The Construction of Variant Translators for Icon*

Ralph E. Griswold

TR 83-19a

December 31, 1983; Revised June 2, 1984

Department of Computer Science

The University of Arizona

Tucson, Arizona 85721

*This work was supported by the National Science Foundation under Grant MCS81-01916

The Construction of Variant Translators for Icon

1. Introduction

A preprocessor, which translates text from source language $\mathcal{A}$ to source language $\mathcal{B}$,

$$\mathcal{A} \rightarrow \mathcal{B}$$

is a popular and effective means of implementing $\mathcal{A}$, given an implementation of $\mathcal{B}$. $\mathcal{B}$ is referred to as the target language. Ratfor [1] is perhaps the best known and most widely used example of this technique, although there are many others.

In some cases $\mathcal{A}$ is a variant of $\mathcal{B}$. An example is Cg [2], a variant of C that includes a generator facility similar to that of Icon [3]. Cg consists of C and some additional syntax that a preprocessor translates into standard C. A run-time system provides the necessary semantic support for generators. Note that the Cg preprocessor is a source-to-source translator:

$$\mathrm{Cg} \rightarrow \mathrm{C}$$

where Cg differs from C only in the addition of a few syntactic constructs. This can be viewed as an instance of a more general paradigm:

$$\mathcal{A}^{+} \rightarrow \mathcal{A}$$

There are many other forms of variant translators. Some possibilities are:

- the deletion of features in order to subset a language
- the addition of monitoring code, written in the target language
- the insertion of termination code to output monitoring data
- the insertion of initialization code to incorporate additional run-time facilities
- the insertion of code for debugging and checking purposes [4,5]

Such translations can be characterized by

$$\mathcal{A}^{-} \rightarrow \mathcal{A}$$

and

$$\mathcal{A} \rightarrow \mathcal{A}$$

In the latter case, the input text and the output text may be different, but they are both in $\mathcal{A}$. In both cases, the output of the variant translator can be processed by a standard translator for the target language $\mathcal{A}$.

One way to implement a variant language is to modify a standard translator, avoiding the preprocessor. This approach may or may not be easy, depending on the translator. In general, it involves modifying the code generator, which often is tricky and error prone. Furthermore, if the variant is a trial one, the effort involved may discourage experiments.

The standard way to produce variant translators is the one that is most often used for preprocessors in general, including ones that do not fit the variant translator paradigm — writing a stand-alone program in any convenient language. In the case of Ratfor, the preprocessor is written in Ratfor, providing the advantages of bootstrapping.

This approach presents several problems. In the first place, writing a complete, efficient, and correct preprocessor is a substantial undertaking. In experimental work, this effort may be unwarranted, and it is common to write the preprocessor in a high-level language, handling only the variant portion of the syntax, leaving the detection of errors to the final translator. Such preprocessors have the virtue of being easy to produce, but they often are slow, frequently unfaithful to the source language, and the failure to parse the input language completely may lead to mysterious results when errors are detected, out of context, by the final translator.

Modern tools such as Lex [6] and Yacc [7], that operate on grammatical specifications, have made the production of compilers (and hence translators in general) comparatively easy and have removed many of the sources of error that are commonly found in hand-tailored translators. Nonetheless, the construction of a translator for a large and complicated language is still a substantial undertaking.

If, however, a translator already exists for a language that is based on the use of such tools, it may be easy to produce a variant translator that is efficient and demonstrably correct by modifying grammatical specifications. The key is the use of these tools to produce a source-to-source translator, rather than producing a source-to-object translator. This technique was used in Cg. An existing Yacc specification for the C compiler was modified to generate C source code instead of object code. The idea is a simple one, but it has considerable utility and can be applied to a wide range of situations.

This report describes a system that uses this approach for the construction of variant translators for Icon. This system runs on the VAX[1] under UNIX[2]. The reader should have a general knowledge of Icon, Yacc, C, and UNIX.

2. Overview of Variant Translators for Icon

The heart of the system for constructing variant translators for Icon consists of an "identity translator" in which the semantic actions in a Yacc grammar echo the input text. The output of this identity translator differs from its input only in the arrangement of nonsemantic "white space" and in the insertion of semicolons between expressions, which are optional in some places in Icon programs. The identity translator corresponds to Version 5.8 of Icon with experimental extensions [8].

The semantic actions are cast as macro definitions, abstracting the format of the output from the grammar itself. A set of standard macro definitions for echoing the input is included in the parser generated by Yacc. Support software allows macro definitions to be changed via specification files, minimizing the clerical work needed to vary the format of the output. There also is a provision for including user functions in the parser, so that more complicated operations can be written in C. Finally, the grammar for the identity translator can be modified in order to make structural changes in the syntax.

The following sections describe this system in more detail and include a number of examples of its use.

3. The Grammar for the Icon Identity Translator

The grammar for the identity translator is listed in Appendix A. Many variant translators can be constructed without modifying this grammar, and minor modifications can be made to it without a detailed knowledge of its structure. Knowledge of a few aspects of this grammar are important, however, to understanding the translation process.

The grammar consists of two main parts: declaration syntax and expression syntax. The semantic actions for declarations output text. For example, the rule for the declaration of global identifiers is

```
GLOBAL idlist { Global($2) } ;
```

where GLOBAL is the token for the reserved word global and idlist is the nonterminal symbol for an identifier list. The definition of the macro Global(x) is

```
#define Global(x) printf("global %s\n", x)
```

The semantic actions for expressions construct text but do not output it. For example, the rule for

[1]VAX is a trademark of Digital Equipment Corporation
[2]UNIX is a trademark of Bell Laboratories

```
        while expr_1 do expr_2
```

is

```
        WHILE expr DO expr {$$ = While2($2, $4),} ,
```

The macro While2(x, y) produces the concatenation of "while ", x, " do ", and y

The rules and the definitions that construct and output strings are provided as part of the identity translator When a variant translator is constructed, changes are necessary only in situations in which the input is not to be echoed in the output

Deletions from the standard syntax can be accomplished by changing macro definitions to produce error messages instead of output text It is generally better, however, to delete rules from the grammar so that all syntactic errors in the input are handled in the same way, by Yacc

Modifications and additions to the standard grammar require a more thorough understanding of the structure of the grammar Examples are given in Sec 10 1

