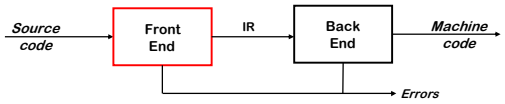
Scanning
(Lexical Analysis)

The Front End

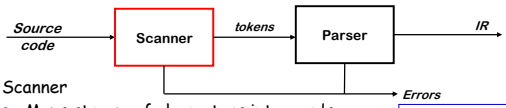


The purpose of the front end is to deal with the input language

- Perform a membership test: code ∈ source language?
- Is the program well-formed (semantically) ?
- Build an IR version of the code for the rest of the compiler

The front end is not monolithic

The Front End

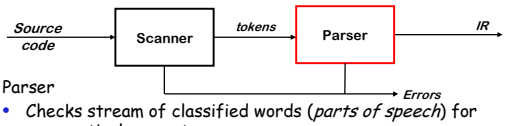


Scanner

- Maps stream of characters into words
  - → Basic unit of syntax
  - → x = x + y ; *becomes*
    <id,x> <eq,=> <id,x> <pl,+> <id,y> <sc,;>
- Characters that form a word are its *lexeme*
- Its *part of speech* (or *syntactic category*) is called its *token type*
- Scanner discards white space & (often) comments

Speed is an issue in scanning
⇒ use a specialized recognizer

The Front End



Parser

- Checks stream of classified words (*parts of speech*) for grammatical correctness
- Determines if code is syntactically well-formed
- Guides checking at deeper levels than syntax
- Builds an IR representation of the code

The Big Picture

- Language syntax is specified with *parts of speech*, not *words*
- Syntax checking matches *parts of speech* against a grammar

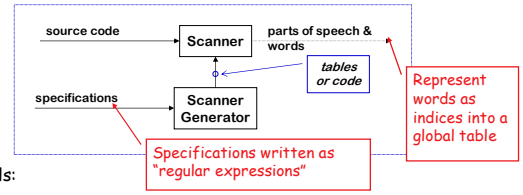
1. *goal* → *expr*
2. *expr* → *expr op term*
3. | *term*
4. *term* → number
5. | id
6. *op* → +
7. | –

S = *goal*
T = { number, id, +, - }
N = { *goal*, *expr*, *term*, *op* }
P = { 1, 2, 3, 4, 5, 6, 7}

The Big Picture

Why study lexical analysis?

- We want to avoid writing scanners by hand



Goals:

- → To simplify specification & implementation of scanners
- → To understand the underlying techniques and technologies

Regular Expressions

Lexical patterns form a *regular language*

**** any finite language is regular ****

Regular expressions (REs) describe regular languages

Ever type "rm *.o a.out" ?

Regular Expression (over alphabet Σ)

- ε is a RE denoting the set {ε}
- If a is in Σ, then a is a RE denoting {a}
- If *x* and *y* are REs denoting *L(x)* and *L(y)* then
  - → *x* | *y* is an RE denoting *L(x)* ∪ *L(y)*
  - → *xy* is an RE denoting *L(x)L(y)*
  - → *x** is an RE denoting *L(x)**

Precedence is *closure*, then *concatenation*, then *alternation*

Set Operations (review)

Operation	Definition
Union of L and M ***Written L ∪ M***	$L \cup M = \{ s \mid s \in L \text{ or } s \in M \}$
Concatenation of L and M ***Written LM***	**LM = {*st* | *s* ∈ *L* and *t* ∈ *M*}**
Kleene closure of L ***Written L****	$L^* = \cup_{0 \le i \le \infty} L^i$
Positive Closure of L ***Written L+***	$L^+ = \cup_{1 \le i \le \infty} L^i$

These definitions should be well known

Examples of Regular Expressions

Identifiers:

Letter → (a|b|c| … |z|A|B|C| … |Z)
Digit → (0|1|2| … |9)
Identifier → *Letter* (*Letter* | *Digit*)*

Numbers:

Integer → (+|-|ε) (0| (1|2|3| … |9)(*Digit**))
Decimal → *Integer* . *Digit**
Real → (*Integer* | *Decimal*) E (+|-|ε) *Digit**
Complex → (*Real* , *Real*)

Numbers can get much more complicated!

