129 F50 TK 52.291

K

KFKI-76-11

L. VARGA

THE ABSTRACTIONS OF MACHINE DEPENDENT PROGRAM FORMS

Hungarian Academy of Sciences

CENTRAL RESEARCH INSTITUTE FOR PHYSICS

BUDAPEST

2017

KFKI-76-11

THE ABSTRACTIONS OF MACHINE DEPENDENT PROGRAM FORMS*

L. Varga

Central Research Institute for Physics, M.T.C. Division
H-1525 Budapest, Box 49

*Presented at the colloqium of Institut für Informatik, Universität Stuttgart, 1975

ISBN 963 371 111 8

ABSTRACT

The Vienna Definition Language /VDL/ may be used for defining the common, machine independent structure and meaning of machine dependent program forms. In this paper an abstraction of a subclass of the relocatable binary forms is specified by a structured model of an abstract linkage editor, which can be regarded as a proposed standard. The structured model is given in VDL and its correctness is proven.

АННОТАЦИЯ

Венский язык определения семантики/VDL/ можно использовать для определения общих, машино-независимых характеристик машино-зависимых форм программ. В данной статье определено машино-независимая абстракция класса перемещаемых двоимных форм с помощю структурного алгоритма абстрактной программы редактора связи. Это еще можно изпользовать, как стандарт определенных типов перемешаемых двоимных форм. Мы задаем доказательство правильности данной структурной модели.

KIVONAT

A Bécsi Definiciós Nyelv megfelelő módositással felhasználható a géptől függő programformák géptől független közös jellemzőinek definiálására. Ebben a tanulmányban az áthelyezhető bináris formák egy osztályának géptől független absztrakcióját definiáljuk egy absztrakt szerkesztőprogram strukturált algoritmusával. Ez bizonyos tipusu áthelyezhető bináris formák szabványaként is számitásba jöhet. A tanulmány tartalmazza a megadott strukturált modell helyességének a bizonyitását is.

1. INTRODUCTION

The program forms, according to their intended purpose, can be grouped into three categories:

- source language forms,
- internal program forms,
- machine code forms.

Their concrete representations are known as machine dependent program forms.

The <u>source program form</u> has to serve the programmer's comfort. For this reason most programming languages are problem oriented. The problem oriented programming languages contain only a few machine dependent elements. Therefore they can be easily defined in a machine independent way. On the other hand, machine oriented languages have a lot of machine dependent elements. Therefore these languages in general are specified in terms of concrete computers.

The <u>internal program forms</u> are created by translators or other system's programs for further processing. Such program forms are for example the wellknown relocatable binary or the absolute binary forms. The specifications of these program forms are also given in terms of concrete computers.

The <u>machine code forms</u> are created for direct execution or interpretation. They have the most machine dependent characteristics, and they are defined in a machine dependent way.

One can give an abstraction of a concrete program form of a given computer. On the other hand, it is also possible to study abstractions of subclasses of one of the three program forms mentioned above. For example, there can be constructed an abstract model of the assembly languages of different machines. This type of abstraction contains the common, machine independent structure and meaning of the elements of a given subclass. We intend to discuss the abstraction in this sense.

The importance of this kind of abstraction is supported by the followings:

1. The abstraction - like that of the high level languages - makes the standardization of machine dependent program forms also possible. In this case a concrete program form can be respected as an implementation of the standard form.

2. The standardization of the program forms facilitates the development of translators and other system programs in a machine independent fashion, down to the lowest possible level. Such a system program can be easily implemented on different computers.

3. The abstraction is of great importance when teaching system programming. It helps the student to grasp the common features of similar program forms.

The operational meaning of a program can be formally defined in two wellknown ways:

- by an interpreter,
- by a translator.

We use the latter method. Assuming that the meaning of the machine code form is given, the meaning of the rest of the program forms can be defined by means of translators. For example, according to the systems existing in practice, for an assembly language we can construct two abstract translators, namely an

- abstract assembler, and an
- abstract linkage editor.

The abstract linkage editor deduces the semantics of the abstract relocatable binary form to the semantics of the abstract machine code.

The abstract assembler defines how the abstract assembly form can be translated into abstract relocatable binary form.

The meaning of abstract machine code form can be defined by an abstract interpreter. But the semantics of the machine code form may be defined on a number of different levels. On the lowest level the meaning of a machine code form is an algorithm for computing values resulting from the execution of the program on its input data. On higher level, however, these activities involve functions, which are executed by the operating system. Thus the abstraction of the machine code form leads to the abstractions of algorithms existing in operating systems.

