Building Variant Translators for Version 7 of Icon*

Ralph E. Griswold
Kenneth Walker

TR 88-8

January 25, 1988

Department of Computer Science

The University of Arizona

Tucson, Arizona 85721

*This work was supported by the National Science Foundation under Grant DCR-8502015.

Building Variant Translators for Version 7 of 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:

$$\text{Cg} \rightarrow \text{C}$$

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:

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

The term ‘‘translator’’ is used here in the general sense, and includes both source-to-source translators, such as preprocessors, and source-to-object translators, such as compilers. In practice, the application of a source-to-source translator (preprocessor) may be followed by the application of a source-to-object translator (compiler). The combination is, of course, also a translator.

The term ‘‘variant translator’’ is used here to refer to a translator that differs in its action, in some respect, from a standard one for a language. The applications described in this report relate to source-to-source translators, although the term ‘‘preprocessor’’ is too restrictive to describe all of them.

There are many uses for variant translators. Some of them are:

- the addition of syntactic constructions to produce a superset of a language, as in the case of Cg
- the deletion of features in order to subset a language
- the translation of one source language into another [4]
- 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 [5, 6]

Note that in several cases, the translations can be characterized by

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

The input text and the output text may be different, but they are both in $\mathcal{A}$. Both the input and 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 translator is to modify a standard source-to-object 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 an experiment, the effort involved may be prohibitive.

The standard way to produce a variant translator 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 [7] and Yacc [8], 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 under UNIX[1]. 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’’. 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 uses the same Yacc grammar as the regular Icon translator, but uses different semantic actions. These semantic actions are cast as macro definitions in the grammar, which are expanded before the grammar is translated by Yacc into a parser. One set of macros is supplied for the regular Icon translator and another set is supplied for the identity translator. The macros used by the regular Icon translator produce code suitable for the Icon linker. The macros used by the identity translator echo the input text, producing source-code output. In addition to the grammar, other code is shared between the two translators, insuring a high degree of consistency between the two systems.

A variant translator is created by first creating an identity translator and then modifying it. There is a shell script for producing identity translators and associated support software to simply the process of making modifications. This 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 Icon grammar 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.

There are two types of semantic actions. The semantic action for a declaration outputs text. The semantic action for a component of a declaration, such as an identifier list or an expression, assigns a string to the Yacc attribute for the component. Declarations are parsed by the production:

[1]UNIX is a trademark of AT&T Bell Laboratories.

```
decl        : record {Recdcl($1);} ;
            | proc {Procdcl($1);} ;
            | global {Globdcl($1);} ;
            | link {Linkdcl($1);} ;
```

The non-terminals record, proc, global, and link each produce a string and the corresponding macro Recdcl, Procdcl, Globdcl, or Linkdcl prints the string.

Because the grammar is used for both the regular Icon translator and the variant translator system, the macro calls must be more general than what is required for either one alone. Consider the production for global:

```
global      : GLOBAL {Global0($1);} idlist   {Global1($1, $2, $3);} ;
```

The macro Global0 is needed in the regular translator, but performs no operation in the identity translator. The macro Global1 does the work in the identity translator; it concatenates "global " with the string produced by idlist, and this new string becomes the result of this production. The macro Global1 is passed $1, $2, and $3 even though it only uses $3. This is done for generality.

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.

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 all the same. For example the definition of Global for the identity translator is:

```
#define Globdcl(x)      if (!nocode) treeprt(x); treeinit()
```

The variable nocode is set when an error is detected during parsing. This helps prevent the variant translator from generating a program with syntax errors. The reason for doing this is that the output of a variant translator is usually piped directly into the regular Icon translator. If syntax errors were propagated, two error messages would result: one from the variant translator and one from the Icon translator. The message from the variant translator is the one that is wanted because it references the line number of the original source whereas the message from the Icon translator references the line number of the generated source.

The function treeprt prints a string and the function treeinit reclaims storage. See the Section 5 for details of string representation.

4.1 Specifications for Macros

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. For example, the macro While1 is used in the rule

```
WHILE expr DO expr {While1($1,$2,$3,$4);} ;
```

A specification for this macro to produce an identity translation is:

```
While1(w,x,y,z)  "while "    x     " do "     z
```

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 with the result being assigned to the Yacc pseudo-variable $$.

Specification lines that begin with # or which are empty are treated as comments. A set of lines delineated by %{ and %} are copied unchanged. The ‘‘braces’’ %{ and %} must each occur alone on a separate line; these two delimiting lines are not copied. This feature allows the inclusion of actual macro definitions, as opposed to specifications, and the inclusion of C definitions. The standard macro definitions supplied for the identity translator include examples of these features. These definitions 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

```
While1()
```

deletes the definition for While1. This is a convenient way to insure a macro is undefined. It is usually used along with the copy feature to introduce macro definitions that cannot be generated by the specification system. For example, the following specifications eliminate reclamation of storage, preserving strings between declarations.

```
Globdcl()
Linkdcl()
Procdcl()
Recdcl()
%{
#define Globdcl(x)      if (!nocode) treeprt(x);
#define Linkdcl(x)      if (!nocode) treeprt(x);
#define Procdcl(x)      if (!nocode) treeprt(x);
#define Recdcl(x)       if (!nocode) treeprt(x);
%}
```

4.2 Macros for Icon Operators

As shown in Appendix A, there is a distinct macro name for each Icon operator. For example, Blim(x, y, z) is the macro for a limitation expression,

$expr_1 \ \backslash \ expr_2$

Note that the parameter y is the operator symbol itself. 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, z)
```

can be used in a specification in place of

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

Unary operators are similar. For example, Uqmark(x, y), which is the macro for ?*expr*, can be specified as ?(x, y). In this case the parameter x is the operator symbol.

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, a 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 \.

For example, the specification for binary operators for identity translations is

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

This specification results in the definition for every binary operator: +(x, y, z), −(x, y, z), and so on. In such a specification, every occurrence of <bop> is replaced by the corresponding operator symbol. 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
```

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].