Regular Expressions (the point)

Regular expressions can be used to specify the words to be translated to parts of speech by a lexical analyzer

Using results from automata theory and theory of algorithms, we can automatically build recognizers from regular expressions

⇒ We study REs and associated theory to automate scanner construction !

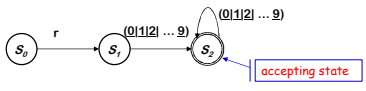
Example

Consider the problem of recognizing ILOC register names

Register → r (0|1|2| … | 9) (0|1|2| … | 9)*

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

RE corresponds to a recognizer (or DFA)



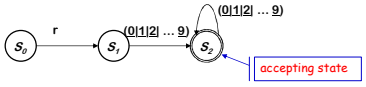
Recognizer for *Register*

Transitions on other inputs go to an error state, s_e

Example (continued)

DFA operation

- Start in state S_0 & take transitions on each input character
- DFA accepts a word *x* iff *x* leaves it in a final state (S_2)



Recognizer for *Register*

So,

- r17 takes it through S_0, S_1, S_2 and accepts
- r takes it through S_0, S_1 and fails
- a takes it straight to S_e

Example (continued)

To be useful, recognizer must turn into code

```
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
```

Skeleton recognizer

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

Table encoding RE

Example (continued)

To be useful, recognizer must turn into code

```
Char ← next character
State ← s0
while (Char ≠ EOF)
    State ← δ(State,Char)
    perform specified action
     Char ← next character
if (State is a final state )
    then report success
    else  report failure
```

Skeleton recognizer

δ	r	0,1,2,3,4, 5,6,7,8,9	*All others*
s_0	s_1 *start*	s_e *error*	s_e *error*
s_1	s_e *error*	s_2 add	s_e *error*
s_2	s_e *error*	s_2 add	s_e *error*
s_e	s_e *error*	s_e *error*	s_e *error*

Table encoding RE

What if we need a tighter specification?

r *Digit Digit** allows arbitrary numbers

- Accepts r00000
- Accepts r99999
- What if we want to limit it to r0 through r31 ?

Write a tighter regular expression

→ *Register* → r ((0|1|2) (*Digit* | ε) | (4|5|6|7|8|9) | (3|30|31))

→ *Register* → r0|r1|r2| … |r31|r00|r01|r02| … |r09

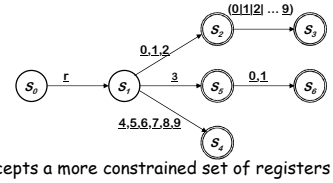
Produces a more complex DFA

- Has more states
- Same cost per transition
- Same basic implementation

Tighter register specification (continued)

The DFA for

Register → r ((0|1|2) (*Digit* | ε) | (4|5|6|7|8|9) | (3|30|31))



- Accepts a more constrained set of registers
- Same set of actions, more states

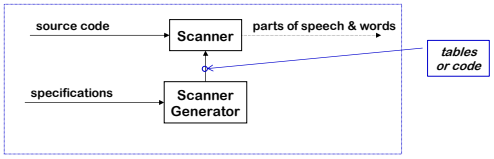
Tighter register specification (continued)

δ	r	0,1	2	3	4-9	All 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_e	s_e	s_e	s_e	s_e	s_e
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

