A Front End for CONSUL

Catherine Chronaki
Douglas Baldwin

The University of Rochester
Computer Science Department
Rochester, New York 14627

Technical Report 319
January 1990

Abstract

CONSUL is a prototype constraint language for programming multiprocessors. In this paper we present a front end for CONSUL that makes the language easier to use than it originally was. The most important features of the front end are compact representation of constraints, type definitions, functional use of relations, and the ability to split programs into multiple files.
1 Introduction

CONSUL [2] is an experimental constraint language designed for general purpose programming. The original motivation for the language was to study implicit parallelism, but the language turned out to be of more general interest. Many programs have been successfully written in a variety of application areas. As these applications were being written, it became clear that a more user friendly syntax for the language was needed.

Constraint programming is a descendant of logic programming in that any constraint corresponds to a logic predicate. However constraint languages have certain relations built in in the form of primitive constraints. The primitive constraints in CONSUL represent basic arithmetic, comparisons and set operations. Because these primitives are so simple, programs written in raw CONSUL\(^1\) tend to be big. A lot of temporary variables and quantifiers are needed to express constraints at the level users find most natural.

The specifications set for the front end were:

- **Simplicity.** The front end should be simple enough for programmers to understand and become comfortable with it in a short time. This means that the notation should be familiar, either from set theory (if possible) or from widely used programming languages.

- **Compactness.** Expressions in the language should be compact, so that even powerful programs are short enough to be readable.

- **Separability.** The front end should enable the programmer to split a program across multiple files.

- **Structure.** The language of the front end should make the structure of CONSUL programs clear.

- **Semantic Checking.** Programmer errors that can be detected by the front end should be, in order to speed up program testing.

The front end has three phases. The first phase, parsing, parses the program into a syntax tree, the second one, optimization, applies basic optimizations to it, while the last one, code generation, generates raw CONSUL code. The syntax tree representation looks pretty much like raw CONSUL: the features of CONSUL that do not have direct analogs in raw CONSUL have been removed and the infix notation has been turned into Polish. The optimization phase performs constant folding and constant expression evaluation. Code generation breaks the s-expressions of the syntax tree into raw CONSUL by introducing the necessary quantifiers and temporary variables.

A number of possible extensions to this work lie ahead. One possibility is optimizing the code produced by the front end. The idea is to extend the limited optimization performed at present to a full blown optimization. There is indication that analogs of most imperative

\(^1\)By raw CONSUL, we mean the language accepted by the original CONSUL interpreter, as opposed to the language of the front end. From now on the term CONSUL will refer to the language of the front end.
language optimizations apply to CONSUL. However the declarative nature of CONSUL
means that the techniques for applying these optimizations to imperative languages won't
necessarily work for CONSUL. For example, terms like data flow analysis are not clearly
defined for CONSUL.

Another extension to the front end would be adding annotations and/or hints on how
to parallelize the program. This could be helpful to compilers for CONSUL.

Finally new features could be added to the language via macros, new statements, or
libraries. The need for such features will evolve as the front end is used in extensive
programming.

2 Raw CONSUL

CONSUL is very much an experimental prototype of a constraint language. It supports
demonstrations of realistic programs for a variety of applications, but in a laboratory set-
ing rather than industrial production. This section describes raw CONSUL, the language
accepted by the original CONSUL interpreter.

CONSUL's formal basis is in set theory. Formally, everything in CONSUL is a set.
Thus, the fundamental data type is the set and the fundamental operators are the logical
connectives (and, or, not) and quantifiers. However a number of abstractions have been
provided to make the language easier to use than raw set theory. For example, there are a
number of built in data types like sequences, integers, characters et cetera. Each of these
data types has a formal set theoretic definition, but most programs need not deal directly
with this aspect of CONSUL. Each built in data type is associated with built in relations
that correspond to common operations on that type. For example, simple comparisons and
arithmetic relations are defined for integers.

As an example of raw CONSUL, consider computing the GCD of two numbers. There
are at least two ways to do this. The first is to give a direct, very declarative, translation
into raw CONSUL of the definition of GCD, namely, the GCD of two integers b and c is the
positive integer a that divides both b and c such that there is no greater integer with the
same property. The second way is to compute the GCD using Euclid's algorithm. Figures 1
and 2 implement these two approaches. Figure 1 uses the declarative approach while figure
2 uses Euclid's algorithm.

The two raw CONSUL programs illustrate some of the ideas behind CONSUL. In figure
1 the user defined relation \texttt{dec\_gcd} (lines 1–12) is the actual definition of the GCD. Line 1
defines \texttt{dec\_gcd} as a subset of the cross product of three integers. This definition reflects
the fact that relations are formally sets. "Subset" and "cross" are examples of so called set
constructors. Set constructors are raw CONSUL's way of allowing programmers to write
constant sets. Line 2 introduces the formal parameters of the relation, \texttt{gcd}, b and c, while
lines 3–12 specify the constraints that \texttt{gcd}, b and c must satisfy for \texttt{gcd} to be the GCD of b
and c. Lines 4–5 state that \texttt{gcd} is a common divisor of b and c. Lines 6–12 state that there
exists no common divisor of b and c that is greater than \texttt{gcd}. All these constraints ought to
hold simultaneously, as indicated by the "and" on line 3. Lines 13–26 form the main part
(1) (define dec-gcd (subset (cross integer integer integer)
 (2)   (rho (gcd b c)
 (3)     (and
 (4)       (remainder 0 b gcd)
 (5)       (remainder 0 c gcd)
 (6)       (not
 (7)         (exists ((d (subset integer
 (8)             (rho (i)
 (9)               (greater i gcd)))))
 (10)         (and
 (11)             (remainder 0 b d)
 (12)             (remainder 0 c d))))))

(13) (exists ((out (sequence integer))
 (14)   (in (sequence integer))
 (15)   (a integer)
 (16)   (b integer)
 (17)   (c integer))

(18)   (and
 (19)     (size in 2)
 (20)     (size out 1)
 (21)     (input in "stdin")
 (22)     (elt a out 0)
 (23)     (elt b in 0)
 (24)     (elt c in 1)
 (25)     (dec-gcd a b c)
 (26)     (output out "stdout"))

Figure 1: Definition of gcd in raw CONSUL

of the program that relates input to output. The existential quantifier in line 13 introduces
the variables for which the problem has to be solved.

