P O C O — Compiler Generator

User Manual

Michael Eulenstein

Universität des Saarlandes

Technischer Bericht Nr. A2 / 85

Contents:

0. Overview.

This paper is to give a short overlook on the POCO - Compiler Generating System. It is not intended to be a full description or even documentation of the system but merely a manual for a possible user of the system. For more detailed information we refer to the literature given at the end of this report.

We start out with a short description of the system's overall structure and go into detail with some of the system's components. We will also describe the generation process and the general form of the generated components.

In the second chapter we will give an annotated description of the POCO compiler generator input language (GIL). A more easily readable form of the GIL syntax will be given in Appendix A in the form of syntax diagrams. We will also give a short example of a complete compiler specification.

As a preliminary operating manual we will conclude with a short survey of how to use POCO on a specific installation and describe the user available options which can be exploited to direct the generation process.

Appendix B will contain a summary of POCO generation time error messages; some messages will be explained more detailed.

1. An Introduction to the POCO - Compiler Generating System.

1.1. Compiler and Compiler Generation.

A **Compiler** is a system used to translate a source language specification of a given task into an equivalent machine language program which can be executed on a computer.

A compiler being itself a highly complex program - in the past it usually took several man years to implement it, a time and money consuming effort - the idea of automatic generation of compilers (or at least parts of a compiler) came up soon after the developement of the first compilers.

At this point it is interesting to note why compiler construction is such an important area in computer science: As there many existing programming languages (and becoming more and more every year) and many different real computers (the number of which is also increasing all the time) one does need a compiler for any of these languages and any of the real machines. This can be demonstrated in to following figure:

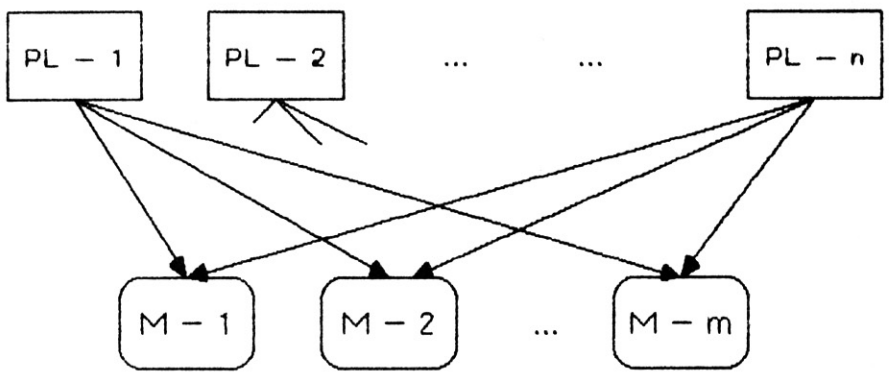


Figure 1.1.: You need n * m (different) compilers for
n given programming languages and
m given real computers

This problem is further complicated, if you consider that the compiler is a program itself and has to be installed on any of the real computers; sometimes the compiler program can be "ported" as whole, most of the time, however, this method is not possible.

Typically, a compiler can be divided into subtasks according to the scheme given below. The specific task of a compiler as noted in a square box is called a compiler **phase** (sometimes, without going deeper into theory, a compiler **pass**). Note that the different phases are logically independent and the input to every phase is the output of the preceeding or the source language program itself.

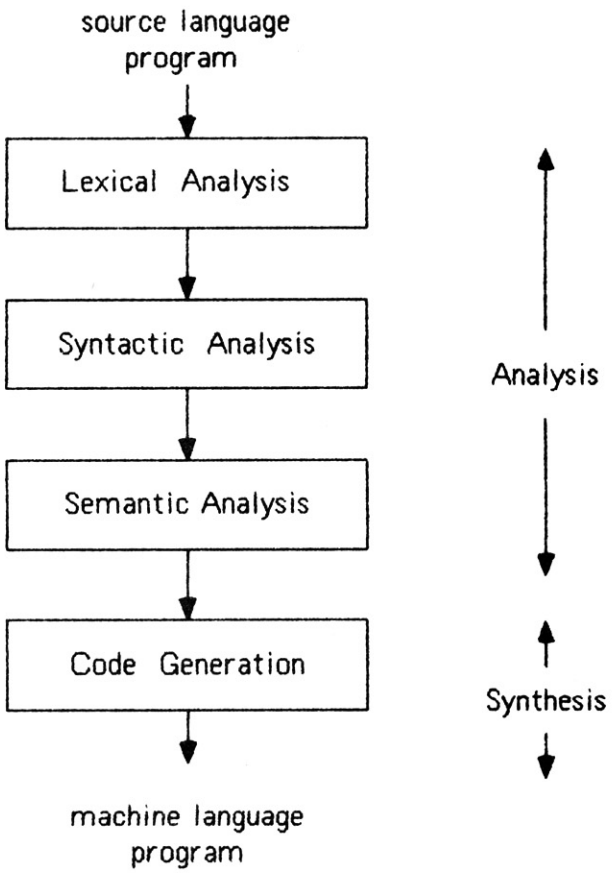


Figure 1.2.: General Structure of a Compiler.

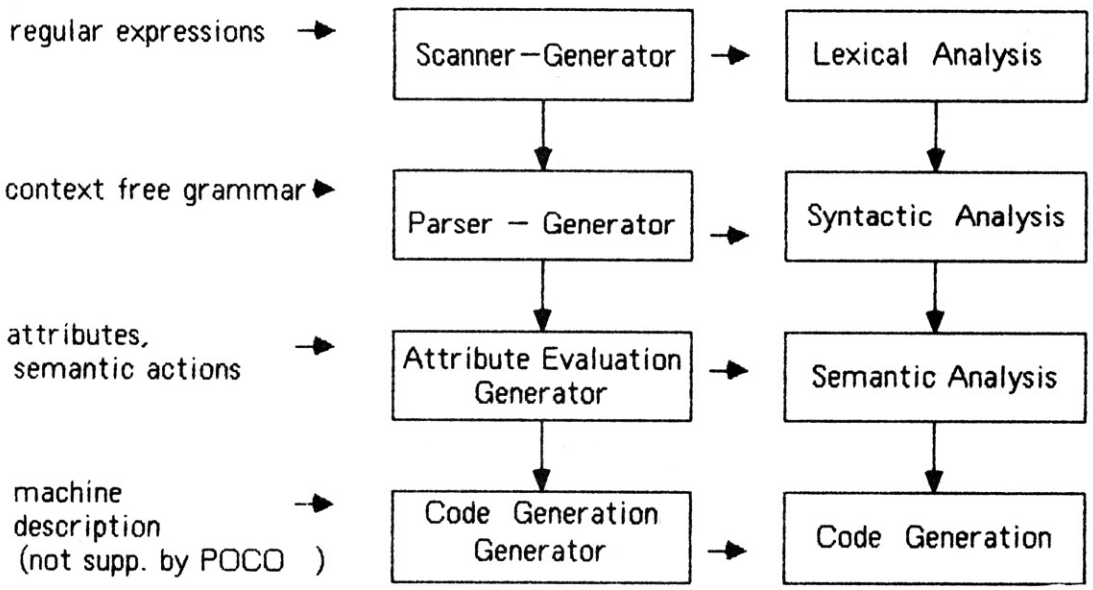
During Lexical Analysis (the equivalent part of the compiler is called the **Scanner**) the character stream of the source language input is translated into a stream of symbols (tokens). This token stream is used by the subsequent syntactic analysis phase (the **parser**) where the input is checked against the formal definition of the syntax of a given programming language.

In the semantic analysis phase, the meaning of the program - as far this can be deduced from the source - is computed and additional information gathered which is then exploited in the final **synthesizing** phases of a compiler in which the actual translation of the input program into the target machine language takes place.

A compiler generator now basicly operates on the following principles:

- For each compiler phase, it accepts a description of the corresponding part of a compiler for a given programming language
- Using this description, it will automacially generate the respective part of a compiler, i.e. the implementation of a program part which allows performing the respective compiler phase

This is shown in the following figure:



Picture 1.3.: Generalized Structure of a Compiler Generator.

1.2. The POCO - System Concept.

Let us now consider the basic concepts of the POCO - Compiler Generating System. The POCO system was developed at the University of Saarbruecken. It is conceptionnaly based on the MUG1 system developed at the Techical University Munich, one of the earliest and most successful compiler generating systems.

Above that the POCO developement is characterized by the following major design goals:

- POCO is intended to be a modern system in regard to

  a) the underlying theoretical concepts
  b) user orientedness
  c) safety of usage according to state of the art standards

- The System is designed so that porting the system itself as well as the generated compilers is feasible and not to complicated a task

- The generated compilers must be efficient

Above that the POCO is not merely a parser generator or a scanner generator but a complete system for generating all of the components of a compiler front end, including highly efficient tools to make work with the system easy and reliable. The **system** POCO is composed of the following elements:

- The POCO Compiler Generator

- A compiler for the programming language Pascal-m, a Pascal extension which allows for modularity (Pascal-m is both implementation language for the POCO system and the generated compilers)

- A machine independent linker to support the portability aspect of the POCO concept

- A semantic module data base mechanism for storing the user supplied semantic procedures/functions; his data base interacts with both the POCO compiler generator and the Pascal-m-Compiler/ Linker components

The POCO system is depictured in the following figure:

