Chapter IV

Development of the Translator

4.1 Components of the Translator

The Translator is composed of four major components:

1. **The Main Routine.** This routine is where the translator begins its execution. It performs all necessary program initializations, calls the parser routine to perform the translation process, and then does necessary wrap-up works before exiting.

2. **The Parser/Code-generator.** This is the translator's part that actually performs the translation process. Since the translator employs the so-called syntax-directed translation scheme in which the code is generated strictly on the basis of the language syntax, the code generator part of the translator is embedded within the parser to form a single parser/code-generator module. The parser calls the scanner routine everytime it requires the next token.

3. **The Scanner.** The scanner, more formally called the lexical analyzer, reads the input stream of characters from the AWK source program and translate it into a stream of basic language elements called tokens. Each call to the scanner by the parser returns the next token in the input stream.

4. **The Symbol-Table Managers.** To support the translation process, the translator records information about various kinds of identifiers in the source program into the symbol tables. These symbol tables are accessed and maintained by the symbol-table managers.

Figure 4.1 summarizes the relationships among the translator's components. The continuous-lined arrows designate the calling relationship while the dotted ones indicate the data flow.

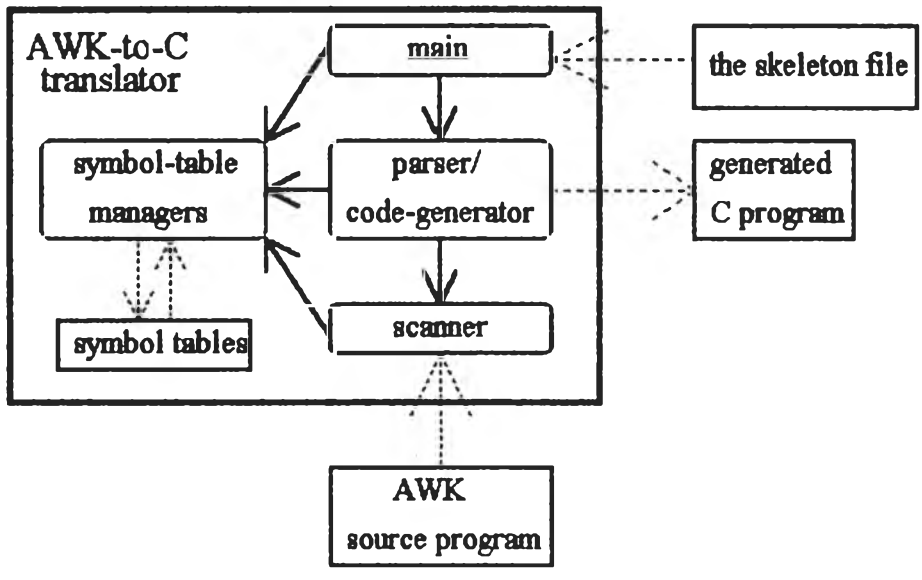

Figure 4.1 Components of the AWK-to-C translator and their relationships

4.2 The Symbol Tables

For translation convenience, the translator divides AWK identifiers into three categories: *keywords*, *global names*, and *parameter names*. Table 4.1 defined the types of identifiers comprising each category. Accordingly, the translator maintains a separate symbol table for each category of identifiers.

Table 4.1 Translation-time Categories of AWK Identifiers

Category	Definition
keywords	all of the AWK keywords including the built-in function names but excluding the built-in variable names
global names	global variable names, both built-in and user-defined, and user-defined function names
parameter names	local variable names

4.2.1 The Keyword Table

The keyword table is a fixed-sized, sorted table of AWK keywords. The data structure for each table entry is a C structure *Kword*, described in Table 4.2. The table itself is implemented in C as a static array of *Kword* structures.

Table 4.2 Fields in the Kword Structure for Keyword-Table Entries

Field	C Data Type	Description
name	char *	The keyword string
token	int	Its corresponding token number

The keyword table is used solely by the scanner. Whenever the scanner encounters an identifier in the source program, it looks up the keyword table to see whether the identifier is a keyword. If it is, the scanner returns the token number stored in the *token* field of the corresponding table entry.

