Sane typing for COBOL programs

Emilien Lemaire
emilien.lemaire@ocamlpro.com

June 17, 2022

Contents

List of Tables

List of Figures

1 Introduction

The goal of this document is to present typing rules for COBOL. There will be two sets of rules presented in the document, one following the 1985 ANSI standard COBOL, another following stricter rules that we will introduce. This decision has been taken as the COBOL standard does not set any typing rules per se, and we want to give a guideline for any new COBOL compiler that would follow more recent typing standards.

As this document does not aim to teach the syntax of COBOL, it will be assumed all along the document that the source code of the program is parsed and that the syntax rules of COBOL are respected and verified before any presented typing or analysis.

2 Data Structures in COBOL

In COBOL we can differentiate three types of data structures for user defined data area:

- **Elementary** which represents a data item that cannot be subdivided in smaller items;
- **Group** which represents a data item that can be subdivided into one or more **Group** or **Elementary** items;
- **Table** which represents an array of items that have the same structure.

2.1 Elementary

An elementary item has a data class and a data category, which are defined in the standard. Using these categories we can create types for each elementary item. The category and class of the data item is derived from either a PICTURE clause or a USAGE clause. The USAGE clause can be defined in the data description entry of the elementary item or of any group the elementary item is in.

2.2 Group

A group can have subgroups and sub elementary items. A group ends when another item of the same level number is defined in the division. From these subgroups and sub elementary items, we can give each group a type, that would be comparable to a record type.

2.3 Tables

A table is an array of tables, groups or elementary items, which can have a fixed or a variable size. A variable sized table is delimited by a minimum and a maximum size depending on the value of a data item that must be declared in the program. We can type the table with a type comparable to an array or a vector (but the data is not dynamically allocated).

3 Category and classes

The category and classes of data in COBOL are the following ones:

Class	**Category**
Alphabetic	Alphabetic
Numeric	Numeric
Alphanumeric	Alphanumeric
	Alphanumeric-edited
	Numeric-edited

Table 1: Category per class for COBOL

The standard gives a class and a category to each elementary item depending on its PICTURE clause or USAGE clause. The class given to a group is Alphanumeric and the category depends on the items forming the group.

3.1 Category and class compatibility

As group are considered Alphanumeric of alphanumeric class, there must be a compatibility between certain classes and categories. One of the most important is that an Alphanumeric item can receive data of any class, thus it is compatible with every data class. An alphabetic item can receive data from an alphanumeric item if and only if this data item contains letters ([a-zA-Z]) but it cannot receive any data from a numeric item. A numeric item can receive data from an alphanumeric item if and only if this item contains only numbers, it cannot receive any data from an alphabetic item. The following table summarizes the data class compatibility:

Table 2: Summary of class compatibility

Sending item class	Receiving item class	Restrictions
Alphanumeric	Alphanumeric	None
	Alphabetic	The sending item can only contain letters
	Numeric	The sending item can only contain numbers
Alphabetic	Alphabetic	None
	Alphanumeric	None
Numeric	Numeric	None
	Alphanumeric	None

Some COBOL compiler do not enforce this basic rules and allow any data to be put in any item without any error at compile time or during execution. We would suggest giving at least a compiler warning when the compiler is able to know the content of the sending item at compile time and the receiving item is of an incompatible class.

4 Types

We will here give the types that are forming our typing system for COBOL. As said in the Introduction 1, these types will be part of our stricter typing system.

4.1 Strict typing system

For the definition of every type, please see Figure 1.

```
type cob_type =
    | Elementary of cob_class
    | Group of (string * cob_type) list
    | Table of cob_type * cob_table_length

type cob_class =
    | Numeric of numeric
    | Alphabetic of int
    | Alphanumeric of alphanumeric_category
    | Conditional

type cob_table_length =
    | Fixed of int
    | OccursDepending of occurs_depending

type occurs_depending = {
    min_length: int;
    max_length: int;
    depending_on: string;
}

type numeric = {
    signed: bool;
    integer_length: int;
    decimal_length: int;
}

type alphanumeric_category =
    | Alphanumeric of (character_kind * int) list
    | AlphanumericEdited of (alphanumeric_edited_character_kind * int) list
    | NumericEdited of (numeric_edited_character_kind * int) list

type character_kind =
    | AlphabeticChar
    | AlphanumericChar
    | NumericChar

type alphanumeric_edited_character_kind =
    | A | X | Nine | B | Zero | Slant

type numeric_edited_chatacter_kind =
    | B      | Slant | P     | V     | Zero | Nine | Comma
    | Period | Star  | Plus  | Minus | CR   | DB   | Currency
```

Figure 1: COBOL data types

4.1.1 `cob_type`

`cob_type` is divided into three base types:

- `Elementary` which represents an elementary item and has a class
- `Group` which represents a group item and has type associated with each sub item that is contained in the group, this could be compared to field of classic records type.
- `Table` which represents a table item and is defined by the type of the data in the table and its length.

