Building Source-Code Processors for Icon Programs

Ralph E. Griswold

Department of Computer Science, The University of Arizona

There are many situations in which processing an Icon source-language program is useful. Many of these fall into the category of preprocessing to change the form of a program, as in modeling string scanning [1]. In this case, all string scanning expressions are recast in terms of procedure calls:

expr1 ? *expr2* → Escan(Bscan(*expr1*), *expr2*)

Preprocessors to do these kinds of things often are written by hand, an approach that often produces unsatisfactory results. Hand-crafted preprocessors rarely treat source-language code with complete generality. For example, a hand-crafted preprocessor for modeling string scanning might not properly handle a string literal such as

```
write("The form is s ? e")
```

and instead change the literal. Similarly, hand-crafted preprocessors often assume that programs are laid out in reasonable ways, and may not handle something like this:

```
s ?
while
move(
1
)
do
write(
move(
1
)
)
```

And few hand-crafted preprocessors correctly handle programs that contain syntax errors.

The reason for such problems is obvious: A correct and robust preprocessor must correctly parse all source-language programs. This is a major undertaking that usually is short-circuited because of the effort and technical skill that is required.

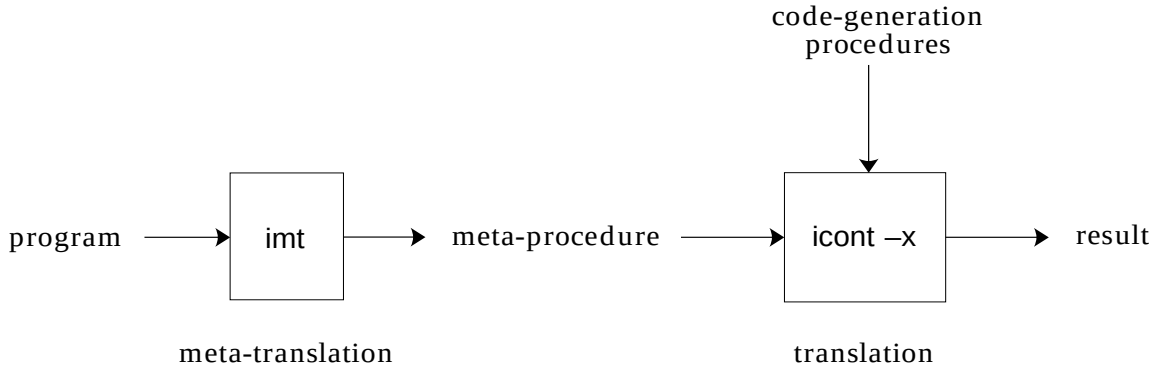
Icon's variant translator system [2] was developed to mitigate these problems. It uses the parser contained in the implementation of Icon together with a system for describing code generation. Variant translators take care of the parsing part of the problem and do it in a way that is as correct and robust as the Icon compiler.

By taking care of the parsing part of the problem, variant translators make it relatively easy to create some kinds of processors. A preprocessor for modeling string scanning can be created by making a few simple changes to the standard code-generation specifications.

Part of the problem with using with variant translators is having to learn how to specify code-generators and dealing with a somewhat awkward syntax. A more serious problem is that supplementary and often tricky C code is needed in many cases.

This report describes a *meta-translator* system for Icon. This term is used because a translator

converts an Icon source-language program into an Icon procedure that, when called, produces a translation of the original program:



The result usually is another Icon program, such as one in which all string scanning expressions have been replaced by procedure calls. But the result can be something entirely different, as shown in an example later in this report.

The program imt is itself a variant translator. It handles only parsing and does not have to be changed to effect different meta-translations.

As an example of meta-translation, consider the simple program

```
procedure main()
  while line := read() do
      line ? process()
end
```

The output of imt for this program is a procedure Mp() (for meta-procedure):

```
procedure Mp()
Proc("main",);
Body(WhileDo(Asgnop(":=",Var("line"),
Invoke(Var("read"),Null())),Scan(Var("line"),
Invoke(Var("process"),Null()))),);
End();
end
```

The procedure calls in Mp() correspond to the syntactic components of the original program. For example, Proc() is a procedure that corresponds to a procedure declaration header in the original program, and WhileDo() is a procedure corresponding to the while–do loop. The procedure Body() writes the code for the procedure body.

