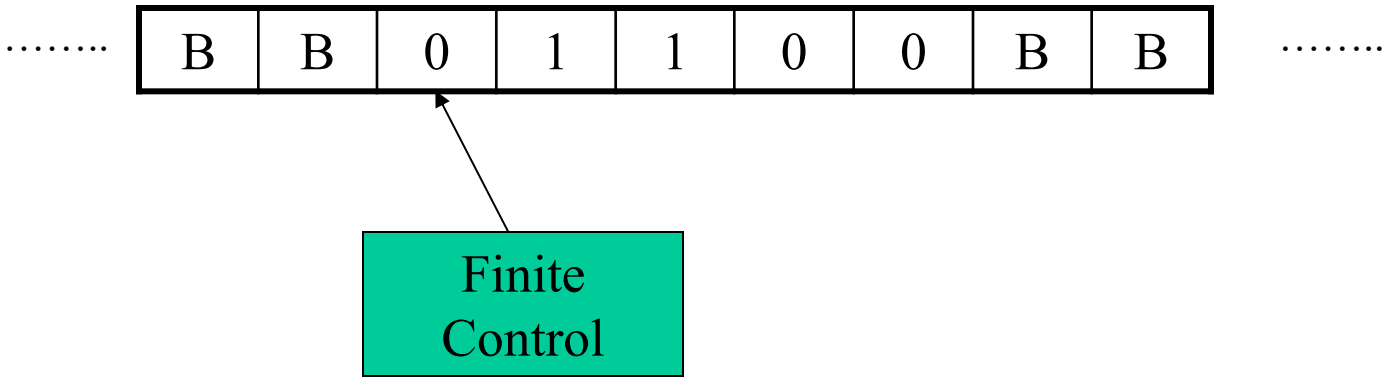
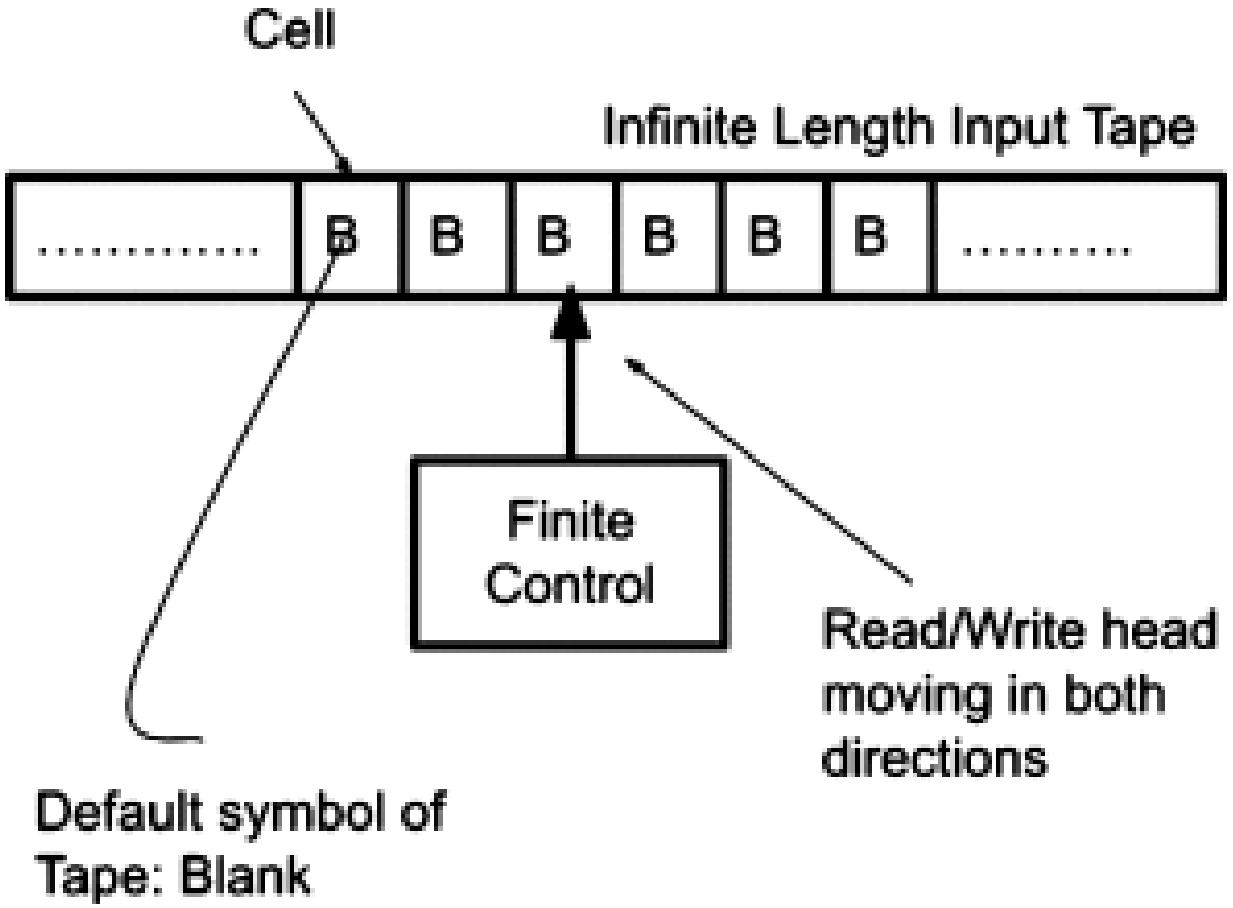
Turing Machine (TM)



- Two-way, infinite tape, broken into cells, each containing one symbol.
- Two-way, read/write tape head.
- An input string is placed on the tape, padded to the left and right infinitely with blanks, read/write head is positioned at the left end of input string.
- Finite control, i.e., a program, containing the position of the read head, current symbol being scanned, and the current state.
- In one move, depending on the current state and the current symbol being scanned, the TM 1) changes state, 2) prints a symbol over the cell being scanned, and 3) moves its' tape head one cell left or right.