As said above, the group items are given a type that differs from the standard, this is to enable better compile time error generation and avoid runtime errors because of the typing of a group item.

4.1.2 Numeric type

To define a numeric type, we must figure out the length of its integer and decimal zone, this enables us to generate compile time warning when an integer item of different integer or decimal size is moved to the numeric item (especially if the item has either a bigger integer or decimal length.)

4.1.3 Alphabetic type

As alphabetic items are the simplest (they only have one kind of data, are not signed, or can be edited) we only need to save the length of the item to type it.

4.1.4 Alphanumeric type

Alphanumeric type is the most complex simple type that we have, because each character in the item can be either numeric, alphabetic or alphanumeric. If we follow the standard, it does not matter what each character is defined as when declaring the item, an alphanumeric item can accept alphanumeric characters for every one of those declared character. But if the user declares an alphanumeric item with different kind of characters we can consider that they wish the alphanumeric item contains the right kind of characters at any given time of the execution of the program, for this reason in our typing system we generate a type that remember the type of each character for the declared data item.

4.1.5 Edited types

Like alphanumeric items edited items have specific kind of characters that are at specific places of the item. For this reason we defined the type as a list of character kinds, so we can trigger warning or errors when the sending item does not match the template of the receiving edited item.

4.2 Standard typing system

Every type is defined in Figure 2.

4.2.1 Alphabetic type

A data item of this type can only contain letters and its picture can only contain one character, so we only need to know about the length of the item to type it.

4.2.2 Numeric type

A data item of this type can only contain numbers, a sign and a decimal point. As the decimal point is not present in the physical data item, we need to know where it is and so we save this information in the type of the item.

4.2.3 Alphanumeric type

A data item of type alphanumeric can contain any character of the computer character set. It does not matter what character where defined in the picture string, if the data item is alphanumeric then all of its character can accept any alphanumeric character. As such we only need to know the size of the item to type it.

```
type cob_type =
  | Alphabetic of int
  | Numeric of numeric
  | Alphanumeric of alphanumeric_category
  | Table of cob_type * cob_table_length
  | Conditional

type cob_table_length =
  | Fixed of int
  | OccursDepending of occurs_depending

type occurs_depending = {
  min_size: int;
  max_size: int;
  depending_on: string;
}

type alphanumeric_category =
  | Alphanumeric of int
  | AlphanumericEdited of int
  | NumericEdited of numeric
  | Group of cob_type list

type numeric = {
  signed: bool;
  integer_length: int;
  decimal_length: int;
}
```

Figure 2: Standard types for COBOL

4.2.4 Edited types

The edited types are here to inform the environment to look for edition rules when moving data from and to an edited typed item. We do not keep any information about the edited characters in the type of the item.

4.2.5 Table type

A data item of this type can contain an array of item of the same type. To be able to type as well as possible the program, we save the length of the table as part of the type.

5 Typing with the strict type system

5.1 Typing data declared with a data description entries

5.1.1 Elementary items

An elementary item will be typed in accordance to its `PICTURE` clause or `USAGE` clause. The typing of the item is as follows:

5.1.1.1 Typing by picture clause

We assume here that the picture string given respects the conditions of a well-formed picture string from the standard. We also consider that the `SPECIAL-NAMES` paragraph has been analyzed to handle the `CURRENCY-SIGN` phrase and the `DECIMAL POINT IS COMMA` phrase. The following algorithm is used to deduce the type of an elementary item in accordance to its picture string. The picture string that is given as an entry of the algorithm is supposed to be of type `(picture_char * int) list`, where `picture_char` is an algebraic type that contains a constructor for every possible character in a picture string and the integer in the second part of the pair is the number of consecutive characters of the same type in the picture string.

We must first define types and functions that will help us get the type of the elementary item in Figure 5.

We assume that the functions:

- `alphanumeric_edited_chars_of_alphabetic_length`,
- `alphanumeric_edited_chars_of_alphanumeric_chars`,
- `alphanumeric_edited_chars_of_ambiguous_chars`,
- `alphanumeric_edited_chars_of_numeric_chars`,
- `numeric_edited_chars_of_numeric_chars`,
- `numeric_edited_chars_of_ambiguous_chars`,
- `ambiguous_chars_of_numeric_chars`,

are defined and fail in the same way as `alphanumeric_chars_of_numeric_chars` when an unexpected character is met.

