1.1 Language Processors

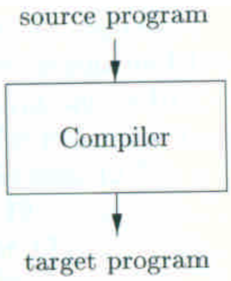


Figure 1.1: A compiler

Figure 1.2: Running the target program

1.1 Language Processors

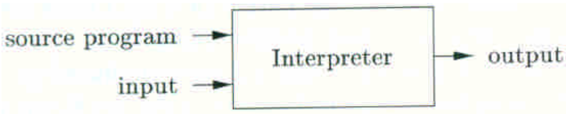


Figure 1.3: An interpreter

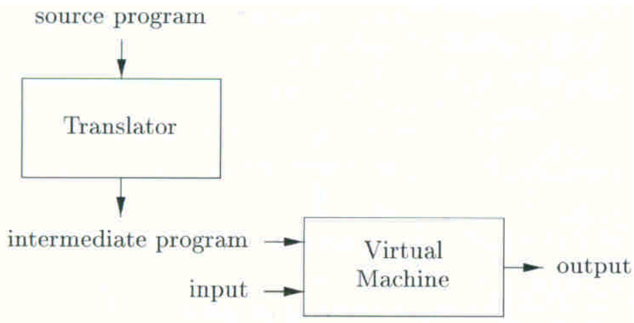
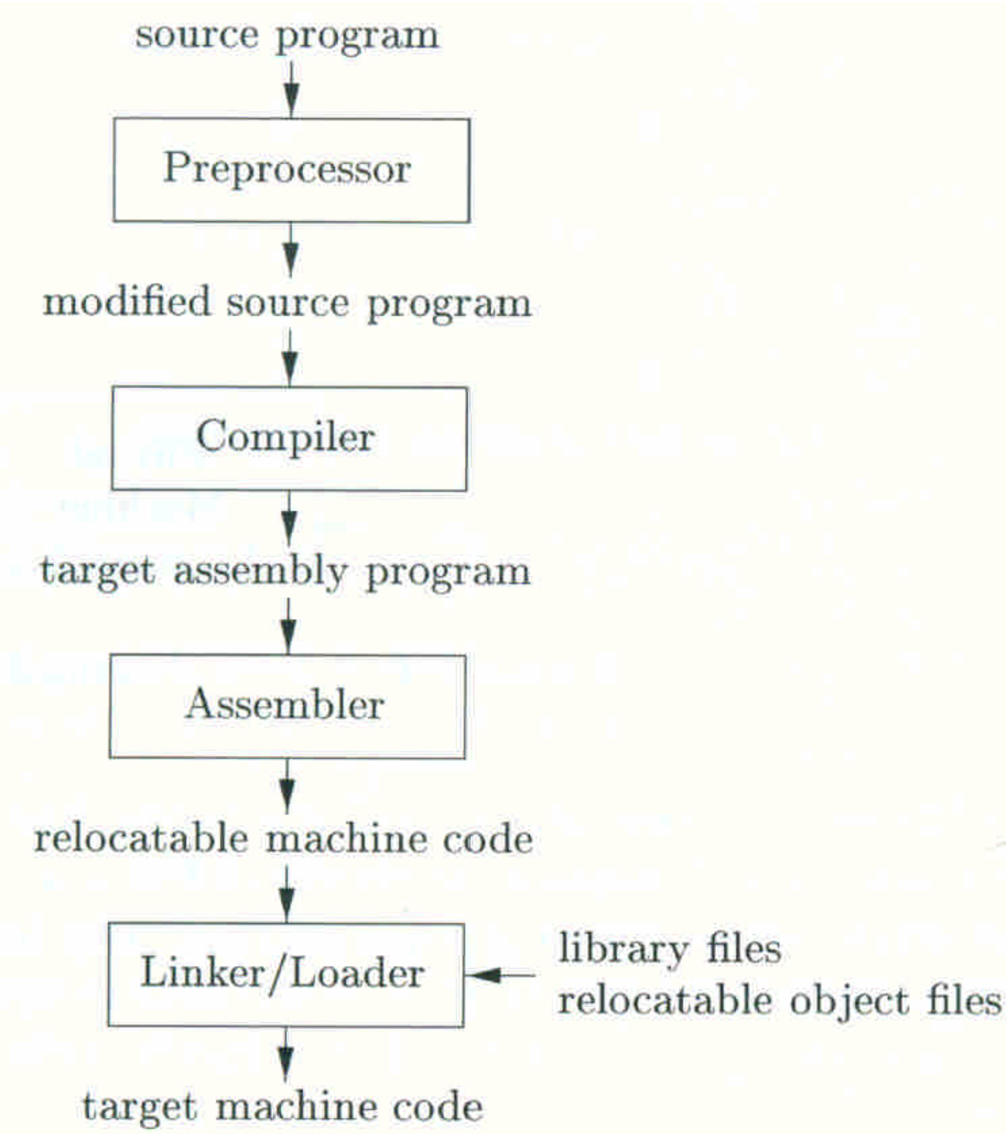
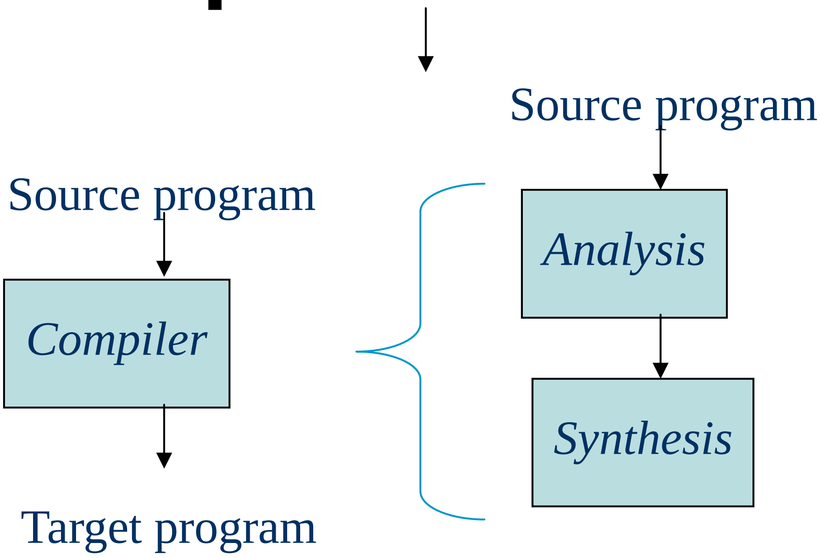