5. String Handling

Strings are represented as binary trees in which the leaves contain pointers to C strings. The building of these trees can be thought of as doing string concatenation using lazy evaluation. The concatenation operation just creates a new root node with its two operands as subtrees. The real concatenation is only done when the strings are written out. Another view of this is that concatenation builds a list of strings with the list implemented as a binary tree. This view allows ‘‘strings’’ to be treated as a list of tokens. This approach is useful in more complicated situations where there is a need to distinguish more than just syntactic structures. For example, the head of the main procedure can be distinguished from the heads of other procedures by looking at the second string in the list for the procedure declaration.

Strings come from three sources during translation: strings produced by the lexical analyzer, literal strings, and strings produced by semantic actions. The lexical analyzer produces nodes. 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. These nodes contain pointers to the strings recognized. (The actual strings are stored in a string space and remain there throughout execution of the translator.) These nodes can be used directly as a tree (of one node) of strings. Other nodes produced by the lexical analyzer, for example those for operators, do not contain strings. However, all of these nodes contain line and column numbers referring to the location of the token in the source text. This line and column information can be useful in variant translators that need to produce output that contains position information from the input.

A literal string must be coerced into a tree of one node. This is done with the C function

```
q(s)
```

This is handled automatically when macros are produced from specifications. For example, the specification

```
Fail(x)                 "fail"
```

is translated into the macro

```
#define Fail(x)         $$ = q("fail")
```

Most semantic actions concatenate two or more strings and produce a string. They use the C function

```
cat(n, t1, t2, …, tn)
```

which takes a variable number of arguments and returns a pointer to the concatenated result. The first argument is the number of strings to be concatenated. The other arguments are the strings in tree format. The result is also in tree format.

As an example, the specification

```
While1(w, x, y, z)  "while "        x                  " do "          z
```

produces the definition

```
#define While1(w, x, y, z) $$ = cat(4, q("while "), x, q(" do "), z)
```

Another function, item(t, n), returns the nth node in the ‘‘list’’ t. For example, the name of a procedure is contained in the second node in the list for the procedure declaration (see Appendix A). Thus, if the procedure heading list is the value of head, item(head, 2) produces the procedure name.

There are three macros that produce values associated with a node. Str0 produces the string. For example, code conditional on the main procedure could be written as follows:

```
if (strcmp(Str0(item(head,2)),"main") == 0) {
                    .
                    .
                    .
   }
```

As this example illustrates, semantic actions may be too complicated to be represented conveniently by macros. In such cases parser functions can be used. A file is provided for such functions. See Section 10 for an example.

The macros Line and Col produce the source-file line number and column, respectively, of the place where the text for the node begins. The use of these attributes is illustrated in Section 10.

In some sophisticated applications, variant translators may need other capabilities that are available in the translator system. For example, if a function produces a string, it may be necessary place this string in a place that survives the function call. The Icon translator has a string allocation facility that can be used for this purpose: the free space begins at strfree and putident(n) installs a string of length n there. The use of such facilities requires more knowledge of the translator system than it is practical to provide here. Persons with special needs should study the translator in more detail.

6. 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. 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.

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.

7. Modifying Yacc

Before building a variant translator, it may be necessary to modify Yacc, since the version of Yacc that normally is distributed with UNIX does not provide enough space to process Icon’s grammar. To build a version of Yacc with more space, 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.

8. Building a Variant Translator

The steps for setting up the directory structure for a variant translator are:

- create a directory for the translator
- make that directory the current directory
- execute the shell script icon_vt supplied with Version 7 of Icon

For example, if the variant translator is to be in the directory xtran and Icon is installed in /usr/icon/v7, the following commands will build the variant translator:

```
mkdir xtran
cd xtran
/usr/icon/v7/icon_vt
```

The shell script icon_vt creates a number of files in the new directory and in two sub–directories: itran and h. 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:

variant.defs	variant macro definitions (initially empty)
variant.c	parser functions (initially empty)
itran/icon_g.c	Yacc grammar for Icon

The translator make file, itran/Makefile, is listed in Appendix D. The make file in the main translator directory just insures that the program define has be compiled and then does a make in the itran directory. Performing a make in the itran directory first combines variant.defs with the standard macro definitions (in ident.defs) and processes them to produce the definition file, itran/gdefs.h. The C preprocessor is then used to expand the macros in itran/icon_g.c using these definitions and the result, after some ‘‘house keeping’’, is put in itran/expanded.g. Next, Yacc uses the grammar in itran/expanded.g to build a new parser, parse.c. There are over 200 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. Finally, all the components of the system are compiled, including variant.c, and linked to produce vitran, the variant translator.

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. Error messages from Yacc or from compiling itran/parse.c refer to line numbers in itran/expanded.g. These errors must be related back to variant.defs or itran/icon_g.c by inspection of itran/expanded.g.

