A PROGRAM HOLDER MODULE

L. Robinson and D.L. Parnas

and

DESIGN AND IMPLEMENTATION OF A MULTI-LEVEL SYSTEM USING SOFTWARE MODULES

L. Robinson

Carnegie-Mellon University
Department of Computer Science
Pittsburgh, Pa. 15213

June, 1973

This research was supported by the National Science Foundation under Grant GJ 32259 and Grant GJ 37728 to Carnegie-Mellon University and also by the Advanced Research Projects Agency of the Office of the Secretary of Defense (F44620-70-C-0107), monitored by the Air Force of Scientific Research.

Abstract

This paper describes a mechanism for holding a program in syntactic form. This mechanism can be useful to any program which processes programs: in program verification, automatic programming, and specialized text editing. In this case the program holder is used to form the basis for a syntax-driven text editor. Formal specifications for the program holder are also given.

L. Robinson and D. L. Parnas

5/16/73

A Program Holder Module

Introduction

This paper describes a mechanism for holding a program in syntactic form. This mechanism can be useful to any program which processes programs: in program verification, automatic programming, and specialized text editing. In this case the program holder is used to form the basis for a syntax-driven text editor.

The program holder is designed so that it can be initialized with any context-free grammar. Thus it can be the basis for a text editor in most programming languages. Editors for different languages will have different features, but can make use of the same progam holder.

The design of the program holder was accomplished by writing specifications for it using the software module specification language of D. L. Parnas [1]. The specifications for the program holder can be found in the appendix. The program holder was then used as the basis of a text editor for the programming tool MUTAS [2].

Specification of the program holder

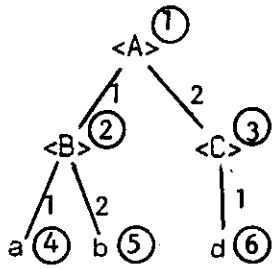
A. Representation as parse tree

In an umbiguous context-free grammar G there exists a unique parse tree for an input string in the language L(G). Thus a parse tree is the most obvious form for holding a program in parsed form. A parse tree is simply an n-ary tree with specialized information at the nodes, so that the structural functions (Table I) of the program holder module represented an n-ary tree (an extension of Parnas' binary tree) [1]. Table I(a) shows the functions for creating and deleting nodes. The nodes are referred to by integers, so that the "name" of the node is simply the integer corresponding to it.

The parse tree is illustrated by the following example. Suppose that the editor is initialized with a grammar G:

```
<A> ::= <B> <C>
<B> ::= ab  |  bc                (1)
<C> ::= d
```

Then the parse tree for the string abd would be

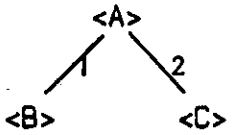


(2)

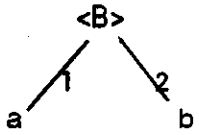
The circled numbers next to each node are the names of the nodes, each node corresponding to a unique integer. Each node can have an arbitrary number of sons, so that the number on the edge connecting a father and one of its sons is the index of that son. The functions which define the interconnections of nodes are listed and described in Table I(b).

B. Representation of grammar and syntactic type assignment

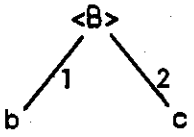
The grammar is represented as a list of smaller trees, each tree corresponding to a production. The grammar G would be represented as



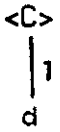
(3.1)



(3.2)



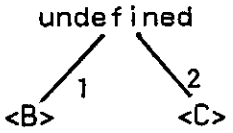
(3.3)



(3.4)

The functions which represent the grammar are listed and explained in Table II.

When a node is initially created, its syntactic type is undefined. There is a function in the program holder which can assign a syntactic type α to a node. Before such an assignment can be succesfully completed, the sons of the node are checked to see if they correspond in syntactic type to the right side of the production in the grammar which defines α. If α is a terminal symbol, no checking need be done, but in this case any node to which α is to be assigned must have no sons. For example, if the incomplete parse tree looked like this



(4)

type <A> could be assigned to the undefined node in (4) by the application of production (3.1). In this manner syntax checking is done one level down from the place of assignment in the program tree. If an exact match is not found, the type assignment is not made and an error call results. The functions which handle type assignments are listed and described in Table III.

In this manner a parse tree can be built from the bottom up. There is also a mechanism for generating a tree top-down.

Note that (3.2) and (3.3) represent two productions which define the nonterminal <B>. When a node is assigned type <B>, it must be specified which alternative of <B> is being assigned. Once the type assignment has been successfully made, the TYPE function for either alternative would have value <B>, and the ALT function would be used to distinguish between different alternative derivations of the same nonterminal symbol (see Table II). This method of referring to productions is useful: in the syntax checking necessary for assignment in (4), the first son must be an instance of <B> -- the alternative is irrelevant; in other cases, such as compilers, it is necessary to know the alternative. Thus it is useful to separate this information, which all sources do not "need to know."

The representation of grammars in the program holder is different from that of context-free grammars in one respect. It enables one to define a nonterminal as a list of zero or more instances of another symbol, separated by instances of yet a third type of symbol. Thus, the productions

<X> ::= <Y> | <X>, <Y> (5)

can be replaced by

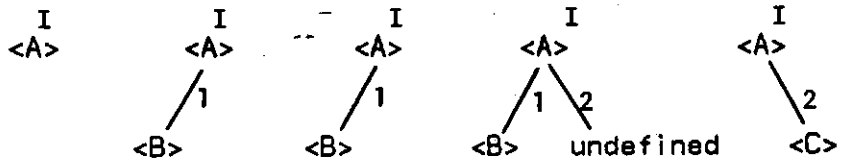
<X> ::=ℓ <Y>, (6)

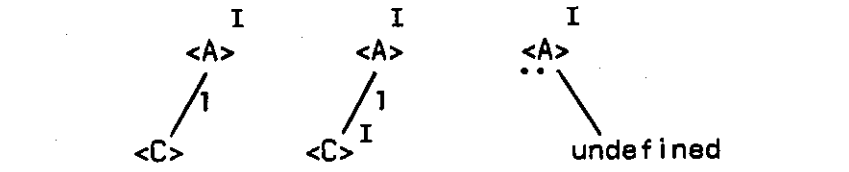
which means that <X> is defined as a list of <Y> with "," as separator. This is useful because it much easier to refer to the nth item of a list when it is represented as in (6) at the module level. Referring to the nth item of a list is very important in text editing and related functions. Many program constructs can be considered as lists: e.g. the list of formal parameters in an Algol procedure, or a compound viewed as a list of statements bracketed by begin . . . end.

C. Incomplete syntax assignment

When a syntax assignment α has been successfully made to a node i, a strong assumption can be made about the subtree of which node i is the root: that the tree represents a valid derivation of the input string (represented by the leaves of the subtree) from symbol α in the grammar contained in the program holder. This assumption is useful in applications such as deterministic bottom-up parsing. However, in other applications, such as top-down parsing or working with incomplete programs, it is desirable to have a tentative or incomplete syntax assignment. In top-down parsing, for example, the goal must be set before a valid derivation exists. With incomplete programs, parts of the code are left out in which decisions are postponed. When the missing information is filled in, a definite syntax assignment can always be made. The functions which control incomplete syntax assingment are described in Table IV.