Figure 1.4: A hybrid compiler

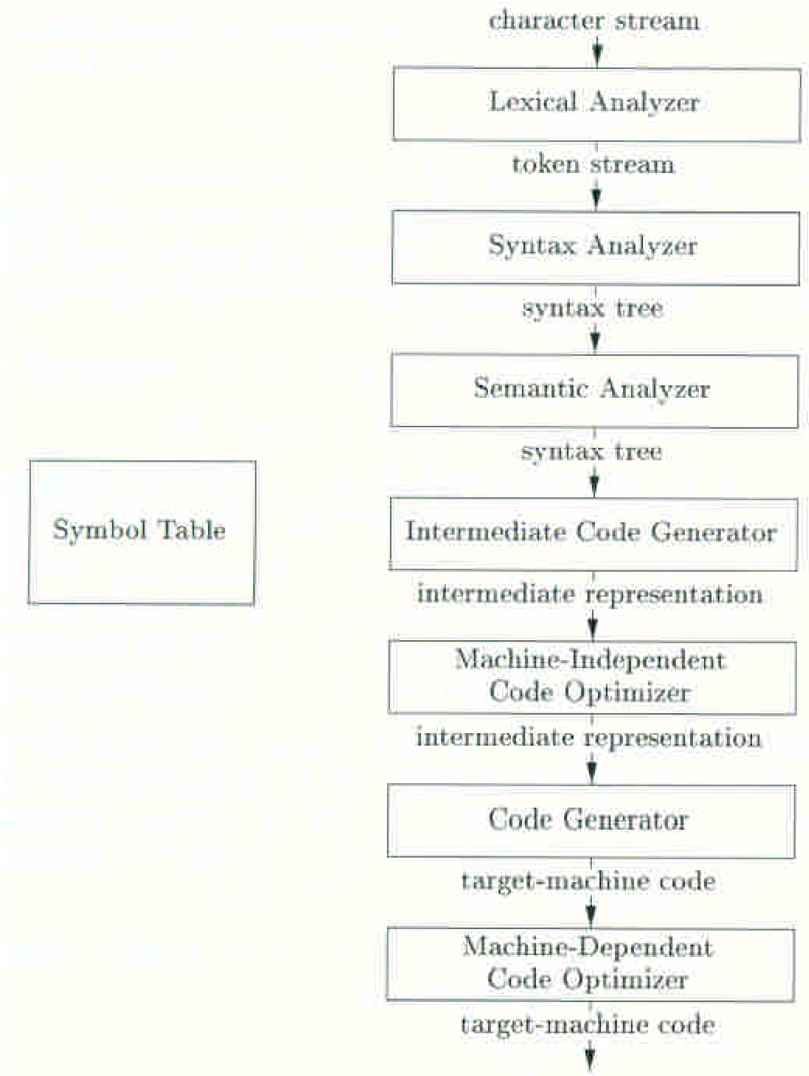
Language Processors



The Structure of a Compiler



The Structure of a Compiler



Formal Languages - Introduction

>What is a language?

> Language Related Problems

> Definition: How do we define a language?

(Remember, there are an infinite number of possible Java programs.)

> Recognition: How do we verify that a given input is in agreement with a given language definition?

(Recognition ! yes/no)

> Parsing: Recognition + building an internal representation.

(e.g. token sequences and syntax trees.)

Theory: Formal Languages

A formal language is defined in terms of:

> Symbols: the smallest identifiable units

> Alphabet Σ : a finite non-empty set of symbols

> Strings: a string (or word) over Σ is a finite sequence of symbols from Σ

Examples

> a, ba, and abba are strings over the alphabet $\Sigma = \{a, b\}$ with symbols a and b.

> Any binary number is a string over the alphabet $\Sigma = \{0, 1\}$ with symbols 0 and 1.

> The length of a string x, denoted $|x|$,

is the number of symbols in x.

> Example: $x = abba \rightarrow |x| = 4$

> The empty string ε is a special string with length zero, i.e. $|\varepsilon| = 0$

> The concatenation of two strings x and y over Σ, denoted xy, is a new string formed by appending y to x.

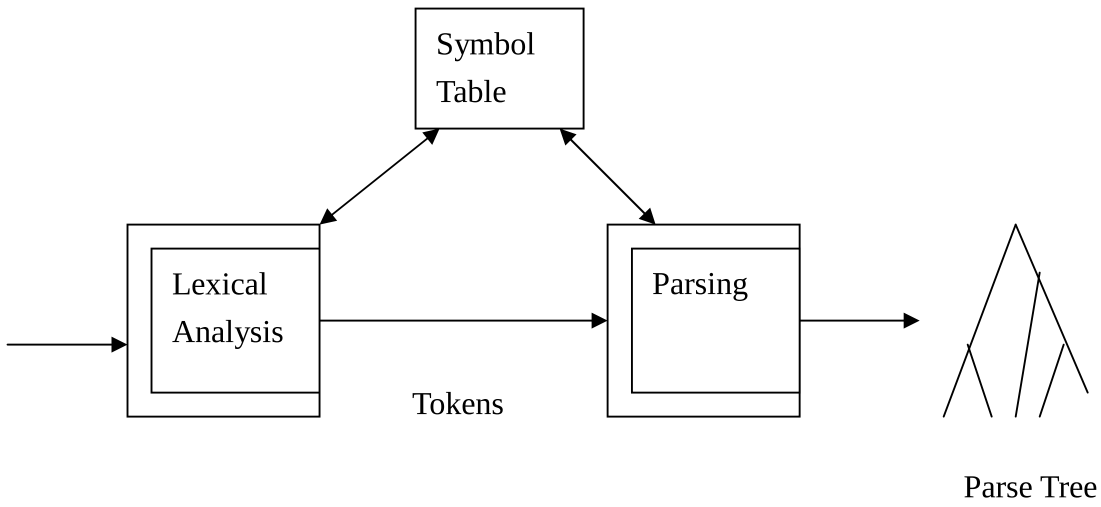
> Example: $x = ab$, $y = ba \rightarrow xy = abba$

> Note, $\forall x \quad x = \varepsilon x = x\varepsilon$

i.e. ε is the identity element under concatenation.

Lexical Analysis

(Performed by the Scanner)



> Read the input characters, identify atomic language constructs, and produce as output a sequence of tokens.

