SRC Technical Note
1997 - 009
June 30, 1997

Juno-2 Language Definition

Greg Nelson and Allan Heydon

digital
Systems Research Center
130 Lytton Avenue
Palo Alto, California 94301
http://www.research.digital.com/SRC/

Contents

1 Preface

Juno-2 is a constraint-based language intended for graphics applications. A Juno-2 program describes a picture; a Juno-2 implementation renders the picture. This paper describes the language only; for a description of the Juno-2 system as a whole, see "The Juno-2 Constraint-Based Drawing Editor" [HN94].

The Juno-2 language is useful for drawing pictures, and also interesting for its simplicity, uniformity, and its provisions for solving constraints. We hope this paper will be useful as a reference to Juno-2 users, and also of interest to programming language users and designers.

The theoretical basis for Juno-2 constraints rests on the following two observations: (1) as an imperative language, Juno-2's semantics are defined in terms of predicate transformers, and (2) a constraint is just another name for a predicate. For afficionados of Dijkstra's calculus of guarded commands [Dij76, Nel87], Juno-2 can be defined in one sentence: it consists of Dijkstra's predicate transformers acting on the predicates of the theory of the real numbers together with a pairing function.

For the rest of us, Juno-2 is an imperative, block-structured, untyped language with a syntax and control structure not too distant from Algol. But its data structures are more like Lisp's than Algol's: instead of arrays, records, and pointers, Juno-2 provides only scalars and ordered pairs, since this is the simplest set of data structures sufficient to represent the points and lists of points needed for the graphics primitives. The primitive graphics operations provided by Juno-2 are similar to those of PostScript [Ado90].

Any constraint expressible in Euclidean geometry (or, equivalently, in the multiplicative theory of the real numbers) can be included in a Juno-2 program. The implementation solves the constraints using numerical methods.

The provisions for constraints make the Juno-2 language novel. For example, the command

```
VAR r IN r * r = 2 -> x := r END
```

sets `x` to one of the square roots of two. The user can supply *hints* for unknowns to control the non-determinism inherent in underconstrained systems.

To support libraries of reusable graphics programs, Juno-2 also includes modules. The module system borrows the notion of a *public view* from Oberon [Wir89].

Juno-2 is an extension of Juno-1 [Nel85]. Juno-2 borrows its overall semantic framework from Juno-1, including hints. Going beyond Juno-1, Juno-2 provides a richer value space and more powerful extensibility, including user-defined constraints and constraints containing existential quantifiers.

2 Values

The universe of values of a Juno-2 program is the smallest set that includes the real numbers, the text strings, the special value `NIL`, and is closed under the formation of ordered pairs.

The language also supports closures (procedures paired with environments). Closures are represented as ordinary Juno-2 pairs according to a convention that is private to the implementation.

3 Terms

A term is an expression denoting a partial function from the state of the computation to the universe of values. Syntactically, a term is either a variable, a literal, or a function applied to arguments that are themselves terms.

4 Variables

Syntactically, a variable is an identifier, possibly qualified by a module name. Global variables are defined in modules (described below), and local variables are defined in blocks (also described below).

Juno-2 is untyped: every variable ranges over the entire universe of values.

5 Literals

Juno-2 has three kinds of literal constants: real (numeric) literals, text literals, and the special literal `NIL`.

5.1 Real Literals

A *real literal* has the form `decimal [ E exponent ]`, where `decimal` is a non-empty sequence of decimal digits, optionally containing a decimal point, and `exponent` is a non-empty sequence of decimal digits, optionally preceded by a "+" or "−". The literal denotes `decimal` times ten raised to the given `exponent`. If "`E exponent`" is omitted, `exponent` defaults to zero. Case is not significant: "`e`" is as good as "`E`". Embedded spaces are not allowed.

For example,

```
1   1.   .5   3.1415   5E1    5.0e-1
```

are legal, but

```
2.0e+a    5.0Ex
```

are not.

