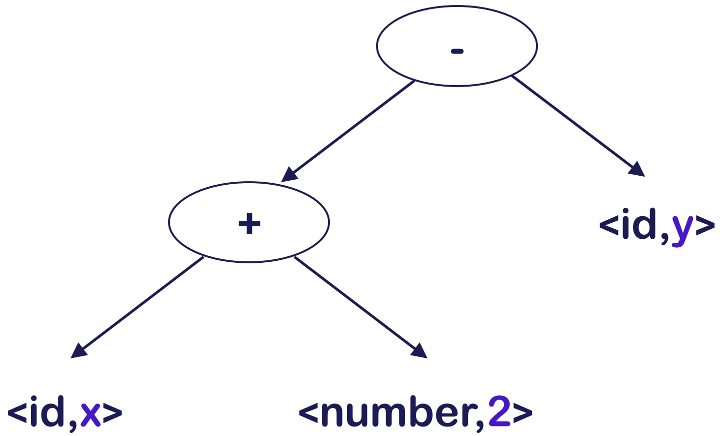


This contains a lot of unneeded information.

1.	*goal*	$\rightarrow$	*expr*
2.	*expr*	$\rightarrow$	*expr op term*
3.		|	*term*
4.	*term*	$\rightarrow$	<u>number</u>
5.		|	<u>id</u>
6.	*op*	$\rightarrow$	+
7.		|	-

Front End - example (3)

- **Compilers often use an *abstract syntax tree (AST)***

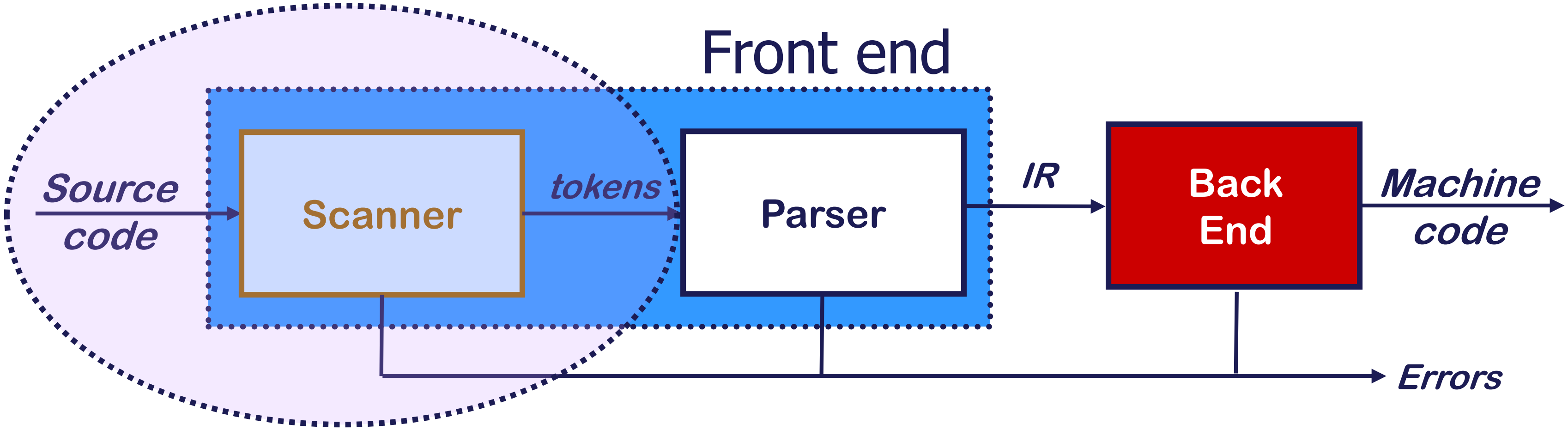


The AST summarizes grammatical structure, without including detail about the derivation