Secondary Tasks

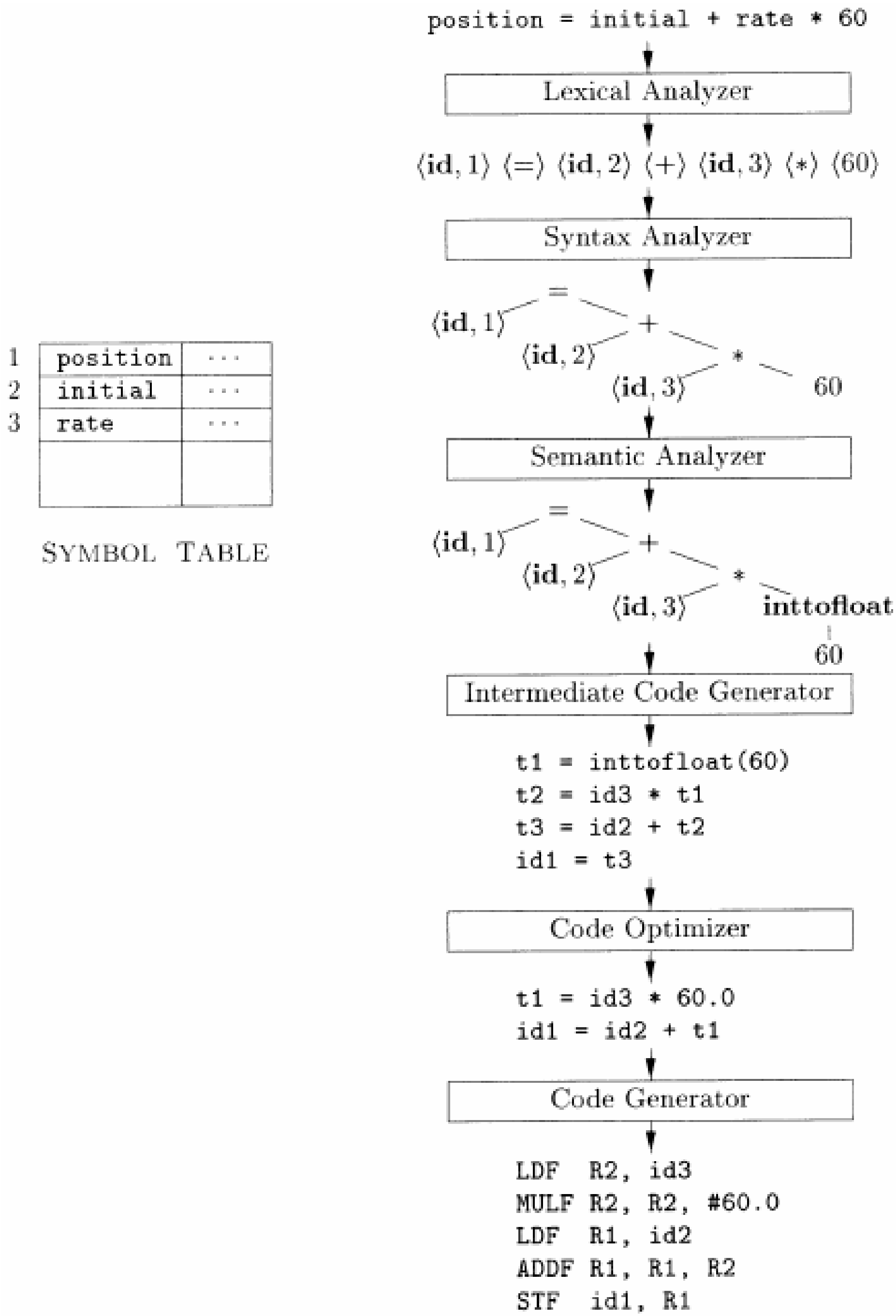
> Remove whitespace and comments.

> Update the symbol table

> Report lexical errors

Lexical Analysis

- **The lexical analyzer reads the stream of characters making up the source program and groups the characters into meaningful sequences called *lexemes*.**
- **For each lexeme, the lexical analyzer produces as output a *token* of the form ⟨*token-name*, *attribute-value*⟩.**


position = initial + rate * 60
Lexical Analyzer
⟨id, 1⟩ ⟨=⟩ ⟨id, 2⟩ ⟨+⟩ ⟨id, 3⟩ ⟨*⟩ ⟨60⟩
Syntax Analyzer
=
⟨id, 1⟩
+
⟨id, 2⟩
*
⟨id, 3⟩
60
Semantic Analyzer
=
⟨id, 1⟩
+
⟨id, 2⟩
*
⟨id, 3⟩
inttofloat
60
Intermediate Code Generator
t1 = inttofloat(60)
t2 = id3 * t1
t3 = id2 + t2
id1 = t3
Code Optimizer
t1 = id3 * 60.0
id1 = id2 + t1
Code Generator
LDF R2, id3
MULF R2, R2, #60.0
LDF R1, id2
ADDF R1, R1, R2
STF id1, R1
1 position ...
2 initial ...
3 rate ...
SYMBOL TABLE

Syntax Analysis

- **The second phase of the compiler is *syntax analysis* or *parsing*.**
- **The parser uses the first components of the tokens produced by the lexical analyzer to create a tree-like intermediate representation that depicts the grammatical structure of the token stream.**

Semantic Analysis

- **The *semantic analyzer* uses the syntax tree and the information in the symbol table to check the source program for semantic consistency with the language definition.**
- **It also gathers type information and saves it in either the syntax tree or the symbol table, for subsequent use during intermediate-code generation.**
- ***Coercions***

Intermediate Code Generation

- **In the process of translating a source program into a target code, a compiler may construct one or more intermediate representations (IRs), having a variety of forms.**
  - ***Syntax trees*: commonly used during syntax and semantic analysis**
  - **After syntax and semantic analysis, compilers generate an explicit lower-level or machine-like IR, a program for an abstract machine.**
  - ***Three-address code (quadruples, triples, indirect triples)***

Code Optimization

- **The machine-independent code optimization phase attempts to improve the intermediate code so that better target code will result.**

Code Generation

- **The code generator takes as input an intermediate representation of the source program and maps it into the target language.**
- **If the target language is machine code, registers or memory locations are selected for each of the variables used by the program.**
- **Then the intermediate instructions are translated into sequence of machine instructions that perform the same task.**

Some Compiler Construction Tools

- **Some commonly used compiler-construction tools include**
  1. ***Parser generators***
     - **Automatically produce syntax analyzers from a grammatical description of a PL.**
  2. ***Scanner generators***
     - **Produce lexical analyzers from a regular-expression description of the tokens of a language.**
  3. ***Syntax-directed translation engines***
     - **Produce a collection of routines for walking a parse tree and generating intermediate code.**
  4. ***Code-generator generators***
     - **Produce a code generator from a collection of rules for translating each operation of intermediate language into the machine language for the target language.**
  5. ***Data-flow analysis engines***
     - **Facilitate the gathering of information about how values are transmitted from one part of a program to each other part. Key part of code optimization.**
  6. ***Compiler-construction toolkits***
     - **Provide an integrated set of routines for constructing various phases of a compiler.**