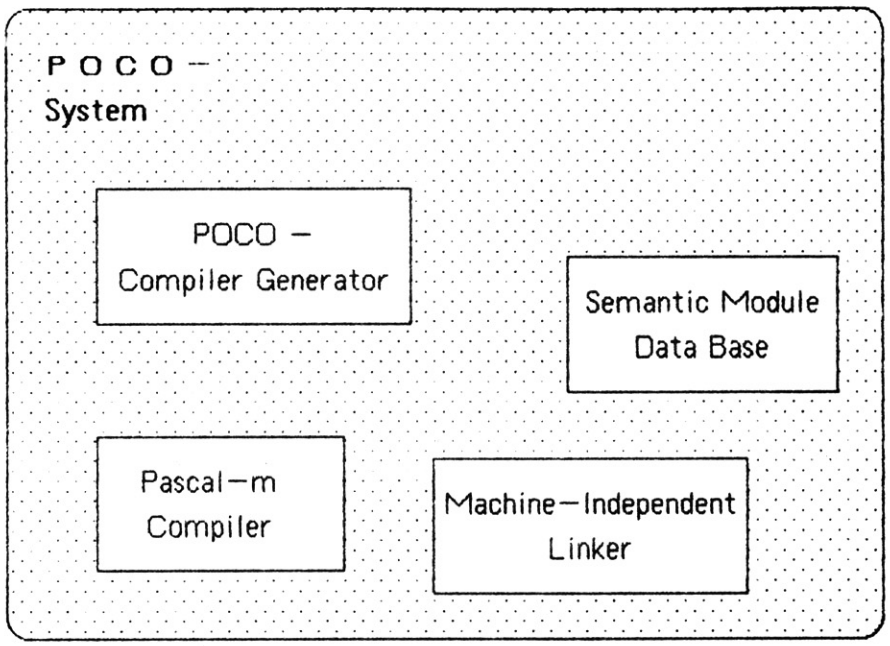


Figure 1.4.: POCO System Components

The POCO Compiler Generator is in itself divided into the following components:

- o a Grammar-Reader to process user input (i.e. the specification of a programming language for which a compiler is to be generated); this component is a small 'compiler' itself: it compiles the user input into an internally kept data structure.
- o an efficient Scanner - Generator
- o an LALR(1) - Parser - Generator
- o a Generator for Compile Time Attribute Evaluation

Each of the above mentioned generators will produce one of the specific parts of a real compiler; these parts are generated as Pascal-m modules. After complete generation these modules can be fed into the Pascal-m compiler and be translated into machine-independent code. The POCO generator components and the corroboration with Pascal-m compiler are shown in Figure 1.5.:

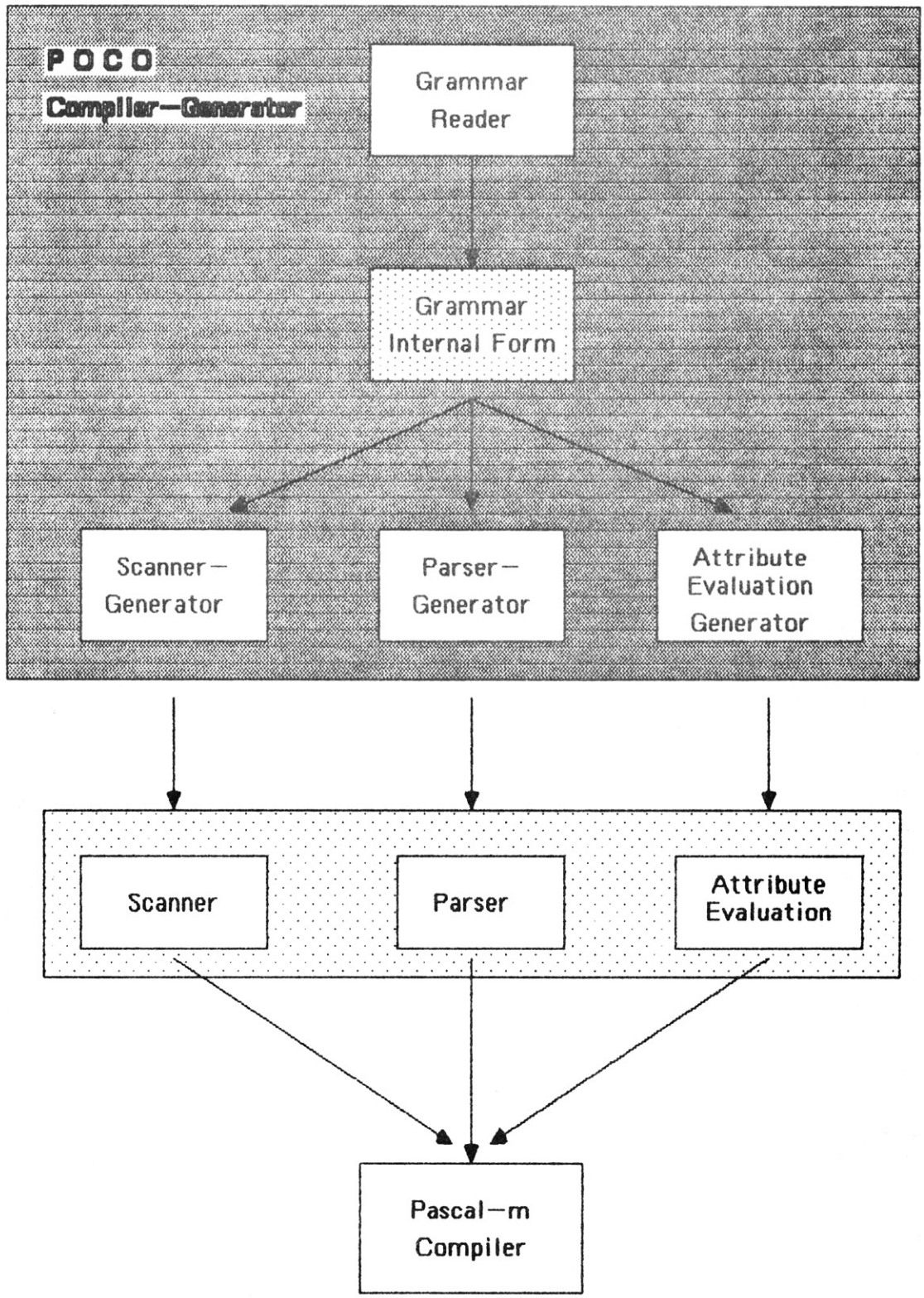


Figure 1.5.: POCO - Compiler Generator Components.

Internally, the generation process is subdivided in several tasks which are performed sequentially in the order given by Figure 1.6.; we call these different tasks "phases" in analogy to the phases of a generated compiler:

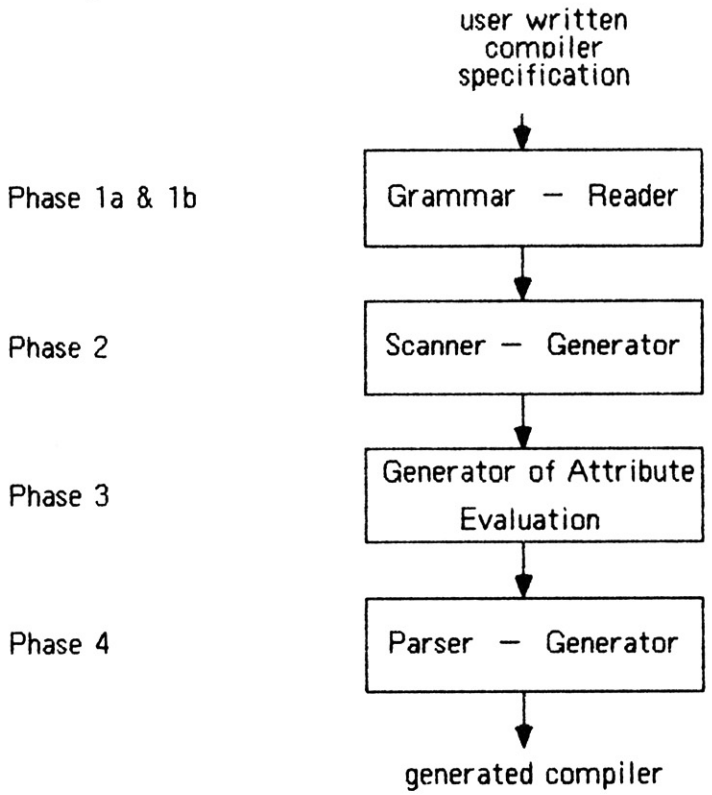


Figure 1.6.: Sequential Structure of the POCO Generation Process.

For complete specification of a compiler the user has to supply the generator with information about the semantic meaning of the programming language for which he wants a compiler to be generated. This is done by specifying so called semantic modules in the generator input in form of external Pascal-m modules. During generation of semantic attribute evaluation these modules are duly considered. They can be written in full offside the compiler specfication and entered into the semantic module data base as source code or after being processed by the Pascal-m compiler.

After complete generation of all compiler components and after supplying all user written semantic modules and translating them by use of the Pascal-m compiler all compiler components are linked together by the machine independent linker. After this process generation of a complete compiler is finished and the resulting compiler program can be installed on a specific real computer.