Input and output are modeled as sequences. Variables that represent these sequences are
bound to external files by the “input” and “output” constraints (lines 21 and 26). Lines 23
and 24 constrain \( b \) and \( c \) to be the first and second element of the sequence \( \) in
respectively. Finally, line 25 declares that the output is the GCD of the inputs.

Figure 2 is similar to Figure 1, except that it uses a different definition of GCD \( \text{rec}_\text{gcd} \).
Note that relations in CONSUL can be defined recursively, as shown in line 11. The main
part of the program (lines 12–31) is changed to reflect that \( b \) has to be greater than \( c \) for
Euclid’s algorithm to work. This is done by the “or” on lines 24–30.

The following features are the main reasons why CONSUL is well suited for general
purpose programming:

- Set theory provides data structures (sets), control structures (logical connectives) and
  block structures (quantifiers), that are all independent of execution order. Within
  these basically unordered semantics, sequences provide a way to describe sequential
  ordering when it is needed. These are the main sources of parallelism in CONSUL.
(1) (define rec-gcd (subset (cross integer integer integer)
(2)   (rho (gcd b c)
(3)     (exists ((tmp integer))
(4)       (or
(5)         (and
(6)           (remainder 0 b c)
(7)           (equal c gcd))
(8)         (and
(9)           (remainder tmp b c)
(10)          (not-equal tmp 0)
(11)         (rec-gcd gcd c tmp))))))
(12) (exists ((out (sequence integer))
(13)   (in (sequence integer))
(14)   (a integer)
(15)   (b integer)
(16)   (c integer))
(17)   (and
(18)     (size in 2)
(19)     (size out 1)
(20)     (input in "stdin")
(21)     (elt a out 0)
(22)     (elt b in 0)
(23)     (elt c in 1)
(24)     (or
(25)       (and
(26)         (greater b c)
(27)         (rec-gcd a b c))
(28)       (and
(29)         (less-equal b c)
(30)         (rec-gcd a c b))
(31)     (output out "stdout"))

Figure 2: Euclid’s algorithm for GCD in raw CONSUL
• Different programming styles are supported by CONSUL, ranging from highly declarative (Figure 1) to highly algorithmic (Figure 2). This gives programmers the ability to control program efficiency through proper choice of algorithms without leaving the declarative framework.

• Being a constraint language, CONSUL can express everything that can be expressed by predicate calculus.

After programming in CONSUL for a while, we realized that we needed a more compact syntax than the one illustrated in Figures 1 and 2. While the semantics of raw CONSUL seem well suited to general purpose constraint programming, its syntax makes it hard to use. For example, defining relations as subsets of cross products is verbose and needlessly exposes programmers to the underlying formalism. Similarly, the use of “rho” in subset definitions is an artifact of the formalism that has little meaning to programmers. The built in constraints are so simple that constructing nontrivial relations requires many temporary variables. The quantifiers that introduce these temporaries break up the structure of programs, making them less readable.

The front end language evolved out of attempts to sketch CONSUL relations. When developing a relation, one wants to concentrate on its behavior, ignoring syntactic details. We thus found ourselves describing CONSUL programs using notations from set theory and familiar programming languages, instead of raw CONSUL syntax.

Raw CONSUL is now the intermediate code for CONSUL. This evolution reverses the usual situation, in which intermediate code is chosen after the source language. In the case of CONSUL, we first concentrated on getting a clean semantics and only afterwards worried about the representation. The clean semantics and simple syntax of raw CONSUL make it an ideal intermediate code, even if it is poorly suited to human use.

3 The Front End

The front end extends raw CONSUL in two ways. First, it completely changes the syntax of the language. In particular, it provides infix notation, renames the primitive constraints, and changes the structure of relation definitions, quantifiers, and set constructors. These features provide compactness and improve the readability of programs.

Second, the front end adds features that make programs easier to maintain. A CONSUL program may be split into multiple files. Definitions may appear anywhere in a program. A definition that depends on other definitions may appear either before or after them, as long as the dependencies are not cyclic.

3.1 New syntax for existing features

The basic correspondence between the front end syntax and raw CONSUL is illustrated in Tables 1–7. A complete formal grammar for the front end language appears in the Appendix.
### Table 1: Correspondence of arithmetic operators

<table>
<thead>
<tr>
<th>Constraint</th>
<th>Raw Consul</th>
<th>Front End</th>
</tr>
</thead>
<tbody>
<tr>
<td>addition</td>
<td>(plus a b c)</td>
<td>a = b + c</td>
</tr>
<tr>
<td>subtraction</td>
<td>(minus a b c)</td>
<td>a = b - c</td>
</tr>
<tr>
<td>multiplication</td>
<td>(times a b c)</td>
<td>a = b * c</td>
</tr>
<tr>
<td>division</td>
<td>(divide a b c)</td>
<td>a = b / c</td>
</tr>
<tr>
<td>remainder</td>
<td>(remainder a b c)</td>
<td>a = b mod c</td>
</tr>
</tbody>
</table>

### Table 2: Correspondence of set constraints

<table>
<thead>
<tr>
<th>Set Constraint</th>
<th>Raw Consul</th>
<th>Front End</th>
</tr>
</thead>
<tbody>
<tr>
<td>subscription</td>
<td>(elt a b c)</td>
<td>a = b</td>
</tr>
<tr>
<td>difference</td>
<td>(set-minus a b c)</td>
<td>a = b :-: c</td>
</tr>
<tr>
<td>union</td>
<td>(union a b c)</td>
<td>a = b :U: c</td>
</tr>
<tr>
<td>intersection</td>
<td>(intersection a b c)</td>
<td>a = b &gt;&gt;: c</td>
</tr>
<tr>
<td>size</td>
<td>(size b a)</td>
<td>a =</td>
</tr>
<tr>
<td>index</td>
<td>(index pos pair)</td>
<td>pos = index(pair)</td>
</tr>
<tr>
<td>datum</td>
<td>(datum value pair)</td>
<td>value = datum(pair)</td>
</tr>
</tbody>
</table>

### Table 3: Correspondence of comparison constraints

<table>
<thead>
<tr>
<th>Comparison Constraint</th>
<th>Raw Consul</th>
<th>Front End</th>
</tr>
</thead>
<tbody>
<tr>
<td>equality</td>
<td>(equal a b)</td>
<td>a = b</td>
</tr>
<tr>
<td>inequality</td>
<td>(not-equal a b)</td>
<td>a != b</td>
</tr>
<tr>
<td>greater</td>
<td>(greater a b)</td>
<td>a &gt; b</td>
</tr>
<tr>
<td>less</td>
<td>(less a b)</td>
<td>a &lt; b</td>
</tr>
<tr>
<td>less-or-equal</td>
<td>(less-equal b a)</td>
<td>a &lt;= b</td>
</tr>
<tr>
<td>greater-or-equal</td>
<td>(greater-equal a b)</td>
<td>a &gt;= b</td>
</tr>
</tbody>
</table>