Applications of Compiler Technology

- **Implementation of high-level programming languages (1.5.1)**
- **Optimizations for computer architectures (1.5.2)**
  - **Parallelism**
  - **Memory hierarchy**
- **Design of new computer architecture (1.5.3)**
  - **RISC**
  - **Specialized architectures**
- **Program translation (1.5.4)**
  - **Binary translation**
  - **Hardware synthesis**
  - **Database query interpreters**
  - **Compiled simulation**
- **Software productivity tools (1.5.5)**
  - **Type checking**
  - **Bounds checking**
  - **Memory-management tools**

Some Programming Languages Basics

- **The static/dynamic distinction (1.6.1)**
  - **If a language uses a policy that allows the compiler to decide an issue, then we say that the language uses a *static policy* or that the issue can be decided at *compiler time*.**
  - **A policy that only allows a decision to be made when we execute the program is said to be a *dynamic policy* or require a decision at *run-time.***
  - ***Scope of declaration***
    - ***Static scope* or *lexical scope***
    - ***dynamic scope***

1.6 Programming Language Basics

- **Environments and states** (1.6.2)

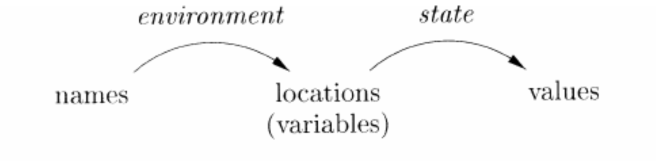


Figure 1.8: Two-stage mapping from names to values

```
...
int i;                  /* global i        */
...
void f(···) {
    int i;              /* local i         */
    ...
    i = 3;              /* use of local i  */
    ...
}
...
    x = i + 1;          /* use of global i */
```

Figure 1.9: Two declarations of the name i

1.6 Programming Language Basics

```
main() {
    int a = 1;
    int b = 1;
    {
        int b = 2;
        {
            int a = 3;
            cout << a << b;
        }
        {
            int b = 4;
            cout << a << b;
        }
        cout << a << b;
    }
    cout << a << b;
}
```

Figure 1.10: Blocks in a C++ program

DECLARATION	SCOPE
`int a = 1;`	$B_1 - B_3$
`int b = 1;`	$B_1 - B_2$
`int b = 2;`	$B_2 - B_4$
`int a = 3;`	B_3
`int b = 4;`	B_4

- **Explicit access control (1.6.4)**
  - **Through the use of keywords like public, private, and protected, OO languages such as C++ or Java provide explicit control over access to member names in a superclass.**

- **Dynamic scope (1.6.5)**
  - **a use of name *x* refers to the declaration of *x* in the most recently called procedure with such a declaration**

    When *x*.*m*() is executed it depends on the class of object denoted by *x* at that time.

    - **A typical example**
      - **There is a class *C* with a method *m*().**
      - ***D* is a subclass of *C*, and *D* has its own method named *m*().**
      - **There is a use of *m* of the form *x.m*(), where *x* is an object of class *C*.**

- **Parameter passing mechanisms (1.6.6)**
  - **call-by-value**
  - **call-by-reference**
  - **call-by-name**
  - **call-by-text-substitution**
  - **call-by-value-result**
- **Aliasing (1.6.7)**

Regular Expressions

• Regular expressions represent languages.

• Languages are set of strings.

• Tokens can be described as regular expressions.

Regular expression

Language

(a) { "a"}

(a) | (b) { "a", "b" }

(a)(b) { "ab" }

(a)* { ""(or ε), "a", "aa", … }

(a)+ { "a", "aa", … }

More Examples

Regular expression

Language

(a)? { “a”, ε }

digit=[0-9] { “0”, “1”, “2”, … }

posint={digit}+ { “3”, “56”, “09”, … }

int=‘-’?{posint} { “-32”, “1024”, … }

real={int}’.’(ε |{posint})

{ “-1.2”, “1.2”, “12.”, … }

[a-zA-Z_][a-zA-Z0-9_]* all identifiers

[^a-z] one char not from a-z

. any single char except \n

Warm-Up Exercise Recognizer Construction

1. Become familiar with the Java language.

a. Download Java J2SE from http://java.sun.com/

b. Follow the instruction to install the Java compiler environment on the computer you will be using.

2. You are to read Section 2.4 first, and write a program to execute on a number of strings. For each string, it should print either “accept”or “reject”.

HW #1 Sample Test Data

a* ; a|b;

; accept ; reject

a; accept a; accept

b; reject b; accept

ab; reject

(a|b)*abb(a|e);

abba; accept

babb; reject

aabba; accept

bbaabb; reject

babbab; reject

(a|c)*(b|e)(a|c)*;

b; accept

aabb; reject

abca; accept

How to Break up Text?

- if8 ??? if8 or if and 8
- if 89 ??? identifier or reserved word if
- Regular expression alone is not enough.
- Disambiguation rules:

1. Longest matching token.
2. Ties resolved by priorities.

Recognizers

- Regular expressions describe the languages that can be recognized by finite automata.
- Translate each token’s regular expression into a non-deterministic finite automaton (NFA).
- Convert the NFA into an equivalent DFA.
- Minimize DFA (to reduce # of states).

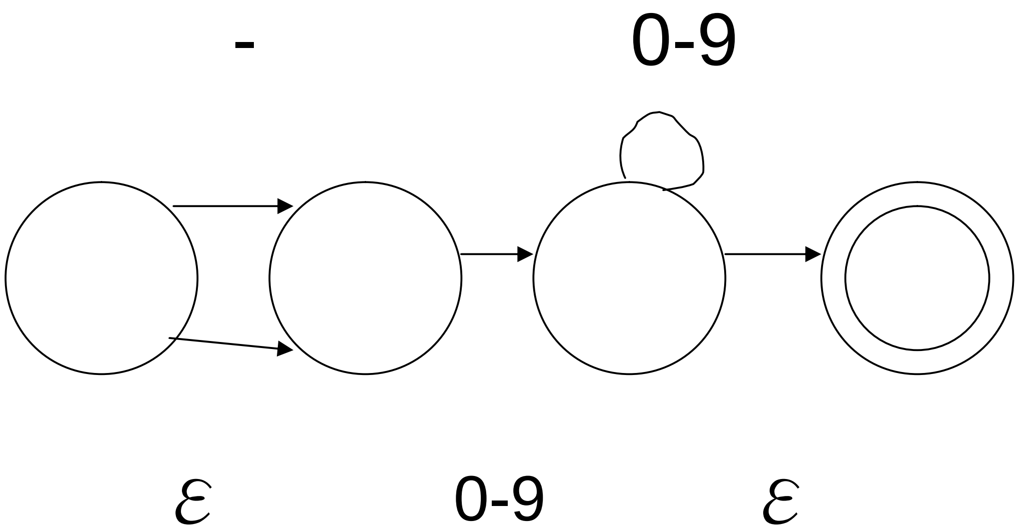
Recognizers (cont)

• Advantage: DFA is efficient for implementation.