This is much more concise

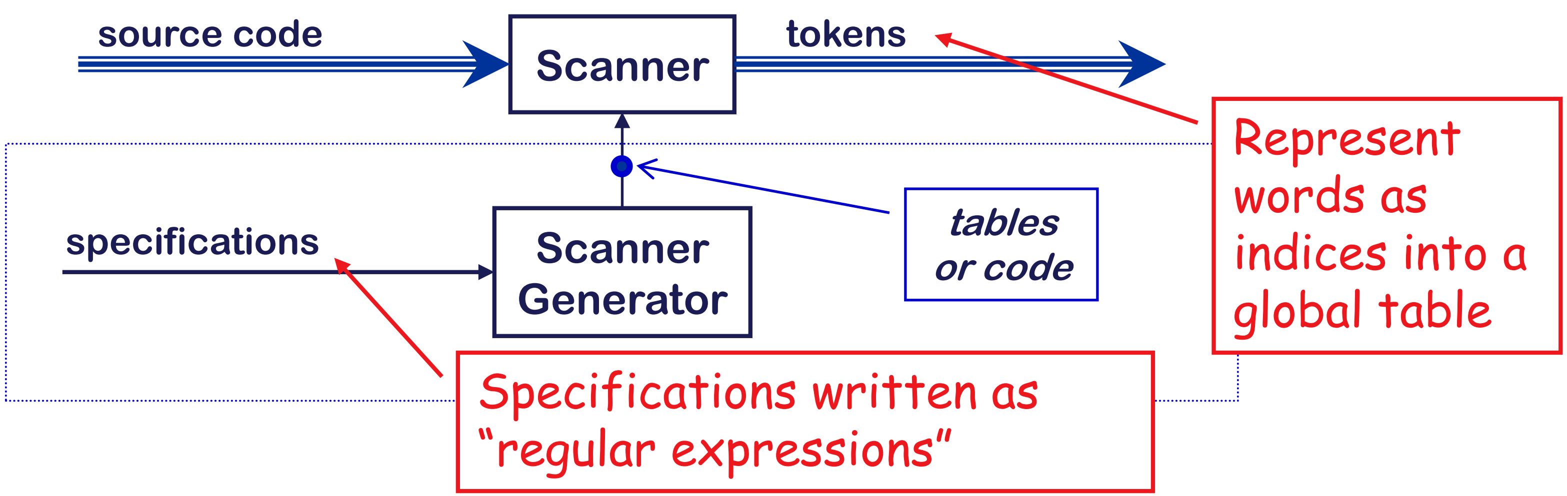
ASTs are one kind of *intermediate representation (IR)*

Scanner


Front end
Source code
Scanner
tokens
Parser
IR
Back End
Machine code
Errors

Scanner Generator

- **We want to avoid writing scanners by hand**
  - The scanner is the first stage in the front end
  - Specifications for tokens can be given using *regular expressions*
  - Build tables and code from a DFA

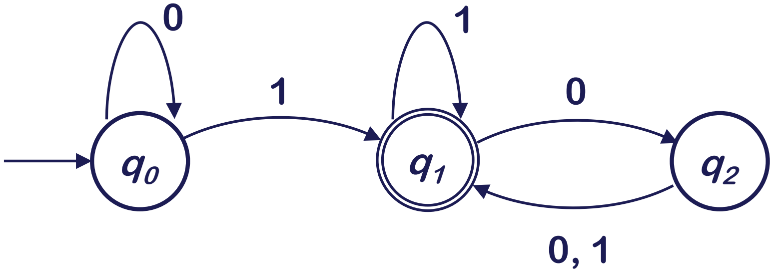


Finite Automata

- **Deterministic Finite Automaton (DFA)**
  - State transition is determined by an input symbol
  - Number of states is finite

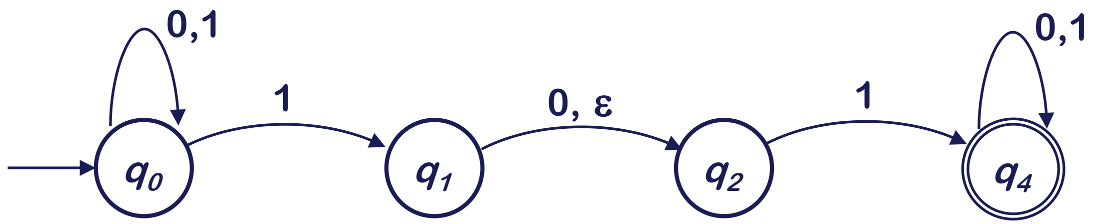
1. $Q = \{q_0, q_1, q_2\}$ // set of states
2. q_0 is the start state // specification of the start state
3. $F = \{q_1\}$ // set of final states (accepting states)
4. $\Sigma = \{0, 1\}$ // set of input symbols
5. δ is described as // transition function