Turing Machine



Formal Definition of a TM

- A TM is a seven-tuple: $M = (Q, \Sigma, \Gamma, \delta, q_0, B, F)$

Q	A finite set of states
Σ	A finite input alphabet, which is a subset of $\Gamma - \{B\}$
Γ	A finite tape alphabet, which is a strict superset of Σ
B	A distinguished blank symbol, which is in Γ
q_0	The initial/starting state, q_0 is in Q
F	A set of final/accepting states, which is a subset of Q
δ	A transition function, which is a *mapping* (i.e., may be undefined) from $Q \times \Gamma \rightarrow Q \times \Gamma \times \{L,R\}$

Intuitively, $\delta(q,s)$ specifies the next state, symbol to be written, and the direction of tape head movement by M after reading symbol s while in state q.

Formal Definitions for TMs

- Let $M = (Q, \Sigma, \Gamma, \delta, q_0, B, F)$ be a TM.

- **Definition:** An *instantaneous description* (ID) is a triple $\alpha_1 q \alpha_2$, where:

  - q, the current state, is in Q
  - $\alpha_1\alpha_2$, is in Γ^*, and is the current tape contents up to the rightmost non-blank symbol, or the symbol to the left of the tape head, whichever is rightmost
  - The tape head is currently scanning the first symbol of α_2
  - At the start of a computation $\alpha_1 = \boldsymbol{\varepsilon}$
  - If $\alpha_2 = \boldsymbol{\varepsilon}$ then a blank is being scanned

- Suppose the following is the current ID of a TM

$$x_1x_2\ldots x_{i-1}qx_ix_{i+1}\ldots x_n$$

Case 1) $\delta(q, x_i) = (p, y, L)$

$$x_1x_2\ldots x_{i-1}qx_ix_{i+1}\ldots x_n \vdash x_1x_2\ldots x_{i-2}px_{i-1}yx_{i+1}\ldots x_n$$

Case 2) $\delta(q, x_i) = (p, y, R)$

$$x_1x_2\ldots x_{i-1}qx_ix_{i+1}\ldots x_n \vdash x_1x_2\ldots x_{i-1}ypx_{i+1}\ldots x_n$$

- **Definition:** Let $M = (Q, \Sigma, \Gamma, \delta, q_0, B, F)$ be a TM, and let w be a string in Σ^*. Then w is *accepted* by M iff

  $$q_0 w \vdash^* \alpha_1 p \alpha_2$$

  where p is in F and α_1 and α_2 are in Γ^*

- **Definition:** Let $M = (Q, \Sigma, \Gamma, \delta, q_0, B, F)$ be a TM. The *language accepted by M*, denoted L(M), is the set

  $$\{w \mid w \text{ is in } \Sigma^* \text{ and } w \text{ is accepted by } M\}$$

- **Example #1:** {w | w is in {0,1}* and w ends with a 0}

0

00

10

10110

Not ε

$Q = \{q_0, q_1, q_2\}$

$\Gamma = \{0, 1, B\}$

$\Sigma = \{0, 1\}$

$F = \{q_2\}$

δ:

q_0 is the start state and the "scan right" state, until hits B
q_1 is the verify 0 state
q_2 is the final state

	0	1	B
->q_0	(q_0, 0, R)	(q_0, 1, R)	(q_1, B, L)
q_1	(q_2, 0, R)	-	-
q_2*	-	-	-

- **Example #2:** $\{0^n1^n \mid n \geq 1\}$

	0	1	X	Y	B
q_0	(q_1, X, R)	-	-	(q_3, Y, R)	-
q_1	$(q_1, 0, R)$	(q_2, Y, L)	-	(q_1, Y, R)	-
q_2	$(q_2, 0, L)$	-	(q_0, X, R)	(q_2, Y, L)	-
q_3	-	-	-	(q_3, Y, R)	(q_4, B, R)
q_4	-	-	-	-	-

- **Sample Computation:** (on 0011), *presume state q looks rightward*

$q_0$0011BB.. |— X$q_1$011
|— X0$q_1$11
|— X$q_2$0Y1
|— q_2X0Y1
|— X$q_0$0Y1
|— XXq_1Y1
|— XXY$q_1$1
|— XXq_2YY
|— Xq_2XYY
|— XXq_0YY
|— XXYq_3Y B…
|— XXYYq_3 BB…
|— XXYYBq_4

- **Roger Ballard's TM for Example #2, without any extra Tape Symbol:** $\{0^n1^n \mid n \geq 0\}$

	0	1	B
q_0	(q_1, B, R)		(q_4, B, R)
q_1	$(q_1, 0, R)$	$(q_1, 1, R)$	(q_2, B, L)
q_2	-	(q_3, B, L)	-
q_3	$(q_3, 0, L)$	$(q_3, 1, L)$	(q_0, B, R)
q_4^*	-	-	-

Logic: Keep deleting 0 and corresponding 1 from extreme ends, until none left.

- q_0 deletes a leftmost 0 and let q_1 scan through end of string, q_0 accepts on epsilon
- q_1 scans over the string and makes q_2 expecting 1 on the left
- q_2 deletes 1 and let q_3 "scan left" looking for the start of current string
- q_3 lets q_0 start the next iteration
- q_4 is the final state

Any bug?

Try on:

000111	00
11	001
011	

- **And his example of a correct TM for the language that goes on infinite loop outside language:** $\{0^n1^n \mid n \geq 0\}$

	0	1	B
q_0	(q_1, B, R)	$(q_3, 1, L)$	(q_4, B, R)
q_1	$(q_1, 0, R)$	$(q_1, 1, R)$	(q_2, B, L)
q_2	-	(q_3, B, L)	-
q_3	$(q_3, 0, L)$	$(q_3, 1, L)$	(q_0, B, R)
q_4^*	-	-	-

Logic: This machine still works correctly for all strings in the language, but start a string with 1 (not in the language), and it loops on B1 for ever.

