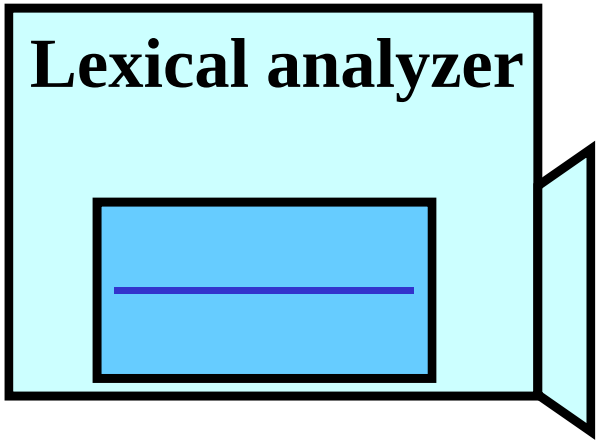
Part V.
Lexical Analysis

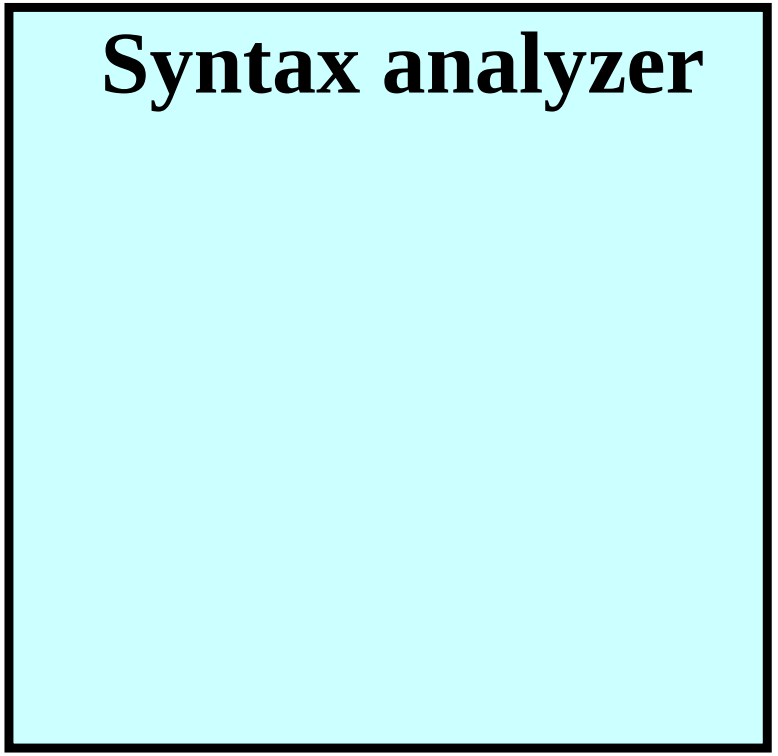
Lexical Analyzer (Scanner)

Example:

Source program:

```
Pos := Rate*60
```



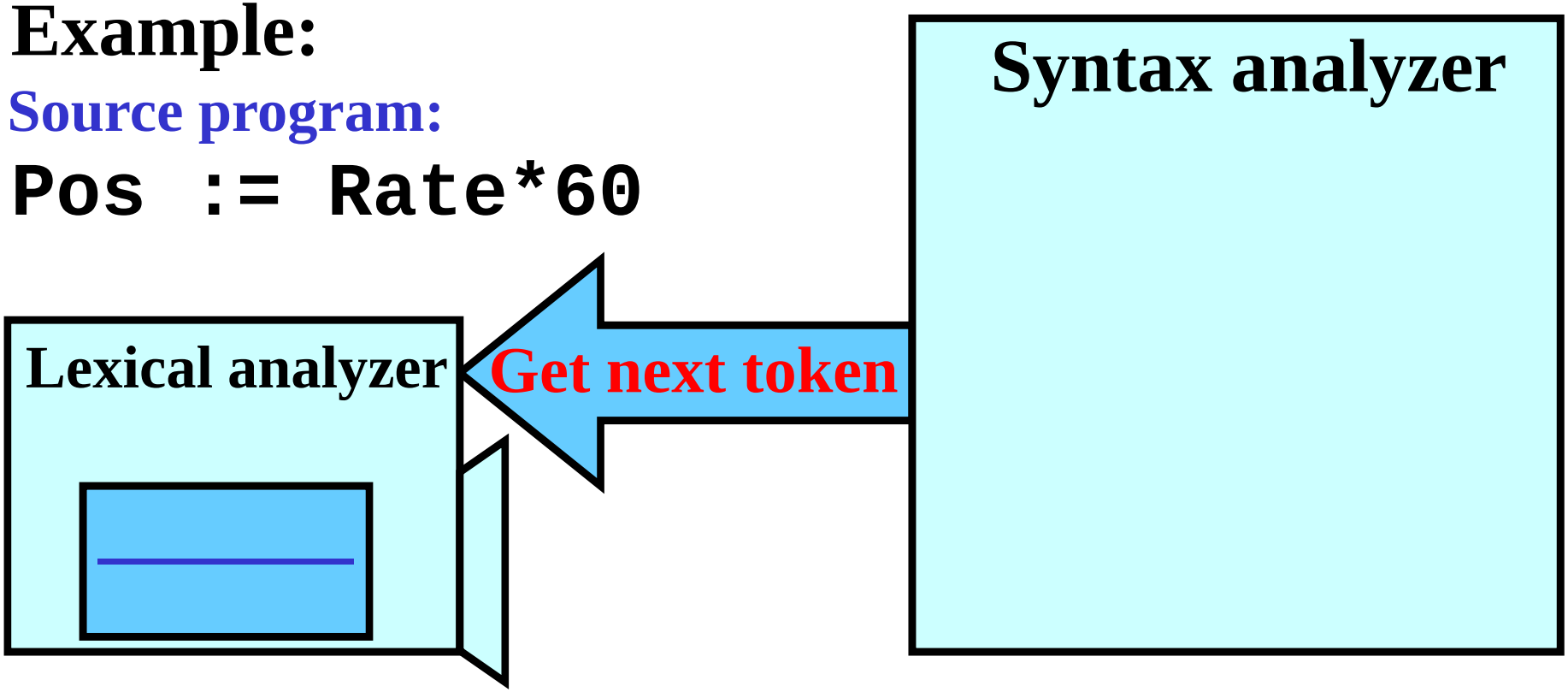


Lexical Analyzer (Scanner)

Example:

Source program:

```
Pos := Rate*60
```

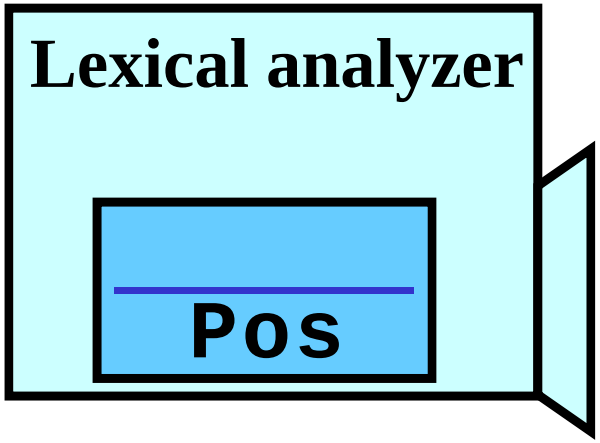


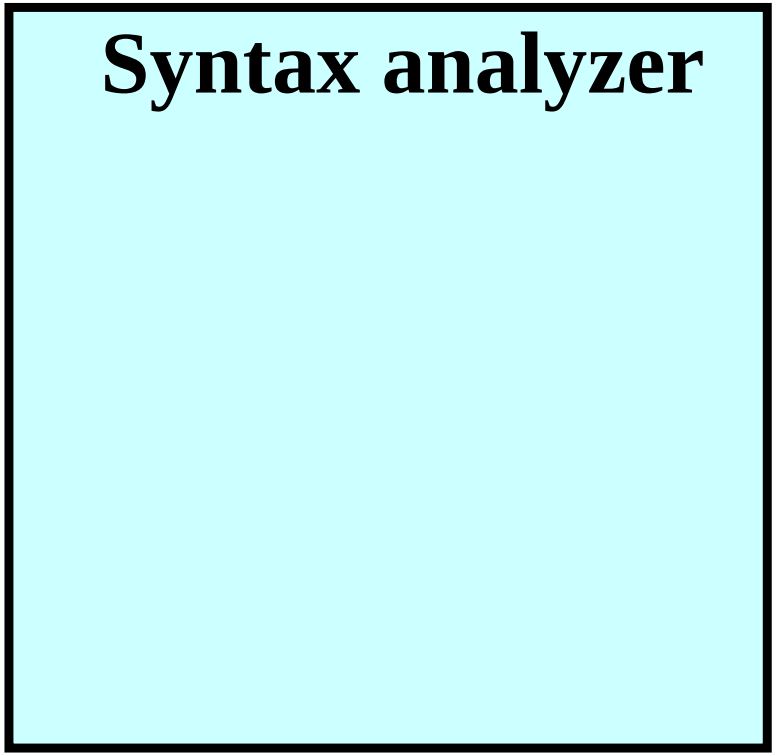
Lexical Analyzer (Scanner)

Example:

Source program:

`Pos := Rate*60`



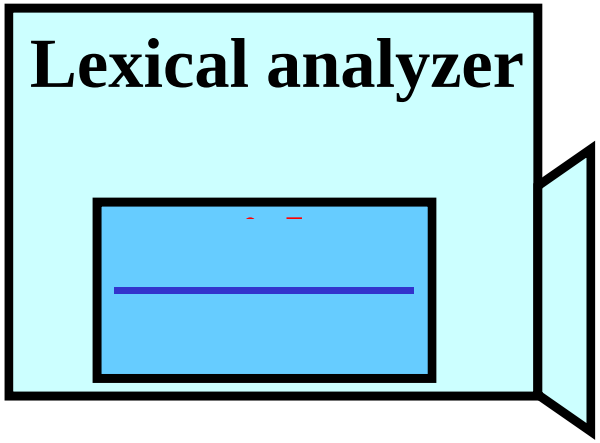


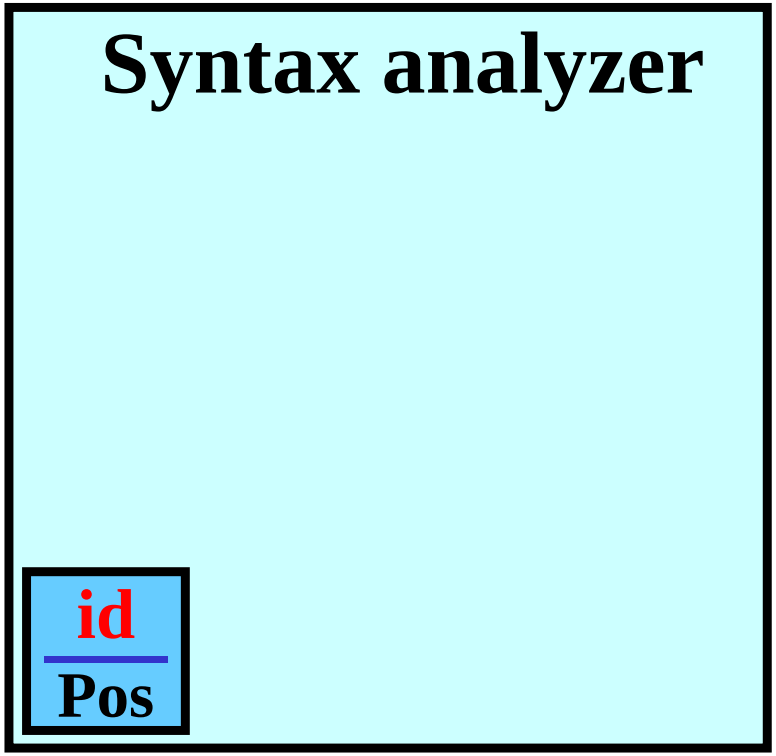
Lexical Analyzer (Scanner)

Example:

Source program:

```
Pos := Rate*60
```



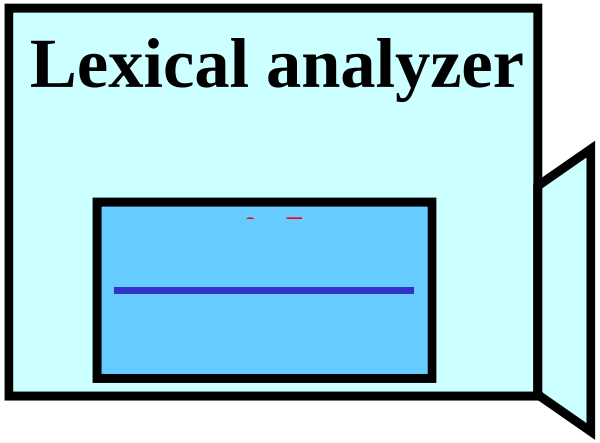


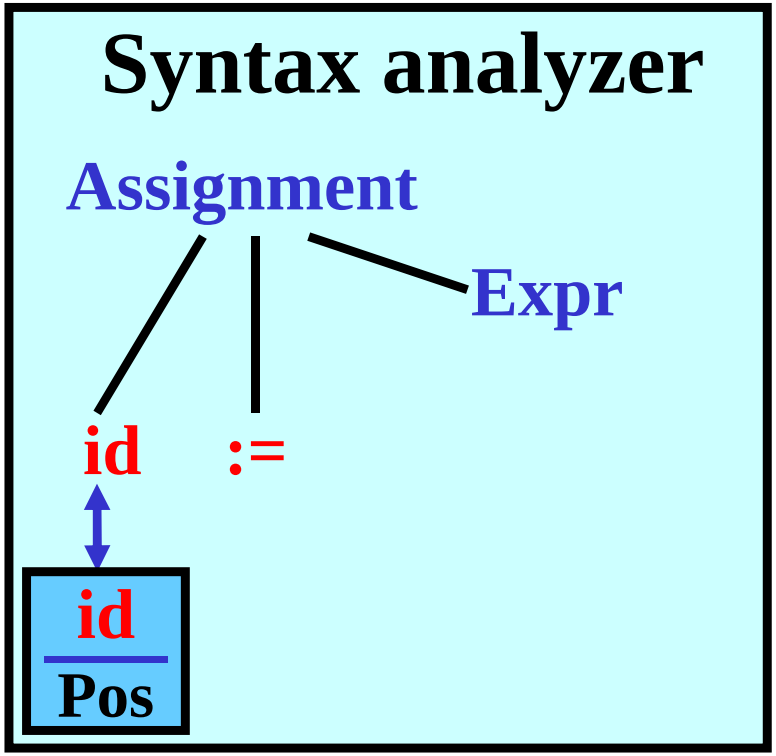
Lexical Analyzer (Scanner)

Example:

Source program:

`Pos := Rate*60`



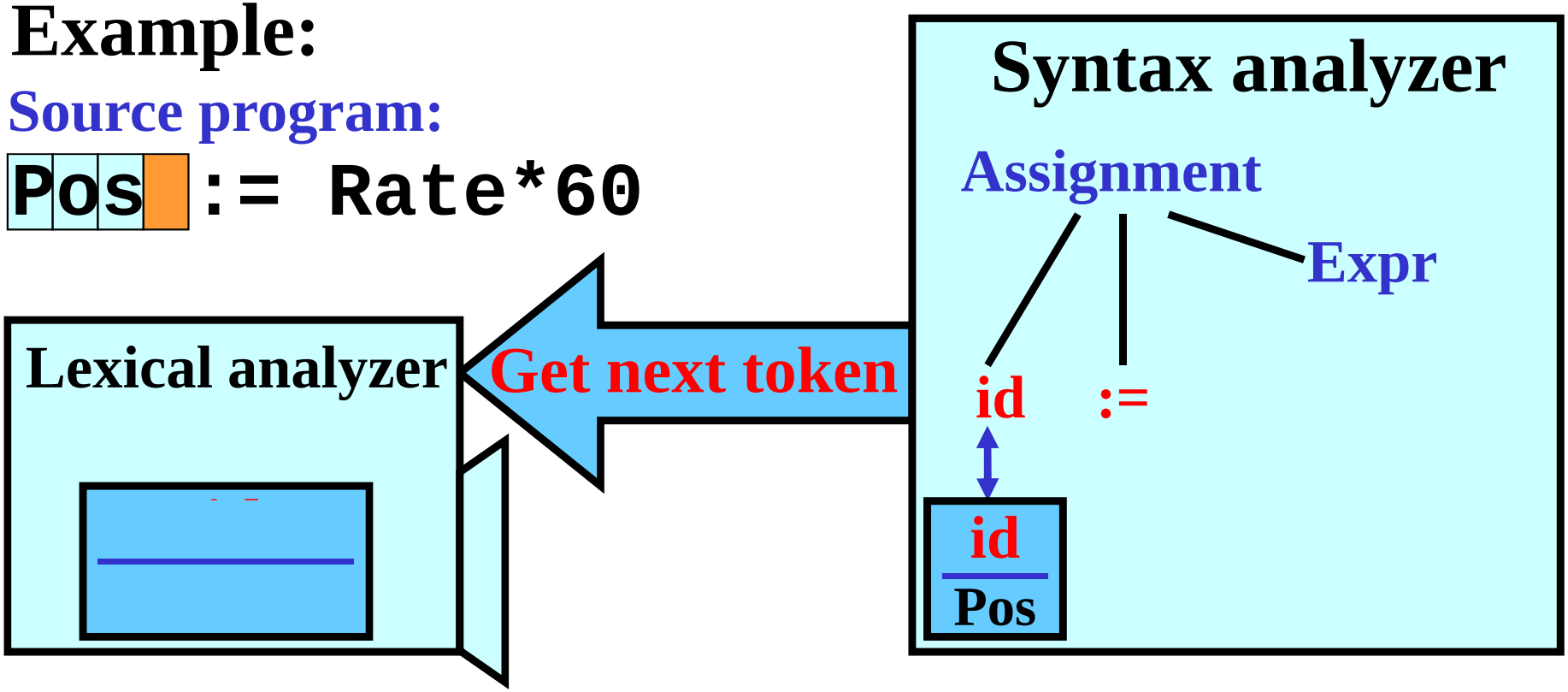


Lexical Analyzer (Scanner)

Example:

Source program:

`Pos := Rate*60`

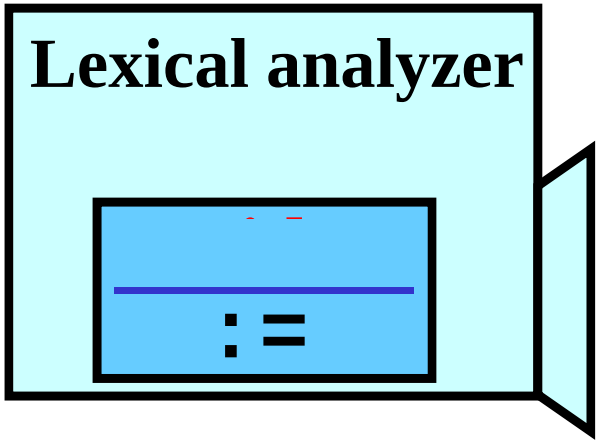


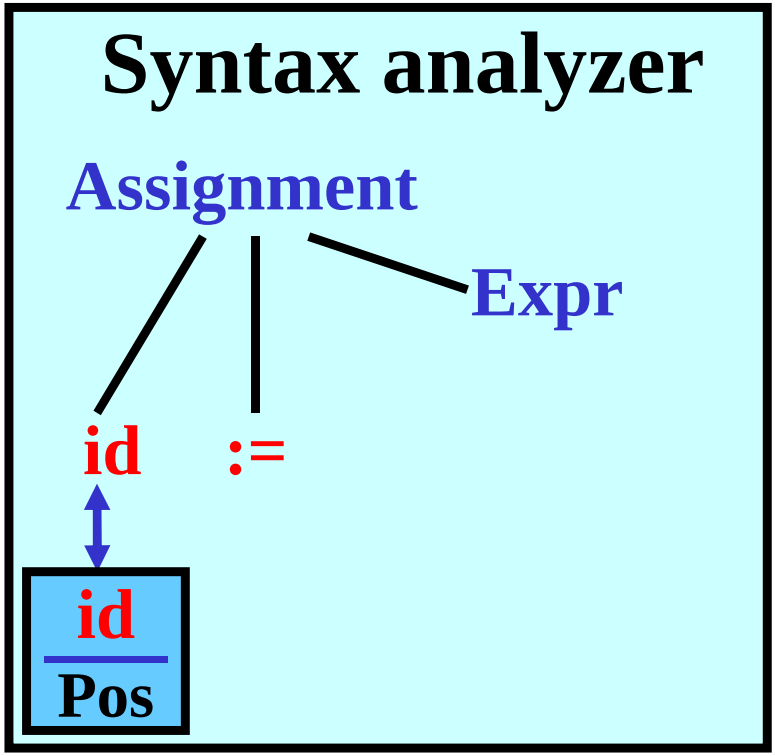
Lexical Analyzer (Scanner)

Example:

Source program:

`Pos := Rate*60`



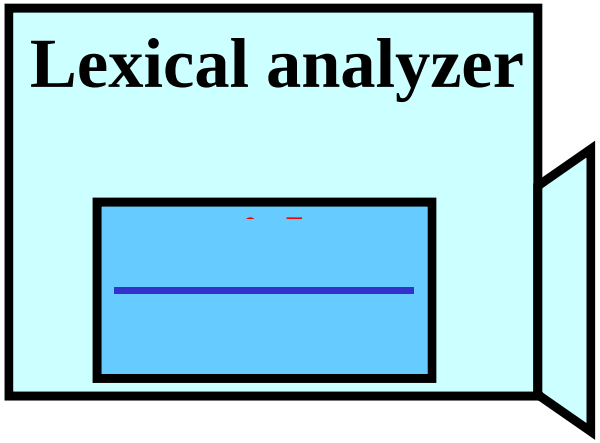


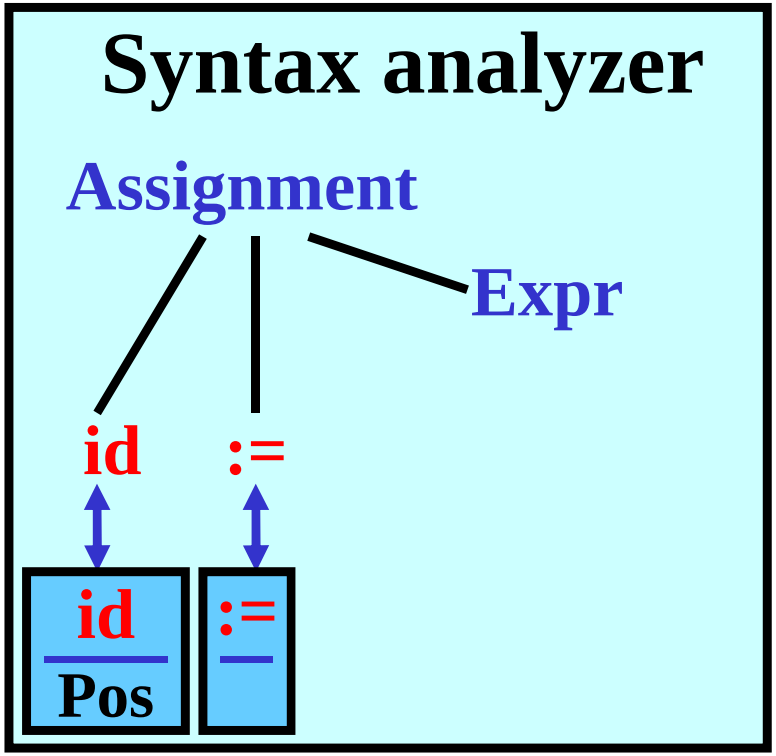
Lexical Analyzer (Scanner)

Example:

Source program:

`Pos := Rate*60`



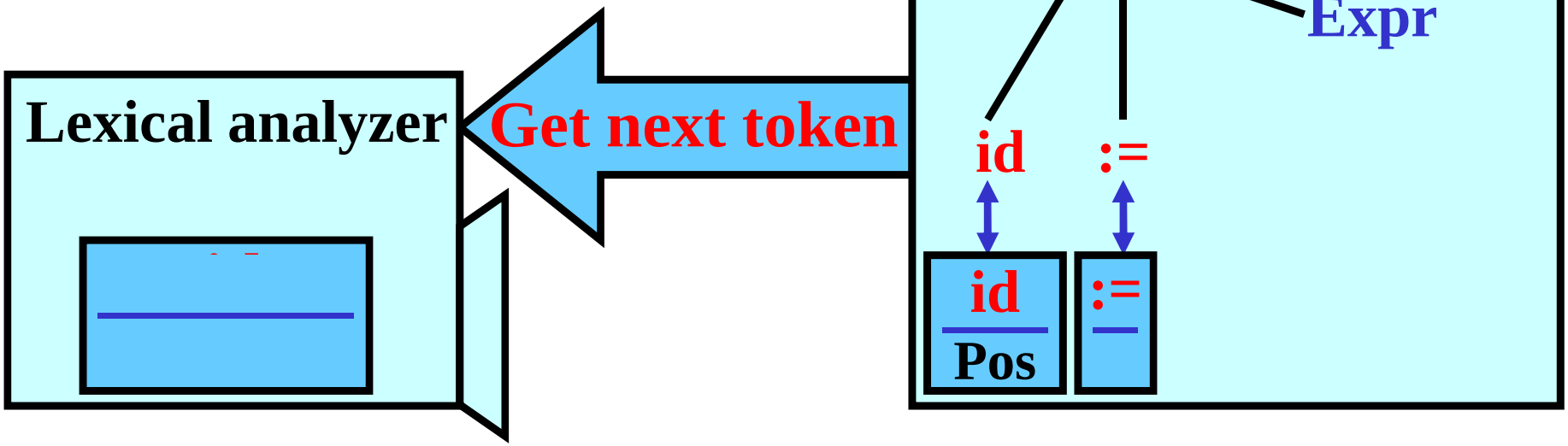


Lexical Analyzer (Scanner)

Example:

Source program:

`Pos := Rate*60`

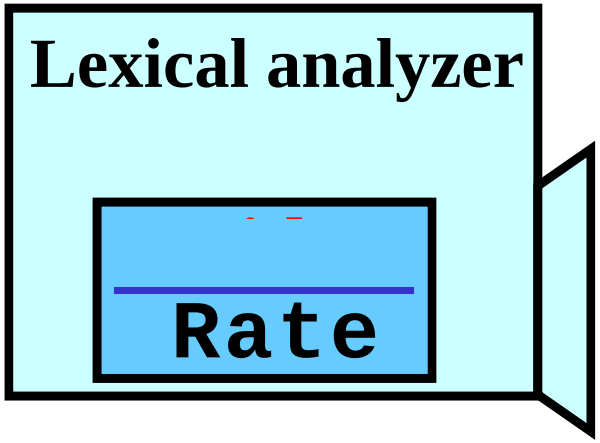


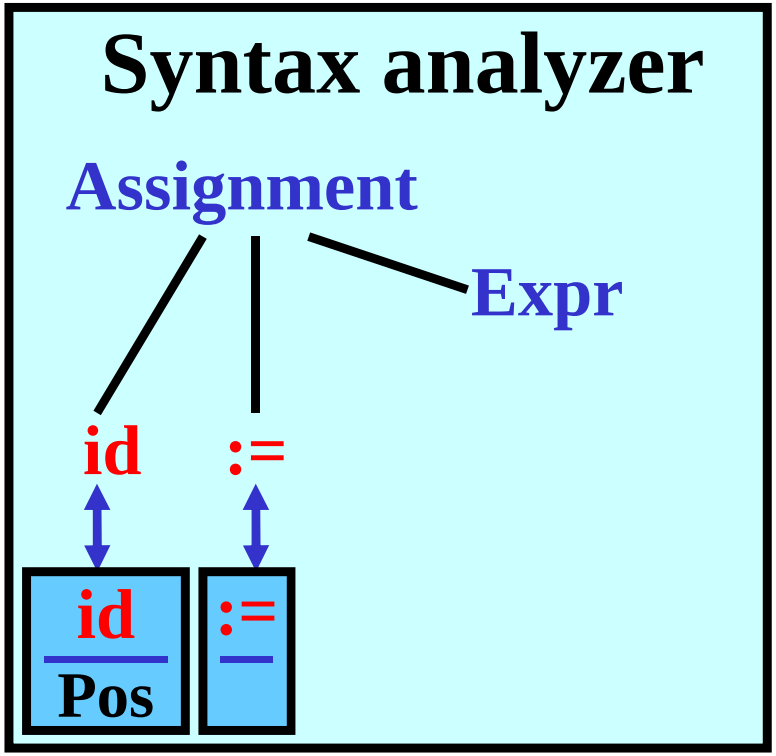
Lexical Analyzer (Scanner)

Example:

Source program:

`Pos := Rate*60`



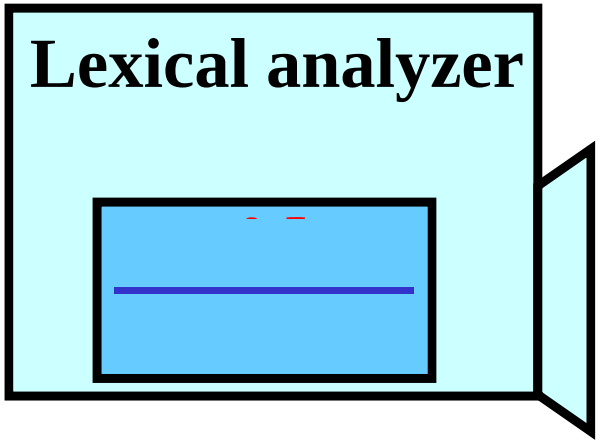


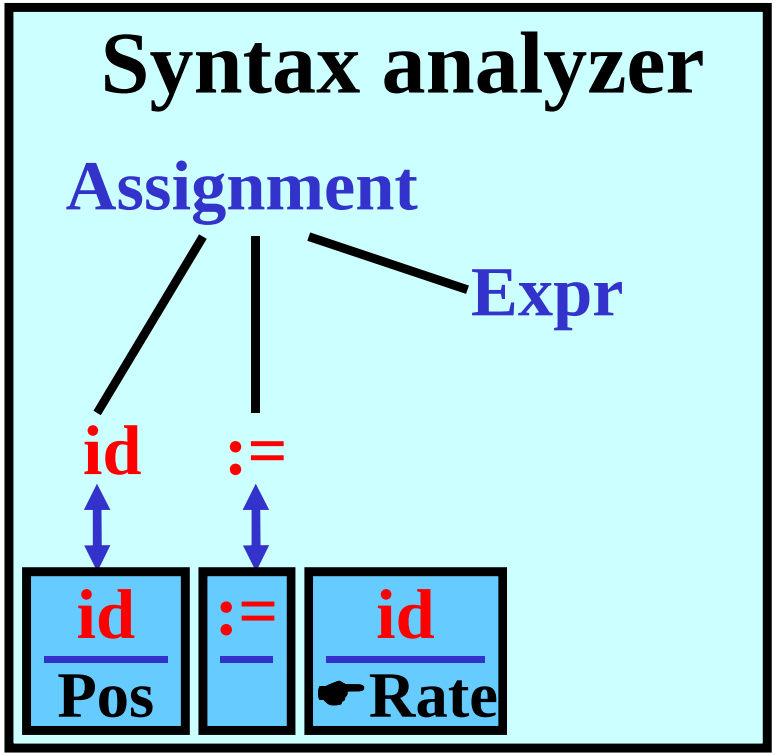
Lexical Analyzer (Scanner)

Example:

Source program:

`Pos := Rate*60`



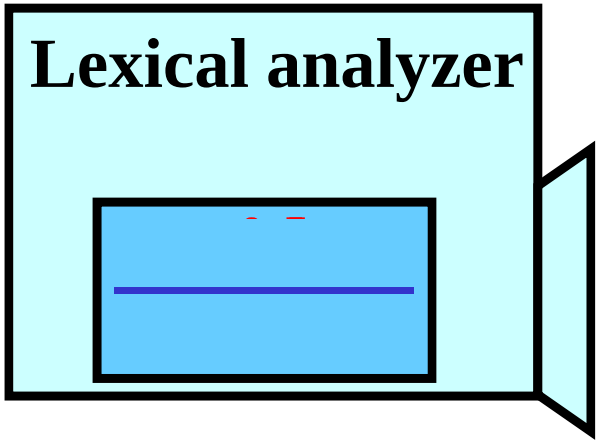


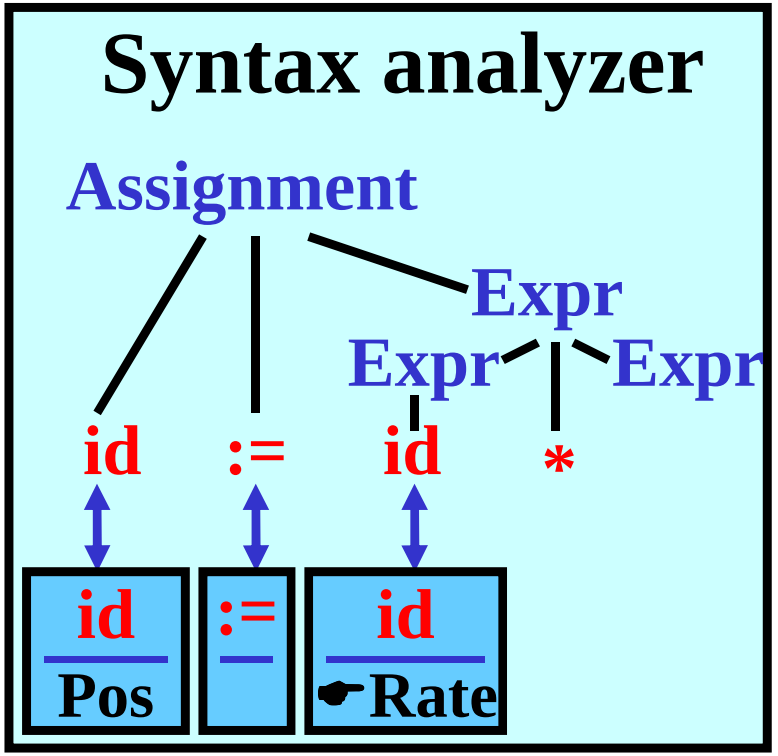
Lexical Analyzer (Scanner)

Example:

Source program:

```
Pos := Rate*60
```



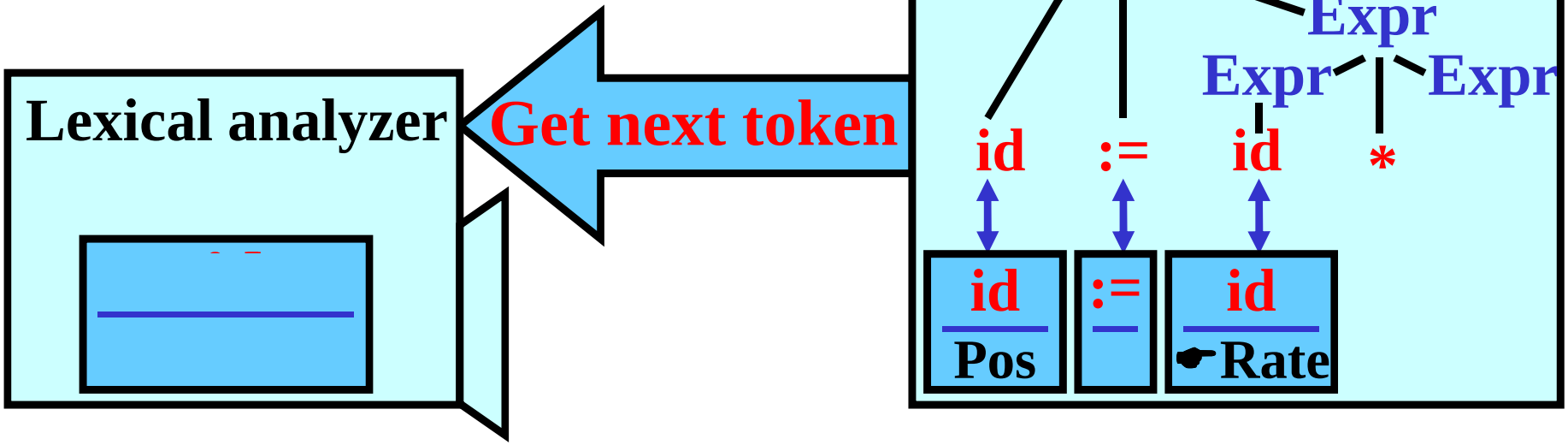


Lexical Analyzer (Scanner)

Example:

Source program:

```
Pos := Rate*60
```

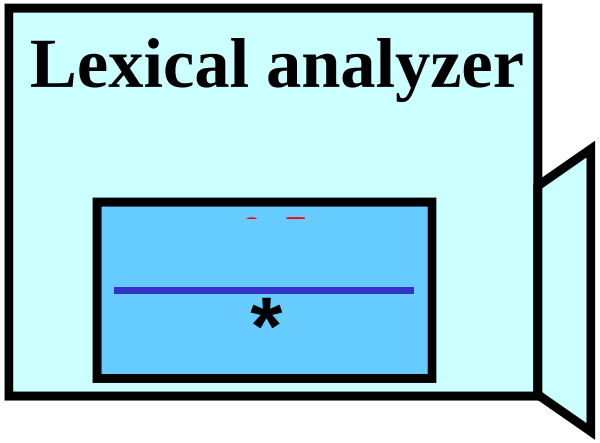


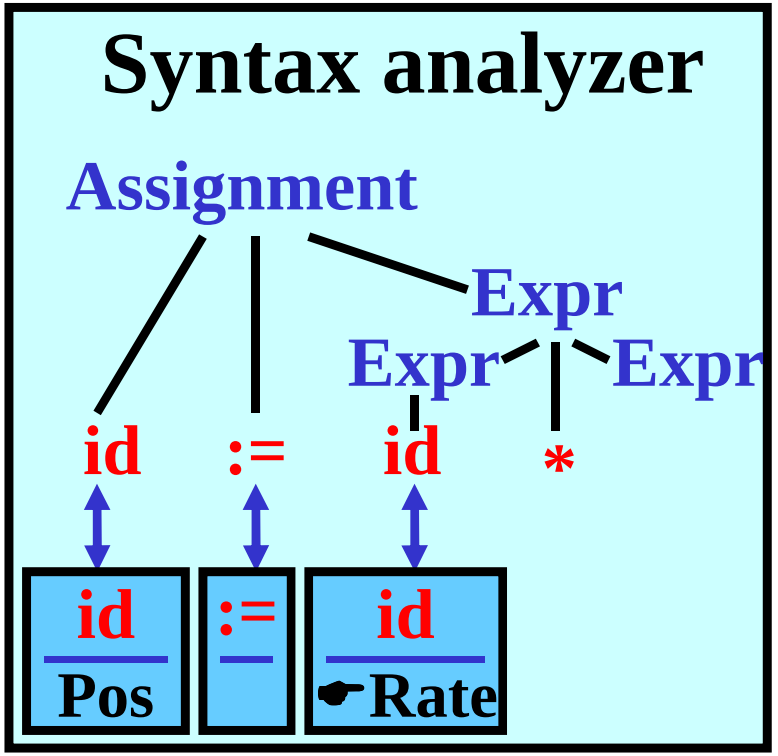
Lexical Analyzer (Scanner)