4. Macro Definitions

The purpose of using macro calls in the semantic actions of the grammar is to separate the structure of the grammar from the format of the output and to allow the output format to be specified without modification of the grammar

The macro definitions for declarations are comparatively simple and consist of calls to printf, such as the one given above for global declarations The macro definitions for expressions produce strings, generally resulting from the concatenation of strings produced by other rules

In order to simplify the definition of macros, a specification format is provided Specifications are processed by a program that produces the actual definitions The specification for While2(x, y) is

```
        While2(x, y)     " while "     x               " do "      y
```

Tabs separate the components of the specification The first component is the prototype for the macro call, which may include optional arguments enclosed in parentheses as illustrated by the example above The remaining components are the strings to be concatenated Lines that begin with # or which are empty are treated as comments The specifications for the standard macro definitions provided with the identity translator are listed in Appendix B

Definitions can be changed by modifying the standard ones or by adding new definitions In the case of duplicate definitions the last one holds Definitions can be provided in several files, so variant definitions can be provided in a separate file that is processed after the standard definitions See Sec 8

Definitions can be deleted by providing a specification that consists only of a prototype for the call For example, the specification

```
        While2(x, y)
```

deletes the definition for While2(x, y) In order to delete a definition, the prototype given must be identical to the standard definition prototype For example,

```
        While2(y, z)
```

does not delete the definition for While2(x, y)

The usual reason for deleting a definition is to use a C function in place of a macro See Sec 6 for an example

4.1 Macros for Operators

As shown in Appendix A, a distinct macro name has been supplied for each operator Thus Blim(x, y) is the macro for a limitation expression

$expr_1$ \ $expr_2$

To avoid having to know the names of the macros for the operators, specifications allow the use of operator symbols in prototypes The symbols are automatically replaced by the appropriate names Thus

```
\(x, y)
```

can be used in a specification in place of

```
Blim(x, y)
```

In the case of unary operators, the parentheses are omitted Thus Uques(x), which is the macro for ?*expr*, can be specified as ?x

In most cases, all operators of the same kind are translated in the same way Since Icon has many operators, a generic form of specification is provided to allow the definition of all operators in a category to be given by a single specification In a specification, as string of the form <*type*> indicates a category of operators The categories are

<uop>	unary operators, except as follows
<ucs>	control structures in unary operator format
<bop>	binary operators, except as follows
<aop>	assignment operators
<bcs>	control structures in binary operator format

The category <ucs> consists only of | The category <bcs> consists of ?, |, and \

The division of operators into categories is based on their semantic properties For example, a preprocessor may translate all unary operators in the same way, but translate the repeated alternation control structure into a programmer-defined control operation [9]

Examples of the use of generic specifications are given in Appendix B For example, the specification for binary operators is

```
<bop>(x, y)         x            " <bop> "   y
```

This specification results in the definition for every binary operator +(x, y), −(x, y), and so on In such a specification, every occurrence of <bop> is replaced by the corresponding operator Note that blanks are necessary to separate the binary operator from its operands Otherwise,

```
i * *s
```

would be translated into

```
i**s
```

which is equivalent to

```
i ** s
```

5. String Handling

The allocation and deallocation of storage for strings that are produced in the translation process is handled automatically, but in some cases it is necessary to understand the protocol that is used

Strings come from three sources during translation strings produced by the lexical analyzer literal strings, and strings produced by semantic actions All semantic actions that produce strings (those for expressions) allocate storage using *malloc(2)* Concatenation is performed by the C function

```
cat(s1, s2,    , sn)
```

which takes an arbitrary number of arguments and returns a pointer to the concatenated result The function is

```
char *cat(strs)                             /* concatenate strings */
   int strs;
   {
   char *s, **cs, *ns, *p;
   int tlen, i, n, *argn;

   argn = &strs - 1;                        /* VAX-specific; not portable! */
   i = n = *argn;
   tlen = 0;
   for (cs = (char **)&strs; i--; tlen += strlen(*cs++));
   ns = p = malloc(tlen+1);
   for (cs = (char **)&strs; n--; cs++) {
      s = *cs;
      while (*p++ = *s++);
      p--;
      free(*cs);                            /* presumed to have been allocated */
      }
   return ns;
   }
```

Note that it is presumed that all arguments of cat have been allocated. Since all strings produced by cat are allocated, the only problems arise with literal strings and strings produced by the lexical analyzer.

The lexical analyzer produces tree nodes that have several fields, including the input line and column numbers of the tokens. The structure of these nodes usually is not of interest in producing variant translators, but see Section 10.2 for a use of the column and line numbers. The cases where the nodes that are produced by the lexical analyzer are of interest occur where strings are recognized for identifiers and literals — the tokens IDENT, STRINGLIT, INTLIT, REALIT, and CSETLIT. For such tokens, the macro Str(*token*) is used and supplies an allocated copy of the appropriate string. See Appendix B for examples. Any new definitions that involve these tokens must use Str in a similar fashion.

Literal strings must be copied into allocated storage. This is done by q(s) which is defined as:

```
#define q(x) strcpy(malloc(strlen(x)+1), x)
```

A call of this macro is provided automatically for any component of a macro specification that begins with a quotation mark. For example,

```
While2(x, y)     "while "     x          " do "     y
```

produces the definition

```
#define While2(x, y) cat(q("while "),x,q(" do "),y)
```

Note that the space allocated by q is freed by cat.

The strings that are written out by semantic actions in the declaration syntax are never used again and also are freed. This is done explicitly as part of the semantic actions. See Appendix A.

6. Parser Functions

In some cases, semantic actions may be too complicated to be represented conveniently by macros. The file ulibe.c is automatically included in the parser and can be used to provide functions that may be needed during parsing.

An example is the automatic provision of initialization code. The easiest way to handle this is to recognize the main procedure and treat it specially, writing out the initialization code when this procedure is encountered. Thus the semantic action that handles procedure declarations must perform different operations, depending on the procedure name.