$\{ \omega \omega^R \mid \omega \in (a+b)^* \}$

$Q = \{ q_0\ q_1\ q_2\ q_3\ q_4\ q_5\ q_6\ q_7\ q_8 \}$

$\Sigma = \{ a, b \}$

$\Gamma = \{ a, b, X, Y, B \}$

$q_0 = q_0$

$B = B$

$F = \{ q_8 \}$

δ:

	a	b	X	Y	B
q_0	(q_1, X, R)	(q_4, Y, R)	(q_7, X, R)	(q_7, Y, R)	
q_1	(q_1, a, R)	(q_1, b, R)	(q_2, X, L)	(q_2, Y, L)	(q_2, B, L)
q_2	(q_3, X, L)				
q_3	(q_3, a, L)	(q_3, b, L)	(q_0, X, R)		
q_4	(q_4, a, R)	(q_4, b, R)	(q_5, X, L)	(q_5, Y, L)	
q_5		(q_6, Y, L)			
q_6	(q_6, a, R)	(q_6, b, L)		(q_0, Y, R)	
q_7			(q_7, X, R)	(q_7, Y, R)	(q_8, B, R)
q_8	←		accept		→

$\{ w c w^R \mid w \in (a+b)^* \}$

ex a b b c b b a or
    a b a c a b a

$Q = \{q_0, q_1, q_2, q_3, q_4, q_5, q_6, q_7, q_8, q_9, q_{10}\}$

$\Sigma = \{a, b, c, \text{B}\}$

$\Gamma = \{a, b, c, x, y, \text{B}\}$

$q_0 = q_0$

$B = B$

$F = \{q_{10}\}$

f:	a	b	c	x	y	B
q_0	$(q_1 x R)$	$(q_5 y R)$	$(q_9 c R)$			
q_1	$(q_1 a R)$	$(q_1 b R)$	$(q_2 c R)$			
q_2	$(q_2 a R)$	$(q_2 b R)$			$(q_3 y L)$	$(q_2 B L)$
q_3	$(q_4 x L)$					
q_4	$(q_4 a L)$	$(q_4 b L)$	$(q_4 c L)$	$(q_0 x R)$		
q_5	$(q_5 a R)$	$(q_5 b R)$	$(q_6 c R)$			
q_6	$(q_6 a R)$	$(q_6 b R)$		$(q_7 x L)$	$(q_7 y L)$	
q_7		$(q_0 y L)$				
q_8	$(q_8 a L)$	$(q_8 b L)$	$(q_8 c L)$		$(q_8 y R)$	
q_9				$(q_9 x R)$	$(q_9 y R)$	(q_{10}, B, R)
q_{10}	←		accepted			→

$a^n b^n c^n \mid n \geq 1$

$Q = \{q_0, q_1, q_2, q_3, q_4, q_5\}$

$\Sigma = \{a, b\}$

$\Gamma = \{a, b, x, y, z, B\}$

$q_0 = q_0$

$B = B$

$F = \{q_5\}$

δ:

	a	b	c	x	y	z	B
q_0	(q_1, x, R)				(q_4, y, R)		
q_1	(q_1, a, R)	(q_2, y, R)			(q_1, y, R)		
q_2		(q_2, b, R)	(q_3, z, L)			(q_2, z, R)	
q_3	(q_3, a, L)	(q_3, b, L)		(q_0, x, R)	(q_3, y, L)	(q_3, z, L)	
q_4					(q_4, y, R)	(q_4, z, R)	(q_5, B, R)
q_5	←		accepted				→

$\{a^n b^{2n} \mid n \geq 1\}$

$Q = \{q_0, q_1, q_2, q_3, q_4\}$

$\Sigma = \{a, b\}$

$\Gamma = \{a, b, x, y, B\}$

$q_0 = q_0$

$B = B$

$F = \{q_4\}$

δ:

	a	b	x	y	B
q_0	(q_1, x, R)				
q_1	(q_1, a, R)	(q_2, y, R)		(q_1, y, R)	
q_2		(q_3, y, R)			
q_3	(q_3, a, L)	(q_3, b, L)	(q_0, x, R)	(q_3, y, L)	(q_4, B, R)
q_4	←		accepted		→

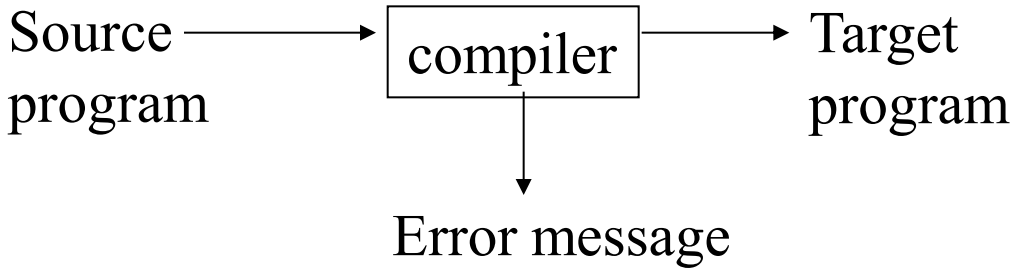
Design a TM to compute addition of two unary numbers

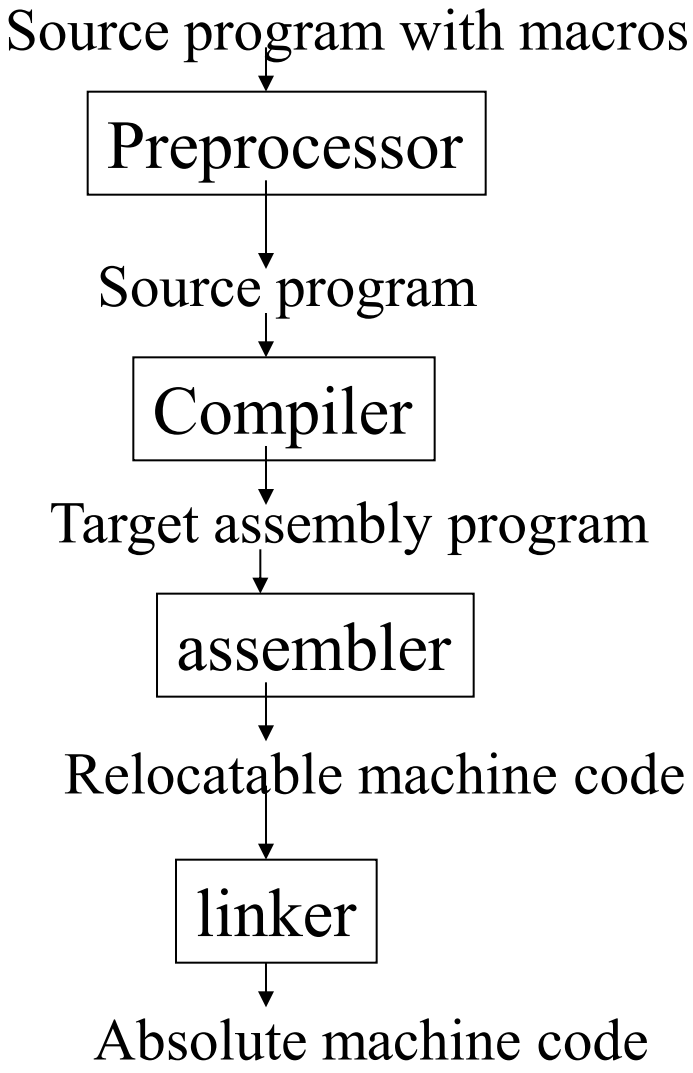
- Q = {q0, q1, q2, q3}
- Σ = {0, #}
- Γ = {0, #, B}
- δ ⇒ Given by the above transition diagram q
- 0 = {q0}
- B = {B}
- F = {q3}