Example:

Source program:

```
Pos := Rate*60
```



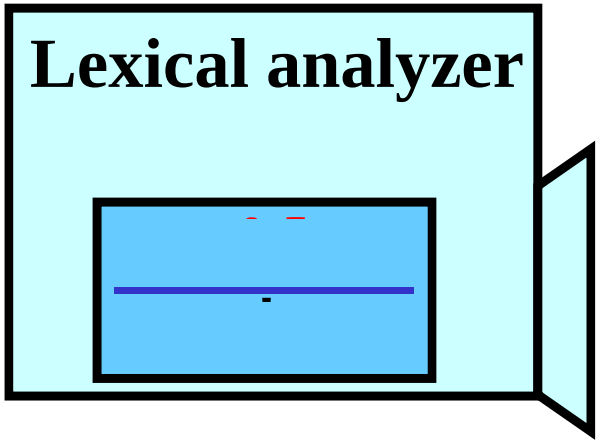


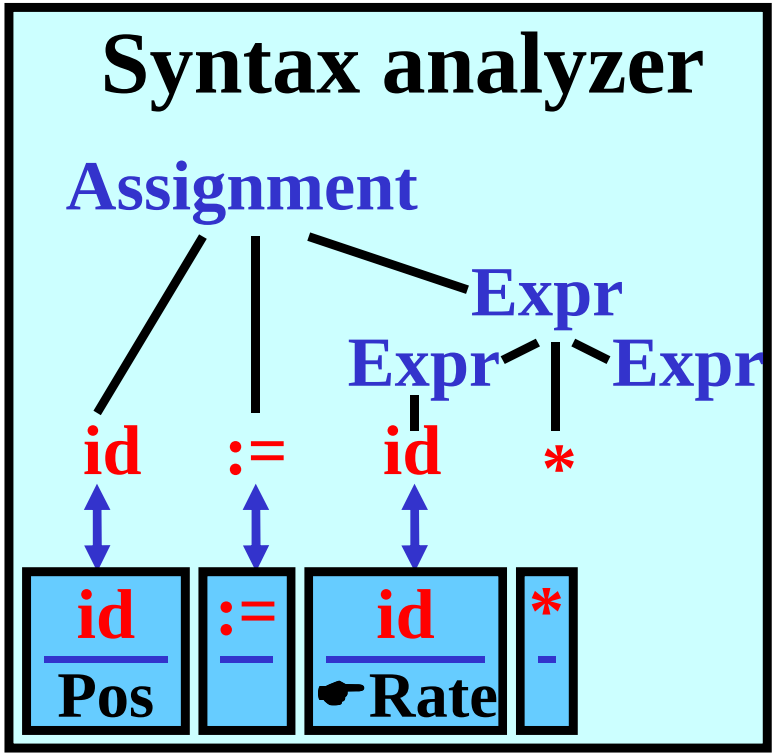
Lexical Analyzer (Scanner)

Example:

Source program:

```
Pos := Rate*60
```



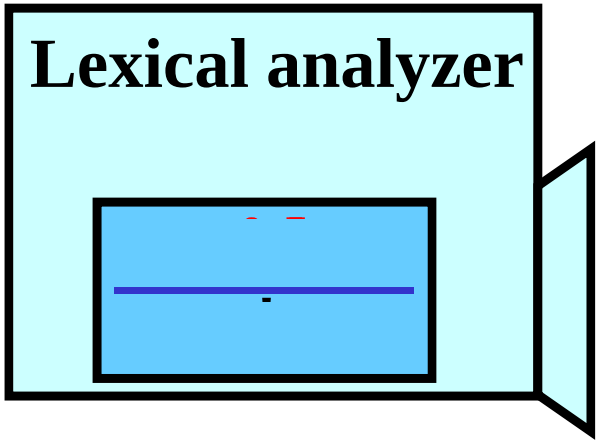


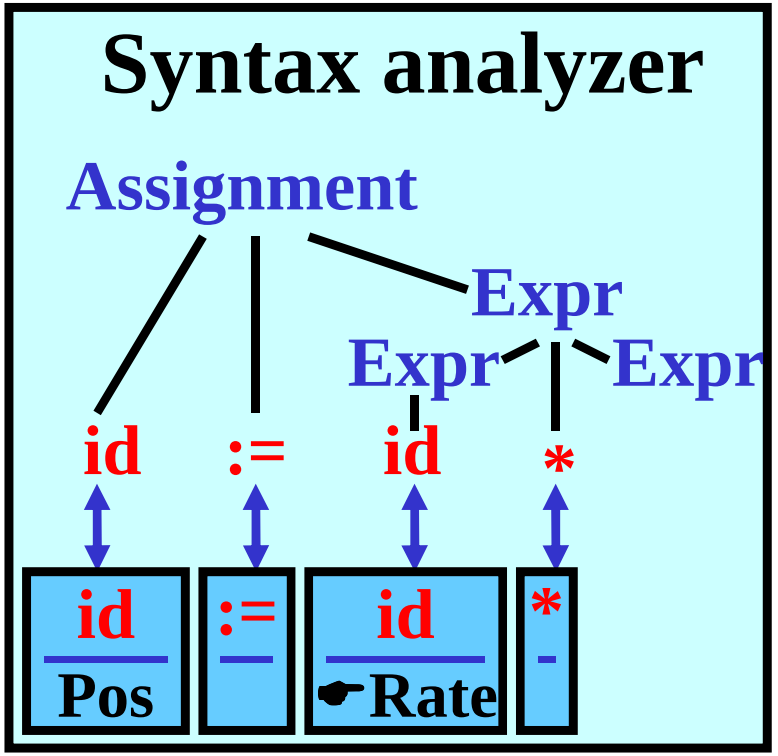
Lexical Analyzer (Scanner)

Example:

Source program:

```
Pos := Rate*60
```



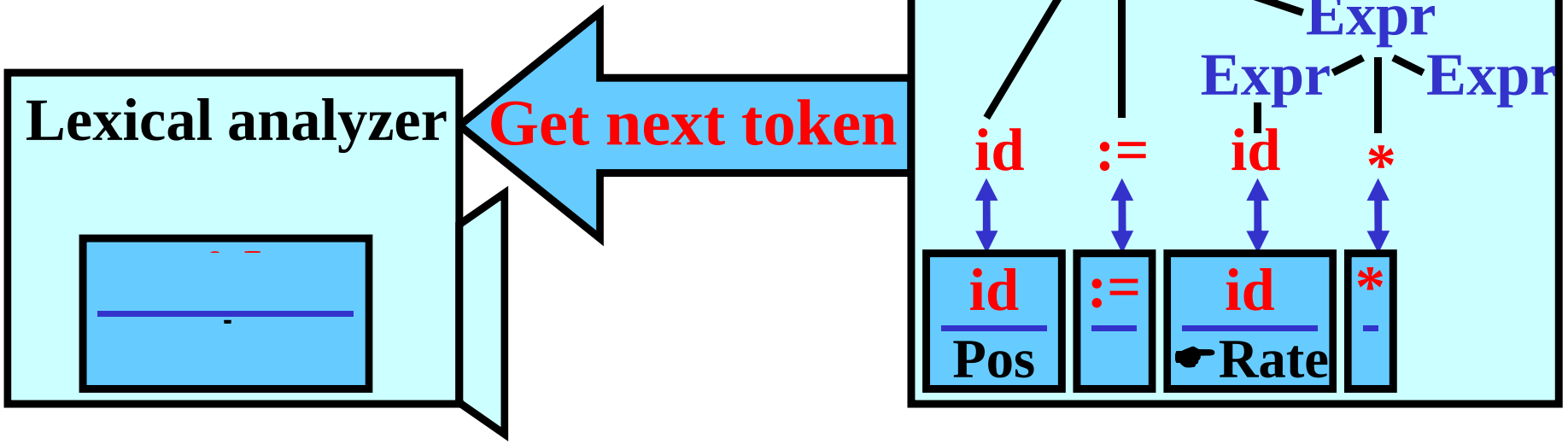


Lexical Analyzer (Scanner)

Example:

Source program:

```
Pos := Rate*60
```

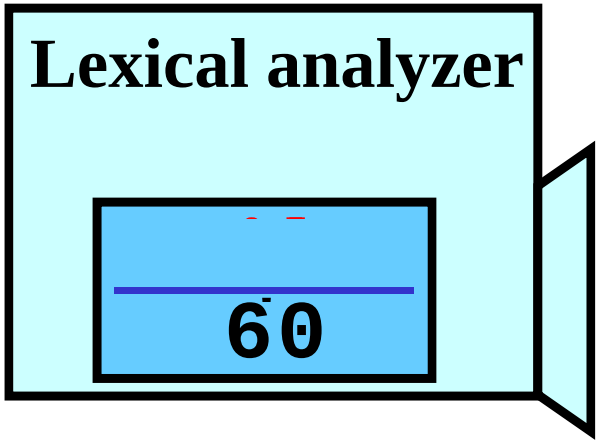


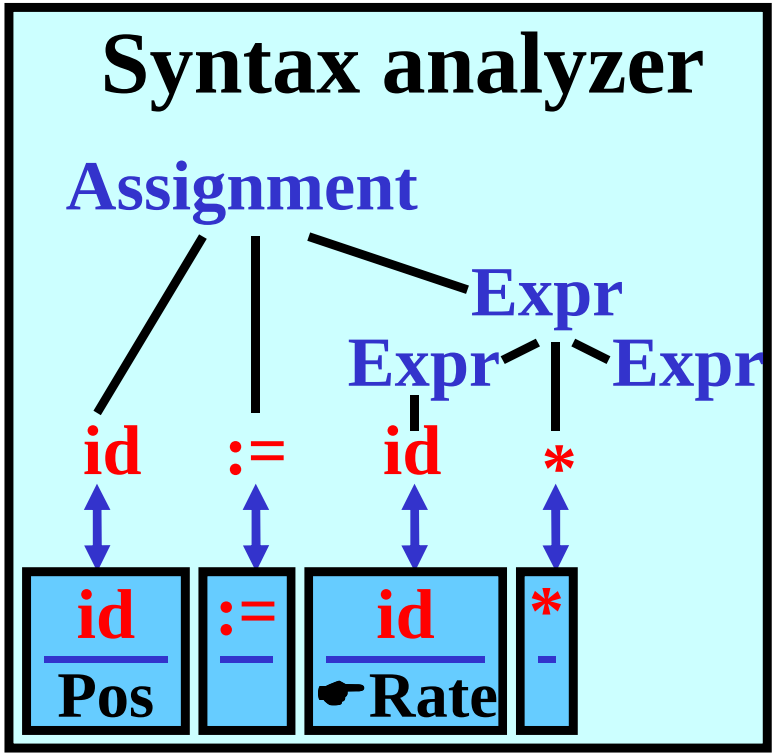
Lexical Analyzer (Scanner)

Example:

Source program:

```
Pos := Rate*60
```



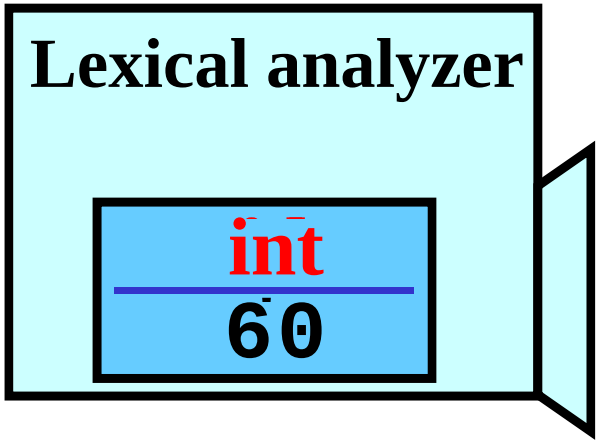


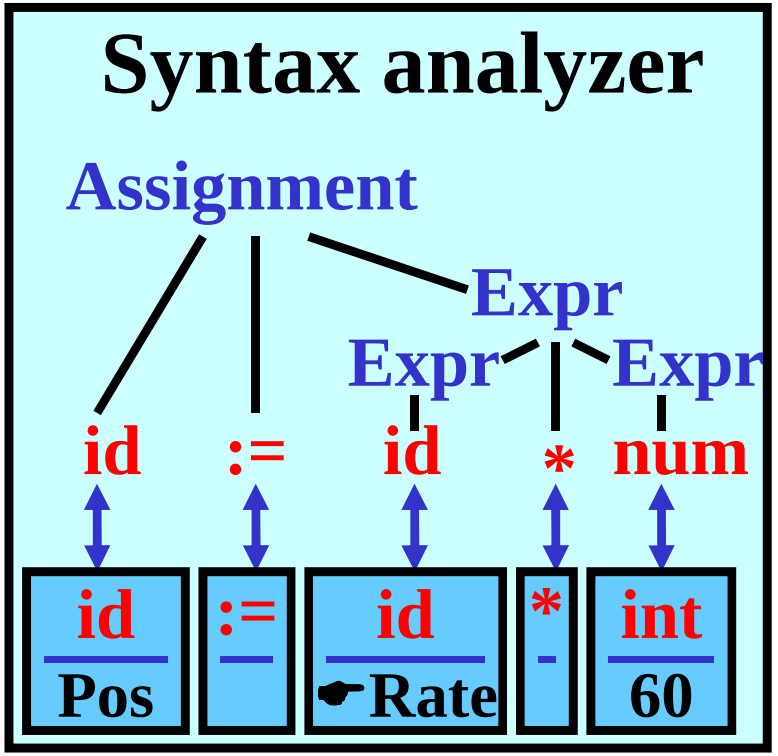
Lexical Analyzer (Scanner)

Example:

Source program:

```
Pos := Rate*60
```

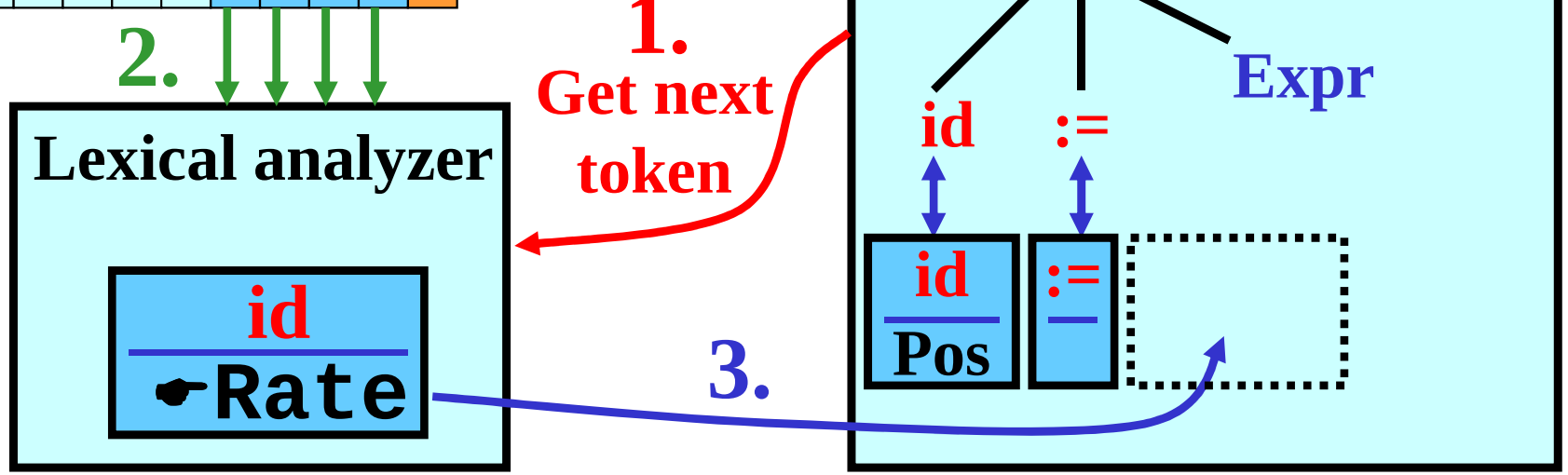




Lexical Analyzer (Scanner)

Example:

Source program:



Scanner: Tasks

Main task

- recognition and classification of lexemes
- representing lexemes by their tokens

Other tasks

- removal of comments and whitespaces
- communication with symbol tables

Relation to Models for RLs

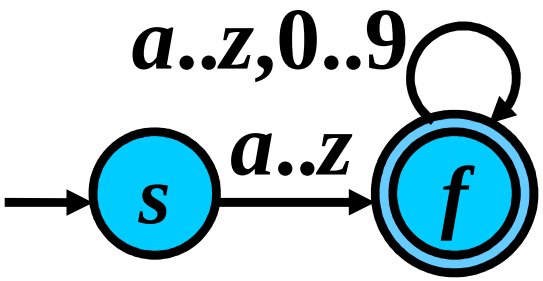
- **Regular expressions** specify lexemes
- **DFAs** underlie scanners

Lexemes Recognized by DFAs 1/2

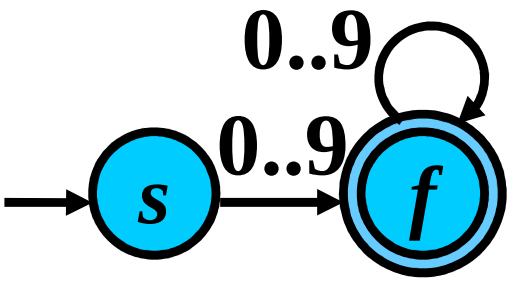
1) Recognition of lexemes by using DFA

Example:

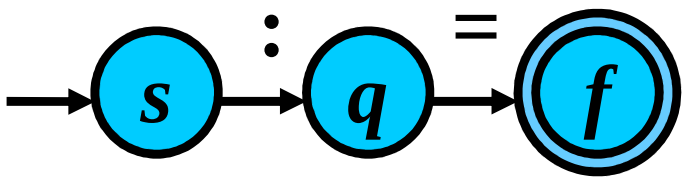
Identifier:



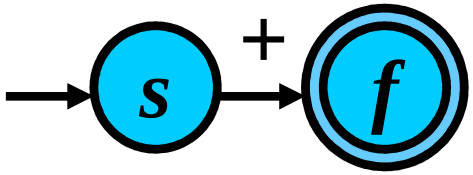
Integer:



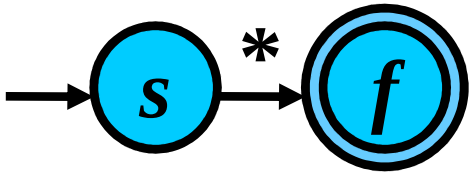
Assignment:



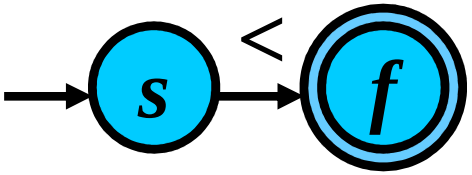
Operator +:



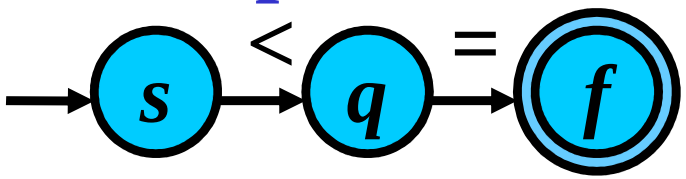
Operator *:



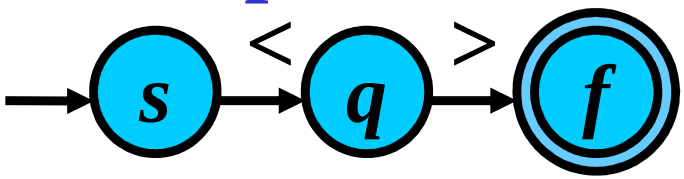
Comparator <:



Comparator <=:



Comparator <>:

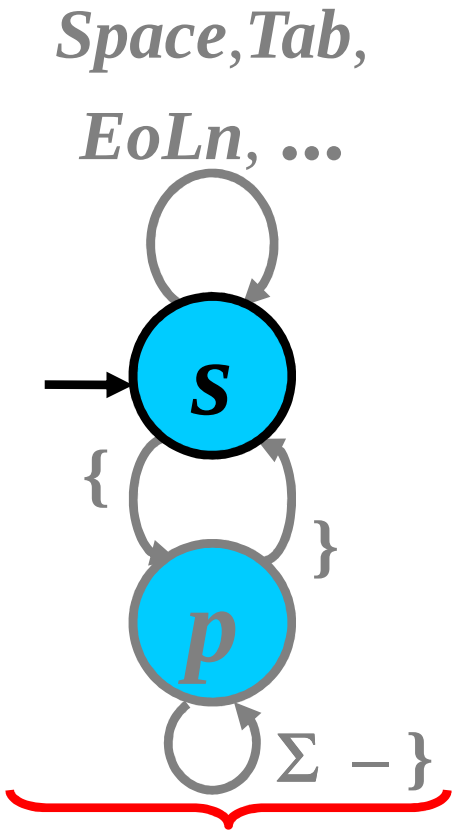


Lexemes Recognized by DFAs 2/2

2) Construction of an FA that accepts all lexemes:

Lexemes Recognized by DFAs 2/2

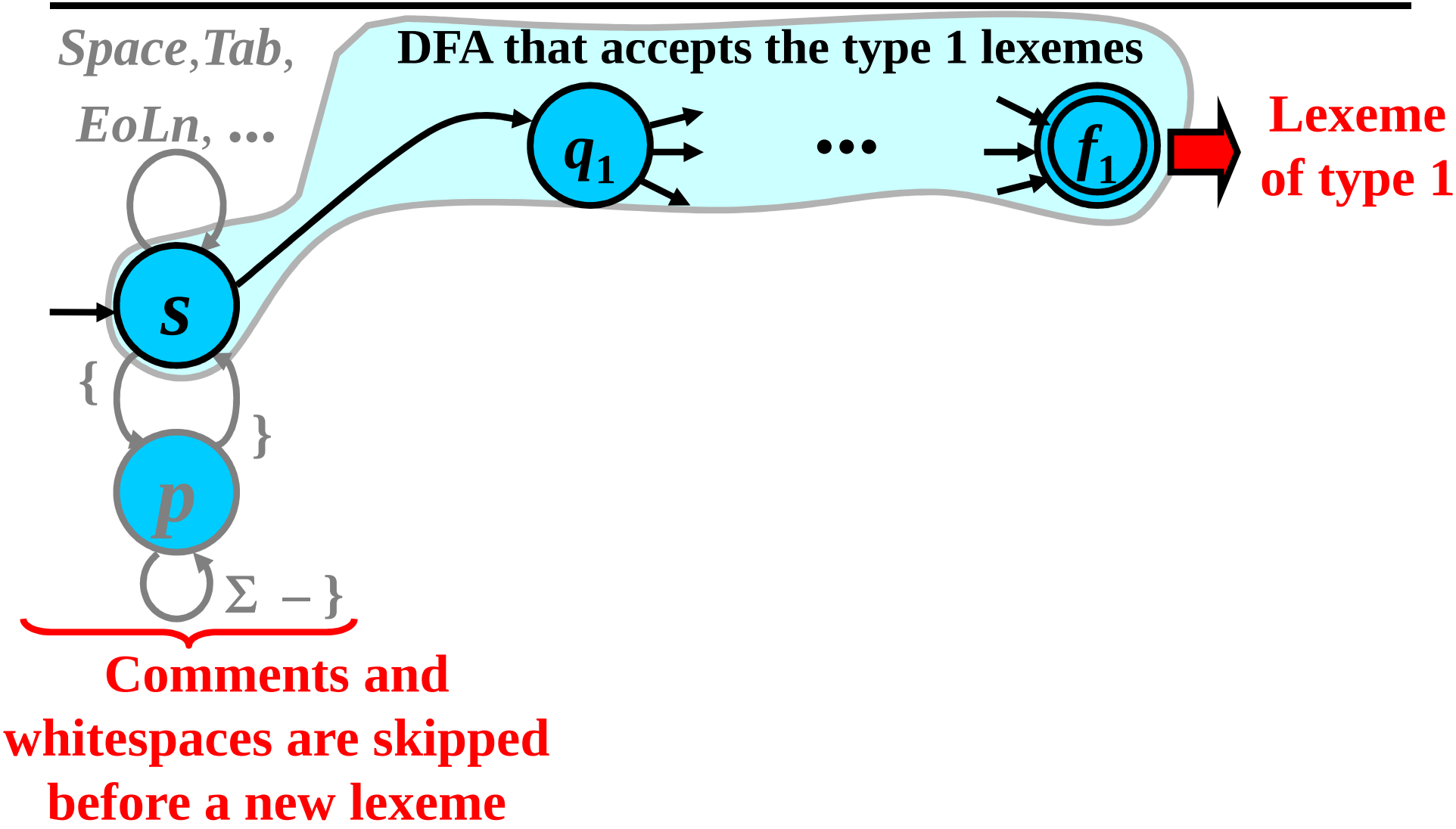
2) Construction of an FA that accepts all lexemes:



Comments and whitespaces are skipped before a new lexeme

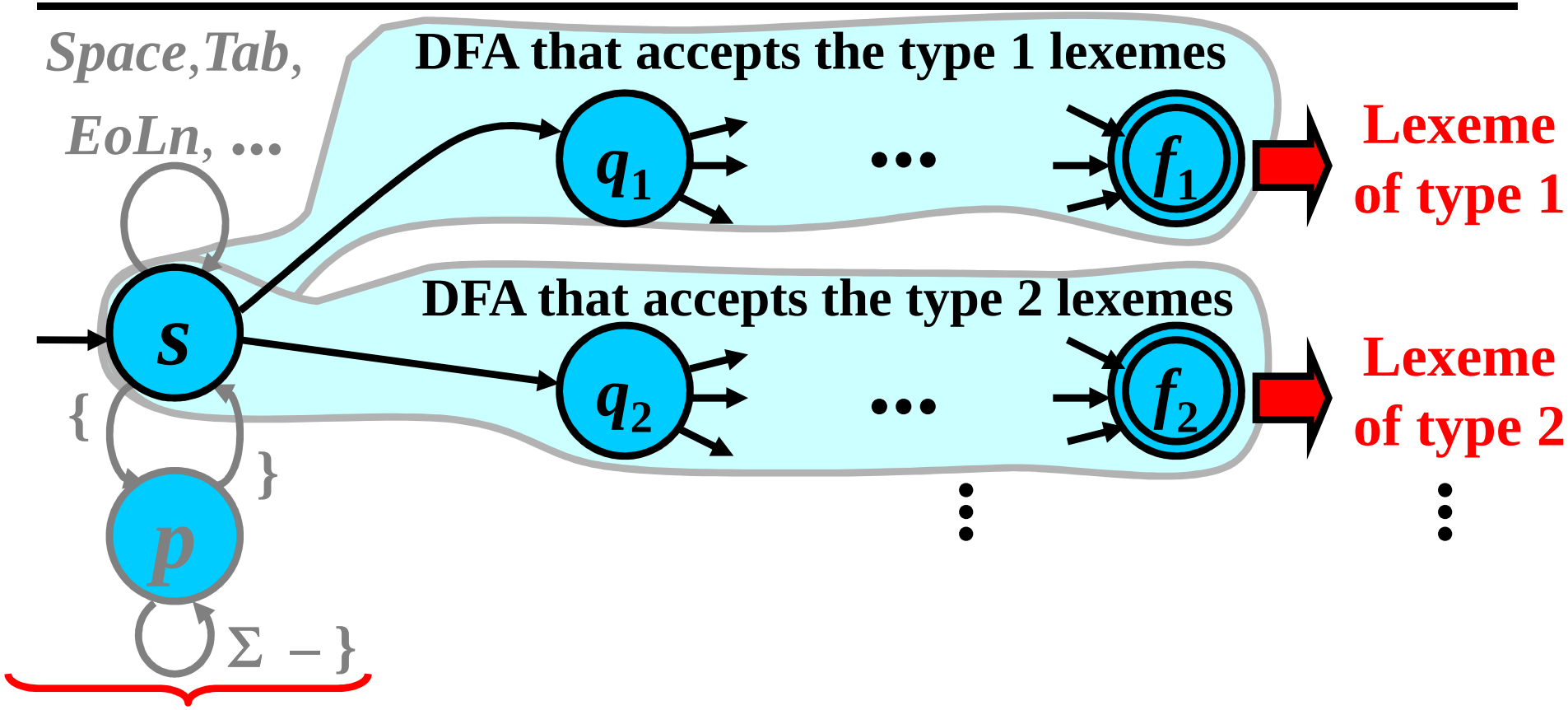
Lexemes Recognized by DFAs 2/2

2) Construction of an FA that accepts all lexemes:



Lexemes Recognized by DFAs 2/2

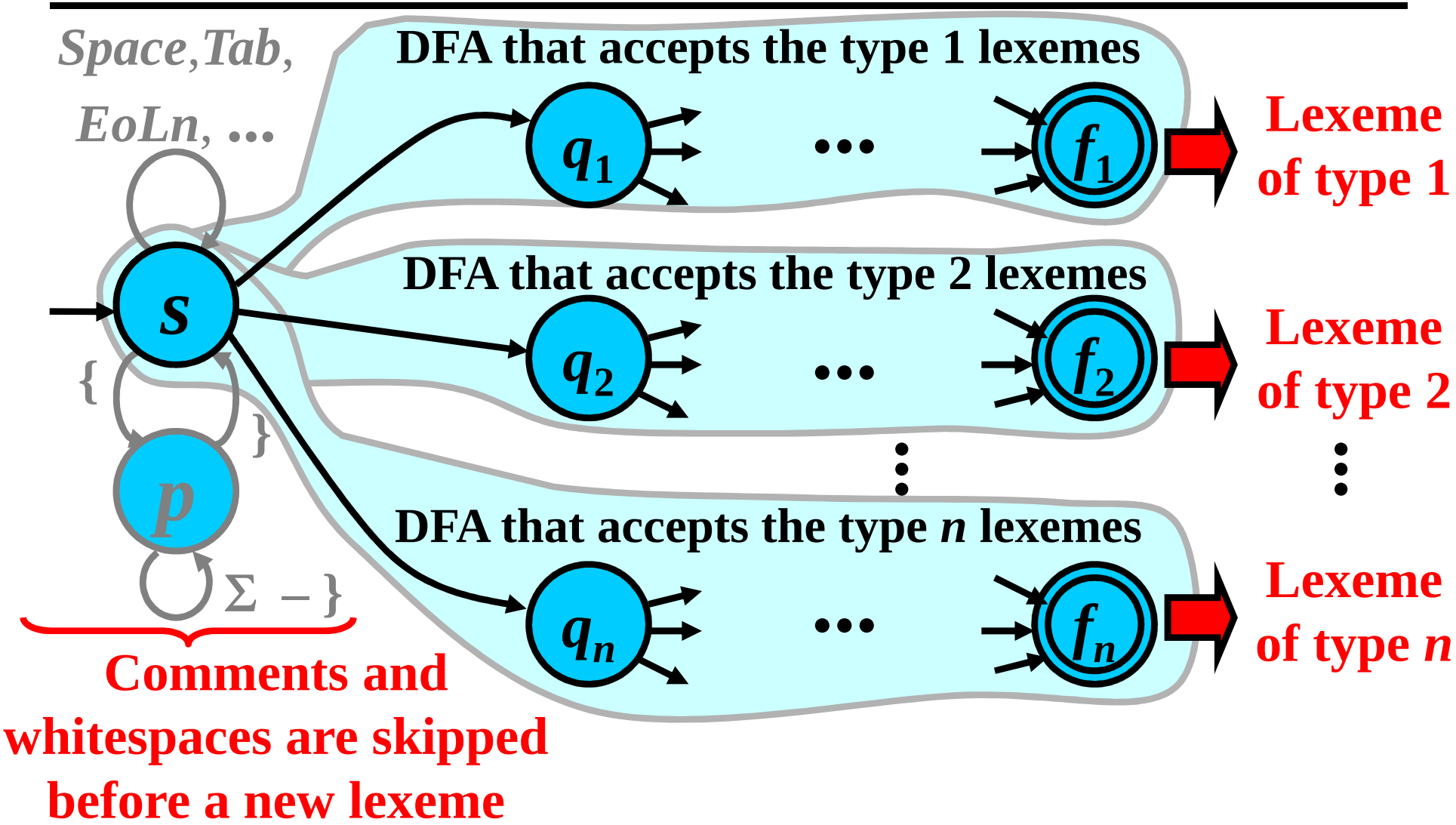
2) Construction of an FA that accepts all lexemes:



Comments and whitespaces are skipped before a new lexeme

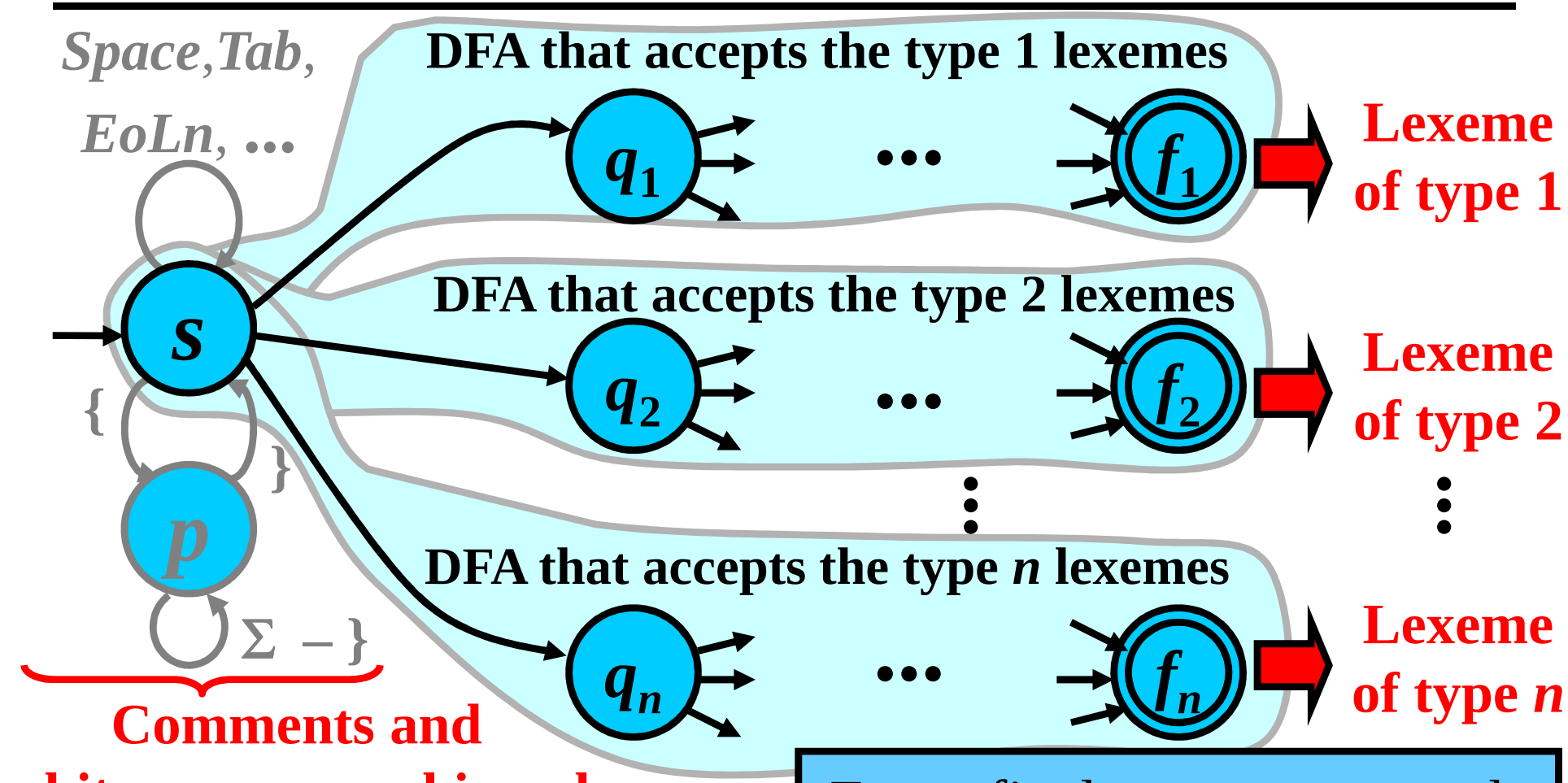
Lexemes Recognized by DFAs 2/2

2) Construction of an FA that accepts all lexemes:



Lexemes Recognized by DFAs 2/2

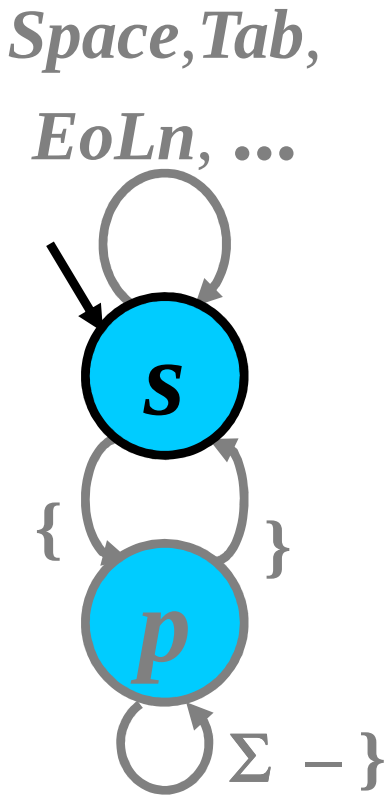
2) Construction of an FA that accepts all lexemes:



Every final state corresponds to **one** type of lexeme

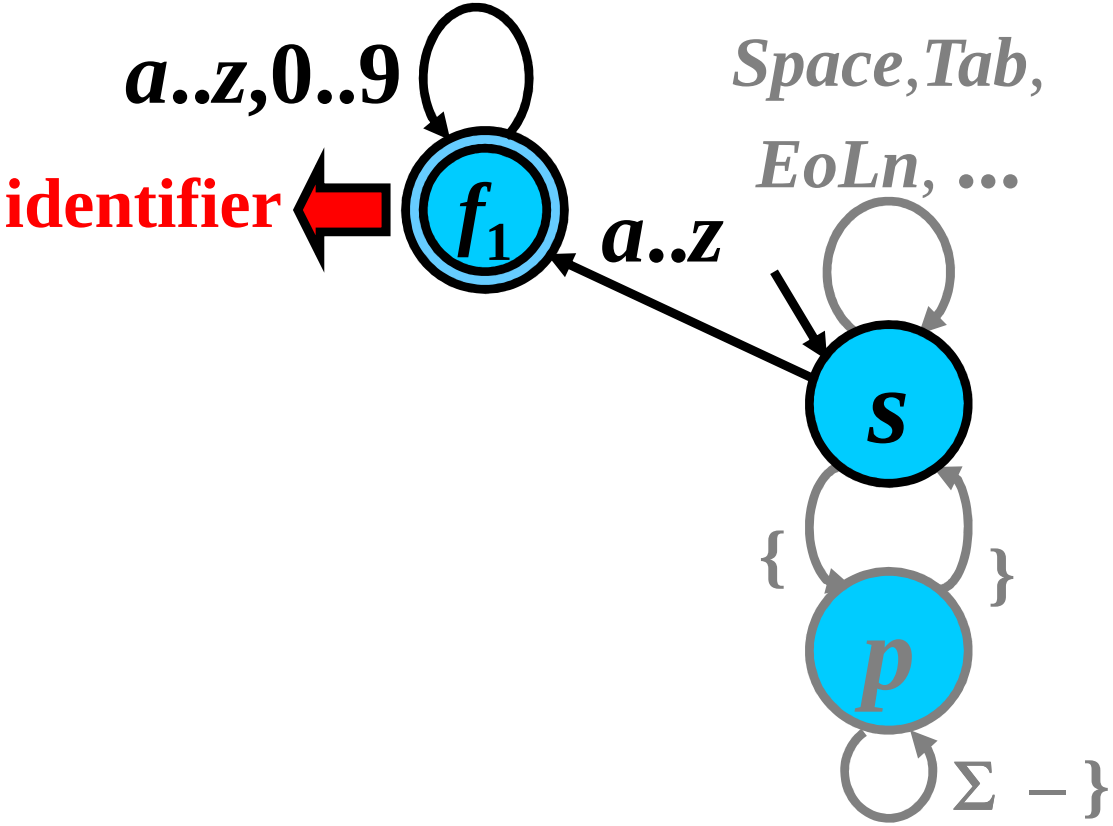
DFA for Lexemes : Example 1/2

- FA that accepts these lexemes:



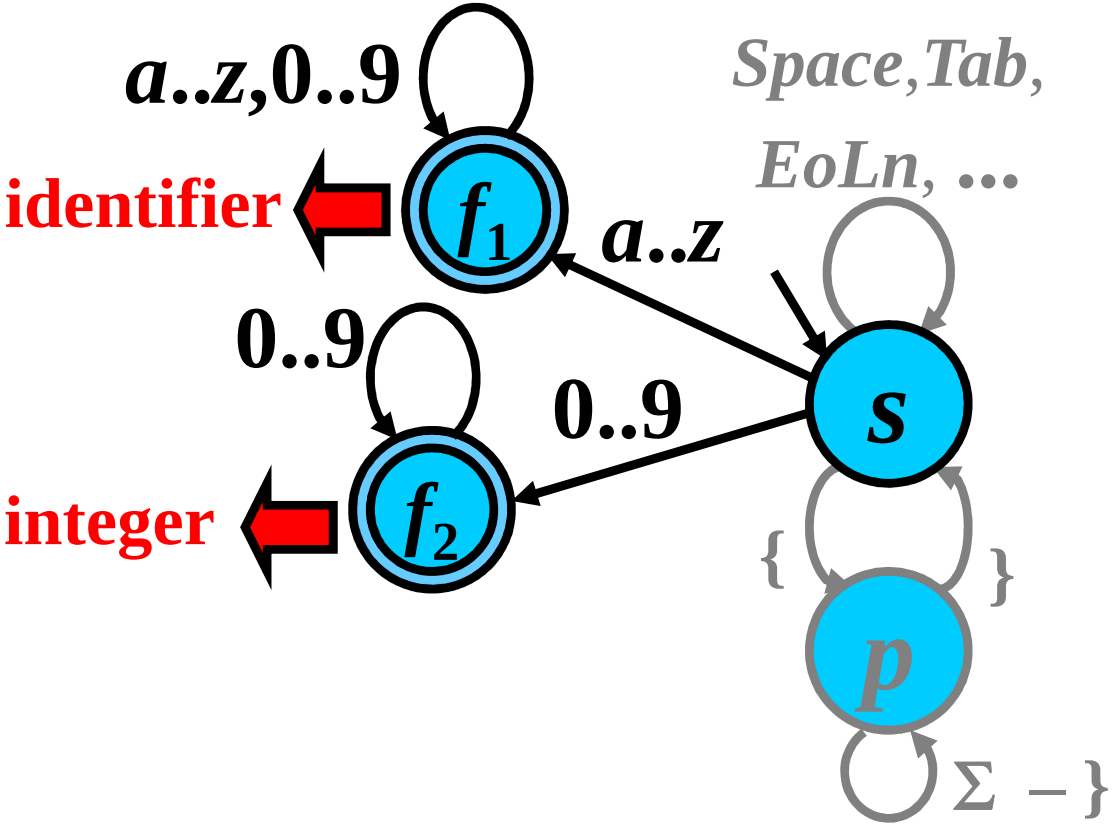
DFA for Lexemes : Example 1/2

- FA that accepts these lexemes:
  identifier,



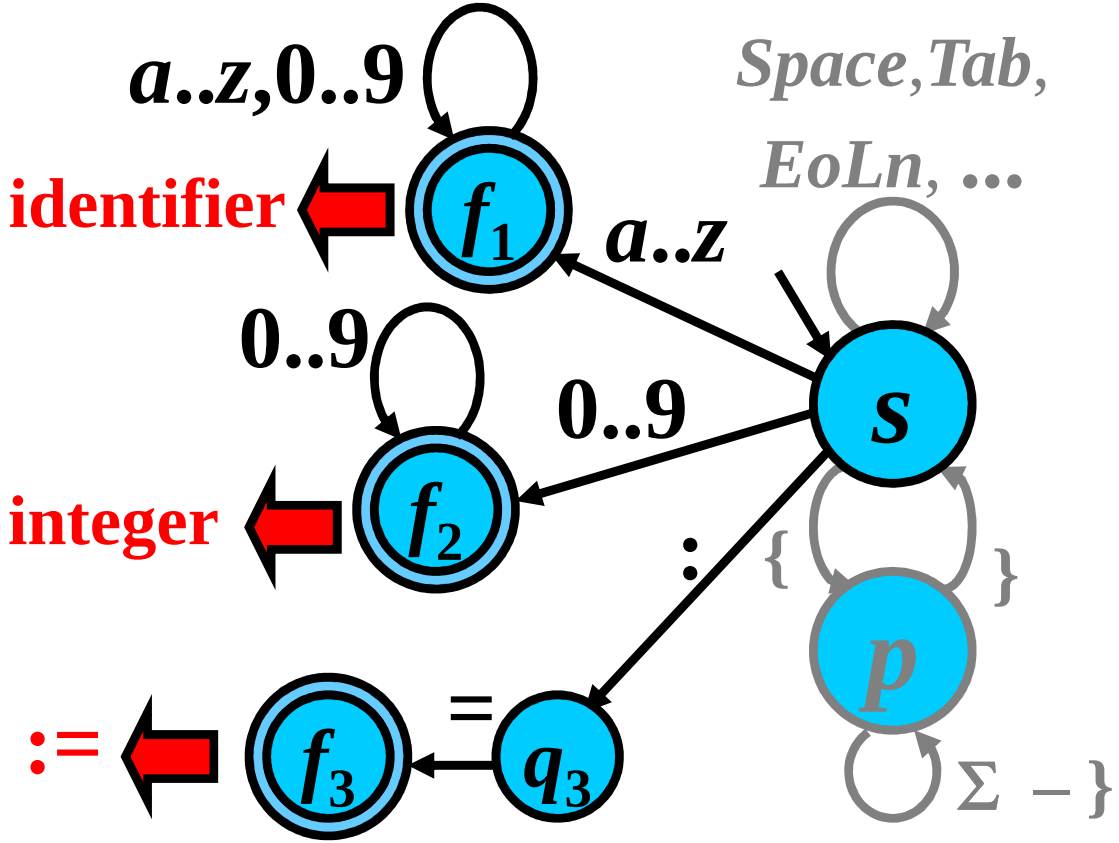
DFA for Lexemes : Example 1/2

- FA that accepts these lexemes:
  identifier, integer,



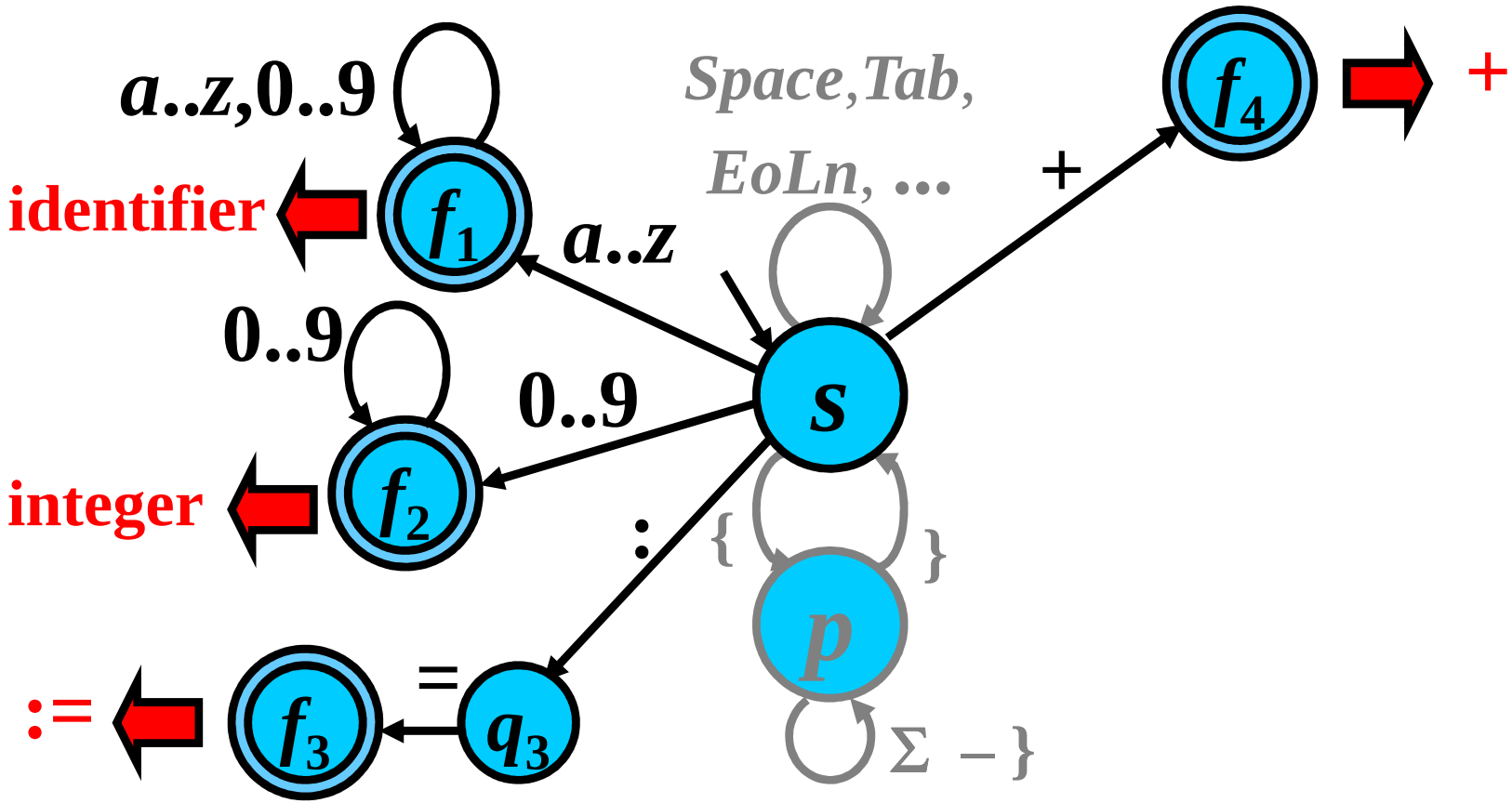
DFA for Lexemes : Example 1/2

- FA that accepts these lexemes:
  identifier, integer, :=,



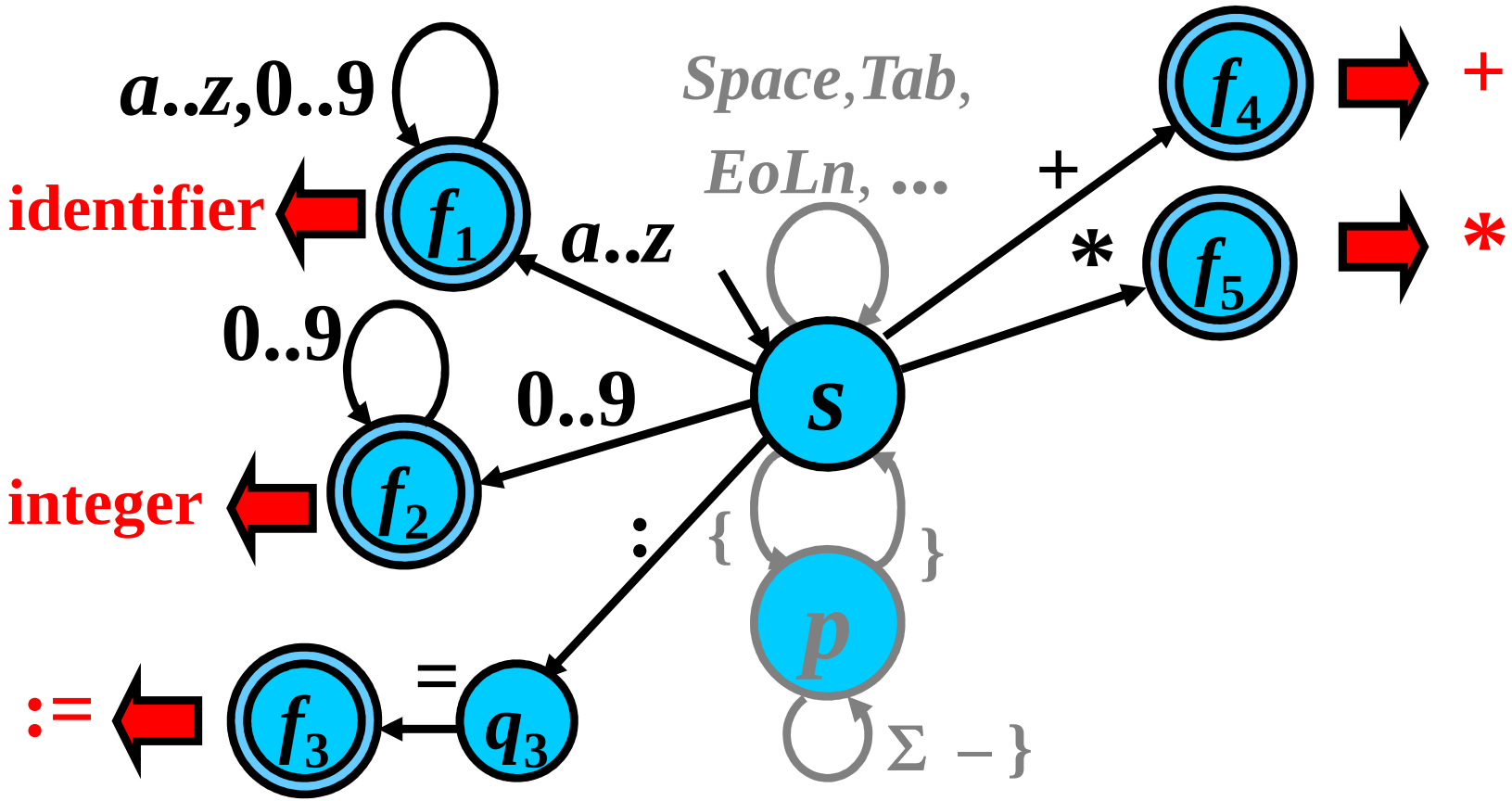
DFA for Lexemes : Example 1/2

- FA that accepts these lexemes:
  identifier, integer, :=, +,



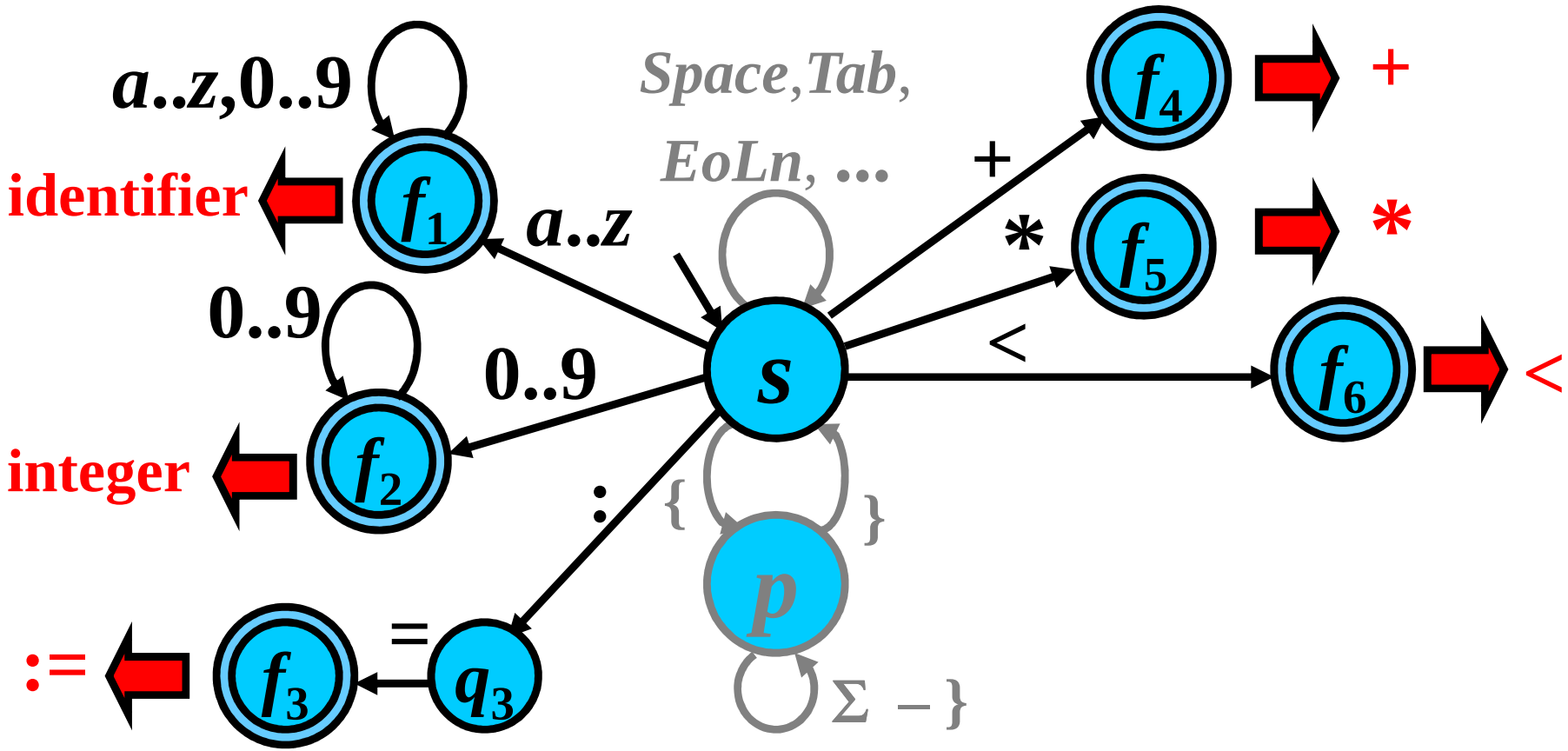
DFA for Lexemes : Example 1/2

- FA that accepts these lexemes:
  identifier, integer, :=, +, *,



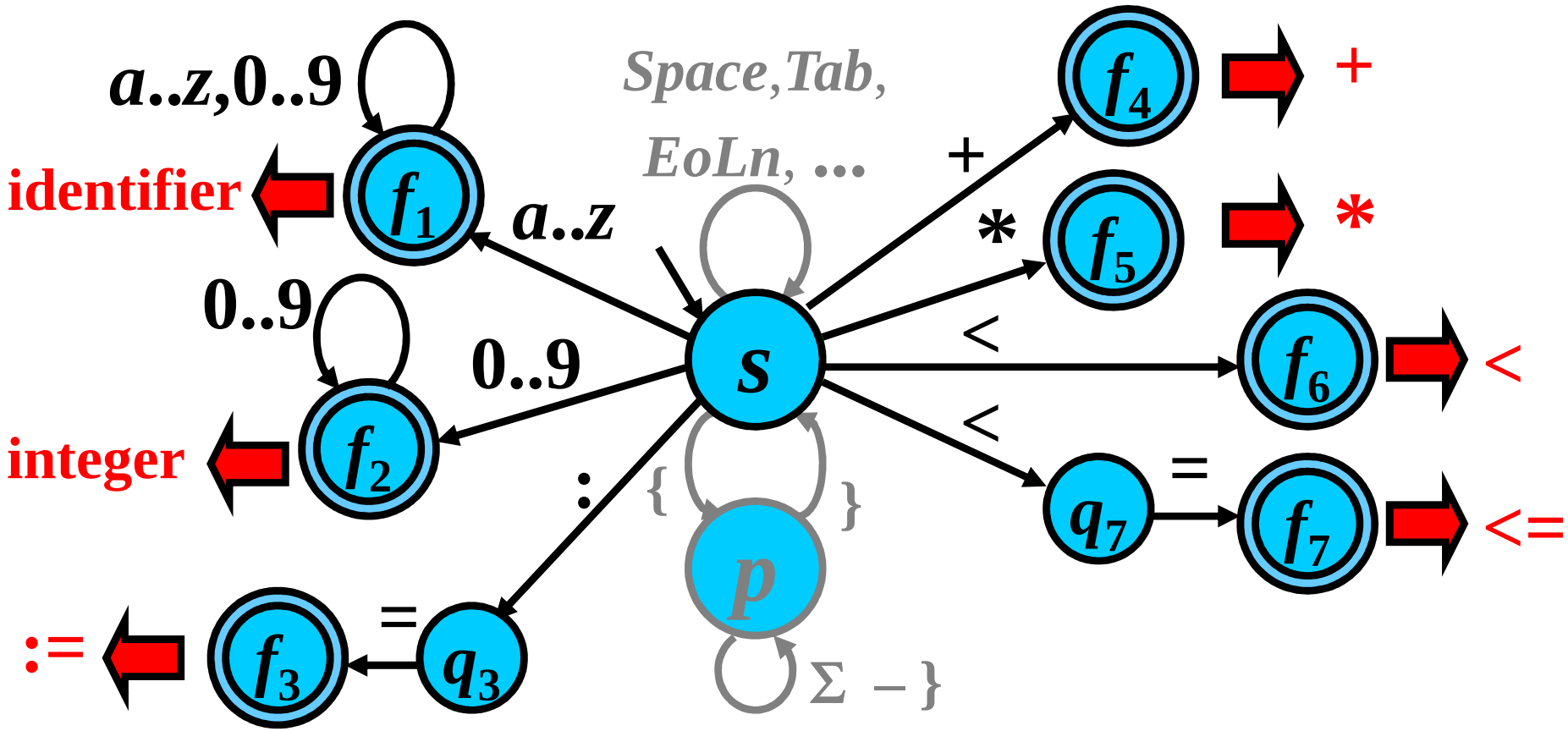
DFA for Lexemes : Example 1/2

- FA that accepts these lexemes:
  identifier, integer, :=, +, *, <,



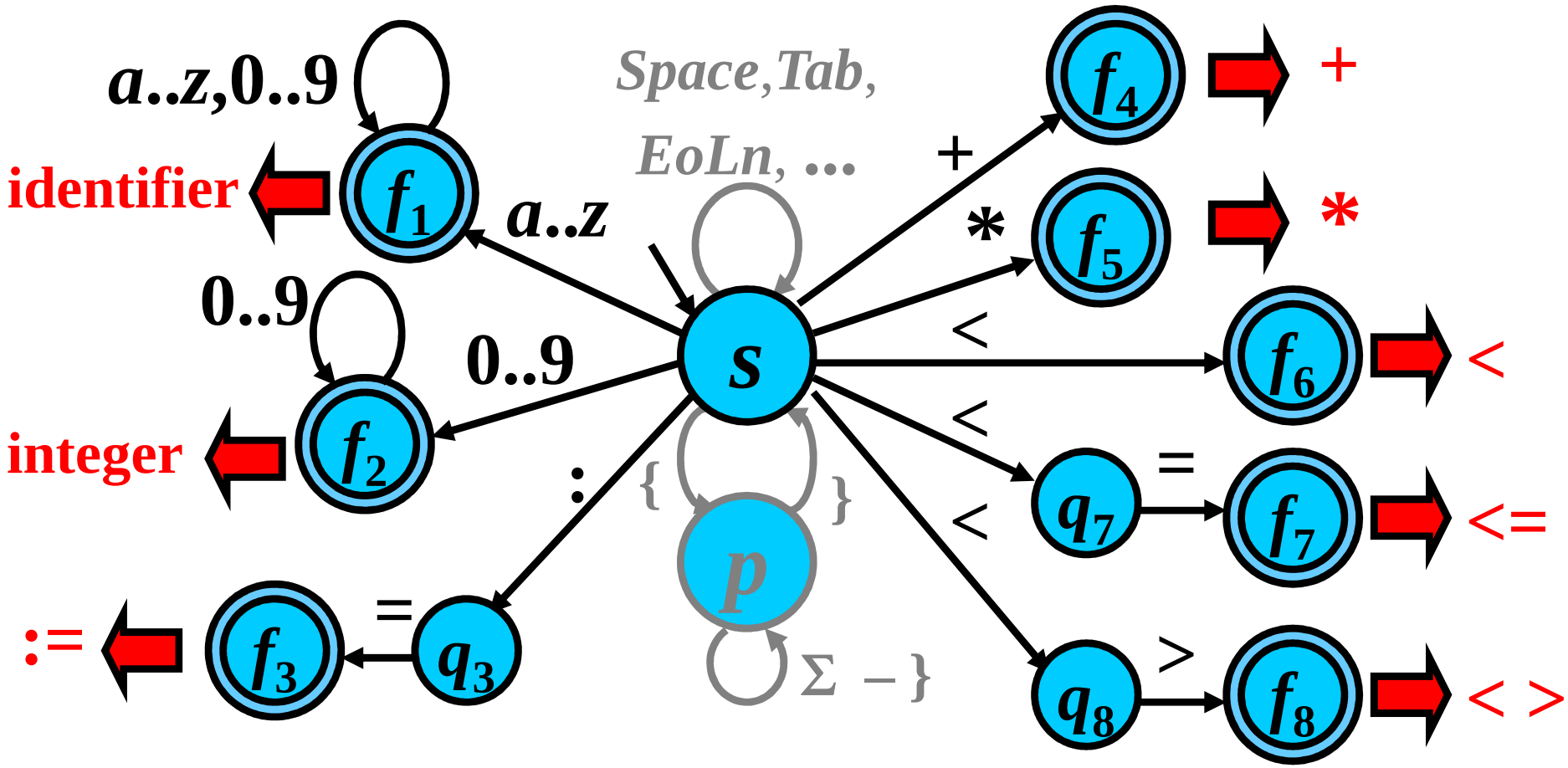
DFA for Lexemes : Example 1/2

- FA that accepts these lexemes:
  identifier, integer, :=, +, *, <, <=,



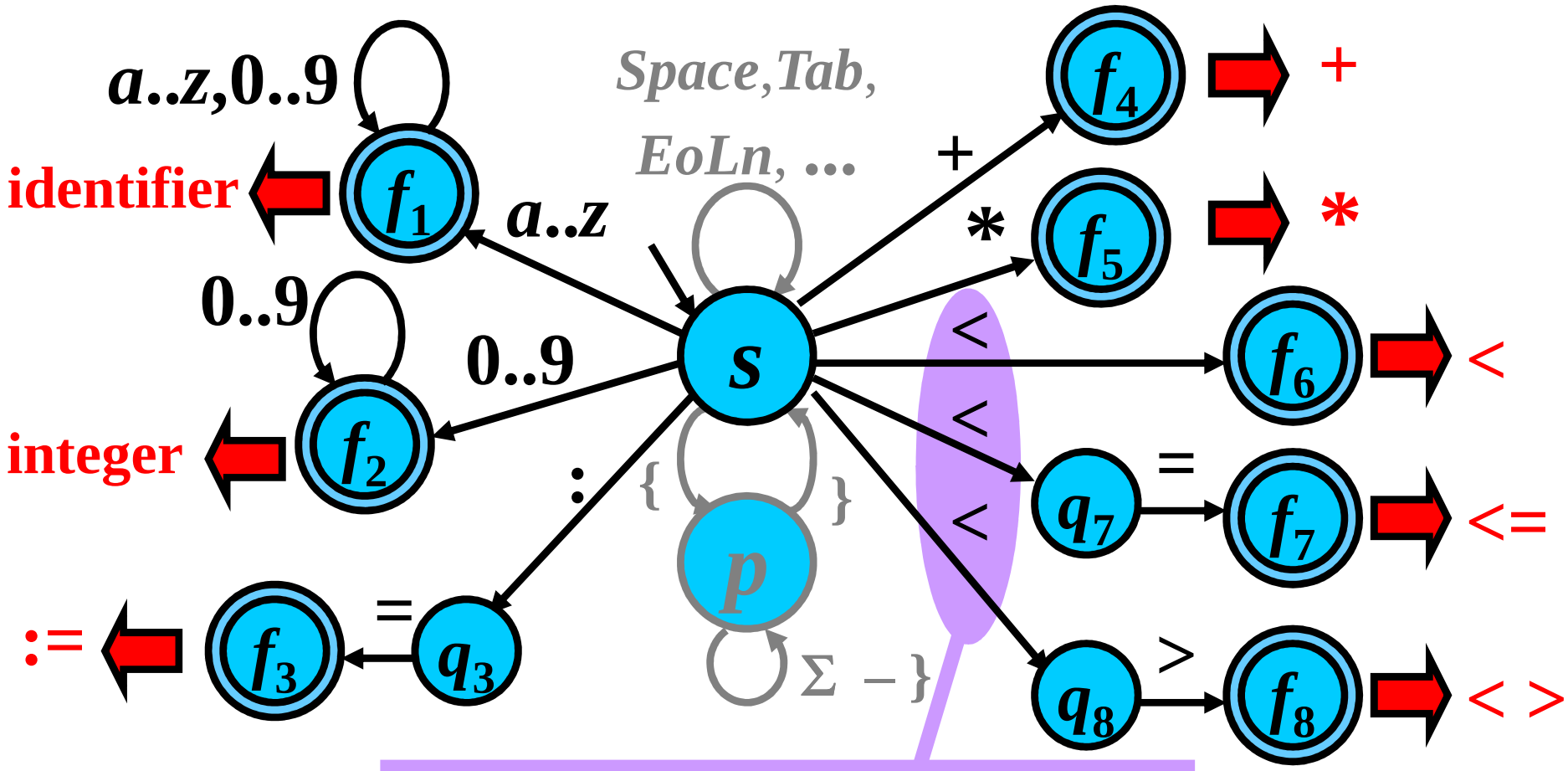
DFA for Lexemes : Example 1/2

- FA that accepts these lexemes:
  identifier, integer, :=, +, *, <, <=, < >



DFA for Lexemes : Example 1/2

- FA that accepts these lexemes:
  identifier, integer, :=, +, *, <, <=, < >

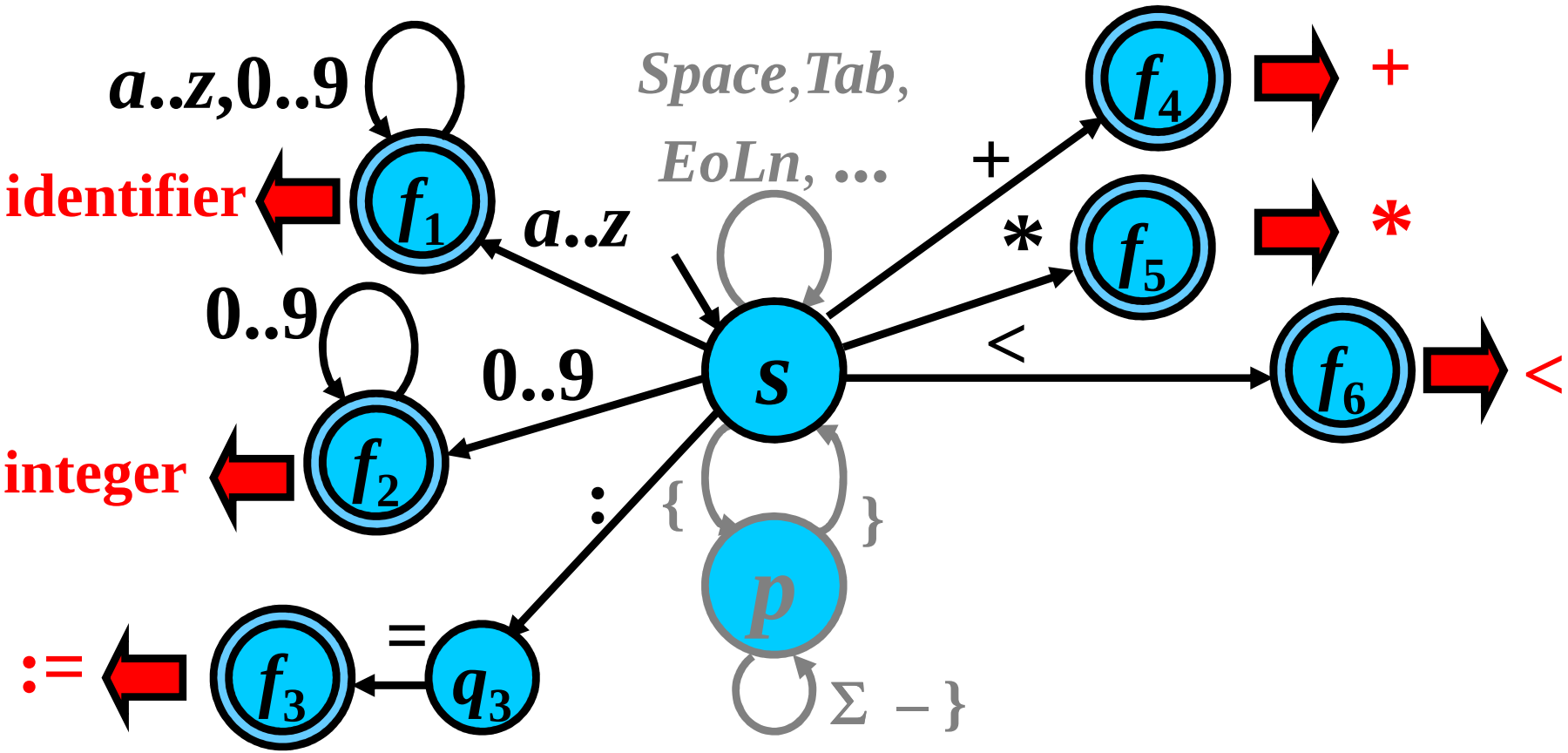


Convert this NFA to DFA.