• Look up next state using current state & look-ahead character.

Regular Expression to NFA

-?[0-9]+ or (-|**ε**)[0-9][0-9]*



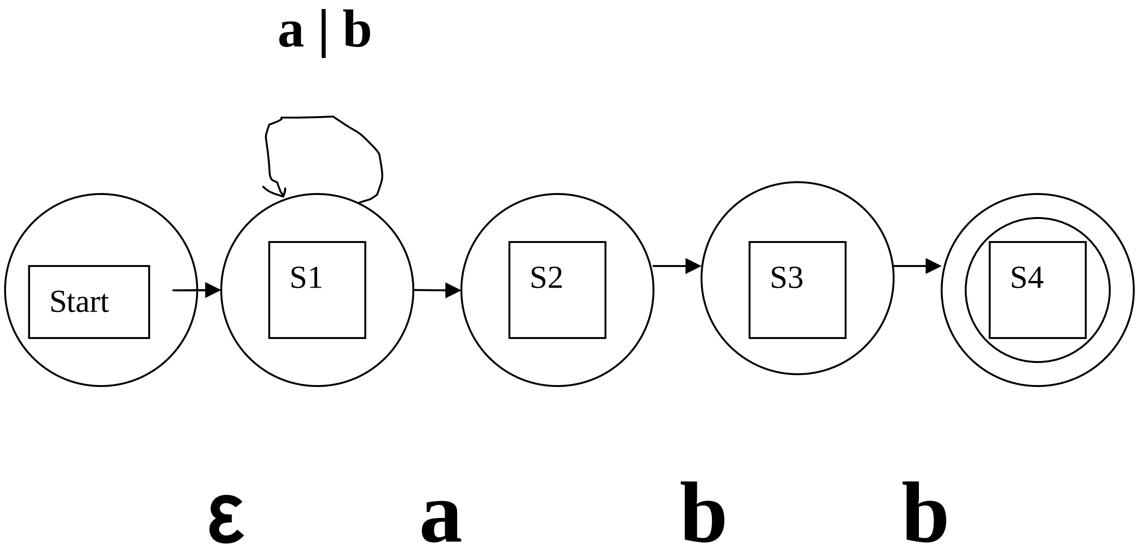
NFA: multiple arcs may have the same labels,

ε transitions do not eat input.

More NFA's

What about the regular expression (a|b)*abb?

1.State start has ε transition



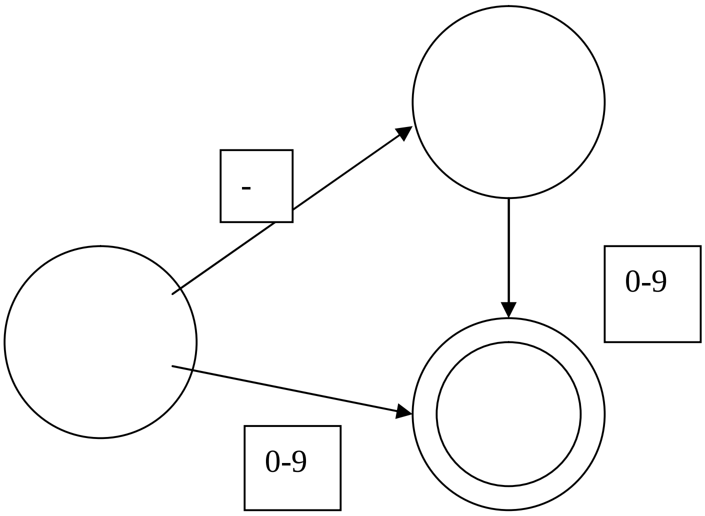
to s1.

2. State s1 has multiple transitions on a.

Different Definition for Accept

A NFA accepts a string x if and only if there is some path through the transition graph from the start state to an accepting state such that the labels along the edges spell x.

NFA to Minimized DFA



- Arc's may not conflict,

no ε transitions.

NFA's versus DFA's

• DFA is a special case of NFA

1. No ε transition.

2. Single-valued transition function.

• DFA can be simulated on a NFA.

• NFA can be simulated on a DFA

1. Simulate sets of simultaneous states.

2. Possible exponential blowup.

Interface to Lexical Analyzer

- Either: Convert entire file to a file of tokens
  - Lexical analyzer is a separate phase
- Or: Parser calls lexical analyzer to get next token
  - This approach avoids extra I/O
  - Parser builds tree incrementally, using successive tokens as tree nodes

Relevant Formalisms: Regular Languages

- Can be defined in terms of
  - Regular (Right Linear, Type 3) Grammars
  - Regular Expressions
  - Finite State Machines (Automata), non-deterministic and deterministic
- All these characterizations are equivalent in expressive power
- Useful for compiler construction, even if hand written

Regular (Right Linear) Grammars

- Regular grammars
  - Non-terminals (arbitrary names)
  - Terminals (characters)
  - Productions limited to the following:
    - Non-terminal $\rightarrow$ terminal
    - Non-terminal $\rightarrow$ terminal Non-terminal
    - Treat character class (e.g. digit) as terminal
  - Regular grammars cannot count (except modulo) or express size limits on identifiers, literals
  - Cannot express proper nesting (parenthesis)
  - Can be generalized by allowing terminal* instead of a single terminal

Regular Grammars

- Grammar for real literals with no exponent
  - REAL ::= digit REAL1
  - REAL1 ::= digit REAL1 (arbitrary size)
  - REAL1 ::= . INTEGER
  - INTEGER ::= digit INTEGER (arbitrary size)
  - INTEGER ::= digit
  - digit ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
- Start symbol is REAL

Regular Expressions

- Regular Expressions (RE) defined by an alphabet Σ (terminal symbols) and three operations
  - Character a for every $a \in \Sigma$
  - Alternation $R_1 \mid R_2$
  - Concatenation $R_1\ R_2$
  - Iteration R^* (zero or more R's)
- Language of RE's = Language of Regular Grammars
  - Regular expressions are more convenient for some applications

Specifying RE's in Unix Tools

- Single characters a b c d \[
- Alternation [bcd] [b-z] ab|cd [^}]
- Any character . (period)
- Repetitions x*y+
- Concatenation abc[d-q]
- Optional RE [0-9]+(\.[0-9]*)?