	0	1
q_0	q_0	q_1
q_1	q_2	q_1
q_2	q_1	q_1



Finite Automata

- **Non-deterministic Finite Automaton (NFA)**
  - Multiple choices for the next state on an input symbol
  - State transition without consuming input symbol (ε-move)
  - NFA has the same expressive power as DFA



Regular Languages & Operations

Regular Language (over alphabet Σ)

- Regular operations on Languages *A* and *B*
  - Union: $A \cup B = \{x \mid x \in A \text{ or } x \in B\}$
  - Concatenation: $A \circ B = \{xy \mid x \in A \text{ and } y \in B\}$
  - Star: $A^* = \{x_1x_2\ldots x_k \mid k \geq 0 \text{ and each } x_i \in A\}$
    - * operation is also called Kleene star, Kleene operator, or Kleene closure

- Let the alphabet $\Sigma = \{a, b, \ldots, z\}$, $A = \{good, bad\}$ and $B = \{cat, dog\}$
  - $A \cup B$ = {good, bad, cat, dog}
  - $A \circ B$ = {goodcat, gooddog, badcat, baddog}
  - A^* = {ε, good, bad, goodgood, goodbad, badgood, badbad, goodgoodgood, goodgoodbad, goodbadgood, goodbadbad, badgoodgood, badgoodbad, badbadgood, badbadbad, …}

Regular Expressions

Regular Expression

- ε is a *regular expression* denoting the set $\{\varepsilon\}$
  - which is a language containing a single string – the empty string
- If $\underline{a} \in \Sigma$, then $\underline{a}$ is a *regular expression* denotation for $\{\underline{a}\}$
- If x and y are *regular expressions* denoting L(x) and L(y) then
  - x|y is a *regular expression* denoting $L(x) \cup L(y)$
  - xy is a *regular expression* denoting L(x)L(y)
  - x^* is a *regular expression* denoting $L(x)^*$

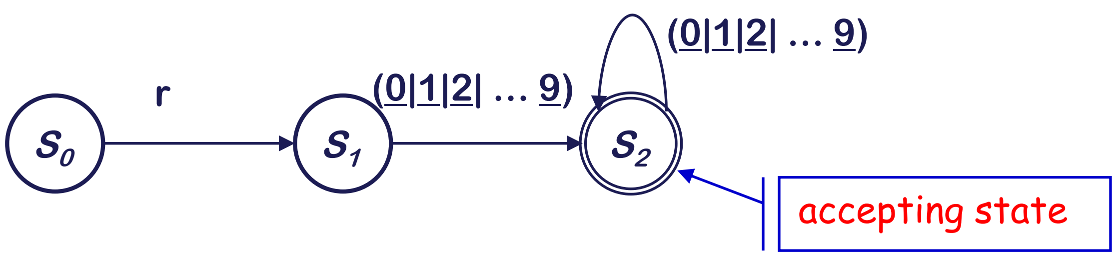
Regular Expression – Example

- **RE for recognizing register names**

  Register $\rightarrow$ r ($\underline{0}|\underline{1}|\underline{2}|$ … $|\ \underline{9}$) ($\underline{0}|\underline{1}|\underline{2}|$ … $|\ \underline{9})^*$

  - Allows registers of arbitrary number
  - Requires at least one digit

- **RE corresponds to a recognizer (or DFA)**

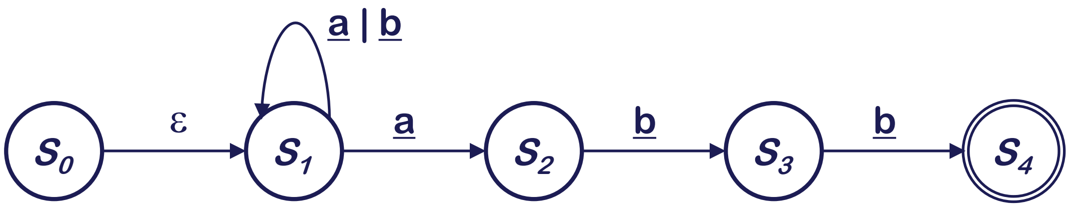


Recognizer for *Register*

Transitions on other inputs go to an error state, s_e

Non-deterministic Finite Automata (NFA)

