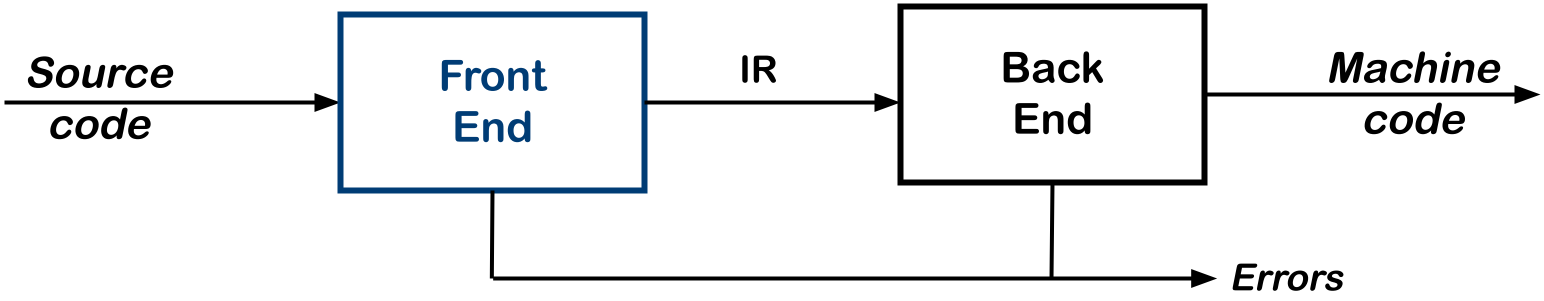
CENG513 Compiler Design and Construction
Lexical Analysis

Note by Işıl ÖZ:

Our slides are adapted from Cooper and Torczon's slides that are prepared for COMP 412 at Rice.

The Front End

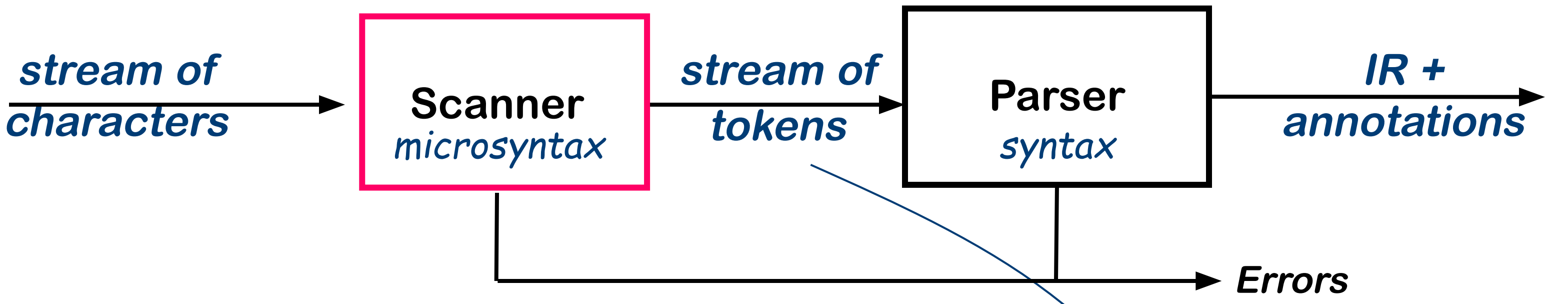


The purpose of the front end is to deal with the input language

- Perform a membership test: code $\in$ 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)

The Front End



Why separate the scanner and the parser?

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

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

Separation simplifies the implementation

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

token is a pair
<part of speech, lexeme>

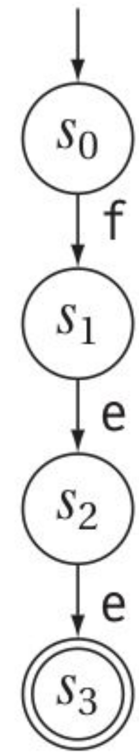
Formal Language

- Alphabet: finite set of symbols
- String: sequence of symbols from a given alphabet
- Language: set of strings
  - Formal language
- Programming language is a formal language with mathematical properties and well-defined meanings