9. Using a Variant Translator

The translator, vitran, takes an input file on the command line and translates it. The specification – in place of an input file indicates standard input. The output of vitran is written to standard output. For example,

```
vitran pre.icn >post.icn
```

translates the file pre.icn and produces the output in post.icn. Assuming the variant translator produces Icon source language, post.icn can be translated into object code by

```
icont post.icn
```

where icont is the standard Icon command processor.

Variant translators accept the same options for translation that the standard Icon translator does. For example, the option –s causes the translator to work silently. See the manual page for icont for details [10].

10. An Example

As an example of the construction of a variant translator, consider the problem of monitoring string concatenation in Icon programs, writing out the size of each string constructed by concatenation. One way to do this, of course, is to modify Icon itself, adding the necessary monitoring code to the C function that performs concatenation. An alternative approach, which does not require changes to Icon itself, is to produce a variant translator that translates concatenation operations into calls of an Icon procedure, but leaves everything else unchanged:

$expr_1$ || $expr_2$ → Cat($expr_1$,$expr_2$)

The procedure Cat might have the form:

```
procedure Cat(s1,s2)
   write(&errout,"concatenation: ",*s1 + *s2," characters")
   return s1 || s2
end
```

Such a procedure could be added to a preprocessed program (Cat is not preprocessed itself) in order to produce the desired information when the program is run.

A single definition in variant.defs suffices:

```
||(x,y,z)        "Cat("        x        ","        z        ")"
```

Note, however, that Icon also has an augmented assignment operator for string concatenation:

$expr_1$ ||:= $expr_2$

This operation can be handled by the definition

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

Observe that this definition is not precisely faithful to the semantics of Icon, since it causes $expr_1$ to be evaluated twice, while $expr_1$ is evaluated only once in the true augmented assignment operation. This problem cannot be avoided here, since all arguments are passed by value in Icon, but in practice, this discrepancy is unlikely to cause problems.

In the application of such a monitoring facility, it may be useful to have a provision whereby concatenation can be performed without being monitored. This can be accomplished by adding an alternative operator symbol for concatenation, such as

$expr_1$! $expr_2$ → $expr_1$ || $expr_2$

Adding a new operator to the syntax of Icon requires modifying the grammar in itran/icon_g.c. Since this alternative concatenation operator should have the same precedence and associativity as the regular concatenation operator, it can be added to the definition of expr5 (see Appendix A):

```
expr5        : expr6 ;
             | expr5 CONCAT expr6 {Bcat($1,$2,$3);} ;
             | expr5 BANG expr6 {Bacat($1,$2,$3);} ;
             | expr5 LCONCAT expr6 {Blcat($1,$2,$3);} ;
```

where BANG is the token name for ! . Then the definition of Bacat can be added to variant.defs:

```
Bacat(x,y,z)        x        " || "        z
```

Such changes to icon_g.c usually increase the number of shift/reduce conflicts encountered by Yacc.

One difficulty with monitoring concatenation as described above is that the procedure Cat must be added to the translated program. This can be accomplished automatically by arranging to have the code for Cat written out when the variant translator encounters the main procedure. This is a case where a parser function, as mentioned in Section 5, is more appropriate than a macro definition.

The first step is to change the specifications. The definition for the macro, Proc1, that produces procedure declarations is replaced by a call to a parser function. The changes to variant.defs are:

```
%{
nodeptr proc();
%}
Proc1(u,v,w,x,y,z)            proc(u,w,x,y)
```

The C declaration for proc is included in the file expanded.g and subsequently incorporated by Yacc into parse.c where the call to proc is compiled. Note that proc returns a nodeptr.

The C function is placed in variant.c. It might have the form

```
#include "tran/tree.h"

nodeptr item(), cat(), q();

nodeptr proc(u, w, x, y)
nodeptr u, w, x, y;
   {
   static char *catproc = "procedure Cat(s1, s2)\n\
write(&errout,\"concatenation:  \", *s1 + *s2,\" characters\")\n\
return s1 || s2\n\
end\n";

   if (strcmp(Str0(item(u, 2)), "main") == 0)
      return cat(7, q(catproc), u, q(";\n"), w, x, y, q("end\n"));
   else
      return cat(6, u, q(";\n"), w, x, y, q("end\n"));
   }
```

Thus, when the main procedure is encountered, the text for Cat is written out before the text for the main procedure, but all other procedures are written out as they would be in the absence of this function.

One disadvantage of this way of providing the text for Cat is that the literal string is long, complicated, and difficult to change. In addition, it is necessary to rebuild the variant translator in order to change Cat. Since monitoring of this kind is likely to suggest changes to the format or nature of the data being written, it is useful to be able to change Cat more easily. One solution to this problem is to produce a link declaration for the file containing the translated procedure rather than the text of the procedure. With this change, the parser function might have the form

```
nodeptr proc(u, w, x, y)
nodeptr u, w, x, y;
   {

   if (strcmp(Str0(item(u, 2)), "main") == 0)
      return cat(7, q("link cat\n\n"), u, q(";\n"), w, x, y, q("end\n"));
   else
      return cat(6, u, q(";\n"), w, x, y, q("end\n"));
   }
```

The monitoring facility described above produces information about all string concatenation operations, but it is not possible to distinguish among them. It might be more useful to know the amount of concatenation performed by each concatenation operation. This can be done if the location of the operator in the source program can be identified. As mentioned in Section 5, tree nodes contain line and column information provided by the lexical analyzer. Thus, the translation for the concatenation operations could provide this addition information as extra arguments to Cat, which then could print out the locations along with information about the amount of concatenation.

```
procedure Cat(s1,s2,i,j)
   write(&errout,"concatenation: ",*s1 + *s2," characters at [",i,",",j,"]")
   return s1 || s2
end
```