Runs in the same skeleton recognizer

Table encoding RE for the tighter register specification

Constructing a Scanner - Quick Review



→ The scanner is the first stage in the front end

→ Specifications can be expressed using regular expressions

→ Build tables and code from a DFA

Goal

- We will show how to construct a finite state automaton to recognize any RE
- Overview:
  - → Direct construction of a nondeterministic finite automaton (NFA) to recognize a given RE
    - Requires ε-transitions to combine regular subexpressions
  - → Construct a deterministic finite automaton (DFA) to simulate the NFA
    - Use a set-of-states construction
  - → Minimize the number of states
    - Hopcroft state minimization algorithm
  - → Generate the scanner code
    - Additional specifications needed for details

More Regular Expressions

- All strings of 1s and 0s ending in a 1

  (0 | 1)* 1

- All strings over lowercase letters where the vowels (a,e,i,o, & u) occur exactly once, in ascending order

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

- All strings of 1s and 0s that do not contain three 0s in a row:

More Regular Expressions

- All strings of 1s and 0s ending in a 1

  (0 | 1)* 1

- All strings over lowercase letters where the vowels (a,e,i,o, & u) occur exactly once, in ascending order

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

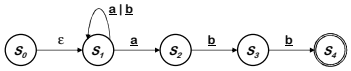
- All strings of 1s and 0s that do not contain three 0s in a row:
  (1* (ε |01 | 001) 1*)* (ε | 0 | 00)

Non-deterministic Finite Automata

Each RE corresponds to a *deterministic finite automaton* (DFA)

- May be hard to directly construct the right DFA

What about an RE such as (a | b)* abb ?



This is a little different

- S_0 has a transition on ε
- S_1 has two transitions on a

This is a *non-deterministic finite automaton* (NFA)

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*
- Transitions on ε consume no input
- To "run" the NFA, start in s_0 and *guess* the right transition at each step
  - → Always guess correctly
  - → If some sequence of correct guesses accepts x then accept

Why study NFAs?

- They are the key to automating the RE→DFA construction
- We can paste together NFAs with ε-transitions

Relationship between NFAs and DFAs

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)*

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

Automating Scanner Construction

To convert a specification into code:

1. Write down the RE for the input language
2. Build a big NFA
3. Build the DFA that simulates the NFA
4. Systematically shrink the DFA
5. Turn it into code

Scanner generators

- Lex, Flex, JLex work along these lines
- Algorithms are well-known and well-understood
- Key issue is interface to parser *(define all parts of speech)*
- You could build one in a weekend!

Automating Scanner Construction

RE→ NFA *(Thompson's construction)*

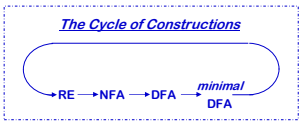
- Build an NFA for each term
- Combine them with ε-moves

NFA → DFA *(subset construction)*

- Build the simulation

DFA → Minimal DFA

- Hopcroft's algorithm



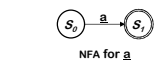
DFA →RE *(Not part of the scanner construction)*

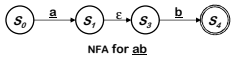
- All pairs, all paths problem
- Take the union of all paths from s_0 to an accepting state

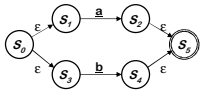
RE →NFA using Thompson's Construction

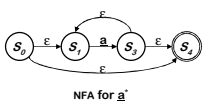
Key idea

- NFA pattern for each symbol and each operator
- Each NFA has a single start and accept state
- Join them with ε moves in precedence order







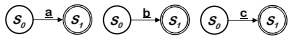


Ken Thompson, CACM, 1968

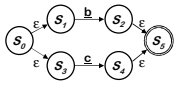
Example of Thompson's Construction

Let's try a (b | c)*

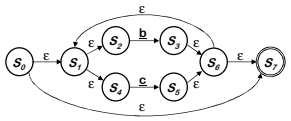
1. a, b, & c



2. b | c

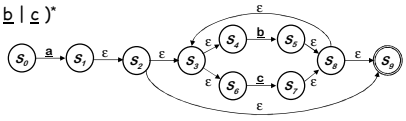


3. (b | c)*

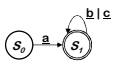


Example of Thompson's Construction (*con't*)

4. a (b | c)*



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



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

NFA →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, a) for each $a \in \Sigma$, and take its ε-closure
- Iterate until no more states are added

Sounds more complex than it is...

