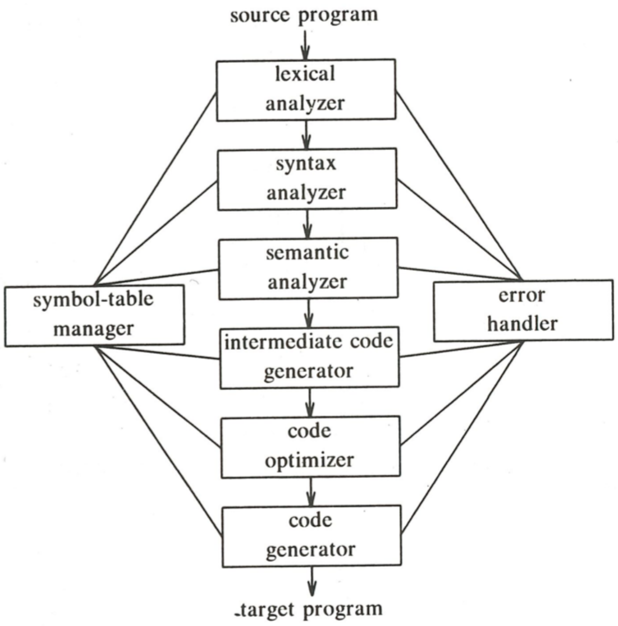
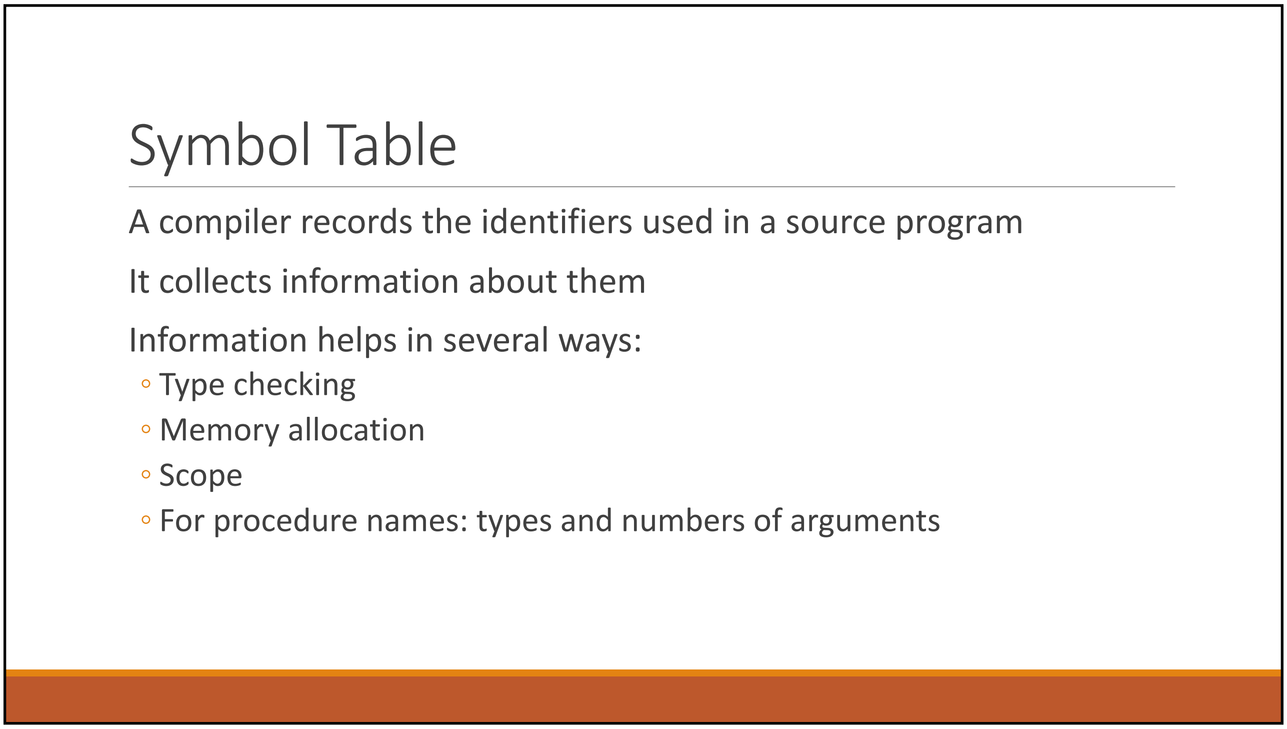
CSSE 404: Compilers Lexical Analysis

MICHAEL WOLLOWSKI

Many, but not all of the materials in this presentation are from the 1st ed. of the Dragon book

