Published: 7/11/66

Identification

Standard Format for the Segment Symbol Table
D.B. Wagner

Purpose

This paper specifies the "segment symbol table" which is expected to be part of the output produced by every translator used in Multics. If the symbol table is missing, many features of Multics will not be available to the procedure.

Definitions

An identifier is as in PL/I: a string of characters used as an indivisible entity in a source program. A symbol is an identifier defined by a programmer in a source program (i.e., an identifier whose use is not fixed in the language). In PL/I for example a symbol may be a scalar variable name, structure name, array name, statement label (including block name and entry name), external entry name, file name, or condition name.

A segment symbol table is a by-product of a compilation or assembly which provides information about the symbols used in the associated program segment. This contains, for each symbol defined in the source program, two kinds of information: implementation information and attribute information. Implementation information provides the necessary correlation between the source program and the associated object program. Attribute information is information present in the source program which, for example, the user of a debugging aid should not have to repeat.

Location of the Symbol Table

For a program segment <a>, the base of the Segment Symbol Table is at

<a.symbol>|[symbol_table]

The Symbol Table

The symbol table is structured, that is in the form of a tree, for at least two reasons: First, PL/I blocks may be nested. Second, the best way to give information concerning a PL/I data-structure is in a tree. There are also other places where structuring is useful. For example, the entry in the symbol table for a PL/I pointer variable may contain pointers

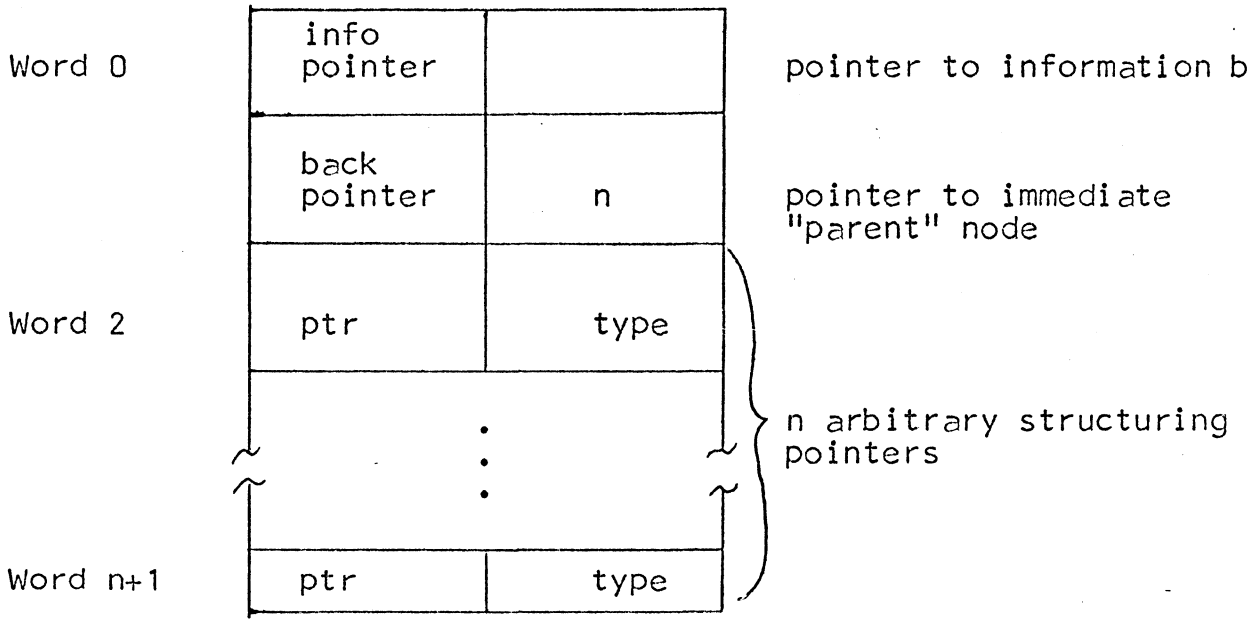
to the entries for any based variables which have this pointer specified in their declarations. Languages other than PL/I will undoubtedly have other uses for structuring in the symbol table.

All structuring of the table is done in a standard way which does not depend on content. This is done so that a standard subroutine can be used to interpret the structure, no matter what translator produced the table or what the structuring "actually means".

The segment symbol table consists of a header (described later), which contains fixed information concerning the translator, and a collection of entries, one for each symbol used in the source program, tied together into a tree by 18-bit pointers (relative to the header).

The Entry

Each entry in the table consists of an agglomeration of pointers, called a node, and an information block giving all information about the symbol and its use. The node looks as follows:



Pointers equal to zero are considered null.

The information block contains the symbol name, address information, attributes, etc. Its format is entirely up to the creators of a particular translator, except for the following: a PL/I-style dope vector included in the header to the segment symbol table should make this block look