The functions `<cob_class>_<char>_of_int` are defined for every character of every class from the `cob_class'` type (e.g. the function `numeric_nine_of_int` will transform the integer 4 into the `numeric_char list`: `[Nine; Nine; Nine; Nine]`)

The algorithm presented in Figure 6 gives us the COBOL class of the elementary item. Then from this information, we can deduce the type of the item.

5.1.1.2 Typing with USAGE clause

From the standard, there must not be any `PICTURE` clause when the `USAGE IS INDEX` phrase is used. This means that we must deduce the type from the `USAGE` clause itself.

As the clause describes it `USAGE IS INDEX` means that the elementary item will be used as an index for table subscripting. This means a some information can be extracted from that usage:

- The item must be an integer;
- and it must contain a value of 1 or more.

From this we can deduce that we can give it a numeric type. This numeric type will actually depend on the platform you are compiling your program on (or at least for). Usually it will be a numeric type of size 4 or 8 bytes. So the final type of the item with usage `INDEX` will be something close to:

```
let elem_type = Elementary (
  Numeric {
    signed = false;
    integer_length = 4 (*or 8*);
    decimal_length = 0;
  }
)
```

We could also implement another type that is `Index` so that we don't have to make it platform specific.

5.1.1.3 Typing without `USAGE` clause or `PICTURE` clause

There might be a few time when an elementary item does not have a `USAGE` clause or a `PICTURE` clause. This means that the item is in a group where a higher level item has a `USAGE IS INDEX`. In this case every elementary item of the groups under the item that is defined as `USAGE IS INDEX` is an index.

5.1.2 Group items

A group item is typed recursively with every element inside the group (may it be a table, a group or an elementary item).

Each new group or elementary item directly under the considered group adds an element to the list of the group type.

We consider that the code has been parsed in such a way that the data description entries are in the form of:

```
type data_description_entries =
  | DDEGroup of group_description_entry
  | DDEElementary of data_description_entry (* contains all the information from the data
                                               description entry in the COBOL program *)
type group_description_entry = {
    sub_items: data_description_entries list;
    group_description: data_description_entry
  }
```

Then we can type the group with the following algorithm:

```
let rec type_of_data_description_entries dde =
  match dde with
  | DDEGroup {sub_items; group_description} ->
      Group (List.map (fun sub_item ->
        let name = (get_name_of_data_description_entry sub_item) in
        let typ = (type_of_data_description_entries sub_item) in
        name, typ
      )) sub_items
  | DDEElementary data_description_entry ->
      type_of_data_description_entry data_description_entry
```

Figure 3: Strict typing of group items

In this algorithm, `type_of_data_description_entry` takes a parsed data description entry and uses the algorithm defined in 5.1.1 to get an elementary type.

5.1.3 Table items

If an `OCCURS` clause is present in the data description entry, then it is a table. Except for the `OCCURS` clause, all other clauses in the data description entry of the table is applied to every subordinate items. For multidimensional tables, the other `OCCURS` must be in a subordinate data description entry.

Hence, when a table is typed the type inside it is either `Elementary _` or `Group _` and never `Table _`.

To type a table we can adapt the algorithm from Figure 3 to look for occurs clauses:

```
let rec type_of_data_description_entries dde =
  match dde with
  | DDEGroup {sub_items; group_description} ->
      let occurs_opt = get_occurs_opt_from_data_description_entry group_description in
      let data_type = Group (List.map (fun sub_item ->
        let name = (get_name_of_data_description_entry sub_item) in
        let typ = (type_of_data_description_entries sub_item) in
        name, typ
      )) sub_items in
      (match occurs_opt with
       | Some occurs ->
        let table_size = get_table_length_from_occurs occurs in
        Table (data_type, tab_size)
       | None -> data_type)
  | DDEElementary data_description_entry ->
      let data_type = type_of_data_description_entry data_description_entry in
      (match occurs_opt with
       | Some occurs ->
        let table_size = get_table_length_from_occurs occurs in
        Table (data_type, tab_size)
       | None -> data_type)
```

Figure 4: Strict typing of group items

5.1.4 RENAMES clause

When typing an item that is declared with a RENAMES clause, you must first type the item or items that are renamed. The renamed items cannot be or contain a table item. Then we have three cases:

- The RENAMES clause does not provide a THROUGH phrase and the renamed item is Elementary _, then the item the clause declares is of the same Elementary _ type.
- The RENAMES clause does not provide a THROUGH phrase and the renamed item is Group _, then the item the clause declares is of the same Group _ type.
- The RENAMES clause provides a THROUGH phrase, then every element in the range of the renamed items is an element of the Group _ type of the defined item.