A function is preferable to a macro definition in this case because of the amount of code involved The form of the rule for a procedure declaration is

```
proc  : prochead SEMICOL locals initial procbody END { Proc($1, $3, $4, $5),} ,
```

The first step is to delete the macro definition foi procedure declarations by the specification

```
Proc(x, y, z, w)
```

Then a C function by this name is added to ulibe c Such a function might have the form

```
char *Proc(x, y, z, w)
   char *x, *y, *z, *w,
   {
      if (strncmp(x, "procedure main(", 15) == 0) {

            write out any global, record, and link declarations

         printf("%s,\n%s%s", x, y, z),          /* declaration heading, locals,     */

            write out any code to be executed before user code

         printf("%s\n", w),                     /* procedure body */

            write out code to be executed after user code

         printf("end\n"),
         }
      else printf("%s,\n%s%s%send\n", x, y, z, w),
   }
```

The argument x, which is produced by a macro in the rule for prochead is always in a standard form with a single blank between the reserved word procedure and the procedure name, regardless of the form of the declaration in the input Similarly, semicolons and linefeeds are inserted by macro calls for the declarations and between the expressions in the procedure body See Appendix B

7. Modifying Lexical Components of the Translator

The lexical analyzer for Icon is written in C rather than in Lex in order to make it easier to perform semicolon insertion and other complicated tasks that occur during lexical analysis [10] Specification files are used to build portions of the lexical analyzer, making it easy to modify The three kinds of changes that are needed most often are the addition of new keywords, reserved words, and operators

The identity translator accepts any identifier as a keyword, leaving its resolution to subsequent processing by the Icon translator Nothing need be done to add a new keyword except for processing it properly in the variant translator See the examples in Sec 10 1

The specification file tokens contains a list of all reserved words and operator symbols Each symbol has associated flags that indicate whether it can begin or end an expression These flags are used for semicolon insertion

To add a new reserved word, insert it in proper alphabetical order in the list of reserved words in tokens and give it a new token name To add a new operator, insert it in the list of operators in tokens (order there is not important) and give it a new token name The new token names must be added to the grammar See Appendix A

The addition of a new operator also requires modifying the specification of a finite-state automaton, optab Its structure is straightforward

8. Building a Variant Translator

In order to build a variant translator, it first is necessary to modify Yacc, since the version of Yacc that normally is distributed with UNIX does not have enough memory size for itran.g. To build a version of Yacc with more memory, edit the Yacc source file dextern and change the definition of MEMSIZE in the HUGE section to

```
#define MEMSIZE 22000
```

and use

```
#define HUGE
```

in files. Then rebuild Yacc.

The files that comprise a variant translator are listed in Appendix C. Unless changes to the lexical analyzer are needed, at most three files need to be modified to produce a new translator:

itran.g	Yacc grammar for the translator
ltran.defs	variant macro definitions (initially empty)
ulibe.c	parser functions (initially empty)

The translator *Makefile* is listed in Appendix D. A *make* first builds a new parser, parse.c. There are 210 shift/reduce conflicts in the identity translator. All of these conflicts are resolved properly. More conflicts should be expected if additions are made to the grammar. Reduce/reduce conflicts usually indicate errors in the grammar. After parse.c is built, ltran.defs is added to the standard macro specifications given in Appendix B and processed to produce a definition file, tdefs.h, which is included along with ulibe.c when parse.c is compiled. Finally, all the components of the system are linked to produce itran, the variant translator.

Rebuilding parse.c is a time-consuming process and it is preferable to confine changes, when possible, to ltran.defs and ulibe.c.

If no changes have been made in the lexical analyzer, it is faster to use

```
make parser
```

which forces *make* to rebuild parse.c without rebuilding the the lexical analyzer, even if components of the lexical analyzer are out of date. Once parse.c is built, the translator itself can be made with another *make*.

Most of the errors that may occur in building a variant translator are obvious and easily fixed. Erroneous changes to the grammar, however, may be harder to detect and fix.

9. Using a Variant Translator

The translator, itran, takes an input file on the command line and translates it. The specification − indicates standard input. The output of itran is written to standard output. The translator accepts the same options for translation that Icon does. For example, the option −s causes the translator to work silently instead of listing procedures as they are translated. See *icont(1)* for details [11].

If a memory error occurs when a variant translator is run, the most likely cause is the freeing of an unallocated string. If this happens, check to be sure that all literal arguments to cat are copied by q. Also be sure that all identifiers and literals produced by the lexical analyzer are obtained via Str. Check the files ltran.defs and ulibe.c in particular.

10. Examples

10.1 List Scanning

One use of a variant translator is to support an experimental list-scanning facility for Icon [12]. In this facility, string and list scanning are fused and modeled by a procedure Cat in a run-time library. The specifications for variant macros for these operations are

||(x, y)	"Cat("	x	","	y	")"
|||(x, y)	"Cat("	x	","	y	")"
||:=(x, y)	x	" := Cat("	x	","	y	")"
|||:=(x, y)	x	" := Cat("	x	","	y	")"

The built-in list and string scanning operations must be available also, since the run-time library is written using the string scanning facility itself The operators normally used for transmission and remaindering are used instead:

@(x, y)	x	" || "	y
%(x, y)	x	" ||| "	y
@:=(x, y)	x	" ||:= "	y
%:=(x, y)	x	" |||:= "	y

An alternative to sacrificing the transmission and remaindering operations would be to add new operator symbols to the lexical analyzer

The list-scanning facility also replaces the built-in scanning expression by a programmer-defined control operation

?(x, y)	"Scan{"	x	","	y	"}"	
?:=(x, y)	x	" := Scan{"	x	","	y	"}"

A new binary operation, *expr*$_1$! *expr*$_2$, is included in the list-scanning facility Its addition requires changing the grammar The new binary operation, which uses an operator that already exists for element generator in unary form, has the highest infix operator precedence — the same as field references — and is added to the grammar at that place:

```
                    .
                    .
                    .
| expr11 DOT IDENT {$$ = Field($1, $3);} ;
| expr11 BANG expr10 {$$ = Bbang($1, $3);} ;
| CONJUNC FAIL {$$ = Kfail;} ,
| CONJUNC IDENT {$$ = Keyword($2);} ;
                    .
                    .
                    .
```

The new operator corresponds to subscripting in standard Icon and has the macro specification

Bbang(x, y)	x	"["	y	"]"

The list-scanning facility also adds a new syntactic construction for explicitly constructed sets:

{*expr*$_1$, *expr*$_2$, . , *expr*$_n$}

A rule for this construction is added to the grammar in the section that handles expressions enclosed in braces and brackets·

```
                    .
                    .
                    .
| LPAREN exprlist RPAREN {$$ = Paren($2);} ;
| LBRACE compound RBRACE {$$ = Brace($2),} ;
| LBRACE setlist RBRACE {$$ = Set($2);} ;
| LBRACK exprlist RBRACK {$$ = Bracket($2);} ,
                    .
                    .
                    .
```