NFA →DFA with Subset Construction

The algorithm:

$s_0 \leftarrow \varepsilon\text{-closure}(q_0)$
add s_0 to S
while (S is still changing)
  for each $s_j \in S$
    for each $a \in \Sigma$
      $s_p \leftarrow \varepsilon\text{-closure}(\text{move}(s_j, a))$
      if ($s_p \notin S$) then
        add s_p to S as s_k
        $T[s_j, a] \leftarrow s_k$

Let's think about why this works

The algorithm halts:

1. S contains no duplicates (test before adding)
2. 2^Q is finite
3. while loop adds to S, but does not remove from S *(monotone)*

⇒ the loop halts

S contains all the reachable NFA states

It tries each character in each s_k

It builds every possible NFA configuration.

⇒ *S and T form the DFA*

NFA →DFA with Subset Construction

Example of a *fixed-point* computation

- Monotone construction of some finite set
- Halts when it stops adding to the set
- Proofs of halting & correctness are similar
- These computations arise in many contexts

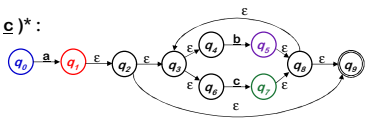
Other fixed-point computations

- Canonical construction of sets of LR(1) items
  - → Quite similar to the subset construction
- Classic data-flow analysis
  - → Solving sets of simultaneous set equations

We will see many more fixed-point computations

NFA →DFA with Subset Construction

a (b | c)* :



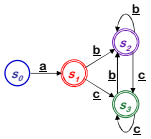
Applying the subset construction:

		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

NFA →DFA with Subset Construction

The DFA for a (b | c)*



δ	a	b	c
s_0	s_1	-	-
s_1	-	s_2	s_3
s_2	-	s_2	s_3
s_3	-	s_2	s_3

- Ends up smaller than the NFA
- All transitions are deterministic

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 really part of scanner construction)*

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

The Cycle of Constructions

DFA Minimization

The Big Picture

- Discover sets of equivalent states
- Represent each such set with just one state

DFA Minimization

The Big Picture
- Discover sets of equivalent states
- Represent each such set with just one state

Two states are equivalent if and only if:
- $\forall\ a \in \Sigma$, transitions on a lead to equivalent states (DFA)
- a-transitions to distinct sets $\Rightarrow$ states must be in distinct sets

DFA Minimization

The Big Picture
- Discover sets of equivalent states
- Represent each such set with just one state

Two states are equivalent if and only if:
- $\forall\ a \in \Sigma$, transitions on a lead to equivalent states (DFA)
- a-transitions to distinct sets $\Rightarrow$ states must be in distinct sets

A partition P of S
- Each state $s \in S$ is in exactly one set $p_i \in P$
- The algorithm iteratively partitions the DFA's states

DFA Minimization

Details of the algorithm
- Group states into maximal size sets, *optimistically*
- Iteratively subdivide those sets, as needed
- States that remain grouped together are equivalent

Initial partition, P_0, has two sets: $\{F\}$ & $\{Q-F\}$ $(D = (Q, \Sigma, \delta, q_0, F))$

Splitting a set ("partitioning a set by $\underline{a}$")
- Assume q_a, & $q_b \in s$, and $\delta(q_a, \underline{a}) = q_x$, & $\delta(q_b, \underline{a}) = q_y$
- If q_x & q_y are not in the same set, then s must be split
  - → q_a has transition on a, q_b does not $\Rightarrow$ $\underline{a}$ splits s

DFA Minimization

The algorithm

```
P ← { F, {Q-F}}
while ( P is still changing)
   T ← { }
   for each set S ∈ P
      for each a∈ Σ
         T ← T ∪ split(S, a)
   P ← T

split(S, a):
 if partition S by a
    into S1 and S2 then
       return {S1 , S2}
    else return S
```

This is a fixed-point algorithm!