5.1.5 REDEFINES clause

Typing a REDEFINES clause is the same as typing a basic data description entry, with the exception that it cannot be or contain any Table _ item. If the REDEFINES clause is inside a Group _ then this group must contain a field with the data name of the clause. This field may be flagged as a redefines but not necessarily as this information can be stored somewhere else in the program environment.

5.1.6 Conditions

When an item is declared at level 88, it is a condition. The item holds the value TRUE if the item declared just above it holds a value in the range of the declared value of the VALUE clause of the declared item. But in COBOL there are no boolean class or category, so the typing of those items can be a bit tricky. For this reason we added the Conditional class to our typing system. Hence, any item declared at level 88 will be a Conditional. The value that it holds and the data item that it is linked to can be saved somewhere else in the program environment.

5.1.7 Example

Let's suppose that you have the following program:

```
WORKING-STORAGE SECTION.
01 DEPENDING-1 PIC 9999.
01 TABLE-1 OCCURS 10 TIMES.
    02 ITEM-1 OCCURS 5 TIMES PIC S9(5)V9.
01 ITEM-1.
  05 ITEM-1-1.
    10 ITEM-1-1-1  PIC X(9).
  05 ITEM-1-2 PIC 99/99/9999.
01 TABLE-2 OCCURS 1 TO 10 TIMES
     DEPENDING ON DEPENDING-1
     PIC AAAA.
```

In your environment the type table would be:

```
["DEPENDING-1", Elementary (Numeric {
  is_signed = false;
  integer_length = 4;
  decimal_length = 0;
 });
 "TABLE-1", Table (Group [
  "ITEM-1", Table (Elementary (Numeric {
    is_signed = true;
    integer_length = 5;
    decimal_length = 1;
  }), Fixed 5)
 ], Fixed 5);
 "ITEM-1", Group ([
  "ITEM-1-1", Group ([
    "ITEM-1-1-1", Elementary (Alphanumeric [AlphanumericChar, 9])
  ]);
  "ITEM-1-2", Elementary (NumericEdited [(Nine, 2);
                                         (Slant, 1);
                                         (Nine, 2);
                                         (Slant, 1);
                                         (Nine, 4)])
 ]);
 "TABLE-2", Table (
  Elementary (Alphabetic 4),
  OccursDepending {
    min_length = 1;
    max_length = 10;
    depending_on = "DEPENDING-1";
  }
 )
]
```

6 Typing with the standard type system

6.1 Typing data declared with a data description entry

6.1.1 Elementary item

As for the strict typing system, to type an elementary item with the standard type system we have to adapt on how the data is declared. It may be typed using its PICTURE clause, USAGE clause or the USAGE clause of the group it is in.

6.1.1.1 Typing with the PICTURE clause

These rules are derived from the standard. An elementary item is:

- Alphabetic if its PICTURE string contains only A.
- Numeric if its PICTURE string contains only Nine, P, S and V.
- Alphanumeric Alphanumeric if its PICTURE string contains only X, A and Nine. The PICTURE string must contain at least one X or both a A and a Nine.
- Alphanumeric AlphanumericEdited if its PICTURE string contains only A, X, Nine, B, Zero and Slant. The PICTURE string must contain at least one A or X and one B or Zero or Slant
- Alphanumeric NumericEdited if its PICTURE string contains only B, Slant, P, V, Z, Zero, Nine, Comma, Period, Star, Plus, Minus, CR, DB and Currency.

The value of the numeric record when it is part of the type is determined by the presence or absence of S for the signed field and the presence or absence and position of the V or P for the integer_length and decimal_length fields.

6.1.1.2 Typing with USAGE clause

As for the strict type system, when the USAGE IS INDEX clause is present in the data description entry, the type of the item is Numeric and the content of the numeric record will depend on the platform the program is compiled for. It will still have to be an unsigned integer only data item.

6.1.1.3 Typing without USAGE clause or PICTURE clause

If an elementary item does not have a USAGE clause and does not have a PICTURE clause then we must type it using the USAGE clauseof its group, if there are no USAGE clause in the group of the item, then it is an error.

6.1.2 Group items

Using the standard type system, every group is an Alphanumeric item. As we want to stay as close as we can to the standard in this type system, we will consider them also as Alphanumeric. But as we also want to be able to trigger warnings (if not errors) we still keep information about the content of each group so we have the type Alphanumeric Group, and this type contains information about the content of the group as each sub item of the group is typed and saved in the Group constructor.

6.1.3 Table items

To type a table we must look for the OCCURS clause in the data description entry of the item. Then depending on the presence of the FROM TO phrase in the OCCURS clause, we can deduce the type of the table length and we must look for sub-items and type them to know if the table contains a group or an elementary item. If the data description entry of the table also contains a PICTURE clause then it is a table of elementary items, otherwise it is a table of a group item.