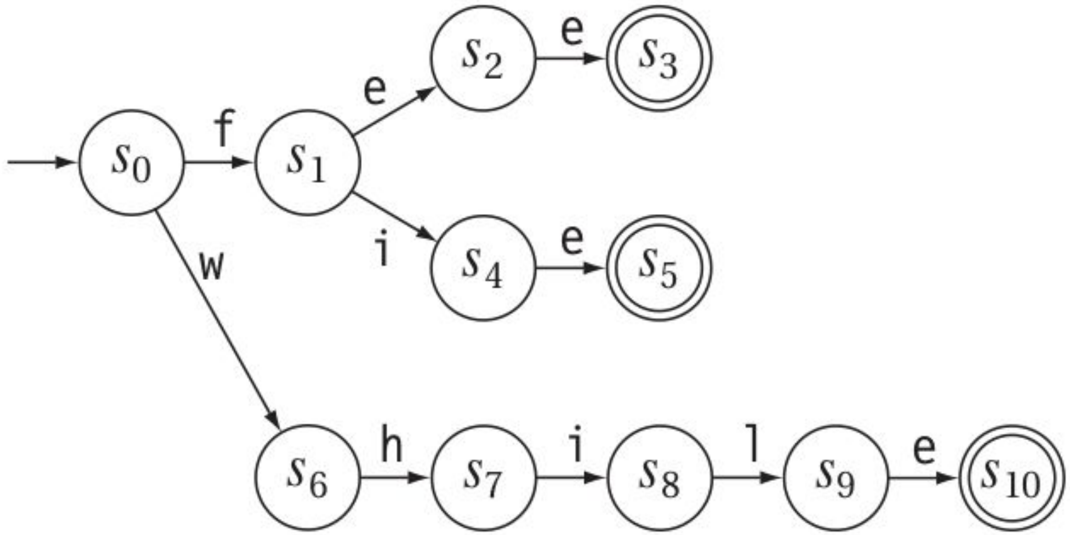
- Recognizing a language
- Given a string, tells you whether the string belongs to the language (valid or not)

Recognizing the Word "fee"

```
c ← NextChar()
if (c ≠ 'f')
   then do something else
   else
      c ← NextChar()
      if (c ≠ 'e')
         then do something else
         else
            c ← NextChar()
            if (c ≠ 'e')
               then do something else
               else report success
```



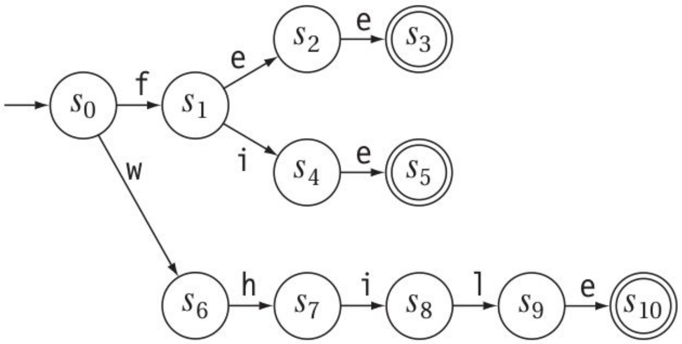
Recognizing the Words "fee", "fie", "while"



Finite Automata

Transition diagrams can be viewed as formal mathematical objects, called finite automata, that specify recognizers
$(S, \Sigma, \delta, s_0, S_A)$

- S: finite set of states
- Σ: alphabet (symbols)
- δ: transition function
- s_0 : start state
- S_F : set of accepting states



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

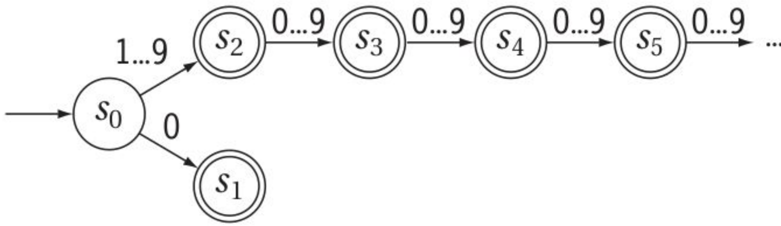
$$\Sigma = \{\mathtt{e}, \mathtt{f}, \mathtt{i}, \mathtt{h}, \mathtt{l}, \mathtt{w}\}$$