In an incomplete syntax assignment, the derivation may be incomplete but is never inconsistent. In other words, a tree in which incomplete assignments are present can always be completed by addition to form a valid derivation in the language. Below are five valid incomplete assignments (7.1), designated by an I, and three invalid ones (7.2). The grammar is that defined above.





Conclusion

The program holder module has been implemented, and we are currently investigating its uses in tasks which involve operations on other programs: text editing, verification, compilation, and interpretation.

Table I
Structural Functions for Program Holder

a) Creation and deletion of nodes

function	value	explanation
SETU	-	creates a new node and sets the value of NEWNODE to the name (a unique integer) of the newly created node
NEWNODE	integer	holds the name of the last node created
SPACE	integer	number of storage locations left
DESYN(i)	-	deletes node named i
EXISTS(i)	boolean	true iff node named i exists (i.e. created by SETU, but not deleted)

b) Connecting nodes to form an n-ary tree

function	value	explanation
ERSI(i, n)	boolean	refers to the nth son of node i (true iff such a relation exists)
RSI(i, n)	integer	name of the nth son of node i (defined iff ERSI(i, n) is true)
ELSI(i)	boolean	refers to the father of node i (true iff such a relation exists)
LSI(i)	integer	name of the father of node i (defined iff ELSI(i) is true)
LSIX(i)	integer	if ELSI(i) is true, RSI(LSI(i), LSIX(i)) = i
EMRS(i)	boolean	true iff $\exists n$ [ERSI(i, n) = true]
MRS(i)	integer	maximum index of sons of i (defined iff EMRS(i) is true)
SREL(i, j, n)	-	sets up connection between nodes i and j such that RSI(i, n) = j ERSI(i, n) = true LSI(j) = i ELSI(j) = true LSIX(j) = n
DREL(i, n)	-	deletes father-son relation between nodes i and RSI(i, n)

Table II

Functions Describing
Representation of Grammar in Program Holder

function	value	explanation
NTERM(ty)	boolean	true iff ty refers to a nonterminal symbol
TERM(ty)	boolean	true iff ty refers to a terminal symbol
LIST(ty)	boolean	true if NTERM(ty) is true, and if ty refers to a list of symbols
SYMB(ty)	boolean	true iff ty refers to a symbol table entry
NALT(ty)	integer	number of alternative right hand sides which have symbol ty on the left (defined if NTERM(ty) is true, and if LIST(ty) is false)
NPROD(ty, a)	integer	number of elements in right hand side of the ath alternative production, of which symbol ty is on the left (defined iff NALT(ty) ≤ a)
PROD(ty, n, a)	integer	syntactic type of nth symbol in the ath alternative production, of which symbol ty is on the left (defined iff NPROD(ty, a) ≤ n)
ITEM(ty)	integer	symbol of which ty is a list (defined iff LIST(ty) is true)
SEP(ty)	integer	symbol which separates items of list designated by ty (defined iff LIST(ty) is true)

Table III
Functions Regulating
Assignment of Syntactic Types to Nodes of Program Tree

function	value	explanation
SETTRM(i, ty)	-	assigns type ty to node i if TERM(i) is true
SNTERM(i, ty, a)	-	assigns type ty (alternative a) to node if NTERM(ty) is true and if a match with the grammar exists
SSYMB(i, ty, p)	-	assigns type ty to node i if SYMB(ty) is true. p is the symbol table pointer (type integer)
ETYPE(i)	boolean	true iff node i has a syntactic type assignment
TYPE(i)	integer	integer designating syntactic type of node i (defined iff ETYPE(i) is true)
ALT(i)	integer	if NTERM(TYPE(i)) is true, designates the index of the alternative which corresponds to this instance of TYPE(i)
PSYMB(i)	integer	pointer to symbol table (defined iff SYMB(TYPE(i)) is true)
DTYPE(i)	-	removes syntactic type assignment node i

Table IV

Functions Regulating
Incomplete Syntax Assignments

function	value	explanation
SPTYPE(i, ty, a)	-	makes incomplete syntactic type assignment of ty (alternative a) to node i (NTERM(ty) must be true)
EPTYPE(i)	boolean	true if node i has an incomplete syntactic type assignment
PTYPE(i)	integer	incomplete syntactic type assigned to node i (defined iff EPTYPE(i) is true)
DPTYPE(i)	-	removes incomplete syntactic assignment from node i

Appendix*

Function FILLED

PV: integers
PA: none
IV: 0
EF: none

Function NEWNODE

PV: integers
PA: none
IV: undefined
EF: none

Function EXISTS(i)

PV: true, false
PA: integer i
IV: false
EF:
EC51 if (i < 0) ∨ (i > p1)

*Note: Grammar functions are shown initialized with a specific grammar.

Function EPTYPE(i)

PV: true, false
PA: integer i
IV: undefined
EF:
EC52 if (i < 0) ∨ (i > p1)
EC53 if 'EXISTS'(i) = false

Function PTYPE(i)

PV: integers
PA: integer i
IV: undefined
EF:
EC54 if (i < 0) ∨ (i > p1)
EC55 if 'EXISTS'(i) = false
EC56 if 'EPTYPE'(i) = false

Function DPTYPE(i)

PV: none
PA: integer i
IV: n/a
EF:
EC57 if (i < 0) ∨ (i > p1)
EC58 if 'EXISTS'(i) = false
EC59 if 'EPTYPE'(i) = false
EPTYPE(i) = false
PTYPE(i) = undefined

Function ETYPE(i)

PV: true, false
PA: integer i
IV: undefined
EF:
EC60 if (i < 0) ∨ (i > p1)
EC61 if 'EXISTS'(i) = false

Function TYPE(i)

PV: integer i
PA: integers
IV: undefined
EF:
EC62 if (i < 0) ∨ (i > p1)
EC63 if 'EXISTS'(i) = false
EC64 if 'ETYPE'(i) = false

Function DTYPE(i)

PV: none
PA: integer i
IV: n/a
EF:
EC65 if (i < 0) ∨ (i > p1)
EC66 if 'EXISTS'(i) = false
if 'ELSI'(i) = true then
EC68 if 'ETYPE'('LSI'(i)) = true
EC67 if 'ETYPE'(i) = false
ETYPE(i) = false
TYPE(i) = undefined
if 'NTERM'('TYPE'(i)) = true ∨
'SYMB'('TYPE'(i)) = true
then ALT(i) = undefined

Function ELSI(i)

PV: true, false
PA: integer i
IV: undefined
EF:

EC69 if (i < 0) ∨ (i > p1)
EC70 if 'EXISTS'(i) = false

Function LSI(i)

PV: integers
PA: integer i
IV: undefined
EF:

EC71 if (i < 0) ∨ (i > p1)
EC72 if 'EXISTS'(i) = false
EC73 if 'ELSI'(i) = false

Function DREL(i, n)

PV: none
PA: integer i, n
IV: n/a
EF:

EC74 if (i < 0) ∨ (i > p1)
EC75 if (n < 0) ∨ (n > p1)
EC76 if 'EXISTS'(i) = false
EC77 if 'ERSI'(i, n) = false
ELSI('RSI'(i, n)) = false
LSI('RSI'(i, n)) = undefined
LSIX('RSI'(i, n)) = undefined
ERSI(i, n) = false
RSI(i, n) = undefined
if ¬∃m [m ≠ n
'ERSI'(i, m) = true]
then begin
MRS(i) = undefined
EMRS(i) = false
end
else ∃m [m ≠ n
'ERSI'(i, m) = true
∀p (p ≠ m
p ≠ n
'ERSI'(i, p) = true)
[p < m]
MRS(i) = m]

Function ERSI(i, n)

PV: true, false
PA: integer i, n
IV: undefined
EF:

EC78 if (i < 0) ∨ (i > p1)
EC79 if (n < 1) ∨ (n > p1)
EC80 if 'EXISTS'(i) = false

Function RSI(i, n)

PV: integers
PA: integer i, n
IV: undefined
EF:

EC81 if (i < 0) ∨ (i > p1)
EC82 if (n < 1) ∨ (n > p1)
EC83 if 'EXISTS'(i) = false
EC84 if 'ERSI'(i) = false

Function LSIX(i)

PV: integers
PA: integer i
IV: undefined
EF:

EC90 if (i < 0) ∨ (i > p1)
EC91 if 'EXISTS'(i) = false
EC92 if 'ELSI'(i) = false

Function ALT(i)

PV: integers
PA: integer i
IV: undefined
EF:

EC93 if (i < 0) ∨ (i > p1)
EC94 if 'EXISTS'(i) = false
EC95 if ('ETYPE'(i) =false) ∨ ('EPTYPE'(i) = false)
if 'ETYPE'(i) = true then
EC96 if 'NTERM'('TYPE'(i)) = false

Function SETU

PV: none
PA: none
IV: n/a
EF:

EC98 if 'FILLED' ≥ p2
∃k [k > 0
k < p1
'EXISTS'(k) = false
EXISTS(k) = true
EPTYPE(k) = false
ETYPE(k) = false
ELSI(k) = false
∀ j (j > 0
j < p1)[ERSI(k, j) = false]
NEWNODE = k]
FILLED = 'FILLED'+ 1

Function DESYN(i)

PV: none
PA: integer i
IV: n/a
EF:

EC99 if (i < 0) ∨ (i > p1)
EC100 if 'EXISTS'(i) =false
EC101 if ∃j ['ERSI'(i, j) = false]
EC102 if 'ELSI'(i) = true
EPTYPE(i) = undefined
PTYPE(i) = undefined
EXISTS(i) = false
TYPE(i) = undefined
ETYPE(i) = undefined
FILLED = 'FILLED' - 1

Function SETTRM(i, j)

PV: none
PA: integer i, j
IV: n/a
EF:

EC103 if (j < 0) ∨ (j > p1)
EC104 if (i < 0) ∨ (i > p1)
EC105 if 'EXISTS'(i) = false
EC106 if 'ETYPE'(i) = true
EC107 if 'TERM'(j) = false
EC108 if 'EPTYPE'(i) = true
EC109 if ∃k ['ERSI(i, k) = true]

```
if 'ELSI'(i) = true then
  begin if 'EPTYPE'('LSI'(i)) = true then
        begin
        EC189 if 'NTERM'('PTYPE'('LSI'(i))) = false
        if 'LIST'('PTYPE'('LSI'(i))) = true then
                begin
                if 'LSIX'(i) is odd then
                        EC187 if j ≠ 'ITEM'('PTYPE'('LSI'(i)))
                else EC188 if j ≠ 'SEP'('PTYPE'('LSI'(i)))
                end
        end
    begin
      EC111 if 'PROD'('PTYPE'('LSI'(i)), 'LSIX'(i),
        'ALT'('LSI'(i))) ≠ j
    end
  end
ETYPE(i) = true
TYPE(i) = j
```

```
Function SNTERM(i, ty, m)

        PV:    none
        PA:    integer i, ty, m
        IV:    n/a
        EF:
               EC128 if (i < 0) ∨ (i > p1)
               EC129 if (m < 0) ∨ (m > p1)
               EC130 if (ty <  0) ∨ (ty > p1)
               EC131 if 'EXISTS'(i) = false
               EC132  if 'ETYPE'(i) = true
               EC133 if 'EPTYPE'(i) = true
               EC134 if 'NTERM'(ty) = false
               if 'ELIST'(ty) = false then
                      begin
                      EC135 if ∃n [ 'EPROD'(ty, n, m) = true
                             'ERSI'(i, n) = false ]
                      EC136 if ∃n [ 'EPROD'(ty, n, m) = false
                             'ERSI'(i, n) = true ]
                      ∀n ( 'ERSI'(i, n) = true )
                      [ EC137 if 'ETYPE'('RSI'(i, n)) = false
                             EC138 if 'PROD'(ty, n, m) ≠ 'TYPE'('RSI'(i, n)) ]
                      end
               else begin
                      ∀n ( 'ERSI'(i, n) = true )
                      [ EC139 if 'ETYPE'('RSI'(i, n)) = false
                             EC140 if ∃k [ k = n
                                    k is odd
                                    'TYPE'('RSI'(i, k)) ≠ 'ITEM'(ty) ]
                             EC194 if ∃ k [ k = n
                                    k is even
                                    'TYPE'('RSI'(i, k)) ≠ 'SEP'(ty) ]
                             EC195 if ∃k [ k < n
                                    'ERSI'(i, k) = false ] ]
                      end
                end
               if 'ELSI'(i) = true then
                      begin
                      if 'EPTYPE'('LSI'(i)) = true then
                             EC197 if 'PROD'('PTYPE'('LSI'(i)),
                             'LSIX'(i),'ALT'('LSI'(i)) ≠ ty
                      end
               ETYPE(i) = true
               TYPE(i) = ty
               ALT(i) = m
```

Function TERM(i)

PV: true, false
PA: integer i
IV: $\forall k$ ($k \geq 51$
$k \leq 100$)
[TERM(k) = true]
all others false
EF:
EC141 if (i < 0) $\vee$ (i > p1)

Function EPROD(ty, n, k)

PV: true, false
PA: integer ty, n, k
IV: EPROD(1, 1, 1) = true
EPROD(1, 2, 1) = true
EPROD(1, 3, 1) = true
EPROD(1, 4, 1) = true
EPROD(1, 5, 1) = true
EPROD(1, 6, 1) = true
EPROD(1, 7, 1) = true
.
.
.
all others false
EF:
EC142 if (ty < 0) $\vee$ (ty > p1)
EC143 if (n < 0) $\vee$ (n > p1)
EC144 if (k < 0) $\vee$ (k > p1)
EC198 if 'NTERM'(ty) = false
EC199 if 'ELIST'(ty) = true

Function PROD(ty, n, k)