Phases of a Compiler





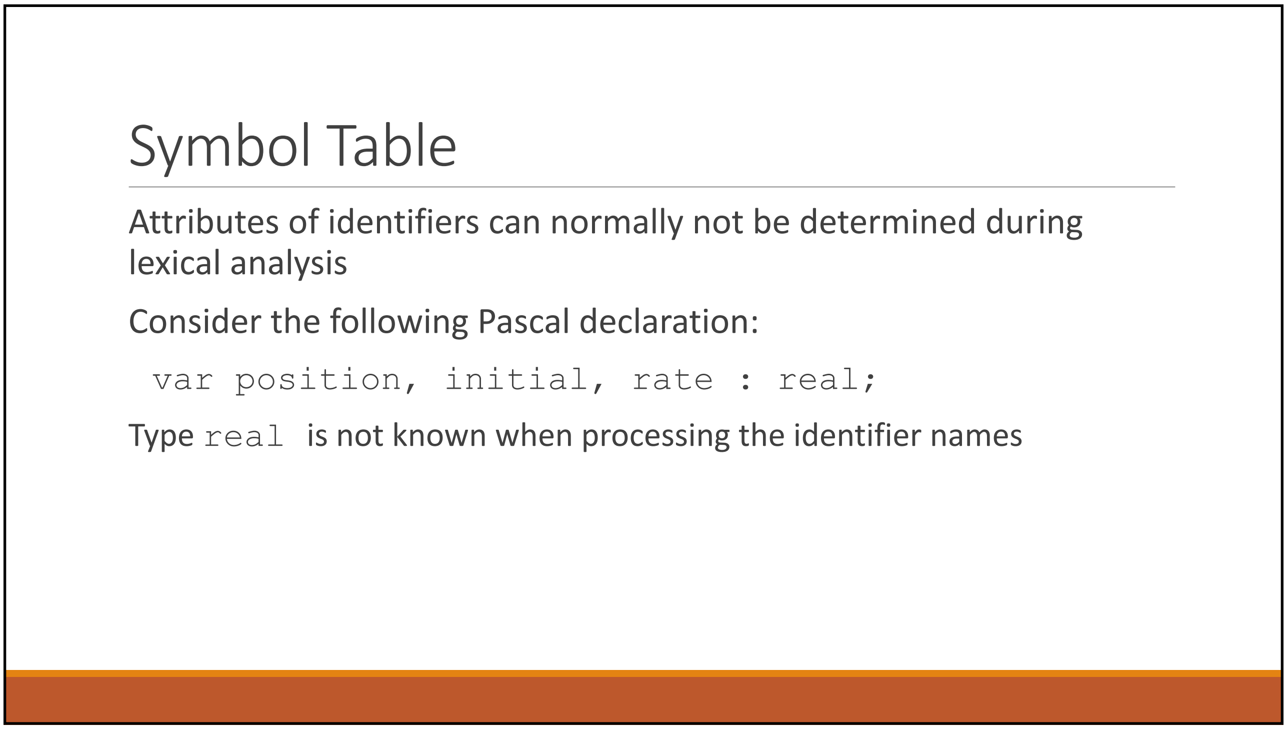
Symbol Table

A compiler records the identifiers used in a source program

It collects information about them

Information helps in several ways:

- Type checking
- Memory allocation
- Scope
- For procedure names: types and numbers of arguments



Symbol Table

Attributes of identifiers can normally not be determined during lexical analysis

Consider the following Pascal declaration:

```
var position, initial, rate : real;
```

Type `real` is not known when processing the identifier names

A more In-Depth Look into the Front-End Lexical Analysis

Consider the statement:

```
position := position + rate * 60
```

We have the following tokens:

- Identifier: `position`
- Assignments symbol:
- Identifier: `position`
- Plus sign
- Identifier: `rate`
- Multiplication sign
- Number: 60

A more In-Depth Look into the Front-End Syntax Analysis

Here, we group tokens into grammatical phrases

We produce a parse tree

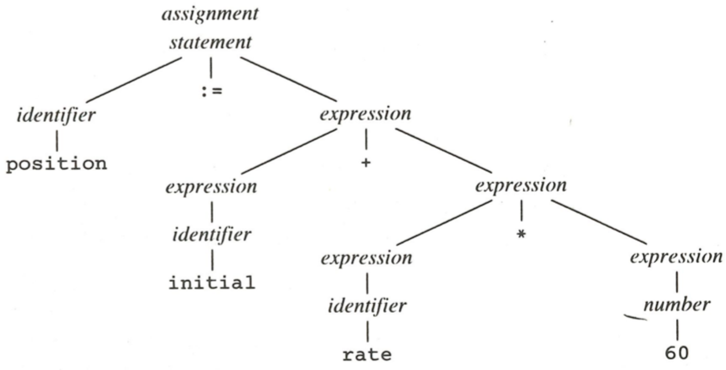


Fig. 1.4. Parse tree for `position := initial + rate * 60`.