Transition table, δ

Q \ Γ	0	#	B
→ q_0	$(q_0, 0, R)$	$(q_0, 0, R)$	-
q_1	$(q_1, 0, R)$	-	(q_2, B, R)
q_2	(q_3, B, N)	-	-
* q_3	ϕ	ϕ	ϕ

- What is a compiler?
  - A program that reads a program written in one language (source language) and translates it into an equivalent program in another language (target language).
    - Two components
      - Understand the program (make sure it is correct)
      - Rewrite the program in the target language.
  - Traditionally, the source language is a high level language and the target language is a low level language (machine code).





A typical compilation process

Other Applications

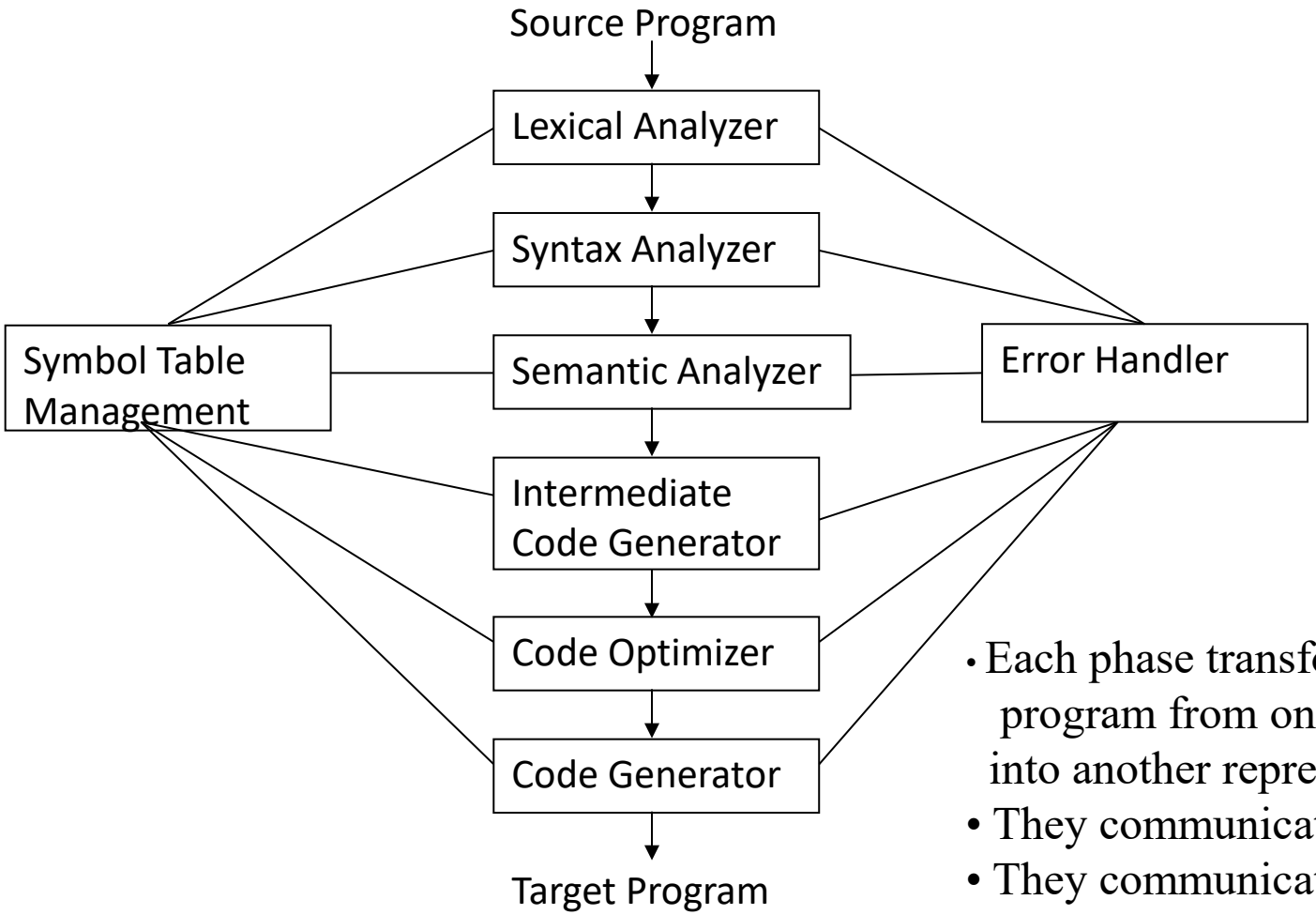
- In addition to the development of a compiler, the techniques used in compiler design can be applicable to many problems in computer science.
  - Techniques used in a lexical analyzer can be used in text editors, information retrieval system, and pattern recognition programs.
  - Techniques used in a parser can be used in a query processing system such as SQL.
  - Most of the techniques used in compiler design can be used in Natural Language Processing (NLP) systems.

Major Parts of Compilers

- There are two major parts of a compiler: **Analysis** and **Synthesis**

- In analysis phase, an intermediate representation is created from the given source program.
  - Lexical Analyzer, Syntax Analyzer and Semantic Analyzer are the parts of this phase.
- In synthesis phase, the equivalent target program is created from this intermediate representation.
  - Intermediate Code Generator, Code Generator, and Code Optimizer are the parts of this phase.

Phases of Complier



- Each phase transforms the source program from one representation into another representation.
- They communicate with error handler.
- They communicate with the symbol table.