The specifications for the translation of the concatenation operations might be changed to

```
%{
nodeptr proc(), Locargs();
%}
Proc1(u,v,w,x,y,z)              proc(u,w,x,y)
||(x,y,z)        "Cat("         x            ","         z          Locargs(y)   ")"
||:=(x,y,z)      x              " := Cat("   x          ","         z            Locargs(y)")"
Bacat(x,y,z)     x              " || "       z
```

where Locargs is a parser function that produces a string consisting of the line and column numbers between commas. This function might have the form

```
nodeptr Locargs(x)
nodeptr x;
{
        sprintf(strfree,",%d,%d",Col(x),Line(x));
        return q(putident(strlen(strfree)+1));
}
```

The C function sprintf is used to do the formatting, placing the resulting string in the translator's allocation region as mentioned in Section 5. The string is installed by putident; the additional character allows for the fact that such strings are stored as Icon strings, not C strings, and the null character terminating the C string must be included [11].

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 a list scanning facility [12], a animated display of pattern matching [13], An experimental language for manipulating sequences [14, 15], a SNOBOL4-like language with a syntax similar to Icon [4], an Icon program formatter, a tool for monitoring expression evaluation events, and a number of simpler tools.

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 was 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 7 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 7 probably would not have been undertaken if it had been necessary to modify the code generator and the run-time system.

The usefulness of the system described here depends heavily on its support software. The ability to specify macro definitions in a simple format, and particularly to be able to provide a single specification for the translation for all operators in a class, makes it easy to write many variant translators that otherwise would be impractically tedious.

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, Janalee O'Bagy, 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. B. W. Kernighan, ‘‘Ratfor — A Preprocessor for a Rational Fortran’’, *Software—Practice & Experience 5*(1975), 395-406.
2. T. A. Budd, ‘‘An Implementation of Generators in C’’, *J. Computer Lang. 7*(1982), 69-87.
3. R. E. Griswold and M. T. Griswold, *The Icon Programming Language*, Prentice-Hall, Inc., Englewood Cliffs, NJ, 1983.
4. R. E. Griswold, *Rebus — A SNOBOL4/Icon Hybrid*, The Univ. of Arizona Tech. Rep. 84-9, 1984.
5. J. L. Steffen, ‘‘Ctrace — A Portable Debugger for C Programs’’, *UNICOM Conference Proceedings*, Jan. 1983, 187-191. San Diego, California.
6. S. C. Kendall, ‘‘Bcc: Runtime Checking for C Programs’’, *USENIX Software Tools Summer 1983 Toronto Conference Proceedings*, 1983, 5-16.
7. M. E. Lesk and E. Schmidt, *Lex — A Lexical Analyzer Generator*, Bell Laboratories, Murray Hill, New Jersey, 1979.
8. S. C. Johnson, *Yacc: Yet Another Compiler-Compiler*, Bell Laboratories, Murray Hill, New Jersey, 1978.
9. R. E. Griswold and M. Novak, ‘‘Programmer-Defined Control Operations’’, *Computer J. 26*, 2 (May 1983), 175-183.
10. R. E. Griswold, *ICONT(1)*, manual page for *UNIX Programmer’s Manual*, The Univ. of Arizona Icon Project Document IPD19, 1988.
11. R. E. Griswold and M. T. Griswold, *The Implementation of the Icon Programming Language*, Princeton University Press, 1986.
12. A. J. Anderson and R. E. Griswold, *Unifying List and String Processing in Icon*, The Univ. of Arizona Tech. Rep. 83-4, 1983.
13. K. Walker and R. E. Griswold, *A Pattern-Matching Laboratory; Part I — An Animated Display of String Pattern Matching*, The Univ. of Arizona Tech. Rep. 86-1, 1986.
14. R. E. Griswold and J. O’Bagy, *Seque: A Language for Programming with Streams*, The Univ. of Arizona Tech. Rep. 85-2, 1985.
15. R. E. Griswold and J. O’Bagy, *Reference Manual for the Seque Programming Language*, The Univ. of Arizona Tech. Rep. 85-4, 1985.
16. C. A. Coutant, R. E. Griswold and D. R. Hanson, ‘‘Measuring the Performance and Behavior of Icon Programs’’, *IEEE Trans. on Software Eng. SE-9*, 1 (Jan. 1983), 93-103.