PV: integers
PA: integer ty, n, k
IV: PROD(1, 1, 1) = 51
PROD(1, 2, 1) = 41
PROD(1, 3, 1) = 2
PROD(1, 4, 1) = 78
PROD(1, 5, 1) = 52
PROD(1, 6, 1) = 77
PROD(1, 7, 1) = 9
.
.
.
all others undefined
EF:
EC145 if (ty < 0) $\vee$ (ty > p1)

EC146 if (n < 0) ∨ (n > p1)
EC147 if (k < 0) ∨ (k > p1)
EC200 if 'NTERM'(ty) = false
EC201 if 'ELIST'(ty) = true
EC148 if 'EPROD'(ty, n, k) = false

Function SPNTERM(i, ty, m)

PV: none
PA: integer i, ty, m
EF:

EC149 if (i < 0) ∨ (i > p1)
EC150 if (ty < 0) ∨ (ty > p1)
EC151 if (m < 0) ∨ (m > p1)
EC152 if 'EXISTS'(i) = false
EC153 if 'EPTYPE'(i) = true
EC154 if 'ETYPE'(i) = true
EC155 if 'NTERM'(ty) = false
if 'ELIST'(ty) = false then
∀n ('ERSI'(i, n) = true)
[EC156 if 'EPROD'(ty, n, m) = false
∀k (k = n
'ETYPE'('RSI'(i, k)) = true)
[EC157 if 'PROD'(ty, k, m) ≠ 'TYPE'(RSI'(i, k))]
∀k (k = n
'EPTYPE'('RSI'(i, k)) = true)
[EC158 if 'PROD'(ty, k, m) ≠ 'PTYPE'('RSI'(i, k))]]
else ∀n ('ERSI'(i, n) = true)
[∀k (k = n
'ETYPE'('RSI'(i, k)) = true)
[if k is odd then EC159 if
'TYPE'('RSI'(i, k)) ≠ 'ITEM'(ty)
else EC202 if 'TYPE'('RSI'(i, k)) ≠ 'SEP'(ty)]
∀k (k = n
'EPTYPE'('RSI'(i, k)) = true)
[if k is odd then EC203 if 'PTYPE'('RSI'(i, k)) ≠ 'ITEM'(ty)
else EC204 if 'PTYPE'('RSI'(i, k)) ≠ 'SEP'(ty)]]
EPTYPE(i) = true
PTYPE(i) = ty
ALT(i) = m

Function SREL(i, j, n)

PV: none
PA: integer i, j, n
IV: n/a
EF:

```
EC160 if (i < 0) ∨ (i > p1)
EC161 if (j < 0) ∨ (j > p1)
EC162 if (n < 0) ∨ (n > p1)
EC163 if 'EXISTS'(i) = false
EC164 if 'EXISTS'(j) = false
EC165 if 'ELSI'(j) = true
EC166 if 'ERSI'(i, n) = true
EC167 if 'ETYPE'(i) = true
if 'ETYPE'(j) = true then
  begin if 'EPTYPE'(i) = true then
    begin EC168 if 'EPROD'('PTYPE'(i), n, 'ALT'(i)) = false
      EC205 if 'PROD'('PTYPE'(i), n, 'ALT'(i)) ≠ 'TYPE'(j)
    end
  end
  else if 'EPTYPE'(i) = true then
    begin EC206 if 'EPROD'('PTYPE'(i), n, 'ALT'(i)) = false
      if 'EPTYPE'(j) = true then
        EC207 if 'PROD'('PTYPE'(i), n, 'ALT'(i)) ≠ 'PTYPE'(j)
    end
ERSI(i, n) = true
RSI(i, n) = j
ELSI(j) = true
LSI(j) = i
LSIX(j) = n
if 'MRS'(i) < n then MRS(i) = n
```

Function DSTR(i)

PV: none
PA: integer i
IV: n/a
EF:

EC169 if (i < 0) ∨ (i > p1)
EC170 if 'EXISTS'(i) = false
EC171 if 'LSI'(i) = true
EXISTS(i) = false
∀n ('ERSI'(i, n) = true)
[RSI(i, n) = undefined
ERSI(i, n) = undefined]
ETYPE(i) = undefined
TYPE(i) = undefined
FILLED = 'FILLED' - 1
EPTYPE(i) = undefined
PTYPE(i) = undefined
LSI(i) = undefined
ELSI(i) = undefined
LSIX(i) = undefined
MRS(i) = undefined
∀k ('ELSI'(k) = true
ELSI(k) = undefined)
[EXISTS(k) = false
∀n ('ERSI'(k, n) = true)
[RSI(k, n) = undefined
ERSI(k, n) = undefined]
ETYPE(k) = undefined
TYPE(k) = undefined
EPTYPE(k) = undefined
PTYPE(k) = undefined
LSI(k) = undefined
ELSI(k) = undefined
LSIX(k) = undefined
MRS(k) = undefined
FILLED = FILLED - 1]

Function LIST(i)

PV: true, false
PA: integer i
IV:

LIST(2) = true
LIST(6) = true
LIST(9) = true
LIST(16) = true
LIST(19) = true
LIST(31) = true
LIST(34) = true
LIST(40) = true
LIST(44) = true
LIST(45) = true
all others false

EF:

EC177 if (i < 0) ∨ (i > p1)
EC178 if 'NTERM'(i) = false

Function SEP(i)

PV: integers
PA: integer i
IV:

SEP(2) = 78
SEP(6) = 79
SEP(9) = 79
SEP(16) = 79
SEP(19) = 79
SEP(31) = 79
SEP(34) = 79
SEP(40) = 79
SEP(44) = 79
SEP(45) = 79
all others undefined

EF:

EC179 if (i < 0) ∨ (i > p1)
EC180 if 'NTERM'(i) = false
EC181 if 'LIST'(i) = false

Function PSYMB(i)

PV: integers
PA: integer i
IV: undefined
EF:
EC115 if (i < 0) ∨ (i > p1)
EC116 if 'EXISTS'(i) = false
EC117 if 'ETYPE'(i) = false
EC118 if 'SYMB'('TYPE'(i)) = false

Function SYMB(i)

PV: true, false
PA: integer i
IV: ∀k (k > 0
k ≤ 100
k ≠ 41
k ≠ 42
k ≠ 43)
[SYMB(k) = false]
SYMB(41) = true
SYMB(42) = true
SYMB(43) = true
EF:
EC119 if (i < 0) ∨ (i > p1)

Function SSYMB(i, j, k)

```
PV:   none
PA:   integer i, j, k
IV:   n/a
EF:
      EC120 if (i < 0) ∨ (i > p1)
      EC121 if (j < 0) ∨ (j > p1)
      EC122 if (k < 0) ∨ (k > p1)
      EC123 if 'EXISTS'(i) = false
      EC124 if 'ETYPE'(i) = true
      EC125 if 'EPTYPE'(i) = false
      EC126 if 'SYMB'(j) = false
      if 'ELSI'(i) = true then begin
            if 'EPTYPE'('LSI'(i)) = true then begin
                  if 'LIST'('PTYPE'('LSI'(i))) = true then
                        begin
                        if 'LSIX'(i) is odd then
                              EC190 if j ≠ 'ITEM'('PTYPE'('LSI'(i)))
                        else EC191 if j ≠ 'SEP'('PTYPE'('LSI'(i)))
                        end
                  else begin
                        EC192 if 'EPROD'('PTYPE'('LSI'(i)), 'LSIX'(i), 'ALT'('LSI'(i))) = false
                        EC193 if 'PROD'('PTYPE'('LSI'(i)), 'LSIX'(i), 'ALT'('LSI'(i))) ≠ j
                        end
                  end
            end
      ETYPE(i) = true
      TYPE(i) = j
      PSYMB(i) = k
```