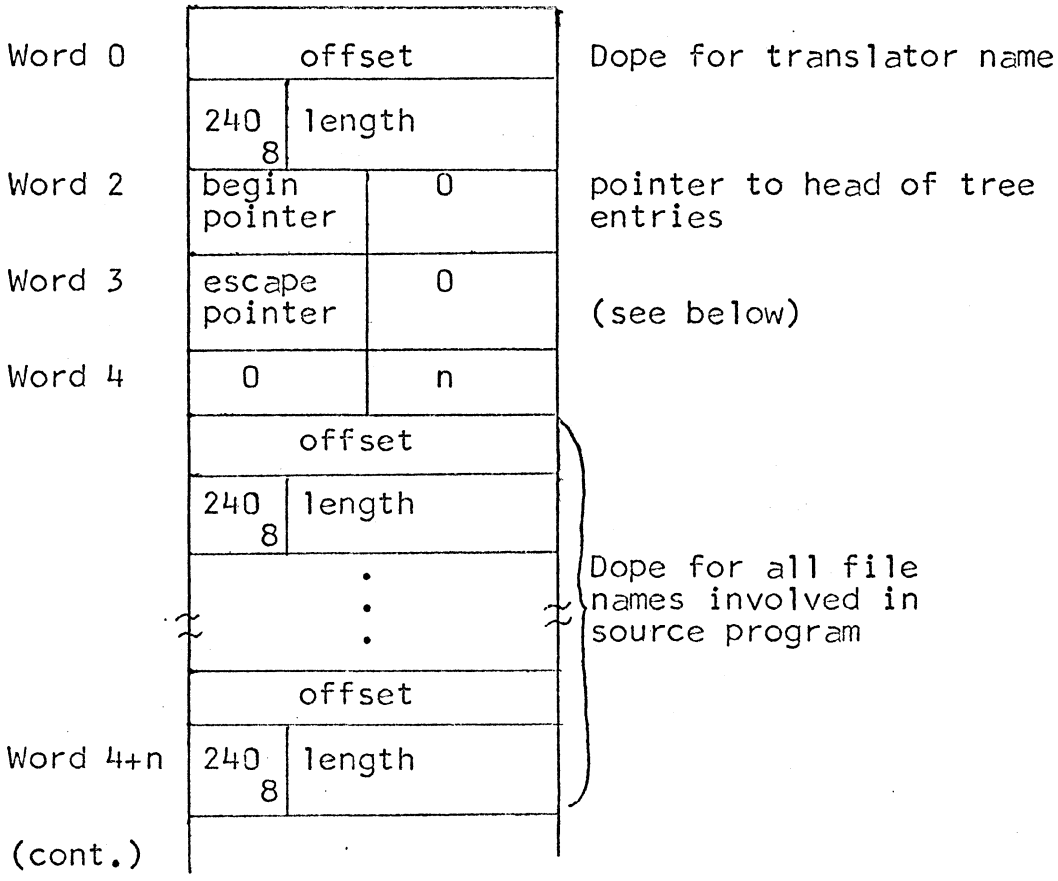
like a PL/I data-structure, and this data-structure should contain the name and address associated with a symbol in a fixed place. This is discussed further below.

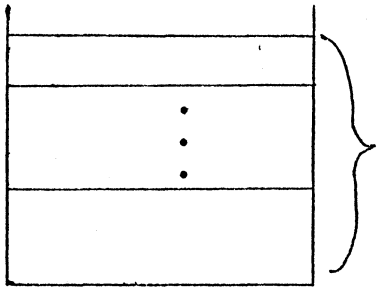
Each pointer in the node may have associated with it an 18-bit "type number" particular to the translator involved. In the PL/I symbol table, for example, type numbers are defined for the pointer-variable controlling a based variable, pointers to substructures of a data-structure, and other kinds of pointers.

The pointers may be considered to be of two kinds: "branches" and "links" (using the terminology made popular by the file system). Branches are pointers to nodes which logically are "descendants" of the present node in the tree structure and links are pointers to nodes which are best considered to be "cutting across" the tree structure. Operationally, within the symbol table a branch is a pointer to a node whose back-pointer points to the present node, and a link is a pointer to a node whose back-pointer points to some other node (or is null, as in the case of the root of the tree).

The Header

The header includes information concerning the translator, the source files, and the format of the "information blocks" used in the rest of the symbol table.





The character-strings mentioned above are somewhere in the same segment as the symbol table, and the "offsets" above indicate where. All the offsets are from Word 0 of the header.

The translator name is a character-string something like "PL/I 01/01/67". It is used in determining the interpretation of the entry dope vector contained in the header.

The reason for providing for more than one file name in the source program is that many translators will include an insert feature such that an entire file can be inserted at some point in a program.

The "escape pointer" points beyond all the variable-length information which follows it to any additional information added to the symbol table format which is not mentioned in this document.

Information Blocks

The "dope vector for information blocks" makes each information block in the table look like a PL/I structure. Naturally it is not necessary that this structure be one that PL/I would or could possibly compile itself. For example, as long as use is unambiguous bit-strings may overlap and arrays (if any) may be indicated as having limits of ± infinity. A complete description of the PL/I table entry is given in Section BD.2.01 and can serve as an example.

The only requirement for the information block which applies to all translators is that its PL/I structure declaration begins as follows:

```
dcl 1 entry ctl(p), 2 name_size bit (9),
                    2 name char (p->entry.name_size),
                    2 address bit (18),
                    2 address_type bit (18)
                     ...
```

Where address_type is a number specifying how address is to be interpreted:

0 = irrelevant (no meaningful address)

1 = location in segment

2 = argument number

3 = stack address (offset from sb←sp)

4 = linkage address (offset from lb←lp)

Address_type's up to 2**17 are reserved for coordinated expansions of this list. Numbers above 2**17 may be used for "special" address-interpretations germane to only one translator.

The reason why the address and address interpretation have been legislated here is that if a "new" translator's symbol table is presented to a debugging program which does not yet know how to handle the data-descriptions in the table, the debugger can at least allow a user to refer to the addresses associated with variables etc. even if it cannot perform all its functions for him. In the early stages of Multics, PL/I will have the status of a new translator.