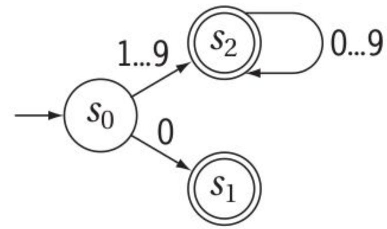
$$\delta = \left\{ \begin{array}{lllll} s_0 \xrightarrow{\mathtt{f}} s_1, & s_0 \xrightarrow{\mathtt{w}} s_6, & s_1 \xrightarrow{\mathtt{e}} s_2, & s_1 \xrightarrow{\mathtt{i}} s_4, & s_2 \xrightarrow{\mathtt{e}} s_3, \\ s_4 \xrightarrow{\mathtt{e}} s_5, & s_6 \xrightarrow{\mathtt{h}} s_7, & s_7 \xrightarrow{\mathtt{i}}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

$$s_0 = s_0$$

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

A Recognizer for Unsigned Integers





char ← *NextChar()*
state ← s_0

while (char ≠ *eof and state* ≠ s_e*)*
  state ← δ*(state,char)*
  char ← *NextChar()*

if (state ∈ S_F*)*
  then report acceptance
  else report failure

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

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

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

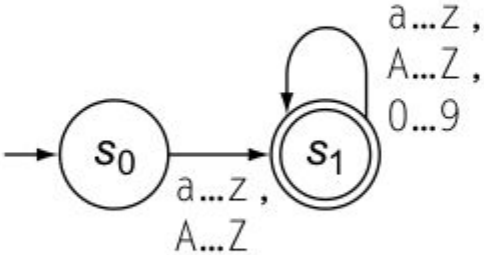
$$S_F = \{s_1, s_2\}$$

Represent the Transition Diagram 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

Identifier Names in C

An identifier consists of an alphabetic character followed by zero or more alphanumeric characters



Regular Expressions

The set of words accepted by an FA forms a language (regular language)

The transition diagram of the FA specifies that language, which is not so intuitive for humans

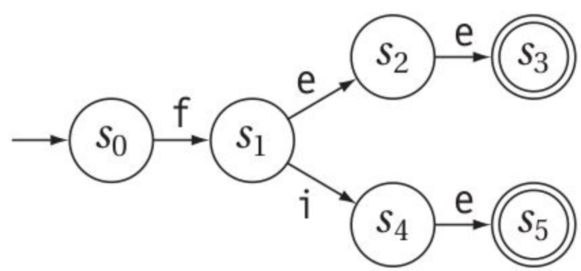
Regular expressions (RE) describe regular languages

Simple recognizers have simple RE specifications:

The language consisting of the two words “fee” or “fie” can be written as

$fee \mid fie$

$f(e \mid i)e$



Formalizing the Notation (review)

Operation	Definition
Union (alternation) of L and M written $L \mid M$	$L \mid M = \{ s \mid s \in L \text{ or } s \in M \}$
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^+	$L^+ = \bigcup_{1 \le i \le \infty} L^i$

Regular Expressions

Regular expressions over alphabet Σ

- ε is a RE denoting the set $\{\varepsilon\}$, empty string
- If $\underline{a}$ is in Σ, then $\underline{a}$ is a RE denoting $\{\underline{a}\}$
- If x and y are REs denoting $L(x)$ and $L(y)$ then
  - $x \mid y$ is an RE denoting $L(x)$ *union* $L(y)$
  - xy is an RE denoting $L(x)$ *concatenate* $L(y)$
  - x^* is an RE denoting $L(x)^*$

Precedence is *parentheses, closure, then concatenation, then union*

Regular Expressions

How do these operators help?