Appendix A — The Icon Grammar

```
/*
 * Grammar for Icon Version 7.
 *
 * NOTE: Any modifications of this grammar should be
 * propagated to any affected macro in gdefs.h.
 */

/* 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 */
          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           /* XXXX>=:= */
        AUGGT           /* XXXX>:= */
        AUGLE           /* XXXX<=:= */
        AUGLT           /* XXXX<:= */
        AUGNE           /* ~=:= */
        AUGNEQV         /* ~===:= */
        AUGSEQ          /* ==:= */
        AUGSGE          /* >XXXX>=:= */
        AUGSGT          /* >XXXX>:= */
        AUGSLE          /* <XXXX<=:= */
        AUGSLT          /* <XXXX<:= */
        AUGSNE          /* ~==:= */
        BACKSLASH       /* \ */
        BANG            /* ! */
        BAR             /* | */
        CARET           /* ^ */
        CARETASGN       /* ^:= */
        COLON           /* : */
        COMMA           /* , */
        CONCAT          /* || */
        CONCATASGN      /* ||:= */
        CONJUNC         /* & */
        DIFF            /* -- */
        DIFFASGN        /* --:= */
        DOT             /* . */
        EQUIV           /* === */
        INTER           /* ** */
        INTERASGN       /* **:= */
        LBRACE          /* { */
        LBRACK          /* [ */
        LCONCAT         /* ||| */
        LCONCATASGN     /* |||:= */
        LEXEQ           /* == */
        LEXGE           /* >XXXX>= */
        LEXGT           /* >> */
        LEXLE           /* <XXXX<= */
        LEXLT           /* << */
        LEXNE           /* ~== */
        LPAREN          /* ( */
        MCOLON          /* -: */
        MINUS           /* - */
        MINUSASGN       /* -:= */
        MOD             /* % */
        MODASGN         /* %:= */
        NOTEQUIV        /* ~=== */
        NUMEQ           /* = */
        NUMGE           /* XXXX>= */
        NUMGT           /* > */
```

```
        NUMLE           /* XXXX<= */
        NUMLT           /* < */
        NUMNE           /* ~= */
        PCOLON          /* +: */
        PLUS            /* + */
        PLUSASGN        /* +:= */
        QMARK           /* ? */
        RBRACE          /* } */
        RBRACK          /* ] */
        REVASSIGN       /* XXXX<- */
        REVSWAP         /* XXXX<-> */
        RPAREN          /* ) */
        SCANASGN        /* ?:= */
        SEMICOL         /* ; */
        SLASH           /* / */
        SLASHASGN       /* /:= */
        STAR            /* * */
        STARASGN        /* *:= */
        SWAP            /* :=: */
        TILDE           /* ~ */
        UNION           /* ++ */
        UNIONASGN       /* ++:= */
%{
** #include "itran.h"
** #include "sym.h"
** #include "tree.h"
** #include "../h/keyword.h"
** #define YYSTYPE nodeptr
** #define YYMAXDEPTH 500
#include "gdefs.h"
%}

%%

program : decls EOFX {Progend($1,$2);} ;

decls   : ;
        | decls decl ;

decl    : record {Recdcl($1);} ;
        | proc {Procdcl($1);} ;
        | global {Globdcl($1);} ;
        | link {Linkdcl($1);} ;

link    : LINK lnklist {Link($1, $2);} ;

lnklist : lnkfile    ;
        | lnklist COMMA lnkfile {Lnklist($1,$2,$3);} ;

lnkfile : IDENT {Lnkfile1($1);} ;
        | STRINGLIT {Lnkfile2($1);} ;

global  : GLOBAL {Global0($1);} idlist  {Global1($1, $2, $3);} ;

record  : RECORD {Record1($1);} IDENT LPAREN fldlist RPAREN {
                Record2($1,$2,$3,$4,$5,$6);
                } ;
```

```
fldlist      : {Arglist1();} ;
             | idlist {Arglist2($1);} ;

proc         : prochead SEMICOL locals initial procbody END {
                         Proc1($1, $2, $3, $4, $5, $6);
                         } ;

prochead: PROCEDURE {Prochead1($1);} IDENT LPAREN arglist RPAREN {
                         Prochead2($1, $2, $3, $4, $5, $6);
                         } ;

arglist      : {Arglist1();} ;
             | idlist {Arglist2($1);} ;
             | idlist LBRACK RBRACK {Arglist3($1);} ;

idlist       : IDENT {
                         Ident($1);
                         } ;
             | idlist COMMA IDENT {
                         Idlist($1, $2, $3);
                         } ;

locals       : {Locals1();} ;
             | locals retention idlist SEMICOL {Locals2($1, $2, $3, $4);} ;

retention: LOCAL {Local($1);} ;
             | STATIC {Static($1);} ;
             | DYNAMIC {Dynamic($1);} ;

initial      : {Initial1();} ;
             | INITIAL expr SEMICOL {Initial2($1, $2, $3);} ;

procbody: {Procbody1();} ;
             | nexpr SEMICOL procbody {Procbody2($1, $2, $3);} ;

nexpr        : {Nexpr();} ;
             | expr ;

expr         : expr1a      ;
             | expr CONJUNC expr1a{Bamper($1, $2, $3);} ;

expr1a       : expr1       ;
             | expr1a QMARK expr1  {Bques($1, $2, $3);} ;
```

```
expr1       : expr2 ;
            | expr2 SWAP expr1 {Bswap($1, $2, $3);} ;
            | expr2 ASSIGN expr1 {Bassgn($1, $2, $3);} ;
            | expr2 REVSWAP expr1 {Brswap($1, $2, $3);} ;
            | expr2 REVASSIGN expr1 {Brassgn($1, $2, $3);} ;
            | expr2 CONCATASGN expr1 {Baugcat($1, $2, $3);} ;
            | expr2 LCONCATASGN expr1 {Bauglcat($1, $2, $3);} ;
            | expr2 DIFFASGN expr1 {Bdiffa($1, $2, $3);} ;
            | expr2 UNIONASGN expr1 {Buniona($1, $2, $3);} ;
            | expr2 PLUSASGN expr1 {Bplusa($1, $2, $3);} ;
            | expr2 MINUSASGN expr1 {Bminusa($1, $2, $3);} ;
            | expr2 STARASGN expr1 {Bstara($1, $2, $3);} ;
            | expr2 INTERASGN expr1 {Bintera($1, $2, $3);} ;
            | expr2 SLASHASGN expr1 {Bslasha($1, $2, $3);} ;
            | expr2 MODASGN expr1 {Bmoda($1, $2, $3);} ;
            | expr2 CARETASGN expr1 {Bcareta($1, $2, $3);} ;
            | expr2 AUGEQ expr1 {Baugeq($1, $2, $3);} ;
            | expr2 AUGEQV expr1 {Baugeqv($1, $2, $3);} ;
            | expr2 AUGGE expr1 {Baugge($1, $2, $3);} ;
            | expr2 AUGGT expr1 {Bauggt($1, $2, $3);} ;
            | expr2 AUGLE expr1 {Baugle($1, $2, $3);} ;
            | expr2 AUGLT expr1 {Bauglt($1, $2, $3);} ;