The rule for setlist is

```
setlist       : exprlist COMMA nexpr {$$ = Exprlist($1, $3);} ;
```

Note that these rules require at least two expressions in a set The construction

{expr}

is a compound expression. A set expression is translated into a call on a run-time library function that constructs a set from a list of values. The macro specification is:

```
Set(x)          "set(["      x          "])"
```

The list-scanning facility also illustrates the handling of keywords in a variant translator. Two standard keywords, &pos and &subject, are translated into Pos and Subject, respectively. There are also two keywords that are not in standard Icon: &element and &visit, which are translated into Subject[Pos] and Visit(), respectively. The translation of keywords is handled by removing the macro definition

```
Keyword(x)
```

and adding the following function to ulibe.c:

```
char *Keyword(x)
   char *x;
   {
   x = Str(x);
   if (strcmp(x, "element") == 0) return q("Subject[Pos]");
   else if (strcmp(x, "visit") == 0) return q("Visit()");
   else if (strcmp(x, "subject") == 0) return q("Subject");
   else if (strcmp(x, "pos") == 0) return q("Pos");
   else return cat(q("&"), x);
   }
```

Note the use of Str to obtain the string for the keyword.

10.2 A Cinematic Display of Pattern Matching

A variant translator also is used in a facility for displaying the details of the pattern-matching process as it takes place [13]. The translation of string-scanning expressions is similar to that for the list-scanning facility. In the display of pattern matching, however, scanning operators are highlighted on the terminal screen in order to show which scanning operation is presently being evaluated. Highlighting requires knowing the position of each scanning operator in the program. The rules for scanning expressions in the grammar for the identity translator are:

```
                          :
        | expr1a QMARK expr1 {$$ = Bques($1, $3);} ;
                          :
        | expr2 SCANASGN expr1 {$$ = Baugques($1, $3);} ;
                          :
```

The nodes produced by the lexical analyzer for the tokens QMARK and SACNASGN contain the necessary column and line information in COL and LINE fields. However, these tokens are not included in the macro calls[1]. It therefore is necessary to use other macros:

[1]The identity translator would be more general if all tokens were included as arguments in all macros. This would considerably complicate the handling of the most frequent kinds of translations, however.

```
                                   :
          | expr1a QMARK expr1 {$$ = Scan($1, $2, $3),} ,
                                   :
          | expr2 SCANASGN expr1 {$$ = Scana($1, $2, $3),} ,
                                   :
```

The specifications for these macros are

```
Scan(x, y, z)     locer(y)  "Scan(create " x        ", create "      z          ")}"
Scana(x, y, z)    locer(y)  x        " := "   "Scan(create " x      ", create "z      ")}"
```

where locer is included in ulibe.c

```
char *locer(x)
   {
   char locbuf[30],
   sprintf(locbuf, "{Loc := [%d, %d],", COL(x), LINE(x)),
   return q(locbuf);
   }
```

Thus the translation for

$expr_1$? $expr_2$

is

{Loc := [*i*, *j*], Scan(create $expr_1$, create $expr_2$)}

where *i* and *j* are the column and line numbers of the ? operator in the input program Loc is a global variable, and the run-time library procedure Scan uses the values in Loc to highlight the ? operator

11. Conclusions

The system described here for producing variant translators for Icon has been used successfully to provide support for a number of language variants and tools These include the list scanning facility mentioned in Sec 10 1, the cinematic display of pattern matching mentioned in Sec 10 2, an experimental language for manipulating sequences [14,15], an Icon program formatter, and a tool for monitoring expression evaluation events

The value of being able to construct a variant translator quickly and easily is best illustrated by the tool for monitoring expression evaluation events This translator copies input to output, inserting calls on procedures that tally expression activations, the production of results, and expression resumptions A similar system was built for Version 2 of Icon [16] and used to analyze the performance and behavior of generators In that case, the code generator and run-time system were modified extensively This involved weeks of tedious and difficult work that required expert knowledge of the internal structure of the Version 2 system The variant translator for Version 5 was written in a few hours, and required only a knowledge of the format of variant macro specifications and the Icon source language itself The monitoring of expression evaluation events in Version 5 probably would not have been done if it had been necessary to modify the code generator and the run-time system

Although the system described in this report is specifically tailored to Icon, the techniques have much broader applicability The automatic generation of such systems from grammatical specifications is an interesting project

Acknowledgements

Tim Budd's Cg preprocessor was the inspiration for the Icon variant translator system described here Bill Mitchell assisted in adapting the standard Icon translator to its use here Tim Budd, Dave Hanson, Bill Mitchell, and Steve Wampler made a number of helpful suggestions on the variant translator system and the presentation of the material in this report

References

1 Kernighan, Brian W "RATFOR — A Preprocessor for a Rational Fortran", *Software — Practice and Experience*, Vol 5 (1975), pp 395-406

2 Budd, Timothy A "An Implementation of Generators in C", *Computer Languages*, Vol 7 (1982), pp 69-87

3 Griswold, Ralph E and Madge T Griswold *The Icon Programming Language*, Prentice-Hall, Inc, Englewood Cliffs, New Jersey 1983

4 Steffen, J L "Ctrace — A Portable Debugger for C Programs", *UNICOM Conference Proceedings*, San Diego, California, January 1983 pp 187-191

5 Kendall, Samuel C "Bcc Runtime Checking for C Programs", *USENIX Software Tools Summer 1983 Toronto Conference Proceedings*, 1983, pp 5-16

6 Lesk, M E and E Schmidt *Lex — A Lexical Analyzer Generator*, Bell Laboratories, Murray Hill, New Jersey 1979

7 Johnson, S C *Yacc Yet Another Compiler-Compiler*, Bell Laboratories, Murray Hill, New Jersey 1978

8 Griswold, Ralph E and William H Mitchell *Experimental Extensions to Version 5 8 of Icon*, technical report, Department of Computer Science, The University of Arizona 1983

9 Griswold, Ralph E and Michael Novak "Programmer-Defined Control Operations", *The Computer Journal*, Vol 26, No 2 (May 1983), pp 175-183