### Table 4: Correspondence of logic operators

<table>
<thead>
<tr>
<th>Logic Operator</th>
<th>Raw Consul</th>
<th>Front End</th>
</tr>
</thead>
<tbody>
<tr>
<td>and</td>
<td>(and a₁ ... aₙ)</td>
<td>a₁ ,... ,aₙ</td>
</tr>
<tr>
<td>or</td>
<td>(or a₁ ... aₙ)</td>
<td>a₁ ; ... ;aₙ</td>
</tr>
<tr>
<td>not</td>
<td>(not a)</td>
<td>!a</td>
</tr>
</tbody>
</table>
### Table 5: Correspondence of quantifiers

<table>
<thead>
<tr>
<th>Quantifier</th>
<th>Raw Consul</th>
<th>Front End</th>
</tr>
</thead>
<tbody>
<tr>
<td>Existential</td>
<td>(exists $((v_1 \text{ Set}_1) \cdots (v_m \text{ Set}_1) \cdots) \cdots (v_k \text{ Set}_n) \cdots (v_l \text{ Set}_n))$</td>
<td>$\exists v_{n1}, \ldots, v_n : \text{Set}<em>1; \ldots; v</em>{m}, \ldots, v_l : \text{Set}_n \mid \text{body}$</td>
</tr>
<tr>
<td>Universal</td>
<td>(forall $((v_1 \text{ Set}_1) \cdots (v_m \text{ Set}_1) \cdots) \cdots (v_k \text{ Set}_n) \cdots (v_l \text{ Set}_n))$</td>
<td>$\forall v_1, \ldots, v_m : \text{Set}_1; \ldots; v_k, \ldots, v_l : \text{Set}_n \mid \text{body}$</td>
</tr>
</tbody>
</table>

### Table 6: Correspondence of set constructors

<table>
<thead>
<tr>
<th>Set Constructor</th>
<th>Raw Consul</th>
<th>Front End</th>
</tr>
</thead>
<tbody>
<tr>
<td>Set</td>
<td>$\text{set elt}_1 \cdots \text{elt}_n$</td>
<td>${\text{elt}_1, \ldots, \text{elt}_n}$</td>
</tr>
<tr>
<td>Sequence</td>
<td>$\text{seq elt}_1 \cdots \text{elt}_n$</td>
<td>$&lt;&lt; \text{elt}_1, \ldots, \text{elt}_n&gt;&gt;$</td>
</tr>
<tr>
<td>Subset</td>
<td>$\text{subset set restr}$</td>
<td>${v_1 : \text{Set}_1; \cdots; v_n : \text{Set}_n \mid \text{restr}}$</td>
</tr>
<tr>
<td>Powerset</td>
<td>$\text{powerset set}$</td>
<td>$\text{powerset(set)}$</td>
</tr>
<tr>
<td>Set of all sequences</td>
<td>$\text{sequence set}$</td>
<td>$\text{sequence(set)}$</td>
</tr>
<tr>
<td>Set union</td>
<td>$\text{set-union set}_1 \cdots \text{set}_n$</td>
<td>$\text{set}_1 : \text{U}; \cdots; \text{U} : \text{set}_n$</td>
</tr>
<tr>
<td>Set difference</td>
<td>$\text{set-difference set}_1 \cdots \text{set}_n$</td>
<td>$\text{set}_1 : -; \cdots; :- : \text{set}_n$</td>
</tr>
<tr>
<td>Set intersection</td>
<td>$\text{set-intersection set}_1 \cdots \text{set}_n$</td>
<td>$\text{set}_1 : *; \cdots; * : \text{set}_n$</td>
</tr>
<tr>
<td>Cross product</td>
<td>$\text{cross set}_1 \cdots \text{set}_n$</td>
<td>$\text{set}_1 : \text{X}; \cdots; \text{X} : \text{set}_n$</td>
</tr>
</tbody>
</table>

### Table 7: Correspondence of input/output constraints

<table>
<thead>
<tr>
<th>I/O statement</th>
<th>Raw Consul</th>
<th>Front End</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input</td>
<td>(input seq &quot;filename&quot;)</td>
<td>seq = input &quot;filename&quot;</td>
</tr>
<tr>
<td>Output</td>
<td>(output seq &quot;filename&quot;)</td>
<td>seq = output &quot;filename&quot;</td>
</tr>
</tbody>
</table>
We wanted to use symbols for CONSUL constraints that were as close to standard set theory and programming languages as possible given the limitations of the computer keyboard. This explains the symbols chosen for basic arithmetic, comparisons, and subscription. Sets and subsets are enclosed in braces, reflecting standard mathematical notation. Similarly the size constraint is represented by vertical bars.

We borrowed our notation for conjunction and disjunction from Prolog (comma for "and", semi-colon for "or"). We borrowed C's exclamation point for negation. See Table 4.

The hardest constraints to find symbols for were those dealing with sets: union, intersection, set difference, et cetera. In most programming languages these operations are overloaded on arithmetic symbols, for example, "+" for union, "-" for set difference, et cetera. Unfortunately, this approach doesn't work for CONSUL: Because every value in CONSUL is formally a set, programmers can apply set constraints to the same objects to which they apply arithmetic constraints. Set constraints and arithmetic constraints may have different solutions when applied to the same objects — for example, the integers 2 and 1 are formally the sets \{0, \{0\}\} and \{0\}, respectively. The set difference of 2 and 1 is thus the set \{\{0\}\}, whereas the arithmetic difference is 1 (i.e., \{0\}). The symbols we finally adopted for set constraints are shown in Tables 2 and 6.

Note that some set constraint symbols are overloaded with set constructors. This overloading streamlines the language by providing a uniform notation for concepts for which raw CONSUL has multiple representations. Which raw CONSUL form is meant by an overloaded symbol can be determined by context.