            | expr2 AUGNE expr1 {Baugne($1, $2, $3);} ;
            | expr2 AUGNEQV expr1 {Baugneqv($1, $2, $3);} ;
            | expr2 AUGSEQ expr1 {Baugseq($1, $2, $3);} ;
            | expr2 AUGSGE expr1 {Baugsge($1, $2, $3);} ;
            | expr2 AUGSGT expr1 {Baugsgt($1, $2, $3);} ;
            | expr2 AUGSLE expr1 {Baugsle($1, $2, $3);} ;
            | expr2 AUGSLT expr1 {Baugslt($1, $2, $3);} ;
            | expr2 AUGSNE expr1 {Baugsne($1, $2, $3);} ;
            | expr2 SCANASGN expr1 {Baugques($1, $2, $3);} ;
            | expr2 AUGAND expr1 {Baugamper($1, $2, $3);} ;
            | expr2 AUGACT expr1 {Baugact($1, $2, $3);} ;

expr2       : expr3 ;
            | expr2 TO expr3 {To0($1, $2, $3);} ;
            | expr2 TO expr3 BY expr3 {To1($1, $2, $3, $4, $5);} ;

expr3       : expr4 ;
            | expr4 BAR expr3 {Alt($1, $2, $3);} ;
```

```
expr4       : expr5 ;
            | expr4 LEXEQ expr5 {Bseq($1, $2, $3);} ;
            | expr4 LEXGE expr5 {Bsge($1, $2, $3);} ;
            | expr4 LEXGT expr5 {Bsgt($1, $2, $3);} ;
            | expr4 LEXLE expr5 {Bsle($1, $2, $3);} ;
            | expr4 LEXLT expr5 {Bslt($1, $2, $3);} ;
            | expr4 LEXNE expr5 {Bsne($1, $2, $3);} ;
            | expr4 NUMEQ expr5 {Beq($1, $2, $3);} ;
            | expr4 NUMGE expr5 {Bge($1, $2, $3);} ;
            | expr4 NUMGT expr5 {Bgt($1, $2, $3);} ;
            | expr4 NUMLE expr5 {Ble($1, $2, $3);} ;
            | expr4 NUMLT expr5 {Blt($1, $2, $3);} ;
            | expr4 NUMNE expr5 {Bne($1, $2, $3);} ;
            | expr4 EQUIV expr5 {Beqv($1, $2, $3);} ;
            | expr4 NOTEQUIV expr5 {Bneqv($1, $2, $3);} ;

expr5       : expr6 ;
            | expr5 CONCAT expr6 {Bcat($1, $2, $3);} ;
            | expr5 LCONCAT expr6 {Blcat($1, $2, $3);} ;

expr6       : expr7 ;
            | expr6 PLUS expr7 {Bplus($1, $2, $3);} ;
            | expr6 DIFF expr7 {Bdiff($1, $2, $3);} ;
            | expr6 UNION expr7 {Bunion($1, $2, $3);} ;
            | expr6 MINUS expr7 {Bminus($1, $2, $3);} ;

expr7       : expr8 ;
            | expr7 STAR expr8 {Bstar($1, $2, $3);} ;
            | expr7 INTER expr8 {Binter($1, $2, $3);} ;
            | expr7 SLASH expr8 {Bslash($1, $2, $3);} ;
            | expr7 MOD expr8 {Bmod($1, $2, $3);} ;

expr8       : expr9 ;
            | expr9 CARET expr8 {Bcaret($1, $2, $3);} ;

expr9       : expr10 ;
            | expr9 BACKSLASH expr10 {Blim($1, $2, $3);} ;
            | expr9 AT expr10 {Bact($1, $2, $3);};
```

```
expr10    : expr11 ;
          | AT expr10 {Uat($1,$2);} ;
          | NOT expr10 {Unot($1,$2);} ;
          | BAR expr10 {Ubar($1,$2);} ;
          | CONCAT expr10 {Uconcat($1,$2);} ;
          | LCONCAT expr10 {Ulconcat($1,$2);} ;
          | DOT expr10 {Udot($1,$2);} ;
          | BANG expr10 {Ubang($1,$2);} ;
          | DIFF expr10 {Udiff($1,$2);} ;
          | PLUS expr10 {Uplus($1,$2);} ;
          | STAR expr10 {Ustar($1,$2);} ;
          | SLASH expr10 {Uslash($1,$2);} ;
          | CARET expr10 {Ucaret($1,$2);} ;
          | INTER expr10 {Uinter($1,$2);} ;
          | TILDE expr10 {Utilde($1,$2);} ;
          | MINUS expr10 {Uminus($1,$2);} ;
          | NUMEQ expr10 {Unumeq($1,$2);} ;
          | NUMNE expr10 {Unumne($1,$2);} ;
          | LEXEQ expr10 {Ulexeq($1,$2);} ;
          | LEXNE expr10 {Ulexne($1,$2);} ;
          | EQUIV expr10 {Uequiv($1,$2);} ;
          | UNION expr10 {Uunion($1,$2);} ;
          | QMARK expr10 {Uqmark($1,$2);} ;
          | NOTEQUIV expr10 {Unotequiv($1,$2);} ;
          | BACKSLASH expr10 {Ubackslash($1,$2);} ;

expr11    : literal ;
          | section ;
          | return ;
          | if ;
          | case ;
          | while ;
          | until ;
          | every ;
          | repeat ;
          | CREATE expr {Create($1,$2);} ;
          | IDENT {Var($1);} ;
          | NEXT {Next($1);} ;
          | BREAK nexpr {Break($1,$2);} ;
          | LPAREN exprlist RPAREN {Paren($1,$2,$3);} ;
          | LBRACE compound RBRACE {Brace($1,$2,$3);} ;
          | LBRACK exprlist RBRACK {Brack($1,$2,$3);} ;
          | expr11 LBRACK nexpr RBRACK {Subscript($1,$2,$3,$4);} ;
          | expr11 LBRACE        RBRACE {Pdco0($1,$2,$3);} ;
          | expr11 LBRACE pdcolist RBRACE {Pdco1($1,$2,$3,$4);} ;
          | expr11 LPAREN exprlist RPAREN {Invoke($1,$2,$3,$4);} ;
          | expr11 DOT IDENT {Field($1,$2,$3);} ;
          | CONJUNC FAIL {Kfail($1,$2);} ;
          | CONJUNC IDENT {Keyword($1,$2);} ;

while     : WHILE expr {While0($1,$2);} ;
          | WHILE expr DO expr {While1($1,$2,$3,$4);} ;
```

```
until       : UNTIL expr {Until0($1,$2);} ;
            | UNTIL expr DO expr {Until1($1,$2,$3,$4);} ;

every       : EVERY expr {Every0($1,$2);} ;
            | EVERY expr DO expr {Every1($1,$2,$3,$4);} ;

repeat      : REPEAT expr {Repeat($1,$2);} ;

return      : FAIL {Fail($1);} ;
            | RETURN nexpr {Return($1,$2);} ;
            | SUSPEND nexpr {Suspend0($1,$2);} ;
            | SUSPEND expr DO expr {Suspend1($1,$2,$3,$4);};

if          : IF expr THEN expr {If0($1,$2,$3,$4);} ;
            | IF expr THEN expr ELSE expr {If1($1,$2,$3,$4,$5,$6);} ;

case        : CASE expr OF LBRACE caselist RBRACE {Case($1,$2,$3,$4,$5,$6);} ;

caselist: cclause ;
            | caselist SEMICOL cclause {Caselist($1,$2,$3);} ;

cclause     : DEFAULT COLON expr {Cclause0($1,$2,$3);} ;
            | expr COLON expr {Cclause1($1,$2,$3);} ;

exprlist: nexpr
            | exprlist COMMA nexpr {Exprlist($1,$2,$3);} ;

pdcolist: nexpr {
                        Pdcolist0($1);
                        } ;
            | pdcolist COMMA nexpr {
                        Pdcolist1($1,$2,$3);
                        } ;

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

section     : expr11 LBRACK expr sectop expr RBRACK {Section($1,$2,$3,$4,$5,$6);} ;

sectop      : COLON {Colon($1);} ;
            | PCOLON {Pcolon($1);} ;
            | MCOLON {Mcolon($1);} ;

compound: nexpr ;
            | nexpr SEMICOL compound {Compound($1,$2,$3);} ;

program     : error decls EOFX ;
proc        : prochead error procbody END ;
expr        : error ;
%%
```