Function MRS(i)

```
PV:   integers
PA:   integer i
IV:   undefined
EF:
      EC185 if (i < 0) ∨ (i > p1)
      EC186 if 'EXISTS'(i) = false
      EC208 if 'EMRS'(i) = false
```

Function NTERM(i)

```
PV:   true, false
PA:   integer i
IV:   ∀k (k ≥ 1
            k ≤ 43 )
      [ NTERM(k) = true ]
```

all others false

EF:

EC187 if (i < 0) ∨ (i > p1)

Function EMRS(i)

PV: true, false
PA: integer i
IV: undefined
EF:

EC209 if (i < 0) ∨ (i > p1)
EC210 if 'EXISTS'(i) = false

Function SSYMB(i, j, k)

```
PV:   none
PA:   integer i, j, k
IV:   n/a
EF:
      EC120 if (i < 0) ∨ (i > p1)
      EC121 if (j < 0) ∨ (j > p1)
      EC122 if (k < 0) ∨ (k > p1)
      EC123 if 'EXISTS'(i) = false
      EC124 if 'ETYPE'(i) = true
      EC125 if 'EPTYPE'(i) = false
      EC126 if 'SYMB'(j) = false
      if 'ELSI'(i) = true then begin
            if 'EPTYPE'('LSI'(i)) = true then begin
                  if 'LIST'('PTYPE'('LSI'(i))) = true then
                        begin
                        if 'LSIX'(i) is odd then
                              EC190 if j ≠ 'ITEM'('PTYPE'('LSI'(i)))
                        else EC191 if j ≠ 'SEP'('PTYPE'('LSI'(i)))
                        end
                  else begin
                        EC192 if 'EPROD'('PTYPE'('LSI'(i)), 'LSIX'(i), 'ALT'('LSI'(i))) = false
                        EC193 if 'PROD'('PTYPE'('LSI'(i)), 'LSIX'(i), 'ALT'('LSI'(i))) ≠ j
                        end
                  end
            end
      ETYPE(i) = true
      TYPE(i) = j
      PSYMB(i) = k
```

Function MRS(i)

```
PV:   integers
PA:   integer i
IV:   undefined
EF:
      EC185 if (i < 0) ∨ (i > p1)
      EC186 if 'EXISTS'(i) = false
      EC208 if 'EMRS'(i) = false
```

Function NTERM(i)

```
PV:   true, false
PA:   integer i
IV:   ∀k (k ≥ 1
            k ≤ 43 )
      [ NTERM(k) = true ]
```

all others false

EF:

EC187 if (i < 0) ∨ (i > p1)

Function EMRS(i)

PV: true, false
PA: integer i
IV: undefined
EF:

EC209 if (i < 0) ∨ (i > p1)
EC210 if 'EXISTS'(i) = false

Function ITEM(i)

PV: EC182 if (i < 0) v (i > p1)
EC183 if 'NTERM'(i) = false
EC184 if 'LIST'(i) = false

Function NALT(ty)

PV: integers
PA: integer ty
IV: NALT(1) = 1
NALT(2) = 1
NALT(3) = 1
NALT(4) = 1
NALT(5) = 2
.
.
.
all others undefined
EF:
EC175 if (ty < 0) v (ty > p1)
EC176 if 'NTERM'(ty) = false

References

[1] Parnas, D.L., "A Technique for Software Module Speifications with Examples," Communications of the ACM, May 1972. Available as a Technical Report, Computer Science Department, Carnegie-Mellon University, 1971.

[2] Chang, H.C. and D.L. Parnas, "General Purpose Macro Expander," Unpublished manuscript.

Abstract

A mechanism for holding programs in syntactic form was desired for use by systems which operate on other programs: program verifiers, automatic programming systems, and specialized text editors. The mechanism was designed using the software module specification language of D. L. Parnas, and implemented in SAIL, an Algol-like language on the PDP-10. It is suggested that specifications assist in both the design and implementation processes.

L. Robinson
5/20/73

Design and Implementation of a Multi-Level System Using Software Modules

Introduction

A mechanism for holding programs in syntactic form was desired for use by systems which operate on other programs: program verifiers, automatic programming systems, and specialized text editors. The mechanism was designed using the software module specification language of D. L. Parnas [1], and implemented in SAIL, an Algol-like language on the PDP-10.

This paper describes the process of design and implementation of this system by one person. It has been shown that software module specifications are useful in group projects [2]. This paper will demonstrate how the specifications can be useful in an individual programming effort by isolating programming problems, by allowing programming problems to be approached in an organized manner, and by simplifying the process of getting the programs to run correctly.

Motivation for a syntax-driven text editor

It has been the case in construction of programs with standard text editors, for programmers to use various gimmicks to perform "intelligent" text editing functions. For example, suppose one were constructing a LISP program and wanted to see if the parentheses matched, and how the nested S-expressions related to one another. For a large enough program one could spend hours with a standard text editor, or one could use one of the many specialized editors which take advantage of the LISP syntax. It is argued that such features would be useful in the construction of programs in other languages as well. The desired features would vary depending on the language to be edited.

Most standard text editors consider a program as a set of lines, each line containing a character string. A syntax-driven editor would treat a program as a set of lexemes related by the application of the syntactic rules of that language. If one wanted to change all instances of the identifier "A" to the identifier "B", a standard text editor (without extensions) would be of almost no use. There are text editors [3] which have been extended to permit substitution of <separator>A<separator> for <separator>B<separator>, but these editors have already made a concession to defining a lexical unit and to utilizing syntactic generality. Another use for the syntax-driven text editor is for insertion, deletion, and substitution based on a syntactic pattern. Knowing that quick changes to a program are syntactically correct saves much compile time. The syntax-driven text editor gives some of the benefits of incremental compilation for changing programs. This is especially useful in writein programs for languages with highly optimizing compilers, where compilation time is a major cost consideration.

Motivation for a program holder

For a programming system such as a syntax-driven text editor, some mechanism is necessary for inputting a program, for attributing syntax information to the program text, and for changing the text of the program in a manner limited by the syntax of the language. These capabilities are universal to a syntax-driven text editor for any language. We have specified a mechanism called a program holder which realizes these capabilities. The program holder is useful for an task which operates on programs: verification, automatic programming, compilation, and interpretation.

Description of the program holder

The program to be held in syntactic form is stored as the values of functions which describe a parse tree. The parse tree is simply an n-ary tree with specialized information at the nodes. The program holder has acces to a representation of the grammar of the language. This enables syntax checking within the module. root of a valid parse tree, one could use functions to assign this information to the node. Thus the functions of the program holder module could be divided into three disjoint sets:

1) Structural functions -- creation, deletion, and linking of nodes to form a parse tree.