Parse Tree Data Structure

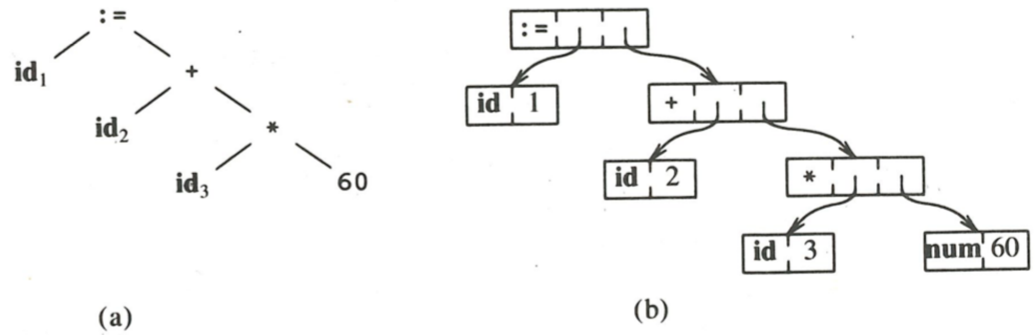


Fig. 1.11. The data structure in (b) is for the tree in (a).

A more In-Depth Look into the Front-End Semantic Analysis

Here, we may insert code to satisfy the grammar.

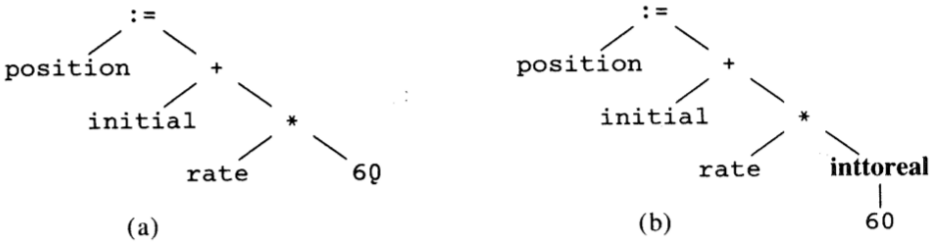
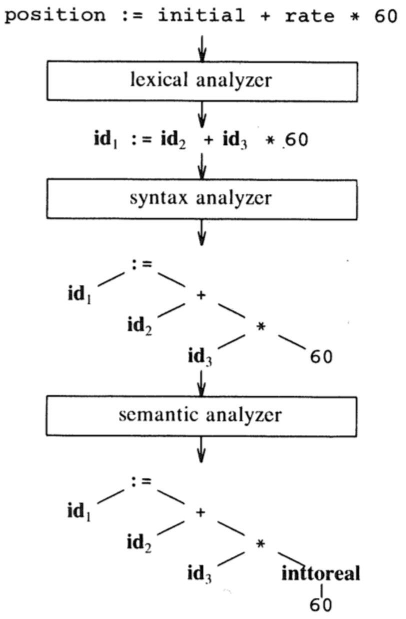


Fig. 1.5. Semantic analysis inserts a conversion from integer to real.

Translation of a Statement

SYMBOL TABLE

1	position	· · ·
2	initial	· · ·
3	rate	· · ·
4		



```
position := initial + rate * 60
```