10 Griswold, Ralph E, William H Mitchell, and Stephen B Wampler *The C Implementation of Icon, A Tour Through Version 5*, Technical Report TR 83-11a, Department of Computer Science, The University of Arizona 1983

11 Griswold, Ralph E and William H Mitchell *Icont(1)*, manual page for *UNIX Programmer's Manual*, Department of Computer Science, The University of Arizona 1983

12 Anderson Allan J and Ralph E Griswold *Unifying List and String Processing in Icon*, Technical Report TR 83-4, Department of Computer Science, The University of Arizona 1983

13 Griswold, Ralph E *Understanding Pattern Matching — A Cinematic Display of String Scanning* Technical Report TR 83-14 Department of Computer Science, The University of Arizona 1983

14 Griswold, Ralph E *Seque An Experimental Language for Manipulating Sequences*, Technical Report TR 83-16, Department of Computer Science, The University of Arizona 1983

15 Griswold, Ralph E *The Implementation of an Experimental Language for Manipulating Sequences* Technical Report TR 83-20, Department of Computer Science, The University of Arizona 1983

16 Coutant, Cary A, Ralph E Griswold, and David R Hanson "Measuring the Performance and Behavior of Icon Programs", *IEEE Transactions on Software Engineering*, Vol SE-9, No 1 (January 1983), pp 93-103