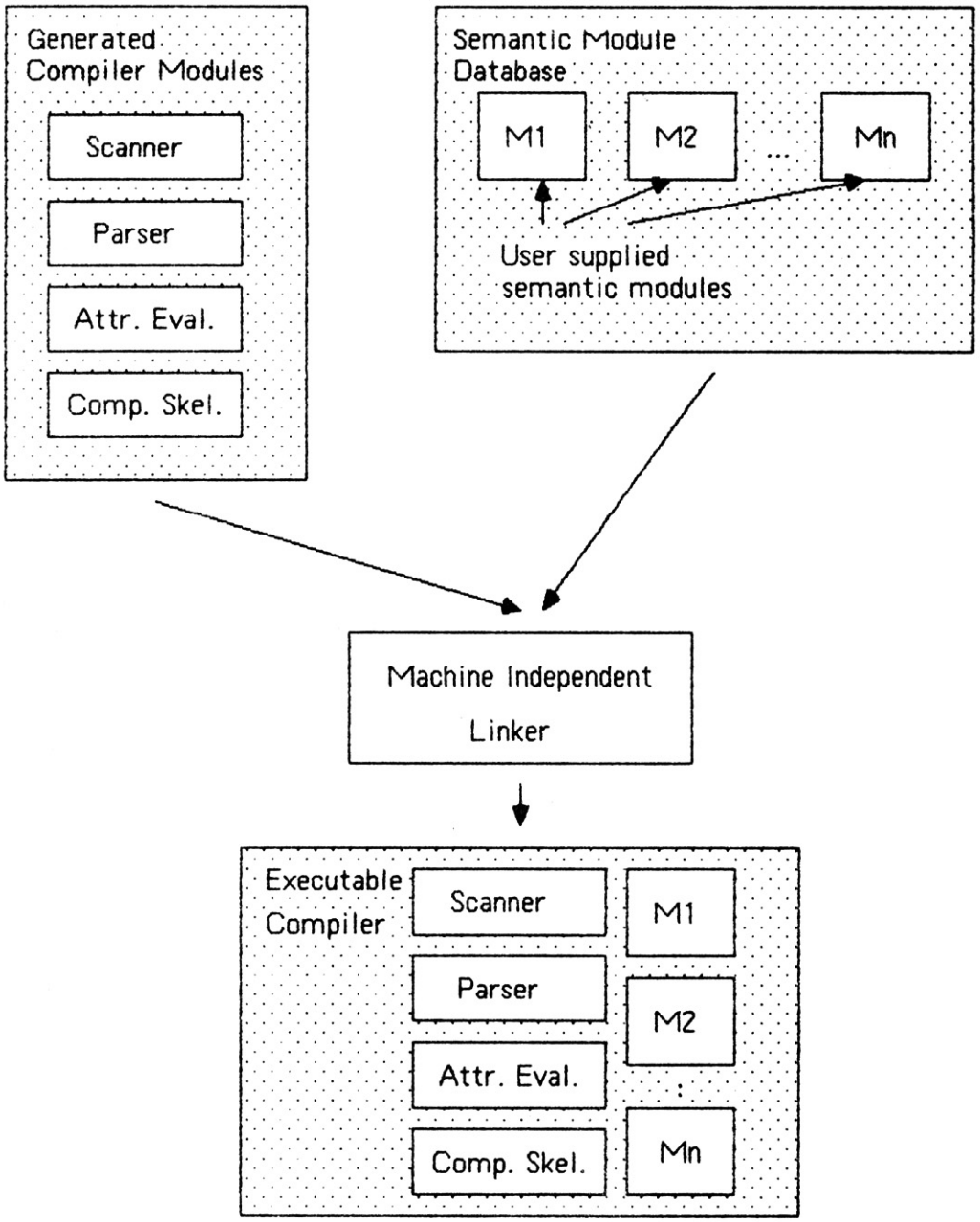


Figure 1.7.: Linking the generated compiler modules.

2. POCO Input Language Description.

An input to the POCO compiler generating system which may be expressed by means of the GIL contains all information which is needed in the process of generating the standard compiler components as well as the precise definition of the interface to user supplied semantic action procedures. This allows for extensive adequate control of the compiler description. Because of POCO's design the GIL is very close to the programming language PASCAL.

Notation:

The GIL syntax will be expressed in Backus Naur notation; the following symbols are the usual metasymbols and do not belong to the GIL:

::= | { }

"|" denotes alternatives, curly brackets allow for expressing repetition of elements in the following sense:

X ::= {Y} stands for X ::= | XY

nonterminals are inclosed in < ... >, terminal symbols are underlined.

2.1. Input Language Basic Vocabulary.

The basic vocabulary of the POCO input language consists of basic symbols classified into letters, digits and special symbols with:

<letter> ::= A | ... | Z | a | ... | z | _

<digit> ::= 0 | ... | 9

Special symbols are all special symbols of the programming language PASCAL, i.e.:

+ | - | * | / | (|) | [|] | = | <= | >= | < | > |

, | . | ; | : | ' | ^ |

if | **do** | **to** | **in** | **or** | **end** |

for | **div** | **var** | **mod** | **set** | **and** |

not | **else** | **with** | **then** | **goto** |

type | **case** | **then** | **file** | **begin** |

until | **while** | **array** | **const** | **label** |

repeat | **record** | **downto** | **packed** |

forward | **program** | **function** | **procedure**

as well as the additional special symbols of the PASCAL-m extension:

module | **interface** | **implementation** | **use**

and the following additional special symbols

language | **terminals** | **axiom** | **productions**

finis | **error** | **call** | **allbut**

and |

(which will be underlined in the syntax description to distinguish it from the meta symbol |).

A **comment** is defined as

{ any sequence of characters not containing "}" **}**

or

(* any sequence of characters not containing "*)" ***)**

It can appear between any two identifiers, numbers, reserved words or special symbols; comments may be nested by using the alternate lead in (closing) symbols.

2.2. POCO Input Lanugage Description.

1. Generator Input:

An input to the POCO compiler generating system is of the following form:

```
<compiler_descr>  ::=  <language_identification>
                       <terminal_definition_part>
                       <constant_definition_part>
                       <type_definition_part>
                       <module_declaration_part>
                       <axiom_definition>
                       <production_definition>
                        finis
```

2. Language Identification.

The identifier given in the language identification is interpreted as a name for the subsequently given grammar. This name will also be used as the program name of a generated compiler.

```
<language_identification> ::= language <identifier>
                            | <empty>
```

Note: If no language identifer is given the program name of the generated compiler will just be 'COMPILER'.

3. Terminal Definition Part:

In the terminal definition part of the POCO input all terminal symbols of the subsequently given grammar are declared and their lexical structure is defined.

The terminal declaration part is divided into the **Character Class**

Definition Part and the **Symbol Class Definition Part:**

```
<terminal_definition_part> ::= terminals
                               {<char_class_definition>}
                                <symbol_class_definition>
                               {<symbol_class_definition>}
```

In the **Character Class Definition Part** all basic characters may be grouped into character classes which can be used in the Symbol Class Definition Part in kind of a short hand fashion.

```
<character_class_definition> ::= <char_class_mode>
                                 <char_class_def>

<char_class_mode> ::= - | <empty>

<char_class_def>  ::= <char_class identifier> =
                      <char_class> {, <char_class> }


<char_class> ::= <character_constant> |
                 <character_constant> - <character_constant>
```

A character constant may be any character representable in a program. For the hidden characters 'carriage return' and 'line feed' there exist predefined special character classes which may be addressed by the predeclared names 'CR' and 'LF' which can be used on the right hand side of a character class definition. Note that the single quote character must be given as '''' in the usual Pascal way.

The **Character Class Mode** serves to express whether a character class is to be interpreted as an **ignorable** class (marked with '-'). An ignorable character class can be seen as a set of characters that is used by a generated scanner to separate one symbol from another but which will not be a part of of recognized symbol.

A terminal **Symbol Class** is defined by giving a (unique) symbol class identification number which is called the symbol's **Class Code,** the

symbol class Mode and a regular expression which describes the lexical structure of the symbol. Each definition of a symbol class is terminated by a semicolon.

```
<symbol_class_definition> ::= <kcode> <symb_class_mode>
                              <symbol_class_name> =
                              <regular_expr> {<regular_expr>} ; |
                              <kcode> <symbol_class_name>
```

Note that if no generation of a scanner is intended the specification of class code (kcode) and symbol class name will suffice.

```
<symbol_class_name> ::= <identifier>

<kcode> ::= <unsigned_integer>

<symbol_class_mode> ::= - | + | * | <empty>
```

The **Class Mode** may be used to indicate in which way the generated scanner is to treat members of a specific symbol class. There are the following choices:

' ' : The scanner will interpret this symbol class as a constant class which is not altered during the scanning process, i.e. it can not be altered during the scanning process.

'-' : These classes consist of **ignorable** symbols which will not be passed on to the calling parser by the scanner.

'+' : The scanner will keep track of all symbols of these classes. Each of these symbols will receive a unique natural number called its **relative code**. The scanner passes this relative code (together with the symbol's class code) to the parser every time it finds an occurrence of the symbol. Typically, identifiers in a programming language are defined to form such a class.

'*' : The scanner keeps track of all members belonging to such a class. It will not, however, look it up among those already found. Instead it will assign a new relative code for each occurrence of a member of this class. One might give the class of strings this class mode, since it may be rather expensive to compare each string with all others already found.

The regular expression describes the lexical structure of all elements of a symbol class; in this description Character Classes and already defined Symbol Classes may be used. It is also possible to explicitly give a string in this description.

```
<regular_expression> ::= <symbol_class_name> |
                         <character_class_name> |
                         <character_constant> |
                         <string> |
                         <alternative> |
                         <equivalence_expression> |
                         <iteration_option> |
                         <allbut_expression>

<alternative> ::= ( <regular_expression> |
                    {<regular_expression> |}
                    <regular_expression> )

<concatenation> ::= <regular_expression>
                    { <regular_expression> }

<equivalence_expression> ::= ( <regular_expression> ,
                               {<regular_expression> ,}
                               <regular_expression> )

<iteration_option> ::= [ <regular_expression> ] |
                       * [ <regular_expression> ] |
                       * <range> [ <regular_expression> ]

<range> ::= <unsigned_integer> - <unsigned_integer> |
            <unsigned_integer> - |
            - <unsigned_integer>


<allbut_expression> ::= <regular_expression>
                        allbut ( <regular_expression> )
```

Notes:

(1) A sequence of <character_constants> will contain a space character ("white space") between each of the single characters. Thus, the input of '<''=' (instead of '<'_'=', with _ denoting a white space) will erroneously lead to the definition of a **string** '<'=' (containing a single quote) instead of the envisaged sequence of characters.

(2) Sequences of characters or groups of characters as constituents of symbols can be defined by exploiting the <iteration_option> alternative. In any case, the body describing the elements of the sequence is enclosed in brackets ('[' and ']').

(3) An **option**, i.e. a sequence of 0 or 1 instances of the body, is specified by given nothing else but the body enclosed in brackets. To denote a sequence of 0 or more instances of the body the bracketed body is preceded by an asterisk symbol. Besides, upper and lower limits of the number of instances may be expressed by a range specification.

The form '*n-m[...]' specifies that at least n instances of '...' are needed and at most m instances are allowed to form a sentence. Any of the two limit specifications may be omitted at a time thus giving '*n-[...]' to express a lower limit n and '*-m[...] to express an upper limit m only. In the latter case a lower limit n=0 is assumed; if no limits are given at all, the number of instances may vary between 0 and infinity.

(4) **Equivalence Classes** are used to group different representations for the same symbol. All members of such a class are regarded as semantically equivalent and for all of them the same pair (class code, relative code) is passed to the parser.

(5) **Allbut expression** are used when it is easier to describe the complement of something then the "something" itself. Typically, strings and comments are described using allbut expressions.