4.2.2 The Global-name Table

The global-name table stores information of each global name in the source program. This table grows dynamically; new entry will be added as a new global name is encountered. The data structure for each table entry is a C structure *NameTabNode*, which is described in Table 4.3. The table itself is implemented by using a hashing scheme.

Table 4.3 Fields in the NameTabNode Structure for the Global-Name Table Entries

Field	C Data Type	Description
name	char *	The identifier's lexeme
type	int	The identifier's category, which can be one of the following: ID_B_VAR for built-in variable ID_U_VAR for user-defined variable ID_U_FUNC for user-defined function
next	NameTabNode *	A pointer to the next node in the linked list

During program initialization, the main routine initializes the global-name table by preloading it with all of the AWK built-in variable names. As the translator is parsing the source program, new entry will be added into the table for each new global name encountered. The parser mainly uses this table to make sure that identifiers used for variables are not to be used later as function names, and vice versa. The code generator also looks up this table to generate proper C declaration code for each global name.

4.2.3 The Parameter-name Table

As the translator begins parsing a user-defined function, it records the function's local variable names in the parameter-name table in the order as they appears in the function's parameter list. The code generator uses this table to generate codes concerning the allocation/deallocation of the function's local variables and the parameter passing mechanism. This table will be emptied immediately after the parsing/code-gerating process for the current user-defined function has finished.

The data structure for parameter-name table is simply a list of names, implemented in C as an array of character pointers. An integer variable *nParam* is used to record the current number of names in the table.

4.3 The Main Routine

The main routine is the starting point of execution for the translator. It has a simple algorithm shown in figure 4.2.

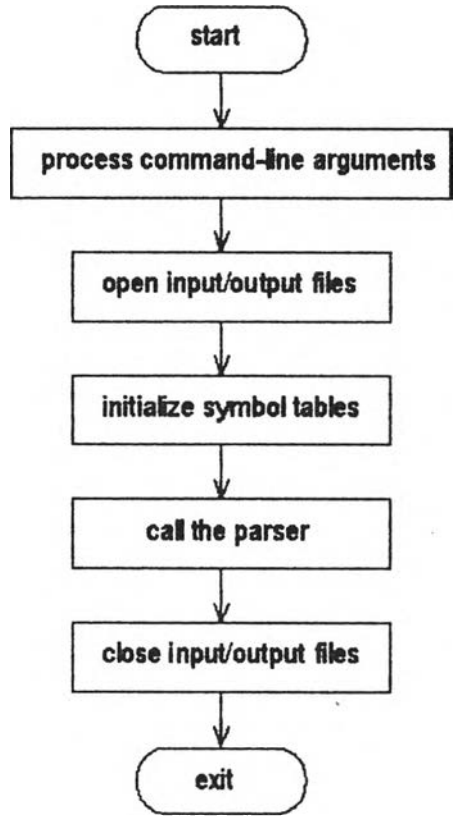

Figure 4.2 Algorithm for the translator's main routine

4.4 The Scanner

4.4.1 Development of the Scanner

The Unix scanner generator *lex* was used to produce the scanner module of the translator. A lex specification file named *scanner.l*, written in a special specification language recognized by lex, contains a set of rules for matching tokens and corresponding

groups of C action statements that are executed when a token is matched. Lex reads the file scanner.l as its input and generates the C code for the scanner in a file named *scanner.c*. The C function *yylex* contained in scanner.c is the lexical analyzer (or scanner) routine that the parser calls to get the next token. Figure 4.3 summarizes the scanner development process.

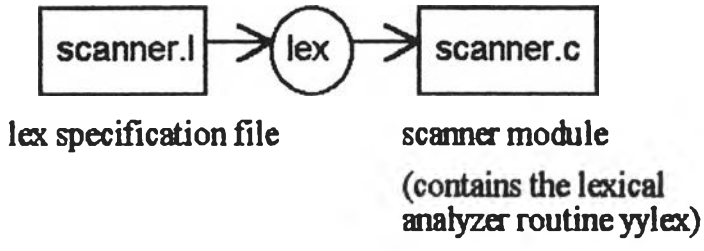

Figure 4.3 The scanner development process

Figure 4.4 illustrates the algorithm of the scanner routine. The scanner tries to match the input characters against each specified pattern in order. If a pattern is matched, it executes the corresponding action and then returns the token number corresponding to the matched pattern to the parser. If there is no pattern matched, the special token ERROR is returned.

