stmt		**if** *e* **then** *stmt*
	|	**if** *e* **then** *stmt* **else** *stmt*
	|	**while** *e* **do** *stmt*
	|	**begin** *list* **end**
	|	*s*
list		*list* *stmt*
	|	*stmt*

Figure 4 56 A grammar for certain kinds of statements

4 9 Parser Generators

This section shows how a parser generator can be used to facilitate the construc tion of the front end of a compiler We shall use the LALR parser generator `Yacc` as the basis of our discussion since it implements many of the concepts discussed in the previous two sections and it is widely available `Yacc` stands for yet another compiler compiler re ecting the popularity of parser generators in the early 1970s when the rst version of `Yacc` was created by S C Johnson `Yacc` is available as a command on the UNIX system and has been used to help implement many production compilers

4 9 1 The Parser Generator `Yacc`

A translator can be constructed using `Yacc` in the manner illustrated in Fig 4 57 First a le say `translate y` containing a `Yacc` speci cation of the translator is prepared The UNIX system command

```
yacc translate y
```

transforms the le `translate y` into a C program called `y tab c` using the LALR method outlined in Algorithm 4 63 The program `y tab c` is a repre sentation of an LALR parser written in C along with other C routines that the user may have prepared The LALR parsing table is compacted as described in Section 4 7 By compiling `y tab c` along with the `ly` library that contains the LR parsing program using the command

```
cc y tab c  ly
```

we obtain the desired object program `a out` that performs the translation spec i ed by the original `Yacc` program [7] If other procedures are needed they can be compiled or loaded with `y tab c` just as with any C program

A `Yacc` source program has three parts

[7] The name `ly` is system dependent

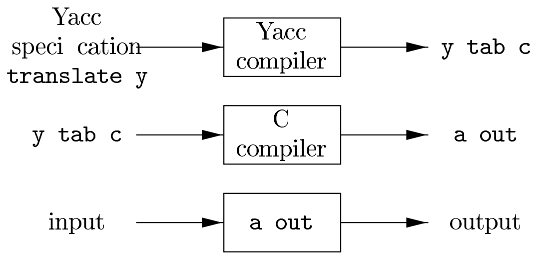

Figure 4 57 Creating an input output translator with `Yacc`

declarations

translation rules

supporting C routines

Example 4 69 To illustrate how to prepare a `Yacc` source program let us construct a simple desk calculator that reads an arithmetic expression evaluates it and then prints its numeric value We shall build the desk calculator starting with the with the following grammar for arithmetic expressions

$$\begin{array}{lll} E & & E \quad T \mid T \\ T & & T \quad F \mid F \\ F & & \quad E \quad \mid \textbf{digit} \end{array}$$

The token **digit** is a single digit between 0 and 9 A `Yacc` desk calculator program derived from this grammar is shown in Fig 4 58 □

The Declarations Part

There are two sections in the declarations part of a `Yacc` program both are optional In the rst section we put ordinary C declarations delimited by and Here we place declarations of any temporaries used by the translation rules or procedures of the second and third sections In Fig 4 58 this section contains only the include statement

```
include  ctype h
```

that causes the C preprocessor to include the standard header le `ctype h` that contains the predicate `isdigit`

Also in the declarations part are declarations of grammar tokens In Fig 4 58 the statement

```
token DIGIT
```

```
 include  ctype h

 token DIGIT

line     expr   n          printf   d n    1

expr     expr     term            1    3
         term

term     term     factor          1    3
         factor

factor       expr                 2
         DIGIT


yylex
    int c
    c   getchar
    if  isdigit c
        yylval   c  0
        return DIGIT

    return c
```

Figure 4 58 `Yacc` speci cation of a simple desk calculator

declares `DIGIT` to be a token Tokens declared in this section can then be used in the second and third parts of the `Yacc` speci cation If `Lex` is used to create the lexical analyzer that passes token to the `Yacc` parser then these token declarations are also made available to the analyzer generated by `Lex` as discussed in Section 3 5 2