The abstract models - like the formal definitions of high level languages - can be specified using the Vienna Definition Language, which can be extended as so it is suitable for the formal description of all the systems programs.

From the machine dependent program forms, a formal description of a certain family of assembly languages was given in [4]. In this paper an abstraction of a subclass of the relocatable binary forms is specified by an abstract linkage editor, which can be regarded as a proposed standard. This standard specifies, first of all, the common characteristics of linkage editors existing on mini computers.

A structured model of the abstract linkage editor is given using three decision levels. The abstract syntax and semantics of the relocatable binary program segment and statement are specified on the first, second and third decision level respectively.

An excellent tutorial treatment of linkers and loaders can be found in [5]. The present paper is supported by these results.

For the discussion of the Vienna Definition Language see for instance [1], [2], [3].

2. ABSTRACT LINKAGE EDITOR

The linkage editor is responsible for linking the relocatable binary segments together to form a complete program and transforming it into absolute binary or machine code form. These two functions can be specified separately or together. In the first case linking is carried out after translation but before load time. This method is called indirect loading. Linking can be carried out along with the loading too. This method is known as direct loading. Only indirect loading will be discussed here.

As it is known, a linking process can be accomplished in one or two passes. From theoretical standpoint the one and two pass linkers are equivalent. Only the abstraction of a one pass linkage editor is described here.

It is supposed, that the modules of the program are loaded in the main memory one by one without gaps, and each module occupies a continguous area.

Definition 0.1. The state of an abstract linkage editor is determined by the following VDL object:

```
is-state = (<s-input:is-r/b-library>,
            <s-output:is-bin-program>,
            <s-table:is-G-table>,
            <s-basis:is-integer>,
            <s-rest:is-stmt-list>,
            <s-control:is-control>)
```

where the immediate components are further detailed by definitions 0.2-0.5.

Axiom 0.1. The r/b-library is a set of r/b segments, where each r/b-segment can be selected by its name:

```
is-r/b-library = ({<s:is-r/b-segment>|is-segment-name(s)})
```

Axiom 0.2. A binary program is an ordered set of statements. Each statement specifies the deposition of an absolute value to an absolute address. The value is an integer. In terms of VDL:

```
is-bin-program = is-abs-stmt-list
is-abs-stmt = (<s-address:is-integer>,
              <s-value:is-integer>)
```

Axiom 0.3. The r/b segments communicate with each other by global names.

Definition 0.2. The G-table contains an entry for each global name. An entry specifies the value of the corresponding global name. The value of a name may be definite or indefinite. The definite value is an integer.

```
is-G-table = ({<s:is-value>|is-global-name(s)})
is-value = is-integer v is-undefined
is-global-name = is-name
```

Definition 0.3. A global name having a definite value is called predefinite name. Let be

predefined(n) = is-integer(n(s-table(ξ))),

where

is-name(n) = TRUE.

A global name having an undefined value is called postdefinite name. Let be

postdefined(n) = is-undefined(n(s-table(ξ))),

where

is-name(n) = TRUE.

Axiom 0.4. The basis is an address in the main memory, from which the current r/b-segment is to be loaded. It is an integer:

is-integer(s-basis(ξ)) = TRUE.

Definition 0.4. The statements of the r/b-segment, which contain post-definite names are stored in a VDL list waiting for the value of the names. The component s-rest(ξ) containes these statements.

Definition 0.5. The component s-control(ξ) contains the control tree during the translation.

Definition 0.6. The initial state ξ_0 of the abstract linkage editor consits of the following objects:

a) The r/b-library contains all the r/b-segments to be processed.
b) is-<>(s-output(ξ_0)) = TRUE.
c) The G-table contains all the root names (see definition 1.2) of the program as postdefinite names, that is if

   n $\in$ s-table(ξ_0) ,

   then

   postdefined(n) = TRUE.
d) The component s-basis(ξ) contains the address of a memory area available for the program.
e) is-<>(s-rest(ξ_0)) = TRUE
f) s-control(ξ_0) = link-program

Decision level 1.

Syntactic definition

Axiom 1.1. The r/b-program is a connected set of r/b-segments. The connections are estabilished by global names. An r/b-segment contains at most two kinds of global names:

- entry name,
- external name.

Accordingly, an r/b-segment has three components:

- a list of entry names,
- a list of external names,
- a code part.