In raw CONSUL, applying a set to one or more arguments constrains the tuple of arguments to be a member of the set. Since relations are sets, this convention is commonly used to "call" a relation. However, the same notation is used for other membership constraints. For example, if \( P \) is a set, then the raw CONSUL form \((P \ s)\) asserts that \( s \) is one of the elements of \( P \). Making membership assertions look like calls is one of the most baroque consequences of raw CONSUL's syntax. The front end provides the "IN" constraint as an alternative membership assertion. Thus, the above example could be written in CONSUL as "s IN P". Programmers usually think of calls and membership assertions as distinct things. The "IN" notation lets them clearly indicate which they mean. Furthermore, uses of "IN" are type-checked (the second argument must be a set), providing programmers an added measure of safety.

Operator precedence in CONSUL is similar to that in other languages. The default precedence for logical connectives is negation first, then conjunction, and finally disjunction. Precedence of arithmetic and set operators is summarized in Table 8. Low numbers indicate high precedence. The default precedence of any connective or operator can be overridden by parentheses.

CONSUL allows primitive constraints to be composed into elaborate expressions in a single statement. This composition eliminates the need for temporary variables that plagued raw CONSUL. The front end translates these statements into multiple raw CONSUL forms, adding the necessary temporary variables. For example, consider the geometric problem of finding the perpendicular bisector of the line segment between points \((x_0, y_0)\) and \((x_1, y_1)\). The solution is the set of points \((x_r, y_r)\) that are equidistant from \((x_0, y_0)\) and \((x_1, y_1)\).
Table 8: Precedence rules for arithmetic and set operators

<table>
<thead>
<tr>
<th>Name</th>
<th>Symbol</th>
<th>Precedence</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cross product</td>
<td>:X:</td>
<td>1</td>
</tr>
<tr>
<td>Intersection</td>
<td>:*:</td>
<td>2</td>
</tr>
<tr>
<td>Union</td>
<td>:U:</td>
<td>3</td>
</tr>
<tr>
<td>Difference</td>
<td>:−:</td>
<td>4</td>
</tr>
<tr>
<td>Multiplication, Division</td>
<td>*, \</td>
<td>5</td>
</tr>
<tr>
<td>Remainder</td>
<td>MOD</td>
<td>5</td>
</tr>
<tr>
<td>Addition, subtraction</td>
<td>+, −</td>
<td>6</td>
</tr>
</tbody>
</table>

From the formula for distance in a plane, \(x_r\) and \(y_r\) are solutions to the following CONSUL constraint:

\[
(x_1 - x_r) \times (x_1 - x_r) + (y_1 - y_r) \times (y_1 - y_r) = (x_0 - x_r) \times (x_0 - x_r) + (y_0 - y_r) \times (y_0 - y_r)
\]

The equivalent raw CONSUL code is much more verbose:

\[
\begin{align*}
\text{(exists} & \quad ((a_1 \text{ integer}) \\
& \quad (a_2 \text{ integer}) \\
& \quad (a_3 \text{ integer}) \\
& \quad (a_4 \text{ integer}) \\
& \quad (a_5 \text{ integer}) \\
& \quad (a_6 \text{ integer}) \\
& \quad (a_7 \text{ integer}) \\
& \quad (a_8 \text{ integer}) \\
& \quad (a_9 \text{ integer})) \\
\text{(and} & \quad (\text{minus } a_1 \times x_1 \times x_r) \\
& \quad (\text{minus } a_2 \times y_1 \times y_r) \\
& \quad (\text{minus } a_3 \times x_0 \times x_r) \\
& \quad (\text{minus } a_4 \times y_0 \times y_r) \\
& \quad (\text{times } a_5 \times a_1 \times a_1) \\
& \quad (\text{times } a_6 \times a_2 \times a_2) \\
& \quad (\text{times } a_7 \times a_3 \times a_3) \\
& \quad (\text{times } a_8 \times a_4 \times a_4) \\
& \quad (\text{plus } a_9 \times a_5 \times a_6) \\
& \quad (\text{plus } a_9 \times a_7 \times a_8))\end{align*}
\]

Calls on user defined relations can be embedded in composite constraints. For example, the statement

\[
d = a + f(\%, b, c)
\]
corresponds to the raw CONSUL statements

(exists
   ((t₁ integer))
   (and
      (f t₁ b c)
      (plus d a t₁)))

The idea is that that value "shared" between the called relation and its caller (t₁ in the example) is denoted by a "%", thus eliminating the need for a temporary variable. This notation was borrowed from Steele [6].

As another example of embedded calls, suppose we want to write a constraint that binds sqmax to the maximum of x₁, x₂, x₃, x₄, and x₅. We assume a relation "MAX(m,x,y)" which holds when m is the maximum of x and y. Since "maximum" is associative, we can nest calls on "MAX" to get the CONSUL constraint:

MAX(sqmax, MAX(% , MAX(%, x₃, x₂), x₁), MAX(%, x₅, x₄)).

This example demonstrates calls both with and without the "%" notation. Writing the same relation without embedded calls takes more code and temporary variables:

exists a₁, a₂, a₃: integer
   (MAX (a₁, x₃, x₂),
    MAX (a₂, a₁, x₁),
    MAX (a₃, x₅, x₄),
    MAX (sqmax, a₂, a₃))

3.2 Program Structure

We have defined a structure for CONSUL programs that makes program organization clearer than it is in raw CONSUL. A CONSUL program consists of relation definitions, type definitions, constant definitions, and the main program body. Relation, type, and constant definitions must all come before the program body, but can appear in any order relative to each other. File inclusion statements allow parts of a program to come from different files.

Relations, types, and constants are defined by the "RELATION", "TYPE", and "DEF" statements, respectively. The syntax of "RELATION" is similar to that of procedure declarations in Pascal:

RELATION name(v₁₁, ⋯ ,v₁n : T₁ ; ⋯ ; vᵢ₁, ⋯ , vᵢm : Tᵢ)
   body.
name is the name of the relation. The \( v_{ij} \) are the formal parameters, the \( T_i \) are their types. 

Body is the system of constraints that defines the relation. Type and constant definitions are written as

\[
\text{TYPE name value.}
\]
\[
\text{DEF name value.}
\]

name is the name that is defined while value is the type or constant it represents. These statements are discussed in more detail in Section 3.3.

The program body is introduced by the "MAIN" keyword:

\[
\text{MAIN}
body .
\]

File inclusion is controlled by the "INCLUDE" and "IFNDEF" statements, which are fully described in Section 3.3.

Figures 3 and 4 summarize the discussion so far by rewriting the GCD programs from Section 2 in the new CONSUL syntax. Figure 3 shows the declarative version; Figure 4 the algorithmic one. Note how much shorter these programs are than the originals, due to composition of constraints (e.g., Fig. 3 line 16; Fig. 4 lines 3, 11, and 12) and more concise forms for subsets (Fig. 3, line 5) and relation definitions (both programs, line 1). The new syntax should also be easier to understand, due to the clearer structure and notational similarity to conventional languages.