Finite State Machines

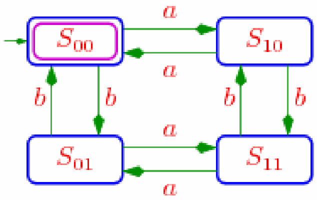
- A language defined by a grammar is a (possibly infinite) set of strings
- An automaton is a device that determines, by reading a string one character at a time, whether the strong belongs to a specified language
- A finite state machine (FSM, NFA) is an automaton that recognizes regular languages (regular expressions)
- Simplest automaton: memory is an element of a finite set

Graphical Representation of an FSM

- A set of labeled states, represented as nodes in a digraph
- Directed edges labeled with a character are drawn between states
- One or more states designated as terminal (accepting)
- One or more states designated as initial
- On reading character $a \in \Sigma$, automaton may move from state S_1 to state S_2 if there exists an a-labeled edge connecting S_1 to S_2
- A string belongs to the language if, after reading the string, the automaton may mover from an initial state to an accepting state

Example: Even Numbers of a's and b's

In the following diagram, we present an NFA which recognizes the language of all strings over $\Sigma : \{a, b\}$ which have an even number of a's and b's.



Note that each of the states has a meaning of its own. For example, all strings which cause the automaton to reach state S_{01} have an even number of a's and an odd number of b's.

In Mathematical Notation

An NFA (FSM) is given by a tuple $A : \langle \Sigma, Q, Q_0, \delta, F \rangle$, where

- Σ — is the alphabet (set of terminal symbols)
- Q — The (finite) set of states
- $Q_0 \subseteq Q$ — Set of initial states
- $\delta : Q \times \Sigma \rightarrow 2^Q$ — The transition function. For each $q \in Q$ and $a \in \Sigma$, $\delta(q, a) \subseteq Q$ is the set of a-successors of q
- $F \subseteq Q$ — Set of accepting states.

Example: The Even-a-Even-b Automaton

For the case of the Even-a-Even-b Automaton the relevant constructs are

- $\Sigma : \{a, b\}$
- $Q : \{S_{00}, S_{01}, S_{10}, S_{11}\}$
- $Q_0 : \{S_{00}\}$
- The transition function δ is given by the table

$\delta(q, c)$:

	S_{00}	S_{01}	S_{10}	S_{11}
a	$\{S_{10}\}$	$\{S_{11}\}$	$\{S_{00}\}$	$\{S_{01}\}$
b	$\{S_{01}\}$	$\{S_{00}\}$	$\{S_{11}\}$	$\{S_{10}\}$

- $F : \{S_{00}\}$

Partition the input strings into 4 equivalence classes:

S_{00} – Even number of a’s and b’s -- final (accept) state
S_{01} – Even number of a’s, Odd number of b’s
S_{10} – Odd number of a’s, Even number of b’s
S_{11} – Odd number of a’s and b’s

Runs and Acceptance

Let $A : \langle Q, Q_0, \delta, F \rangle$ be an NFA over the alphabet Σ, and let $\sigma : a_1, \ldots, a_k$ be a word (string) over Σ. A run of A over σ is a sequence of states $r : q_0, q_1, \ldots, q_k$ such that

- $q_0 \in Q_0$, and
- $q_{i+1} \in \delta(q_i, a_{i+1})$, for each $i = 0, \ldots, k-1$.

The run r is called accepting if $q_k \in F$. The word σ is accepted by A if there exists a run r over σ, such that r is accepting.

The language defined by A, denoted $L(A)$, is the set of all words which are accepted by A.

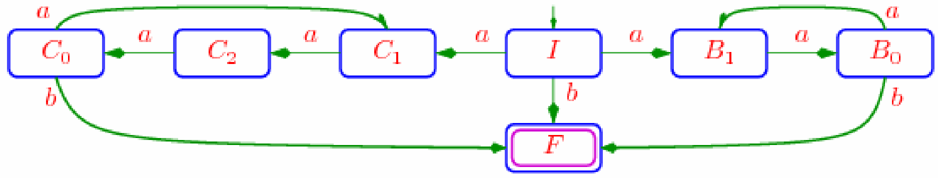
A language L is said to be regular if there exists an NFA A, such that $L = L(A)$.

Deterministic and Non-Deterministic Automata

An automaton $A : \langle Q, Q_0, \delta, F\rangle$ such that $|\delta(q,a)| = 1$ for all $q \in Q, a \in \Sigma$ is called deterministic. In such a case, we refer to the automaton as a deterministic finite automaton (DFA) and represent the transition function as $\delta : Q \times \Sigma \to Q$.

Non-deterministic automata (NFA's) are often more succinct than their deterministic counterparts.

For example, following is an NFA that recognizes the language $((aa)^* + (aaa)^*)b$ which consists of all words having a string of a's of length which is a multiple of 2 or of 3 followed by a b.



Determinization of an NFA

Claim 1. *A language L is regular iff it is recognizable by a DFA*

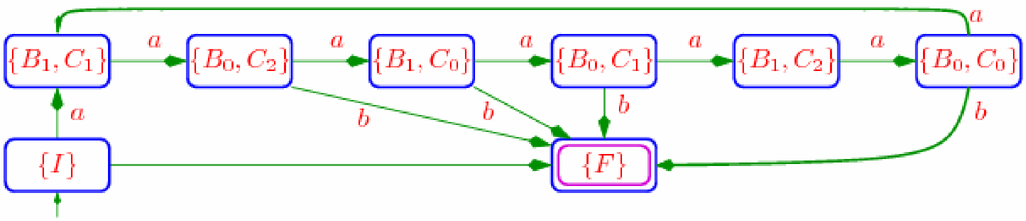
Namely, for every NFA $A : \langle Q, Q_0, \delta, F\rangle$, there exists a DFA $\widetilde{A} : \langle \widetilde{Q}, \widetilde{Q}_0, \Delta, \widetilde{F}\rangle$ such that $L(A) = L(\widetilde{A})$.

Define the automaton $\widetilde{A}$ as follows:

- $\widetilde{Q} = 2^Q$. That is, a state of $\widetilde{A}$ is a set of A-states.
- $\widetilde{Q}_0 = Q_0$. The initial state of $\widetilde{A}$ is the set of all initial A-states.
- $\Delta(S, a) = \bigcup_{q \in S} \delta(q, a)$. The state $\Delta(S, a)$ contains all the A-states which are a-successors of some state in S.
- $\widetilde{F} = \{S \mid S \cap F \neq \emptyset\}$. A state (set) $S \subseteq Q$ is accepting if it contains some accepting A-state.