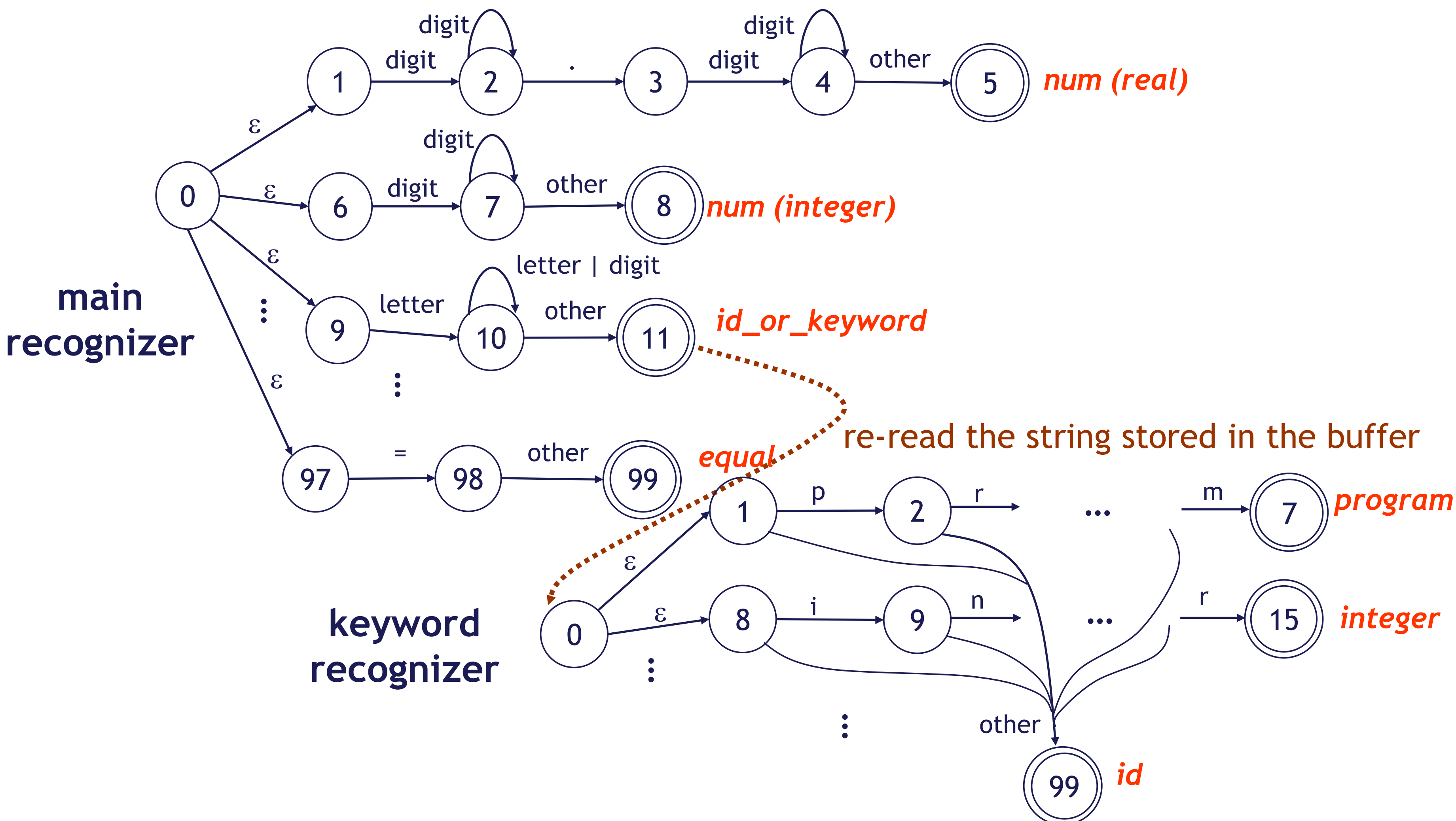
- **Each RE corresponds to a *deterministic finite automaton* (DFA)**
  - May be hard to directly construct the right DFA
  - NFA for RE such as (a | b)* abb



- **NFA is a little different from DFA**
  - S_0 has a transition on ε
  - S_1 has two transitions on a

Token Recognizer

- **Tokens are recognized by NFA**



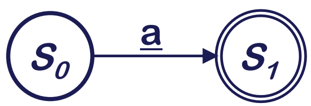
Automating Scanner Construction

- **RE→ NFA** ***(Thompson's construction)***
  - Build an NFA for each term
  - Combine them with ε-moves
- **NFA → DFA** ***(subset construction)***
  - Build the simulation
- **DFA → Minimal DFA**
  - Hopcroft's algorithm
- **DFA →RE** ***(Not part of the scanner construction)***
  - All pairs, all paths problem
  - Take the union of all paths from s_0 to an accepting state