### 3.3 Extended Features

Up until this point, we have described front end features that do little more than provide better syntax for raw CONSUL functionality. There are also two areas in which the front end provides functionality that raw CONSUL does not have at all: File inclusion and static semantic checking.

File inclusion is done via the "INCLUDE" statement:

\[
\text{INCLUDE "filename"} .
\]

The effect of inclusion is the same as if the contents of "filename" had appeared in place of the "INCLUDE" statement. Note that the "MAIN" statement cannot be in an included file.

The "IFNDEF" statement makes managing multi-file programs easier. Its syntax is:

\[
\text{IFNDEF name statement .}
\]
(1) RELATION \text{dec-gcd}(gcd, b, c: \text{integer})
(2) \quad gcd > 0,
(3) \quad b \mod gcd = 0,
(4) \quad c \mod gcd = 0,
(5) \quad \exists d : \{i : \text{integer} \mid i > gcd\}
(6) \quad \mid
(7) \quad (b \mod d = 0,\ c \mod d = 0).

(9) \textbf{Main}
(10) \exists \text{in, out : sequence(integer)}
(11) \mid
(12) \quad (|\text{out}| = 1,\n(13) \quad |\text{in}| = 2,\n(14) \quad \text{in = INPUT "stdin"},\n(15) \quad \text{out = OUTPUT "stdout"},\n(16) \quad \text{dec-gcd(out[0], in[0], in[1])}).

Figure 3: CONSUL program that computes the GCD declaratively.

(1) RELATION \text{rec-gcd}(gcd, b, c: \text{integer})
(2) \quad b \mod c = 0,\ gcd = c
(3) \quad ; b \mod c \neq 0, \text{rec-gcd}(gcd, c, b \mod c).

(4) \textbf{Main}
(5) \exists \text{in, out : sequence(integer)}
(6) \mid
(7) \quad (|\text{out}| = 1,\n(8) \quad |\text{in}| = 2,\n(9) \quad \text{in = INPUT "stdin"},\n(10) \quad \text{out = OUTPUT "stdout"},\n(11) \quad (\text{in[0]} > \text{in[1]}, \text{rec-gcd(out[0], in[0], in[1])})\n(12) \quad ; \text{in[0]} \leq \text{in[1]}, \text{rec-gcd(out[0], in[1], in[0])}).

Figure 4: CONSUL program that computes the GCD using Euclid's algorithm.
CONSUL's "IFNDEF" is a simplified version of C's "#IFNDEF/#ENDIF": If name has not been defined when the "IFNDEF" is parsed, then statement is processed as if it had appeared in place of the "IFNDEF" form. If name is defined when the "IFNDEF" is parsed, then the entire "IFNDEF" statement is ignored.

Raw CONSUL has only one way to define a named constant. This mechanism exploits the fact that formally everything is a set to define constants, data types, and relations. However, it makes programs hard to read and vulnerable to inconsistent-usage bugs, because an object's definition cannot indicate how that object is to be used. The front end provides three definition forms, each indicating the intended uses of the defined object. Static checking ensures that defined objects are only used in the intended ways. The front end also detects cyclic definition errors (i.e., definitions that ultimately depend on themselves). Finally, uses of variables are checked for consistency with the variables' types.

Definitions may depend on other definitions. For example,

\[
\begin{align*}
\text{DEF } a & \ 1. \\
\text{DEF } b & \ a.
\end{align*}
\]

is a legal series of definitions. Dependent definitions may appear in any order relative to each other; the front end will sort out the dependencies as it processes the program. As mentioned above, however, cyclic dependencies are errors.

The "TYPE" and "RELATION" statements assert that the defined name is a data type or relation, respectively. Attempts to use a name defined via "TYPE" as anything other than a type, or to do anything other than call a name defined via "RELATION", will be detected as errors. 2 "DEF" is used to define arbitrary sets, whose uses are not checked by the front end. "DEF" is normally used to define constants, although it actually provides the full power of raw CONSUL definitions.

Type checking in the front end is limited to checking that the actual parameters to user-defined relations have the same types as the corresponding formals. The front end also checks that user-defined relations are called with the correct number of arguments. It would be easy to extend these checks to include the arguments to primitive constraints as well.

We have limited type checking in the front end while we consider its interactions with the formal foundations of CONSUL. In some formal sense, there is only one type for all CONSUL objects, namely "set". Many type checks are thus hard to justify formally. For example, quantifying a variable over the integer 7 is technically legal, albeit probably a programmer error. Correct handling of sub-types is another issue with which we are not yet comfortable. We believe that the present type checks could be strengthened without restricting the practical use of CONSUL, but we need to be careful in doing so.

As an example of a complete CONSUL program, Figure 5 shows a program that solves an instance of the 0-1 Knapsack problem. Specifically, given a set of integers ("{1,3,5}" on line 1), the program finds a subset of it whose members sum to "Sum" (line 2). Note

---

2As a minor exception to this rule, relations can also be passed as parameters to other relations.


Figure 5: CONSUL program that solves the 0-1 Knapsack problem

how "TYPE" is used to define "Ints", which is subsequently used as a type in variable declarations, while "DEF" is used to define the constant "Sum". The subset found is placed on the standard output (line 13). The "Total" relation (lines 3 through 7) defines what it means for a set to sum to a value, while the main program (lines 8 through 13) calls "Total" and outputs the result.

4 Implementation Issues

The front end has been integrated into a CONSUL interpreter running on Explorer™ Lisp Machines. Each of the three phases of the front end is discussed below. The discussion focuses on features unique to the front end. A great deal of standard compiler technology is also used, descriptions of which can be found in texts such as [1] [4].

4.1 Parsing

The first phase parses CONSUL programs into s-expression representations of syntax trees, using a recursive descent parser. It also expands "INCLUDE" and "IFNDEF" statements and checks for cyclic dependences between definition statements. Cyclic dependences are detected by the following algorithm:

1. For each name we have a data structure that contains the following items:
   - Dependency Number. The number of other definitions on which this one depends.
   - Dependents List. A list of names whose definitions depend on this one.
2. While parsing a definition, the parser forms a list of pending or still-undefined names on which that definition depends (the *Wait-For List*).