The Translation Rules Part

In the part of the `Yacc` speci cation after the rst   pair we put the translation rules Each rule consists of a grammar production and the associated semantic action A set of productions that we have been writing

$$\langle\text{head}\rangle \qquad \langle\text{body}\rangle_1 \;|\; \langle\text{body}\rangle_2 \;|\; \qquad |\; \langle\text{body}\rangle_n$$

would be written in `Yacc` as

$$
\begin{array}{lll}
\langle\text{head}\rangle & \langle\text{body}\rangle_1 & \langle\text{semantic action}\rangle_1 \\
 & \langle\text{body}\rangle_2 & \langle\text{semantic action}\rangle_2 \\
 & & \\
 & \langle\text{body}\rangle_n & \langle\text{semantic action}\rangle_n
\end{array}
$$

In a `Yacc` production unquoted strings of letters and digits not declared to be tokens are taken to be nonterminals A quoted single character e g `c` is taken to be the terminal symbol `c` as well as the integer code for the token represented by that character i e `Lex` would return the character code for `c` to the parser as an integer Alternative bodies can be separated by a vertical bar and a semicolon follows each head with its alternatives and their semantic actions The rst head is taken to be the start symbol

A `Yacc` semantic action is a sequence of C statements In a semantic action the symbol refers to the attribute value associated with the nonterminal of the head while i refers to the value associated with the ith grammar symbol terminal or nonterminal of the body The semantic action is performed when ever we reduce by the associated production so normally the semantic action computes a value for in terms of the i s In the `Yacc` speci cation we have written the two E productions

$$E \quad E \quad T \mid T$$

and their associated semantic actions as

```
expr   expr      term              1     3
       term
```

Note that the nonterminal `term` in the rst production is the third grammar symbol of the body while is the second The semantic action associated with the rst production adds the value of the `expr` and the `term` of the body and assigns the result as the value for the nonterminal `expr` of the head We have omitted the semantic action for the second production altogether since copying the value is the default action for productions with a single grammar symbol in the body In general `1` is the default semantic action

Notice that we have added a new starting production

```
line   expr   n        printf   d n    1
```

to the `Yacc` speci cation This production says that an input to the desk calculator is to be an expression followed by a newline character The semantic action associated with this production prints the decimal value of the expression followed by a newline character

The Supporting C Routines Part

The third part of a `Yacc` speci cation consists of supporting C routines A lexical analyzer by the name `yylex` must be provided Using `Lex` to produce `yylex` is a common choice see Section 4 9 3 Other procedures such as error recovery routines may be added as necessary

The lexical analyzer `yylex` produces tokens consisting of a token name and its associated attribute value If a token name such as `DIGIT` is returned the token name must be declared in the rst section of the `Yacc` speci cation The attribute value associated with a token is communicated to the parser through a `Yacc` de ned variable `yylval`

The lexical analyzer in Fig 4 58 is very crude It reads input characters one at a time using the C function `getchar` If the character is a digit the value of the digit is stored in the variable `yylval` and the token name `DIGIT` is returned Otherwise the character itself is returned as the token name

4 9 2 Using `Yacc` with Ambiguous Grammars

Let us now modify the `Yacc` speci cation so that the resulting desk calculator becomes more useful First we shall allow the desk calculator to evaluate a sequence of expressions one to a line We shall also allow blank lines between expressions We do so by changing the rst rule to

```
lines   lines expr   n      printf   g n    2
        lines   n
           empty
```

In `Yacc` an empty alternative as the third line is denotes

Second we shall enlarge the class of expressions to include numbers with a decimal point instead of single digits and to include the arithmetic operators both binary and unary and The easiest way to specify this class of expressions is to use the ambiguous grammar

$$E \quad E \quad E \mid E \quad E \mid E \quad E \mid E \quad E \mid \quad E \mid \quad E \mid \textbf{number}$$

The resulting `Yacc` speci cation is shown in Fig 4 59

Since the grammar in the `Yacc` speci cation in Fig 4 59 is ambiguous the LALR algorithm will generate parsing action con icts `Yacc` reports the num ber of parsing action con icts that are generated A description of the sets of items and the parsing action con icts can be obtained by invoking `Yacc` with a `v` option This option generates an additional le `y output` that contains the kernels of the sets of items found for the grammar a description of the parsing action con icts generated by the LALR algorithm and a readable represen tation of the LR parsing table showing how the parsing action con icts were resolved Whenever `Yacc` reports that it has found parsing action con icts it