TO: Distribution

FROM: Gary C. Dixon

DATE: May 28, 1974

SUBJECT: Creating Special-Purpose Translators

Often, system programmers must define a new, special-purpose language and write a compiler, interpreter, or other form of translator for that language. Usually, such languages are used: to specify the contents of, or to manipulate items in, a particular data base; to generate ALM code; or to specify some process to be performed. Examples of such languages in Multics include: exec_com control language; runoff control language; bind control language; the input language for set_search_rules; error_table_ language; IO Daemon parameter language; project master file language; IO compiler language; etc.

Languages like exec_com, bind control and runoff control language are used heavily and therefore deserve special-purpose translators which are optimized for peak performance. However, most special-purpose languages are used infrequently as part of some maintenance or development process. The translators for such lightly-used languages should be quick to write, simple to understand and maintain, and easy to extend, rather than being optimized for high performance.

Multics should provide a tool which creates a translator from a simple specification of the syntax and semantics of the language to be translated. Such a tool would make it easier to write special-purpose, lightly-used translators. In addition, the use of a single translator generation tool would guarantee that all of the translators would have the same structure and would share the same method of processing their input language. Instead of 20 translators with 20 different methods of language specification and 20 unique translation algorithms, there would be 20 translators which translate languages defined in a common language definition language and which share a common language translation algorithm. This would greatly simplify the task of understanding and maintaining all of the translators which seem to be proliferating in Multics at an alarming rate.

MIT's course 6.251, "Programming Systems", presented such a language definition language called the reduction language. In this language, the phrases in the language to be translated are defined in Backus-Naur Form (BNF). A set of action routines are associated with each of the defined phrases. These action routines assign some semantic meaning to input strings which match the defined phase. The reduction language is easy to use, can define a large class of languages, and can be compiled into an efficient, table-driven translator.

To test out the theoretical benefits of a translator compiler, I have created a compiler for the 6.251 reduction language. As input, this reduction_compiler accepts an ASCII segment containing a set of reductions defining the language to be translated, and a set of PL/I subroutines which are the action routines referenced by those reductions. The compiler converts the reductions into PL/I declarations for the tables which drive a pre-coded translation subroutine. The compiler outputs the table declarations, the translation subroutine, and the action routines into a PL/I source segment, which can then be compiled.

I found significant benefits in the code generated by the reduction_compiler. First, it is easier and faster to generate a translator with the reduction_compiler than to generate an equivalent translator by hand. The reduction_compiler was used to bootstrap itself, and the total time for coding and bootstrapping was about two man-days. Tom VanVleck used the reduction_compiler to create a new version of cv_pmf in about one man-day. Ross Klinger used the reduction_compiler to create the tape_in/tape_out command, the translation portion of which took about one man-day to code.

Second, the very nature of the reduction language forces the programmer to separate the definition of language syntax from the coding of action routines. This separation in the code forces a beneficial separation in the mind of the programmer in a way which simplifies the process of defining the translator. The separation also makes the translator easier to understand for other programmers who must maintain or extend the translator at a later date.

Third, an important side effect of the separation between syntax analysis and action routines is the ease of debugging reduction-generated translators. Because the translation code is compiler-generated, it is error-free, thus eliminating what is usually an important source of bugs in most translators. Also, the bugs which do appear are isolated from one another and are easy to identify. In general, they are caused by a bad reduction statement which causes one phrase in the language being translated to be rejected by the translator; or they are caused by a bad action routine which causes only one or a few phrases to be translated improperly.

Fourth, the structure of reduction-generated translators makes is easy to create one action routine that can be used in several different translators. For example, I have created an error message action routine which prints a compiler-style error message from a table of messages. The subroutine substitutes values into the error message text, reports the error severity and line number, and prints the statement or line which was in error. This ease of sharing useful routines could facilitate the development of a library of general-purpose action routines which would further simplify the creation of a compiler, while at the

same time easing the maintenance problem by promoting the use of common code.

Finally, since the translation is a table-driven process which uses an efficient translation algorithm, translator performance is competitive with the most carefully hand-coded translators. VanVleck's new version of cv_pmf operates about 30% faster than the installed version. Some of this performance improvement is due to the use of EIS instructions in the reduction_compiler and its associated parsing routine, but I believe the table-driven translation process accounts for at least a 10% speed-up in the translation.

Given these benefits, I feel that it would be useful to install the reduction_compiler or a similar tool. I would appreciate your comments on this proposed installation.

If you are interested in detailed specifications for the reduction language which is input to the reduction_compiler, or for the lex_string_ lexical analyzer which is an adjunct of the reduction_compiler, you can dprint the following runoff segments.

```
>udd>pdo>gd>doc>p>reduction_compiler.sps.runoff
>udd>pdo>gd>doc>p>lex_string_.sps.runoff
>udd>pdo>gd>doc>p>lex_error_.sps.runoff
```

If you want to try out the reduction_compiler, you will need the include segments in

```
>udd>pdo>gd>installed_source
```

and the object programs in

```
>udd>pdo>gd>object
```