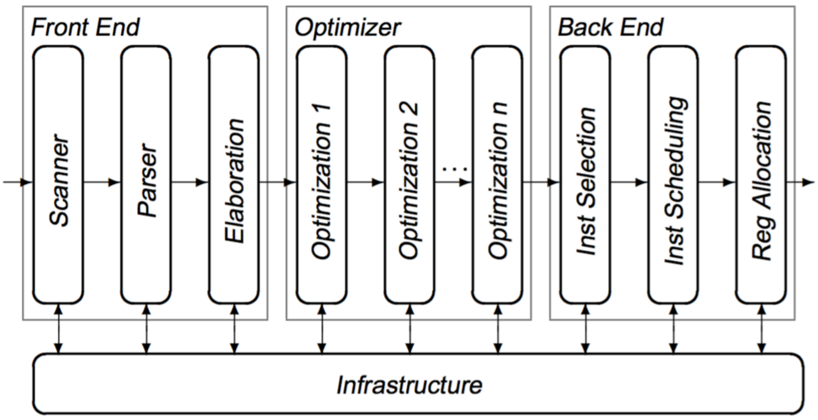
Lexical Analysis

Note by Baris Aktemur:
Our slides are adapted from Cooper and Torczon's slides that they prepared for COMP 412 at Rice.

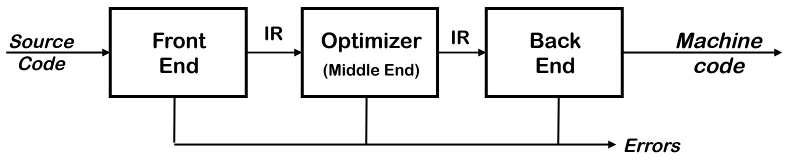
Traditional Three-part Compiler



■ **FIGURE 1.1** Structure of a Typical Compiler.

1

The Front End



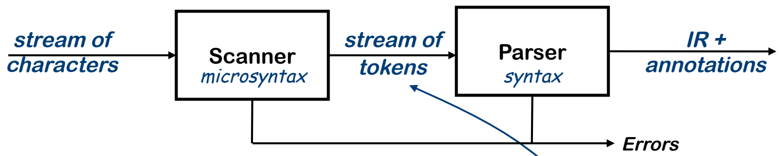
The purpose of the front end is to deal with the input language

- 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 deals with form (syntax) & meaning (semantics)

2

The Front End



Scanner is only pass that touches every character of the input.

Why separate the scanner and the parser?

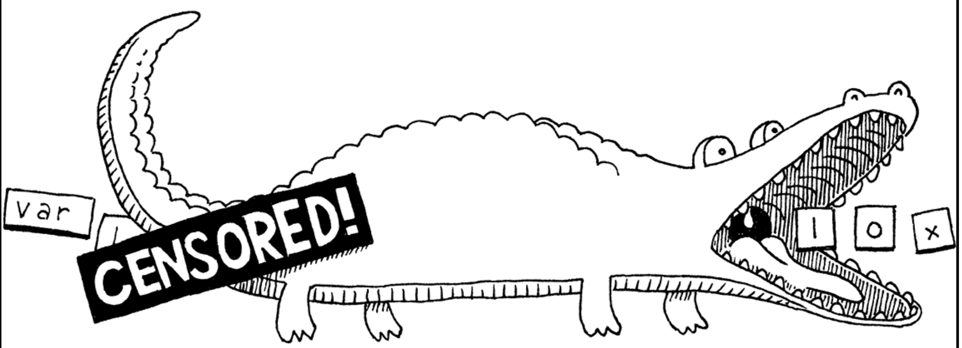
- Scanner classifies words
- Parser constructs grammatical derivations
- Parsing is harder and slower

Separation simplifies the implementation

- Scanners are simple
- Scanner leads to a faster, smaller parser

token is a pair
<part of speech, lexeme>

3

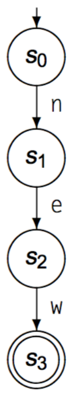


Lexical analygator.
[http://www.craftinginterpreters.com/scanning.html]

4

Recognizing the word "new"

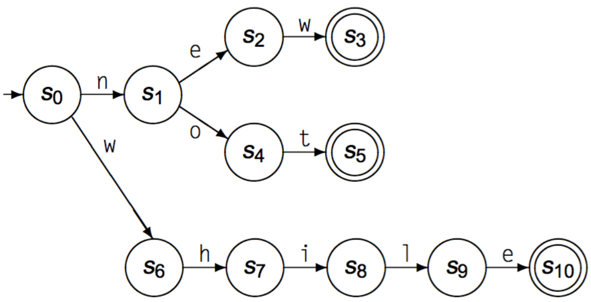
```
c ← NextChar();
if (c = 'n')
   then begin;
      c ← NextChar();
      if (c = 'e')
         then begin;
            c ← NextChar();
            if (c = 'w')
               then report success;
               else try something else;
         end;
         else try something else;
   end;
   else try something else;
```



■ **FIGURE 2.1** Code Fragment to Recognize "new".