Why does this work?
- Partition $P \in 2^Q$
- Start off with 2 subsets of Q *{F} and {Q-F}*
- *While* loop takes $P_i \rightarrow P_{i+1}$ by splitting 1 or more sets
- P_{i+1} is at least one step closer to the partition with $|Q|$ sets
- Maximum of $|Q|$ splits

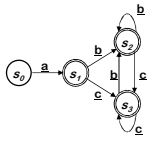
Note that
- Partitions are never combined

DFA Minimization

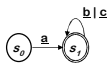
Then, apply the minimization algorithm

	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



We observed that a human would design a simpler automaton than Thompson's construction & the subset construction did.

Minimizing that DFA produces the one that a human would design!

Abbreviated Register Specification

Start with a regular expression

r0 | r1 | r2 | r3 | r4 | r5 | r6 | r7 | r8 | r9

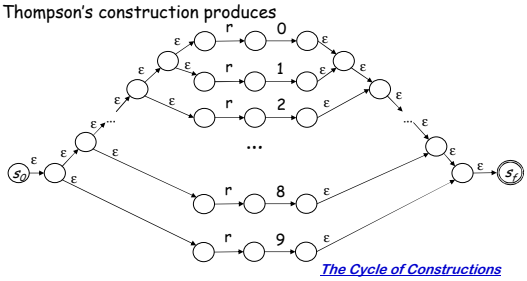
Note: "|" is left associate

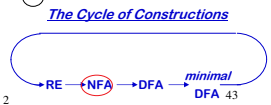
The Cycle of Constructions

RE → NFA → DFA → *minimal* DFA

Abbreviated Register Specification

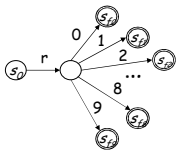
Thompson's construction produces



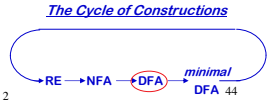


Abbreviated Register Specification

The subset construction builds

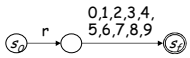


This is a DFA, but it has a lot of states …

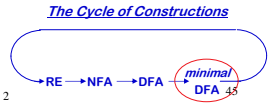


Abbreviated Register Specification

The DFA minimization algorithm builds



This looks like what a skilled compiler writer would do!



Limits of Regular Languages

Advantages of Regular Expressions

- Simple & powerful notation for specifying patterns
- Automatic construction of fast recognizers
- Many kinds of syntax can be specified with REs

Example — an expression grammar

Term → [a-zA-Z] ([a-zA-z] | [0-9])*
Op → + | - | * | /
Expr → (*Term Op*)* *Term*

Of course, this would generate a DFA …

If REs are so useful …

Why not use them for everything?

Limits of Regular Languages

Not all languages are regular

RL's ⊂ CFL's ⊂ CSL's

You cannot construct DFA's to recognize these languages

- $L = \{ p^k q^k \}$ *(parenthesis languages)*
- $L = \{ wcw^r \mid w \in \Sigma^* \}$

Neither of these is a regular language *(nor an RE)*

But, this is a little subtle. You can construct DFA's for

- Strings with alternating 0's and 1's
  (ε | 1) (01)* (ε | 0)
- Strings with and even number of 0's and 1's
  ???
- Strings with the same number of 0's and 1's
  ???

What can be so hard?

Poor language design can complicate scanning

- Reserved words are important
  if then then then = else; else else = then (PL/I)
- Insignificant blanks (Fortran & Algol68)
  do 10 i = 1,25
  do 10 i = 1.25
- String constants with special characters (C, C++, Java, …)
  newline, tab, quote, comment delimiters, …
- Limited identifier "length" (Fortran 66 & PL/I)