2) Functions which describe the grammar.

3) Functions which assign syntactic information to the nodes.

Design evolution of a hierarchical system

The specification in [4] describes the program holder as a set of functions which comprise a single module. In the implementation these functions must map to a data structure which stores the function values, or to a procedure or macro which calculates the values. The mapping could be done by coding the module directly in a given programming language. For a module as complex as the program holder (36 functions, 200 error calls), there are two disadvantages:

1) Direct implementation would be messy to do without relying on some specific lower-level abstractions.

2) The direct implementation of a complicated system may generate design decisions which are very difficult to change.

The approach which we chose was to decide which lower level abstractions are needed to implement the top level module (in this case the program holder), and to write software module specifications for these lower level abstractions. This process continues until the designer judges that the mapping from the current lowest level to a program is straightforward.

At this point the designer has specifications for the different layers or abstractions in his system. The specifications for various modules are independent of one another, in the sense that the specifications make reference only to functions in the same module. The process of creating mappings between different abstractions (or virtual machines) constitutes the implementation of the system.

Example of the hierarchical design process -- syntax-driven text editor

At first, the design effort involved an enumeration of the kinds of operations desired of the text editor:

1) Input of programs or parts of programs.

2) Internal storage of program in parsed form.

3) Searching and substitution on strings classified by syntactic type.

4) Ability to work with incomplete programs for the purposes of stepwise refinement.

It was only after examination of the desired capabilities that we determined that the editor should be built around the storage of programs in parsed form, rather than around any particular capability. The other parts of the system would make use of this program storage mechanism and could therefore be built later. The mechanism for such storage was called the program holder; other parts of the editor would be a parser, a lexical analyzer, an input/output module, and a pattern matching and substitution module (see figure 1).

Several design decisions went into the specifications for the program holder. Here is a list of decisions which were made in approximately chronological order:

1) Representation of the program as a parse tree [excludes representation as an unparsed string of tokens].

2) The structural functions of the parse tree should be those of an n-ary tree where n is variable [rules out restriction of n-ary tree with n fixed, specifically a binary tree].

3) The assignment of syntax information to the nodes required two design decisions:
   a) To include a representation of the grammar in a submodule of the program holder module.
   b) To specify automatic syntax checking as part of the holder.

It turned out that a) and b) could be incorporated into specifications which describe a program holder for any context-free grammar, instead of having a different set of specifications for each grammar dealt with. This was a generalization beyond our initial goals.

4) For special applications (working with incomplete programs and top-down parsing) a second type of syntax assignment -- incomplete syntax assignment -- was incorporated into the specifications. This mechanism will detect errors if an incomplete program is syntactically inconsistent.

5) A special type of nonterminal was provided, which was defined as a list of zero or more instances of another syntactic type.

These factors were all considered in the development of a set of specifications. Next it was time to consider the implementation of the program holder in terms of lower level abstractions. The major problem was how to implement the n-ary tree, which was the structural basis for the program holder. Perhaps the most flexible means was to devise specifications for a list processing system which could allocate and link elements of arbitrary size. This decision allowed many possible formats for the nodes of the tree and permitted various schemes to specify the linkage of one node to an arbitrary number of other nodes (a requirement for an n-ary tree). List processing systems require some means of dynamic storage allocation, so that the third (lowest level) abstraction to be specified was that of a storage allocator mechanism.

At this point the design consisted of three module specifications which were independent of one another. The implementation then consisted only of the designer's idea of the mappings between the levels of abstraction; these mappings were the last design decisions to be made: they were still flexible at a time when the specifications were relatively fixed. Although the specifications for this three-level system were written from the top down, the implementations were were written in reverse order, to facilitate the debugging and testing of the system. (If the higher of two levels is to be tested, one needs either an implementation of the lower level or a simulation of it. Since the lower level implementation had to be built anyway, it was natural to implement this system from the bottom up.) Table I describes the implementation of the multi-level system in terms of design decisions. The design decisions are divided into the following categories: tasks to be performed, information to be exchanged with the outside of the module (or program), information hidden inside the module, information unknown to the module (which must be kept outside), and the implementation of the module in terms of lower level abstractions. A detailed description of the program holder specification can be found in [4].

In the syntax-driven text editor the design phase consisted of several distinct processes:

1) Consideration of the entire problem in terms of a set of capabilities for the finished product.

2) Decomposition of this practical problem into one or more collections of shared information (incorporated with operations upon it). Each collection becomes a software module. Examples:
   a) Building the editor around a program holding mechanism.
   b) Building the n-ary tree of the program holder from a list processing system.
   c) Building the list processing system from a storage allocation mechanism.

3) Writing the formal specifications for a module once its role has been identified.

4) Implementaion of the module specifications, either in terms of lower-level abstractions or directly in a programming language.

Of the above processes, 1) and 4) are common to all methods of program design. 2) is an extremely difficult process, because there are many modular decompositions which will describe a particular system -- some of which are very poor from the viewpoint of information hiding. The Parnas specifications [1] assist in process 2), because they limit the effective size of a given module by forcing the designer to enumerate all external connections (functions) and internal state changes (effects). In our experience bad decompositions yield large, unwieldy specifications. Once a decomposition has been arrived at, process 3) is rather mechanical, involving many small decisions. Given the module specifications from process 3), process 4) is easy. However, it is a difficult task to produce an efficient implementation of a modular program, because the modules limit the potentially destructive but efficient "tricks" which the programmer can perform. There will always be some efficiency sacrificed in the modularization of a programming system. We expect that higher reliability and increased efficiency from improved modular programming techniques may minimize complaints about inefficiency of modular programming with formal specifications.

Variability of design decisions using the modules

Several questions come to mind concerning the constraints imposed by the use of the modules, and the ability to change design decisions already made:

1) How large should a module be?

2) How difficult is it to change design decisions in the specification stage?

3) How difficult is it to change design decisions in the implementation stage?

There is no clearcut answer to the first question. In the case of the program holder two separate abstractions were present - the parse tree and the representation of the grammar. If these two abstractions were in separate modules, the syntax checking desired in the parse tree could not be explicity included in the parse tree module (because a given software module specification must not make references to functions in another module). Thus the parse tree and the representation of the grammar were kept together in one large module so that explicit syntax checking could be included in the specification. In other cases, a design decision might indicate the breakdown of a large module into several smaller ones.

Changing design decisions during the specification process is easy, because the specification process involves the writing of a "first draft" of a specification, and then refining the original work. Besides the maximum flexibility during this phase (one is not burdened with having started the implementation concurrently), the information contained in the specification makes it easy to enumerate design decisions, to decide appropriate changes, and to make the desired changes in the specifications. One method of determining whether or not the specifications are adequate is to write programs (or flowcharts) of desired operations using the modules. It becomes apparent which desired capabilities are impossible, or at least difficult, to accomplish. When such difficulties occur, changes in the specifications are indicated.