The simplest of all translations is an identity translation in which the result is semantically equivalent to the original program. These code-generation procedures are listed in the appendix to this report and are contained in the library file identgen.icn. The code-generation procedure for the identity translation of while–do is

```
procedure WhileDo(e1, e2)
   return code_gen("while ", e1, " do ", e2)
end
```

The procedure code_gen() produces the concatenation of its arguments. It can be changed for other translations.

The result of linking identgen.icn with the procedure Mp() produced by imt and executing the result is:

```
procedure main()
while (line := read()) do (line ? process())
end
```

This is functionally equivalent to the original program; only the layout is different.

To see how other translations can be done, consider the string-scanning transformation given at the beginning of this article. The identify translation for string scanning is

```
procedure Scan(e1, e2)
  return code_gen("(", e1 , " ? ", e2, ")")
end
```

To get a translation for modeling string scanning, it's only necessary to change this procedure to

```
procedure Scan(e1, e2)
  return code_gen("Escan(Bscan(", e1,  "),",  e2,  ")"
end
```

For completeness, augmented string scanning should be translated in a similar manner. See Augscan() in the appendix.

As another example, consider converting calls of map() to calls of Map() so that mapping can be traced. The declaration for Map() might be

```
procedure Map(s1, s2, s3)
  return map(s1, s2, s3)
end
```

The meta-translation can be accomplished by adding the line

```
if e == "map" then e := "Map"                # check procedure name
```

at the beginning of the procedure Invoke() shown in the appendix.

A procedure declaration for Map() must be included in the final program. This can be done by adding the following lines to the beginning or end of main() in the code-generator procedures:

```
write("procedure Map(s1, s2, s3)")
write("   return map(s1, s2, s3)")
write("end")
```

An alternative approach, which is more desirable in the case of elaborate translations, is to add

```
write("link libe")
```

at the beginning or end of main() and provide the code to be linked with the final program in *libe*.icn.

Incidentally, to trace a generator, suspend must be used instead of return; otherwise the procedure won't produce all the results produced by the generator. For example, for seq(), the procedure would be

```
procedure Seq(i1, i2)
  suspend seq(i1, i2)
end
```

In fact, it doesn't hurt to use suspend for functions that aren't generators.

Other Kinds of Translations

As shown in the examples above, code generators for meta-translation typically produce Icon source-language code. They can, however, do other things. An example is the static analysis of Icon programs to produce a tabulation of the syntactic tokens that occur in them.

Since Mp() describes the syntactic structure of a program with a procedure call for every token, the code-generation procedure it calls can count the number of times they are called instead of producing source code. Consider, for example, string scanning expressions. As shown above, the procedure for identity translation for string scanning is

```
procedure Scan(e1, e2)
   return code_gen("(", e1 , " ? ", e2, ")")
end
```

To count the number of string scanning expressions in a program, all that's needed is to replace this procedure by

```
procedure Scan(e1, e2)
   token["e1 ? e2"] +:= 1
end
```

where token is a table created by the main procedure before Mp() is called. When Mp() returns, the tabulation of tokens can be written out. (Table subscripts like "e1 ? e2" are used to make the results easy to read.)

There are may possible variations on this approach. For example, when tabulating all the tokens in a program, it may be better to use several tables to segregate operators, controls structures, and so forth. See Reference 3 for the description of such a token tabulator.

Many simpler but useful translations are trivially easy to write. For example, to just count the number of string-scanning expressions in a program and ignore all other kinds of tokens, all the code-generations procedures except Scan() and Augscan() can be simply do nothing; no code is required for them. (A collection of procedure wrappers for the code-generations procedures is contained in emptygen.icn.)

Conclusions

Meta-translators allow translators for Icon programs to be written entirely in Icon. This provides the power of a high-level language and one that is likely to be familiar to persons writing source-code processors for Icon. In many cases, the necessary changes to the code-generation procedures are obvious and easily accomplished. If every code-generation procedure needs to be changed, as in the case of a complete token tabulator, the amount of clerical work can be significant. For some kinds of translations, it may be necessary to understand the Yacc grammar for Icon. Consult the source code for icont for this.