Constraints:

(1) Names of symbol classes must be pairwise different and not be equal to names of character classes.

(2) Characters which have been declared ignorable may not occur in symbol class definition.

(3) Names used in a symbol class definition may only be names of nonignorable character classes or of textually preceeding symbol classes. In the latter case these classes must not be constant nor contain an allbut expression.

(4) (Strings)

a) Strings may occur in constant symbol classes only.

b) Strings may not be concatenated with other regular expressions.

c) For every string there must be a defined symbol class decribing the *pattern* of the string specified. The generated scanner will accept the pattern symbol class; in the subsequent screening part, only, the appropriate symbol class number of the specified string will be assigned.

(5) A regular expression describing a constant symbol class has the following form:

(a) a character or the name of a character class
(b) a concatenation of elements of (a)
(c) a string
(d) an equivalence class formed of elements of (a)-(c)
(e) alternatives of expressions as constructed by (a)-(d)

(6) Semantic Equality may only be used in constant symbol classes.

(7) The regular expressions in allbut-expressions have the following form:

(a) a character or the name of a character class
(b) concatenations of elements of (a)
(c) alternatives of (a) and (b)

(8) A regular expression of the form R1 **allbut** (R2) may not be part of any other regular expression (e.g. as part of a alternative specification).

(9) Ignorable Symbol Classes must not be used as terminal symbols in the syntax definition part of the POCO input as they are absorbed by the scanner and will not eventually be returned to the calling parser.

4. Declaration of Named Constants

The declaration of named constants allows for easy attributation and facilitating the specification of types; this part of the GIL is equal to the syntax of the programming language PASCAL except for the decla-

ration of **real**-constants which is not supported.

```
<constant_definition_part> ::= <empty> |
                               const  <constant_definition>
                              { ; <constant_definition> } ;

<constant_definition> ::= <identifier> = <constant>

<constant> ::= <unsigned_number> |
               <sign> <unsigned_number> |
               <constant_identifier > |
               <sign> <constant_identifier> |
               <string>

<unsigned_number> ::= <unsigned_integer>

<unsigned_integer> ::= <digit> { <digit> }

<sign> ::= + | -

<string> ::= ' <character> {<character>} '

<constant_identifier> ::= <identifier>
```

5.Definition of Attribute Types:

The definition of attribute types is performed in correspondance to the declaration of types in the programming langugage PASCAL. Only few restrictions exist such as the definition of **file** types (which is not allowed) or the omission of the **real** standard type.

```
<type_definition_part> ::= type <type_definition>
                                {; <type_definition>}
                              | <empty>

<type_definition> ::=  <identifier> = <type>

<type> ::= <simple_type> |
           <structured_type> |
           <pointer_type>

<simple_type> ::= <scalar_type> |
                  <subrange_type> |
                  <type_identifier>

<type_identifier> ::= <identifier>
```

```
<scalar_type> ::= ( <identifier> {, <identifier> } )

<subrange_type> ::= <constant> .. <constant>


<structured_type> ::= <unpacked_structured_type> |
                      packed <unpacked_structured_type>


<unpacked_structured_type> ::= <array_type> |
                               <record_type> |
                               <set_type>


<array_type> ::= array [ <index_type>
                         {, <index_type>} ] of
                         <component_type>

<index_type> ::= <simple_type>

<component_type> ::= <type>

<record_type> ::= record <field_list> end

<field_list> ::= <fixed_part> |
                 <fixed_part> ; <variant_part> |
                 <variant_part>

<fixed_part> ::= <record_section> {; <record_section>}

<record_section> ::= <field_identifier>
                     {, <field_identifier>} : <type> |
                     <empty>

<variant_part> ::= case <tag_field> <type_identifier>
                     of <variant> {; <variant>}

<tag_field> ::= <field_identifier> :  | <empty>

<variant> ::= <case_label_list>
```

6. **Declaration of Semantic Actions**:

Semantic actions are declared as interface procedures of semantic **modules**; the PASCAL-m module concept is used. Inherited attributes are denoted as *value* parameters, derived attributes as *var* parameters in the usual sense of the programming language PASCAL.

We will present the full syntax of the PASCAL-m language extension as

a matter of completeness. Note, however, that in the GIL all modules have to be declared as external, i.e. as is separately compiled modules.

```
<module declaration>  ::= module <module identifier> ;
                           <interface part>
                           <implementation part>

<interface part>      ::= interface
                           <constant definition part>
                           <type definition part>
                           <procedure/function heading>
                          {<procedure/function heading>}

<implementation part> ::= implementation <block>
```

In the interface part of a module the semantic action procedures of the attributed grammar may be declared. Note that (normally) at least one interface procedure must be specified, otherwise the module might be useless as its internally declared data cannot be accessed from outside. In this case these data can only be accessed during the execution of the module body which will take place automatically on entering a block to which the module is declared local.

For a more detailed description of the module concept cf. the description of PASCAL-m programming language.

A module may be declared in a PASCAL block according to the following rule:

```
<block> ::= <label declaration part>
            <constant definition part>
            <type definition part>
            <variable declaration part>
            <module declaration part>
            <procedure and function declaration part>
            <statement part>
```

and

```
<module declaration part> ::= { <module declaration> ; }
```

Modules may be declared forward to allow for intermodule references. The way to express this feature corresponds to the forward declaration of PASCAL procedures:

```
<module declaration>  ::= module <module identifier> ;
                          <interface part>
                          forward
```

A module may be declared external according to the following rule:

```
<module declaration> ::= module <module identifier> ;
                         <interface part> use
```

This allows for separate compilation of modules. The corresponding rule in PASCAL-m is:

```
<compilation unit> ::= <program> | <module>

<module>           ::= <module declaration>
                       {; <module declaration>} .
```

7. Definition of the Grammar Axiom:

```
<axiom_definition> ::= axiom <nonterminal>

<nonterminal> ::= <identifier>
```

8. Specification of the Productions of the Attributed Grammar:

```
<production_definition> ::= productions <production>
                                { <production> }
```

Productions are written in *van Wijngarden* notation, i.e. each produc-

tion is terminated by ".", alternatives to a production are separated by ";", the elements of an alternative are separated by ",". Between the left hand side of a production and its right hand side stands an ":".

```
<production> ::= <left_side> : <right_side> .

<left_side> ::= <nonterminal> { <lhs_attributes>}

<lhs_attributes> ::= ( <attribute_group>
                       {; <attribute_group> } )

<attribute_group> ::= <inh_attribute_group> |
                      <der_attribute_group>

<inh_attribute_group> ::= <attribute_list> | <empty>

<der_attribute_group> ::= var <attribute_list> | <empty>
```

Note that inherited (derived) attributes are denoted as *value* (*var*) parameters in the sense of the programming language PASCAL; this is a concession to more notational uniformity.

```
<attribute_list> ::= <attribute_identifier>
                     { , <attribute_identifier> }
                       : <type identifier>
```

In correspondance to a PASCAL procedure call, an attribute type is specified by giving the name of a declared type. An explicite type specification is not possible.

```
<right_side> ::= <alternative> {; <alternative>}

<alternative> ::= <alt_element> { , <altelement> } |
                  <empty>

<alt_element> ::= <alt_header> |
                  <alt_header> ( <rhs_attribute>
                               { , <rhs_attribute> } )

<alt_header> ::= <nonterminal> | <terminal> |
                 call <sem_action>
```

Note: The use of the **call** symbol is redundant and is included for reasons of improved readability, only.

```
<sem_action> ::= <procedure identifier> |
                 <procedure identifier> ( <actual_parameter>
                                { , <actual_parameter> } )

<actual_parameter> ::= <attribute_identifier>

<attribute_identifier> ::= <identifier> |
                           <constant_identifier>

<rhs_attribute> ::= <attribute_identifier>

<empty> ::=
```

Notes:

(1) The use of name constants on attribute positions or parameter positions in semantic action calls is allowed only if they are used on derived positions of the rule's left hand side or inherited positions on the rule's right hand side.

(2) No Inherited Attributes are allowed to be associated with the underlying CFG's axiom.

(3) Each Derived Attribute associated with the left hand side nonterminal must be assigned a value, i.e. must appear on on derived position of the rule right hand side.

3. POCO Generator Options:

There exists a wide range of options which may be used to alter the generation process or to influence the amount of generation lists and diagnostics.

An option is specified as a pseudo-comment according to the following format:

```
<option_spec> ::= (*$ <option_list> *)

<option_list> ::= <option> {, <option> } |
                  <empty>

<option> ::= <option_id> + | <option_id> - |
             <option_id> <opt_number>

<opt_number> := '0' | '1' | '2' | '3' | '4'
```

where '+' stands for : option set, '-' option is not set; option numbers are evaluated only if given with the 'L'-option (cf. below!).

For <option_id> there exist the following choices; we denote in parentheses the preset initial value of the option upon start of the generator:

```
L (+)     : List Input
X (-)     : Generate Cross Reference List
S (-)     : Generate Scanner
P (-)     : Generate Parser
A (-)     : Generate Compile Time Attribute Evaluation
O (-)     : Optimize Generated Compiler Modules
T (-)     : List Internal Tables
D (-)     : List Additional Generation Values
C (+)     : Generate Modules from Internal Data Structure
Z (-)     : Pretty Print Options (may lead to a lot of paper)
G (-)     : Print AG-Transformed Grammar
B (-)     : Generate Parser Tables, not Code
```

Example:

The option specification

(*$L+,P+,C-*)

will lead to

L + : Complete Input Protocol list

P + : Generation of an LALR(1) - Parser

C - : without generating PASCAL-m source code
for the generated Parser module.

List Option Specials:

As the lists produced by the generator tend to be rather bulky the user is free to specify more precise list options. This is achieved by giving an 'L'-option with a subsequent option_number. The option numbers habe the following meaning:

L0 : No lists at all (same as L-)

L1 : List generator input only

L2 : List Scanner Generator Output only

L3 : List Attribute Evaluator Generator Output only

L4 : List Parser Generator Output only

Note that any number of these choices can be given, thus

$L1,L4 : will produce input and parser generator lists

$L1,L2,L3,L4 : means the same as L+

Note:

The user may use capital or lower case letters for any option identification; this applies to all available options.