Lexical Analyzer

- **Lexical Analyzer** reads the source program character by character and returns the *tokens* of the source program.
- A *token* describes a pattern of characters having same meaning in the source program. (such as identifiers, operators, keywords, numbers, delimeters and so on)

  Ex: position : = initial + rate * 60 => id1 := id2 + id3 * 60

  tokens:

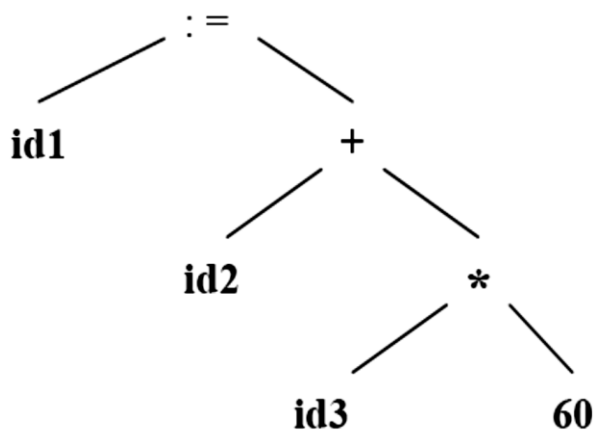
position	identifier
:=	assignment operator
initial	identifier
+	add operator
rate	identifier
*	multiply operator
60	a number

- Puts information about identifiers into the symbol table.
- Regular expressions are used to describe tokens (lexical constructs).
- A (Deterministic) Finite State Automaton can be used in the implementation of a lexical analyzer.

Syntax Analyzer

- A **Syntax Analyzer** creates the syntactic structure (generally a parse tree) of the given program.
- A syntax analyzer is also called as a **parser**.
- A **parse tree** describes a syntactic structure.

Example : position : = initial + rate * 60



- each interior node represents an operation and
- the children of the node represent the arguments of the operation.

Syntax Analyzer (CFG)

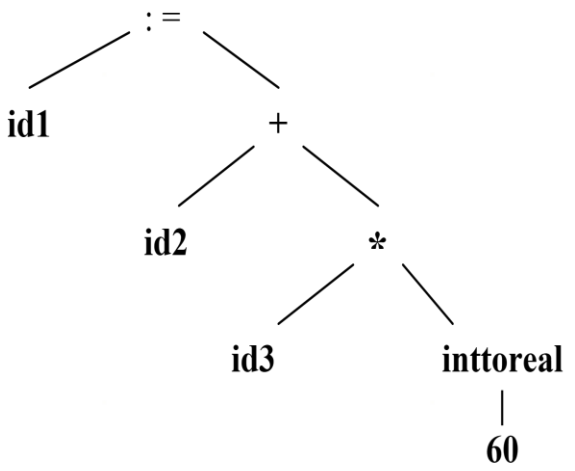
- The syntax of a language is specified by a **context free grammar** (CFG).
- The rules in a CFG are mostly recursive.
- A syntax analyzer checks whether a given program satisfies the rules implied by a CFG or not.
  - If it satisfies, the syntax analyzer creates a parse tree for the given program.
- Ex: We use BNF (Backus Naur Form) to specify a CFG

```
assgstmt    ->  identifier  := expression
expression  ->  identifier
expression  ->  number
expression  ->  expression  *  expression
```

Semantic Analyzer

- A semantic analyzer checks the source program for semantic errors and collects the type information for the code generation.
- Type-checking is an important part of semantic analyzer.
- Normally semantic information cannot be represented by a context-free language used in syntax analyzers.
- Context-free grammars used in the syntax analysis are integrated with attributes (semantic rules)
  - the result is a syntax-directed translation
  - Attribute grammars

Ex:



Intermediate Code Generation

- A compiler may produce an explicit intermediate codes representing the source program.
- These intermediate codes are generally machine (architecture) independent. But the level of intermediate codes is close to the level of machine codes.
- Three address codes consist of a sequence of instructions, each of which has at most three operands.
- Ex: id1 := id2 + id3 * 60.0

Three address code as

```
temp1 : = inttoreal (60)
temp2 := id3 * temp1
temp3 := id2 + temp2
id1     := temp3
```

Code Optimizer (for Intermediate Code Generator)

- The code optimizer optimizes the code produced by the intermediate code generator in the terms of time and space.
- To improve the intermediate code, so that faster running machine code can be generated.
- Ex: Three address code after optimization is
  - temp1 := id3 * 60.0
  - id1 := id2 + temp1

Code Generator

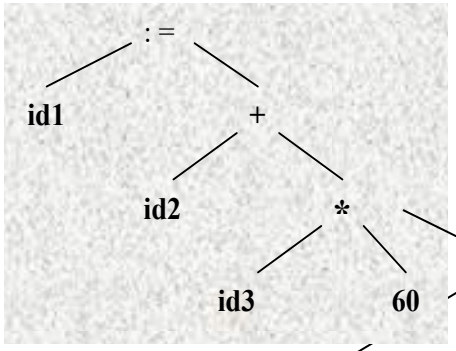
- Produces the target language in a specific architecture.
- The target program is normally is a relocatable object file containing the machine codes.

- Ex: assume that we have an architecture with instructions whose at least one of its operands is a machine register (8086 conversion code)
  - MOVF id3, R2
  - MULF #60.0, R2
  - MOVF id2, R1
  - ADDF R2, R1
  - MOVF R1, id1

- **Symbol Table** is a data-structure maintained throughout all the phases of a compiler.
- All the identifier's names along with their types are stored here.
- The symbol table makes it easier for the compiler to quickly search the identifier record and retrieve it.
- It is also used for scope management. commonly used data structures for implementing symbol table are
  - Linear Table
  - List
  - Tree
  - Hash table

- An **Error Handling Routine** is a set of instructions implemented within a compiler to detect and handle errors during compilation.
- An error is a blank entry in the symbol table.
- The Errors may occur in all phases of the compiler.
- Whenever a phase of the compiler discovers an error, it must report it to the error handler, which issues an appropriate diagnostic message.
- The main components of the Error Handling Routine include Error Detection and Error Reporting.