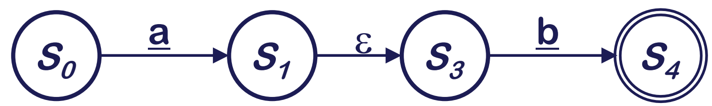
The Cycle of Constructions

RE →NFA using Thompson's Construction

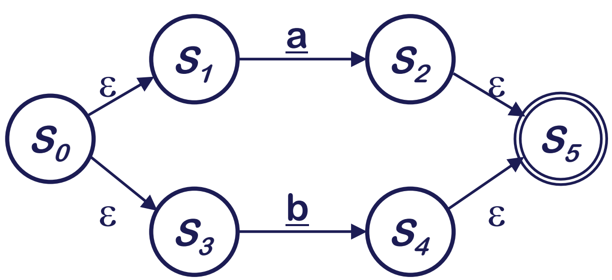
- **Key idea**
  - NFA pattern for each symbol & each operator
  - Join them with ε moves in precedence order



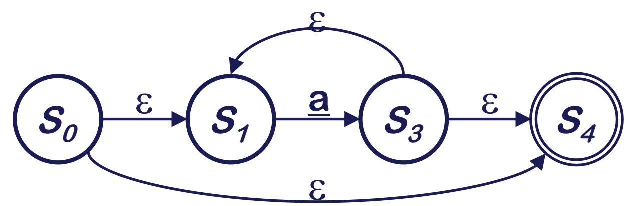
NFA for a



NFA for ab



NFA for a | b



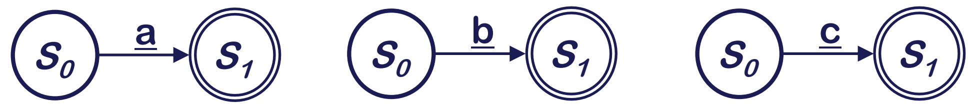
NFA for a^*

Ken Thompson, CACM, 1968

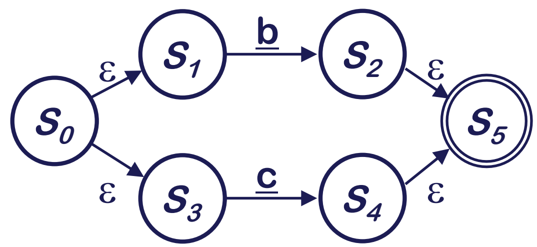
Example of Thompson's Construction

Let's try <u>a</u> (<u>b</u> | <u>c</u>)*

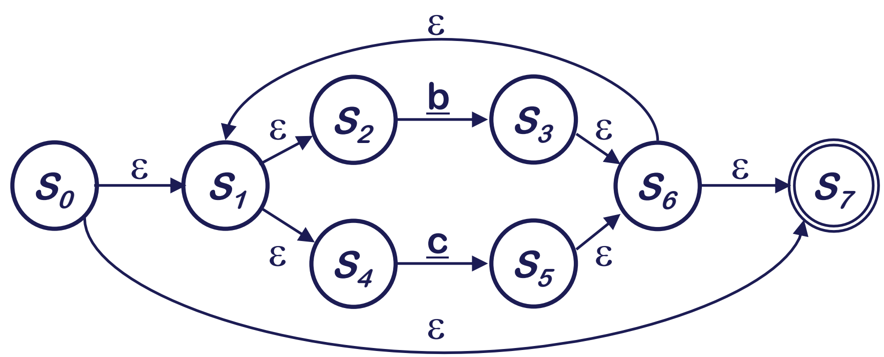
1. <u>a</u>, <u>b</u>, & <u>c</u>



2. <u>b</u> | <u>c</u>

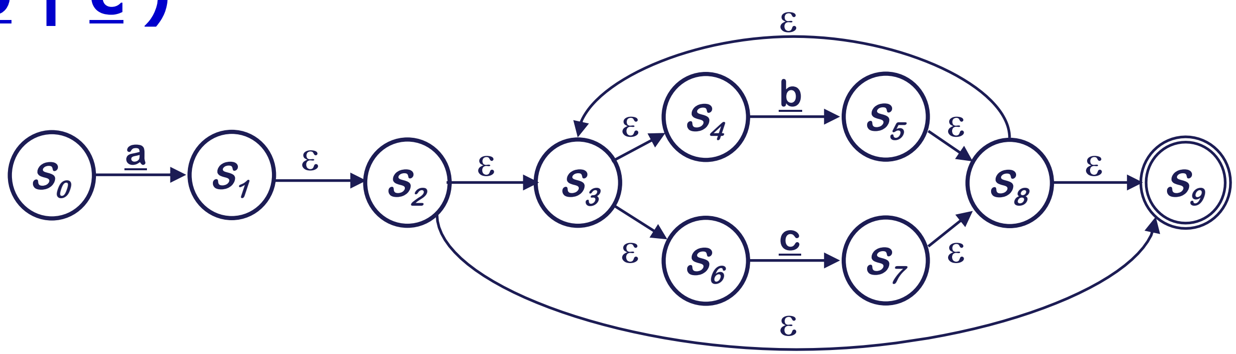


3. (<u>b</u> | <u>c</u>)*

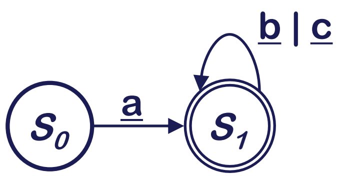


Example of Thompson's Construction (con't)

4. <u>a</u> (<u>b</u> | <u>c</u>)*



Of course, a human would design something simpler ...



But, we can automate production of the more complex one ...