Appendix A — Grammar for the Icon Identity Translator

```
/* Identity Translator, Version 5.8 of Icon with Experimental Extensions */

/* primitive tokens */

%token      CSETLIT
            EOFX
            IDENT
            INTLIT
            REALLIT
            STRINGLIT

/* reserved words */

%token      BREAK                   /* break */
            BY                      /* by */
            CASE                    /* case */
            CREATE                  /* create */
            DEFAULT                 /* default */
            DO                      /* do */
            DYNAMIC                 /* dynamic */
            ELSE                    /* else */
            END                     /* end */
            EVERY                   /* every */
            EXTERNAL                /* external */
            FAIL                    /* fail */
            GLOBAL                  /* global */
            IF                      /* if */
            INITIAL                 /* initial */
            LINK                    /* link */
            LOCAL                   /* link */
            NEXT                    /* next */
            NOT                     /* not */
            OF                      /* of */
            PROCEDURE               /* procedure */
            RECORD                  /* record */
            REPEAT                  /* repeat */
            RETURN                  /* return */
            STATIC                  /* static */
            SUSPEND                 /* suspend */
            THEN                    /* then */
            TO                      /* to */
            UNTIL                   /* until */
            WHILE                   /* while */

/* operators */

%token      ASSIGN                  /* := */
            AT                      /* @ */
            AUGACT                  /* @:= */
            AUGAND                  /* & = */
            AUGEQ                   /* =:= */
            AUGEQV                  /* ===:= */
            AUGGE                   /* >=:= */
            AUGGT                   /* >:= */
            AUGLE                   /* <=:= */
            AUGLT                   /* < = */
            AUGNE                   /* ~=:= */
            AUGNEQV                 /* ~=== = */
```

```
AUGSEQ          /* ==:= */
AUGSGE          /* >>=:= */
AUGSGT          /* >>:= */
AUGSLE          /* <<=:= */
AUGSLT          /* <<:= */
AUGSNE          /* ~==:= */
BACKSLASH       /* \ */
BANG            /* ! */
BAR             /* | */
CARET           /* ^ */
CARETASGN       /* ^:= */
COLON           /* : */
COMMA           /* , */
CONCAT          /* || */
CONCATASGN      /* ||:= */
CONJUNC         /* & */
DIFF            /* -- */
DIFFASGN        /* --:= */
DOT             /* . */
EQUIV           /* === */
INTER           /* ** */
INTERASGN       /* **:= */
LBRACE          /* { */
LBRACK          /* [ */
LCONCAT         /* ||| */
LCONCATASGN     /* |||:= */
LEXEQ           /* == */
LEXGE           /* >>= */
LEXGT           /* >> */
LEXLE           /* <<= */
LEXLT           /* << */
LEXNE           /* ~== */
LPAREN          /* ( */
MCOLON          /* -: */
MINUS           /* - */
MINUSASGN       /* -:= */
MOD             /* % */
MODASGN         /* %:= */
NOTEQUIV        /* ~=== */
NUMEQ           /* = */
NUMGE           /* >e */
NUMGT           /* > */
NUMLE           /* <= */
NUMLT           /* > */
NUMNE           /* ~= */
PCOLON          /* +: */
PLUS            /* + */
PLUSASGN        /* +:= */
QMARK           /* ? */
RBRACE          /* } */
RBRACK          /* ] */
REVASSIGN       /* <- */
REVSWAP         /* <-> */
RPAREN          /* ( */
SCANASGN        /* ?:= */
SEMICOL         /* ; */
SLASH           /* / */
SLASHASGN       /* /:= */
STAR            /* * */
STARASGN        /* *:= */
SWAP            /* :=: */
TILDE           /* ~ */
UNION           /* ++ */
UNIONASGN       /* ++:= */
```

```
%{
#define q(x) strcpy(malloc(strlen(x)+1), x)
#define Str(x) q(STR0(x))
#include "tdefs h"
#include "itran h"
#include "sym h"
#include "tree h"
#define YYSTYPE nodeptr
#define YYMAXDEPTH 500
%}

%%

%{
%}

/*
 * This grammar is organized into the following sections.
 *
 *   declaration syntax
 *   expression syntax
 *   error handling
 *
 */

program       · decls EOFX ,

/* declaration syntax */

decls         . {$$ = Null,} ,
              | decls decl ,

decl          · record ,
              | proc ,
              | global ,
              | link ,

link          : LINK lnklist {
                        Link($2);
                        free($2),                /* free allocated space */
                        } ,

lnklist       . lnkfile ,
              | lnklist COMMA lnkfile {$$ = Linklist($1, $3),} ,

lnkfile       . IDENT {$$ = Linkident($1),} ,
              | STRINGLIT {$$ = Linkstring($1),} ,

global        : GLOBAL idlist {
                        Global($2),
                        free($2),                /* free allocated space */
                        } ,
              | EXTERNAL idlist {
                        External($2),
                        free($2),                /* free allocated space */
                        } ,

record        · RECORD IDENT LPAREN arglist RPAREN {
                        Record($2, $4),
                        free($4),                /* free allocated space */
                        loc_init(),              /* clear local symbol table */
                        } ,
proc           prochead SEMICOL locals initial procbody END {
                         Proc($1 $3, $4, $5),
                         free($1),                /* free allocated space */
                         free($3),
                         free($4),
                         free($5),
                         treeinit(),              /* clear tree space */
                         loc_init(),              /* clear local symbol table */
                         } ,

prochead       PROCEDURE IDENT LPAREN arglist RPAREN {$$ = Prochead($2, $4),}

arglist        {$$ = Null,} ,
             | idlist ,

idlist         IDENT {$$ = Ident($1),} ,
             | idlist COMMA IDENT {$$ = Idlist($1, $3),} ,

locals         {$$ = Null,} ,
             | locals retention idlist SEMICOL {$$ = Locals($1, $2, $3),} ,

retention      LOCAL {$$ = Local,} ,
             | STATIC {$$ = Static,} ,
             | DYNAMIC {$$ = Dynamic,} ,

initial        {$$ = Null,} ,
             | INITIAL expr SEMICOL {$$ = Initial($2),} ,

procbody       {$$ = Null,} ,
             | nexpr SEMICOL procbody {$$ = Procbody($1, $3),} ,

/* expression syntax */

nexpr          {$$ = Null,} ,
             | expr ,

expr           expr1a ,
             | expr CONJUNC expr1a {$$ = Bamper($1, $3),} ,

expr1a         expr1 ,
             | expr1a QMARK expr1 {$$ - Bques($1, $3),} ,
expr1            expr2 ,
               | expr2 SWAP expr1 {$$ = Bswap($1, $3),}
               | expr2 ASSIGN expr1 {$$ = Bassgn($1, $3),} ,
               | expr2 REVSWAP expr1 {$$ = Brswap($1, $3),} ,
               | expr2 REVASSIGN expr1 {$$ = Brassgn($1, $3),} ,
               | expr2 DIFFASGN expr1 {$$ = Bdiffa($1, $3),} ,
               | expr2 UNIONASGN expr1 {$$ = Buniona($1, $3),} ,
               | expr2 PLUSASGN expr1 {$$ = Bplusa($1, $3),} ,
               | expr2 MINUSASGN expr1 {$$ = Bminusa($1, $3),} ,
               | expr2 STARASGN expr1 {$$ = Bstara($1, $3),} ,
               | expr2 INTERASGN expr1 {$$ = Bintera($1, $3),} ,
               | expr2 SLASHASGN expr1 {$$ = Bslasha($1 $3),} ,
               | expr2 MODASGN expr1 {$$ = Bmoda($1, $3),} ,
               | expr2 CARETASGN expr1 {$$ = Bcareta($1, $3),} ,
               | expr2 AUGEQ expr1 {$$ = Bauqeq($1, $3),} ,
               | expr2 AUGEQV expr1 {$$ = Baugeqv($1, $3),} ,
               | expr2 AUGGE expr1 {$$ = Baugge($1, $3),} ,
               | expr2 AUGGT expr1 {$$ = Bauggt($1, $3),} ,
               | expr2 AUGLE expr1 {$$ = Baugle($1, $3),} ,
               | expr2 AUGLT expr1 {$$ = Bauglt($1, $3),} ,
               | expr2 AUGNE expr1 {$$ = Baugne($1, $3),} ,
               | expr2 AUGNEQV expr1 {$$ = Baugneqv($1, $3),} ,
               | expr2 AUGSEQ expr1 {$$ = Baugseq($1, $3),} ,
               | expr2 AUGSGE expr1 {$$ = Baugsge($1, $3),} ,
               | expr2 AUGSGT expr1 {$$ = Baugsgt($1, $3),} ,
               | expr2 AUGSLE expr1 {$$ = Baugsle($1, $3),} ,
               | expr2 AUGSLT expr1 {$$ = Baugslt($1, $3),} ,
               | expr2 AUGSNE expr1 {$$ = Baugsne($1, $3),} ,
               | expr2 CONCATASGN expr1 {$$ = Baugcat($1, $3),} ,
               | expr2 LCONCATASGN expr1 {$$ = Bauglcat($1, $3),} ,
               | expr2 SCANASGN expr1 {$$ = Baugques($1, $3),} ,
               | expr2 AUGAND expr1 {$$ = Baugamper($1, $3),} ,
               | expr2 AUGACT expr1 {$$ = Baugact($1, $3),} ,

expr2            expr3 ,
               | expr2 TO expr3 {$$ = To2($1, $3),} ,
               | expr2 TO expr3 BY expr3 {$$   To3($1, $3, $5),} ,

expr3            expr4 ,
               | expr4 BAR expr3 {$$ = Alt($1, $3),} ,

expr4            expr5 ,
               | expr4 LEXEQ expr5 {$$ = Bseq($1, $3),} ,
               | expr4 LEXGE expr5 {$$ = Bsge($1, $3),} ,
               | expr4 LEXGT expr5 {$$ = Bsgt($1, $3),} ,
               | expr4 LEXLE expr5 {$$ = Bsle($1, $3),} ,
               | expr4 LEXLT expr5 {$$ = Bslt($1, $3),} ,
               | expr4 LEXNE expr5 {$$ = Bsne($1, $3),} ,
               | expr4 NUMEQ expr5 {$$ = Beq($1, $3),} ,
               | expr4 NUMGE expr5 {$$ = Bge($1, $3),} ,
               | expr4 NUMGT expr5 {$$ = Bgt($1 $3),} ,
               | expr4 NUMLE expr5 {$$ = Ble($1, $3),} ,
               | expr4 NUMLT expr5 {$$ = Blt($1, $3),} ,
               | expr4 NUMNE expr5 {$$ = Bne($1, $3),} ,
               | expr4 EQUIV expr5 {$$ = Beqv($1, $3),} ,
               | expr4 NOTEQUIV expr5 {$$ = Bneqv($1, $3),}

expr5            expr6 ,
               | expr5 CONCAT expr6 {$$ = Bcat($1, $3),} ,
               | expr5 LCONCAT expr6 {$$ = Blcat($1, $3),} ,

expr6            expr7 ,
               | expr6 PLUS expr7 {$$ = Bplus($1, $3),} ,
               | expr6 DIFF expr7 {$$ = Bdiff($1, $3),} ,
               | expr6 UNION expr7 {$$ = Bunion($1, $3),} ,
               | expr6 MINUS expr7 {$$ = Bminus($1, $3),} ,
expr7       expr8 ,
          | expr7 STAR expr8 {$$ = Bstar($1, $3),} ,
          | expr7 INTER expr8 {$$ = Binter($1, $3),} ,
          | expr7 SLASH expr8 {$$ = Bslash($1, $3),} ,
          | expr7 MOD expr8 {$$ = Bmod($1, $3),} ,

expr8       expr9 ,
          | expr9 CARET expr8 {$$ - Bcaret($1, $3),} ,

expr9     . expr10 ,
          | expr9 BACKSLASH expr10 {$$ = Blim($1, $3),} ,
          | expr9 AT expr10 {$$ = Bact($1, $3),} ,

expr10    · expr11 ,
          | AT expr10 {$$ = Uat($2),} ,
          | NOT expr10 {$$ = Not($2),} ,
          | BAR expr10 {$$ = Ubar($2),} ,
          | CONCAT expr10 {$$ = Ubar(Ubar($2)),} ,
          | LCONCAT expr10 {$$ = Ubar(Ubar(Ubar($2))),} ,
          | DOT expr10 {$$ = Udot($2),} ,
          | BANG expr10 {$$ = Ubang($2),} ,
          | DIFF expr10 {$$ = Uminus(Uminus($2)),} ,
          | PLUS expr10 {$$ = Uplus($2),} ,
          | STAR expr10 {$$ = Ustar($2),} ,
          | SLASH expr10 {$$ - Uslash($2),} ,
          | CARET expr10 {$$ = Ucaret($2),} ,
          | INTER expr10 {$$ = Ustar(Ustar($2)),} ,
          | TILDE expr10 {$$ = Utilde($2),} ,
          | MINUS expr10 {$$ - Uminus($2),} ,
          | NUMEQ expr10 {$$ = Ueq($2),} ,
          | NUMNE expr10 {$$ = Utilde(Ueq($2)),} ,
          | LEXEQ expr10 {$$ = Ueq(Ueq($2)),} ,
          | LEXNE expr10 {$$ = Utilde(Ueq(Ueq($2))),} ,
          | EQUIV expr10 {$$ = Ueq(Ueq(Ueq($2))),} ,
          | UNION expr10 {$$ = Uplus(Uplus($2)),} ,
          | QMARK expr10 {$$ = Uques($2),} ,
          | NOTEQUIV expr10 {$$ = Utilde(Ueq(Ueq(Ueq($2)))),} ,
          | BACKSLASH expr10 {$$ = Ubacksl($2),} ,

expr11      literal ,
          | section ,
          | return ,
          | if ,
          | case ,
          | while ,
          | until ,
          | every ,
          | repeat ,
          | CREATE expr {$$ = Create($2),} ,
          | IDENT {$$ = Ident($1),} ,
          | NEXT {$$ = Next,} ,
          | BREAK nexpr {$$ = Break($2),} ,
          | LPAREN exprlist RPAREN {$$ - Paren($2),} ,
          | LBRACE compound RBRACE {$$ = Brace($2),} ,
          | LBRACK exprlist RBRACK {$$ = Bracket($2),} ,
          | expr11 LBRACK expr RBRACK {$$ = Subscr($1, $3),} ,
          | expr11 LBRACE exprlist RBRACE {$$ = Pdco($1, $3),} ,
          | expr11 LPAREN exprlist RPAREN {$$ = Invoke($1, $3),} ,
          | expr11 DOT IDENT {$$ = Field($1, $3),} ,
          | CONJUNC FAIL {$$ = Kfail,} ,
          | CONJUNC IDENT {$$ = Keyword($2),} ,

while       WHILE expr {$$ = While1($2),} ,
          | WHILE expr DO expr {$$ - While2($2, $4),} ,
until          UNTIL expr {$$ = Until1($2),} ,
             | UNTIL expr DO expr {$$ = Until2($2, $4),} ,

every          EVERY expr {$$ = Every1($2),} ,
             | EVERY expr DO expr {$$ = Every2($2, $4),} ,

repeat         REPEAT expr {$$ = Repeat($2),} ,

return         FAIL {$$ = Fail,}
             | RETURN nexpr {$$ = Return($2),} ,
             | SUSPEND nexpr {$$ = Suspend($2),} ,

if             IF expr THEN expr {$$ = If2($2, $4),} ,
             | IF expr THEN expr ELSE expr {$$ = If3($2, $4, $6),} ,

case           CASE expr OF LBRACE caselist RBRACE {$$ = Case($2, $5),} ,

caselist       cclause ,
             | caselist SEMICOL cclause {$$ = Clist($1, $3),} ,

cclause        DEFAULT COLON expr {$$ = Default($3),} ,
             | expr COLON expr {$$ = Cclause($1, $3),} ,

exprlist       nexpr {Null,} ,
             | exprlist COMMA nexpr {$$ = Exprlist($1, $3),} ,

literal        INTLIT {$$ = Iliter($1),} ,
             | REALLIT {$$ = Rliter($1),} ,
             | STRINGLIT {$$ = Sliter($1),} ,
             | CSETLIT {$$ = Cliter($1),} ,

section        expr11 LBRACK expr COLON expr RBRACK {$$ = Sect($1, $3, $5),} ,
             | expr11 LBRACK expr PCOLON expr RBRACK {$$ = Psect($1, $3, $5),} ,
             | expr11 LBRACK expr MCOLON expr RBRACK {$$ = Msect($1, $3, $5),} ,

compound       nexpr ,
             | nexpr SEMICOL compound {$$ = Semi($1, $3),} ,

/* error handling */

program        error decls EOFX ,
proc           prochead error procbody END ,
expr           error ,
%%

/* C functions used by the parser */

#include "cater c"           /* string concatenation */
#include "ulibe c"           /* auxiliary functions */
```