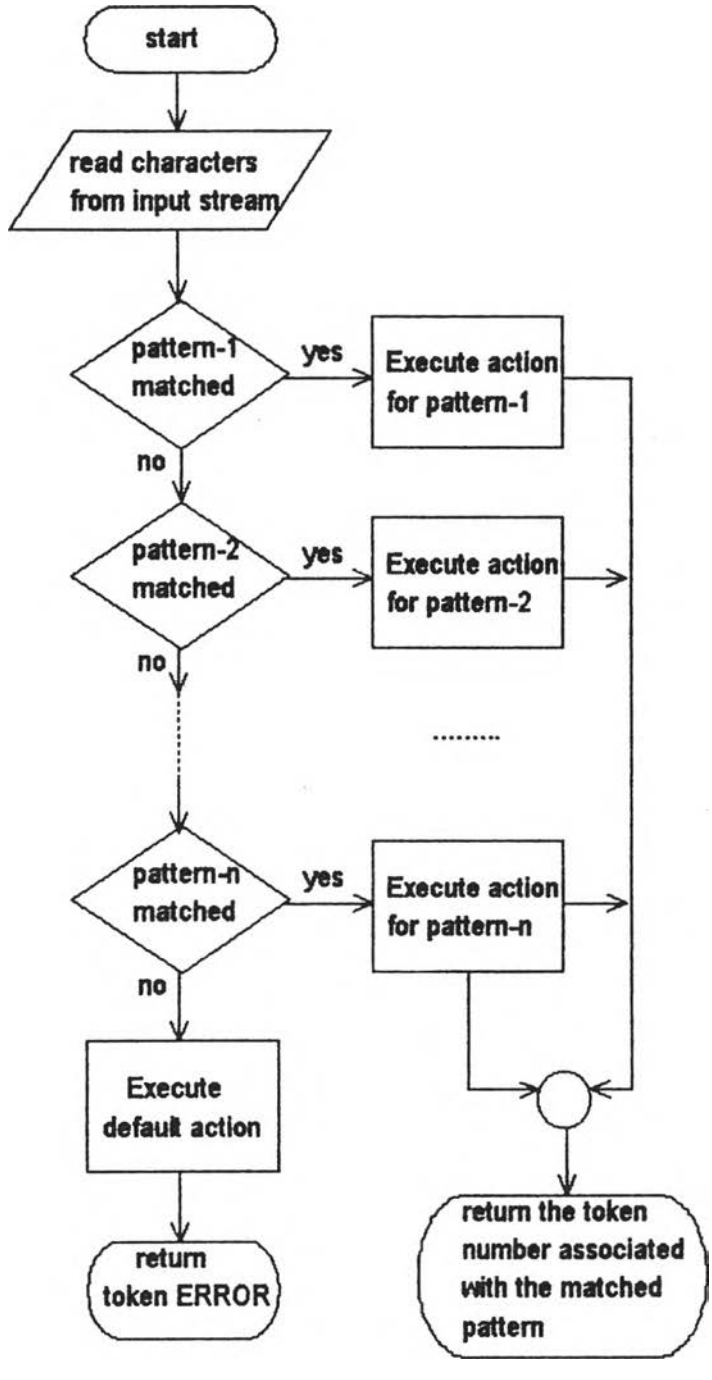

Figure 4.4 Algorithm of the scanner routine

4.4.2 Token Definitions

The scanner partitions the stream of input characters from the AWK source program into basic syntactic elements called tokens. The tokens for a language is inherently defined by the grammar of the language itself. The yacc grammar for the AWK language listed in

Appendix B, upon which the parser is based, defines the total of fifty-seven tokens described in Table 4.4. The first column of the table lists each token's symbolic name for its token number that the scanner actually returns to the parser to identify the token.