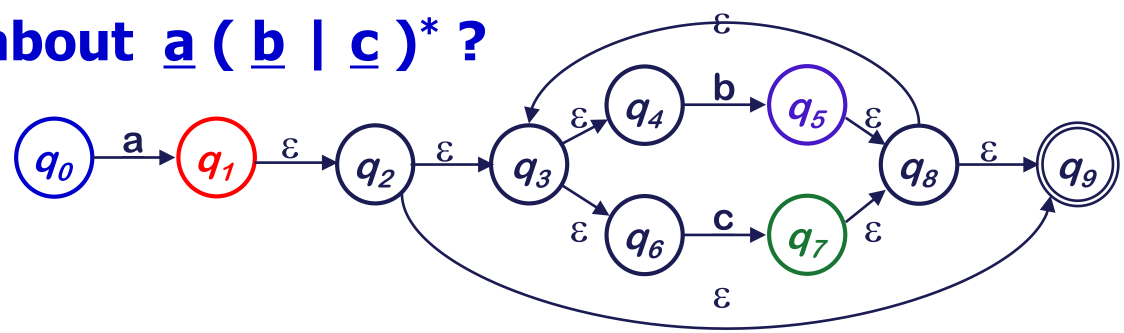
NFA $\rightarrow$DFA with Subset Construction

- **Need to build a simulation of the NFA**
- **Two key functions**
  - *Move*(s_i, a) is set of states reachable from s_i by a
  - *ε-closure*(s_i) is set of states reachable from s_i by ε
- **The algorithm:**
  - Start state derived from s_0 of the NFA
  - Take its ε-closure S_0 = ε-closure(s_0)
  - Take the image of S_0, Move(S_0, α) for each $\alpha \in \Sigma$, and take its ε-closure
  - Iterate until no more states are added

Sounds more complex than it is…

Conversion NFA to DFA

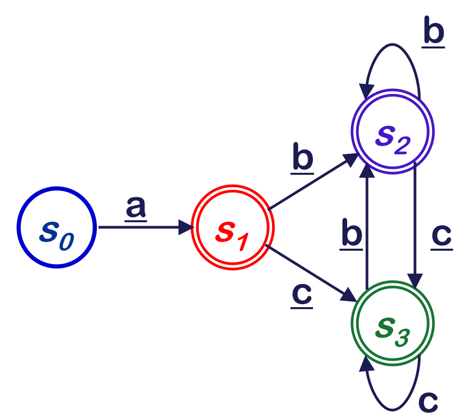
- **What about a (b | c)* ?**



- **First, the subset construction: NFA → DFA**

		ε-closure(move(s,*))		
	NFA states	a	b	c
s_0	q_0	$q_1, q_2, q_3, q_4, q_6, q_9$	none	none
s_1	$q_1, q_2, q_3, q_4, q_6, q_9$	none	$q_5, q_8, q_9, q_3, q_4, q_6$	$q_7, q_8, q_9, q_3, q_4, q_6$
s_2	$q_5, q_8, q_9, q_3, q_4, q_6$	none	s_2	s_3
s_3	$q_7, q_8, q_9, q_3, q_4, q_6$	none	s_2	s_3