When the specifications have become fixed and used in implementing other modules, some design decisions are difficult to change. The easiest decisions to change are the ones which involve no changes in the specifications, because such changes are invisible outside the module in which the changes are made. An example of this was a decision to completely rewrite the implementation of the storage allocator module (bottom level of the program holder). The original implementation had been running far too slowly, and major changes seemed necessary. Since the desired changes could be made without any change in the specifications, it was unnecessary to change any of the code for the higher levels. Usually a change in the lowest level of a system which is not formally specified will result in havoc with the upper levels. With the storage allocator, the change was made easily, without the higher levels being affected.

Making a change which does affect the specifications is far more serious and should be avoided wherever possible. In a group project such a change may propagate errors throughout the system.

Writing code for the system

The language chosen for implementation of the program holder was SAIL (Stanford Artificial Intelligence Language) [5], on a Digital Equipment PDP-10. The language is essentially Algol-60 with separately compiled outer blocks, call-by-reference instead of call-by name, a macro facility, and built-in I/O routines which interface well with the machine. The presence of separately compiled outer blocks was especially helpful for the following reasons:

1) A block could contain all the code for the implementation of a single module. In one module, those procedures whose text is written inside the block, but whose names would be available outside the module (as functions) would be declared "INTERNAL." Any separately compiled block (another module) which wanted to use the procedures must declare them "EXTERNAL" (i.e. supplied from outside) at the outermost block level in which they are used.

2) Different implementations of a module could be trivially "plugged in" without recompiling.

Error call implementation

When an implementation of a function from a module specification is written, tests are made for all the error conditions in the "EFFECTS" section of the function. Then the function's value is caluclated, or the state of the module is changed as indicated in the specifications.

In the implementation of a multi-level system, error checking on a higher level may prevent an incorrect function call at a lower level. Nevertheless, error checking at a lower level is still carried out, causing a large amount of time to be spent in rarely useful error checking. Since many errors on a higher level correspond to errors on a lower level (such as an out-of-bounds argument), some multi-level error checking is redundant. These difficulties can be handled in two ways:

1) The lower-level error calls can be eliminated. This would increase effieiency, but would require the "custom tailoring" of a module implementation every time a higher level was added. In addition this solution assumes that the implementation of the next highest level is correct, in the sense that it does not make any incorrect calls to the lower level. This assumption cannot always be made. If an incorrect call were made of a lower-level function which had no error checking, recovery from (or even detection of) the error would be impossible.

2) Instead of making a time-consuming test for an error condition, each upper-level error call would set a trap location for a certain lower level error call. This would allow the errors to be detected at a lower level but recovery could be specified by the highest level at which the error could have occurred. Parnas [6] has suggested the feasibility of this method of handling error calls in a multi-level system. This system will be implemented for the program holder in the near future.

Note that only in the second case do all modules meet their specifications.

Program testing and verification using the specifications

Having software module specifications for a program can simplify the process of testing or verifying the progam. In program testing the specifications provide a complete description of the classes of external program behavior, so that writing a test program for these external cases can be done directly from the specifications. Of course the module speicfications do not provide any indication of the classes of internal behavior of the program to be tested. In most cases some analysis of the implementation algorithms is necessary in order to determine and test internal cases which are not manifested on the outside. However, the specifications have eliminated much of the work in deciding what to test for and where to test it.

In the case of verification of an implementation, one can interpret that each statement in the "EFFECTS" section of a function specification corresponds to an assertion. One can verify that the desired effect has taken place by inspection of the part of the function implementation which corresponds to that particular effect. Care must be taken to ensure that an assertion which is true at a given point in the function implementation is also true on exit from the function call. Considering the specifications as assertions is also useful for more formal verification procedures. The specifications constitute a complete set of assertions, so that a programmer is less likely to "forget" about an important property of the program if he can refer to the specifications. Such forgetfulness is likely to occur when large programs are being written.

Note that when the behavior of simple functions is established (either by testing or verification), this behavior forms a base from which other functions related to the simple ones may be tested or verified. Picking the correct order can speed the testing or verification of an entire module.

The following results were obtained with the program holder implementation:

1) For each of the three modules written, the times for coding a module and testing/verification were approximately equal. The specification time for the two lower-level modules was approximately equal to that of coding them. The top level of the program holder, in which many design decisions were made, took twice as long to specify as to code.

2) For the total system (approximately 2300 lines of SAIL code), in 5 months there has been only one "bug" detected after the testing/verification procedure (an error was detected by the wrong error checking statement).

Conclusion

One big problem in software development is that the total time spent in the coding and debugging of large systems is exponential in the size of the system. Software module specifications, because they fragment the programming problem and because debugging time seems to be reduced, may be a way to help reduce the magnitude of the "software crisis." This conjecture has yet to be rigorously tested in a large-scale software effort. There is a group experiment which indicates that the presence of software module specifications can result in the successful interface of programs written by many different inexperienced programmers [2]. The experience of one person in the implementation of the program holder mechanism suggests that the module specifications are also useful in a one-man effort of writing a large system.

Table I -- Design decisions in syntax-driven text editor

Program	Editor
Type of specification	Informal -- in terms of tasks that the editor should perform
Design decisions	
I. Tasks to be performed	1. Should be able to perform all standard text editing functions (smallest unit may be a lexeme instead of a character). 2. Specialized text editing functions dependent on the syntax of the program a) Checks for syntax errors b) Can match on a pattern of terminals and nonterminals c) Indenting d) Printing of parse tree e) Working with incomplete, but syntactically consistent programs 3. Original intended use -- to help generate the module specifications
II. Information exchanged with outside of module	Input Program text Grammar (on initialization) Editor commands Output Altered program text Formatted program text Parse tree Editor responses and diagnostics
III. Information hidden within module	1. Internal representation of program 2. Parsing algorithm 3. Symbol table format 4. Horizontal decomposition at lower level
IV. Information unknown to module (which must be kept outside)	not finally decided
V. Implementation of module in terms of lower level abstractions	Tasks performed by algorithmic modules I/O Searching, substitution, and pattern matching Parsing Information stored in Parnas type modules Program holder Lexical analyzer

Table I (continued)

Program	**Program holder**
Type of specification	**Parnas (formal) specification**
Design decisions	
I. Tasks to be performed	1. Process any context-free language 2. Representation of program as parse tree (n-ary tree) 3. Grammar -- fixed at initialization 4. Trees can be built up and deleted dynamically 5. Syntax assignment a) Automatic syntax checking ensures valid parse below b) Representation of grammar must be internal to module 6. Need incomplete syntax assignment a) Top down parsing b) Stepwise refinement of programs 7. Consideration of a special class of nonterminal defined as a list of symbols
II. Information exchanged with outside of module	Input Creation and deletion of nodes Data associated with nodes Grammar -- at initialization Output Existence of nodes Relations between nodes Data assigned to nodes Productions of grammar
III. Information hidden within module	1. Manner in which nodes are given values 2. Internal representations of parse tree and productions of grammar 3. Algorithm for syntax checking 4. Means of determining existence of nodes and whether data in nodes is defined
IV. Information unknown to module (which must be kept outside)	1. Node indices not referred to in context (i.e. pointers into the tree are required) 2. Correspondence of integers to syntactic types 3. Symbolic information associated with the nodes (i.e. a symbol table is required)
V. Implementation of module in terms of lower level abstractions	1. Grammar is stored in an array with links and is accessed by macros 2. Program holder is implemented by calls to functions of list processing module a) Nodes in parse tree correspond to list elements b) Pointers to sons are stored in blocks linked to father node