In terms of VDL:

```
is-r/b-program = ({<s:is-r/b-segment>|is-segment-name(s)})
is-r/b-segment = (<s-entries:is-entry-name-list>,
                  <s-externals:is-external-name-list>,
                  <s-code-part:is-code-part>)
is-entry-name = is-name
is-external-name = is-name
```

Axioms 1.2, 1.3, 1.4 (see below) are postulated.

<u>Definition 1.1.</u> If

$$\text{is-r/b-program}(p) = \text{TRUE}, \quad t_1 \in p, \quad t_2 \in p, \quad (t_1 \neq t_2)$$

and

$$(\exists i)(\exists j)(\text{elem}(i)(\text{s-externals}(t_1)) = \text{elem}(j)(\text{s-entries}(t_2)))$$

where

$$1 \leq i \leq \text{length}(\text{s-externals}(t_1))$$

and

$$1 \leq j \leq \text{length}(\text{s-entries}(t_2)),$$

then we say that t_1 <u>refers to</u> t_2 by n and write

$$t_1 \overset{n}{\dashrightarrow} t_2 \ ,$$

where

$$n = \text{elem}(i)(\text{s-externals}(t_1)).$$

Definition 1.2. If

$$\text{is-r/b-program}(p) = \text{TRUE}, \qquad t_1 \in p,$$

and there exists at least one n such that

$$n = \text{elem}(i)(\text{s-entries}(t_1)) ,$$

where

$$1 \leq i \leq \text{length}(\text{s-entries}(t_1))$$

but there exists no t_2 and j pair for which $t_2 \in p$ and

$$\text{elem}(i)(\text{s-entries}(t_1)) = \text{elem}(j)(\text{s-externals}(t_2)) ,$$

then the name n is called root-name and the r/b-segment t_1 is said to be a master segment:

$$\text{is-root-name}(n) = \text{TRUE} ,$$
$$\text{is-master}(t_1) = \text{TRUE}.$$

Definition 1.3. If

$$\text{is-r/b-program}(p) = \text{TRUE}, \quad t_i \in p, \quad i=1,2,\dots,k,$$

and

$$t_1 \overset{n}{-\!\!\rightarrow} t_2 \overset{m}{-\!\!\rightarrow} \dots \overset{r}{-\!\!\rightarrow} t_k$$

then we say that there exists a reference path from t_1 to t_k in p, or t_k is accessable from t_1. It is written as follows:

$$t_1 -\!\!\rightarrow^* t_k$$

Axiom 1.2. If

$$\text{is-r/b-program}(p) = \text{TRUE}, \quad t_1 \in p,$$
$$\text{length}(\text{s-externals}(t_1)) \neq 0$$
$$n = \text{elem}(i)(\text{s-externals}(t_1)),$$

where

$$1 \leq i \leq \text{length}(\text{s-externals}(t_1)),$$

then there exists just one $t_2 \in p$, where $t_1 \overset{n}{-\!\!\rightarrow} t_2$

Axiom 1.3. If

$$\text{is-r/b-program}(p) = \text{TRUE},$$

then

$$(\exists t \in p)(\text{is-master}(t) = \text{TRUE})$$

Axiom 1.4. If

$$\text{is-r/b-program}(p) = \text{TRUE}, \quad t \in p,$$

and

$$\text{is-master}(t) = \text{FALSE},$$

then

$$(\exists m)(m \rightarrow {*t}),$$

where

$$\text{is-master}(m) = \text{TRUE}.$$

Theorem 1.1. If

$$\text{is-r/b-program}(p) = \text{TRUE}, \quad t \in p,$$

then

$$\text{s-entries}(t) \neq \langle\rangle$$

Proof. If is-master(t) = TRUE, then the statement follows from Definition 1.2. Otherwise, Axiom 1.4. states that there exists a t' which refers to t and therefore s-entries(t) $\neq \langle\rangle$.

Semantic definition

Definition 1.4. Let

$$\text{postdef}(x), \quad /\text{is-G-table}(x) = \text{TRUE}/$$

be a function such that if

$$n = \text{postdef}(\text{s-table}(\xi))$$

then

$$\text{postdefined}(n) = \text{TRUE}$$

whenever

$$\{s \mid \text{is-undefined}(s(\text{s-table}(\xi)))\} \neq \emptyset ,$$

and

$$\text{postdef}(\text{s-table}(\xi)) = \text{NIL}$$

otherwise.