Output of each Phases



Source Program

position : = initial + rate * 60

Lexical Analyzer

id1 := id2 + id3 * 60

Syntax Analyzer

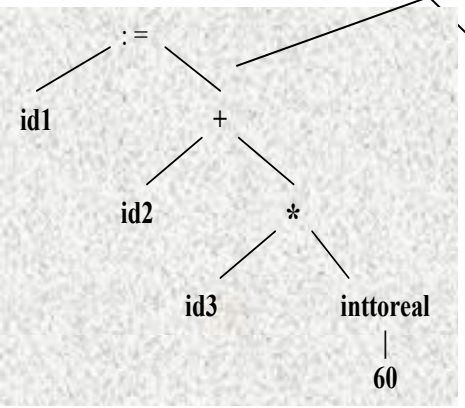
Symbol Table Management

Semantic Analyzer

Error Handler

temp1 : = inttoreal (60)
temp2 := id3 * temp1
temp3 := id2 + temp2
id1 := temp3

Intermediate Code Generator



Code Optimizer

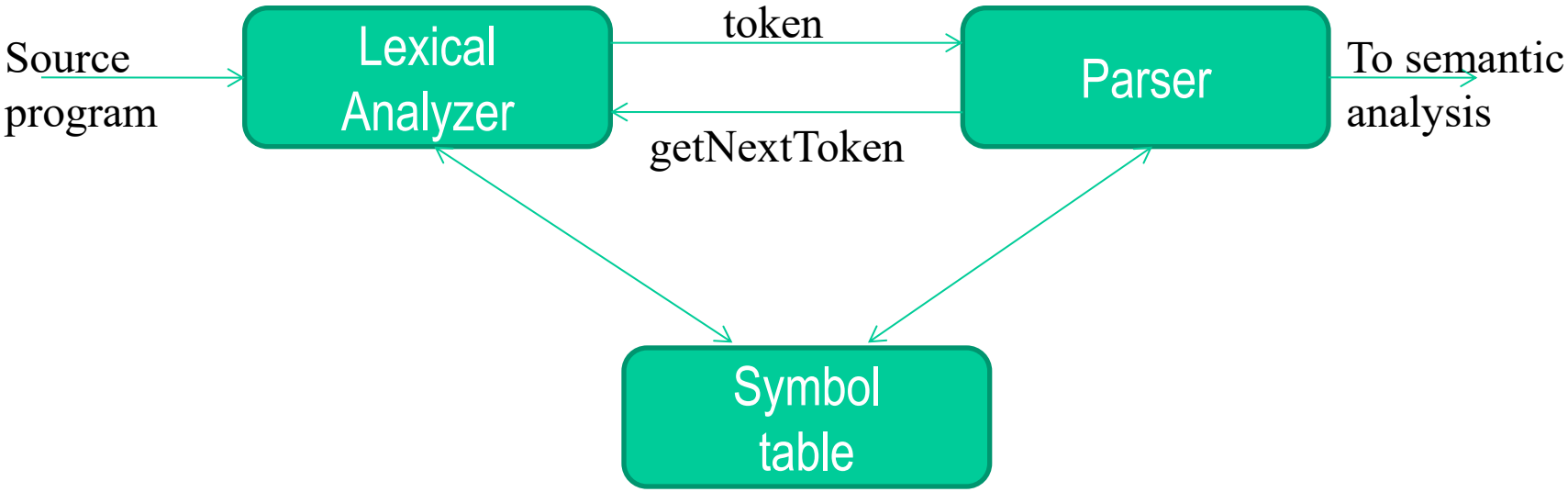
temp1 := id3 * 60.0
id1 := id2 + temp1

Code Generator

MOVF id3, R2
MULF #60.0, R2
MOVF id2, R1
ADDF R2, R1
MOVF R1, id1

Target Program

The role of lexical analyzer



- Its main task is to read the input characters and produce as output a sequence of tokens that the parser uses for syntax analysis.
- Receiving a "get next token" command from the parser, the lexical analyzer reads input characters until it can identify the next token.
- Lexical Analyzer also interacts with the symbol table. When the lexical analyzer discovers a lexeme constituting an identifier, it needs to enter that lexeme into the symbol table.

Its additional tasks are,

1. from the source program stripping out comments and white spaces in the form of blank, tab, new line characters.
2. Correlating error messages generated by the compiler with the source program. For instance, the lexical analyzer may keep track of the number of newline characters seen, so it can associate a line number with each error message.
3. If the source program uses a macro-preprocessor, the expansion of macros may also be performed by the lexical analyzer.

FUNCTIONS :

- ✓ It produces the stream of tokens.
- ✓ It eliminates blank and comments.
- ✓ It generates symbol table which stores the information about identifiers, constants encountered in the input.
- ✓ It keeps track of line number.
- ✓ It reports the error encountered while scanning for the tokens.

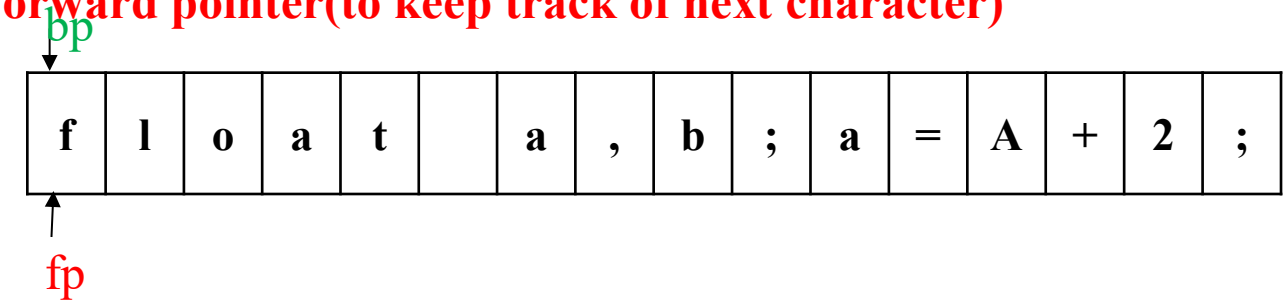
Why to separate Lexical analysis and parsing