DFAs for Lexemes: Example 2/2

- Equivalent DFA:

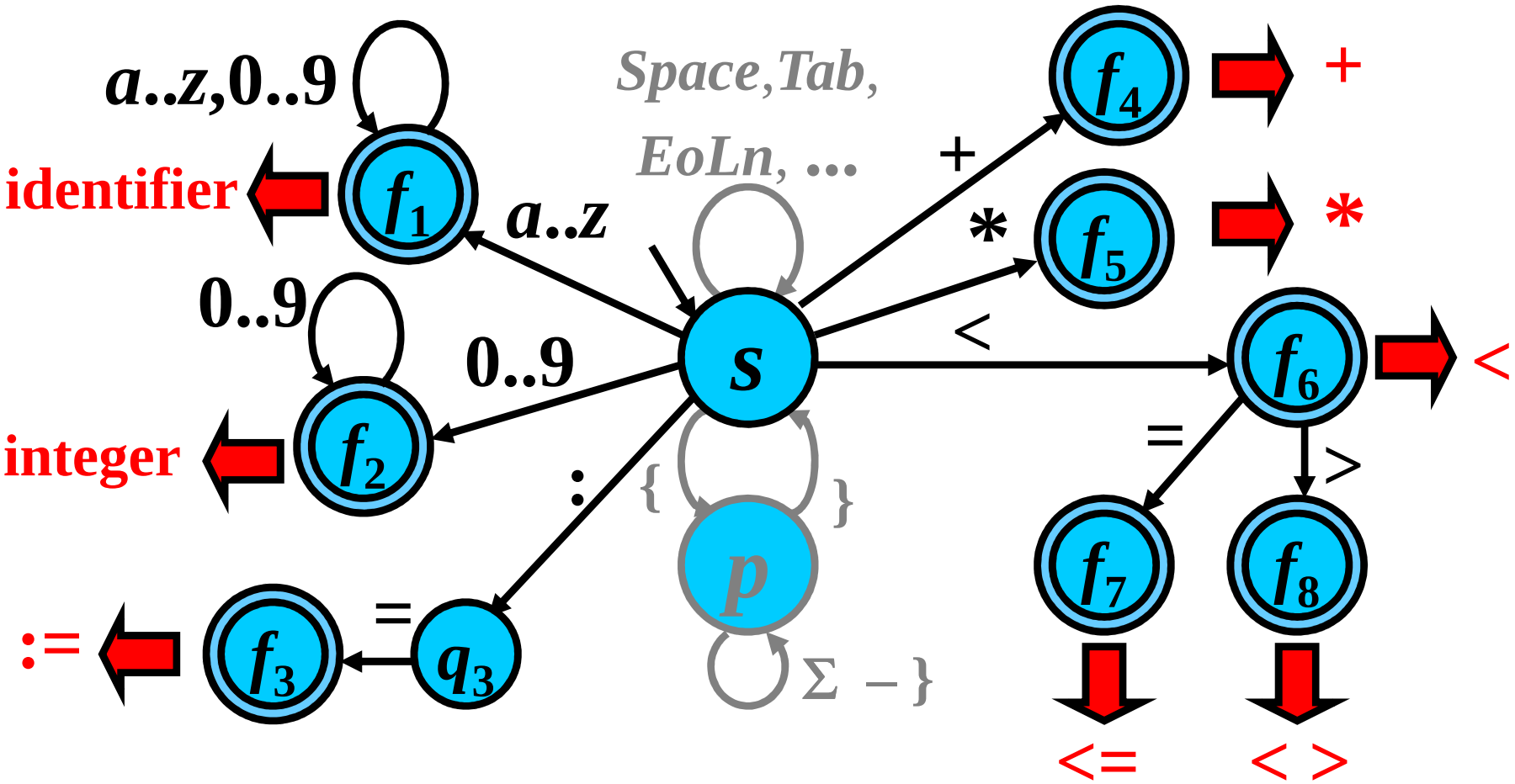
identifier, integer, :=, +, *, <, <=, < >



DFAs for Lexemes: Example 2/2

- Equivalent DFA:

identifier, integer, :=, +, *, <, <=, < >

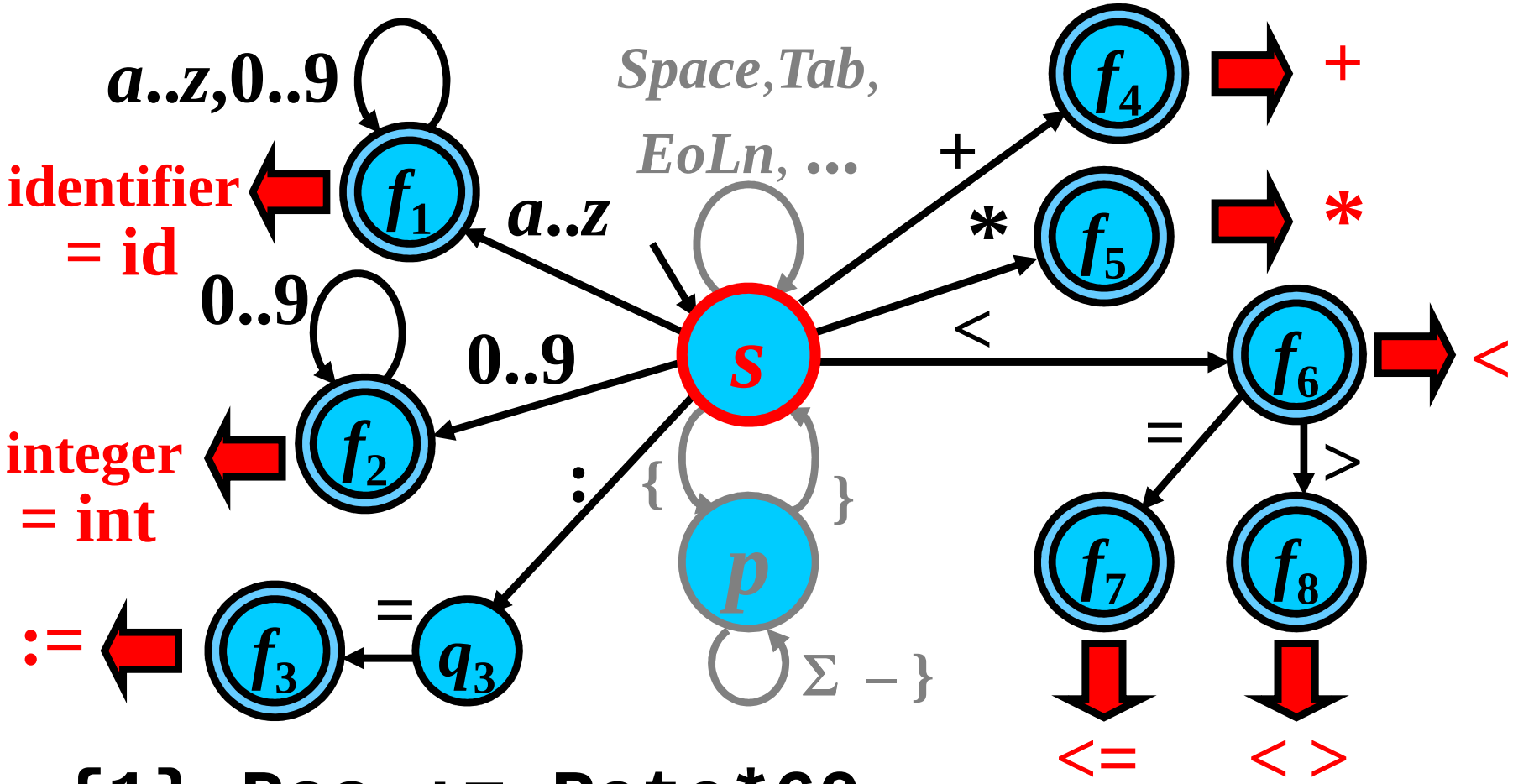


Algorithm: Type of Lexeme

- **Input:** DFA *M* for the source-program lexemes
- **Output:** determination of the lexeme type

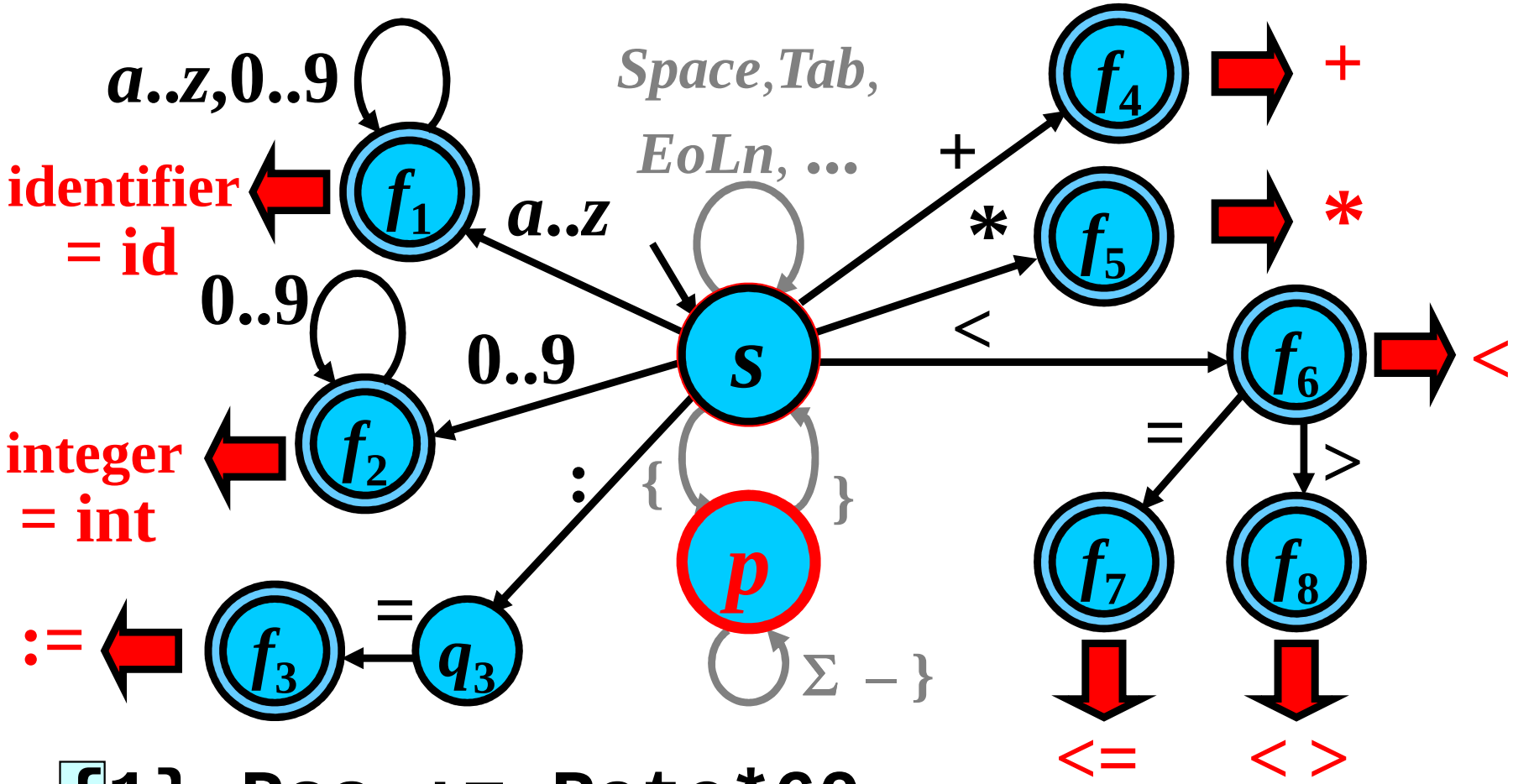
- **Method:**
- **while** ***a*** is the next symbol (character) in SP
  and *M* can make a move with ***a*** **do:**
    - read ***a***
    - make the move with ***a***
- **if** *M* is in a final state **then**
    determine the corresponding lexeme type
  else handle the lexical error (write message etc)

Type of Lexemes: Example



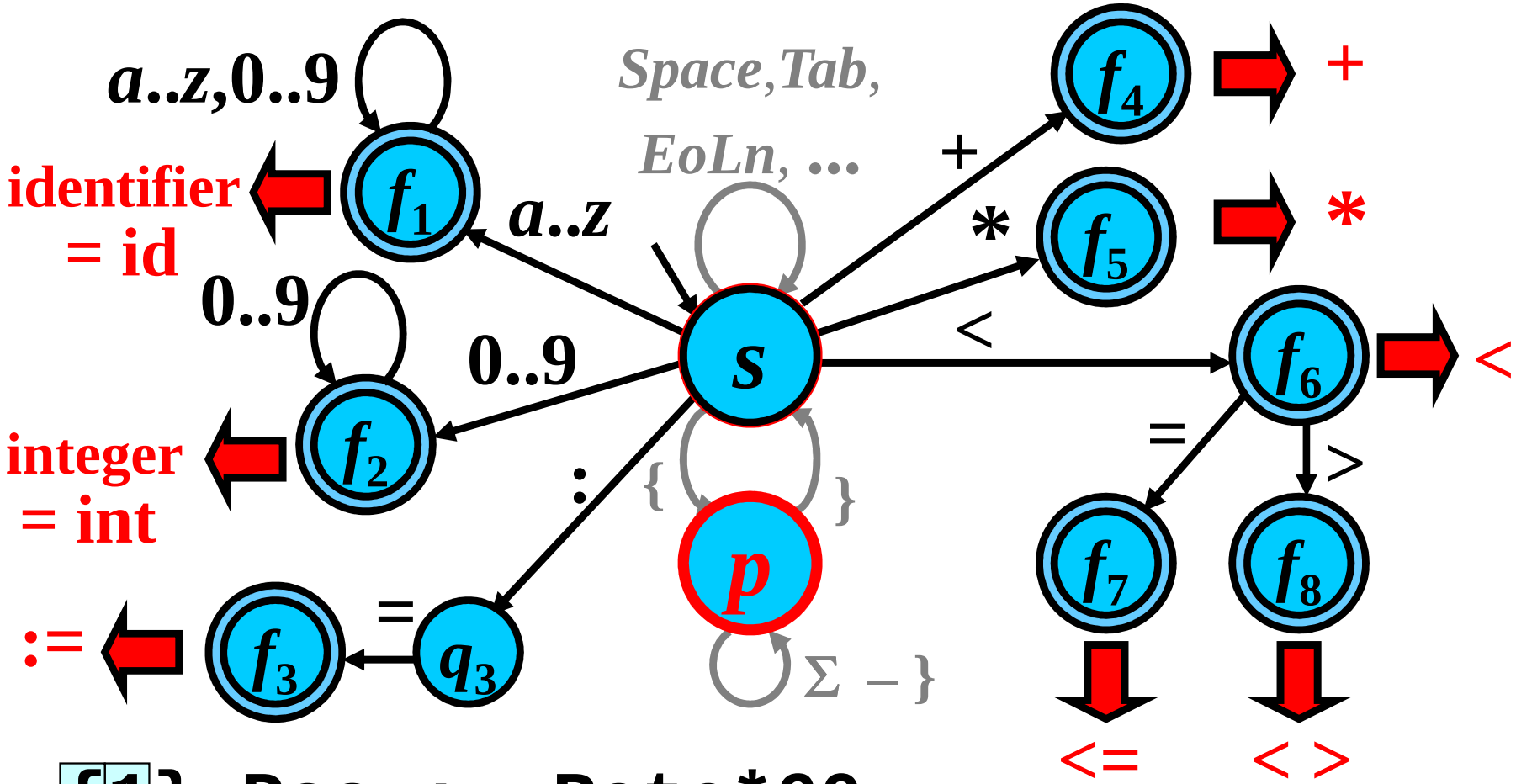

```
{1} Pos := Rate*60
```

Type of Lexemes: Example



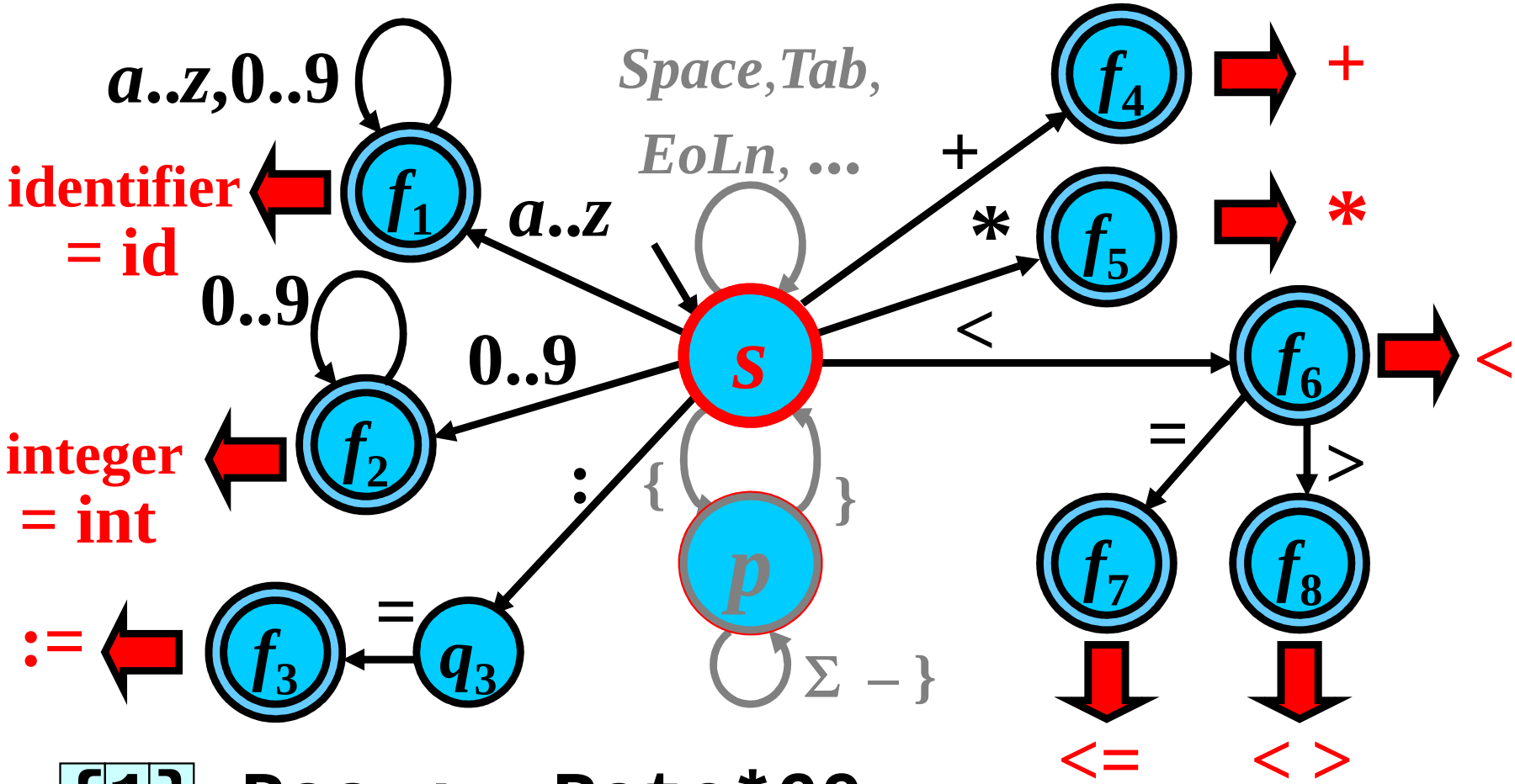

```
{1} Pos := Rate*60
```

Type of Lexemes: Example



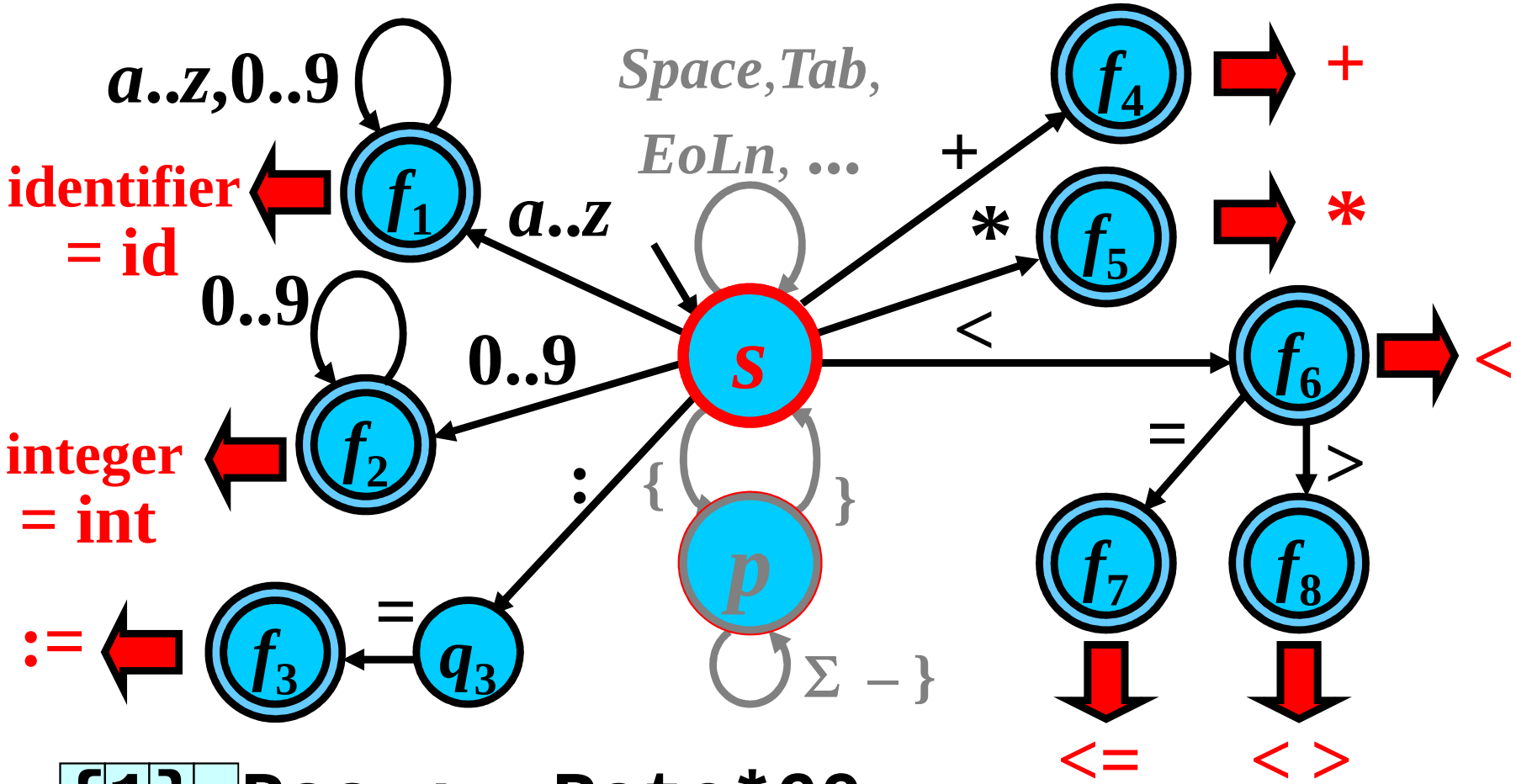

```
{1} Pos := Rate*60
```

Type of Lexemes: Example



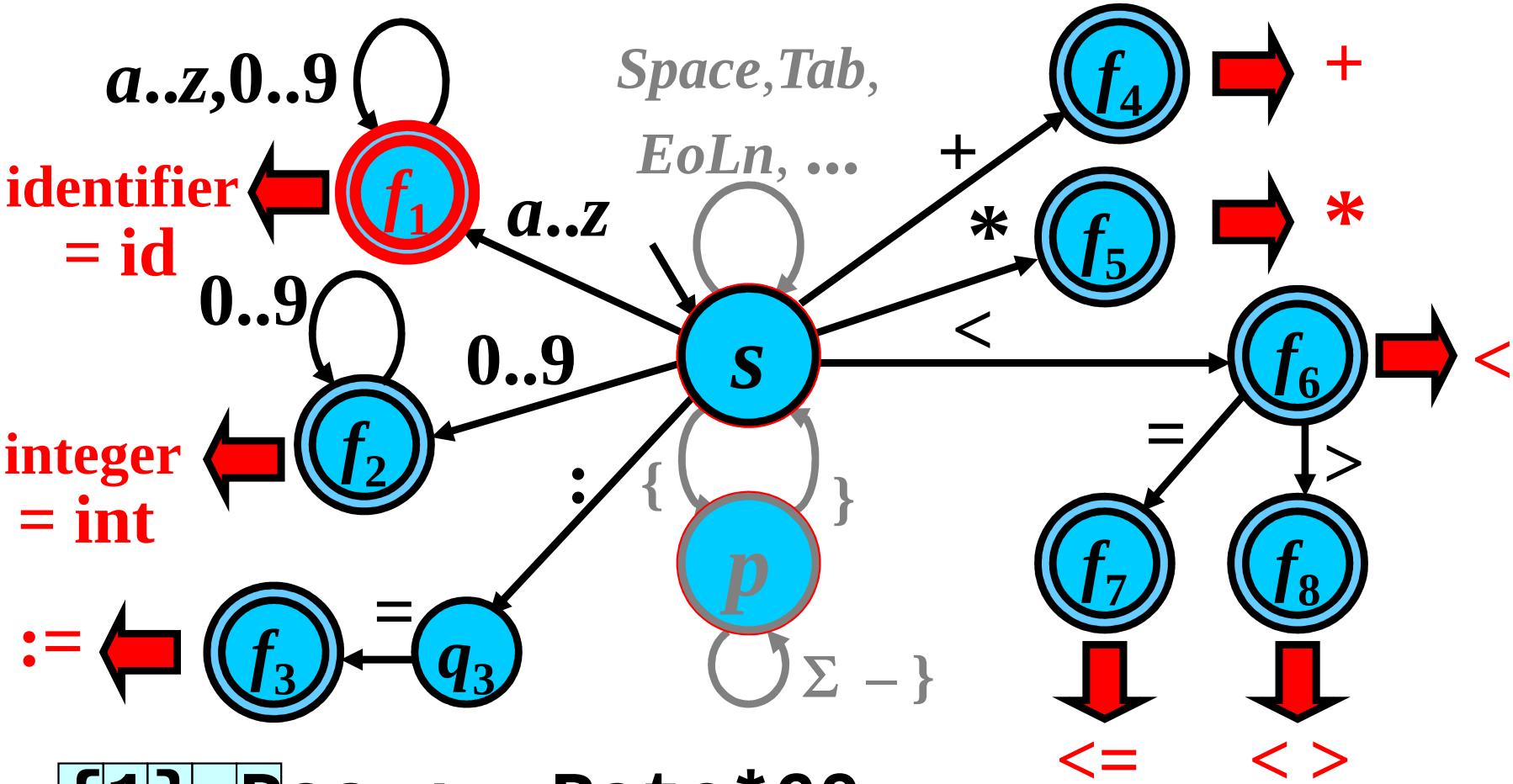

```
{1} Pos := Rate*60
```

Type of Lexemes: Example




```
{1} Pos := Rate*60
```

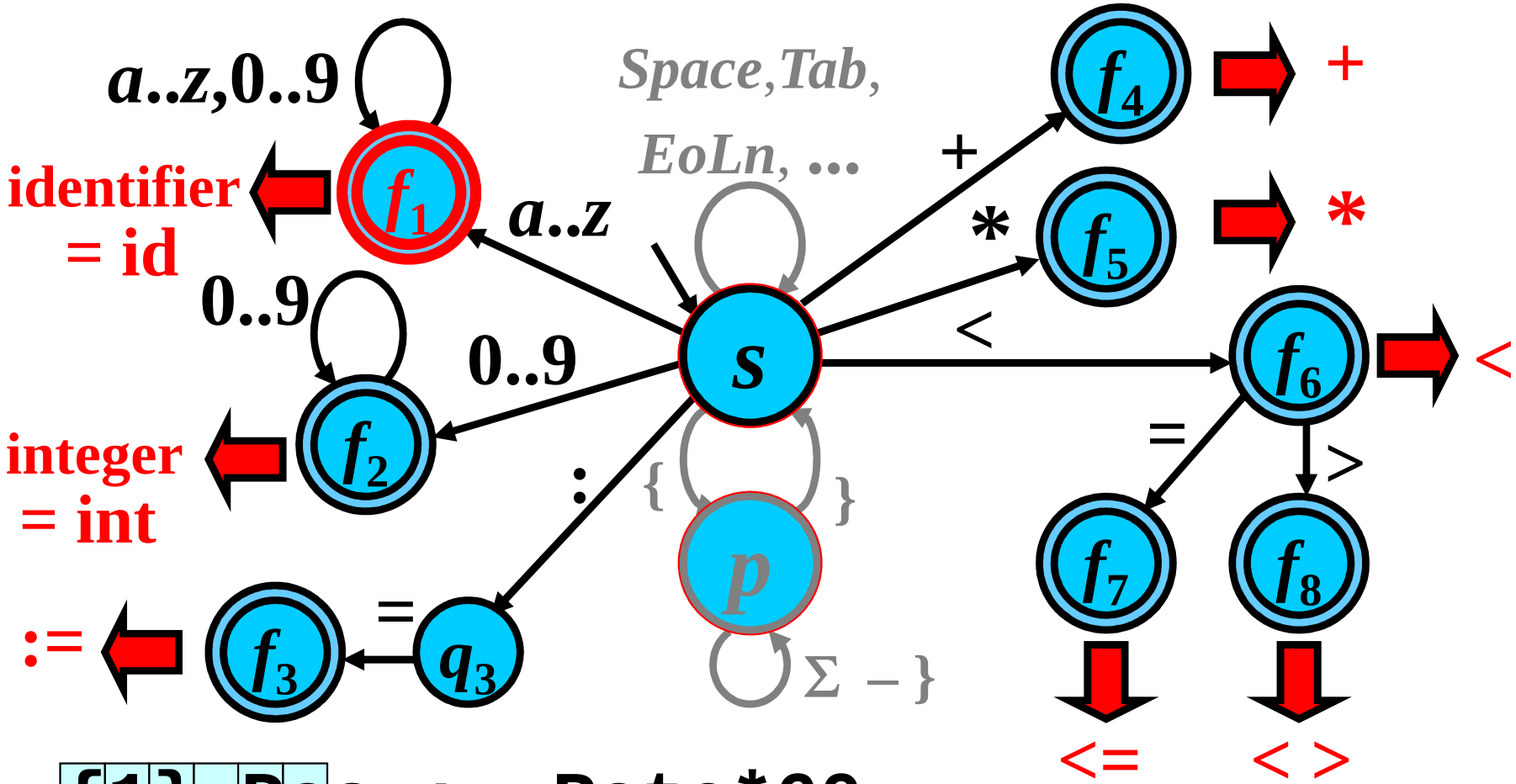
Type of Lexemes: Example




```
{1} Pos := Rate*60
```

P

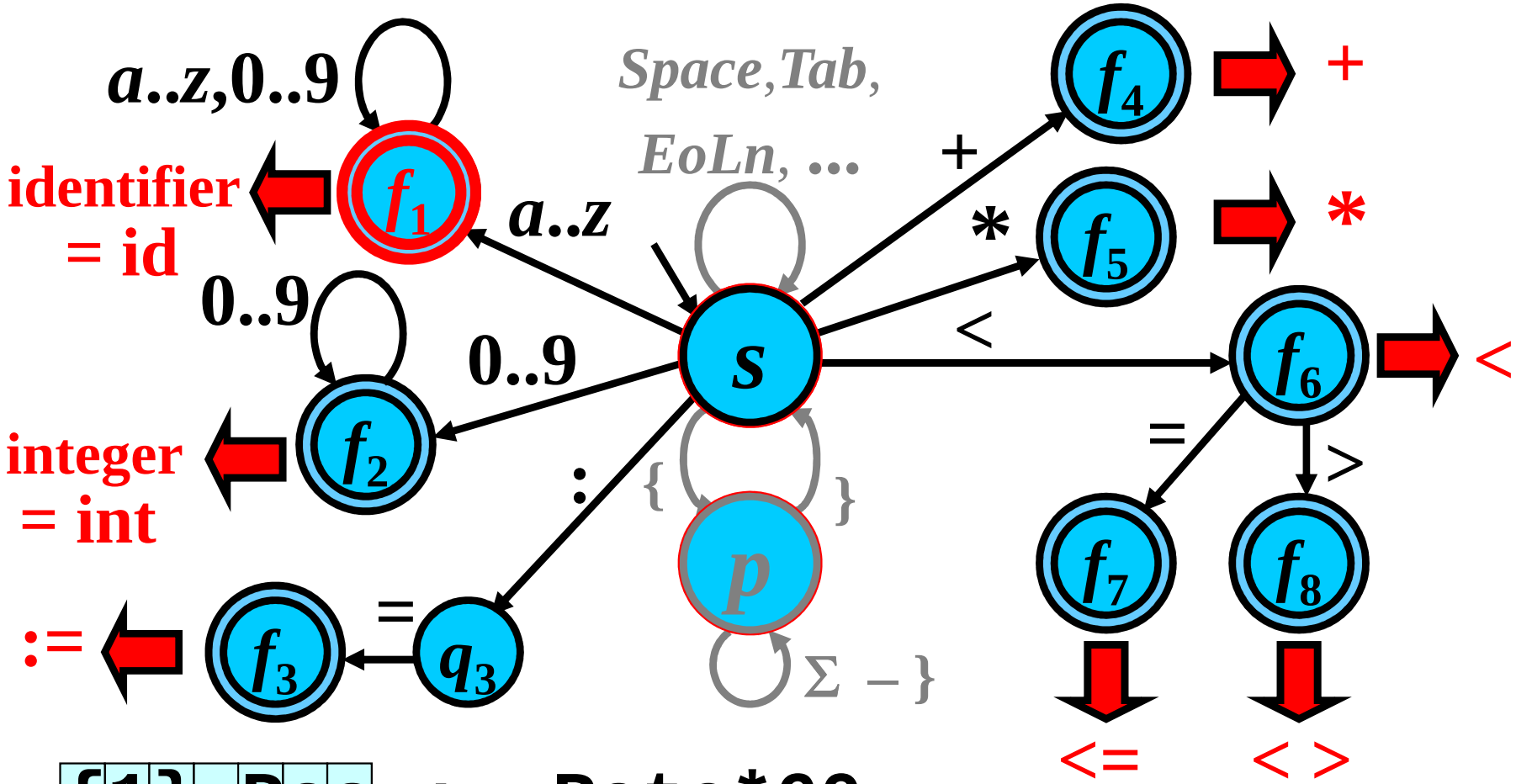
Type of Lexemes: Example



{1} Pos := Rate*60

Po

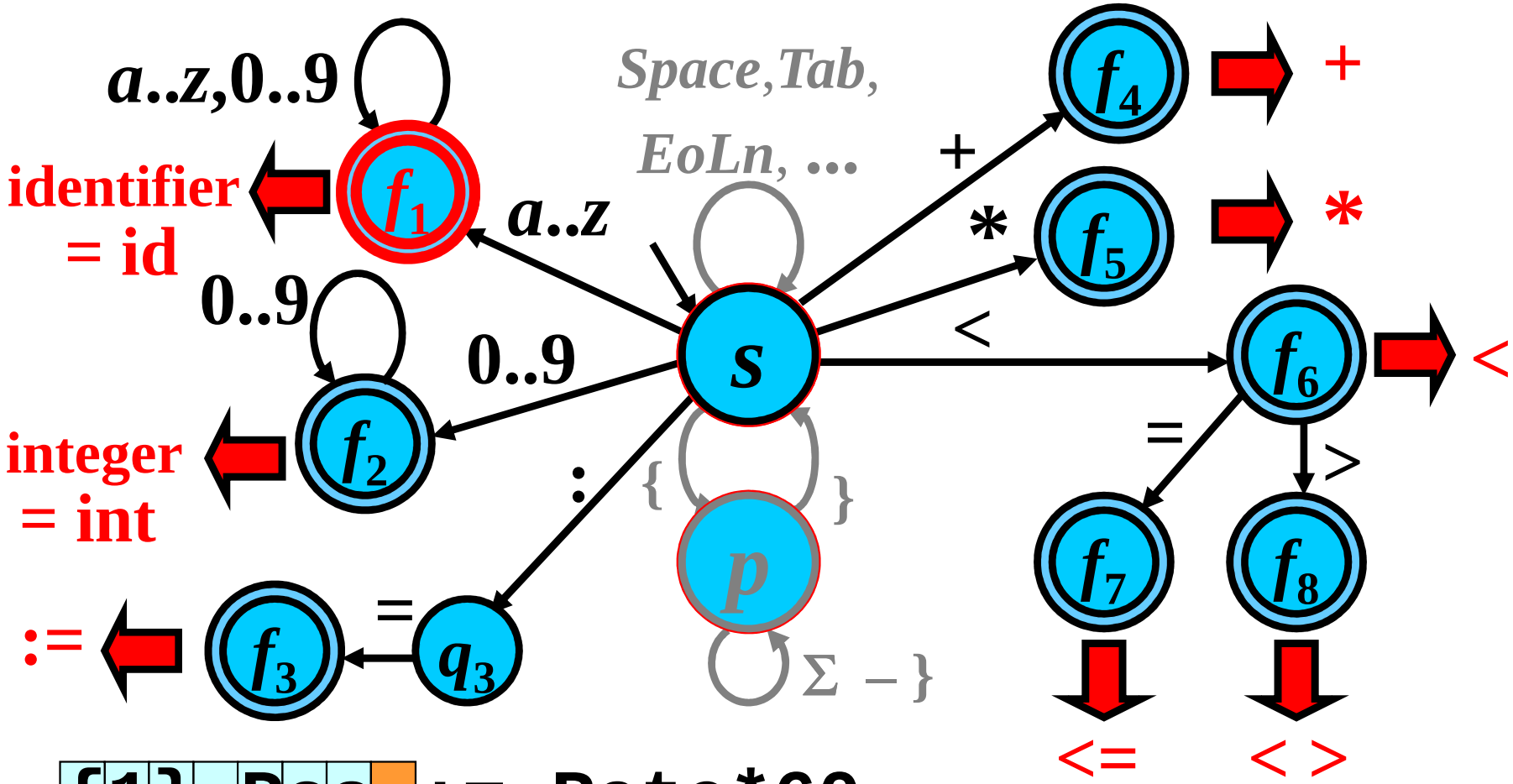
Type of Lexemes: Example



| {1} | | Pos | | | | | | := Rate*60

Pos

Type of Lexemes: Example

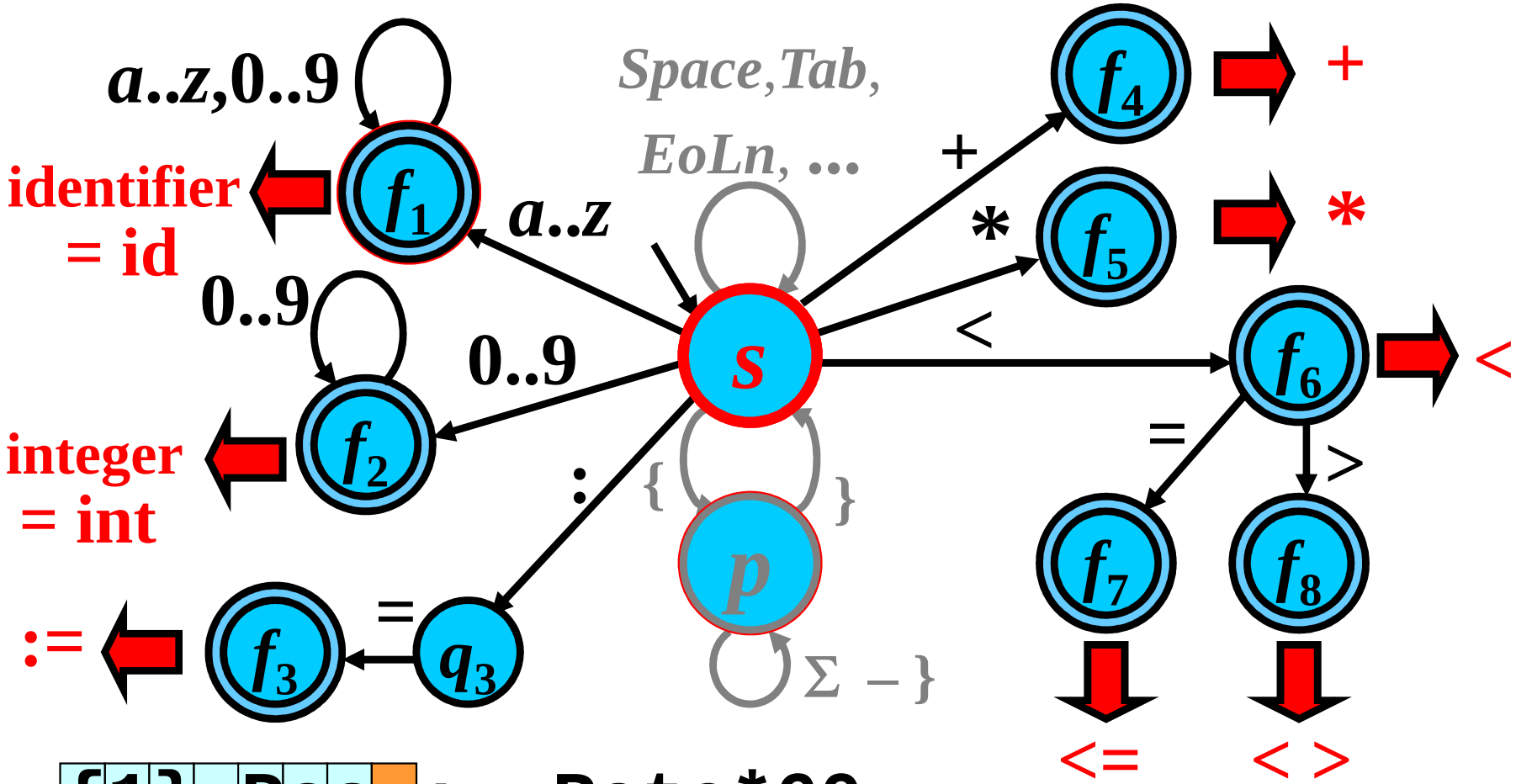


{1} Pos := Rate*60

id
Pos

No next configuration!