Table 4.4 Tokens Recognized by the Scanner

Symbolic Name	Description
ERROR	the error token to be returned when the scanner has found an error in the input stream
TNUMBER	numeric constant
TSTRING	string constant between a pair of double quotes
REGEXP	regular expression constant between a pair of /'s
FUNC_CALL	non-keyword identifier that is followed by a left parenthesis
NAME	non-keyword identifier that is not followed by a left parenthesis
APPEND_OP	the redirection operator >>
ASSIGNOP	the assignment operators
DECREMENT	the increment operator ++
INCREMENT	the decrement operator --
MATCHOP	the matching operators ~ and !~
RELOP	all of the relational operators except > and <
TOK_AND	the operator &&
TOK_BEGIN	the keyword BEGIN
TOK_BREAK	the keyword break
TOK_BUILTIN	the built-in functions
TOK_CLOSE	the keyword close
TOK_CONTINUE	the keyword continue
TOK_DELETE	the keyword delete
TOK_DO	the keyword do
TOK_ELSE	the keyword else
TOK_END	the keyword END
TOK_EXIT	the keyword exit
TOK_FOR	the keyword for

Table 4.4 Tokens Recognized by the Scanner (continued)

Symbolic Name	Description
TOK_FUNCTION	the keyword function
TOK_GETLINE	the keyword getline
TOK_IF	the keyword if
TOK_IN	the keyword in
TOK_LENGTH	the keyword length
TOK_NEXT	the keyword next
TOK_OR	the operator ||
TOK_PRINT	the keyword print
TOK_PRINTF	the keyword printf
TOK_RETURN	the keyword return
TOK_WHILE	the keyword while
NEWLINE	the end of the line
','	the literal ,
'!'	the literal !
'$'	the literal $
'%'	the literal %
'('	the literal (
')'	the literal)
'*'	the literal *
'+'	the literal +
'-'	the literal -
'/'	the literal /
':'	the literal :
';'	the literal ;
'<'	the literal <
'>'	the literal >
'?'	the literal ?
'['	the literal [
']'	the literal]
'^'	the literal ^
'{'	the literal {
'}'	the literal }
'|'	the literal |

4.5 The Parser/Code-Generator

4.5.1 Development of the Parser/Code-generator

The Unix parser generator *yacc* was used to produce the parser/code-generator module of the translator. A yacc specification file named *parser.y*, written in a special specification language recognized by yacc, contains a yacc grammar for the AWK language with embedded C action statements that generate appropriate code for each grammatical construct. Yacc reads the file parser.y as its input and generates the C code for the parser/code-generator in a file named *parser.c*, and also the header file *parser.h* containing token definitions needed by the scanner module. The C function *yyparse* contained in parser.c is the actual parser/code-generator routine that the main routine calls to parse the AWK source program and generate the C-code output. Figure 4.5 summarizes the parser development process.

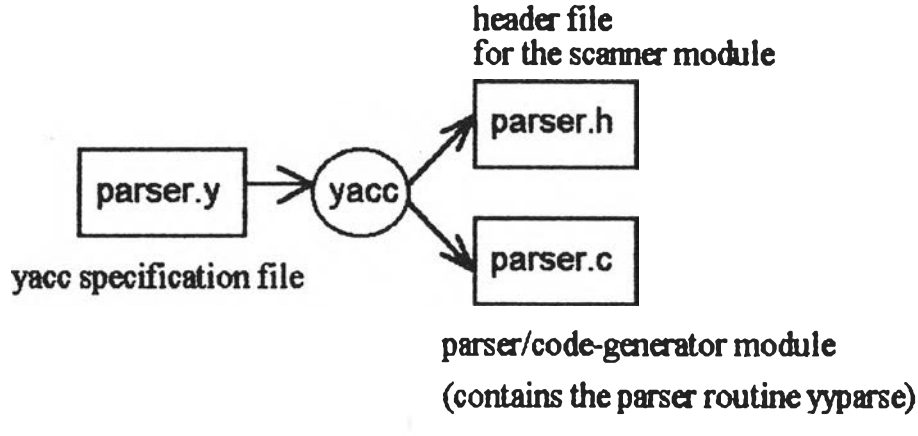

Figure 4.5 The parser development process

4.5.2 The Grammar for AWK Used by the Parser

The yacc specification file parser.y was designed and written around a yacc grammar for the AWK language. The grammar, listed in Appendix B, is derived and adapted from the grammar used by the Free Software Foundation's GNU gawk, which is a freely available AWK language processor.