Apply to $((aa)^* + (aaa)^*)b$

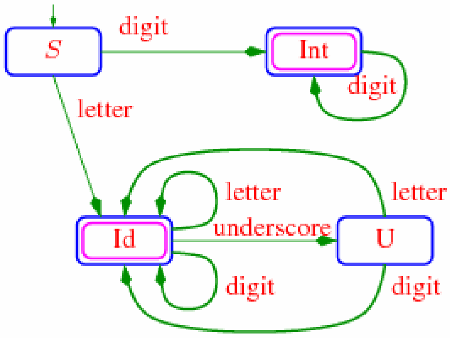
When we apply the determinization procedure to the NFA which recognizes the language $((aa)^* + (aaa)^*)b$, we obtain



Example: Recognizing Identifier/Integer

The following DFA recognizes (and classifies) Identifier/Integer

$$
\begin{array}{lcl}
S & ::= & \ell \ \mathsf{Id} \mid \ell \\
S & ::= & d \ \mathsf{Int} \mid d \\
\mathsf{Int} & ::= & d \ \mathsf{Int} \mid d \\
\mathsf{Id} & ::= & d \ \mathsf{Id} \mid \ell \ \mathsf{Id} \mid \_\mathsf{U} \\
\mathsf{Id} & ::= & d \mid \ell \\
\mathsf{U} & ::= & d \ \mathsf{Id} \mid \ell \ \mathsf{Id} \mid d \mid \ell
\end{array}
$$



From Regular Expressions to NFA

There are several approaches to the construction of an Automaton correpsonding to an RE. We will present one based on the notion of a derivative.

For a regular expression R and a letter $a \in \Sigma$, the derivative of R relative to a, denoted $\frac{\partial R}{\partial a}$ is a set of RE's $\{R_1, \ldots, R_k\}$ such that, for every word $w \in \Sigma^*$,

$$aw \in L(R) \quad \text{iff} \quad w \in L(R_1 + \cdots + R_k)$$

Computing the Derivatives

- $\frac{\partial a}{\partial a} = \{\epsilon\} \qquad \frac{\partial b}{\partial a} = \emptyset$
- $\frac{\partial(R_1+R_2)}{\partial a} = \frac{\partial R_1}{\partial a} \cup \frac{\partial R_2}{\partial a}$
- $\frac{\partial R^*}{\partial a} = \frac{\partial(RR^*)}{\partial a}$
- $\frac{\partial(aR)}{\partial a} = \{R\} \qquad \frac{\partial(bR)}{\partial a} = \emptyset$
- $\frac{\partial((R_1+R_2)R)}{\partial a} = \frac{\partial(R_1R)}{\partial a} \cup \frac{\partial(R_2R)}{\partial a}$
- $\frac{\partial((R_1R_2)R)}{\partial a} = \frac{\partial(R_1(R_2R))}{\partial a}$
- $\frac{\partial(R_1^*R_2)}{\partial a} = \frac{\partial(R_1(R_1^*R_2))}{\partial a} \cup \frac{\partial R_2}{\partial a}$

Apply to Even-a-Even-b

Consider

$$E = (aa + bb + (ab + ba)(aa + bb)^*(ab + ba))^*$$

Or abbreviating

$$E = X^*$$
$$X = aa + bb + (ab + ba)Y^*Z$$
$$Y = aa + bb \qquad Z = ab + ba$$

Derivatives are

R	$\frac{\partial R}{\partial a}$	$\frac{\partial R}{\partial b}$
X^*	$\{aX^*,\ bY^*ZX^*\}$	$\{bX^*,\ aY^*ZX^*\}$
aX^*	$\{X^*\}$	$\emptyset$
bX^*	$\emptyset$	$\{X^*\}$
aY^*ZX^*	$\{Y^*ZX^*\}$	$\emptyset$
bY^*ZX^*	$\emptyset$	$\{Y^*ZX^*\}$
Y^*ZX^*	$\{aY^*ZX^*,\ bX^*\}$	$\{bY^*ZX^*,\ aX^*\}$

Construct an NFA from an RE

The closure of a regular expression R, denoted $Cl(R)$, is the set of expressions that arise through successive derivatives.

For example, the closure of the expression

$$(X : aa + bb + (ab + ba)(Y : aa + bb)^*(Z : ab + ba))^*$$

is given by the set

$$\{X^*,\ aX^*,\ bX^*,\ aY^*ZX^*,\ bY^*ZX^*,\ Y^*ZX^*\}$$

We construct an NFA for expression R as follows:

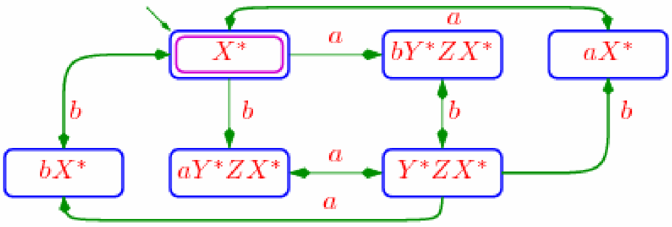
- States are the elements of $Cl(R)$
- The Initial state is R
- The accepting states are ϵ, and any state of the form $U^* \in Cl(R)$
- For each expressions $U, V \in Cl(R)$ and letter $a \in \Sigma$, such that $V \in \frac{\partial U}{\partial a}$, we draw an a-labeled edge connecting U to V

Apply to Even-a-Even-b

From the derivative table

R	$\frac{\partial R}{\partial a}$	$\frac{\partial R}{\partial b}$
X^*	$\{aX^*,\ bY^*ZX^*\}$	$\{bX^*,\ aY^*ZX^*\}$
aX^*	$\{X^*\}$	$\emptyset$
bX^*	$\emptyset$	$\{X^*\}$
aY^*ZX^*	$\{Y^*ZX^*\}$	$\emptyset$
bY^*ZX^*	$\emptyset$	$\{Y^*ZX^*\}$
Y^*ZX^*	$\{aY^*ZX^*,\ bX^*\}$	$\{bY^*ZX^*,\ aX^*\}$

we construct the following NFA:



States represent 6 equivalence classes of strings of a’s and b’s:

X* - even number of a’s and b’s

bX* - odd number of b’s, even number of a’s, not followed by an “a”

aY*ZX* - odd number of b’s, even number of a’s, followed by an “a”

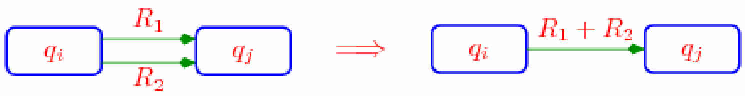
Y*ZX* - odd number of a’s and b’s

bY*ZX* - odd number of a’s, even number of b’s, followed by a “b”