3. If the wait-for list is empty then
   - The newly defined name's dependency number is set to 0,
   - The dependency number of each name in this one's dependents list is decremented, and
   - For each dependent whose dependency number is now 0, recursively do step 3.

4. If the wait-for list is not empty, then
   - The new name’s dependency number is set to the length of the wait-for list, and
   - The new name is added to the dependents list of each name in the wait-for list.

5. Cyclic dependences are evident at the end of parsing: Any name that still has a non-zero dependency number is involved in a cyclic dependence and is reported to the user.

As an example of definition tracking, consider the statements:

```
DEF a b + c.
DEF c a - b.
DEF b 4.
```

The first definition processed is a's. Since a depends on b and c, which have not been defined yet, its dependency number is set to 2 and it is put on the dependents lists of b and c. The definition of c similarly depends on b and a (which are still pending). Finally b is processed. It does not depend on any other definitions, but has a and c on its dependents list. Thus the dependency numbers of a and c are decremented. However, neither dependency number goes to zero, and so a and c cannot be processed further at this stage. When parsing is finished, a and c still have non-zero dependency numbers, and so must belong to a cycle.

### 4.2 Optimization

The optimization phase performs simple optimizations of constants. Additional optimizations may be added in the future, as discussed in Section 5.1. All optimizations are performed on the syntax tree produced by the parser, prior to raw CONSUL code generation.

The first thing the optimizer does is replace uses of names defined via "DEF" and "TYPE" by their definitions. Note that a definition may refer to other names (secondary names), which are in turn replaced by their own definitions. Secondary names are evaluated in the scope that was active when the primary name was defined. "DEF" and "TYPE" could also be translated directly into raw CONSUL "define" forms, avoiding the need for explicit substitution. However, explicit substitution makes program execution faster, because it eliminates the need to look up defined names in the run time symbol table.
After expanding any uses of defined names in an expression, the optimizer performs constant folding on the result. For now, only arithmetic expressions are subject to constant folding, but in the future we may choose to fold constant sets, characters, et cetera as well.

### 4.3 Raw CONSUL Code Generation

The s-expressions produced by the parser are almost usable as raw CONSUL code. The only difference is that the parser can produce arbitrarily deep sub-trees from composite constraints. These composites have to be broken into raw CONSUL's simple primitives, accompanied by a quantifier to introduce the necessary temporary variables. The code generator is simply a procedure that collapses composite sub-trees into raw CONSUL.

Syntax trees are collapsed into raw CONSUL by a post-order traversal. The traversal takes a syntax tree and returns two values. The first (Forms) is a list of CONSUL forms that need to be included in the final raw CONSUL, the second (Temps) is a list of temporaries that have to be introduced to execute the raw CONSUL. The algorithm to traverse Tree is as follows:

1. If Tree represents an “exists”, then its body is recursively traversed. An existential quantifier is created to be the returned Forms. The variables introduced by this quantifier are the union of those from Tree and the Temps list from the recursive traversal. The body of the new quantifier is the conjunction of the recursively generated Forms. The returned Temps is empty.

2. If Tree represents a “for all”, then its body is recursively traversed. A new body (NewBody) is created as follows: If the recursively generated Temps is nonempty then NewBody is an existential quantifier whose quantified variables are Temps and whose body is the conjunction of Forms. If Temps is empty then NewBody is just the conjunction of Forms. The body of Tree is replaced by NewBody and the result is returned, accompanied by an empty list of temporaries.

3. If Tree represents a connective, then the connected statements are recursively traversed. They are then replaced in Tree by the union of the generated Forms sets. The resulting tree is returned, accompanied by the union of the recursively generated Temps.

4. If Tree represents an equality statement, then do the following:
   - If both arguments to Tree are variables or constants, then Tree can be returned as Forms, with an empty list of temporaries.
   - If exactly one argument to Tree is a variable or constant, then recursively traverse the other argument. Replace the marker (see case 6 below) in the resulting Forms with the variable or constant argument. Return the result of this replacement, accompanied by the list of temporaries from the recursive traversal.

---

3 Usually Forms contains a single raw CONSUL form that completely implements the tree; there are a few cases where Forms contains multiple forms that will serve as part of the raw CONSUL for a node at a higher level.
• If neither argument to Tree is a variable or constant, then recursively traverse both arguments. Generate a new temporary. Replace the markers in the recursively generated Forms sets with this temporary. Return the union of the resulting Forms sets, building the accompanying Temps by adding the new temporary to the union of the recursively generated sets of temporaries.

5. If Tree represents a non-equality statement, then replace its arguments as follows:

• Variables and constants need no replacement.
• For each other argument, a new temporary is generated and the argument is recursively traversed. The argument in Tree and the marker in the recursively generated Forms are both replaced by the new temporary.

Return the union of the modified Tree and any recursively generated Forms sets. The accompanying temporaries are those created above and any generated during recursive traversals.

6. If Tree represents an expression, then its raw CONSUL form is a primitive with one more argument than Tree has children. This extra argument represents the “value” of the expression. Process Tree as in step 5, adding a special marker to represent the “value” argument to the raw CONSUL primitive.

This algorithm is designed not to introduce unnecessary quantifiers or temporaries. The Temps result is a way of accumulating temporaries until either an existing existential quantifier is found in which to put them or creation of a new one is forced by a “forall”. In this way our algorithm introduces temporaries via existing quantifiers, instead of creating new ones, whenever possible. The treatment of equality constraints avoids introducing unnecessary temporaries. It also avoids redundant raw CONSUL equality constraints. It does this by using a single temporary to represent the values of the expressions on both sides of the equality. Since both values are represented by a single name, equality is implicit in the semantics of CONSUL. As an example of code generation, consider the statement:

\[ a \times b = \text{arrayA}[j] \]

This is parsed into:

\[
\text{begin}
\begin{align*}
\text{(equal} & \text{ (times a b) (elt arrayA j))}
\end{align*}
\text{end}
\]

When the code generator encounters the “equal” node, it finds that neither child is a variable or constant. It thus recursively processes both children, producing raw CONSUL forms

\[
\text{(times marker a b)}
\]

and
A temporary is then generated to replace the markers, yielding

\[(\text{times } t_1 \ a \ b)\]

and

\[(\text{elt } t_1 \ \text{arrayA} \ j)\]

Code generation for the "equal" returns these two forms as its Forms result, and the list \((t_1)\) as Temps. Code generation at higher levels of the tree will embed the two forms in an "and", and will appropriately quantify \(t_1\).