4. POCO System Operating Procedures.

The POCO system can be used on the Siemens 7561 machine running under the BS2000 operating system as well as the VAX 11/780 under the Berkeley UNIX operating system.

Operating procedures are rather simple, but do differ somewhat as will be explained in detail. We will consider the POCO compiler generator's operating procedures only.

The following general conventions hold:

(1) The POCO generator operates on three different files, each of them being of type TEXT in the normal Pascal sense.

(2) File #1 (internally assigned to the file variable 'input') is used as input file and must contain the specification of a programming language for which a compiler is to be generated.

(3) File #2 (interally assigned to the file variable 'output') is used as a list file documenting the generation process and including a readable version of the generated compiler components. As most of the time this file will be rather bulky cf. the generator options decription for selection of less output.

(4) File #3 (internally assigned to the file variable 'prr') is used as an output file for the generated compiler. This file can immediately be fed into the Pascal-m compiler for translation. Note that the 'C+' option has to be set in order to obtain code on this file.

4.1. Running POCO on a Siemens BS2000 installation.

The POCO system can most easily be accessed on a Siemens BS2000 installation by means of a /DO - procedure. An example of such a procedure is given below:

```
/PROCEDURE C,(&GRAM,&LIB='BSP.GRAMMATIK'),SUBDTA=&
/SYSFILE SYSDTA=(SYSCMD)
/EXEC $FMS
OPEN &LIB
SEL GRAM.&GRAM
END
/FILE #PRR.&GRAM,FCBTYPE=SAM,LINK=PRR
/FILE #LIST.&GRAM,FCBTYPE=SAM,LINK=OUTPUT
/FILE GRAM.&GRAM,LINK=INPUT
/EXEC $KI.POCO
/RELEASE INPUT
/RELEASE OUTPUT
/RELEASE PRR
/STEP
/ER GRAM.&GRAM
/ENDP
```

A user may practically access the POCO system by giving the following command:

```
/DO KOM.GEN,name,LIB=bsp.eingabe
```

with:

name	being the suffix in GRAM.name a deck in a SIEMENS FMS-System file
bsp.eingabe	being a SIEMENS FMS-System file bearing GRAM.name as a deck

Using the /DO-procedure given above the following List- and Code-Files will be created:

#LIST.name	List - File
#PRR.name	Code - File (name is the suffix in GRAM.name)

These files are allocated on the temporary file system and have to be further processed before a /LOGOFF - command is given.

Note:

On the Universitaet des Saarlandes installation a predefined user procedure can be accessed via

```
/DO $ki.kom.gen, ...
```

with the parameters being the same as given above; the LIB-parameter may be omitted if it is the same as the predefined in the /DO-procedure.

4.2. VAX 11/780 Operating Procedures.

On a VAX 11/780 installation running under Berkeley UNIX the POCO system can be accessed by issuing the following command:

```
% POCO gram.name list.name code.name
```

with:

gram.name	POCO input file
list.name	protocol file
code.name	output file; holds a generated compiler

Note:

(1) File access paths must (of course) be fully specified.

(2) The names of all files can be freely chosen.

5. Example Input to the POCO System:

The following example is a legal (however simple) input to the

POCO compiler generating system:

```
(*$L+,X+,S+,O+,C+,A+,P+,G+,Z-*)

(* Options selected:
   ----------------

        L+ : List Input and Generator Report
        X+ : Cross Reference List of symbols
        S+ : Generate Scanner
        A+ : Generate Compile Time Attr. Comp.
        P+ : Generate Parser
        O+ : Optimize Parse Tables
        C+ : Generate PASCAL-m Code
        G+ : List Transformed Grammar
        Z- : No Pretty Print
*)

LANGUAGE AREX5

TERMINALS
                                    (* Character Class Definitions: *)
  BU = 'A' - 'Z';                   { letter character class }
  ZI = '0' - '9';                   { digit character class }
  AST = '*';
  PLUS = '+'  ;
  MINUS = '-' ;
  RKA   = '(' ;
  RKZ   = ')' ;
  EQU   = '=' ;
  DVD   = '/';
                                    (* Symbol Class Definitions: *)
  1 + ID = BU *-7[(BU|ZI)] ;        { identifier: up to 8 characters,
  2 ADDOP = (PLUS|MINUS) ;            the first one being a letter}
  3 MULOP = ('*'|DVD) ;
  4 RKAUF = RKA;
  5 RKZU  = RKZ;
  6 + INTCONST = ZI*[ZI] ;          { integers: any number of digits }
  7 EXPOP = AST AST ;
  8 RELOP = EQU ;
  99 - COMMENT = DVD AST ALLBUT (AST DVD);  { /* ... */ is ignor. }

                                    (* some dummy const/type decl. *)
CONST EINS = 1;
      ZEHN = 10; MZEHN = -ZEHN;
TYPE TYPEART = (INT,REEL);
     BSPTYP = ARRAY[MZEHN:ZEHN] OF CHAR;
MODULE M1;
  INTERFACE
     PROCEDURE GETTYP(IDNO : INTEGER; VAR TYP : TYPEART);
     PROCEDURE CHECKTYP(TYP1,TYP2 : TYPEART; VAR TYP : TYPEART);
USE;
```

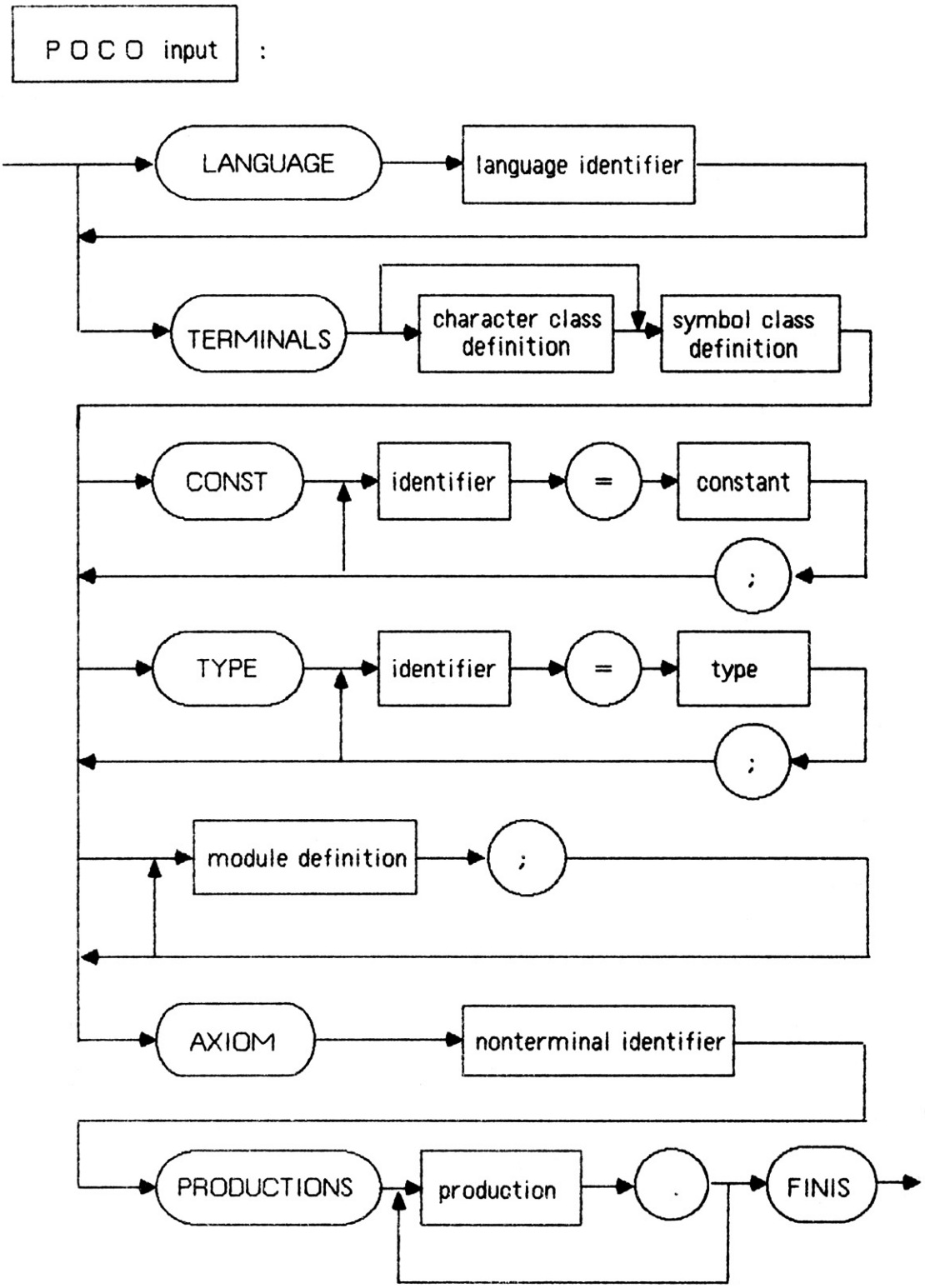
```
AXIOM AREX

PRODUCTIONS

  AREX (VAR TYP : TYPEART) : AREX (TYP1), ADDOP, TERM(TYP2),
                             CALL CHECKTYP (TYP1,TYP2,TYP);
                             TERM(TYP).

  TERM (VAR TYP : TYPEART) : TERM (TYP1), MULOP, FACTOR(TYP2),
                             CALL CHECKTYP(TYP1,TYP2,TYP);
                             FACTOR(TYP).

  FACTOR (VAR TYP : TYPEART): FACTOR(TYP), EXPOP, INTCONST;
                              FACTOR(TYP), EXPOP, RELOP, INTCONST;
                              PRIMEX(TYP).

  PRIMEX(VAR TYP : TYPEART): ID(IDNO), CALL GETTYP(IDNO,TYP);
                             RKAUF,AREX(TYP),RKZU.

FINIS
```