1. Simplicity of design
2. Improving compiler efficiency
3. Enhancing compiler portability

INPUT BUFFERING

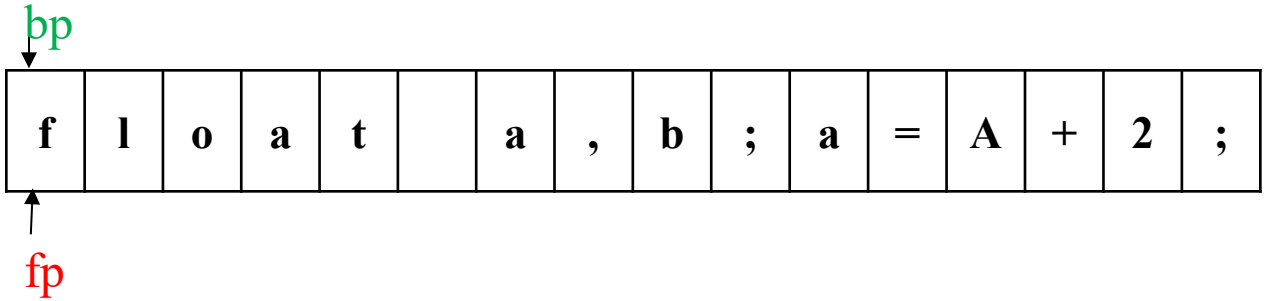
During the analysis, the scanner scans the input string from left to right one character at a time to identify tokens. It uses two pointers for doing this analysis

1. **Begin pointer (to keep track of first character for each token).**
2. **Forward pointer(to keep track of next character)**



Step 1 :

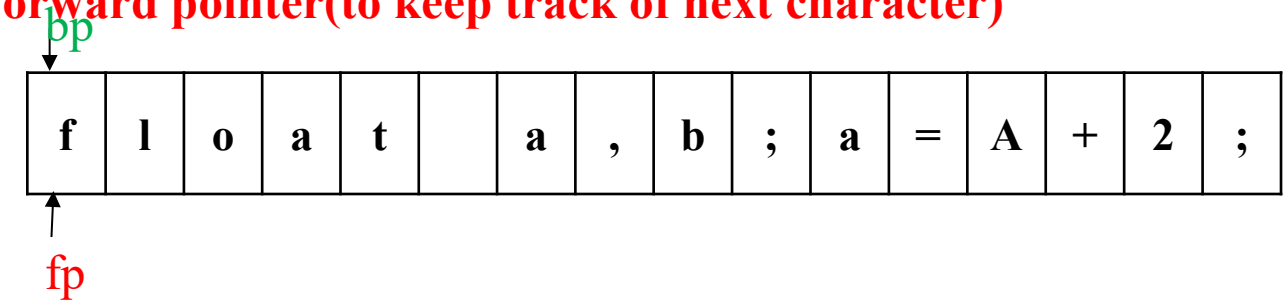
Initially, both begin pointer and forward pointer points to the first character of the lexeme.



INPUT BUFFERING

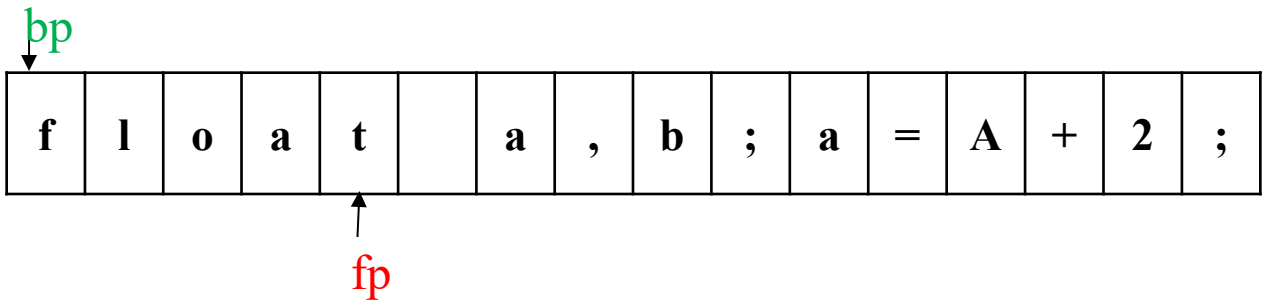
During the analysis, the scanner scans the input string from left to right one character at a time to identify tokens. It uses two pointers for doing this analysis

1. **Begin pointer (to keep track of first character for each token).**
2. **Forward pointer(to keep track of next character)**



Step 2:

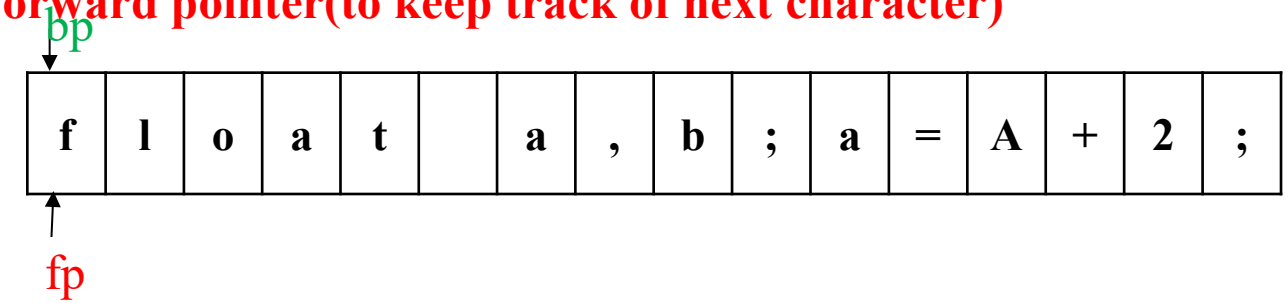
The fp scans the buffer until there is a match with the described token is found.