Informally, the function postdef applied to the G-table furnishes a postdefinite name, whenever such a name exists, and furnishes the object NIL otherwise.

<u>Definition 1.5.</u> Let

segment-name(x), /is-name(x) = TRUE/

be a function such that

is-segment-name(segment-name(x)) = TRUE

and

$(\exists\, i)(\text{elem}(i)(\text{s-entries}(t)) = x)$,

where

$t = \text{segment-name}(x)(\text{s-input}(\xi))$.

Informally, the function segment-name applied to a global name x furnishes the name of the segment that contains x as an entry name.

<u>Assumption 1.1.</u> Let us supposes that the macro

<u>process-segment</u>(t), /is-r/b-segment(t) = TRUE/

executes the following processes in due succesion:

a) Each entry name of t gets value using the value s-basis(ξ). At the same time the entry name in question becomes predefinite in the G-table.

b) Those external names of t which have no corresponding element in the

G-table, are entered as postdefinite names into the G-table.

c) The code part of the r/b-segment t is translated into an appropriate absolute binary form using the value s-basis(ξ).

d) The basis value of the next r/b-segment is calculated.

<u>Axiom 1.5.</u> Linking r/b-segments together to create a complete binary program means the application of the macro

<u>process-segment</u>(t)

to each r/b-segment of the given r/b-program in arbitrary order.

<u>Theorem 1.2.</u> The following program links the r/b-segments together, and forms a complete binary program, which is defined by the initial state ξ_0.

```
link-program =
    postdef(s-table(ξ)) = NIL → null
    T → link-program;
            process-segment(a);
                a:next-segment(b);
                    b:next-postdef-name

next-postdef-name =
    PASS:postdef(s-table(ξ))

next-segment(n) =
    PASS:segment-name(n)(s-input(ξ))
```

<u>Proof.</u> Let us prove that the control tree is reduced to the instruction <u>null</u> if and only if the linking procedure has been finished.

<u>If condition</u>: let us suppose that all the r/b-segments of the given r/b-program have been processed by the macro

<u>process-segment</u>

Then all the names of the r/b-program have been entered to the G-table either as a root name or as an entry name or as an external name.

The root names become predefinite as a result of processing the master segments.

The entry names were set predefinite in the G-table by definition.

From Axiom 1.4. it follows, that each external name is defined as an entry name in another r/b-segment of the given program. Thus processing this r/b-segment made the external name in question predefinite.

Hence, when all the r/b-segments of a given program have been processed, the G-table contains only predefinite names and the control tree is reduced to the instruction <u>null</u>.

<u>Only if condition</u>: let us suppose that the control tree has been reduced to the instruction <u>null</u>. Then the G-table does not contain any postdefinite name. But in accordance with definition 0.6/c, when starting the process, the component s-table(ξ_σ) does contain postdefinite names, that is the root names of the program.

Now let us consider the algorithm to be proven.

1. Clearly, the instruction <u>next-postdef-name</u> always furnishes a postdefinite name if the G-table contains such a name at all.
2. The instruction <u>next-segment</u> always selects that r/b-segment, which defines the value of the name furnished by the instruction <u>next-postdef-name</u>.
3. Finally, the instruction <u>process-segment</u> is always carried out.

This procedure is repeated until the G-table does contain any postdefinite name. Hence, when the G-table does not contain any postdefinite name, all the master segments must have been processed. But in this case, all the segments accessable from one of the master segments also must have been processed. Since all the segments of the program are accessable from at least one master segment, all the segments of the program also must have been processed. This completes the proof.

Decision level 2.

Syntactic definition

Axiom 2.1. The code part of the r/b-segment consists of:

- a set of label definition statements, which specify the values of the entry names defined within the segment,
- an ordered set of load statements, which specify the data to be loaded and their memory addresses,
- the length of the segment in memory.

Formally:

$$\begin{aligned}\text{is-code-part} &= (\langle\text{s-label-def:is-label-def}\rangle,\\ &\quad\ \langle\text{s-code-def:is-stmt-list}\rangle,\\ &\quad\ \langle\text{s-length:is-integer}\rangle)\\ \text{is-label-def} &= (\{\langle\text{s:is-integer}\rangle \mid \text{is-name(s)}\})\end{aligned}$$

Semantic definition

Assumption 2.1. Let

$\underline{\text{process-stmt}}(t)$, /is-stmt(t) = TRUE/

be a macro, which processes the load statement t in the following way:

a) The load statement, that does not contain any postdefinite name is translated into an ppropriate load statement of absolute binary form using the values of the predefinite names and the actual basis.

b) The load statement, that contains postdefinite names is preserved for a later process /see definition 0.4./ transforming it into an intermediate form, such a way, that it does not contain anymore the actual basis as unknown parameter.