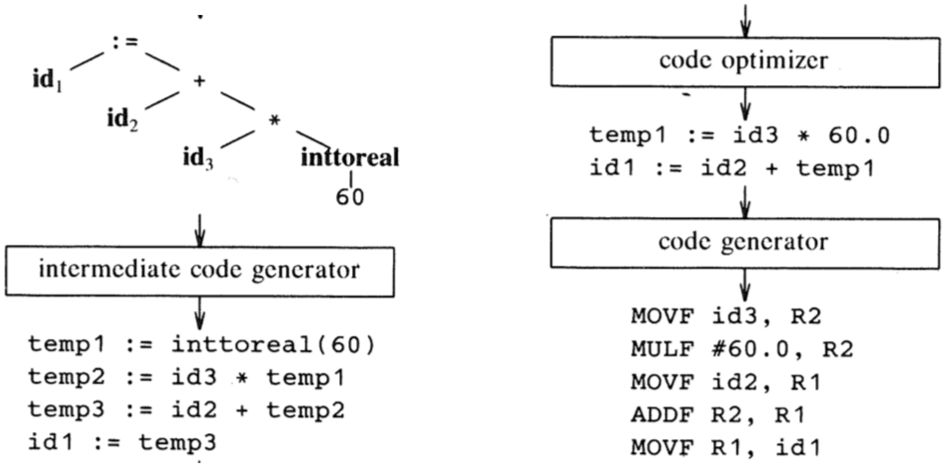
lexical analyzer

id_1 := id_2 + id_3 * 60

syntax analyzer

semantic analyzer

Translation of a Statement



intermediate code generator

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

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
```

Lexical Analysis: Interaction with Parser

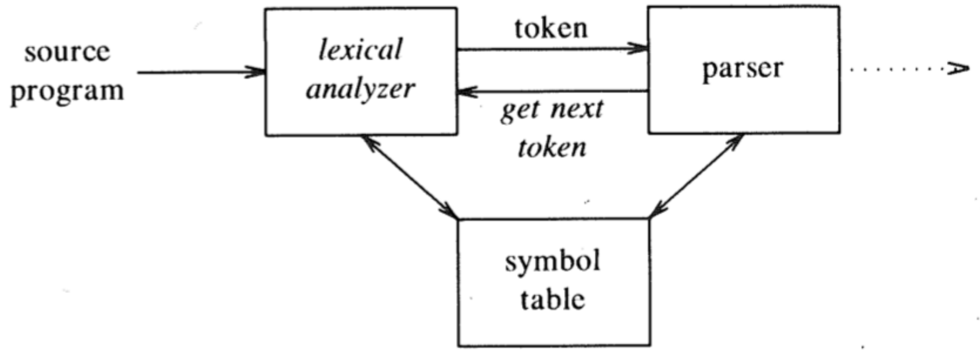
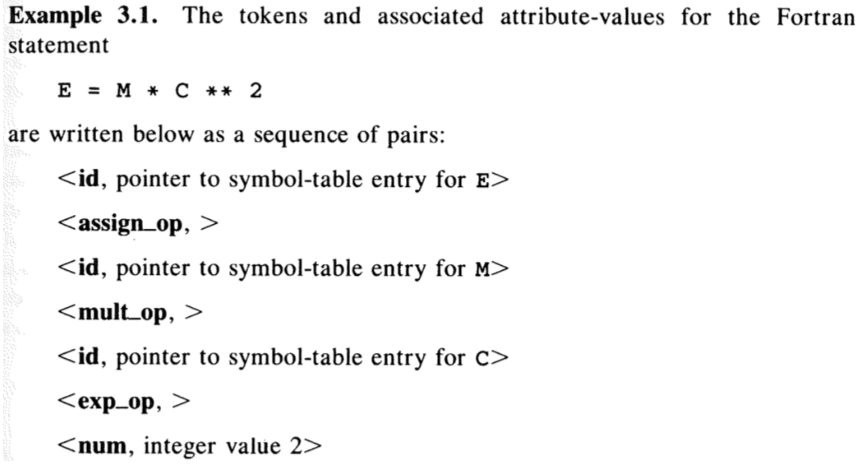


Fig. 3.1. Interaction of lexical analyzer with parser.

Lexical Analysis: Objective



Example 3.1. The tokens and associated attribute-values for the Fortran statement