Final states

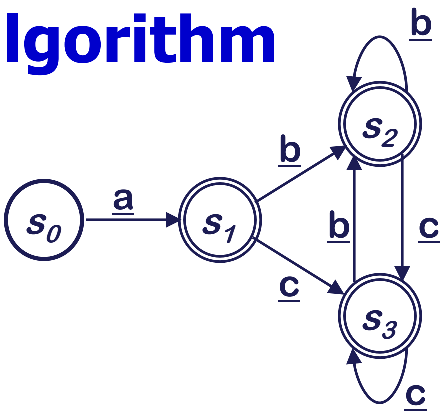


DFA Minimization

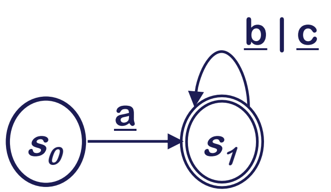
- **Then, apply the minimization algorithm**

	Current Partition	Split on a	Split on b	Split on c
P_0	$\{s_1, s_2, s_3\}$ $\{s_0\}$	*none*	*none*	*none*

final states

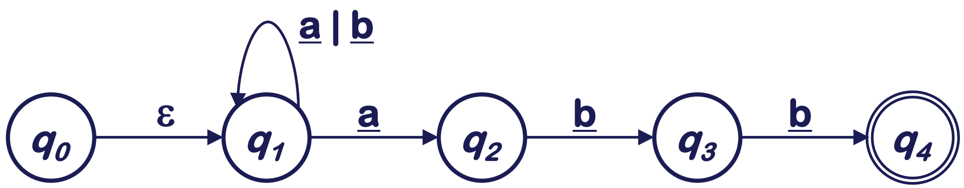


- **To produce the minimal DFA**



Another Example

- **Remember (a | b)* abb ?**



Our first NFA

- **Applying the subset construction:**

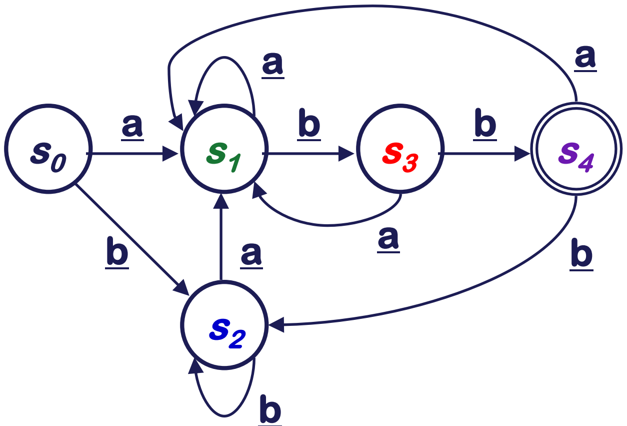
Iter.	State	Contains	ε-closure(move(s_i,a))	ε-closure(move(s_i,b))
0	s_0	q_0, q_1	q_1, q_2	q_1
1	s_1	q_1, q_2	q_1, q_2	q_1, q_3
	s_2	q_1	q_1, q_2	q_1
2	s_3	q_1, q_3	q_1, q_2	q_1, q_4
3	s_4	q_1, q_4	q_1, q_2	q_1

contains q_4 (final state)

Iteration 3 adds nothing to *S*, so the algorithm halts

Another Example (cont'd)