Type of Lexemes: Example

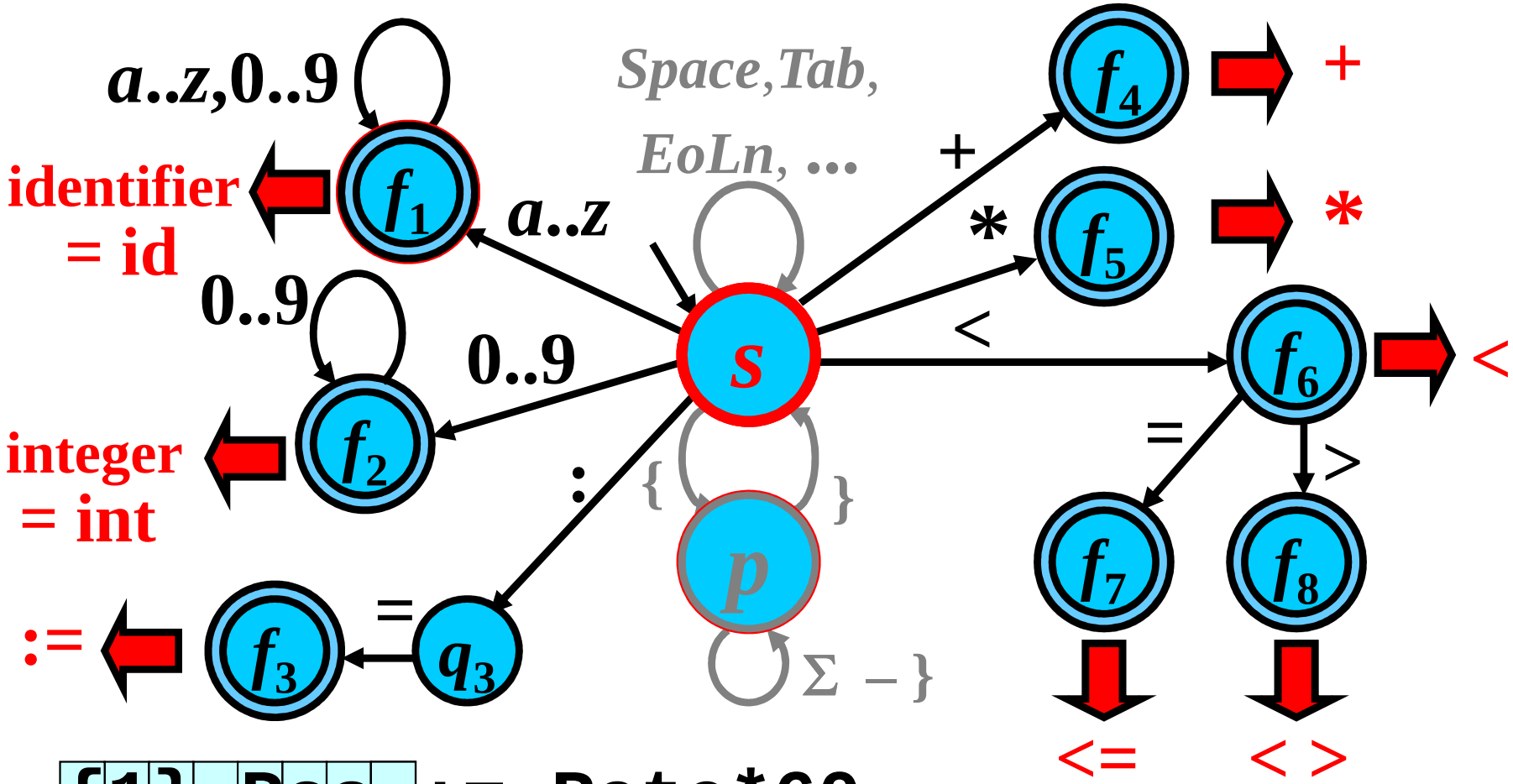


`{1} Pos := Rate*60`

id

Pos

Type of Lexemes: Example

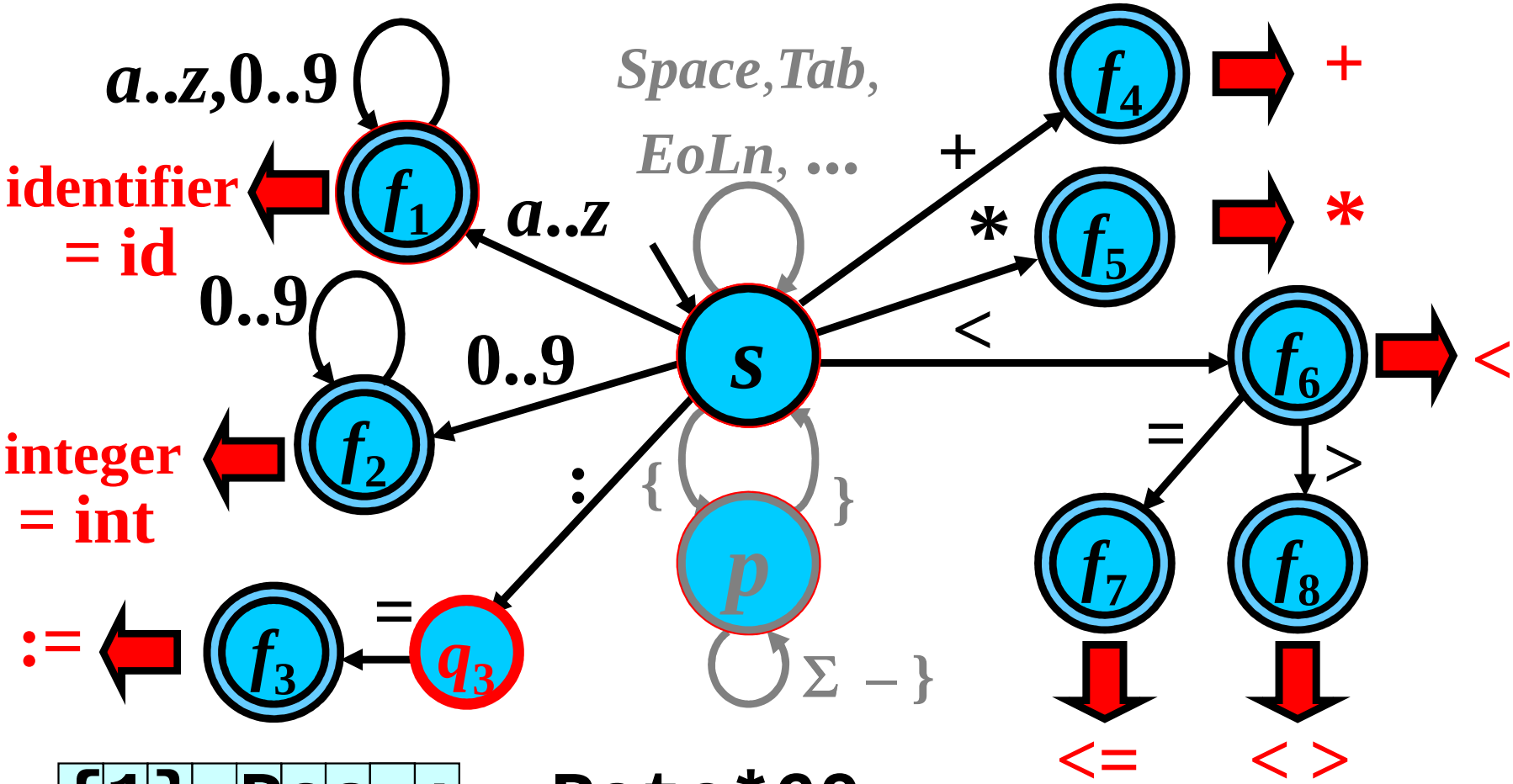


`{1} Pos := Rate*60`

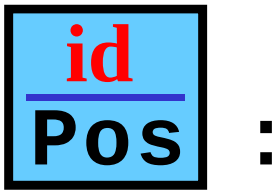
id

Pos

Type of Lexemes: Example

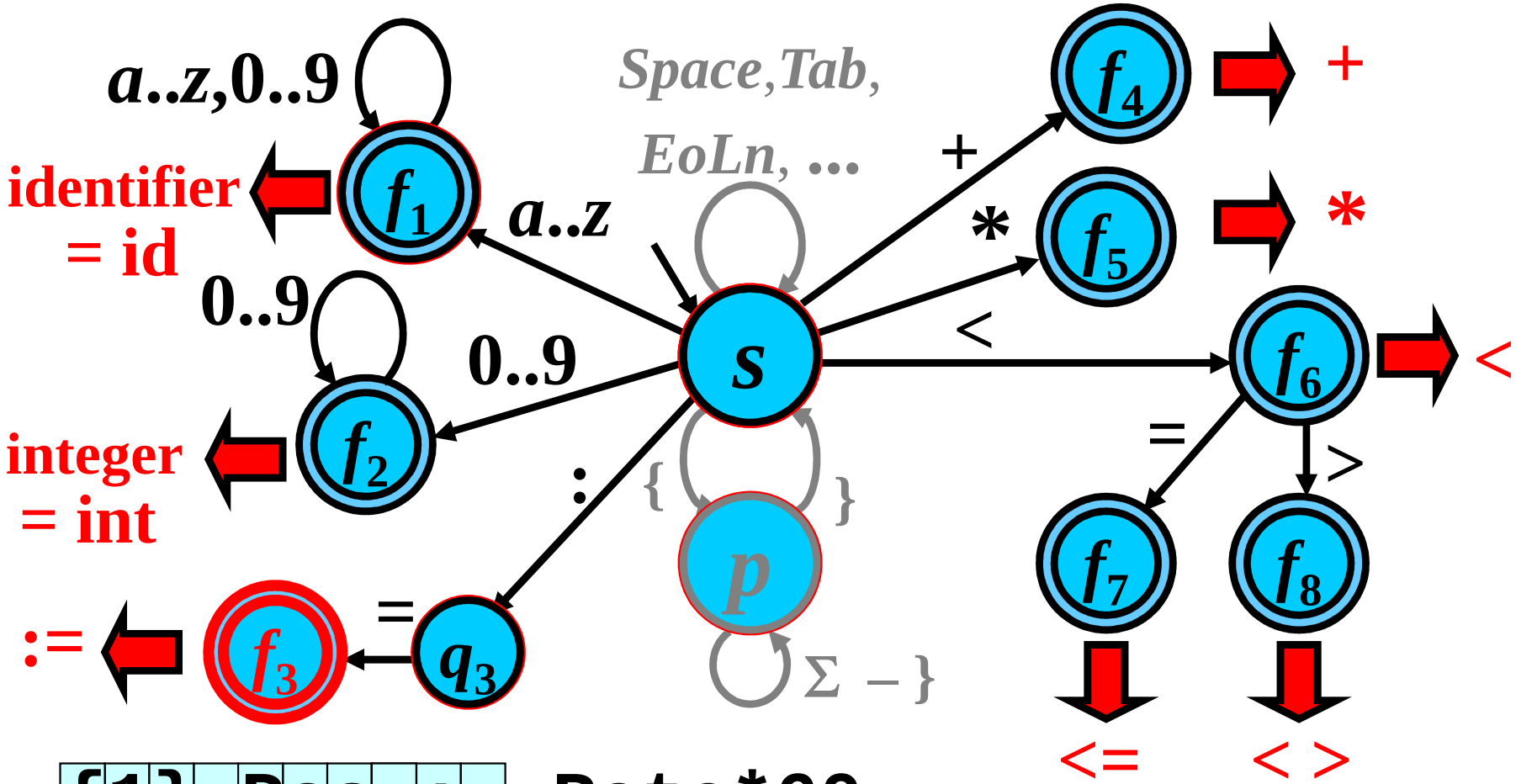


{1}		Pos									:= Rate*60



:

Type of Lexemes: Example



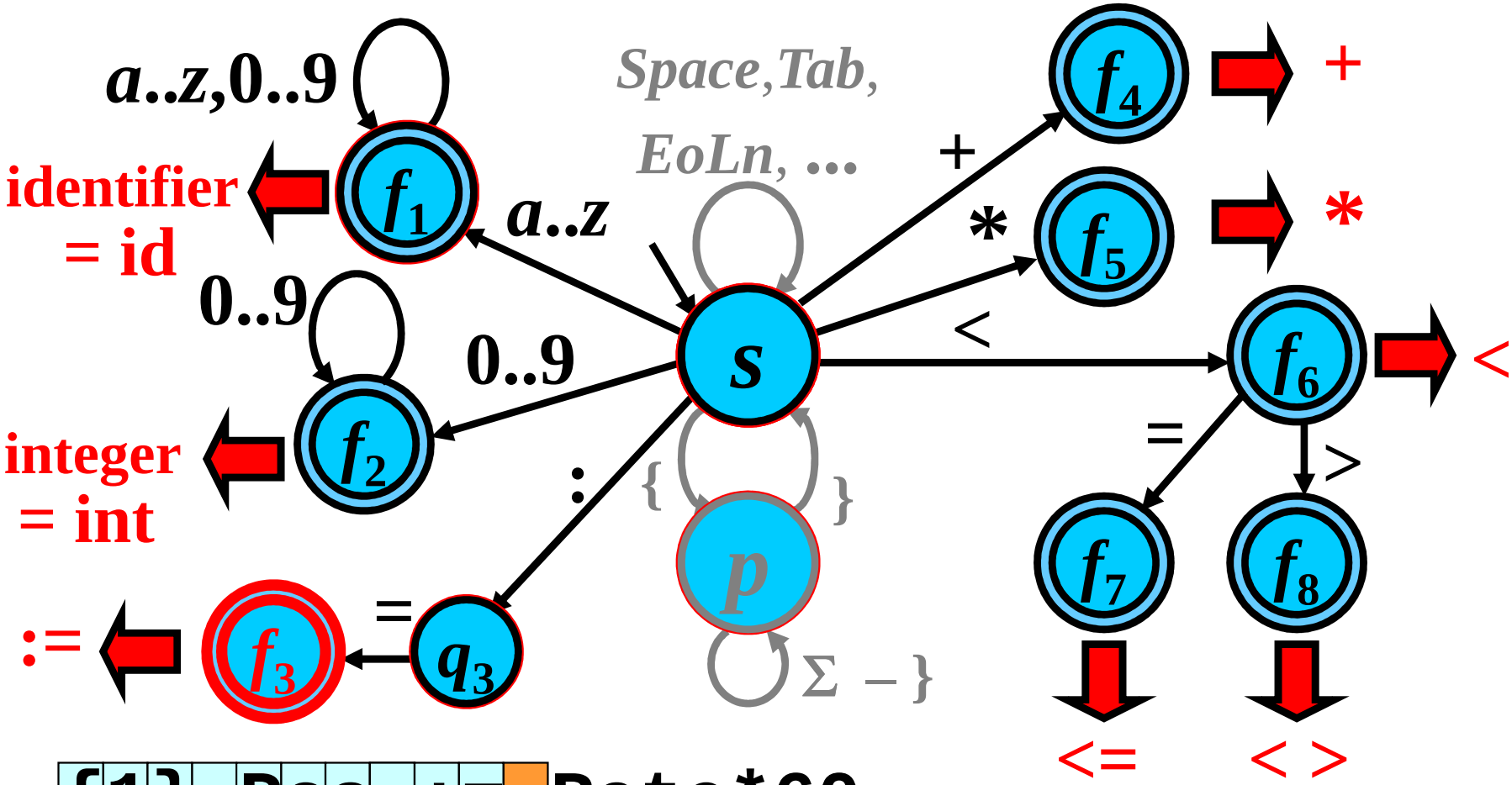
{1}		Pos					:=	

Rate*60

id
Pos

:=

Type of Lexemes: Example

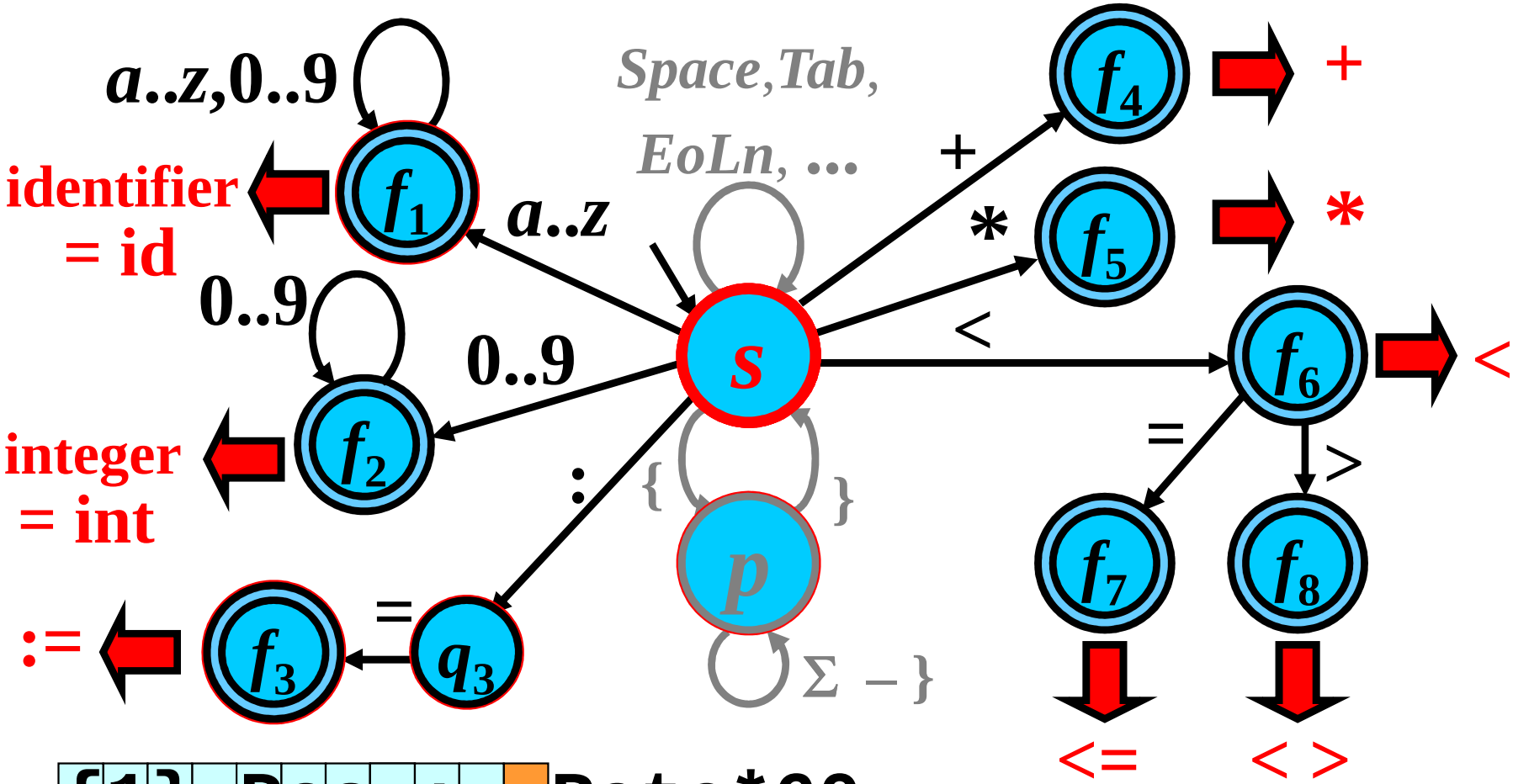


{1}		P	o	s			:	=		R	a	t	e	*	6	0

id	:=
Pos	

No next configuration!