Appendix B — Specifications for the Standard Macros

```
#  Declaration Syntax
#
#       declarations
#
External(x)     printf("external %s\n", x)
Global(x)       printf("global %s\n", x)
Link(x)         printf("link %s\n", x)
Proc(x, y, z, w) printf("%s,\n%s%s%send\n", x, y, z, w)
Record(x, y)    printf("record %s(%s)\n", Str(x), y)

#
#       syntax subsidiary to declarations
#
Dynamic         "dynamic "
Initial(x)      "initial "      x               ",\n"
Linklist(x, y)  x               ", "            y
Linkident(x)    Str(x)
Linkstring(x)   "\""            Str(x)          "\""
Local           "local "
Locals(x, y, z) x               y               z               ",\n"
Procbody(x, y)  x               ",\n"           y
Prochead(x, y)  "procedure "    Str(x)          "("             y               ")"
Static          "static "

#
#  Expression Syntax
#
#       elements
#
Cliter(x)       "'"             Str(x)          "'"
Ident(x)        Str(x)
Idlist(x, y)    x               ", "            Str(y)
Iliter(x)       Str(x)
Keyword(x)      "&"             Str(x)
Kfail           "&fail"
Null            ""
Rliter(x)       Str(x)
Sliter(x)       "\""            Str(x)          "\""
```

```
#
#       reserved-word syntax
#
Break(x)        "break "        x
Case(x, y)      "case "         x               " of {\n"       y               "\n}"
Cclause(x, y)   x               ":"             y
Clist(x, y)     x               ";\n"           y
Create(x)       "create "       x
Default(x)      "default:"      x
Every1(x)       "every "        x
Every2(x, y)    "every "        x               " do "          y
Fail            "fail"
If2(x, y)       "if "           x               " then "        y
If3(x, y, z)    "if "           x               " then "        y               " else "        z
Next            "next "
Not(x)          "not "          x
Repeat(x)       "repeat "       x
Return(x)       "return "       x
Suspend(x)      "suspend "      x
To2(x, y)       x               " to "          y
To3(x, y, z)    x               " to "          y               " by "          z
Until1(x)       "until "        x
Until2(x, y)    "until "        x               " do "          y
While1(x)       "while "        x
While2(x, y)    "while "        x               " do "          y

#
#       operator syntax
#
#       binary operators
#
<bop>(x, y)     x               " <bop> "       y
<aop>(x, y)     x               " <aop> "       y
<bcs>(x, y)     x               " <bcs> "       y

#
#       unary operators
#
<uop>x          "<uop>"         x
<ucs>x          "<ucs>"         x

#
#       miscellaneous expressions
#
Brace(x)        "{\n"           x               "\n}"
Bracket(x)      "["             x               "]"
Exprlist(x, y)  x               ", "            y
Field(x, y)     x               "."             Str(y)
Invoke(x, y)    x               "("             y               ")"
Msect(x, y, z)  x               "["             y               "-:"            z               "]"
Paren(x)        "("             x               ")"
Pdco(x, y)      x               "{"             y               "}"
Psect(x, y, z)  x               "["             y               "+:"            z               "]"
Sect(x, y, z)   x               "["             y               ":"             z               "]"
Semi(x, y)      x               ";\n"           y
Subscr(x, y)    x               "["             y               "]"
```