Appendix B — Specifications for the Identity Translator

```
%{
nodeptr q();
nodeptr cat();
%}

#   Declaration Syntax
#
#       declarations
#
%{
#define Globdcl(x)      if (!nocode) treeprt(x); treeinit()
#define Linkdcl(x)      if (!nocode) treeprt(x); treeinit()
#define Procdcl(x)      if (!nocode) treeprt(x); treeinit()
#define Recdcl(x)       if (!nocode) treeprt(x); treeinit()
%}

#
#       syntax subsidiary to declarations
#
Arglist1()      ""
Arglist2(x)     x
Arglist3(x)     x           "[]"
Dynamic(x)      "dynamic "
Global0(x)      ""
Global1(x, y, z)            "global "   z           "\n"
Initial1()      ""
Initial2(x, y, z)           "initial "  y           ";\n"
Link(x, y)      "link "     y           "\n"
Lnkfile1(x)     x
Lnkfile2(x)     "\""        x           "\""
Lnklist(x, y, z)            x           ", "        z
Local(x)        "local "
Locals1()       ""
Locals2(w, x, y, z)         w           x           y           ";\n"
Record1(x)      ""
Proc1(u, v, w, x, y, z)     u           ";\n"       w           x           y           "end\n"
Record2(u, v, w, x, y, z)   "record "   w           "("         y           ")\n"
Procbody1() ""
Procbody2(x, y, z)          x           ";\n"       z
Prochead1(x)                ""
Prochead2(u, v, w, x, y, z) "procedure "w           "("         y           ")"
Static(x)       "static "
```