Regular Expression (over alphabet Σ)

- ε is a RE denoting the set $\{\varepsilon\}$
- If $\underline{a}$ is in Σ, then $\underline{a}$ is a RE denoting $\{\underline{a}\}$
  - → the spelling of any specific word is an RE
- If x and y are REs denoting $L(x)$ and $L(y)$ then
  - — $x \mid y$ is an RE denoting $L(x) \cup L(y)$
    - → any finite list of words can be written as an RE ($w_0 \mid w_1 \mid \ldots \mid w_n$)
  - — xy is an RE denoting $L(x)L(y)$
  - — x^* is an RE denoting $L(x)^*$
    - → we can use concatenation & closure to write more concise patterns and <u>to specify infinite sets</u> that have finite descriptions

Examples of Regular Expressions

Identifiers:

Letter → (<u>a</u>|<u>b</u>|<u>c</u>| … |<u>z</u>|<u>A</u>|<u>B</u>|<u>C</u>| … |<u>Z</u>)

Digit → (<u>0</u>|<u>1</u>|<u>2</u>| … |9)

Identifier → *Letter* (*Letter* | *Digit*)*

shorthand for

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

Numbers:

Integer → (<u>+</u>|<u>-</u>|ε) (<u>0</u>| (<u>1</u>|<u>2</u>|<u>3</u>| … |<u>9</u>)(*Digit* *))

Decimal → *Integer* <u>.</u> *Digit* *

Real → (*Integer* | *Decimal*) <u>E</u> (<u>+</u>|<u>-</u>|ε) *Digit* *

Complex → <u>(</u> *Real* <u>,</u> *Real* <u>)</u>

Numbers can get much more complicated!

underlining indicates a letter in the input stream

More Complex Example

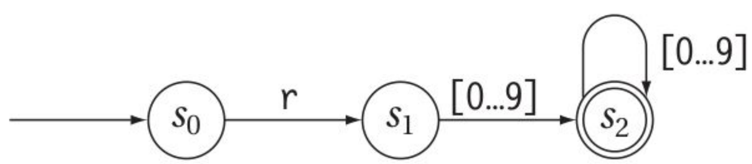
Consider the problem of recognizing *register names*

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

$$r[0\ldots9]^+$$

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

RE corresponds to a recognizer (or NFA)

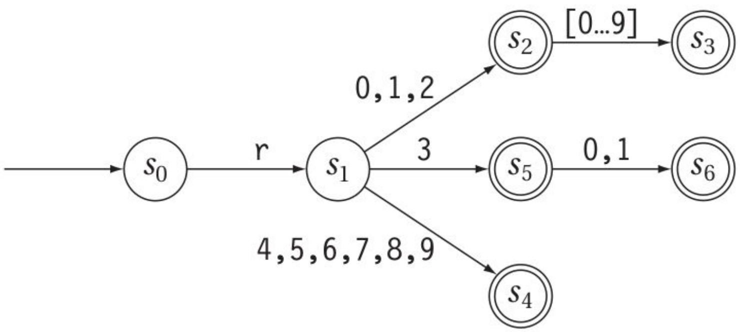


Transitions on other inputs go to an error state, s_e

More Complex Example

On a real computer, the set of register names is severely limited, restrict to registers from 0 to 31

$$r\,(\,(0|1|2)\ \ ([0\ldots9]\,|\,\epsilon)\,|\,(4|5|6|7|8|9)\,|\,(\,3\,(0|1|\epsilon))\,)$$



Alternative regular expression (simpler but longer)

$$r0\,|\,r00\,|\,r1\,|\,r01\,|\,r2\,|\,r02\,|\,r3\,|\,r03\,|\,r4\,|\,r04\,|\,r5\,|\,r05\,|\,r6\,|\,r06\,|\,r7\,|\,r07\,|\,r8\,|\,r08\,|\,r9\,|\,r09\,|\,r10\,|\,r11\,|\,r12\,|\,r13\,|\,r14\,|\,r15\,|\,r16\,|\,r17\,|\,r18\,|\,r19\,|\,r20\,|\,r21\,|\,r22\,|\,r23\,|\,r24\,|\,r25\,|\,r26\,|\,r27\,|\,r28\,|\,r29\,|\,r30\,|\,r31$$

Additional Examples

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

  $(0 \mid 1)^* 1$

- 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:

  $(1^* (\varepsilon \mid 01 \mid 001) 1^*)^* (\varepsilon \mid 0 \mid 00)$

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

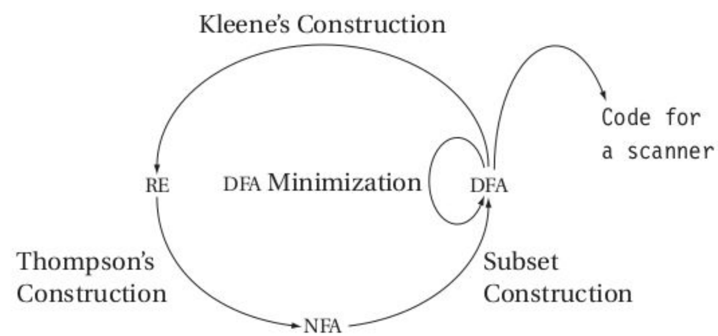
⇒ We study REs and associated theory to automate scanner construction !