5

Recognizing "new", "not", "while"

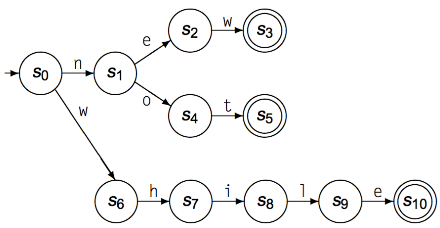


6

Finite Automata

$(S, \Sigma, \delta, s_0, S_A)$

- S: finite set of states
- Σ: alphabet
- δ: transition function
- s_0: start state
- S_A: set of accepting states



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

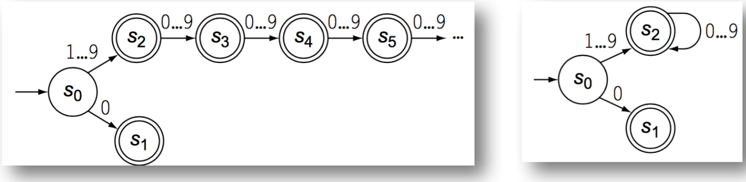
$$\Sigma = \{\mathtt{e}, \mathtt{h}, \mathtt{i}, \mathtt{l}, \mathtt{n}, \mathtt{o}, \mathtt{t}, \mathtt{w}\}$$

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

$$s_0 = s_0$$

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

More complex words




```
char ← NextChar( );
state ← s0 ;

while (char ≠ eof and state ≠ se) do
   state ← δ(state,char);
   char ← NextChar( );
end;

if (state ∈ SA)
    then report acceptance;
    else report failure;
```

$S = \{s_0, s_1, s_2, s_e\}$

$\Sigma = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$

$$\delta = \left\{ \begin{array}{ll} s_0 \xrightarrow{0} s_1, & s_0 \xrightarrow{1-9} s_2 \\ s_2 \xrightarrow{0-9} s_2, & s_1 \xrightarrow{0-9} s_e \end{array} \right\}$$

$S_A = \{s_1, s_2\}$

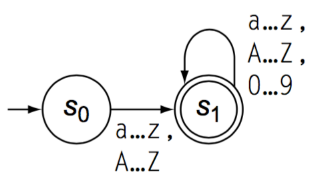
■ **FIGURE 2.2** A Recognizer for Unsigned Integers.

Represent the transition function 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

The recognizer code can be used for other cases as well. E.g:

Just change the table.



The next question

Finite automata are good and useful, but not concise.

We need a concise notation that can be transformed into FA's.

Regular Expressions

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	$LM = \{ st \mid s \in L \text{ and } t \in M \}$
*Kleene closure of L written L**	$L^* = \bigcup_{0 \le i \le \infty} L^i$
Positive closure of L written L⁺	$L^+ = \bigcup_{1 \le i \le \infty} L^i$

These definitions should be well known

Regular Expressions

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 $L(x) \cup L(y)$
  - *xy* is an RE denoting *L(x)L(y)*
  - x^* is an RE denoting $L(x)^*$

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

12

Examples of Regular Expressions

Identifiers:

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

shorthand for

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

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!

Using symbolic names does not imply recursion

underlining indicates a letter in the input stream

13

Regular Expressions

So what's the point?

We use regular expressions to specify the mapping of words to parts of speech for the lexical analyzer

Using results from automata theory and theory of algorithms, we can automate construction of recognizers from REs

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

⇒ Fortunately, the automatic techiques lead to fast scanners

→ used in text editors, URL filtering software, …

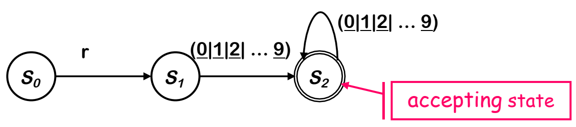
Example

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

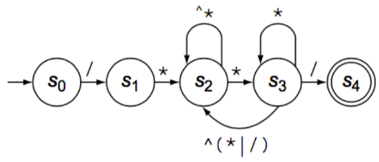
RE for C/Java-style single-line comments

```
//(^\n)*\n
```

RE for C/Java-style multi-line comments

/★ (^★) * ★/

/★ (^★ | ★+^/) * ★/ (better)



More states implies a larger table. The larger table might have mattered when computers had 128 KB or 640 KB of RAM. Today, when a cell phone has megabytes and a laptop has gigabytes, the concern seems outdated.

Examples

- 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

  Let Cons be (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)

17

Next Step

RE → NFA *(Thompson's construction)*

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

NFA → DFA *(Subset construction)*

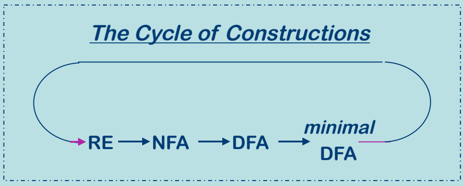
- Build the simulation

DFA → Minimal DFA

- Hopcroft's algorithm

DFA → RE

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



In another course…

18

What About Hand-Coded Scanners?

Many (most?) modern compilers use hand-coded scanners

- Starting from a DFA simplifies design & understanding
- Avoiding straight-jacket of a tool allows flexibility
  - Computing the value of an integer
    - → In LEX or FLEX, many folks use *sscanf()* & touch chars many times
    - → Can use old assembly trick and compute value as it appears

```
begin;
  RegNum ← RegNum × 10 + (char - '0');
  goto s2;
end;
```

  - Combine similar states
- Handling keywords
  - Instead of having explicit RE for each keyword, first recognize them as ordinary identifiers, then look up in a hash table.
  - A good example of perfect hashing, because of fixed set of keys.

> We preferred this approach in our Cool scanner.
>
> Clang and GCC's front ends are also hand-written.

19