Lemma 2.1. Let be

is-data-list(l) = TRUE

then the

```
    process-data-list(l) =
          length(l) = 0 → null
          T → process-data-list(tail(l));
                    process-data(head(l))
```

algorithm executes the instruction process-data exactly once for each element of list l.

Proof. This results from the definition of the functions "head" and "tail".

Theorem 2.1. The following program executes the process specified by assumption 1.1.:

```
    process-segment(t) =
          process-code-part(s-code-part(t));
                process-externals(s-externals(t)).
                process-entries(s-entries(t),s-code-part(t))

    process-entries(l,t) =
          length(l) = 0 → null
          T → process-entries(tail(l),t);
                    set-table(head(l),head(l)(s-label-def(t))

    set-table(n,v) =
          s-table:μ(s-table(ξ);<n:v>)

    process-externals(l) =
          length(l) = 0 → null
          T → process-externals(tail(l));
                   process-ext(head(l))

    process-ext(n) =
          is-integer(n.s-table(ξ)) → null
          T → set-table(n,undefined)
                             /undefined ∈ is-undefined/
                                               ^

    process-code-part(t) =
          update-basis(s-length(t));
             process-stmt-list(l);
                   delete-rest;
                      l:pass(s-code-def(t)⌢s-rest(ξ))
```

```
pass(t) =
    PASS:t

delete-rest =
    s-rest:<>

process-stmt-list(l) =
    length(l) = 0 → null
    T → process-stmt-list(tail(l))
            process-stmt(head(l))

update-basis(v) =
    s-basis:s-basis(ξ)+v
```

Proof. Let is-r/b-segment(t)=TRUE. By lemma 2.1 clearly the macro

process-entries(s-entries(t),s-code-part(t))

executes the statement

set-table(n,n(s-label-def(s-code-part(t)))

for each entry name n defined by t. Therefore the assumption 1.1./a is realized.

Similarly by lemma 2.1. the macro statement

process-externals(s-externals(t))

realizes assumption 1.1./b.

Obviously the assumption 1.1./c could have been reduced directly from lemma 2.1., assumption 2.1./a and assumption 2.1./b taking into consideration the algorithm of

process-stmt-list(s-code-def(s-code-part(t)) $\frown$ s-rest(ξ))

Clearly the assumption 1.1./d is realized by

update-basis(s-length(s-code-part(t))).

Finally, we have to proof, that the order in which the statements are given is proper, Obviously the order of the execution of the statements

<u>process-entries</u>
<u>process-externals</u>

is arbitrary. But the statement <u>process-entries</u> uses the actual basis value and therefore its execution has to precede the execution of statement <u>process-code-part</u>. Similarly, the execution of <u>process-stmt--list</u> also has to precede the execution of statement <u>update-basis.</u>

<u>Decision level 3.</u>

<u>Syntactic definition</u>

<u>Axiom 3.1.</u> A load statement of the r/b-segment contains an address and a data component:

```
is-stmt = (<s-address:is-address>,
           <s-data:is-data>)
```

where

a) An address may be an absolute address or a relative address.The latter is an expression which contains the segment basis as a parameter:

```
is-address = (<s-type:is-type>,
              <s-value:is-value>)
is-type = is-abs v is-b-rel
is-value = is-integer v is-expression
```

b) A data, which is to be loaded may be
- an absolute value,
- an expression containing the segment basis as parameter,
- an expression containing an external name as parameter,
- an expression containing an external name and the segment basis as parameters.

Formally:

```
is-data = (<s-type:is-d-type>
           <s-value:is-d-expression>)
is-d-type = is-abs v is-b-rel v is-ext-rel v is-b-ext
is-d-expression = (<s-basis:is-name>,
                   <s-expression:is-expression>) v is-expression
```

Semantic definition

Definition 3.1. Let

calculate(n,e)

be a function, that calculates the value of the expression e using the value n of a global name.

Definition 3.2. Let

evaluate(b,e)

be a function, that substitutes the actual basis value b for the expression e. (The result may be an integer or an expression which contains a global name as parameter.)