The code generator infers the types of temporaries from the context in which they will be used: Temporaries used in arithmetic constraints must be integers. The types of temporaries used in set-related constraints are derived from the types of the other arguments to the constraint. Doing this may require deducing the type of a variable or constant. The type of a variable can be looked up in the symbol table. The types of integer and character constants are "INTEGER" and "CHARACTER" respectively; the types of constant sets or sequences are derived from the union of their element types. For example, the type of the constant

\[\{1, \ 'c', \ \text{foo}\}\]

is informally "set of type \(\text{of}(1), \ \text{type}\_\text{of}(\text{of}(c)), \ \text{type}\_\text{of}(\text{foo})\)" where "type\_\text{of}(x)" denotes application of the typing algorithm to "x". In CONSUL this description is

\[
powerset\ (\ \text{type}\_\text{of}(1) : U : \ \text{type}\_\text{of}(\text{of}(c)) : U : \ \text{type}\_\text{of}(\text{foo}) )
\]

Carrying out the "type\_\text{of}" applications, and assuming that "foo\_\text{type}" is the type of "foo", gives

\[
powerset\ (\ \text{INTEGER} : U : \ \text{CHARACTER} : U : \ \text{foo\_type} )
\]

Finally, the type of a temporary introduced for a "\%" sign in a relation call is just the type of the corresponding formal parameter.

5 Extensions to the Front End

There are several areas in which the front end could be extended. Some of the extensions we are considering, and initial thoughts about their implementation, are discussed below.
5.1 Optimization

The raw CONSUL produced by the front end can be improved by optimizations analogous to those used in compilers for imperative languages. Constant folding is one example that is already done by the front end. Examples that would be useful but aren’t yet implemented include eliminating multiple occurrences of constraints (analogous to common subexpression elimination), removing from “forall” bodies any constraints that do not depend on the quantified variable (loop invariants), et cetera.

Standard optimizations appear to be useful for CONSUL, but it is not clear that standard ways of performing them can be used. The standard conditions for validity of an optimization are based on data- or control-dependences between statements (or the absence of such dependences). Standard optimization algorithms are driven by analyses of these dependences. The notion of dependence, in turn, is based on the idea that statements will be executed in some order. However, execution order is irrelevant to constraint programs, so data- and control-dependence are ill-defined for them. For example, consider the CONSUL statements

\[ x = 1, y = x + z \]

and

\[ y = x + z, x = 1 \]

Conventional copy propagation could turn the first into

\[ x = 1, y = 1 + z \]

but could do nothing with the second. For CONSUL, however, the analog of copy propagation should be applicable to both sets of statements, since their textual order does not change the fact that \( x \) must be equal to 1 when it is added to \( z \).

Another difference between optimization of constraint programs and of imperative ones is that a constraint may play different roles in a program at different times, or even multiple roles at once. For example, the constraint \( x = 1 \) might be a definition of \( x \), or it might be a test of its value. Which role this constraint plays could depend on the heuristic used to execute the program in which it appears, on what variables were given as “inputs” to a particular run of the program, et cetera. To continue the earlier example, after copy propagation turns

\[ x = 1, y = x + z \]

into

\[ x = 1, y = 1 + z \]

a conventional optimizer might eliminate \( x = 1 \) as dead code. Definitions and some tests can be safely eliminated from a constraint program this way, but some tests cannot be. For example, eliminating the apparently dead equalities to \( x \) in

\( (x = 1, y = 1 + z); (x = 2, y = 2 \times w) \)

produces a system of constraints with 2 solutions for \( y \). If other constraints determine the value of \( x \), one of these solutions will be valid for the original system and one won’t be.
To purists, the whole idea of optimizing constraint programs is meaningless, since con­
straint programs are just mathematical statements that are either true or false. Nothing
in the program itself determines how much time or memory is needed to prove truth or
falsehood. Although this attitude is too extreme for real-world constraint programming, it
does demonstrate an important point: execution efficiency depends on the combination of
a program and its execution heuristic, and both must be included in any understanding of
constraint program optimization.

The front end can produce better raw CONSUL than it does now, but several questions
have to be answered first:

- What optimizations are legal for CONSUL's semantics, and under what circum­
  stances?
- What optimizations make sense for the interpreter's execution method (or the execu­
  tion methods of future compilers)?
- How does one analyze a CONSUL program to detect applicable optimizations?

The interpreter uses local propagation [5] to solve systems of constraints. This fact provides
a starting point for answering some of these questions. Specifically, mode analysis algorithms
for logic programming languages [3] might be adapted to provide data flow information
about local propagation executing CONSUL. Well defined data flow eliminates many of the
barriers to conventional optimization. For example, data flow implies information about
the definers and users of values, so that the roles of constraints become clear; data flow
also implies an order for computations. Much work remains, however, in determining what
optimizations are compatible with CONSUL's semantics and whether mode analysis can
provide accurate enough information to support these optimizations.

5.2 Annotations

Executing or compiling a CONSUL program requires solving certain problems whose solu­
tions cannot be fully automated. Among these problems are solving the constraints in the
first place, figuring out how to parallelize the program, et cetera. It currently seems that
automatable heuristics will be able to solve these problems in many cases, perhaps even all
cases that are practically relevant. However, it is possible that some sort of programmer
assistance will eventually be needed. Annotations in CONSUL programs seem like a good
way of providing this assistance.

We envision annotations as pragmas that can be attached to statements or groups of
statements in a CONSUL program. These pragmas are hints at how a program should be
compiled or executed, but should not affect the program's solutions. For example, anno­
tations might indicate that a particular satisfaction heuristic should be used to execute a
program, that certain parts of a program are good candidates for being solved in parallel
with each other, et cetera. The meaning of a constraint program is more thoroughly sep­
arated from its execution than is the case in more imperative languages. Thus we expect
that it will be relatively easy to develop annotations for CONSUL that let programmers
control compilation and execution without altering the declarative meaning of programs.
The current CONSUL interpreter executes almost all programs without any guidance from the programmer. We hope that the same will be true of future implementations. However, we realize that this may not be possible as we try to transform CONSUL programs in more and more sophisticated ways. Annotations seem like a way of providing any guidance that turns out to be necessary without violating the language's fundamental semantics.

5.3 Higher Level Constraints

Designers of any language have to decide where the language's primitives end and where programmers have to start using those primitives to build their own abstractions. As we use the language, we may decide that flexibility or efficiency requires that the boundary change in CONSUL. There are several ways in which this could be done:

- New primitives could be added to raw CONSUL, with corresponding syntax added to CONSUL. This approach is appropriate where customized satisfiers for the new primitives are necessary.