INPUT BUFFERING

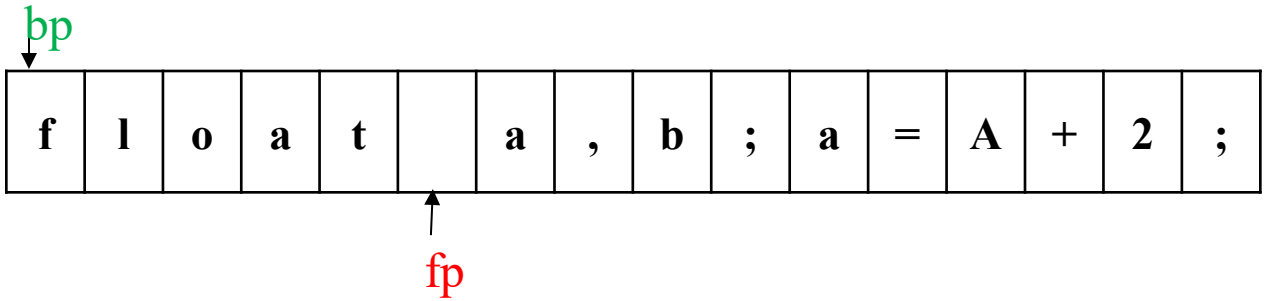
During the analysis, the scanner scans the input string from left to right one character at a time to identify tokens. It uses two pointers for doing this analysis

1. **Begin pointer (to keep track of first character for each token).**
2. **Forward pointer(to keep track of next character)**



Step : 3

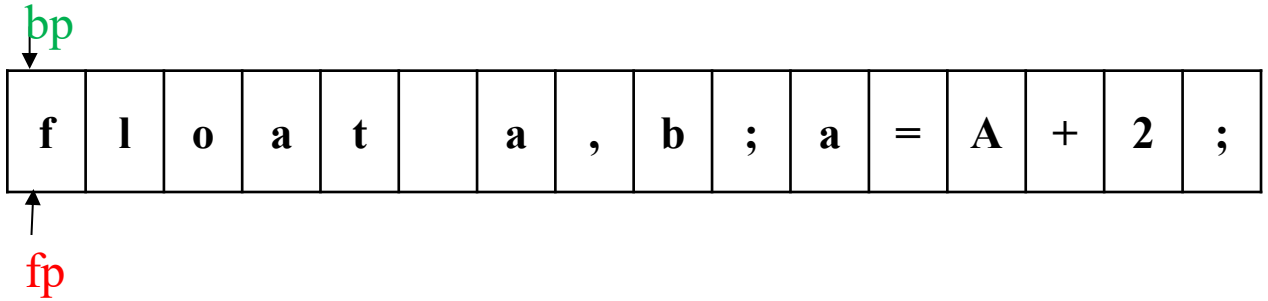
Once the lexeme is found (either a space or a delimiter), the fp will represent the right end to the lexeme.



INPUT BUFFERING

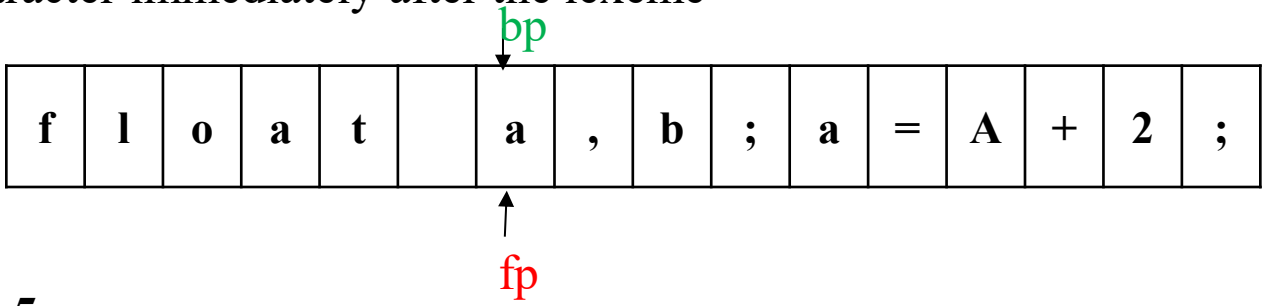
During the analysis, the scanner scans the input string from left to right one character at a time to identify tokens. It uses two pointers for doing this analysis

1. **Begin pointer (to keep track of first character for each token).**
2. **Forward pointer(to keep track of next character)**



Step 4:

After processing the lexeme, both pointers will be set to point the character immediately after the lexeme



Step 5:

This procedure is repeated for the entire source program.

INPUT BUFFERING

Two Types:

1. One buffer scheme
2. Two buffer scheme

One Buffer Scheme:

1.Only one buffer of size 'N' is used.

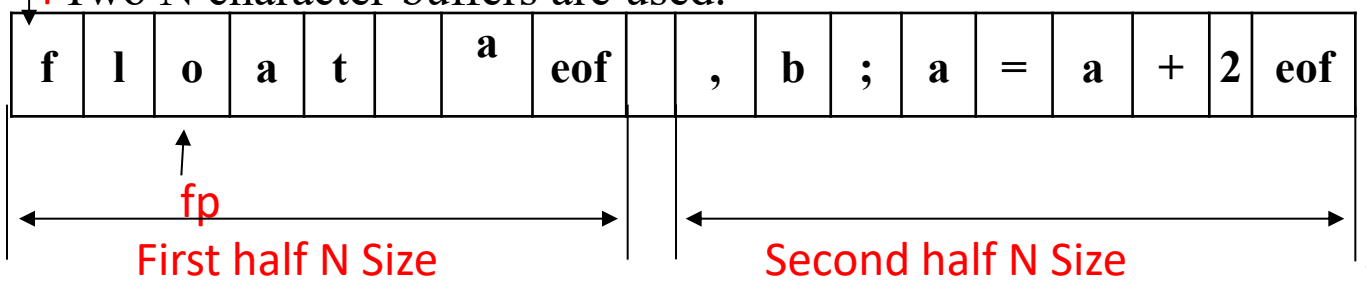
2.First N characters of the input string are read into the buffer. When the fp reaches the end of the buffer, it will be filled with the next set of N characters.

✓ Drawbacks:

The problem with this implementation is that when the size of the token is greater than 'N' this scheme fails to produce the tokens.

Two Buffer Scheme:

Two N character buffers are used.



INPUT BUFFERING

Algorithm for advancing the fp :

```
if fp is at the end of first half then
    begin
            Load second half;
            Increment fp by 1;
    end
else if fp at the end of second half then
     begin
            Load first half;
            Set fp to first character of first
half;
      end
else
            increment fp by 1;
end
```

- Every time whether it has reached its end or not should be checked.
- To reduce the number of comparisons, a special character called sentinel character (usually EOF) is introduced at ends of the buffer halves.

Algorithm for advancing the fp using Sentinel:

```
fp = fp+1;
begin
   if fp = eof then
      if fp at the end of first half then
         begin
            Load second half;
            Increment fp by 1;
          end
      else if fp at the end of second half then
         begin
         Load first half;
         Set fp to first character of first half;
        end
     else
         Terminate lexical analysis;
end
```