A potential problem with meta-translation is the amount of memory required to build the Icon program that produces the final translation. As illustrated above, the output of imt is considerably larger than the input to imt. If the input to imt is a large program, the procedure Mp() that describes it is huge. It's also necessary to link the code-generation procedures with Mp(), adding to the size of the intermediate program. The memory needed usually is not a problem on platforms in the workstation class, but it can be on personal computers.

Getting the Meta-Translator System

The meta-translator system is available by anonymous FTP to cs.arizona.edu; cd /icon/meta and get READ.ME to see what to do next.

References

1. "Modeling String Scanning", *Icon Analyst* 6, pp. 1-2.

2. *Variant Translators for Version 9 of Icon*, Ralph E. Griswold, Icon Project Document IPD245, Department of Computer Science, The University of Arizona, 1995.

3. "Static Analysis of Icon Programs", *Icon Analyst* 27, pp. 5-11.

Appendix Code-Generation Procedures for Identity Translation

```
link cat                                code_gen(s1, s2, ...)

global code_gen

procedure main()
   code_gen := cat                      # so these procedures can be changed easily
   Mp()                                 # call meta–procedure
end

procedure Alt(e1, e2)                   # e1 | e2
   return code_gen("(", e1, "|", e2, ")")
end

procedure Apply(e1, e2)                 # e1 ! e2
   return code_gen("(", e1, "!", e2, ")")
end

procedure Arg(e)                        # procedure argument (parameter)
   return e
end

procedure Asgnop(op, e1, e2)            # e1 op e2
   return code_gen("(", e1, " ", op, " ", | e2, ")")
end

procedure Augscan(e1, e2)               # e1 ?:= e2
   return code_gen("(", e1, " ?:= ", e2, ")")
end

procedure Bamper(e1, e2)                # e1 & e2
   return code_gen("(", e1, " & ", e2, ")")
end

procedure Binop(op, e1, e2)             # e1 op e2
   return code_gen("(", e1, " ",  op, " ",  e2, ")")
end

procedure Body(es[ ])                   # procedure body
   every write(!es)
   return
```

```
end

procedure Break(e)                      # break e
   return code_gen("break ", e)
end

procedure Case(e, clist)                # case e of { caselist }
   return code_gen("case ", e, " of {", clist, "}")
end

procedure Cclause(e1, e2)               # e1 : e2
   return code_gen(e1, " : ", e2, "\n")
end

procedure Clist(cclause1, cclause2)     # cclause1; cclause2
   return code_gen(cclause1, ";", cclause2)
end

procedure Clit(c)                       # 'c'
   return image(c)
end

procedure Compound(es[])                # { e1; e2; ... }
   local result
   if *es = 0 then return "{}\n"
   result := "{\n"
   every result ||:= !es || "\n"
   return code_gen(result, "}\n")
end

procedure Create(e)                     # create e
   return code_gen("create ", e)
end

procedure Default(e)                    # default: e
   return code_gen("default: ", e)
end

procedure End()                         # end
   write("end")
   return
end

procedure Every(e)                      # every e
```

```
   return code_gen("every ", e)
end

procedure EveryDo(e1, e2)                    # every e1 do e2
   return code_gen("every ", e1, " do ", e2)
end

procedure Fail()                             # fail
   return "fail"
end

procedure Field(e, f)                        # e . f
   return code_gen("(", e, ".", f, ")")

end
procedure Global(vs[])                       # global v1, v2, ...
   local result
   result := ""
   every result ||:= !vs || ", "
   write("global ", result[1:–2])
   return
end

procedure If(e1, e2)                         # if e1 then e2
   return code_gen("if ", e1, " then ", e2)
end

procedure IfElse(e1, e2, e3)                 # if e1 then e2 else e3
   return code_gen("if ", e1, " then ", e2, " else ", e3)
end

procedure Ilit(i)                            # i
   return i
end

procedure Initial(e)                         # initial e
   write("initial ", e)
   return
end

procedure Invocable(ss[ ])                   # invocable ... (mt doesn't handle general case)
   if \es then write("invocable all")