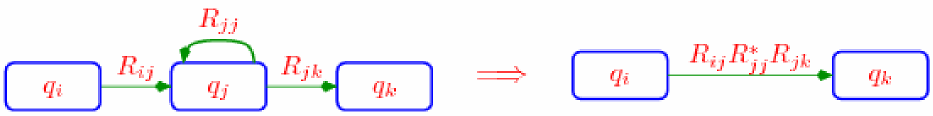
aX* - odd number of a’s, even number of b’s, not followed by a “b”

From NFA to RE

- With no loss of generality, assume that the initial state is q_1 and the accepting state is either q_2 or q_1
- We consider generalized NFA in which there exists at most one edge between q_i and q_j, but it may be labeled by an RE
- To achieve this representation, we may use the transformation

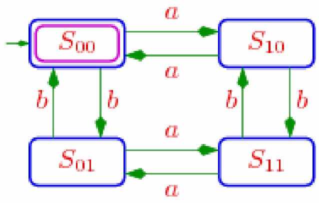


- Starting with q_n and going down incrementally, we successively eliminate each of the states, using the transformation

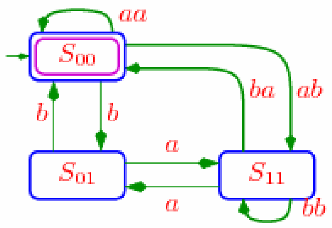


Apply to Even-a-Even-b

Start with



Eliminate S_{10} to get



To eliminate a node, for each (in, out) pair of edges, form the regular expressions

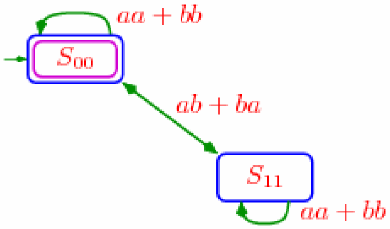
(in out) -- no self loop

or (in self* out) -- self loop

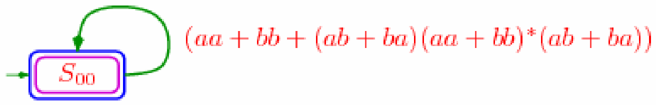
and replace each path by an edge with the corresponding regular expression.

Elimination Continued

Next, we eliminate S_{01} and get



Finally, we eliminate S_{11} and obtain



We conclude that the corresponding RE is

$$(aa + bb + (ab + ba)(aa + bb)^*(ab + ba))^*$$

To eliminate a node, for each (in, out) pair of edges, form the regular expressions

(in out) -- no self loop

or (in self* out) -- self loop

and replace each path by an edge with the corresponding regular expression.

Implementing the Scanner

- Three Methods
  - Hand-Coded approach
    - Draw DFSM, then implement with loop and case statements
  - Hybrid approach
    - Define tokens using regular expressions, convert to NFSM, apply algorithm to obtain minimal DFSM
    - Hand-code resulting DFSM
  - Automated approach
    - Use regular expressions as input to lexical scanner generator (e.g. FLEX)

Automatic Scanner Construction

- FLEX builds a transition table, indexed by state and by character.
- Code gets transition from table:

```
Tab: array [State, Character] of State := . . .
begin
    while More_Input do
        Curstate := Tab[Curstate,Next_Char];
        if Curstate = Error_State
            then . . .
            else Do_Actions(Curstate);
    end while;
```

Flex General Format

- Input to FLEX is a set of rules:
  - Regexp actions (a C statement)
  - Regexp actions (a C statement)
  - ...
- FLEX scans the longest matching string
  - And executes the corresponding actions
  - Among strings of equal length, FLEX prefers the Regexp which appears earlier in the list

An Example of a Flex Script

```
DIGIT                [0–9]
ID                   [a–z][a–z0–9]*
%%
{DIGIT}+          {printf(“an integer %s(%d)\n”,yytext,atoi(yytext));}
{DIGIT}+“.”DIGIT*    {printf(“a float %s(%g)\n”,yytext,atof(yytext));}
if | then | begin | end | procedure | function | program
                     {printf(“a keyword: %s\n”, yytext;}
ID                   {printf(“an identifier: %s\n”, yytext);}
“+”| “-”| “*”| “/”   {printf(“an operator: %s\n”, yytext);}
“=”| “:”| “;”| “:=”  {printf(“a separator: %s\n”, yytext);}
“{” [ˆ}]* “}”        /* eat up Pascal-like comments */
[\t\n]+              /* eat white space */
.                    {printf(“unrecognized character %s\n”, yytext);}
%%
```

More Properties of Regular Languages

The following claim enables us to show that a given language is not regular.

Claim 2. [Pumping Lemma]

For every regular language L, there exists a constant N, such that if $w \in L$ and $|w| > N$, then $w = xyz$ for some $|y| > 0$ such that $xy^r z \in L$ for every $r \geq 0$.

Let $A : \langle Q, Q_0, \delta, F \rangle$ be the DFA recognizing L. We take $N = |Q|$. Assume that $w = a_1 a_2 \cdots a_n$ where $n > N$. Let $q_0, q_1, \ldots, q_n$ be the (unique) A-run accepting w. Since $n > |Q|$, there must exist $i < j$ such that $q_i = q_j$. We claim that, for an arbitrary $r \geq 0$, the word $a_1 \cdots a_i (a_{i+1} \cdots a_j)^r a_{j+1} \cdots a_n$ belongs to L, since its run is accepting. ∎

Illustrate by Proving Irregularity

Consider the language $L : \{a^i b^i \mid i \geq 0\}$.

We will show that L is not regular. Suppose L were regular, and let N be the constant guaranteed by the pumping lemma. Consider the word $w = a^N b^N$. By the lemma, w should be decomposable into $w = xyz$ such that $xy^r z \in L$ for every $r \geq 0$. By considering the three possible cases of y having one of the forms $a^i, a^i b^j, b^j$, with $i, j > 0$, we see that $xy^2 z \notin L$ for all three cases. We conclude that L cannot be regular.

Closure Properties

Regular languages are closed under the following operations on languages:

- **Union** — For regular expressions R_1, R_2,
  $L(R_1) \cup L(R_2) = L(R_1 + R_2)$.
- **Complementation** — For DFA $A : \langle Q, Q_0, \delta, F \rangle$,
  $\Sigma^* - L(A) = L(\langle Q, Q_0, \delta, Q - F \rangle)$
- **Intersection** — For languages L_1, L_2,
  $L_1 \cap L_2 = \overline{(\overline{L_1} \cup \overline{L_2})}$. A direct construction constructs a DFA for $L_1 \cap L_2$ from the DFA's of L_1 and L_2.
- **Reversal** — By reversing a regular expression.
- **Substitution** — Substituting a regular language for a letter. Can be applied to regular expressions.