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

#
#		reserved–word syntax
#
Break(x, y)	"break "	y
Case(u, v, w, x, y, z)		"case "	v	" of {\n"	y	"\n}"
Caselist(x, y, z)		x	";\n"	z
Cclause0(x, y, z)		"default:"	z
Cclause1(x, y, z)		x	":"	z
Create(x, y)	"create "	y
Every0(x, y)	"every "	y
Every1(w, x, y, z)		"every "	x	" do "	z
Fail(x)	"fail"
If0(w, x, y, z)	"if "	x	" then "	z
If1(u, v, w, x, y, z)		"if "	v	" then "	x	" else "	z
Next(x)	"next "
Repeat(x, y)	"repeat "	y
Return(x, y)	"return "	y
Suspend0(x, y)		"suspend "	y
Suspend1(w, x, y, z)		"suspend "	x	" do "	z
To0(x, y, z)	x	" to "	z
To1(v, w, x, y, z)		v	" to "	x	" by "	z
Unot(x, y)	"not "	y
Until0(x, y)	"until "	y
Until1(w, x, y, z)		"until "	x	" do "	z
While0(x, y)	"while "	y
While1(w, x, y, z)		"while "	x	" do "	z

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

```
#
#	unary operators
#
<uop>(x,y)	"<uop>"	y
<ucs>(x,y)	"<ucs>"	y

#
#	miscellaneous expressions
#
Brace(x,y,z)	"{\n"	y	"\n}"
Brack(x,y,z)	"["	y	"]"
Colon(x)	":"
Compound(x,y,z)	x	";\n"	z
Exprlist(x,y,z)	x	", "	z
Field(x,y,z)	x	"."	z
Invoke(w,x,y,z)	w	"("	y	")"
Mcolon(x)	"–:"
Paren(x,y,z)	"("	y	")"
Pcolon(x)	"+:"
Pdco0(x,y,z)	x	"{"	"}"
Pdco1(w,x,y,z)	w	"{"	y	"}"
Pdcolist0(x)	x
Pdcolist1(x,y,z)	x	", "	z
Progend(x,y)	""
Section(u,v,w,x,y,z)	u	"["	w	x	y	"]"
Subscript(w,x,y,z)	w	"["	y	"]"
```

Appendix C — Files for Building a Variant Translator

Makefile	construction of translator
bsyms	macro names for binary operators
cat.c	string handling functions for the parser
define	macro definition program
define.icn	source for define
ident.defs	macro definitions for the identity translator
usyms	macro names for unary operators
variant.c	parser functions for variant translators
variant.defs	macro definitions for the variant translator
vitran	the variant translator
h/config.h	configuration parameters
h/define.h	installation definitions
h/keyword.h	keyword definitions
h/memsize.h	memory size parameters
itran/Makefile	construction of translator
itran/err.c	routines for producing error messages
itran/expanded.g	Yacc grammar with macro definitions expanded
itran/fixgram	clean up grammar file after macro expansion
itran/fixgram.icn	source for fixgram
itran/gdefs.h	macro definitions produced from ident.defs and variant.defs
itran/icon_g.c	Yacc grammar before macro expansion
itran/itran.c	main program that controls translation
itran/itran.h	external definitions used throughout the translator
itran/lex.c	routines for lexical analysis
itran/lex.h	structures and definitions used by the lexical analyzer
itran/mem.c	memory initialization and management
itran/mktoktab	program to build optab.c and toktab.c
itran/mktoktab.icn	source for mktoktab
itran/optab	specifications for operator recognition
itran/optab.c	state tables for operator recognition
itran/parse.c	the parser as modified by pscript
itran/pscript	edit script to modify parser produced by Yacc
itran/sym.c	routines for symbol table management
itran/sym.h	structures for symbol table entries
itran/token.h	token definitions generated by Yacc
itran/tokens	token specifications
itran/toktab.c	initialization of structures containing token information
itran/tree.c	routines to build tree structures
itran/tree.h	parse tree structures and accessing macros

Appendix D — The Variant Translator Makefile

```
SHELL=/bin/sh
CFLAGS= -DVarTran
LDFLAGS=

OBJS=                   cat.o err.o itran.o  lex.o  mem.o optab.o parse.o sym.o\
                                  toktab.o tree.o variant.o

../itran:               $(OBJS)
                        $(CC) $(LDFLAGS) -o ../vitran $(OBJS)

$(OBJS):      ../h/config.h ../h/define.h

cat.o:                  tree.h ../cat.c
                        cc -c $(CFLAGS) ../cat.c

variant.o:    ../variant.c
                        cc -c $(CFLAGS) ../variant.c

err.o:                  itran.h lex.h token.h tree.h
itran.o:      itran.h sym.h token.h tree.h ../h/config.h ../h/define.h
lex.o:                  itran.h lex.h token.h tree.h
mem.o:                  itran.h sym.h tree.h ../h/memsize.h
optab.o:      lex.h
parse.o:      itran.h sym.h tree.h ../h/config.h ../h/define.h
sym.o:                  itran.h sym.h token.h
toktab.o:     itran.h lex.h token.h
tree.o:                 tree.h

parse.c token.h:        expanded.g
                        yacc -d expanded.g      # expect 214 shift/reduce conflicts
                        mv y.tab.c parse.c
                        ed - parse.c <pscript
                        mv y.tab.h token.h

expanded.g:             icon_g.c gdefs.h fixgram
                        cc -E icon_g.c | fixgram >expanded.g

gdefs.h:                ../bsyms ../usyms ../ident.defs ../variant.defs
                        cd ..;        define ident.defs variant.defs >gdefs.h
                        mv ../gdefs.h .

toktab.c optab.c:       tokens optab mktoktab
                        mktoktab

mktoktab:               mktoktab.icn
                        icont -s mktoktab.icn

fixgram:                fixgram.icn
                        icont -s fixgram.icn
```