   else write("invocable ", ss)
```

```
   return
end

procedure Invoke(e, es[ ])                  # e(e1, e2, ...)
   local result
   if *es = 0 then return code_gen(e, "()")
   result := ""
   every result ||:= !es || ", "
   return code_gen(e, "(", result[1:–2], ")")
end

procedure Key(s)                            # &s
   return code_gen("&", s)
end

procedure Limit(e1, e2)                     # e1 \ e2
   return code_gen("(", e1, "\\", e2, ")")
end

procedure Link(vs[ ])                       # link "v1, v2, ..."
   local result
   result := ""
   every result ||:= !vs || ", "
   write("link ", result[1:–2])
   return
end

procedure List(es[ ])                       # [e1, e2, ... ]
   local result
   if *es = 0 then return "[ ]"
   result := ""
   every result ||:= !es || ", "
   return code_gen("[", result[1:–2], "]")
end

procedure Local(vs[ ])                      # local v1, v2, ...
   local result
   result := ""
   every result ||:= !vs || ", "
   write("local ", result[1:–2])
   return
end

procedure Next()                            # next
```

```
   return "next"
end

procedure Not(e)                        # not e
   return code_gen("not(", e, ")")
end

procedure Null()                        # &null
   return ""
end

procedure Paren(es[ ])                  # (e1, e2, ... )
   local result
   if *es = 0 then return "()"
   result := ""
   every result ||:= !es || ", "
   return code_gen("(", result[1:–2], ")")
end

procedure Pdco(e, es[ ])                # e{e1, e2, ... }
   local result
   if *es = 0 then return code_gen(e, "{}")
   result := ""
   every result ||:= !es || ", "
   return code_gen(e, "{", result[1:–2], "}")
end

procedure Proc(n, vs[ ])                # procedure n(v1, v2, ...)
   local result, v
   if *vs = 0 then write("procedure ", n, "()")
   result := ""
   every v := !vs do
      if \v == "[ ]" then result[–2:0] := v || ", "
      else result ||:= (\v | "") || ", "
   write("procedure ", n, "(", result[1:–2], ")")
   return
end

procedure Record(n, fs[ ])              # record n(f1, f2, ...)
   local result, field
   if *fs = 0 then write("record ", n, "()")
   result := ""
   every field := !fs do
```

```
      result ||:= (\field | "") || ", "
   write("record ", n, "(", result[1:–2], ")")
   return
end

procedure Repeat(e)                    # repeat e
   return code_gen("repeat ", e)
end

procedure Return(e)                    # return e
   return code_gen("return ", e)
end

procedure Rlit(r)                      # r
   return r
end

procedure Scan(e1, e2)                 # e1 ? e2
   return code_gen("(", e1 , " ? ", e2, ")")
end

procedure Section(op, e1, e2, e3)      # e1[e2 op  e3]
   return code_gen(e1, "[", e2, op, e3, "]")
end

procedure Slit(s)                      # "s"
   return image(s)
end

procedure Static(vs[ ])                # static v1, v2, ...
   local result
   result := ""
   every result ||:= !vs || ", "
   write("static ", result[1:–2])
   return
end

procedure Subscript(e1, e2)            # e1[e2]
   return code_gen(e1, "[", e2, "]")
end

procedure Suspend(e)                   # suspend e
```

```
   return code_gen("suspend ", e)
end

procedure SuspendDo(e1, e2)                # suspend e1 do e2
   return code_gen("suspend ", e1, " do ", e2)
end

procedure To(e1, e2)                       # e1 to e2
   return code_gen("(", e1, " to ", e2, ")")
end

procedure ToBy(e1, e2, e3)                 # e1 to e2 by e3
   return code_gen("(", e1, " to ", e2, " by ", e3, ")")
end

procedure Repalt(e)                        # |e
   return code_gen("(|", e, ")")
end

procedure Unop(op, e)                      # op e
   return code_gen("(", op, e, ")")
end

procedure Until(e)                         # until e
   return code_gen("until ", e)
end

procedure UntilDo(e1, e2)                  # until e1 do e2
   return code_gen("until ", e1, " do ", e2)
end

procedure Var(v)                           # v
   return v
end

procedure While(e)                         # while e
   return code_gen("while ", e)
end

procedure WhileDo(e1, e2)                  # while e1 do e2
   return code_gen("while ", e1, " do ", e2)
end
```