6.1.4 RENAMES clause

When typing a RENAMES clause, there are two different scenarios:

- The RENAMES clause does not declare a THROUGH phrase, then the type of the newly defined item is the same as the one of the renamed item;
- The RENAMES clause declares a THROUGH phrase, then the type of the newly defined item is Alphanumeric (Group _) with each item included in the renamed items giving an element to the type list of the Group constructor.

If the RENAMES clause is defined inside an Alphanumeric (Group _) item, then its type is added to the type list of the Group constructor, and you can save the information somewhere else in the program environment that this element of the list comes from a RENAMES clause.

6.1.5 REDEFINES clause

When typing a redefines clause you can type it as a basic data description entry. If this REDEFINES clause is inside a group then its type will be added to the Group constructor type list with the information that this element of the list comes from a REDEFINES clause stored somewhere else in your program environment.

6.1.6 Conditions

Data items declared at level 88 are conditions. The value of this item is TRUE if the data item declared just above it holds a value in the range of the declared VALUE clause of the condition. As in COBOL there are not real boolean class, typing this kind of data can be tricky. For this reason we added a Condition type in our standard typing system. Even though this is getting away from the standard, we must use such a decision to be able to type our program as well as possible. So every item declared at level 88 will have type Conditional and the data item it is linked to and the value this item needs to hold for the condition to be true can be saved somewhere else in the program environment.

6.1.7 Example

Let's suppose that you have the following program:

```
WORKING-STORAGE SECTION.
01 DEPENDING-1 PIC 9999.
01 TABLE-1 OCCURS 10 TIMES.
    02 ITEM-1 OCCURS 5 TIMES PIC S9(5)V9.
01 ITEM-1.
  05 ITEM-1-1.
    10 ITEM-1-1-1  PIC X(9).
  05 ITEM-1-2 PIC 99/99/9999.
01 TABLE-2 OCCURS 1 TO 10 TIMES
     DEPENDING ON DEPENDING-1
     PIC AAAA.
```

Then the type environment will be the following (with some basic mangling to avoid name collision):

```
["DEPENDING-1", Numeric {
  signed = false;
  integer_length = 4;
  decimal_length = 0;
 };
 "TABLE-1", Table (
  Table (
    Numeric {
      signed = true;
      integer_length = 5;
      decimal_length = 1;
    },
    Fixed 5
  ),
  Fixed 10
 );
 "ITEM-1-of-TABLE-1", Table (
    Numeric {
      signed = true;
      integer_length = 5;
      decimal_length = 1;
    },
    Fixed 5
 );
 "ITEM-1", Alphanumeric (Group (
  [Alphanumeric (Group
    [Alphanumeric (Alphanumeric 9)]
   );
   Alphanumeric (NumericEdited {
    signed = false;
    integer_length = 8;
    decimal_length = 0;
   })
  ]
 ));
 "ITEM-1-1-of-ITEM-1", Alphanumeric (Group
  [Alphanumeric (Alphanumeric 9)]
 );
 "ITEM-1-1-1-of-ITEM-1-1-of-ITEM-1", Alphanumeric (Alphanumeric 9);
 "ITEM-1-2-of-ITEM-1", Alphanumeric (NumericEdited {
  signed = false;
  integer_length = 8;
  decimal_length = 0;
 });
 "TABLE-2", Table (
   Alphabetic 4,
   OccursDepending {
    depending_on = "DEPENDING-1";
    min_size = 1;
    max_size = 10;
   }
 )
 ]
```

A Algorithms

A.1 Typing elementary item

```
type cob_category' =
 | Alphabetic of int
 | Alphanumeric of alphnumeric_char list
 | Numeric of numeric_char list
 | NumericEdited of numeric_edited_char list
 | AlphanumericEdited of alphanumeric_edited_char list
 | AmbiguousEdited of numeric_or_alphanumeric_edited_char list

type alphanumeric_char =
 | A
 | X
 | Nine

type numeric_char =
 | Nine
 | P
 | S
 | V

type numeric_edited_char =
 | B | Slant | P | V | Z | Zero | Nine | Comma | Period | Star | Plus | Minus | DB | CR
 | Currency

type alphanumeric_edited_char =
 | A | X | Nine | B | Zero | Slant

type numeric_or_alphanumeric_edited_char =
 | Nine | B | Zero | Slant

let alphanumeric_chars_of_alphabetic_length alpha_length =
  let aux acc alpha_length =
    match alpha_lenght with
    | 0 -> acc
    | l -> aux (A::acc) (l - 1)
  in
  aux [] alpha_length

let alphanumeric_chars_of_numeric_chars num_char_l =
  let aux acc char =
    match chars with
    | Nine::tl -> aux Nine::acc tl
    | P::tl -> failwith "P is not allowed in an alphanumeric picture string"
    | S::tl -> failwith "S is not allowed in an alphanumeric picture string"
    | V::tl -> failwith "V is not allowed in an alphanumeric picture string"
  in
  List.rev (aux [] num_char_l)
```