- **The DFA for (a | b)* abb**



δ	a	b
s_0	s_1	s_2
s_1	s_1	s_3
s_2	s_1	s_2
s_3	s_1	s_4
s_4	s_1	s_2

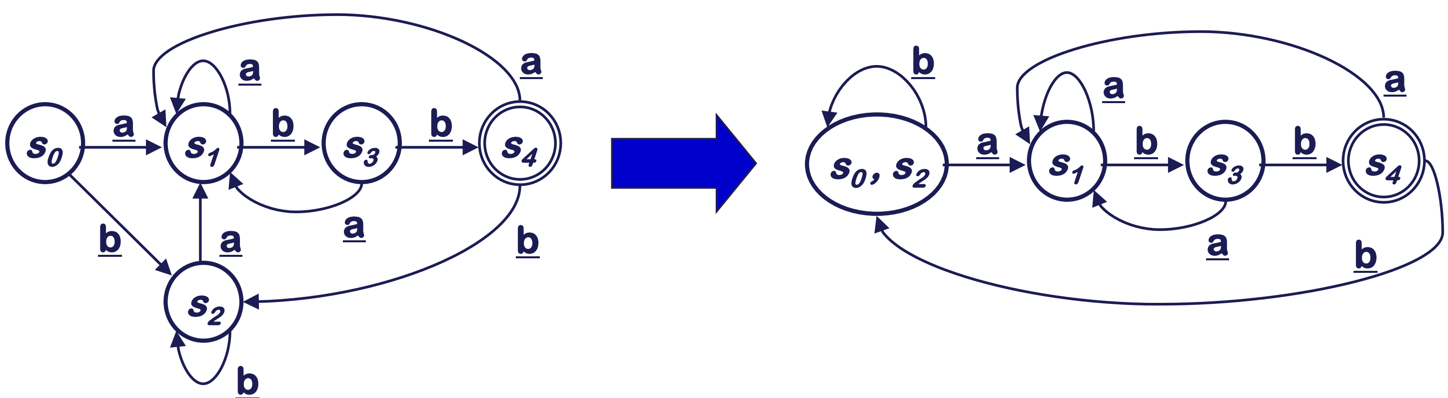
- Not much bigger than the original
- All transitions are deterministic

Another Example (cont'd)

- **Applying the minimization algorithm to the DFA**

	Current Partition	*Worklist*	*s*	*Split on a*	*Split on b*
P_0	$\{s_4\}\ \{s_0,s_1,s_2,s_3\}$	$\{s_4\}$ $\{s_0,s_1,s_2,s_3\}$	$\{s_4\}$	none	$\{s_0, s_1, s_2\}$ $\{s_3\}$
P_1	$\{s_4\}\ \{s_3\}\ \{s_0,s_1,s_2\}$	$\{s_0,s_1,s_2\}$ $\{s_3\}$	$\{s_3\}$	none	$\{s_0, s_2\}\{s_1\}$
P_2	$\{s_4\}\ \{s_3\}\ \{s_1\}\ \{s_0,s_2\}$	$\{s_0,s_2\}\{s_1\}$	$\{s_1\}$	none	none

final state



Building Faster Scanners from the DFA

- **Table-driven recognizers waste effort**
  - Read (& classify) the next character
  - Find the next state
  - Assign to the state variable
  - Trip through case logic in *δ()* & *action()*
  - Branch back to the top

- **We can do better**
  - Encode state & actions in the code
  - Do transition tests locally
  - Generate ugly, spaghetti-like code
  - Takes (many) fewer operations per input character

```
char ← next character;
state ← s0 ;
call action(state,char);
while (char ≠ eof)
  state ← δ(state,char);
  call action(state,char);
  char ← next character;

if T(state) = final then
  report acceptance;
else
  report failure;
```

Building Faster Scanners from the DFA

- **A direct-coded recognizer for r̲ *Digit Digit*****

```
     goto s0;
s0: word ← Ø;
     char  ← next character;
     if (char = 'r')
        then goto s1;
        else goto se;
s1: word ← word + char;
     char ← next character;
     if ('0' ≤ char ≤  '9')
        then goto s2;
        else goto se;
```

```
s2: word ← word + char;
     char  ← next character;
     if ('0' ≤ char ≤  '9')
        then goto s2;
        else if (char = eof)
             then report success;
             else goto se;
se: print error message;
     return failure;
```

- Many fewer operations per character
- Almost no memory operations
- Even faster with careful use of fall-through cases

Summary

- **Building scanner**
  - All this technology automates scanner construction
  - Implementer writes down the regular expressions
  - Scanner generator builds NFA, DFA, minimal DFA, and then writes out the (table-driven or direct-coded) code
  - This reliably produces fast, robust scanners

- **For most modern language features, this works**
  - You should think twice before introducing a feature that defeats a DFA-based scanner
    - insignificant blanks (Fortran: *anint* = *an int* = *an   int*)
    - non-reserved keywords (e.g. int if = 1;)