Table I (continued)

Program	List processing system
Type of specification	Parnas (formal) specification
Design decisions	
I. Tasks to be performed	1. Create and destroy list elements 2. Can specify the number of fields in each element created 3. Can determine from outside whether or not a field has been given a value 4. Can set values of fields
II. Information exchanged with outside of module	Input Creation and deletion of elements Values of fields Output Existence of elements Existence of a value for a field in an element Values of fields
III. Information hidden within module	1. Means of determining the existence of elements 2. What value constitutes an undefined field 3. Method of allocating and choosing indices for elements
IV. Information unknown to module (which must be kept outside)	1. Whether a field is a pointer or data 2. Indices of elements referred to out of context
V. Implementation of module in terms of lower level abstractions	List processor calls storage allocator module a) undefined = 2↑34 - 1 b) Indices of nodes are "addresses" in storage allocator

Table I (continued)

Program	Storage allocator
Type of specification	Parnas (formal) specification
Design decisions	
I. Tasks to be performed	1. Allocate and free blocks of variable size 2. Desire knowledge of whether or not an address is the head of a block (blocks are referred to by address of head) 3. Desire knowledge of whether or not an address is free or allocated 4. Can only free storage which has been allocated as a block
II. Information exchanged with outside of module	Input Size of block for allocation Address of block for freeing Output Whether an address is free or allocated Whether an address is head of a block Whether an address is used by allocator for bookkeeping
III. Information hidden within module	1. Keeping track of free and allocated blocks 2. Determination of heads of blocks 3. Storage allocation strategy
IV. Information unknown to module (which must be kept outside)	1. Compacting process (when free storage becomes tight) 2. Reference to block heads out of context (calling program must remember the addresses of blocks it has allocated)
V. Implementation of module in terms of lower level abstractions	1. Direct implementation in SAIL 2. First fit allocation strategy 3. Bit matrix to determine free storage 4. Storage to be allocated is SAIL integer array

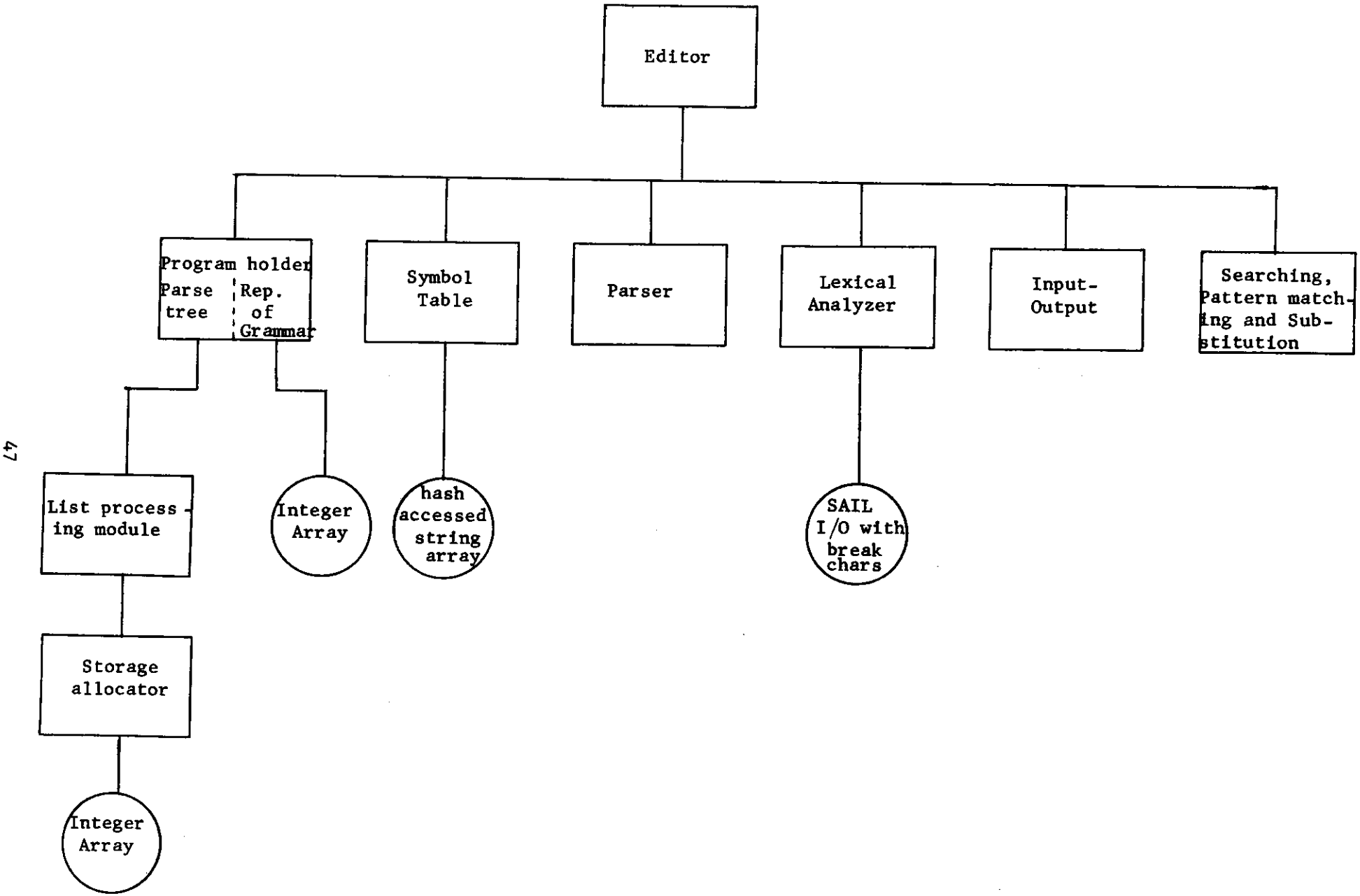


Figure 1 - Structure of a Syntax-Driven Text Editor

References

til Parnas, D.L., "A Technique for Software Module Specifications with Examples." Communications of the ACM, May 1972. Available as a Technical Report, Computer Science Department, Carnegie-Mellon Univereity, 1971.

[21 Parnas, D.L., "Some Conclusions from an Experiment in Software Engineering Techniques," Proceedings FJCC, 1972. Available as a Technical Report, Computer Science Department, Carnegie-Mellon Univereity, 1972.

[31 Savitzky, Steven, Son of Stopgap, Stanford Artificial Intelligence Laboratory, Operating Note 50.1, September, 1969.

[43 Robinson L. and D.L. Parnas, "A Syntax-Driven Text Editor," Technical Report, Computer Science Department, Carnegie-Mellon University, 1973.

[5] Swinehart D. and R. Sproull, SAIL Manual, Stanford Artificial Intelligence Project, Operating Note No. 57, November 1969.

[61 Parnas, D.L., "Response to Detected Errors in Well- Structured Programs," Technical Report, Computer Science Department, Carnegie-Mellon University, 1972.