Theorem 3.1. The process specified by assumption 2.1./a and 2.1./b is realized by the following algorithm:

```
process-stmt(t) =
    process-data(s-data(t),a);
        a:process-address(s-address(t))
process-address(t) =
    is-abs(s-type(t)) → PASS:s-value(t)
    T → PASS:evaluate(s-basis(ξ ),s-value(t))
process-data(t,a)
    is-abs(s-type(t))→ translate(a,s-value(t))
    is-b-rel(s-type(t)) →
            translate(a,evaluate(s-basis( ξ ),s-value(t)))
    is-ext-rel(s-type(t)) →
            process-exp(a,s-basis.s-value(t),s-exp.s-value(t))
    T→ process-exp(a,s-basis.s-value(t),
                         evaluate(s-basis( ξ ),s-exp.s-value(t))
translate(a,v) =
    s-output:s-output( ξ ) ⌢ μ₀ (<s-address:a>,
                                   <s-value:v>)
```

```
process-exp(a,n,e) =
    is-integer(n.s-table($\xi$)) $\rightarrow$
          translate(a,calculate(n.s-table($\xi$),e)
    T $\rightarrow$ set-rest(c,d);
              c:pass-address(a),
              d:pass-data(n,e)

set-rest(c,d) =
    s-rest:s-rest($\xi$)$\frown$<$\mu_0$(<s-address:c>,<s-data:d>)>

pass-address(a) =
    PASS:$\mu_0$(<s-type:abs>,<s-value:a>)
                                  /abs $\in$ is-$\hat{abs}$/

pass-data(n,e) =
    PASS: $\mu_0$ (<s-type:ext-rel>,
             <s-value:$\mu_0$(<s-basis:n>,<s-expression:e>)>)
                                /ext-rel $\in$ is-$\widehat{ext}$-rel/
```

<u>Proof.</u> Let is-stmt(t) = T. By definition 3.1 the macro

<u>process-address</u>(s-address(t))

passes the "a" absolute address of the value to be loaded to the macro

<u>process-data</u>(s-data(t),a).

If one of the followings holds:

- an absolute value is to be loaded,
- an basis relative value is to be loaded,
- a predefinite name relative value is to be loaded,
- a relative value calculated by using a predefinite name and the actual basis is to be loaded,

the statement <u>translate</u> is activated by <u>process-data</u>. Obviously the actual parameters of the statement <u>translate</u> have appropriate values.

If the data expression has a postdefinite name as a parameter, the statement <u>set-rest</u> is executed. It is easy to see that appropriate parameters are passed to it by <u>pass-address</u> and <u>pass-data</u>. Similarly the correctness of the statements <u>translate</u> and <u>set-rest</u> also obvious.

3. ELEMENTARY OBJECTS AND PRIMITIVE FUNCTIONS

The abstract model contains several objects which are not defined further. These are regarded as elementary objects. The elementary objects of our abstract model meet the following predicates:

is-segment-name
is-global-name
is-integer
is-expression
is-undefined
is-abs
is-b-rel
is-ext-rel
is-b-ext

The abstract model does contains some undefined functions too. They are called primitive functions of the abstract model. These are

postdef(x)
segment-name(x)
calculate(x,y)
evaluate(x,y)

The specification of these elementary objects and functions stands outside the scope of an abstraction. They are to be specified when a concrete relocatable binary form is deduced from this abstract model.

The primitive functions "calculate" and "evaluate" can not be specified without the specification of the elementary objects, while the functions "postdef" and "segment name" can be realized by a search algorithm. However, a search algorithm is closely connected to the concrete realization of the program form and therefore they are not discussed here.

4. REFERENCES

[1] Lee, A.N.: Computer semantics.
Van Nostrand Reinhold Co. (1972)

[2] Neuhold, E.J.: The formal description of programming languages.
IBM System Journal <u>10.</u> (1971)

[3] Wegner, P.: The Vienna definition language.
Computing Surveys <u>4.</u> (1972)

[4] Dömölki, B.: Simple abstract assembler model.
Mathematical foundations of computer sciences.
Proceedings of symposium and summer school, Strbske Pleso, 1973.

[5] Presser, L., White, J.R.: Linkers and loaders.
Computing Surveys <u>4.</u> (1972)

62-292

Kiadja a Központi Fizikai Kutató Intézet
Felelős kiadó: Sándory Mihály igazgató
Szakmai lektor: Lőcs Gyula
Nyelvi lektor : Lőcs Gyula
Példányszám: 310 Törzsszám: 76-110
Készült a KFKI sokszorosító üzemében
Budapest, 1975. február hó