4.6 Interface Between the Scanner and the Parser

Lex and yacc were designed to generate the scanner and the parser routines that works cooperatively together. Figure 4.6 illustrates the flow of control in the scanner and the parser routines.

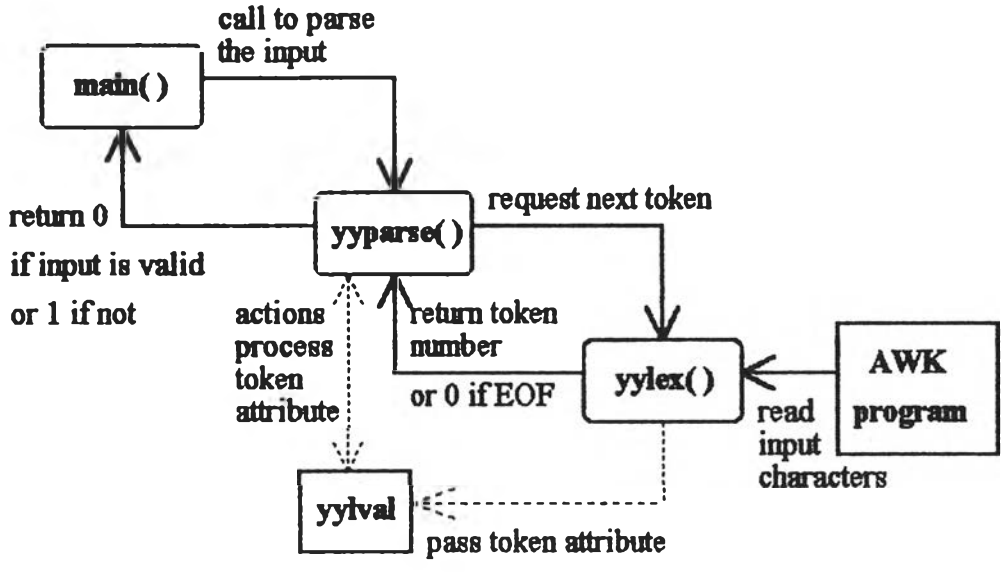

Figure 4.6 How the scanner and the parser routines work together

The main routine of the translator invokes the parser routine *yyparse* to parse the input source program to check whether or not the input is syntactically valid, and to generate C-code output. *Yyparse* invokes the scanner routine *yylex* each time it needs a token. This scanner routine reads the input stream, and for each token that it matches, it returns the token number to the parser. The scanner routine can also pass an attribute value of the token via the external variable *yylval*. The parser's action codes can make use of this attribute value in the process of generating the output.

When the scanner routine has exhausted the input, it returns 0 to the parser. If the parser has recognized the start rule, the top-level in the grammar's hierarchical structure, then the parser returns 0, meaning that the input was syntactically valid. If at any time it receives a token number or a sequence of tokens that it does

not recognize or if the scanner returns 0 before the start symbol has been recognized, then the parser returns 1, reporting a syntax error.

Not every token has a token attribute associated with it. Of all the fifty-seven tokens listed in Table 4.4, only nine, namely FUNC_CALL, NAME, ASSIGNOP, RELOP, MATCHOP, TNUMBER, TSTRING, REGEXP, and TOK_BUILTIN, are required by the parser to have a token attribute passed on the variable *yylval*. The attribute for each of these tokens is defined to be its lexeme. For the other forty-eight tokens, the parser does not need to know its lexeme because there is only one possible lexeme for each of them; thereby the token itself already implies what its lexeme is.