Type of Lexemes: Example

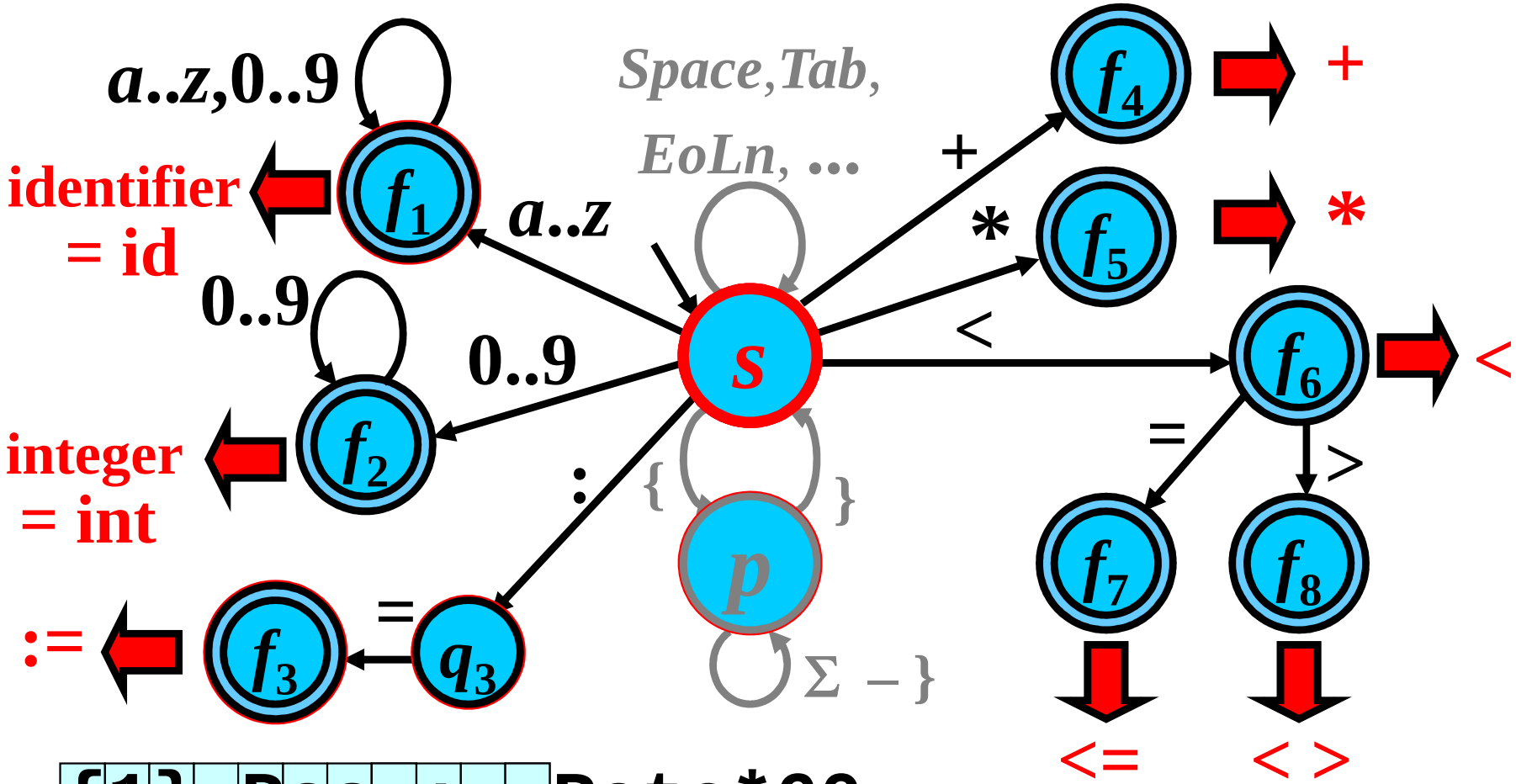


{1}		Pos																	

Rate*60

id Pos	:=

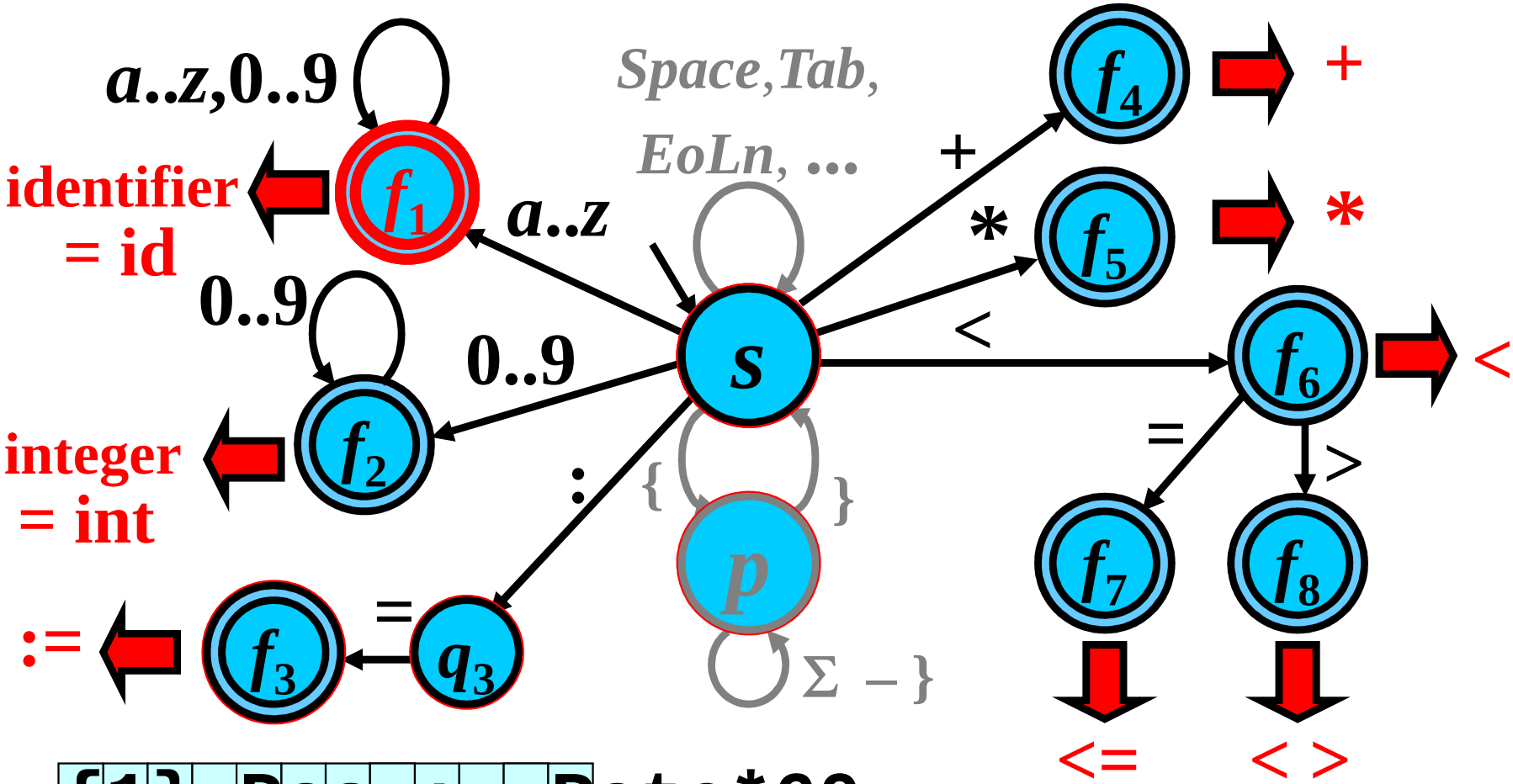
Type of Lexemes: Example



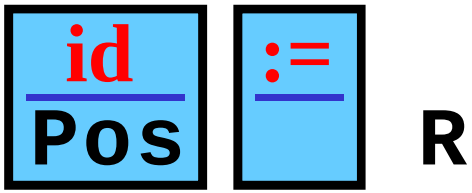
{1}		Pos				:=		Rate*60

id Pos	:=

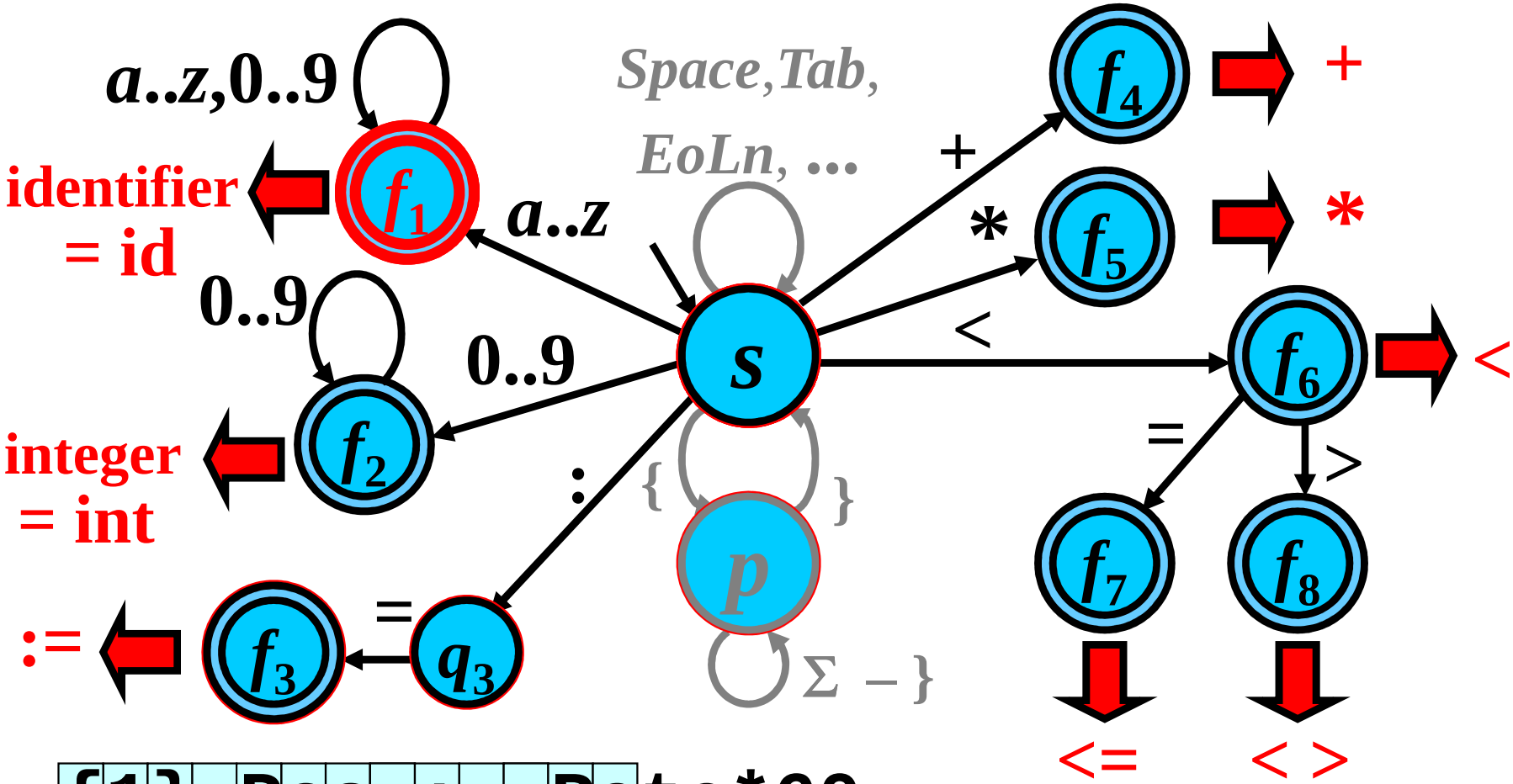
Type of Lexemes: Example



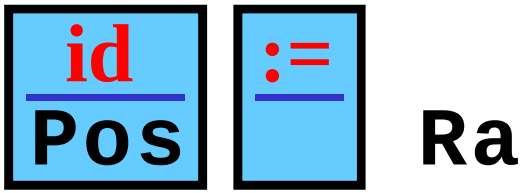
{1}		Pos		:=		Rate*60				



Type of Lexemes: Example

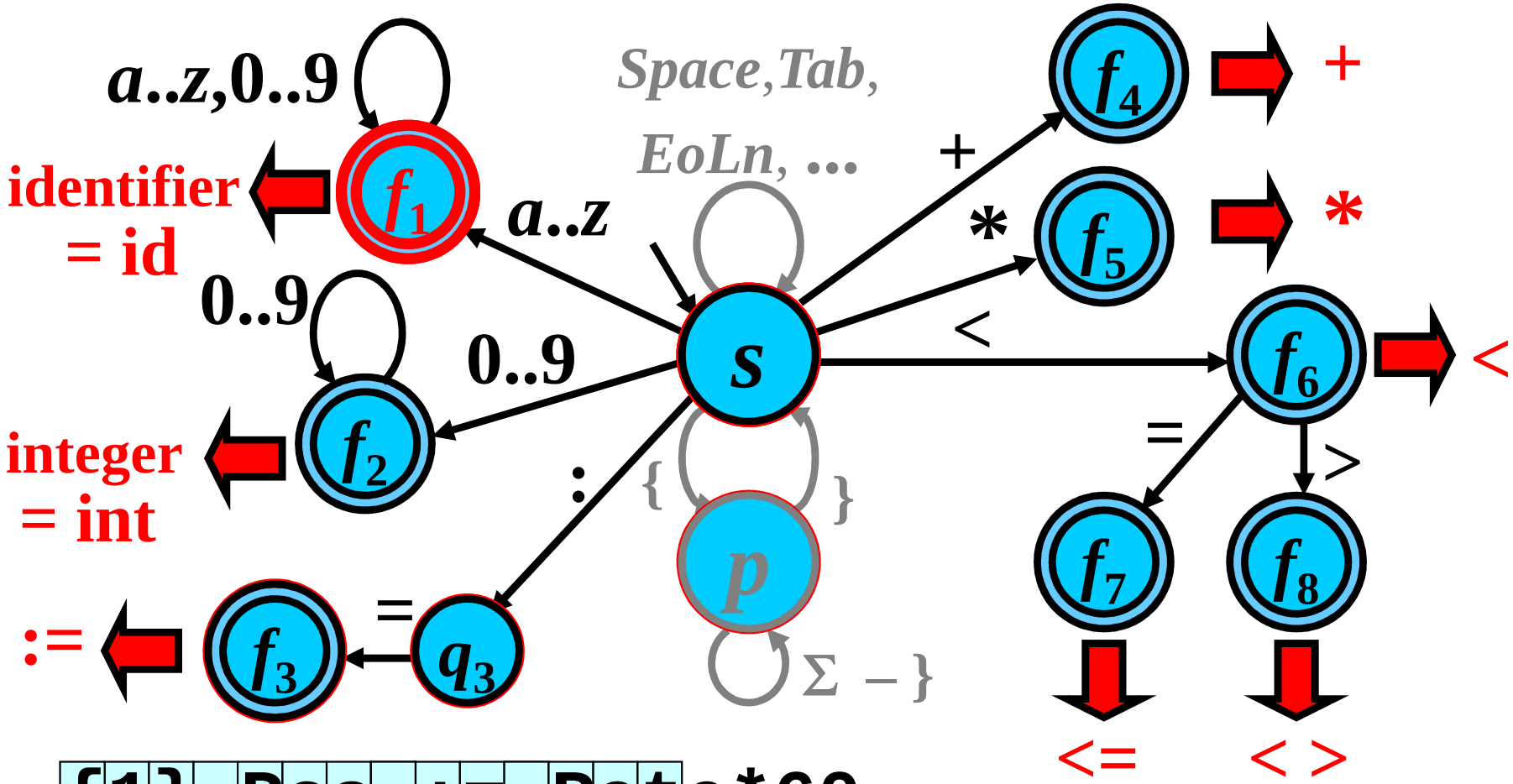


{1} Pos := Rate*60

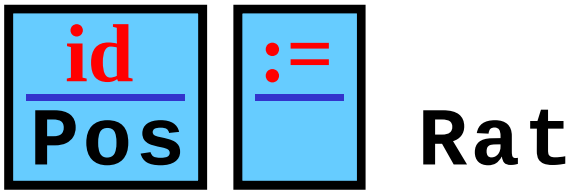


Ra

Type of Lexemes: Example

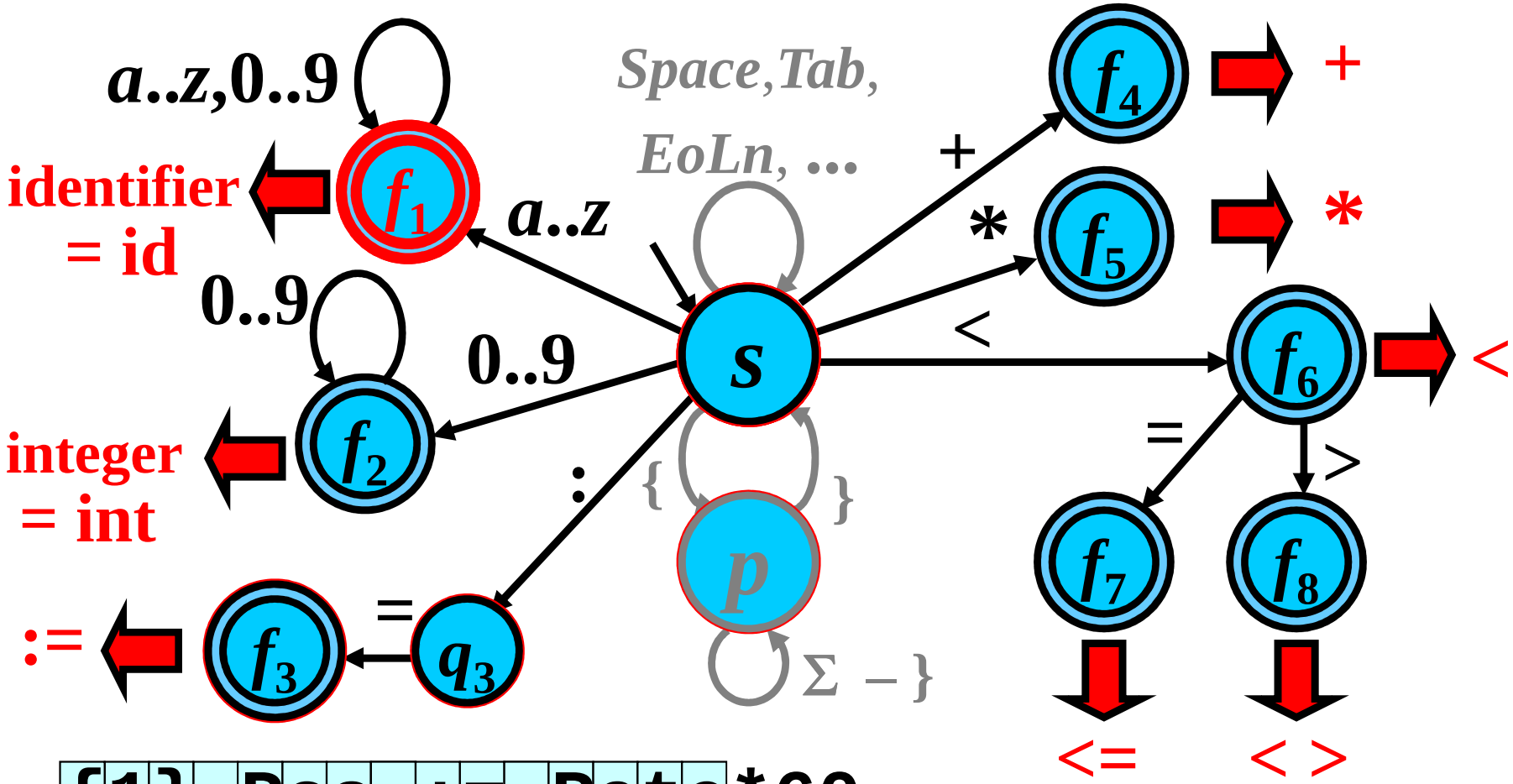



```
{1} Pos := Rate*60
```

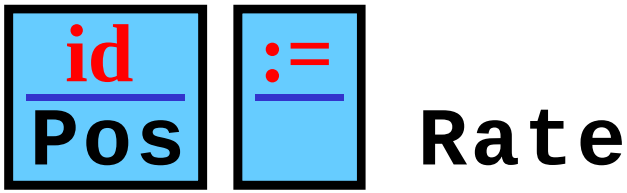


Rat

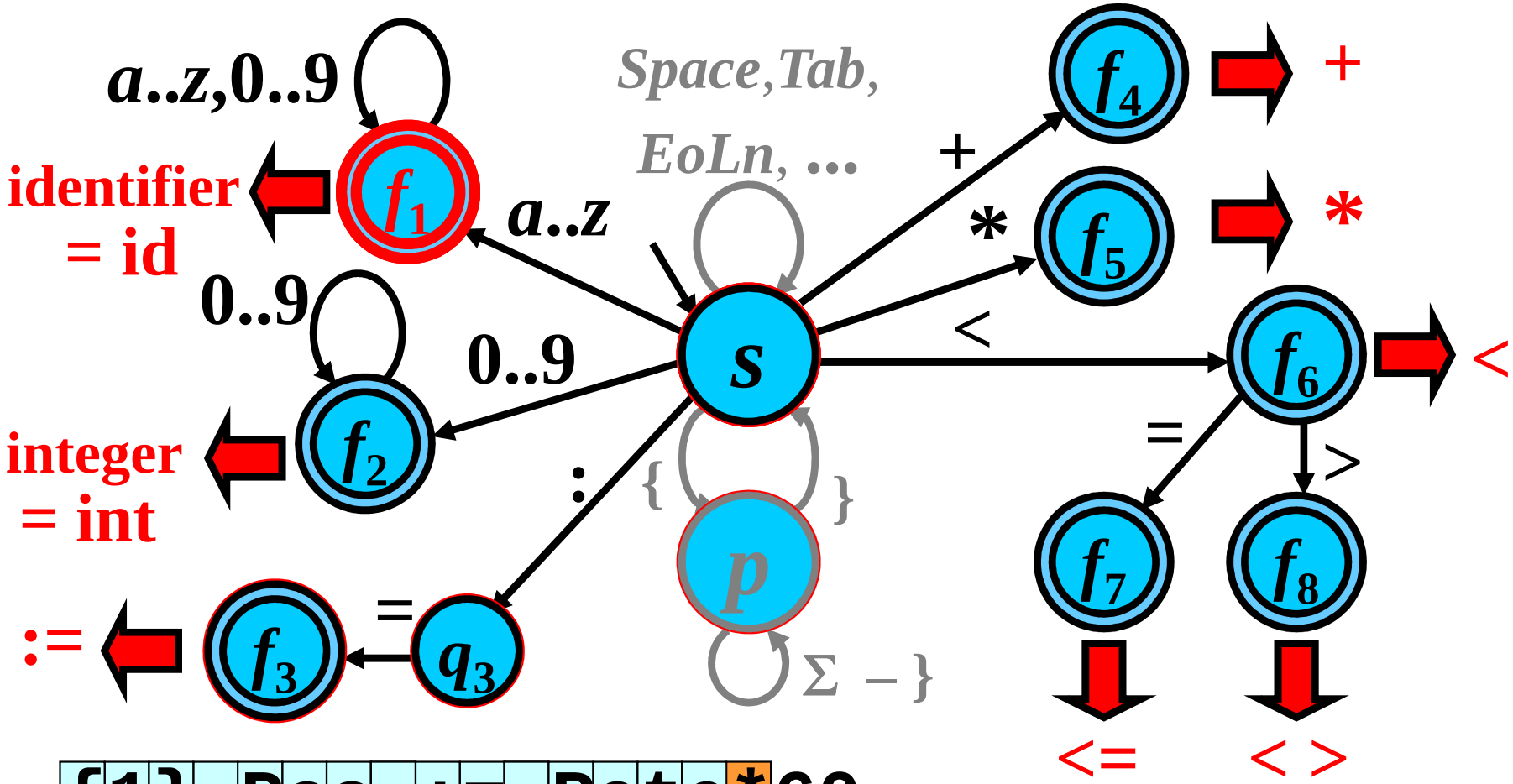
Type of Lexemes: Example



{1} Pos := Rate*60



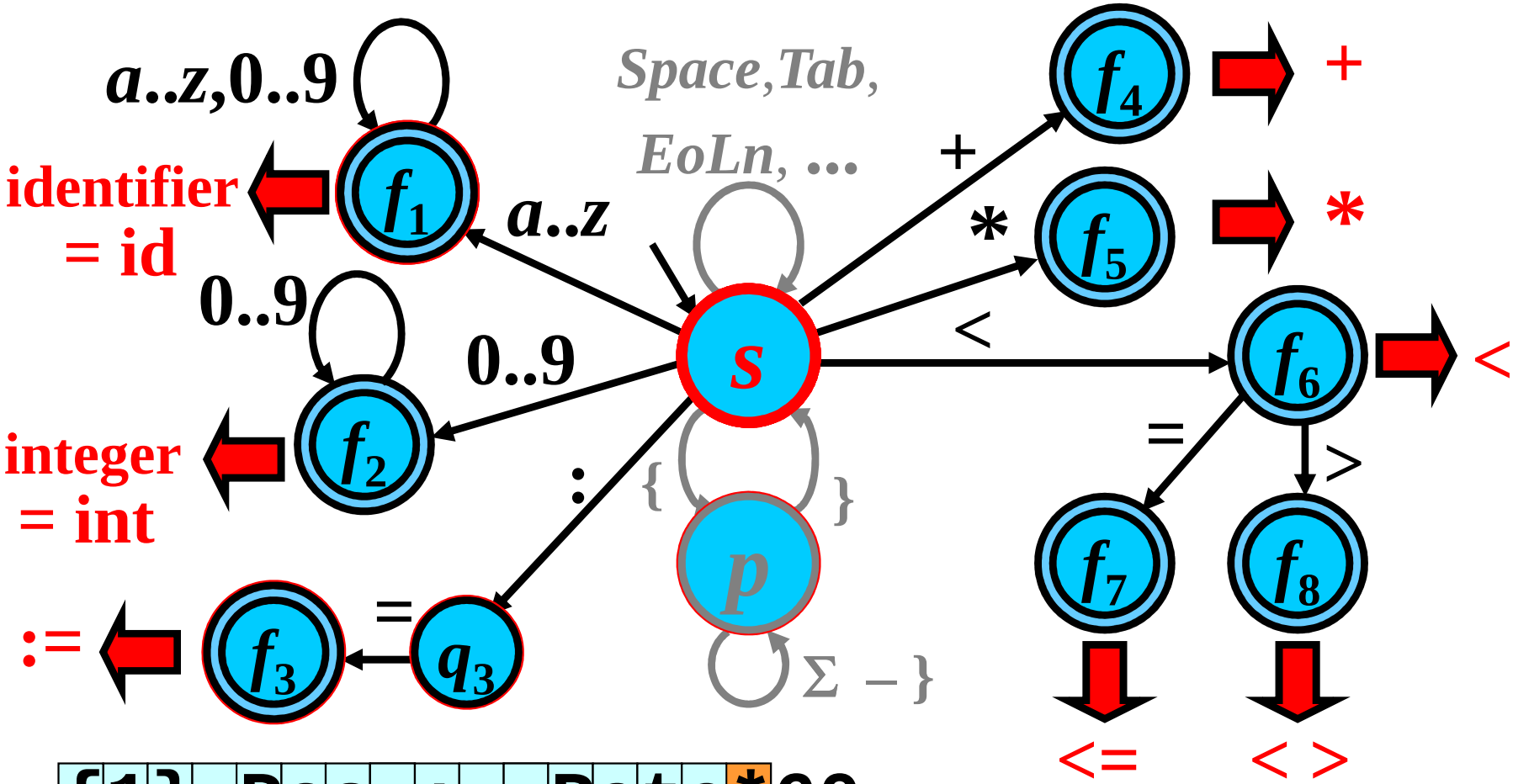
Type of Lexemes: Example



{1}		Pos		:=		Rate	*	60											

id	:=	id	No next configuration!
Pos		Rate	

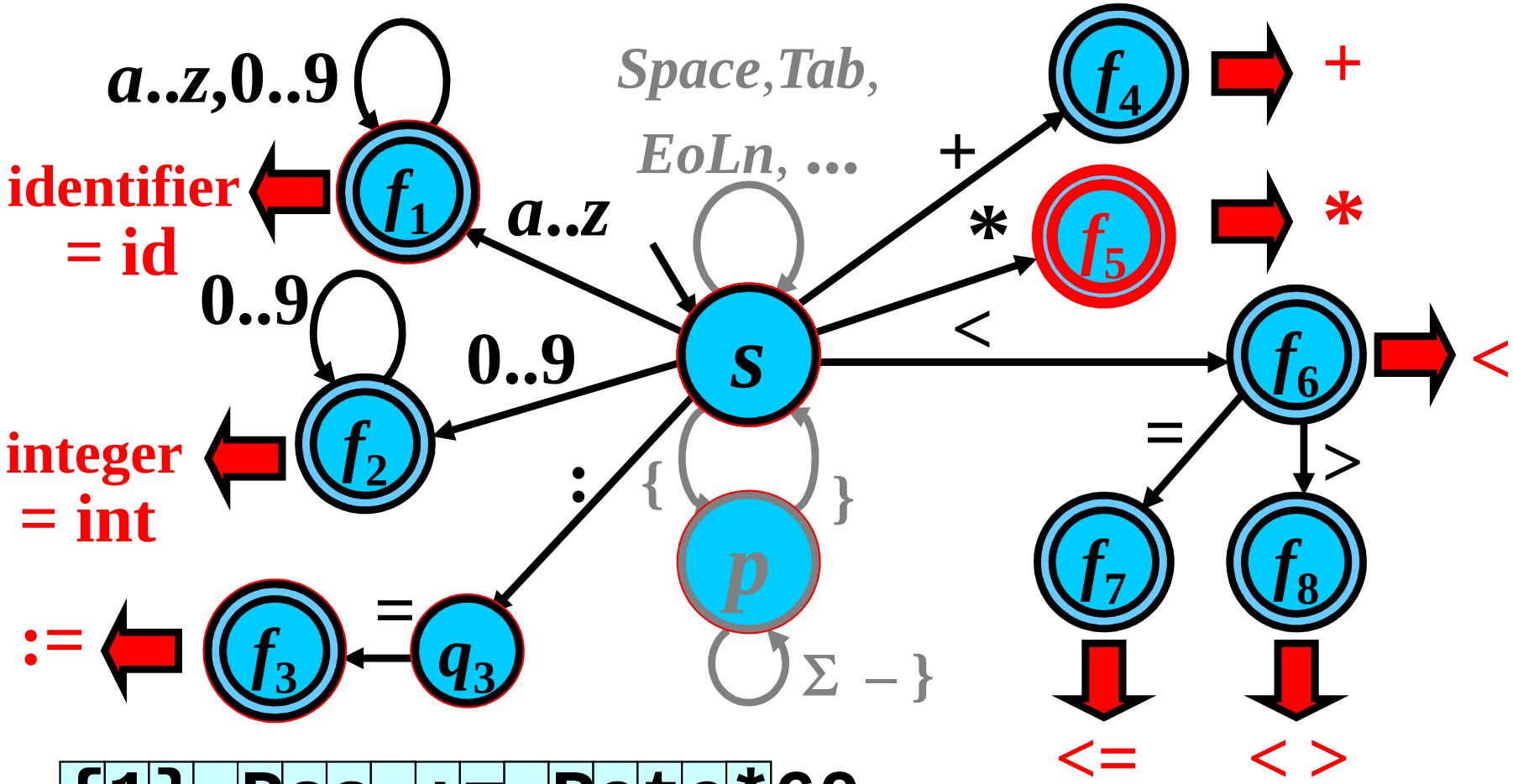
Type of Lexemes: Example



{1}		Pos		:=		Rate	*	60

id	:=	id
Pos		Rate

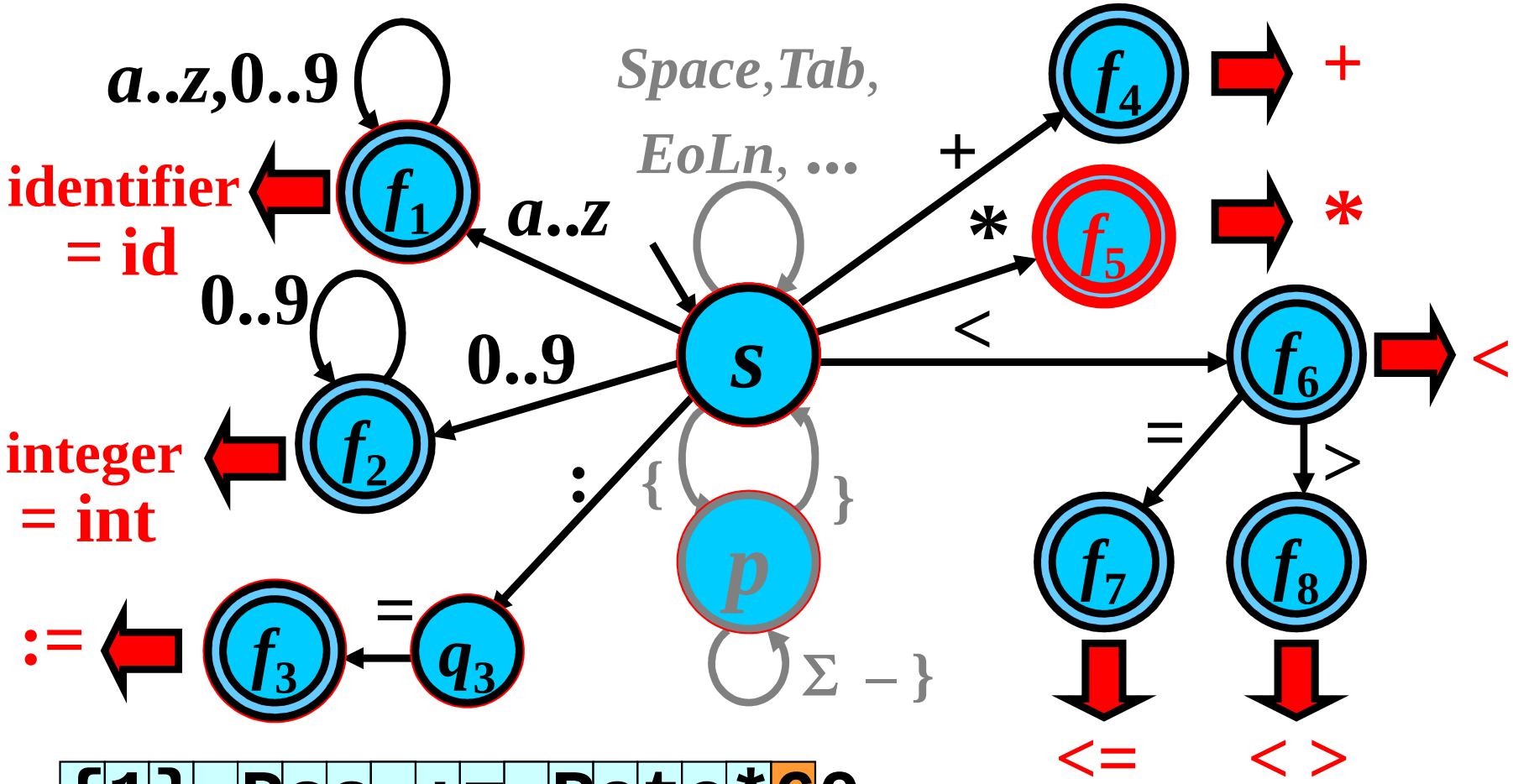
Type of Lexemes: Example



{1}		Pos		:=		Rate	*	60									

id	:=	id	*
Pos		Rate	

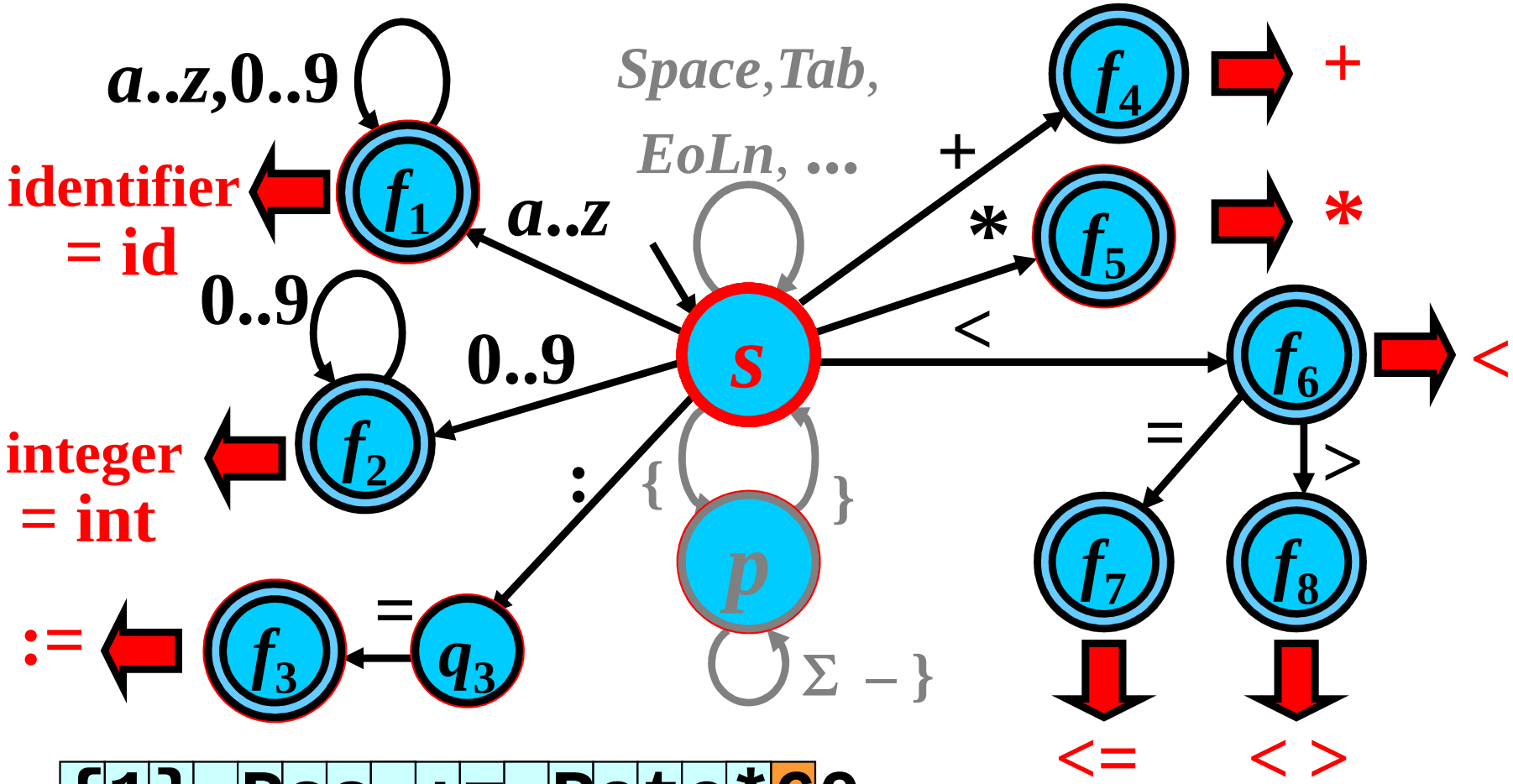
Type of Lexemes: Example



{1}		Pos		:=		Rate	*	60

id	:=	id	*	No next configuration!
Pos		Rate		

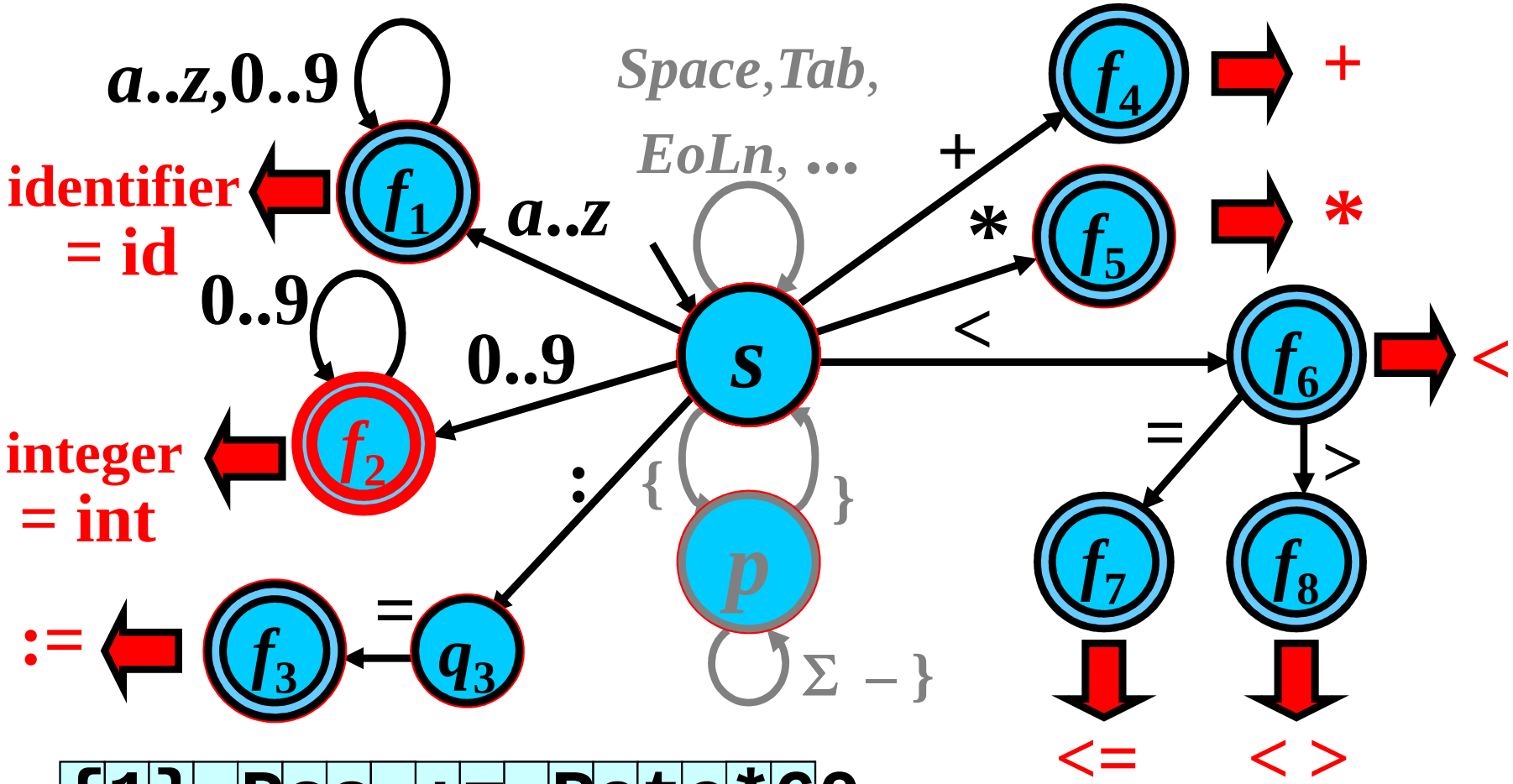
Type of Lexemes: Example



{1}		Pos		:=		Rate	*	60

id	:=	id	*
Pos		Rate	

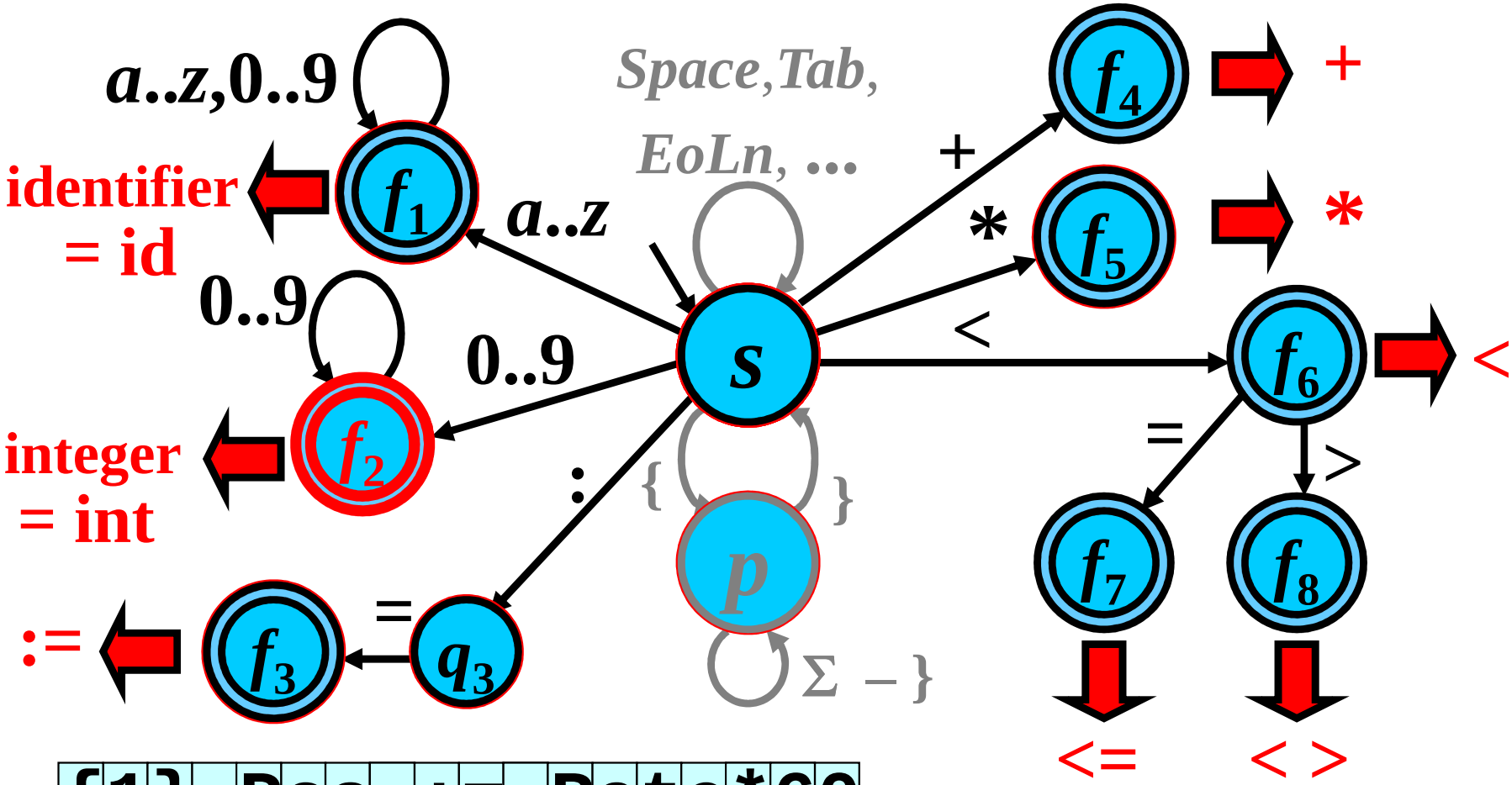
Type of Lexemes: Example



{1}		Pos		:=		Rate	*	60

id Pos	:=	id Rate	*	6

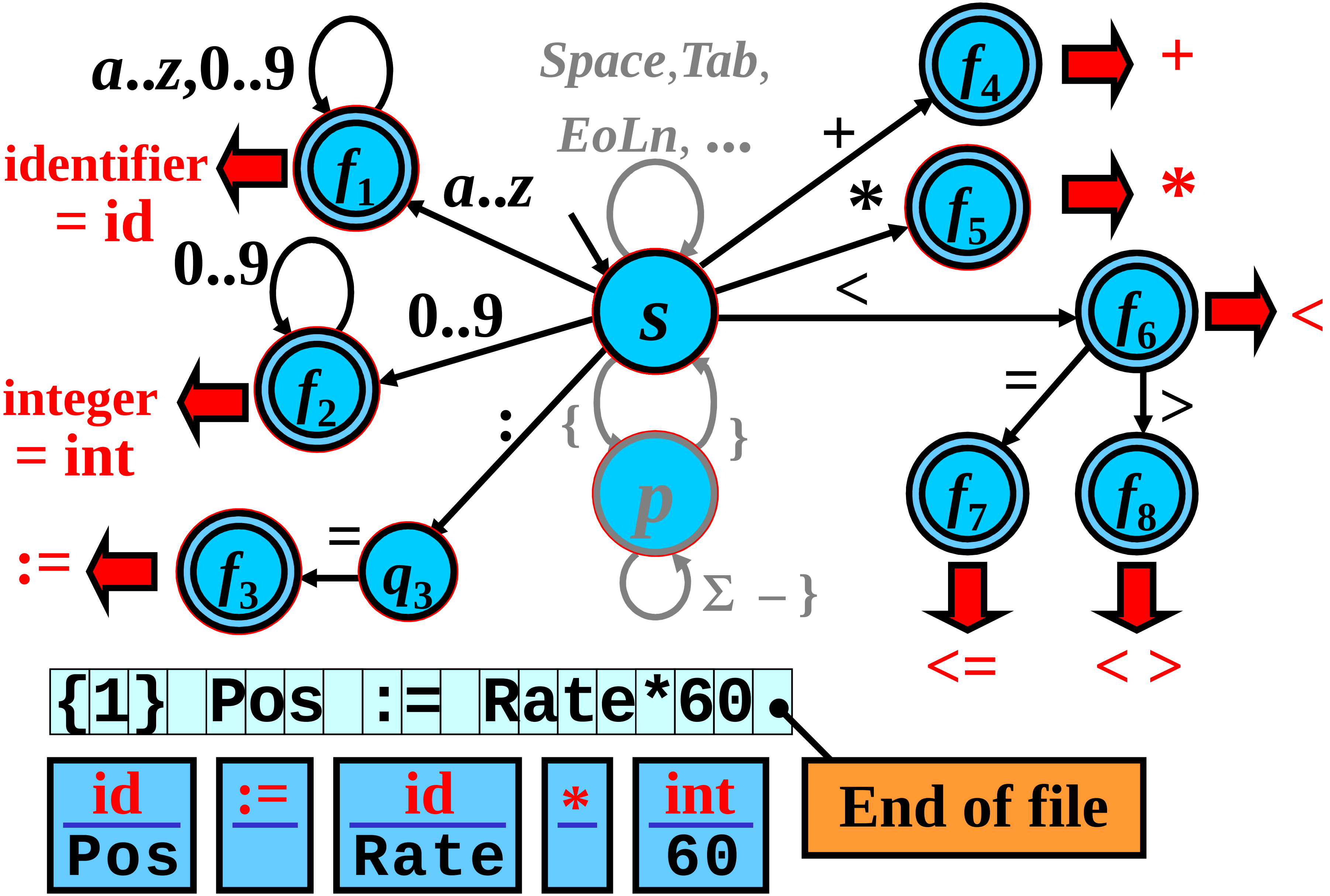
Type of Lexemes: Example



{1}		Pos		:=		Rate	*	60									

id	:=	id	*	
Pos		Rate		60

Type of Lexemes: Example



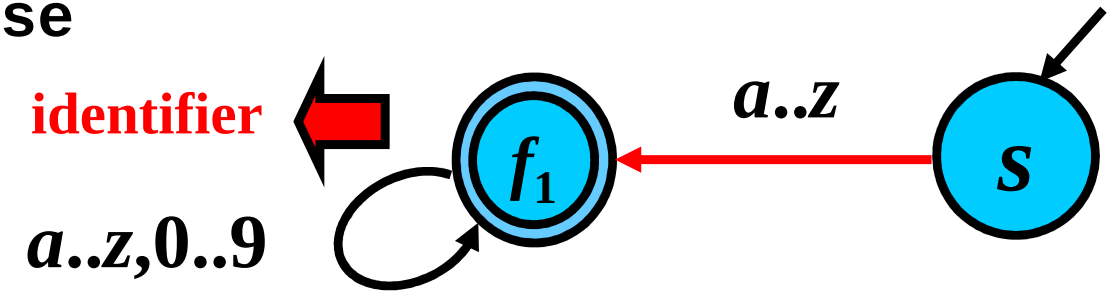
`{1} Pos := Rate*60`

id	:=	id	*	int
Pos		Rate		60

End of file

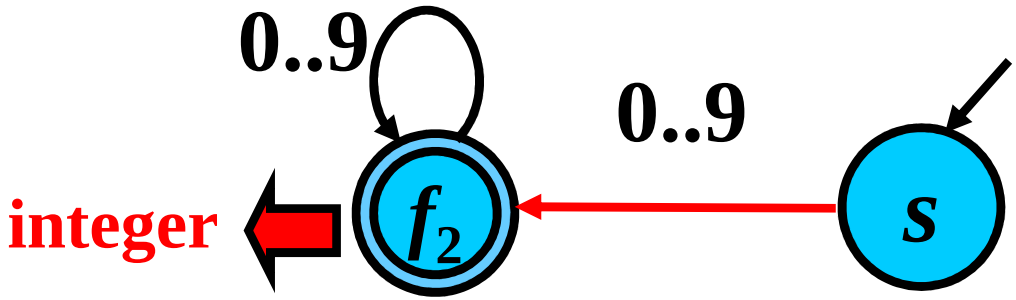
Implementation of DFA 1/10

```
procedure get_Next_Token(var TOKEN: ....);

...                                  {declaration, ...}

str     := '';                       {read string}
state   := S;                        {actual state}
repeat
  symbol = getchar();                {read next character}
  case state of
    s : begin                        {start state}
          if symbol in ['a'..'z'] then
          begin
            state:= f1;              {identifier}
            str   := symbol;
          end else
```



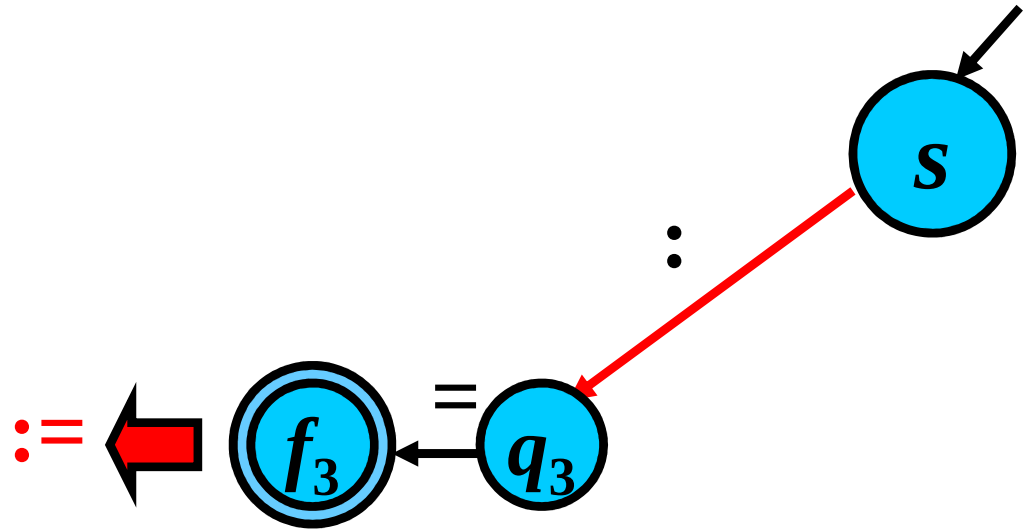
Implementation of DFA 2/10

```
case state of
  s : begin                          {start state}

        ...

        if symbol in ['0'..'9'] then
        begin
          state:= f2;        {integer}
          str   := symbol;
        end else
```