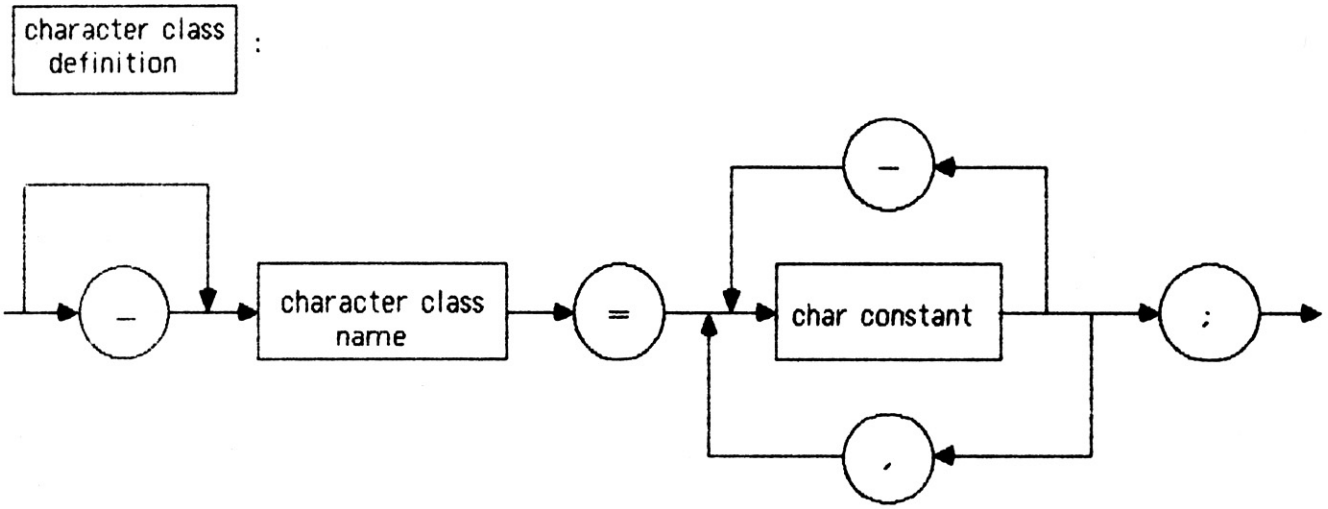
Note:

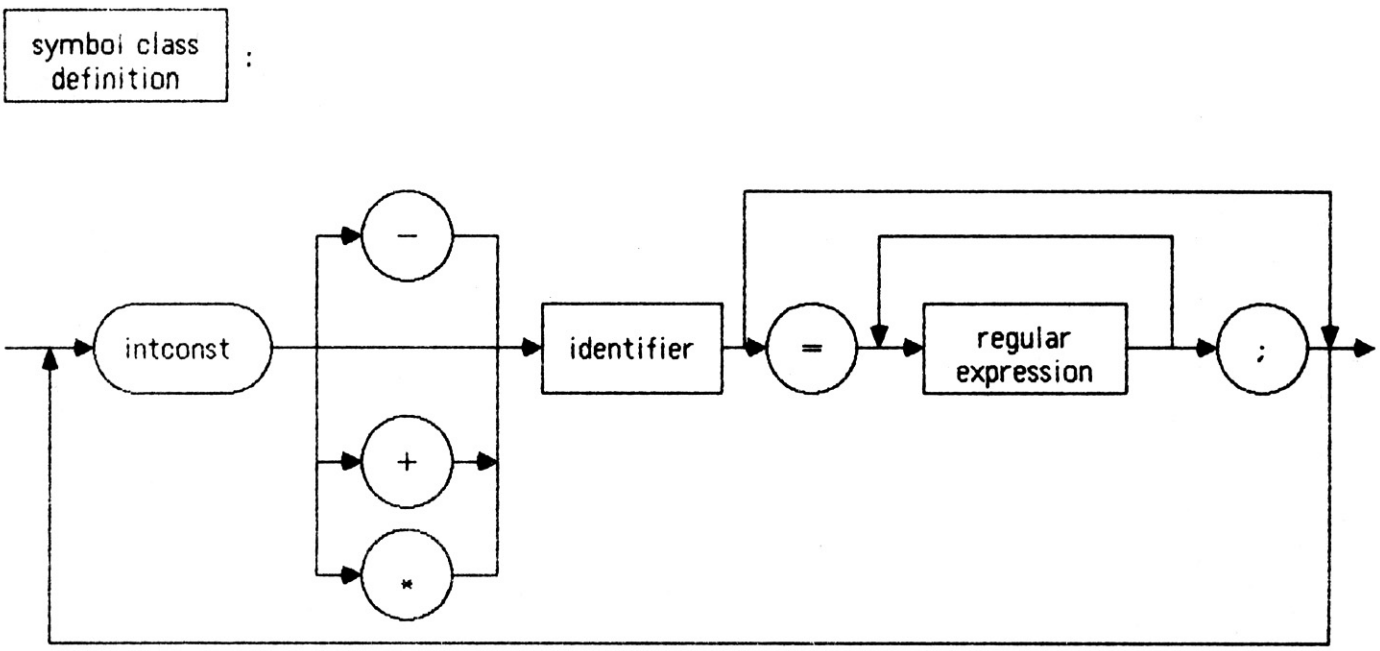
The user is not restricted to use capital letters, but can freely choose between lower or upper case. Internally, however, each symbol will be converted to upper case.

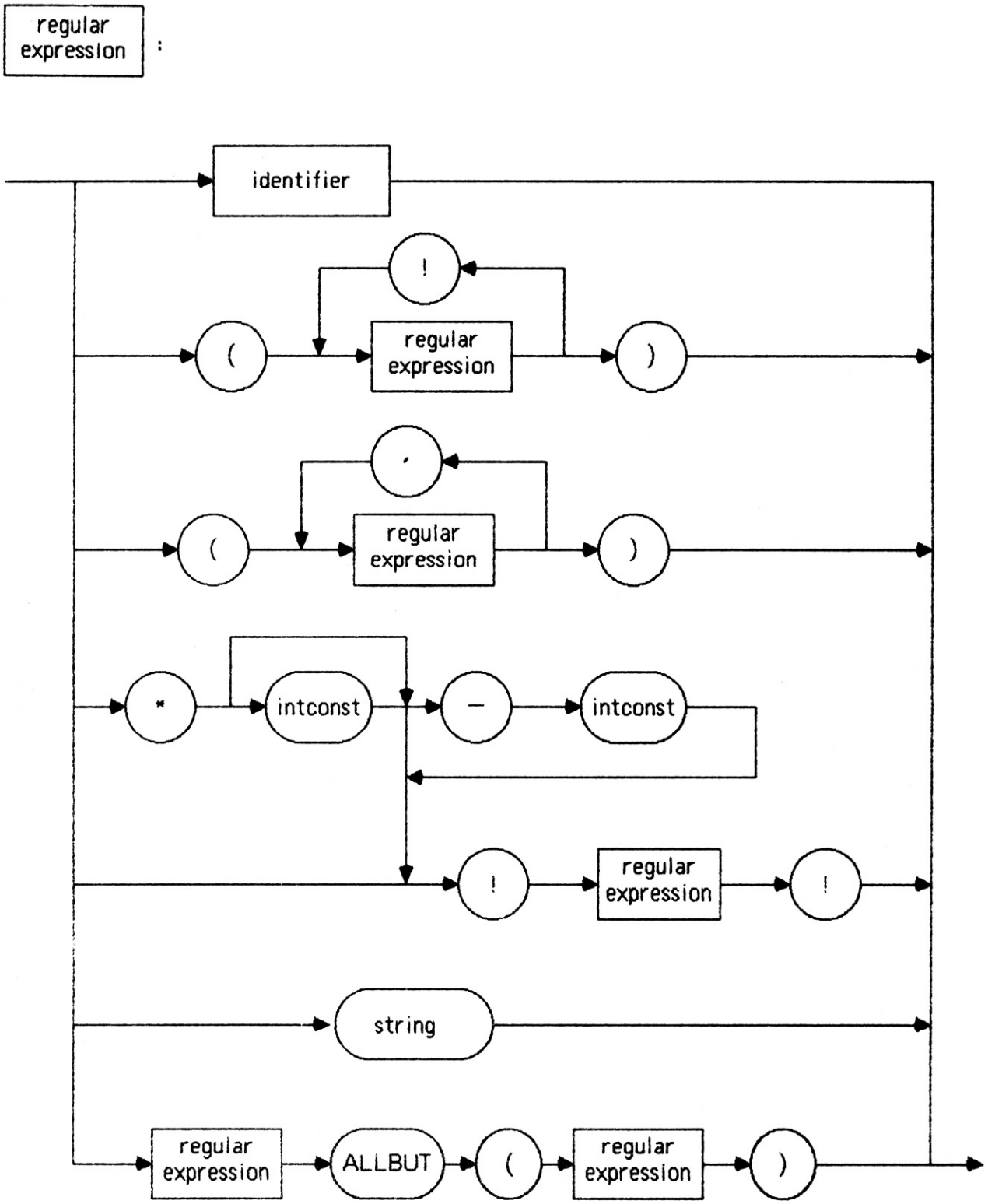
Appendix A:

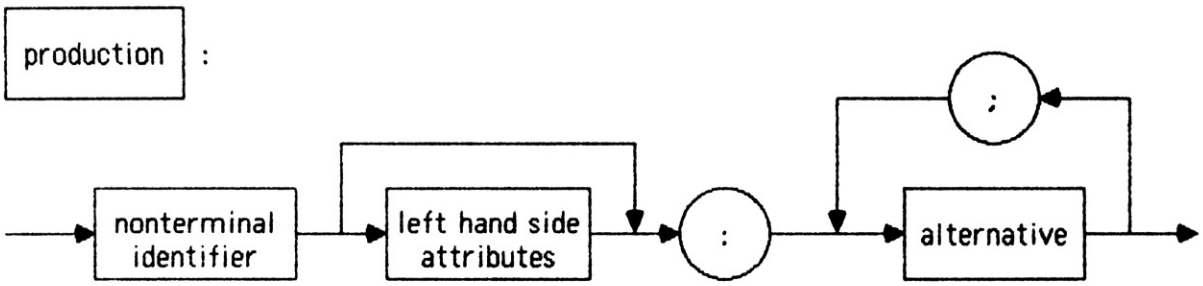
POCO Input Language Syntax Charts

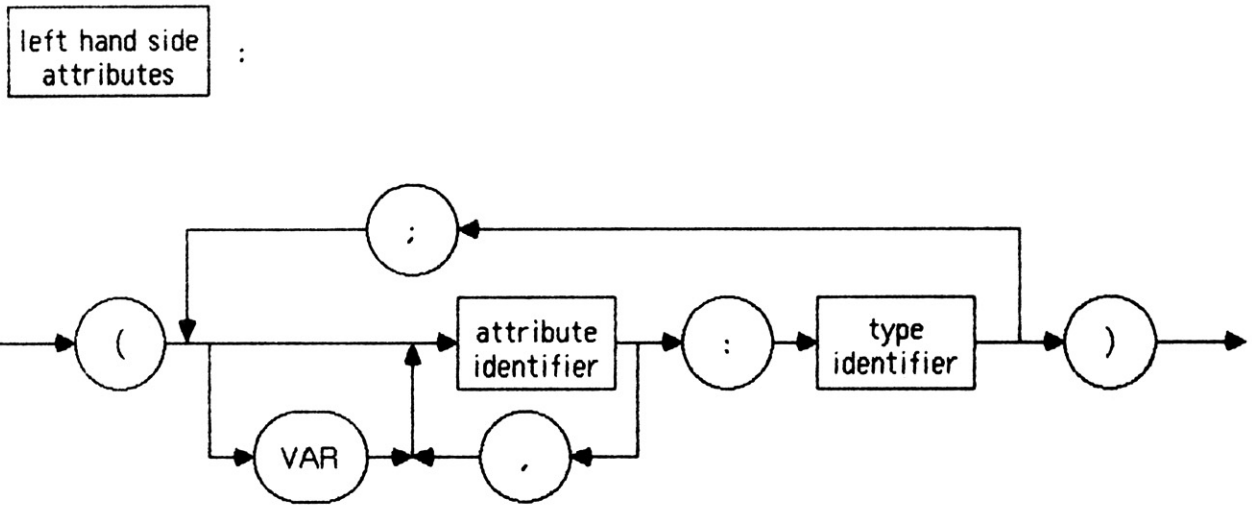


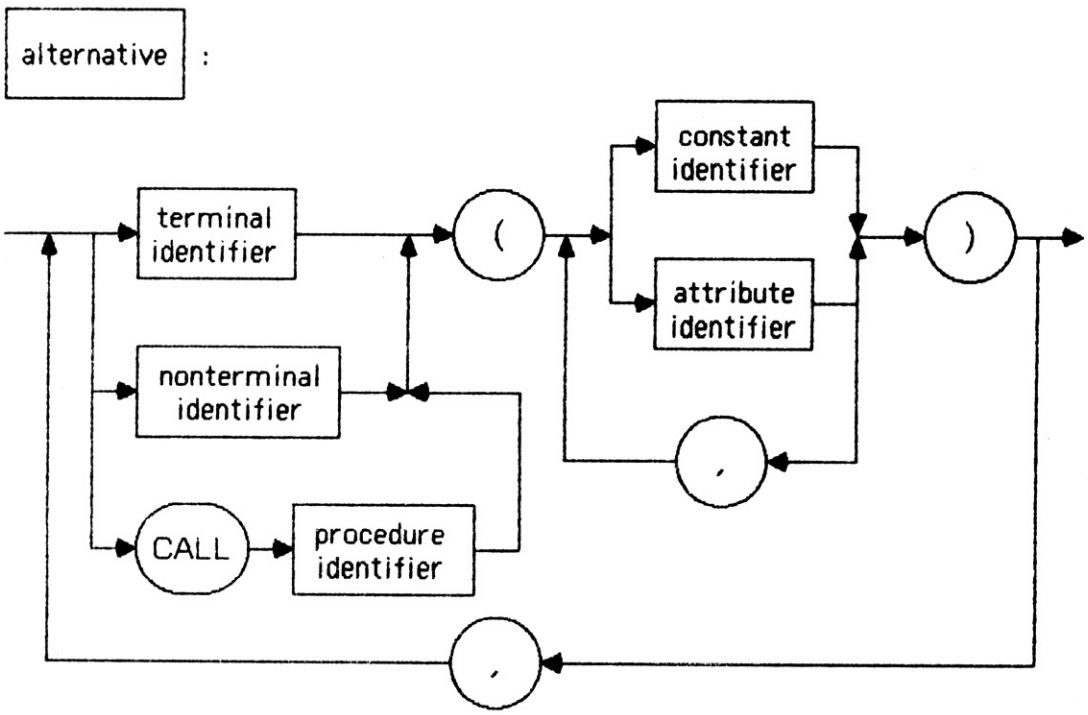

character class definition
:
_
character class name
=
_
char constant
'
;

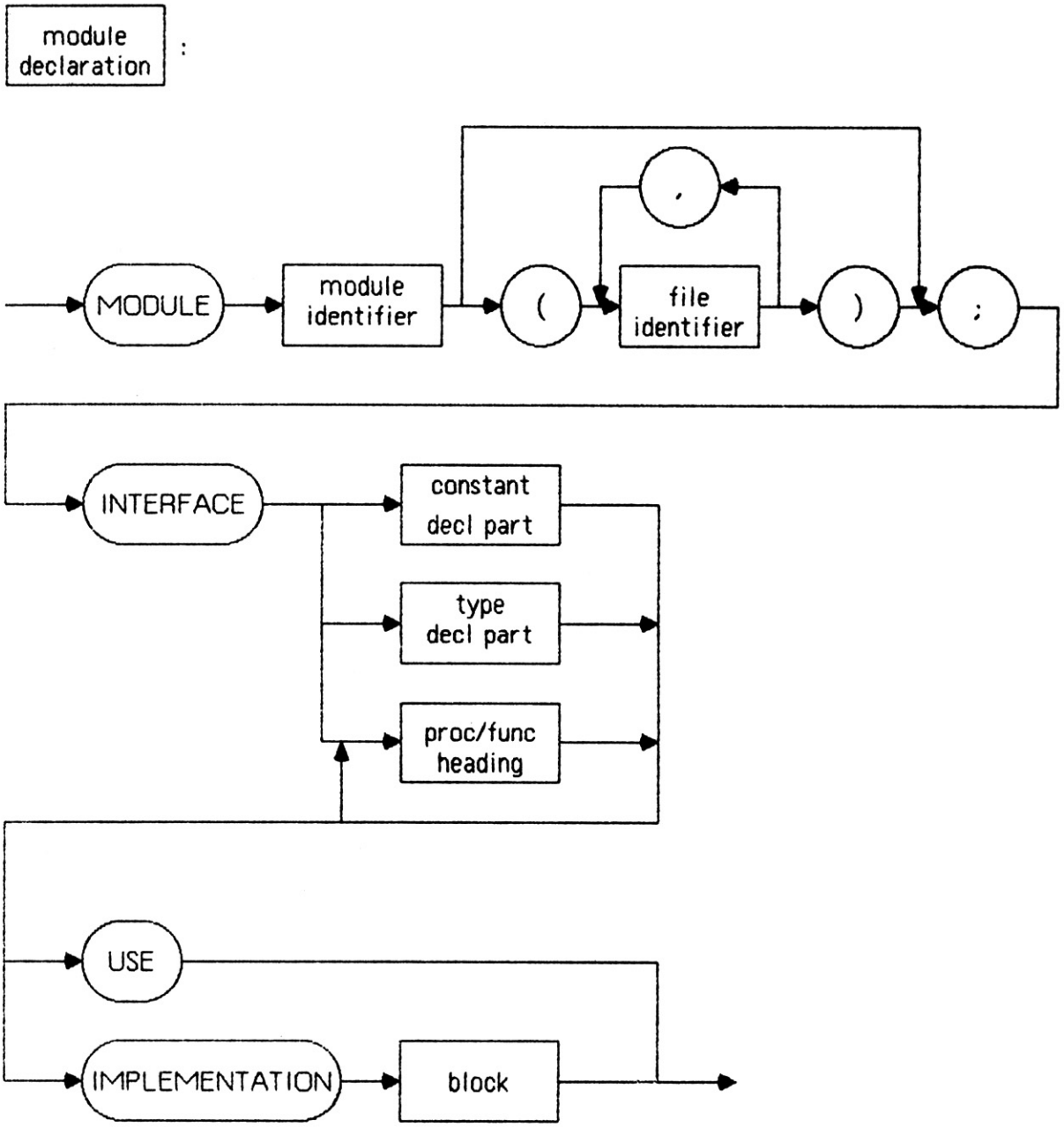

symbol class definition
:
intconst
_
+
*
identifier
=
regular expression
;


regular expression
:
identifier
!
(
regular expression
)
'
(
regular expression
)
*
intconst
–
intconst
!
regular expression
!
string
regular expression
ALLBUT
(
regular expression
)


production
:
nonterminal identifier
left hand side attributes
:
;
alternative


left hand side attributes
:
(
;
VAR
attribute identifier
,
:
type identifier
)


alternative
:
terminal identifier
nonterminal identifier
CALL
procedure identifier
(
constant identifier
attribute identifier
,
)
,



Note:

Any construct for which no syntax chart is given is identical to the programming language Pascal.