⇒ Fortunately, the automatic techniques lead to fast scanners

- → used in text editors, URL filtering software, …

From Regular Expressions to Scanners (Section 2.4)

- Regular expression (RE) given
- Direct construction of a nondeterministic finite automaton (NFA) to recognize a given RE
  - Build in an algorithmic way
  - 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 in the DFA
  - Hopcroft state minimization algor
- DFA to regular expression
  - Kleene's construction
- Generate the scanner code from I
  - Additional specifications needed for the actions



Automating Scanner Construction

To convert a specification into code:

1. Write down the RE for the input language
2. Build a big NFA (RE → NFA *(Thompson's construction)*)
3. Build the DFA that simulates the NFA (NFA → DFA *(Subset construction)*)
4. Systematically shrink the DFA (DFA → Minimal DFA *(Hopcroft's algorithm)*)
5. Turn it into code

Scanner generators

- Lex and Flex work along these lines
- Algorithms are well-known and well-understood

Implementing Scanners (Transform DFA to Code)

- Table-driven scanners
  - Table and skeleton scanner code

```
state ← s0 ;
while (state ≠ exit) do
   char ← NextChar( )          // read next
character
   state ← δ(state,char);      // take the
transition
```

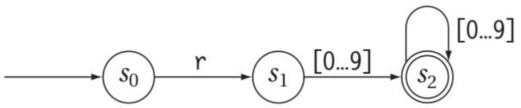
- Direct-coded scanners
  - Each state is implemented as a fragment of code
  - Eliminates memory reference for transition table access

- Hand-coded scanners
  - Instead of having explicit RE for each keyword, first recognize them as ordinary identifiers, then look up in a hash table

All will simulate DFA!

Table-Driven Scanner

$r\,[0\ldots9]^{+}$



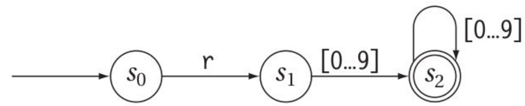
char ← *NextChar()*
state ← s_0
while (char ≠ *eof)*
    state ← *δ(state,char)*
    char ← *NextChar()*
if (state ∈ S_F*)*
    then report acceptance
    else report failure

δ	r	0,1,2,3,4 5,6,7,8,9	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

Lookup table memory overhead

Direct-Coded Scanner

$r\,[0\ldots9]^+$



goto s_0

s_0: char $\leftarrow$ NextChar()
if (char = 'r')
then goto s_1
else goto s_e

s_1: char $\leftarrow$ NextChar()
if ('0' $\leq$ char $\leq$ '9')
then goto s_2
else goto s_e

s_2: char $\leftarrow$ NextChar()
if ('0' $\leq$ char $\leq$ '9')
then goto s_2
else if (char = eof)
then report acceptance
else goto s_e

s_e: report failure

Hand-Coded Scanners

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

- Starting from a DFA/RE simplifies design & understanding

We will see this in our toy language, Kaleidoscope.

Clang and GCC's front ends are also hand-written.

References

Chapter sections from the book:

- 2.1, 2.2, 2.3, 2.5

Selected videos from compiler course from California State University:

- https://www.youtube.com/watch?v=bR5x5D2mMVg&list=PL6KMWPQP_DM97Hh0PYNgJord-sANFTI3i&index=5
- https://www.youtube.com/watch?v=b-MXQ4qVoFU&list=PL6KMWPQP_DM97Hh0PYNgJord-sANFTI3i&index=6
- https://www.youtube.com/watch?v=sb2GbNZ0Fw4&list=PL6KMWPQP_DM97Hh0PYNgJord-sANFTI3i&index=12

Kaleidoscope Lexer

- https://llvm.org/docs/tutorial/MyFirstLanguageFrontend/LangImpl01.html