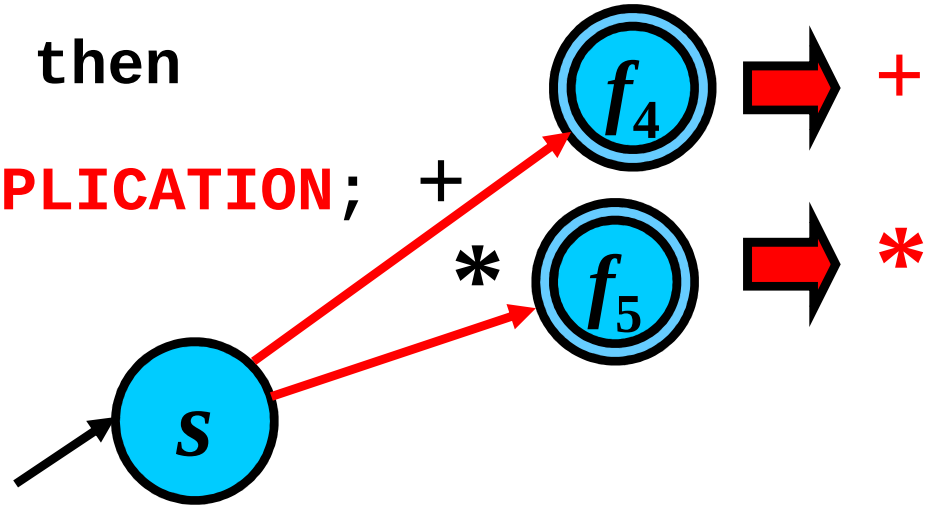
Implementation of DFA 3/10

```
case state of
  s : begin                          {start state}

        ...

        if symbol = ':' then
          state:= q3;                {assignment}
        else
```



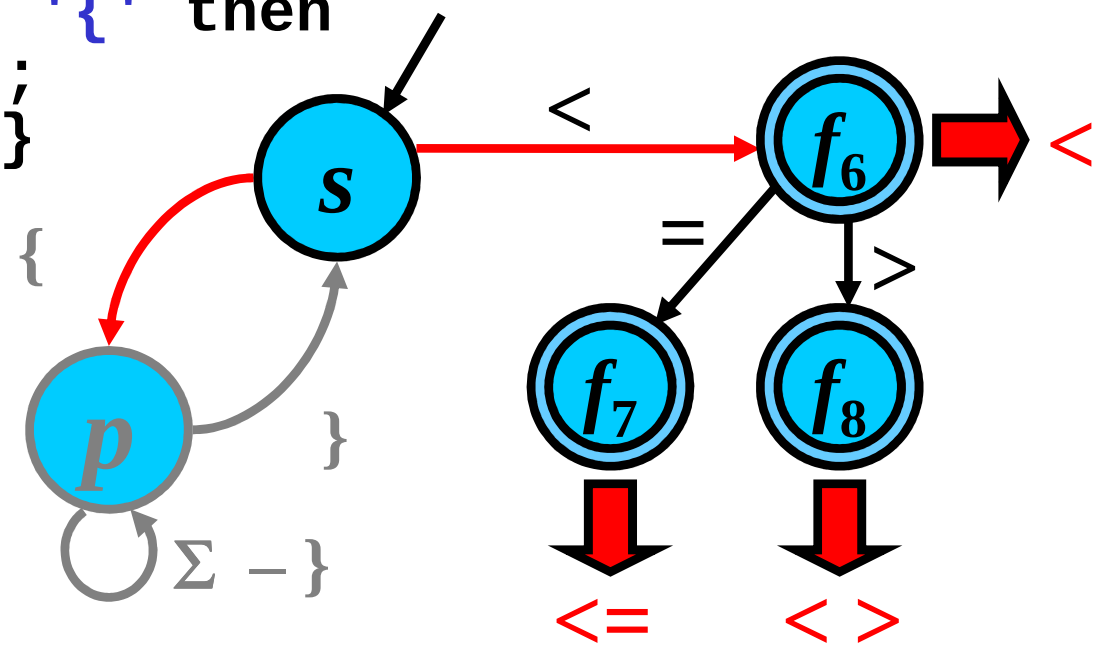
Implementation of DFA 4/10

```
case state of
  s : begin                          {start state}

        ...

        if symbol = '+' then
        begin
          TOKEN:= ADDITION;
          break;
        end else
        if symbol = '*' then
        begin
          TOKEN:= MULTIPLICATION;
          break;
        end else
```



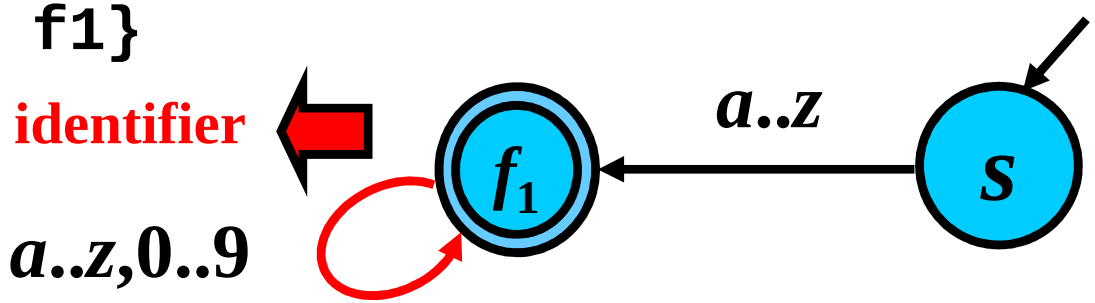
Implementation of DFA 5/10

```
case state of
  s : begin                          {start state}

        ...

        if symbol = '<' then
          state:= f6;
        else
        if symbol = '{' then
          state:= p;
      end; {state s}
```



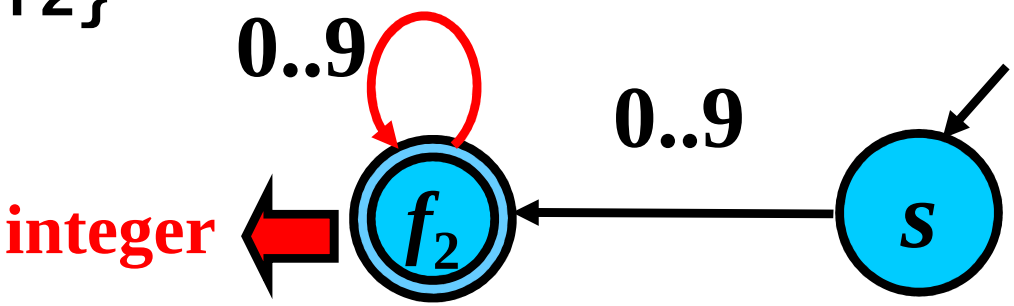
Implementation of DFA 6/10

```
case state of
  ...
  f1: begin                          {identifier}
        if symbol in ['a'..'z', '0'..'9'] then
          str  := str + symbol;
        else
        begin
          ungetchar(symbol);     {return symbol}
          if is_keyword(str) then     {keyword}
            TOKEN := get_keyword(str);
          else
            TOKEN := IDENTIFIER;
          break;
        end;
      end; {state f1}
```



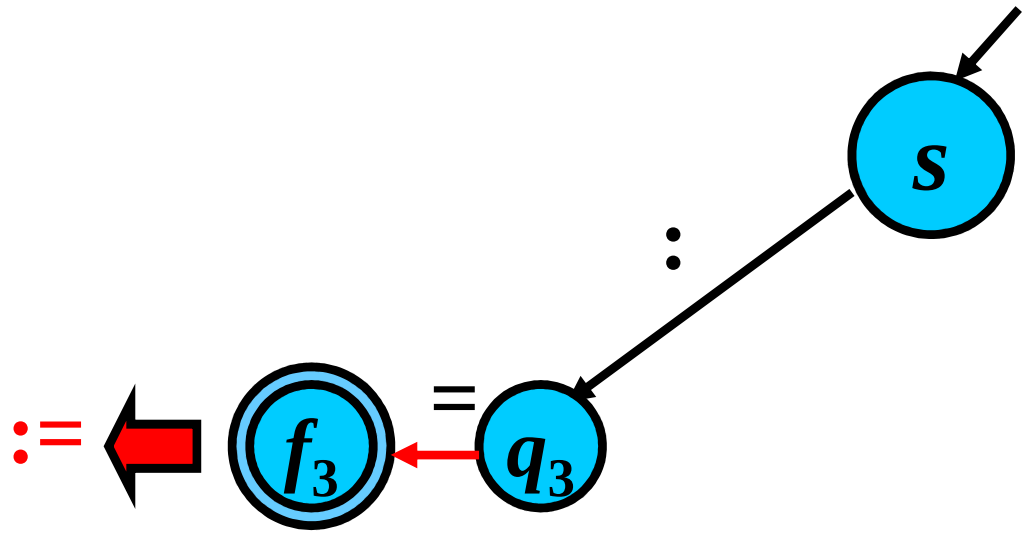
Implementation of DFA 7/10

```
case state of
  ...
  f2: begin                          {integer}
        if symbol in ['0'..'9'] then
          str  := str + symbol;
        else
        begin
          ungetchar(symbol);     {return symbol}
            TOKEN := INTEGER;
            {conversion value of str to integer}
          break;
        end;
      end; {state f2}
```



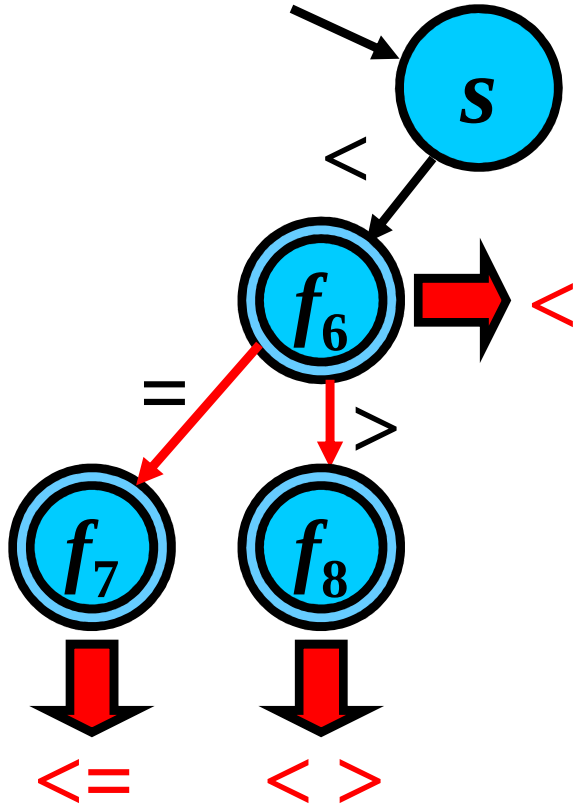
Implementation of DFA 8/10

```
case state of
  ...
  q3: begin                          {assignment}
        if symbol = '=' then
        begin
          TOKEN := ASSIGNMENT;
          break;
      end; {state q3}
```



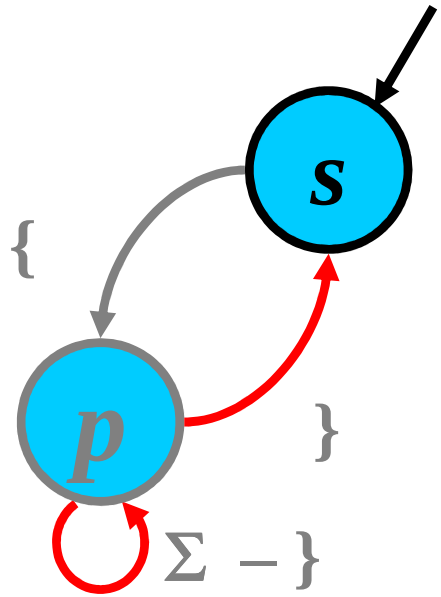
Implementation of DFA 9/10

```
case state of
  ...
  f6: begin
        if symbol = '=' then
        begin
          TOKEN := LEQ;    {<=}
          break;
        end else
        if symbol = '>' then
        begin
          TOKEN := NEQ;    {<>}
          break;
        end else
          ungetchar(symbol); {return symbol}
          TOKEN := LTN;    {<}
          break;
        end;
      end; {state f6}
```



Implementation of DFA 10/10

```
  case state of
    ...
    p : begin                          {comment}
          if symbol = '}' then
            state := s;                {start state}
        end; {state p}

until EOF;

...

end;
```



Tokens in Practice

- tokens represent every SP lexeme in a uniform way
- in general, their form is

[type, attribute]

---

1) Token **attributes** may vary

[id, ☛ Pos], **[int, 60]**, **[*, ␣]**

pointer **integer** **nothing**

---

2) The same form of tokens

[1, 2], **[2, 3]**, **[3, 1]**

NOTE: In practice, we often use tokens whose attributes vary.

The Same Form of Tokens

[id, ☛Pos] ⇩ [,]

[int, 60] ⇩ [,]

[*,] ⇩ [,]

1

Table of id:
1: Id1

2

Table of int:
1: 25
2: 10000

3

Table of op:
1: *
2: /
3: +
4: –

The Same Form of Tokens

[id, ☛Pos] → [1,]

[int, 60] → [,]

[*,] → [,]

1 Table of id:

1: Id1

2 Table of int:

1: 25
2: 10000

3 Table of op:

1: *
2: /
3: +
4: –

The Same Form of Tokens

[id, ☛Pos] [int, 60] [*,]

[1 , 2] [,] [,]

1

Table of id:
1: Id1
2: Pos

2

Table of int:
1: 25
2: 10000

3

Table of op:
1: *
2: /
3: +
4: –

The Same Form of Tokens

[id, ☛Pos] → [1, 2]

[int, 60] → [2,]

[*,] → [,]

1 Table of id:

1: Id1
2: Pos

2 Table of int:

1: 25
2: 10000

3 Table of op:

1: *
2: /
3: +
4: –

The Same Form of Tokens

[id, ☛Pos] → [1, 2]

[int, 60] → [2, 3]

[*,] → [,]

1 Table of id:

1: Id1
2: Pos

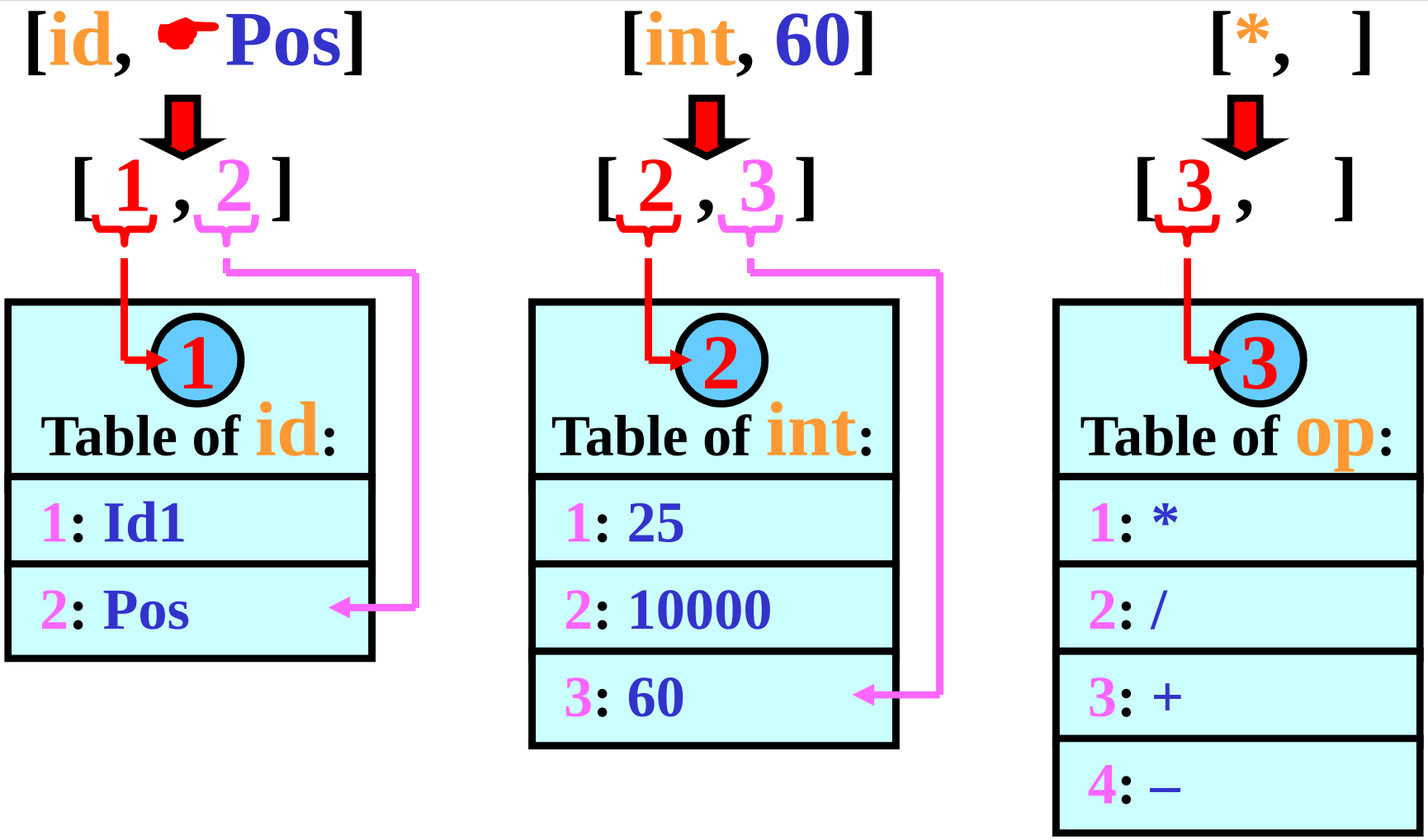
2 Table of int:

1: 25
2: 10000
3: 60

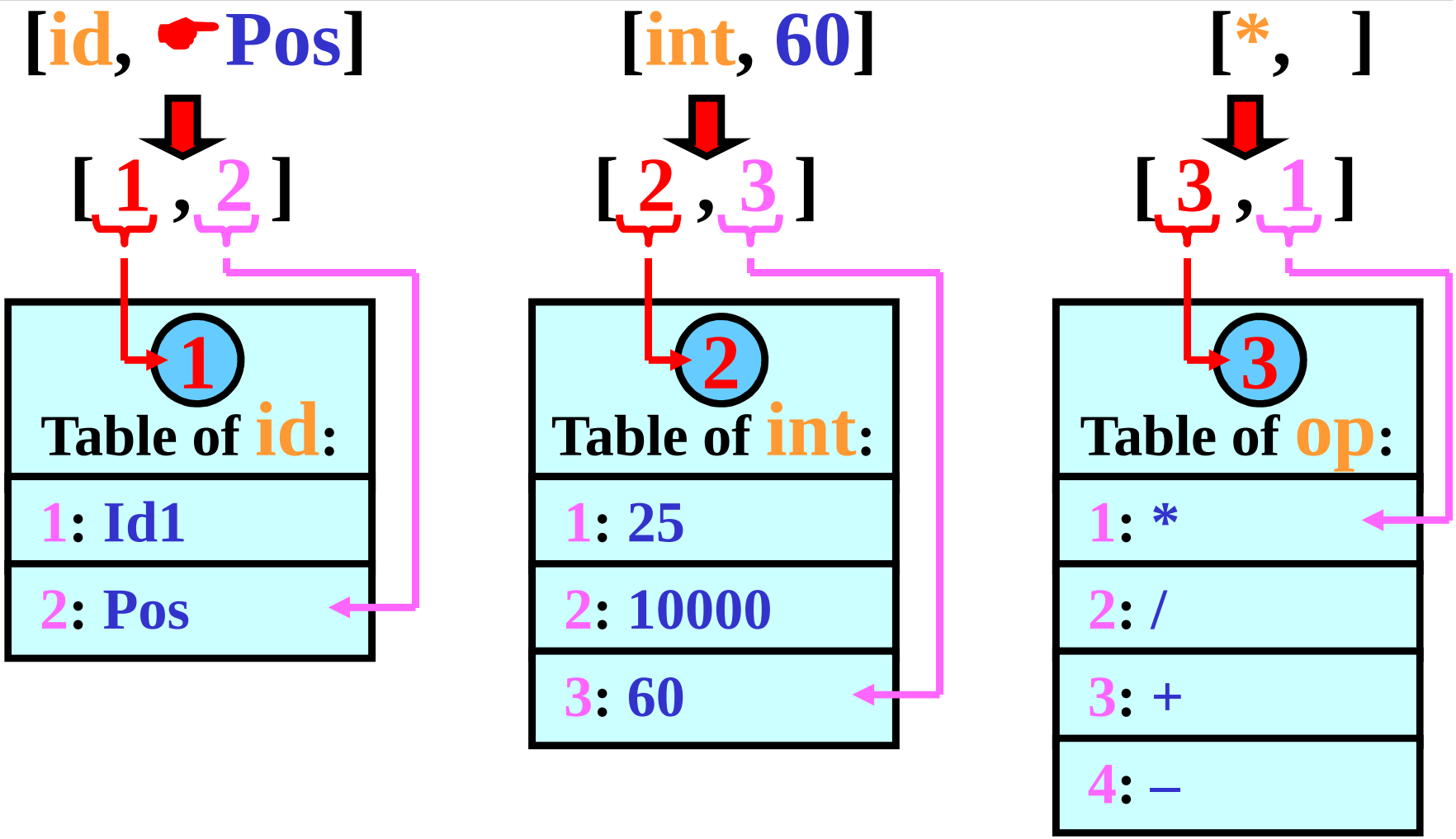
3 Table of op:

1: *
2: /
3: +
4: –

The Same Form of Tokens



The Same Form of Tokens



The Same Form of Tokens

[id, ☛Pos] → [1, 2]

[int, 60] → [2, 3]

[*,] → [3, 1]

1 — Table of id:

1	Id1
2	Pos

2 — Table of int:

1	25
2	10000
3	60

3 — Table of op:

1	*
2	/
3	+
4	–

Uniform form of tokens: [1, 2]; [2, 3]; [3, 1]

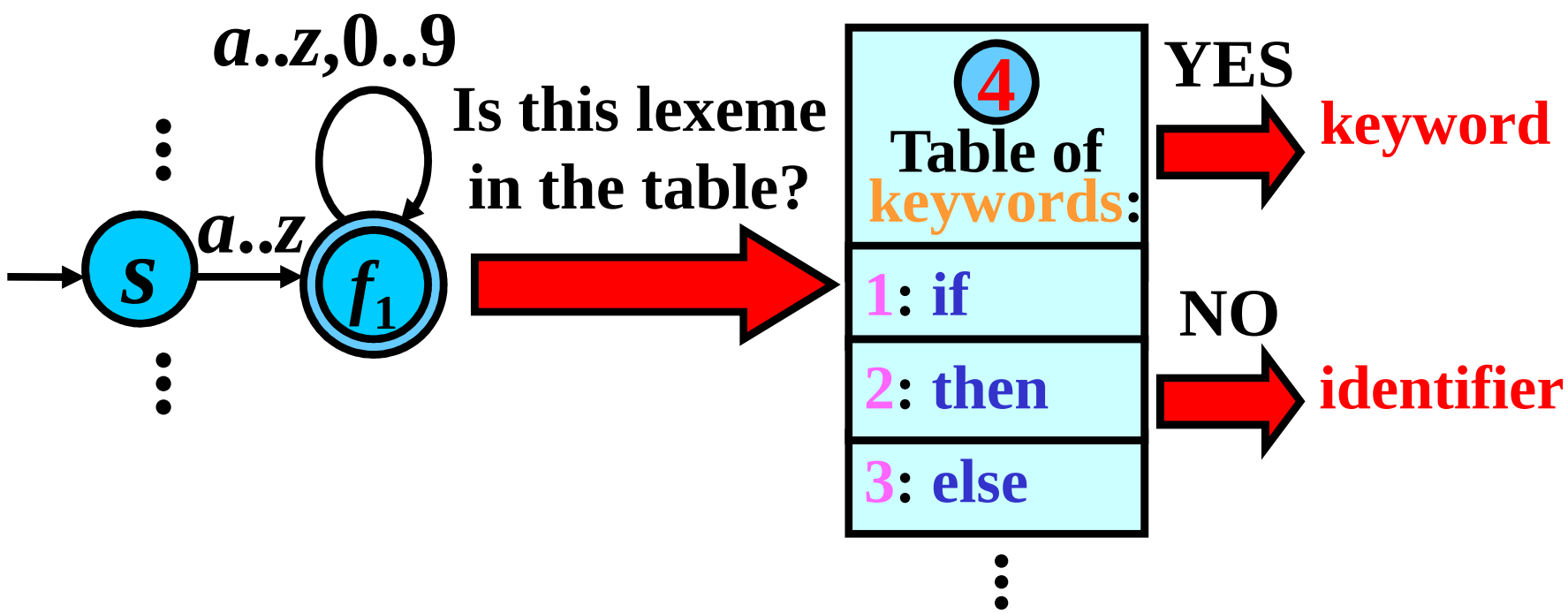
Homogenous structure

Identifiers × Keywords

Question: How to distinguish identifiers from keywords?

`if` ⇒ **keyword** × `ifj` ⇒ **identifier**

Answer: By a table of keywords.
(Tokens have the same form)



Symbol Table (Identifier Table)

Practical problem:

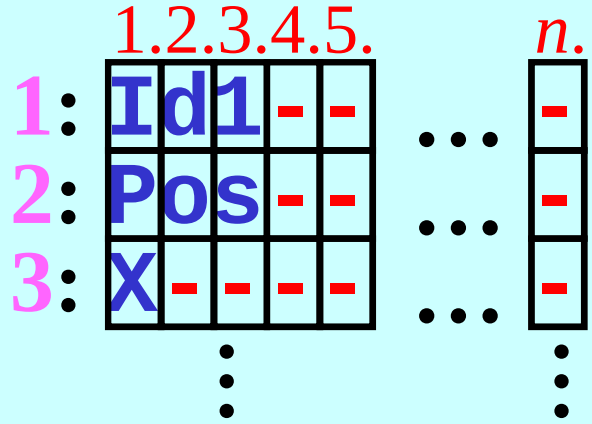
1) Short identifiers:

- Empty spaces in memory (-)

2) Long identifiers:

- Length(Id) ≤ ***n***

Symbol table:



Symbol Table (Identifier Table)

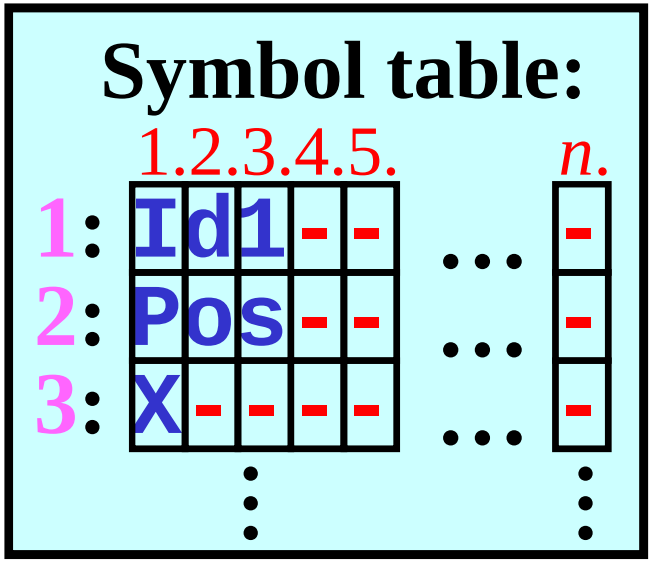
Practical problem:

1) Short identifiers:

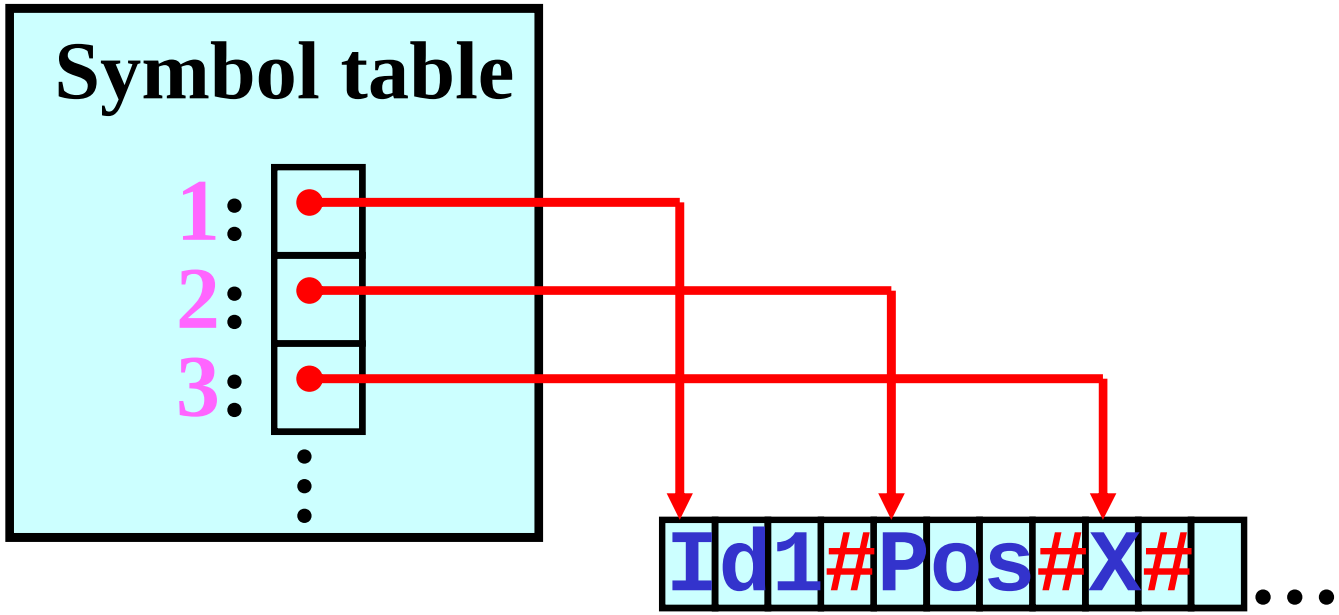
- Empty spaces in memory (-)

2) Long identifiers:

- Length(Id) ≤ ***n***



Solution:



Symbol Table: Structure

- We need many pieces of information about identifiers in ST:
  - **Variable:** name, type, length, …
  - **Constant:** type and value of constant
  - **Procedure:** the number and type of parameters

⋮

Symbol Table: Structure

- We need many pieces of information about identifiers in ST:
  - **Variable:** name, type, length, …
  - **Constant:** type and value of constant
  - **Procedure:** the number and type of parameters

⋮

Final structure of the symbol table:

Symbol table		
	Name	**Info**
1:	Id1	**Variable**; **Type:** integer
2:	Pi	**Constant**; **Type:** real, **Value:** 3.1415927

Scope of Identifiers

- **Problem:**

```
Program P1;
```

Symbol table

Scope of Identifiers

- **Problem:**

```
Program P1;
var x, y: integer;
```

Symbol table		
1:	x	…
2:	y	…

Scope of Identifiers

- **Problem:**

```
Program P1;
var x, y: integer;

Procedure Proc1;
```

Symbol table		
1:	x	…
2:	y	…
3:	Proc1	…

Scope of Identifiers

- **Problem:**

```
Program P1;
var x, y: integer;

Procedure Proc1;
var x, y: integer;
```

Symbol table		
1:	x	…
2:	y	…
3:	Proc1	…
4:	x	…
5:	y	…

Scope of Identifiers

- **Problem:**

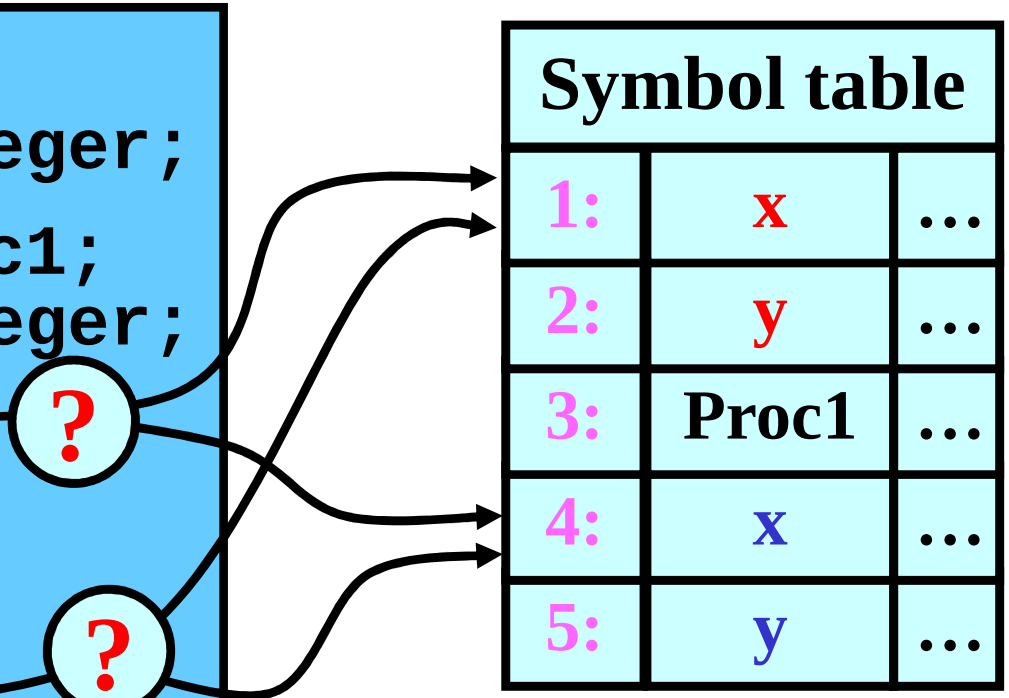
```
Program P1;
var x, y: integer;

Procedure Proc1;
var x, y: integer;
begin
 … x := 1; …
end;
```

Symbol table		
1:	x	…
2:	y	…
3:	Proc1	…
4:	x	…
5:	y	…

Scope of Identifiers

- **Problem:**

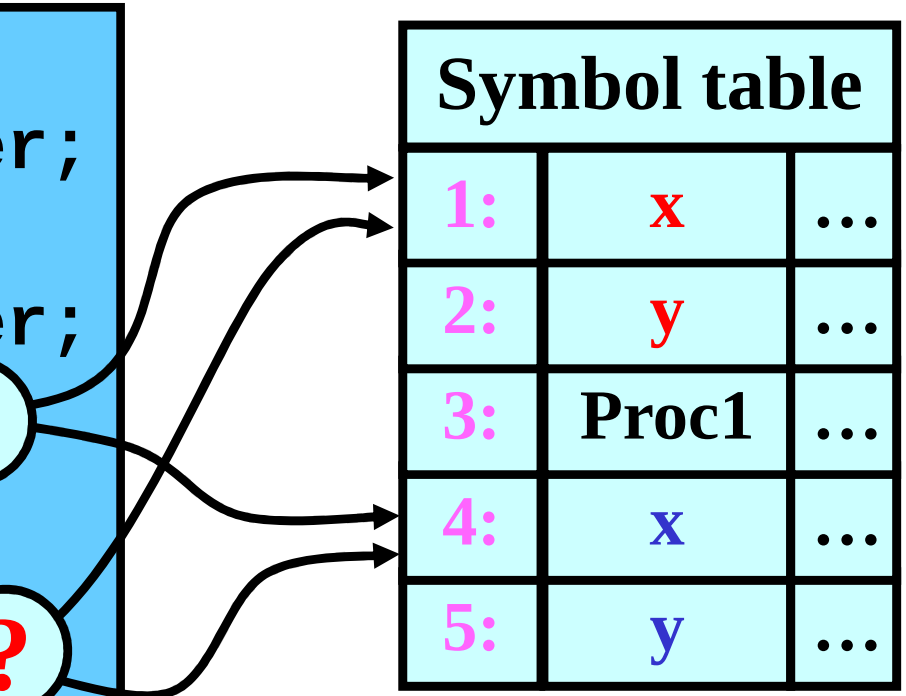
```
Program P1;
var x, y: integer;

Procedure Proc1;
var x, y: integer;
begin
 … x := 1; …
end;

begin
 … x := 2; …
end.
```



Scope of Identifiers

- **Problem:**

```
Program P1;
var x, y: integer;

Procedure Proc1;
var x, y: integer;
begin
 … x := 1; …
end;

begin
 … x := 2; …
end.
```



- **Solution: Scope Rules** (Stack structure of ST)

Scope Rules

Symbol Table=
ST-stack:

Auxiliary Table =
AT-stack:

Scope Rules

Main Block (B0)

Symbol Table= ST-stack:

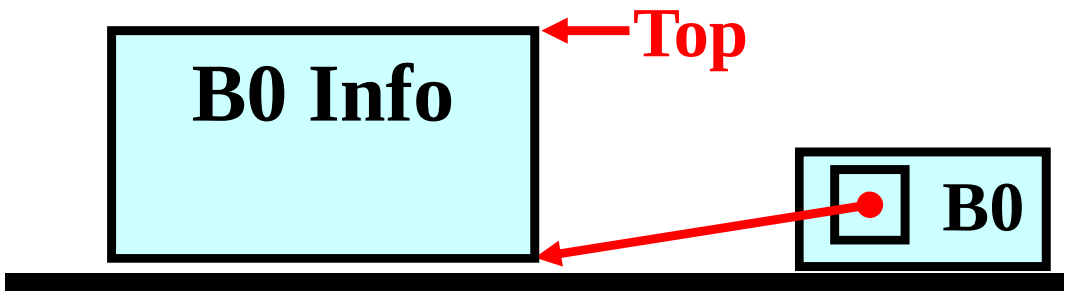
Auxiliary Table = AT-stack:

Scope Rules

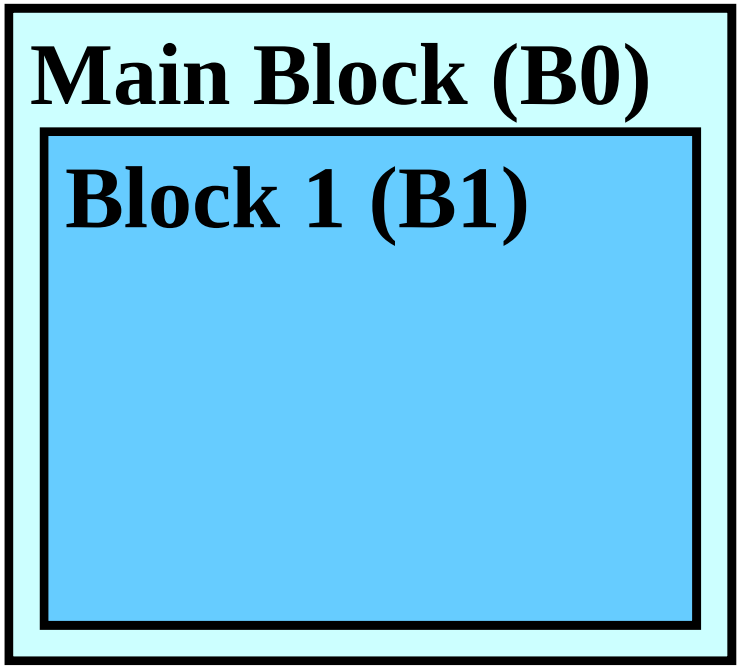
Main Block (B0)

Symbol Table= ST-stack:

Auxiliary Table = AT-stack:

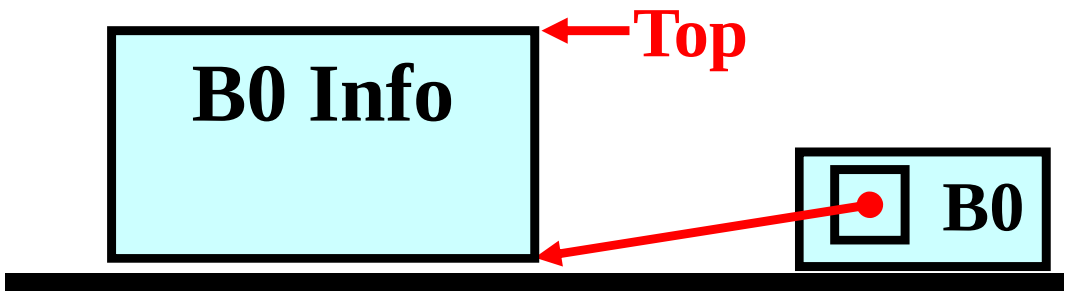


Scope Rules

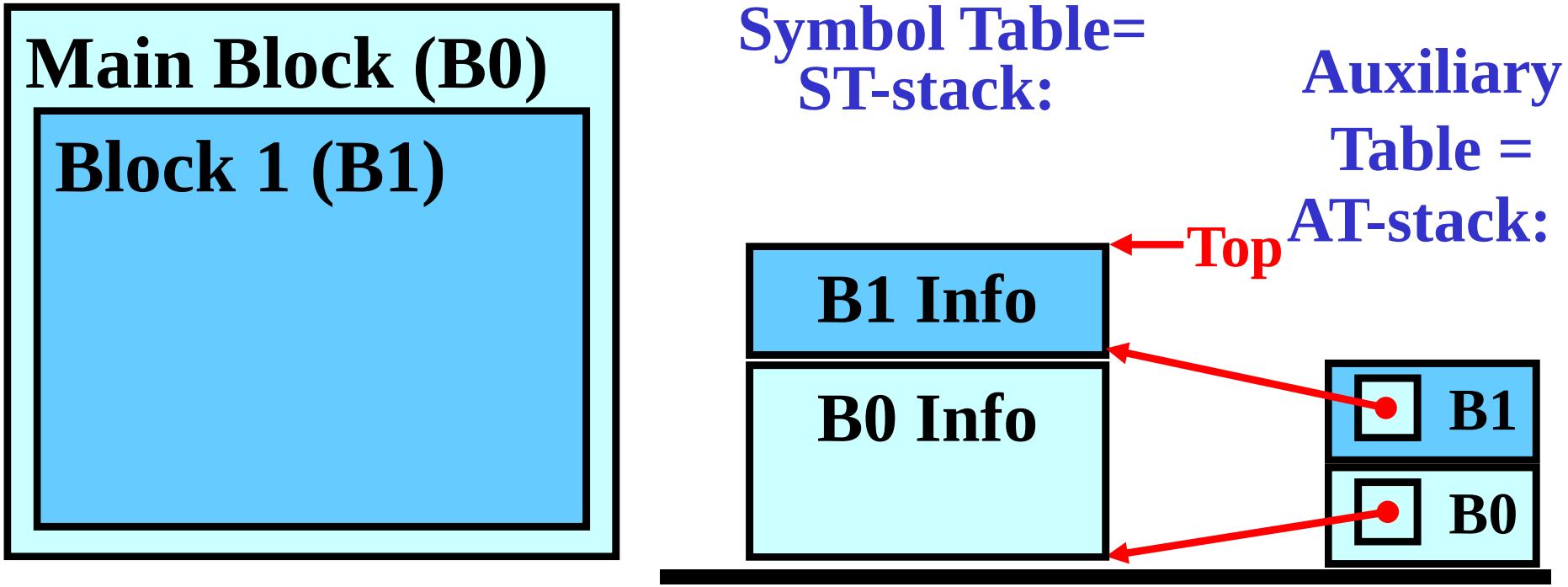


Symbol Table= ST-stack:

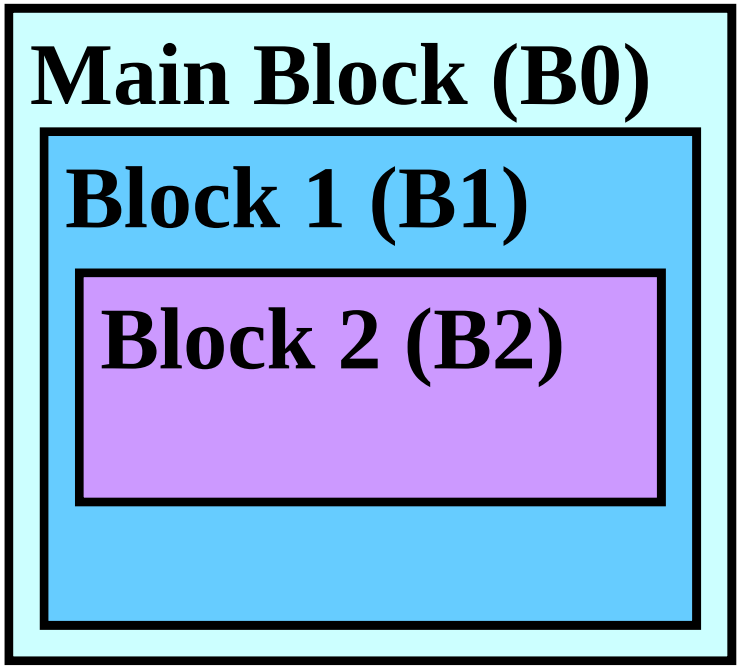
Auxiliary Table = AT-stack:

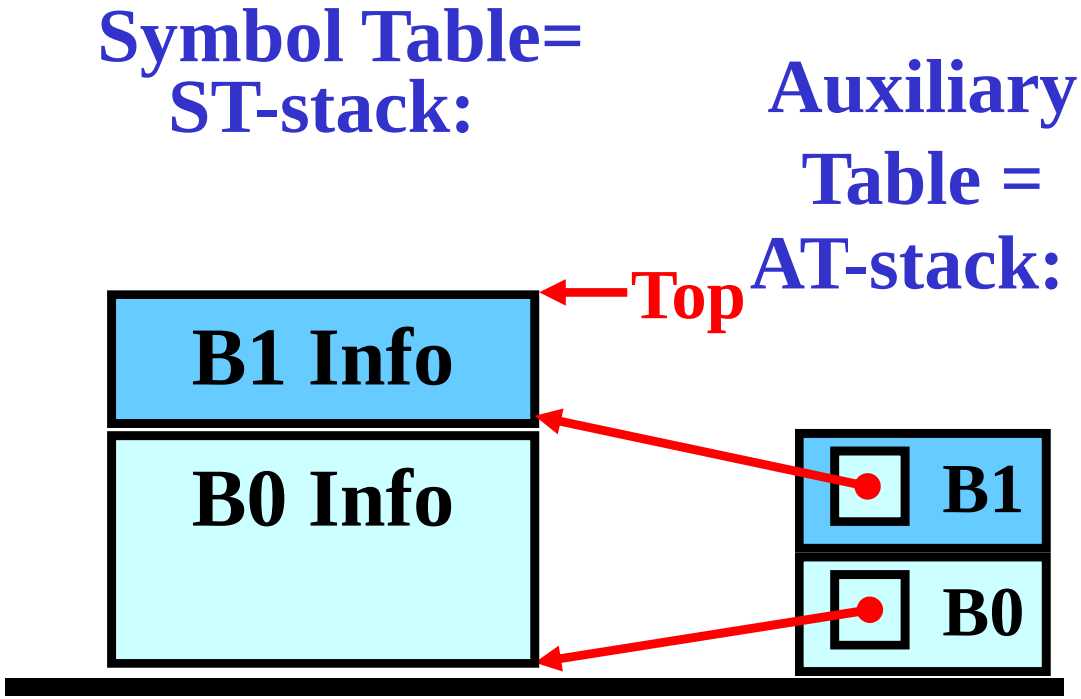


Scope Rules

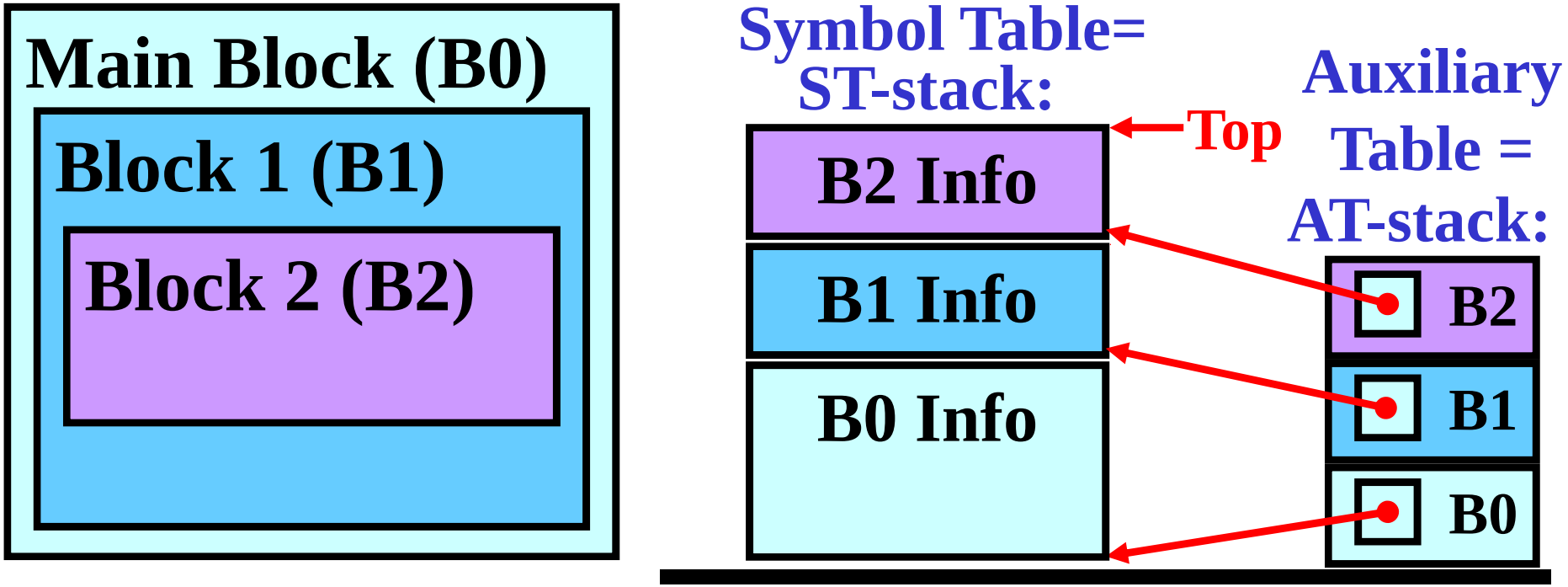

Main Block (B0)
Block 1 (B1)
Symbol Table=
ST-stack:
Auxiliary
Table =
AT-stack:
Top
B1 Info
B0 Info
B1
B0

Scope Rules

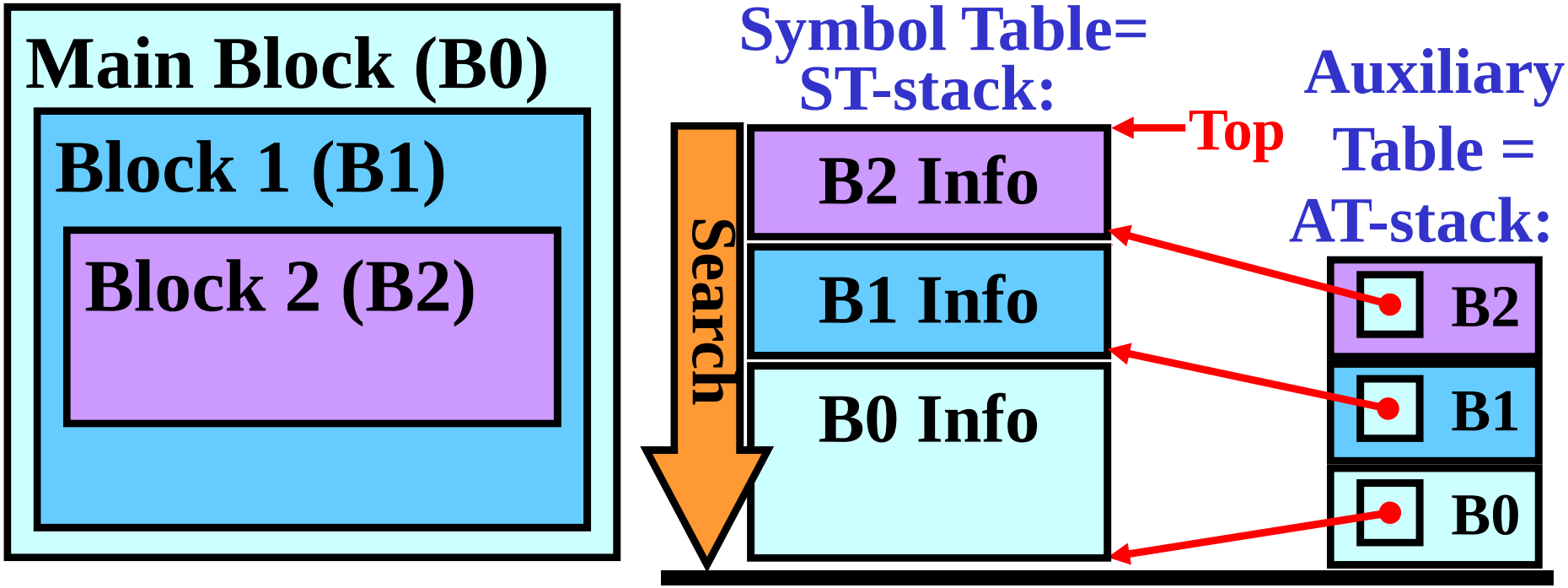

Main Block (B0)
Block 1 (B1)
Block 2 (B2)


Symbol Table=
ST-stack:
Auxiliary
Table =
AT-stack:
Top
B1 Info
B0 Info
B1
B0

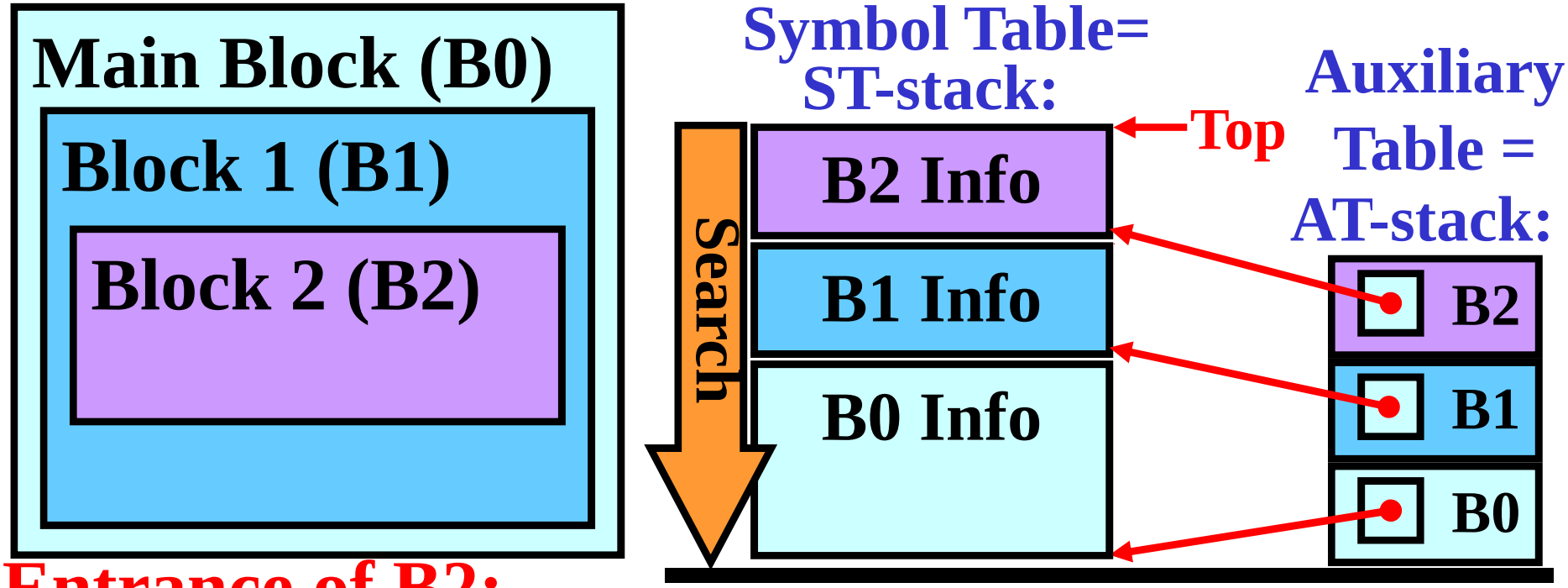
Scope Rules


Main Block (B0)
Block 1 (B1)
Block 2 (B2)
Symbol Table=
ST-stack:
Top
B2 Info
B1 Info
B0 Info
Auxiliary
Table =
AT-stack:
B2
B1
B0

Scope Rules


Main Block (B0)
Block 1 (B1)
Block 2 (B2)
Symbol Table=
ST-stack:
Search
B2 Info
B1 Info
B0 Info
Top
Auxiliary
Table =
AT-stack:
B2
B1
B0

Scope Rules



Entrance of B2:

- Push a pointer to the ST-stack top onto AT-stack

Exit from B2:

- The top of B1 Info becomes the ST-stack top
- Remove the B2 pointer from the AT-stack top

Search in ST:

- from the top towards the bottom

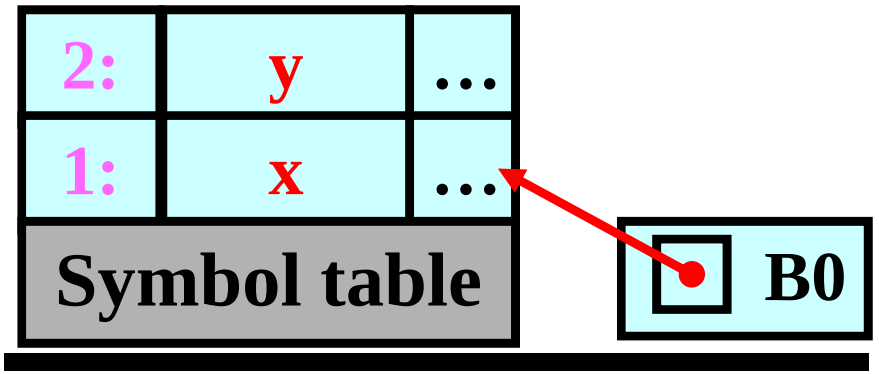
Scope Rules: Example

```
Program P1;
```

Symbol table

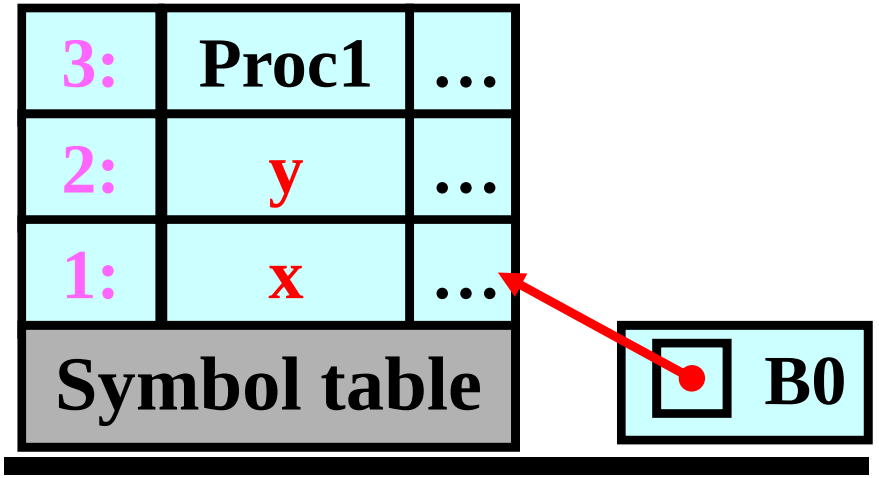
Scope Rules: Example

```
Program P1;
var x, y: integer;
```



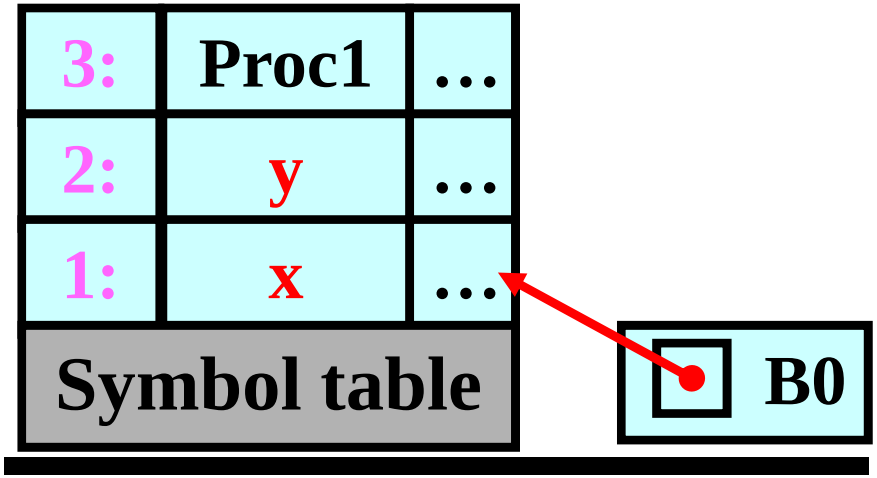
Scope Rules: Example

```
Program P1;
var x, y: integer;

Procedure Proc1;
```



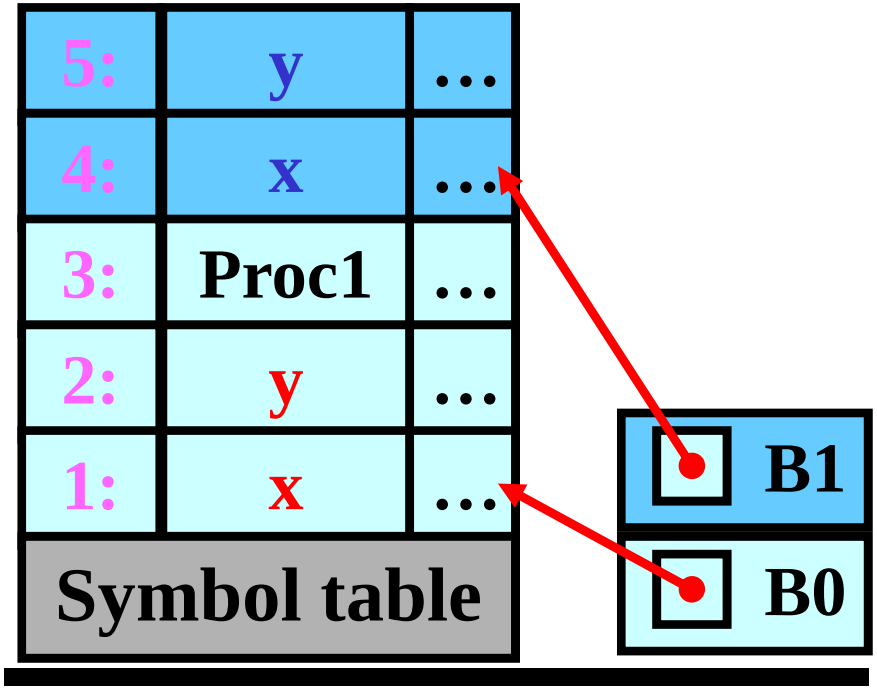
Scope Rules: Example

```
Program P1;
var x, y: integer;

Procedure Proc1;
var x, y: integer;
```



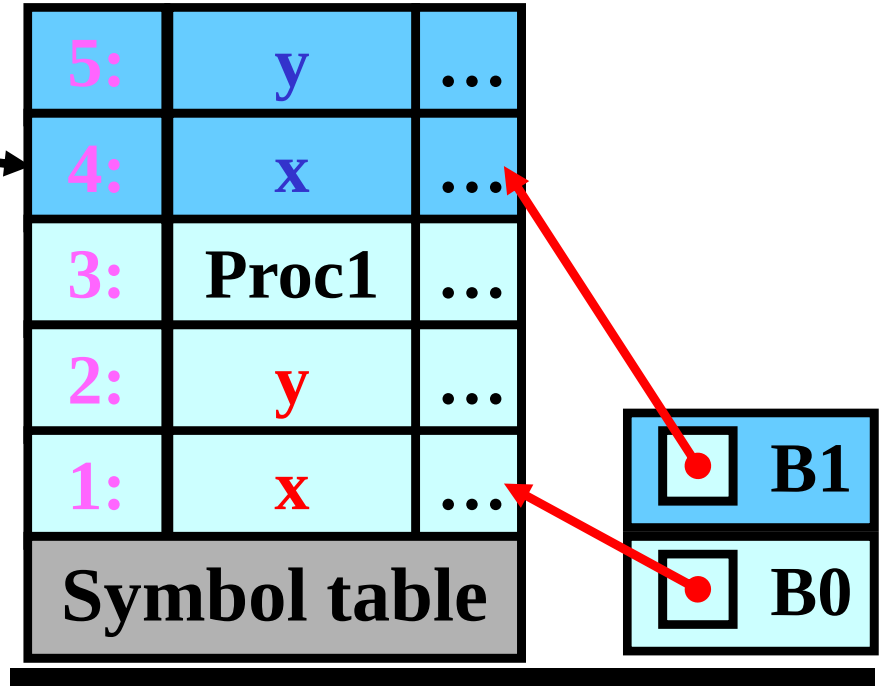
Scope Rules: Example

```
Program P1;
var x, y: integer;

Procedure Proc1;
var x, y: integer;
```



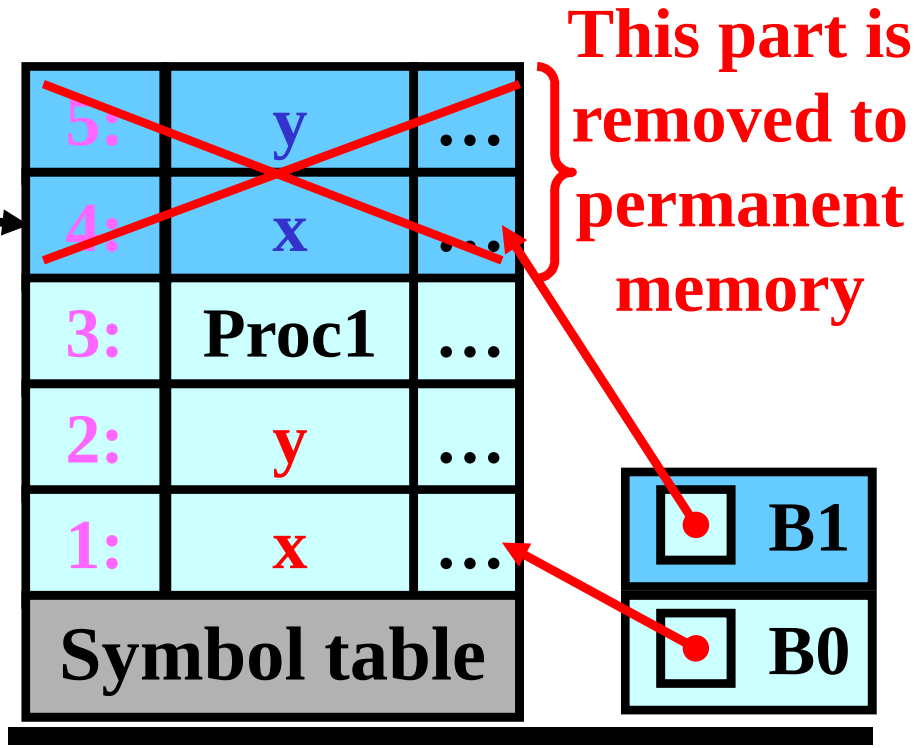
Scope Rules: Example

```
Program P1;
var x, y: integer;

Procedure Proc1;
var x, y: integer;
begin
   … x := 1; …
end;
```



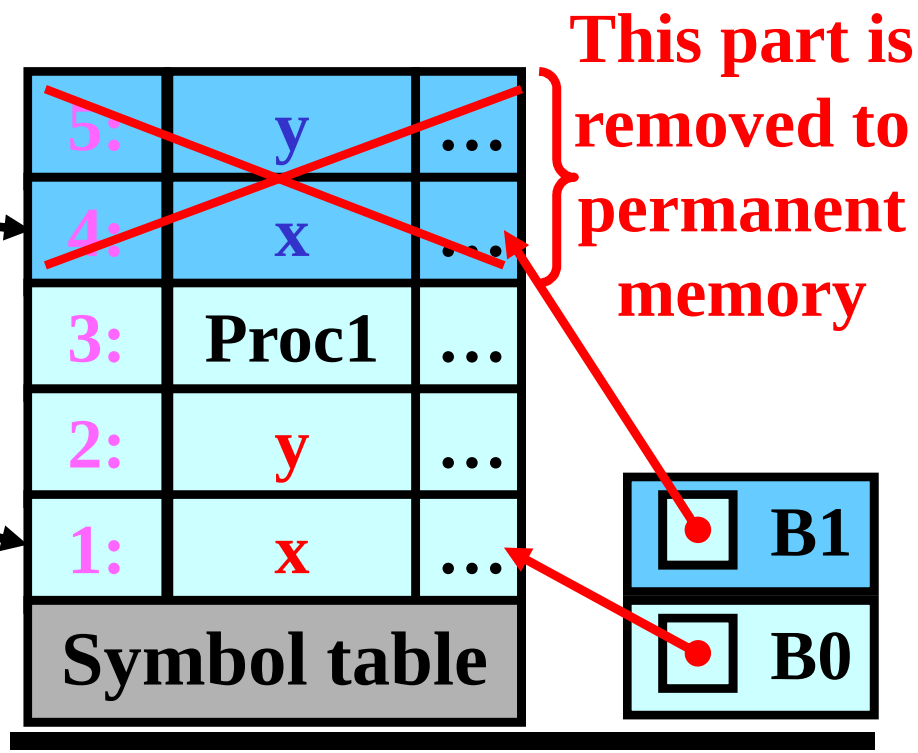
Scope Rules: Example

```
Program P1;
var x, y: integer;

Procedure Proc1;
var x, y: integer;
begin
   … x := 1; …
end;

begin
   … x := 2; …
end.
```



Scope Rules: Example

```
Program P1;
var x, y: integer;

Procedure Proc1;
var x, y: integer;
begin
   … x := 1; …
end;

begin
   … x := 2; …
end.
```



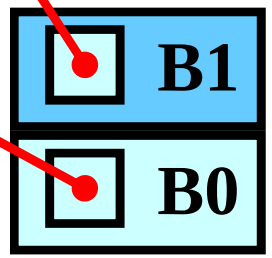
Scope Rules: Example

```
Program P1;
var x, y: integer;

Procedure Proc1;
var x, y: integer;
begin
   … x := 1; …
end;

begin
   … x := 2; …
end.
```

Symbol table:

5:	y	…
4:	x	…
3:	Proc1	…
2:	y	…
1:	x	…

Symbol table

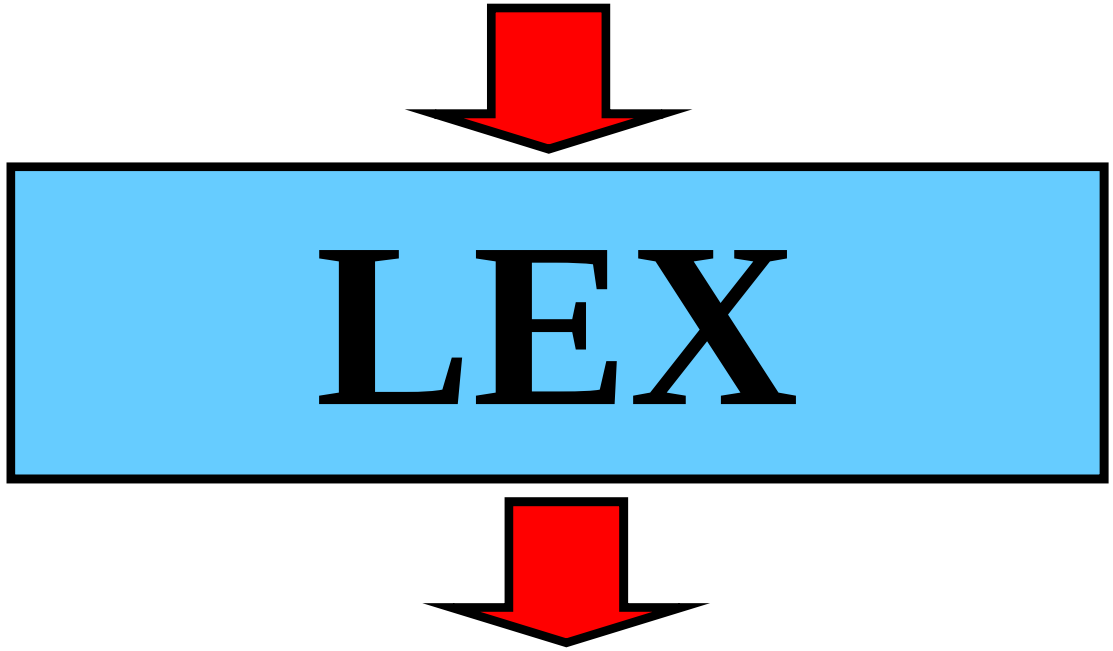


Lex: Basic Idea

- Automatic construction of a **scanner** from **RE**
- Lex compiler and Lex language

Illustration:

Regular expressions



Lexical analyzer (scanner)

Lex: Phases of Compilation

lex.l

(Lex source program)

Lex compiler

lex.yy.c

(C source program)

Tabular representation of a DFA constructed from REs in **lex.l**

lex.yy.c

(C source program)

C compiler

a.out

(unit of scanner)

Source program

a.out

Tokens

Structure of Lex Source Program

/* **Section I:** **Declaration** */

$d_1, d_2, \ \dots \ , d_i$

%% /* **End of Section I*/**

/* **Section II:** **Translation rules** */

$r_1, r_2, \ \dots \ , r_j$

%% /* **End of Section II*/**

/* **Section III:** **Auxiliary procedures*/**

$p_1, p_2, \ \dots \ , p_k$

Basic Regular Expressions in Lex

RE in LEX	Equivalent RE in theory of formal languages
`a`	a
`rs`	$r.s$
`r|s`	$r + s$
`r*`	r^*
`r+`	r^+
`r?`	$r + \varepsilon$
`[a-z]`	$a + b + c + \ldots + z$
`[0-9]`	$0 + 1 + 2 + \ldots + 9$

Section I: Declaration

1) Definitions of manifest constants = token types

2) Definitions based on REs are in the form:

Name_of_RE	**RE**

- **Name_of_RE** represents **RE**
- **{Name_of_RE}** is a reference to **Name_of_RE** used in other REs

Section I: Declaration

1) Definitions of manifest constants = token types

2) Definitions based on REs are in the form:

```
Name_of_RE        RE
```

- `Name_of_RE` represents `RE`
- `{Name_of_RE}` is a reference to `Name_of_RE` used in other REs

Example:

```
#define   IF    256   /* constant for IF */
#define   THEN  257   /* constant for THEN */
#define   ID    258   /* constant for ID */
#define   INT   259   /* constant for NUM */
letter    [a-z]
digit     [0-9]
id        {letter}({letter}|{digit})*
integer   {digit}+
```

Section II: Translation Rules

- Translation rules are in the form:

RE	Action

- **Action** is a program routine that specifies what to do when a lexeme is specified by **RE**

Section II: Translation Rules

- Translation rules are in the form:

RE	Action

- **Action** is a program routine that specifies what to do when a lexeme is specified by **RE**

Example:

```
if          return(IF);
then        return(THEN);
{id}        { yylval = install_id();
              return(ID); }
{integer}   { yylval = install_int();
              return(INT); }
```

yylval: value returned by install_id() = attribute of token

Section III: Auxiliary Procedures

- Auxiliary procedures are needed by translation rules

Section III: Auxiliary Procedures

- Auxiliary procedures are needed by translation rules

Example:

```
…
{id}        { yylval = install_id();
              return(ID); }
…
%%
…
int install_id() {
        /* Procedure to install the lexeme into the symbol
           table and return a pointer thereto */
}
…
```

Complete Source Program in Lex

```
#define   IF    256   /* constant for IF */
#define   THEN  257   /* constant for THEN */
#define   ID    258   /* constant for ID */
#define   INT   259   /* constant for NUM */
int yylval;           /* yylval is visible for parser */
letter    [a-z]
digit     [0-9]
id        {letter}({letter}|{digit})*
integer   {digit}+
%%
if        return(IF);
then      return(THEN);
{id}      {yylval = install_id() ;return(ID) ;}
{integer} {yylval = install_int();return(INT);}
%%
int install_id()  { … }
int install_int() { … }
```