Appendix C — Files for Building a Variant Translator

Bsyms	macro names for binary operators
Define	macro definition program
Define icn	source for Define
Makefile	construction of translator
Usyms	macro names for unary operators
cater c	string handling functions for the parser
char c	initialization for character classification
char h	character classification and transformation macros
code c	routines for traversing parse trees
code h	structures used by code c
err c	routines for producing error messages
itran c	main program that controls translation
itran defs	standard macro definitions for semantic actions
itran g	Yacc grammar for identity translator
itran h	external definitions used throughout the translator
lex c	routines for lexical analysis
lex h	structures and definitions used by the lexical analyzer
ltran defs	variant macro definitions for semantic actions
mem c	memory initialization and management
mktoktab	program to build optab c and toktab c
mktoktab icn	source for mktoktab
optab	specifications for operator recognition
optab c	state tables for operator recognition
pscript	edit script to modify parser produced by Yacc
sym c	routines for symbol table management
sym h	structures for symbol table entries
synerr h	initialization of tables to map error states to messages
tdefs h	macro definitions produced from itran defs and ltran defs
token h	token definitions generated by Yacc
tokens	token specifications
toktab c	*initialization of structures containing token information*
tree h	parse tree structures and accessing macros
ulibe c	parser functions for variant translators

Appendix D — Translator *Makefile*

```
CFLAGS = -O -w
GRAM = itran.g
TRAN = itran
PARSE = parse
DEFS = ltran.defs

$(TRAN):            itran.o $(PARSE).o lex.o sym.o mem.o \
                    err.o char.o optab.o toktab.o code.o
                    cc -o $(TRAN) itran.o $(PARSE).o lex.o sym.o mem.o \
                    err.o char.o optab.o toktab.o code.o

itran.o:            itran.h token.h tree.h sym.h
code.o:             itran.h token.h tree.h code.h sym.h
$(PARSE).o:         tdefs.h ulibe.c cater.c itran.h tree.h sym.h
lex.o:              itran.h token.h lex.h char.h tree.h
sym.o:              itran.h token.h sym.h char.h
mem.o:              itran.h sym.h
err.o:              itran.h token.h tree.h lex.h
char.o:             char.h
optab.o:            lex.h
toktab.o:           itran.h lex.h token.h

$(PARSE).c          token.h: $(GRAM)
                    yacc -d $(GRAM); : expect 210 shift/reduce conflicts
                    mv y.tab.c $(PARSE).c
                    ed $(PARSE).c <pscript
                    mv y.tab.h token.h

parser:             $(GRAM)
                    yacc $(GRAM)
                    mv y.tab.c $(PARSE).c
                    ed $(PARSE).c <pscript
                    touch token.h
                    touch itran.o
                    touch lex.o
                    touch sym.o
                    touch err.o
                    touch toktab.o

toktab.c            optab.c: tokens optab mktoktab
                    mktoktab

mktoktab:           mktoktab.icn
                    icont mktoktab.icn

Define:             Define.icn
                    icont Define.icn

Listall:
                    @pr *.h *.c $(GRAM)
                    @date >List

List:               char.h err.h lex.h sym.h \
                    token.h tree.h itran.h \
                    char.c err.c lex.c mem.c \
                    optab.c parse.c sym.c toktab.c itran.c $(GRAM)
                    @pr $?
                    @date >List

tdefs.h:            Define.icn itran.defs $(DEFS)
                    Define itran.defs $(DEFS) >tdefs.h

roff:               programs.roff
                    itroff programs.roff &
```