```
E = M * C ** 2
```

are written below as a sequence of pairs:

<**id**, pointer to symbol-table entry for `E`>

<**assign_op**, >

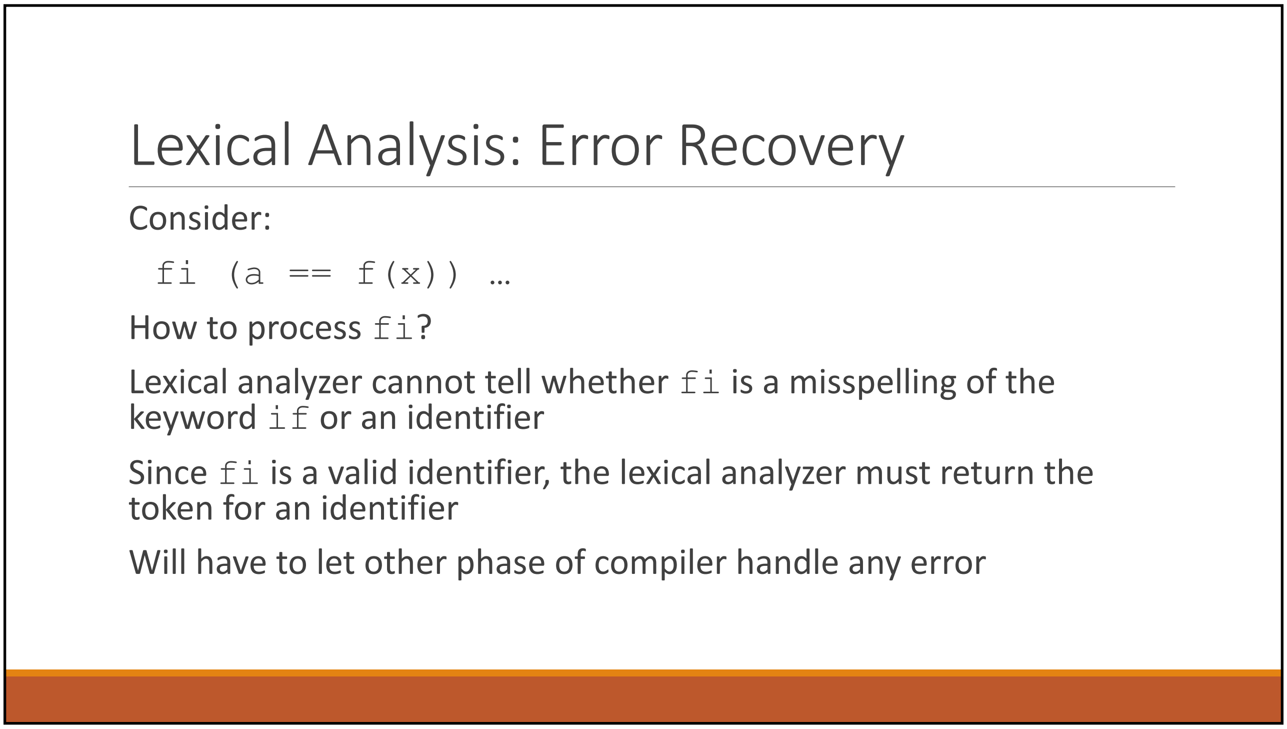
<**id**, pointer to symbol-table entry for `M`>

<**mult_op**, >

<**id**, pointer to symbol-table entry for `C`>

<**exp_op**, >

<**num**, integer value 2>



Lexical Analysis: Error Recovery

Consider:

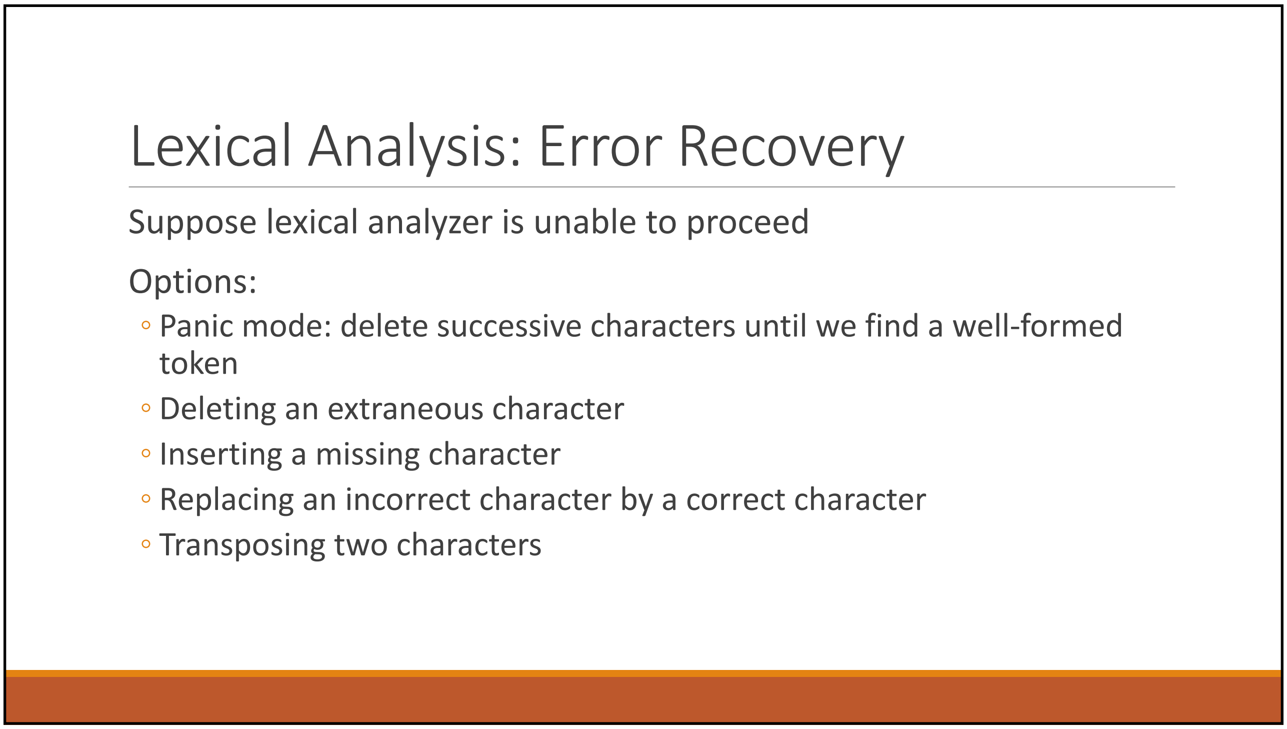
```
fi (a == f(x)) …
```

How to process `fi`?

Lexical analyzer cannot tell whether `fi` is a misspelling of the keyword `if` or an identifier

Since `fi` is a valid identifier, the lexical analyzer must return the token for an identifier

Will have to let other phase of compiler handle any error

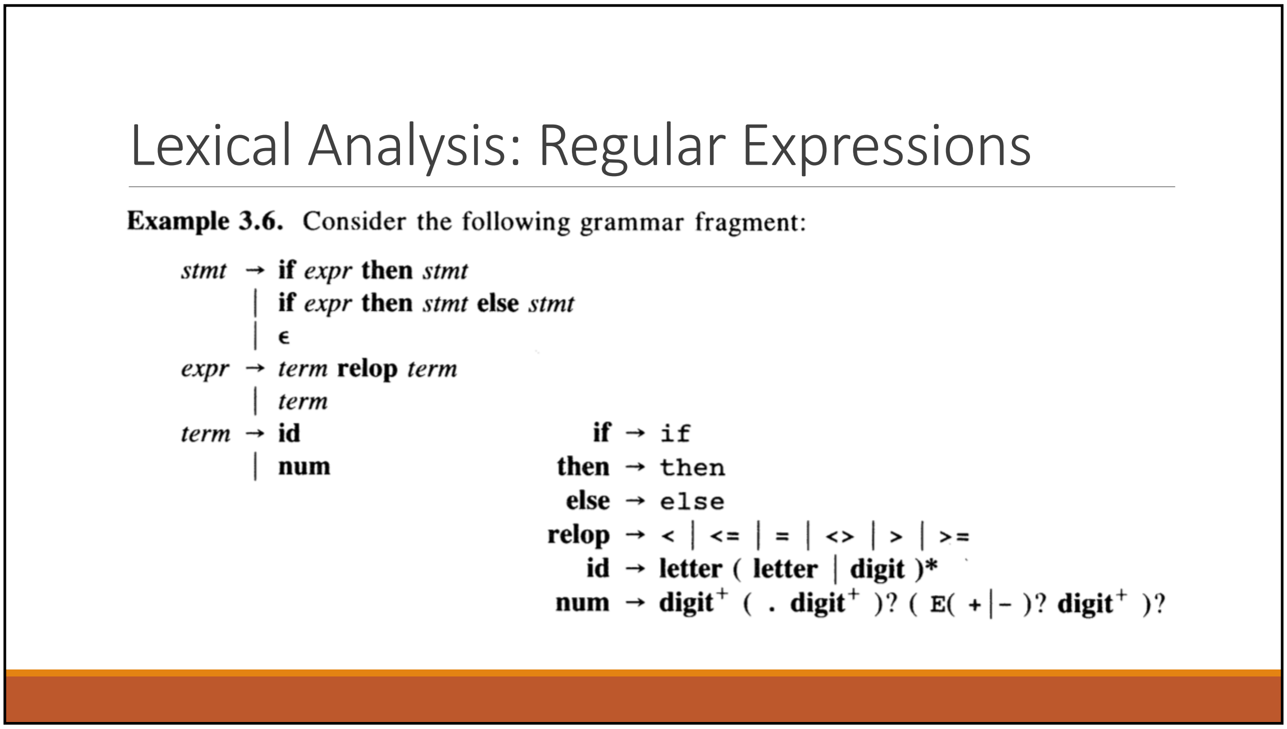


Lexical Analysis: Error Recovery

Suppose lexical analyzer is unable to proceed

Options:

- Panic mode: delete successive characters until we find a well-formed token
- Deleting an extraneous character
- Inserting a missing character
- Replacing an incorrect character by a correct character
- Transposing two characters

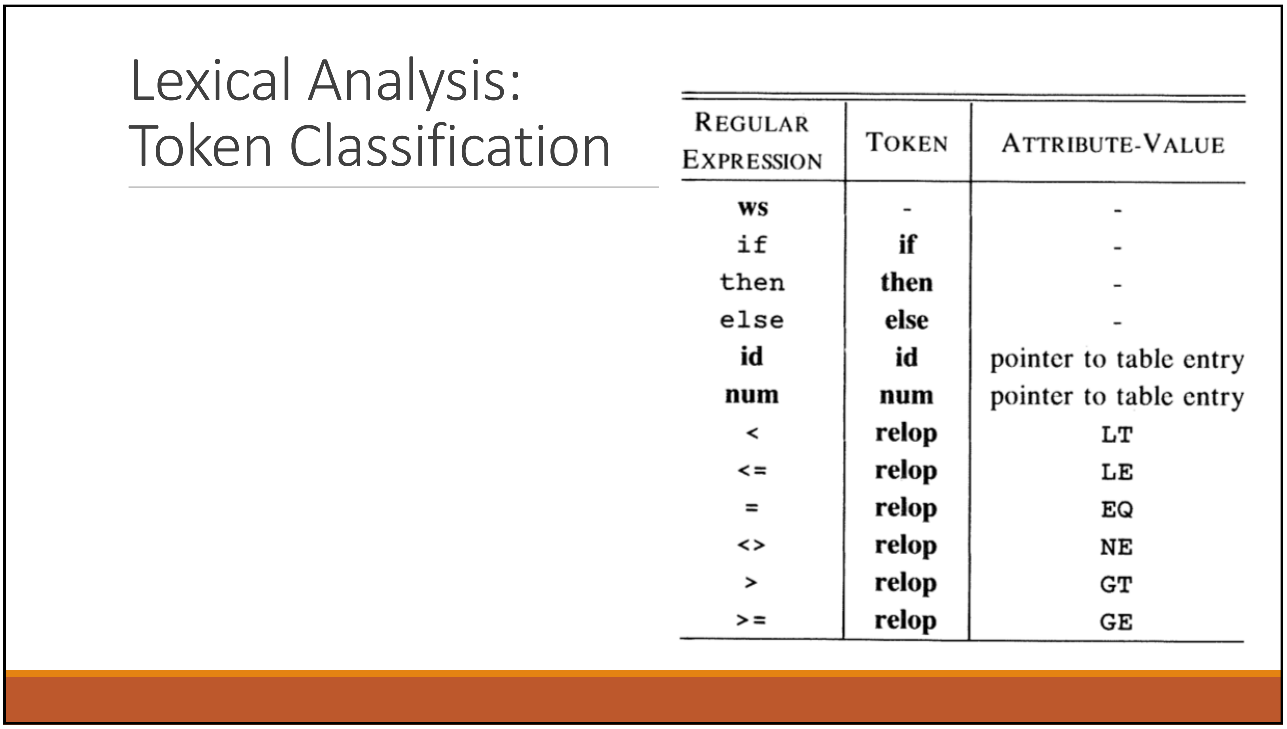


Lexical Analysis: Regular Expressions

Example 3.6. Consider the following grammar fragment:

$$
\begin{array}{lcl}
\textit{stmt} & \rightarrow & \textbf{if}\ \textit{expr}\ \textbf{then}\ \textit{stmt} \\
 & | & \textbf{if}\ \textit{expr}\ \textbf{then}\ \textit{stmt}\ \textbf{else}\ \textit{stmt} \\
 & | & \epsilon \\
\textit{expr} & \rightarrow & \textit{term}\ \textbf{relop}\ \textit{term} \\
 & | & \textit{term} \\
\textit{term} & \rightarrow & \textbf{id} \\
 & | & \textbf{num}
\end{array}
$$

$$
\begin{array}{rcl}
\textbf{if} & \rightarrow & \texttt{if} \\
\textbf{then} & \rightarrow & \texttt{then} \\
\textbf{else} & \rightarrow & \texttt{else} \\
\textbf{relop} & \rightarrow & \texttt{<}\ |\ \texttt{<=}\ |\ \texttt{=}\ |\ \texttt{<>}\ |\ \texttt{>}\ |\ \texttt{>=} \\
\textbf{id} & \rightarrow & \textbf{letter}\ (\ \textbf{letter}\ |\ \textbf{digit}\ )^{*} \\
\textbf{num} & \rightarrow & \textbf{digit}^{+}\ (\ .\ \textbf{digit}^{+}\ )?\ (\ \texttt{E}(\ +\ |\ -\ )?\ \textbf{digit}^{+}\ )?
\end{array}
$$



Lexical Analysis: Token Classification

REGULAR EXPRESSION	TOKEN	ATTRIBUTE-VALUE
ws	-	-
`if`	**if**	-
`then`	**then**	-
`else`	**else**	-
id	**id**	pointer to table entry
num	**num**	pointer to table entry
<	**relop**	`LT`
<=	**relop**	`LE`
=	**relop**	`EQ`
<>	**relop**	`NE`