Figure 5: Helper types and functions for typing an elementary item

```
let cob_cat_of_picture picture =
  let rec aux curr_type pic_char_list =
    match pic_char_list with
     | [] -> curr_type
     | (pic_char, length)::tl ->
      let new_type = match pic_char with
      | A ->
        (match curr_type with
         | Alphabetic l -> Alphabetic (l + length)
         | Alphanumeric al -> Alphanumeric (al@(alphanumeric_a_of_int length))
         | AlphanumericEdited al -> AlphanumericEdited (al@
           (alphanumeric_edited_a_of_int length)
          )
         | AmbiguousEdited al -> AlphanumericEdited (
           (alphanumeric_edited_chars_of_ambiguous_chars al)@
             (alphanumeric_edited_a_of_int length)
          )
         | _ -> failwith "Invalid picture, unexpected character A"
        )
      | B ->
        (match curr_type with
         |Alphanumeric al -> AlphanumericEdited (
          (alphanumeric_edited_chars_of_alphanumeric_chars al)@
            (alphanumeric_edited_b_of_int length)
         )
         | AlphanumericEdited al -> AlphanumericEdited (al@
          (alphanumeric_edited_a_of_int length)
         )
         | Numeric nl -> (*If it only contains 9 then it is ambiguous*)
          (try
            AmbiguousEdited ((ambiguous_edited_chars_of_numeric_chars nl)@
              (ambiguous_edited_b_of_int length)
            )
           with Failure _ ->
            NumericEdited ((numeric_edited_chars_of_numeric_chars nl)@
              (numeric_edited_b_of_int length)
            ))
         | NumericEdited nl -> NumericEdited (nl@
          (numeric_edited_b_of_int length)
         )
         | AmbiguousEdited al -> AmbiguousEdited (al@
          (ambiguous_edited_b_of_int length)
         )
         | _ -> failwith "Invalid picture: Unexpected character B"
        )
```

(a) Algorithm for typing an elementary item with its picture (part 1)

```
| P ->
  (match curr_type with
   | AmbigiousEdited al -> NumericEdited (
    (numeric_edited_chars_of_ambiguous_chars al)@
      (numeric_edited_p_of_int length)
   )
   | NumericEdited nl -> NumericEdited (nl@
    (numeric_edited_p_of_int length)
   )
   | Numeric nl -> Numeric (nl@
    (numeric_p_of_int length)
   )
   | _ -> failwith "Invalid picture: Unexpected character P"
  )
| S ->
  (match curr_type with
   | Numeric nl -> Numeric (nl@(numeric_s_of_int length))
   | _ -> failwith "Invalide picture: Unexpected S character"
  )
| V ->
  (match curr_type with
   | Numeric nl -> Numeric (nl@numeric_v_of_int length)
   | NumericEdited nl -> NumericEdited (nl@
    (numeric_edited_v_of_int length)
   | AmbiguousEdited al -> NumericEdited (
      (numeric_edited_chars_of_ambiguous_chars al)@
        (numeric_edited_v_of_int length)
    )
   )
  )
| X ->
  (match curr_type with
   | Alphabetic l -> Alphanumeric (
    (alphanumeric_a_of_int l)@(alphanumeric_x_of_int length)
   )
   | Alphanumeric al -> Alphanumeric (al@(alphanumeric_x_of_int length))
   | Numeric nl -> Alphanumeric (
    (alphanumeric_chars_of_numeric_chars nl)@(alphanumeric_x_of_int length)
   )
   | AlphanumericEdited al -> AlphanumericEdited (al@
      (alphanumeric_edited_x_of_int length)
    )
   | AmbiguousEdited al -> AlphanumericEdited (
    (alphanumeric_edited_chars_of_ambiguous_chars al)@
      (alphanumeric_edited_x_of_int length)
   )
   | _ -> failwith "Invalid picture: Unexpected X character"
  )
```