- New features could be provided by macros in the front end. In other words, new syntax could be added to CONSUL, but the front end could handle this syntax by replacing it in-line with one or more existing raw CONSUL forms. This approach minimizes impact on CONSUL back ends, but macros will generally be solved less efficiently than built in primitives.

- User-defined relations that appear in many programs could be placed in libraries. At first libraries would be incorporated into programs via the "INCLUDE" statement. If libraries become widely used, other ways of linking them into programs may become necessary. For example, modules, ways of including only library relations that are actually called, and separate compilation into raw CONSUL would all be desirable in a sophisticated library system.

6 Conclusions

We designed the front end to correct a number of deficiencies in raw CONSUL. The front end addresses these deficiencies as follows:

- The new notation is closer to that of mathematics and other programming languages, so programmers should find it simpler to learn and use than raw CONSUL.

- Infix notation, composition, and embedded relation calls make programs more compact than their raw CONSUL equivalents. Raw CONSUL forms that are extraneous to a program's real meaning are compacted out, thus improving program readability.

- "INCLUDE" and "IFNDEF" provide a way of dividing programs into multiple files.

- Distinguishing type, constant, and relation definitions makes program structure much clearer. Structure is also clarified by distinguishing the main body from the rest of the program.
- Semantic checking allows certain programming errors to be detected in the front end instead of during program execution.

Developing the front end also identified a number of exciting topics for further work. Among these are the use of raw CONSUL as an intermediate code, the possibility of optimizing it, and the role of semantic checking in CONSUL. Many of these issues are relevant to constraint languages in general, not only to CONSUL.

CONSUL is now a much more usable language than it was. This should make the language accessible to a wider user community and enable development of larger CONSUL programs. This, in turn, will facilitate experimental tests of CONSUL as a practical, general-purpose, constraint language.
References


7 Appendix A: The grammar of the front end

The terminals of the grammar of the front end are (id), (character), (integer) and (string). An (id) is a sequence of characters that begins with a letter (a–z, A–Z), may contain digits 0-9 as well as the ASCII characters ‘.’ and ‘#’. An (integer) is a signed or unsigned sequence of digits that does not start with 0. A (string) is any sequence of ASCII characters between double quotes. Finally a character is a single printable ASCII character surrounded by single quotes. Anything surrounded by /* */ is considered a comment and is ignored.

\[
\begin{align*}
(program) & \rightarrow (driver) | \\
&\quad (statement) (program) \\
(statement) & \rightarrow (relation) | \\
&\quad (inclusion) | \\
&\quad (type) | \\
&\quad (define) | \\
&\quad (ifndef) \\
(relation) & \rightarrow RELATION (id) (arglist) (body) . \\
(inclusion) & \rightarrow INCLUDE (string) .
\end{align*}
\]
(type) → TYPE (id) (vterm) .
(define) → DEF (id) (exp) .
 ifndef → IFNDEF (id) (statement) .
(driver) → MAIN (body) .
 (arglist) → (idseq) : (exp) | (idseq) : (exp) ; (arglist)
(idseq) → (id) |
 (id) , (idseq)
(bbody) → (and_term) |
 (and_term) ; (body)
 (and_term) → (not_term) |
 (not_term) , (and_term)
 (not_term) → ! (st term) |
 (st term)
 (st term) → (quantifier) |
 (rel_stat) |
 (i/o-stat) |
 ( (body) ) |
 (set-memb) |
 (expression)
 (quantifier) → (qname) (arglist) ' ' (st term)
 (qname) → EXISTS |
 FORALL
 (set-memb) → (exp) IN (vterm)
 (rel_stat) → (id) ( (param list) )
 (param list) → (exp) , (param list) |
 (exp)
 (expression) → (exp) = (exp) |
 (exp) <= (exp) |
 (exp) >= (exp) |
 (exp) < (exp) |
 (exp) > (exp) |
 (exp) != (exp)
 (exp) → (pterm) + (exp) |
 (pterm) - (exp) |
 (pterm)
 (pterm) → (vterm) * (pterm) |
 (vterm) / (pterm) |
 (vterm) MOD (pterm) |
 (vterm)
\( (\text{vterm}) \rightarrow (\text{uterm}) : = : (\text{vterm}) \mid (\text{uterm}) \)

\( (\text{uterm}) \rightarrow (\text{iterm}) : U : (\text{uterm}) \mid (\text{iterm}) \)

\( (\text{iterm}) \rightarrow (\text{cterm}) : * : (\text{iterm}) \mid (\langle \text{exp} \rangle ) \mid (\text{cterm}) \)

\( (\text{cterm}) \rightarrow (\text{term}) : X : (\text{cterm}) \mid \text{POWERSET} ( (\text{cterm}) ) \mid (\text{term}) \)

\( (\text{term}) \rightarrow (\text{integer}) \mid (\text{character}) \mid (\text{id}) \mid (\text{sqmemb}) \mid ' | ' (\text{exp}) ' | ' \mid (\text{funccall}) \mid (\text{set}) \mid (\text{seq}) \mid (\text{subset}) \mid (\text{sequence}) \)

\( (\text{sqmemb}) \rightarrow (\text{id}) [ (\text{exp}) ] \)

\( (\text{funccall}) \rightarrow (\text{id}) ( (\text{actuals}) ) \)

\( (\text{actuals}) \rightarrow (\text{exp}) , (\text{actuals}) \mid \% (\text{actual_tail}) \)

\( (\text{actual_tail}) \rightarrow , (\text{exp}) (\text{actual_tail}) \mid \epsilon \)

\( (\text{i/o-stat}) \rightarrow (\text{id}) = (\text{io-exp}) \mid (\text{seq}) = (\text{io-exp}) \)

\( (\text{io-exp}) \rightarrow \text{INPUT} (\text{string}) \mid \text{OUTPUT} (\text{string}) \)

\( (\text{sequence}) \rightarrow \text{SEQUENCE}( (\text{vterm}) ) \)

\( (\text{subset}) \rightarrow \{ (\text{arglist}) ' | ' (\text{body}) \} \)

\( (\text{set}) \rightarrow \{ (\text{paramlist}) \} \mid \text{INTEGER} \mid \text{CHARACTER} \mid \text{EMPTY} \)

\( (\text{seq}) \rightarrow \langle\langle (\text{paramlist}) \rangle\rangle \)