>	**relop**	`GT`
>=	**relop**	`GE`

Lexical Analysis: Transition Diagrams

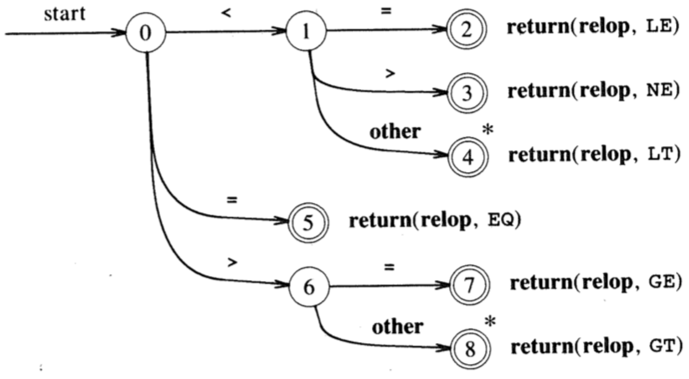


Fig. 3.12. Transition diagram for relational operators.

Lexical Analysis: Transition Diagrams

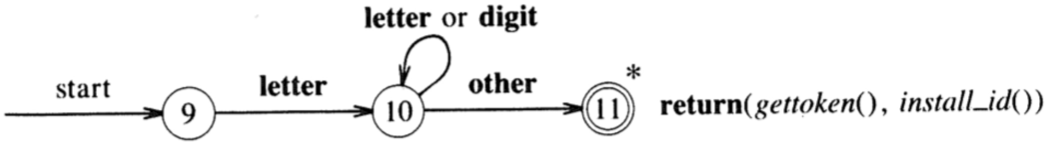


Fig. 3.13. Transition diagram for identifiers and keywords.

Lexical Analysis: Transition Diagrams

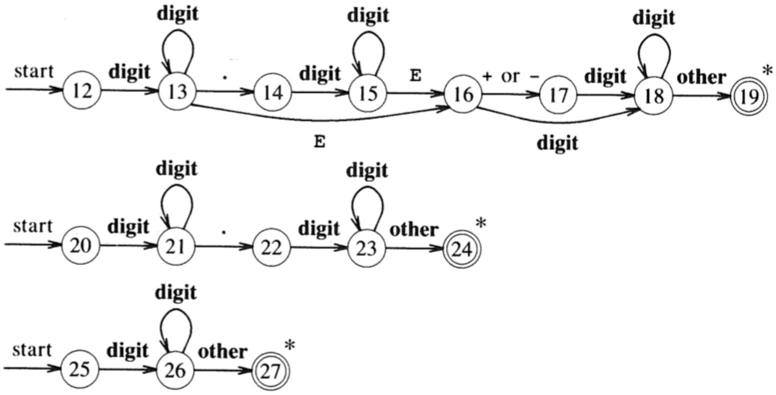
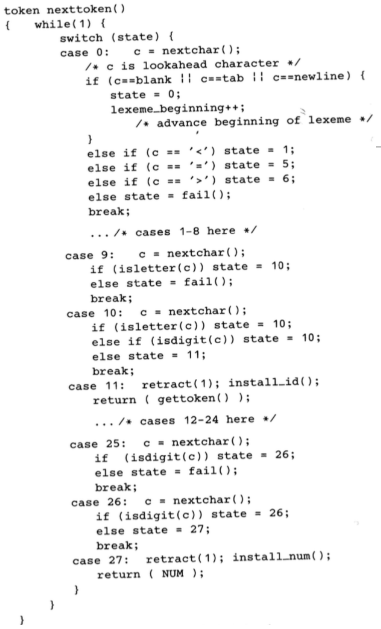


Fig. 3.14. Transition diagrams for unsigned numbers in Pascal.

A Lexical Analyzer



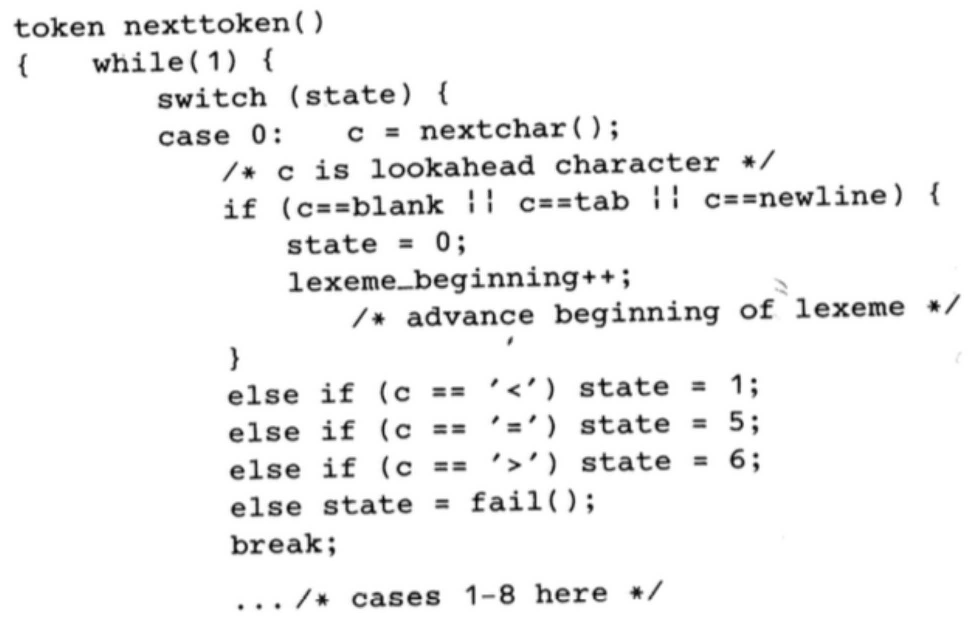
```
token nexttoken()
{   while(1) {
        switch (state) {
        case 0:    c = nextchar();
            /* c is lookahead character */
            if (c==blank || c==tab || c==newline) {
                state = 0;
                lexeme_beginning++;
                    /* advance beginning of lexeme */
            }
            else if (c == '<') state = 1;
            else if (c == '=') state = 5;
            else if (c == '>') state = 6;
            else state = fail();
            break;

            .../* cases 1-8 here */

        case 9:    c = nextchar();
            if (isletter(c)) state = 10;
            else state = fail();
            break;
        case 10:   c = nextchar();
            if (isletter(c)) state = 10;
            else if (isdigit(c)) state = 10;
            else state = 11;
            break;
        case 11:   retract(1); install_id();
            return ( gettoken() );

            .../* cases 12-24 here */

        case 25:   c = nextchar();
            if  (isdigit(c)) state = 26;
            else state = fail();
            break;
        case 26:   c = nextchar();
            if (isdigit(c)) state = 26;
            else state = 27;
            break;
        case 27:   retract(1); install_num();
            return ( NUM );
        }
    }
}
```



```
token nexttoken()
{    while(1) {
         switch (state) {
         case 0:    c = nextchar();
             /* c is lookahead character */
             if (c==blank ¦¦ c==tab ¦¦ c==newline) {
                 state = 0;
                 lexeme_beginning++;
                     /* advance beginning of lexeme */
             }
             else if (c == ’<’) state = 1;
             else if (c == ’=’) state = 5;
             else if (c == ’>’) state = 6;
             else state = fail();
             break;
             .../* cases 1-8 here */
```