(b) Algorithm for typing an elementary item with its picture (part 2)

```
| Zero ->
  (match curr_type with
   | Alphanumeric al -> AlphanumericEdited (
    (alphanumeric_edited_chars_of_alphanumeric_chars al)@
      (alphanumeric_edited_0_of_int length)
   )
   | Numeric nl -> (*If it only contains 9 then it is ambiguous*)
    (try
      AmbiguousEdited ((ambiguous_edited_chars_of_numeric_chars nl)@
        (ambiguous_edited_b_of_int length)
      )
     with Failure _ ->
      NumericEdited ((numeric_edited_chars_of_numeric_chars nl)@
        (numeric_edited_b_of_int length)
      ))
   | AmbiguousEdited al -> AmbiguousEdited (al@(ambiguous_edited_0_of_int length))
   | NumericEdited nl -> NumericEdited (nl@(numeric_edited_0_of_int length))
   | AlphanumericEdited al -> AlphanumericEdited (al@
    (alphanumeric_edited_0_of_int length)
   )
   | _ -> failwith "Invalid picture: Unexpected 0 character"
  )
| Slant ->
  (match curr_type with
   | Alphanumeric al -> AlphanumericEdited (
    (alphanumeric_edited_chars_of_alphanumeric_chars al)@
      (alphanumeric_edited_slant_of_int length)
   )
   | Numeric nl -> (*If it only contains 9 then it is ambiguous*)
    (try
      AmbiguousEdited ((ambiguous_edited_chars_of_numeric_chars nl)@
        (ambiguous_edited_b_of_int length)
      )
     with Failure _ ->
      NumericEdited ((numeric_edited_chars_of_numeric_chars nl)@
        (numeric_edited_b_of_int length)
      ))
   | AmbiguousEdited al -> AmbiguousEdited (al@(ambiguous_edited_slant_of_int length))
   | NumericEdited nl -> NumericEdited (nl@(numeric_edited_slant_of_int length))
   | AlphanumericEdited al -> AlphanumericEdited (al@
    (alphanumeric_edited_slant_of_int length)
   )
   | _ -> failwith "Invalid picture: Unexpected / character"
  )
```

(c) Algorithm for typing an elementary item with its picture (part 3)

```
| Z ->
  (match curr_type with
   | Numeric nl -> NumericEdited (
    (edited_chars_of_numeric_chars nl)@
      (numeric_edited_z_of_int length)
   )
   | NumericEdited nl -> NumericEdited (nl@(numeric_edited_z_of_int length))
   | AmbiguousEdited al -> NumericEdited (
    (numeric_edited_chars_of_ambiguous_chars al)@
      (numeric_edited_z_of_int length)
   )
   | _ -> failwith "Invalid picture: Unexpected Z character"
  )
| Nine ->
  (match curr_type with
   | Alphabetic l -> Alphanumeric (
    (alphanumeric_a_of_int l)@(alphanumeric_9_of_int length)
   )
   | Alphanumeric al -> Alphanumeric (al@alphanumeric_9_of_int length)
   | AlphanumericEdited al -> AlphanumericEdited (al@
    (alphanumeric_edited_9_of_int length)
   )
   | Numeric nl -> Numeric (nl@(numeric_edited_9_of_int length))
   | NumericEdited nl -> NumericEdited (nl@(numeric_edited_9_of_int length))
   | AmbiguousEdited al -> AmbiguousEdited (al@(numeric_edited_9_of_int length))
   | _ -> failwith "Invalid picture: Unexpected 9 character"
  )
| Comma ->
  (match curr_type with
   | Numeric nl -> NumericEdited (
    (edited_chars_of_numeric_chars nl)@
      (numeric_edited_comma_of_int length)
   )
   | NumericEdited nl -> NumericEdited (nl@(numeric_edited_comma_of_int length))
   | AmbiguousEdited al -> NumericEdited (
    (numeric_edited_chars_of_ambiguous_chars al)@
      (numeric_edited_comma_of_int length)
   )
   | _ -> failwith "Invalid picture: Unexpected , character"
  )
```

(d) Algorithm for typing an elementary item with its picture (part 4)

```
| Period ->
  (match curr_type with
   | Numeric nl -> NumericEdited (
    (edited_chars_of_numeric_chars nl)@
      (numeric_edited_period_of_int length)
   )
   | NumericEdited nl -> NumericEdited (nl@(numeric_edited_period_of_int length))
   | AmbiguousEdited al -> NumericEdited (
    (numeric_edited_chars_of_ambiguous_chars al)@
      (numeric_edited_period_of_int length)
   )
   | _ -> failwith "Invalid picture: Unexpected . character"
  )
| Star ->
  (match curr_type with
   | Numeric nl -> NumericEdited (
    (edited_chars_of_numeric_chars nl)@
      (numeric_edited_star_of_int length)
   )
   | NumericEdited nl -> NumericEdited (nl@(numeric_edited_star_of_int length))
   | AmbiguousEdited al -> NumericEdited (
    (numeric_edited_chars_of_ambiguous_chars al)@
      (numeric_edited_star_of_int length)
   )
   | _ -> failwith "Invalid picture: Unexpected * character"
  )
| Plus ->
  (match curr_type with
   | Numeric nl -> NumericEdited (
    (edited_chars_of_numeric_chars nl)@
      (numeric_edited_plus_of_int length)
   )
   | NumericEdited nl -> NumericEdited (nl@(numeric_edited_plus_of_int length))
   | AmbiguousEdited al -> NumericEdited (
    (numeric_edited_chars_of_ambiguous_chars al)@
      (numeric_edited_plus_of_int length)
   )
   | _ -> failwith "Invalid picture: Unexpected + character"
  )
```