Appendix B:

Summary of POCO Generator Error Messages

```
  1 : Error In Simple Type
  2 : IDENTIFIER Expected; DO NOT USE PASCAL RESERVED WORDS!
  3 : 'PROGRAM' Expected
  4 : ')' Expected
  5 : ':' Expected
  6 : Illegal Symbol
  7 : Error In Parameter List
  8 : 'OF' Expected
  9 : '(' Expected
 10 : Error In Type
 11 : '[' Expected
 12 : ']' Expected
 13 : 'END' Expected
 14 : ';' Expected
 15 : INTEGER Expected
 16 : '=' Expected
 17 : 'BEGIN' Expected
 18 : Error In Declaration Part
 19 : Error In Field List
 20 : '.' Expected
 21 : '*' Expected
 22 : ',' Expected

 41 : Module Must Be Declared Local To Program.
 42 : 'USED' Or 'IMPLEMENTATION' Expected
 43 : 'INTERFACE' Expected
 44 : Module Interface Part Must Not Be Empty
 45 : 'IMPLEMENTATION' Expected

 50 : Error In Constant
 51 : ':=' Expected
 52 : 'THEN' Expected
 53 : 'UNTIL' Expected
 54 : 'DO' Expected
 55 : 'TO'/'DOWNTO' Expected
 56 : 'IF' Expected
 57 : 'FILE' Expected
 58 : Error In Factor
 59 : Error In Expression

101 : IDENTIFIER Declared Twice
102 : Low Bound Exceeds High Bound
103 : IDENTIFIER Is Not Of Appropriate Class
104 : IDENTIFIER Not Declared
105 : Sign Not Allowed
106 : Number Expected
107 : Incompatible Subrange Types
108 : File Not Allowed Here
109 : Type Must Not Be Real
110 : Tagfield Type Must Be Scalar Or Subrange
111 : Incompatible with Tagfield Type
112 : Index Type Must Not Be REAL
```

113 : Index Type Must Be Scalar Or Subrange
114 : Base Type Nust Not Be REAL
115 : Base Type Must Be Scalar Or Subrange
116 : Error In Type Of Standard Procedure Parameter
117 : Unsatisfied Forward Reference
118 : Forward Reference Type Identifier In Variable Declaration
119 : Forward Declared; Repetition Of Parameter List Not Allowed
120 : FUNCTION Result Parameter Must Be Scalar, Subrange Or Pointer
121 : FILE Value Parameter Not Allowed
122 : Forward Declared Function; Repetition Of Result
123 : Missing Result Type In FUNCTION Declaration
125 : Error In Type Of STANDARD FUNCTION PARAMETER
126 : Number Of Parameters Does Not Agree With Declaration
127 : Illegal Parameter Substitution
128 : Result Type Of Parameter FUNCTION Does Not Agree With Declaration
129 : Type Conflict Of Operands
130 : Expression Is Not Of SET Type
131 : Tests On Equality Allowed Only
132 : Strict Inclusion Not Allowed
133 : FILE Comparison Not Allowed
134 : Illegal Type Of Operand(s)
135 : Type Of Operand Must Be BOOLEAN
136 : Set Element Type Must Be Scalar Or Subrange
137 : Set Element Types Not Compatible
138 : Type Of Variable Is Not ARRAY
139 : Index Type Is Not Compatible With Declaration
140 : Type Of Variable Is Not Record
141 : Type Of Variable Must Be File Or Pointer
142 : Illegal Parameter Substitution
143 : Illegal Type Of Loop Control Variable
144 : Illegal Type Of Expression
145 : Type Conflict
146 : Assignment Of Files Not Allowed
147 : Label Type Incompatible With Selecting Expression
148 : Subrange Bounds Must Be Scalar
149 : Index Type Must Not Be INTEGER
150 : Assignment To Standard FUNCTION IS Not Allowed
151 : Assignment To Formal FUNCTION IS Not Allowed
152 : No Such Field In This Record
153 : Type Error In READ
154 : Actual Parameter Must Be a Variable
155 : Control Variable Must Not Be Declared On Intermediate Level
156 : Multidefined CASE Label
157 : Too Many Cases In CASE Statement
158 : Missing Correspondent Variant Declaration
159 : REAL or STRING Tagfields Not Allowed
160 : Previous Declaration Was Not Forward
161 : Again Forward Declared
162 : Parameter Size Must be Constant
163 : Missing Variant In Declaration
164 : Substitution Of Standard PROC/FUNC Not Allowed
165 : Multidefined LABEL
166 : Multideclared LABEL
167 : Undeclared LABEL
168 : Undefined LABEL
169 : Error In Base Set
170 : VALUE Parameter Expected
171 : Standard FILE Was Redeclared
172 : Undeclared External FILE
177 : Assignment To FUNCTION Identifier Not Allowed Here

```
178 : Multidefined Record Variant
179 : X-OPT OF ACTUAL PROC/FUNC DOES NOT MATCH FORMAL DECLARATION
180 : Control Variable Must Not Be Formal
181 : Constant Part Of Address Out Of Range

201 : Error In REAL Constant : Digit Expected
202 : STRING Constant Must Not Exceed Source Line
203 : INTEGER Constant Exceeds Range
205 : Empty STRING Not Allowed
206 : INTEGER Part Of REAL Constant Exceeds Range

250 : Too Many Nested Scopes of Identifiers
251 : Too Many Nested PROCEDURES and/or FUNCTIONS
252 : Too Many FORWARD References of PROCEDURE entries
254 : Too Many Long Constants in this PROCEDURE
255 : Too Many Errors in this Source Line

300 : Division By Zero
301 : No Case Provided For This Value
302 : Index Expression Out Of Bounds
303 : Value To Be Assigned Is Out Of Range
304 : Element Expression Out Of Range

390 : String Exceeds Max. Possible Length
398 : Implementation Restriction
399 : Variable Dimension Arrays Not Implemented

450 : Symbol Class Defined Twice
451 : Character Constant Expected
452 : Illegal Neglect Symbol
453 : Character Sets Must Be Ordered
454 : Character Defined In More Than One Symbol Class
455 : Illegal Symbol Class Mode
457 : Regular Expression Table Space Exhausted
458 : Character Class or Symbol Class Name Expected
460 : Ignorable Symbols Not Allowed
461 : Ignorable Symbol Class Name Not Allowed
462 : Lower Bound Exceeds Upper Bound
463 : Strings Not Allowed in Non-Constant Symbol Classes
464 : Concatenation of Strings Not Allowed
465 : Only Single Element Character Classes Allowed in Constant Symbol Class
466 : Non-Constant Symbol Class Name Not Allowed
467 : '-' Expected
468 : Constant Class Name Not Allowed
469 : write(output,'Constant String Used Twice
470 : First Part of 'ALLBUT' Expression Malformed
471 : Symbol Class Name Not Allowed in 'ALLBUT' Argument
472 : Only Single Element Character Classes Allowed in 'ALLBUT' Expression
473 : Strings Not Allowed in 'ALLBUT' Expressions
474 : Option Not Allowed in 'ALLBUT' Expression
475 : Semantic Equality Not Allowed in 'ALLBUT' Expression
476 : Options Not Allowed in Constant Symbol Classes
477 : No Nested 'ALLBUT' Expressions Allowed
478 : 'ALLBUT' Not Allowed in Constant Symbol Classes
480 : Semantic Equality Not Allowed in Non-Constant Symbol Classes
483 : To Many Nested Alternatives/Semantic Equalities
484 : Malformed Constant Symbol Class Definition
495 : Scanner Generator: Internal Table Overflow; Processing Aborted.
499 : Malformed Regular Expression
```

```
501 : Illegal Option Or Option Format
502 : GL: Illegal Terminal Symbol Class Code
503 : GL: Terminal Symbol Class Code Used Twice
504 : Ignorable Symbol Class Not Allowed in Syntax Definition
510 : GL: AXIOM Definition Expected
511 : Axiom Must Not Have Inherited Attributes.
512 : GL: ';' Expected
513 : GL: ','/';'/'.' Expected
520 : GL: 'PRODUCTIONS' Expected
530 : GL: Nonterminal Defined Twice
550 : GL: 'FINIS' Expected
560 : GL: Undeclared Semantic PROCEDURE/FUNCTION
566 : GL: Terminal Symbol Expected

620 : Nonterminal Identifier Expected
621 : Illegal Symbol in Alternative
636 : Inconsistent Number of Terminal Symbol Attributes
637 : Number of Attributes Does Not Agree With Left Side Nonterminal
      Declaration
638 : Inconsistent Number of Attributes For This Nonterminal
      In Preceeding Input
639 : Inconsistent or erroneous Attribute Usage in Preceeding Input
640 : Inherited Attribute Did Not Appear on a Defining Position
641 : (Some) Left Hand Side Derived Attributes Are Not Assigned a Value
660 : GL: Undeclared Terminal
697 : The Following Nonterminal Does Not Derive a Terminal String:
698 : The Following Nonterminal Cannot Be Reached From the Axiom:
699 : The Following NONTERMINAL does not have an Associated Rule:
701 : SG: Error in Symbol Class Definition:
800 : AG:  The Following Attribute is not found on the Attribute Stack:
810 : The Following Constant is not of adequate Type:
811 : Constant on Derived Position of Procedure Not Allowed:
```

For any other error number not mentioned above the following error message will be displayed:

```
  i : POCO - Generator Error: Unspecified.
```

These error codes are used for system maintenance purposes and will not normally appear in user listings.

Literature:

[Deg_85] : W. Degenhard,
"Ein Datenbanksystem fuer Pascal-m Moduln"
Diplom-Arbeit, Universitaet des Saarlandes, 1985.

[Eul_82] : M. Eulenstein,
"An Extension to Pascal for Modular Programming and a Proposal of a Conceptionally Machine Independent Linker"
in: Langmaack/Schlender/Schmidt (Hrsg.),
"Implementierung Pascal-artiger Programmiersprachen"
Teubner-Verlag, 1982.

[Eul_84] : M. Eulenstein,
"POCO - Ein portables System zur Generierung portabler Compiler"
PhD Thesis, Universitaet des Saarlandes, 1984.

[Eul_85] : M. Eulenstein,
"Pascal-m User Manual"
Universitaet des Saarlandes, forthcoming, 1985.

[Gro_85] : N. Gross,
"Direkte Generierung von Parser-Tafeln in Form von p-Code Moduln"
Diplom-Arbeit, Universitaet des Saarlandes, 1985.

[Kep_85] : W. Keper,
"BISAM - Ein maschinenunabhaengiger Binder fuer Pascal-m Moduln"
Diplom-Arbeit, Universitaet des Saarlandes, 1985.

[Man_85] : R. Mansmann,
"Direkte Generierung der Compile-Zeit-Attributberechnung in Form von p-Code Moduln"
Diplom-Arbeit, Universitaet des Saarlandes, 1985.

[WRC_76] : R. Wilhelm, K. Ripken, J. Ciesinger et al.,
"Design Evaluation of the Compiler Generating System MUG1"
Proc. 2nd Intl. Conf. on Software Engineering,
San Francisco, 1976.