(e) Algorithm for typing an elementary item with its picture (part 5)

```
| Minus ->
  (match curr_type with
   | Numeric nl -> NumericEdited (
    (edited_chars_of_numeric_chars nl)@
      (numeric_edited_minus_of_int length)
   )
   | NumericEdited nl -> NumericEdited (nl@(numeric_edited_minus_of_int length))
   | AmbiguousEdited al -> NumericEdited (
    (numeric_edited_chars_of_ambiguous_chars al)@
      (numeric_edited_minus_of_int length)
   )
   | _ -> failwith "Invalid picture: Unexpected - character"
  )
| DB ->
  (match curr_type with
   | Numeric nl -> NumericEdited (
    (edited_chars_of_numeric_chars nl)@
      (numeric_edited_db_of_int length)
   )
   | NumericEdited nl -> NumericEdited (nl@(numeric_edited_db_of_int length))
   | AmbiguousEdited al -> NumericEdited (
    (numeric_edited_chars_of_ambiguous_chars al)@
      (numeric_edited_db_of_int length)
   )
   | _ -> failwith "Invalid picture: Unexpected DB character"
  )
| CR ->
  (match curr_type with
   | Numeric nl -> NumericEdited (
    (edited_chars_of_numeric_chars nl)@
      (numeric_edited_cr_of_int length)
   )
   | NumericEdited nl -> NumericEdited (nl@(numeric_edited_cr_of_int length))
   | AmbiguousEdited al -> NumericEdited (
    (numeric_edited_chars_of_ambiguous_chars al)@
      (numeric_edited_cr_of_int length)
   )
   | _ -> failwith "Invalid picture: Unexpected CR character"
  )
```

(f) Algorithm for typing an elementary item with its picture (part 6)

```
    | Currency ->
      (match curr_type with
       | Numeric nl -> NumericEdited (
        (edited_chars_of_numeric_chars nl)@
          (numeric_edited_currency_of_int length))
       | NumericEdited nl -> NumericEdited (nl@(numeric_edited_currency_of_int length))
       | AmbiguousEdited al -> NumericEdited (
        (numeric_edited_chars_of_ambiguous_chars al)@
          (numeric_edited_currency_of_int length)
       ))
  in
  aux new_type tl
in
let first_type, picture =
  match picture with
  | [] -> failwith "Invalid picture: Unexpected empty picture"
  | (pic_char, length)::tl ->
  let first_type = match pic_char with
    | A -> Alphabetic length
    | B -> AmbiguousEdited (ambiguous_edited_b_of_int length)
    | P -> Numeric (numeric_p_of_int length)
    | S -> Numeric (numeric_s_of_int length)
    | V -> Numeric (numeric_v_of_int length)
    | X -> Alphanumeric (alphanumeric_x_of_int length)
    | Z -> NumericEdited (numeric_edited_z_of_int length)
    | Nine -> Numeric (numeric_9_of_int length)
    | Zero -> AmbiguousEdited (ambiguous_edited_zero_of_int length)
    | Slant -> AmbiguousEdited (ambiguous_edited_slant_of_int length)
    | Comma -> NumericEdited (numeric_edited_comma_of_int length)
    | Period -> NumericEdited (numeric_edited_period_of_int length)
    | Plus -> NumericEdited (numeric_edited_plus_of_int length)
    | Minus -> NumericEdited (numeric_edited_minus_of_int length)
    | CR -> NumericEdited (numeric_edited_cr_of_int length)
    | DB -> NumericEdited (numeric_edited_db_of_int length)
    | Star -> NumericEdited (numeric_edited_star_of_int length)
    | Currency -> NumericEdited (numeric_edited_currency_of_int length)
  in
  first_type, tl
in
let res = aux first_type picture in
(*If the type is ambiguous edited, then it is numeric because it does not meet the
  the threshold to be alphanumeric.
*)
match res with
| AmbigiousEdited al -> NumericEdited (numeric_edited_chars_of_ambiguous_chars al)
| _ -> res
```

(g) Algorithm for typing an elementary item with its picture (part 7)

Figure 6: Algorithm for typing an elementary item with its picture