5.2 Text Literals

A *text literal* is a sequence of zero or more printing characters or escape sequences enclosed in double quotes. Printing characters are all printing ISO-Latin-1 characters except double-quote and backslash.

The only way to include a double-quote, backslash, or non-printing character such as newline in a text literal is with an escape sequence. Here are the legal escape sequences:

`\n`	newline
`\f`	formfeed
`\t`	tab
`\r`	carriage return
`\\`	backslash
`\"`	double quote
`\nnn`	character with octal code `nnn`

A backslash that is not a part of one of these escape sequences is a static error.

5.3 NIL

`NIL` is a distinguished value used to terminate lists. See Section 6.2 below.

6 Functions

A *pure function* is a rule that defines a result value in terms of given argument values; at most one result can be associated with any tuple of arguments. Functions may be partial; that is, it may be that for some argument values no result is defined. Pure functions never have side effects on the machine state; they are not to be confused with functional procedures (defined in Section 9.5).

What happens when you try to compute an undefined function? This is a question about commands and their computations, which we will get to later (see Sections 7.2, 8.2, and 9.5). For now, it is best to remember that a pure function is not a rule for computing a result value, but only a rule for defining it.

6.1 Standard Syntax

The standard syntax for a function application is

$\mathtt{f(t_1,\ t_2,\ \ldots,\ t_n)}$

where `f` names a function and $\mathtt{t}_1$, $\mathtt{t}_2$, ..., $\mathtt{t}_n$ are terms. Several of the built-in functions have non-standard syntax; for example, the infix arithmetic operators. The next section describes Juno-2's built-in functions.

6.2 Built-In Functions

Pair introduction. Within parentheses, the comma is a function that forms ordered pairs. That is, the term `(x,y)` is the ordered pair of `x` and `y`; it is defined for any values `x` and `y`. For example,

```
(0, (1, NIL))
("Alan", "Turing")
```

are valid terms. The inner parentheses in the first example are required. The expression `(x,y,z)` is syntactically invalid; you must instead specify either the pair `(x,(y,z))` or the pair `((x,y),z)`.

Pair elimination. The functions `CAR` and `CDR` are defined on pairs:

`CAR(p)=x`	if and only if	`p=(x,y)`	for some `y`
`CDR(p)=y`	if and only if	`p=(x,y)`	for some `x`

List introduction. Within square brackets, the comma is a function that forms lists. More precisely, the term

$\mathtt{[t_1,\ t_2,\ \ldots,\ t_n]}$

is shorthand for

$\mathtt{(t_1,\ (t_2,\ (\ldots,\ (t_n,\ NIL)\ \ldots)))}$

that is, for the list of $\mathtt{t}_i$ represented by a nest of ordered pairs as in Lisp. The list can have length one; that is, `[t]` is shorthand for `(t,NIL)`. But the list can't have length zero; that is, `[]` is not a legal shorthand for `NIL`.

Arithmetic functions. Juno-2 provides the following arithmetic functions:

`x + y`	`x` plus `y`
`x - y`	`x` minus `y`
`x * y`	`x` times `y`
`x / y`	`x` divided by `y`
`x DIV y`	`FLOOR(x/y)`
`x MOD y`	`x - (y * (x DIV y))`
`- x`	minus `x`
`FLOOR(x)`	the greatest integer not exceeding `x`
`CEILING(x)`	the smallest integer not less than `x`
`ROUND(x)`	the nearest integer to `x`
`MAX(x,y)`	the larger of `x` and `y`
`MIN(x,y)`	the smaller of `x` and `y`
`ABS(x)`	the absolute value of `x`

The infix and prefix operators are listed in groups with equal binding power, and the groups are listed in order of increasing binding power. Parentheses can be used to override the binding powers. For example, `a + - c * d` means `a + ((- c) * d)`.

Infix arithmetic operators with the same binding power are left associative. For example,

`a - b + c` means `(a - b) + c`.

The arithmetic functions are defined only on numbers; for example `NIL+0` is undefined. Also, `x/y`, `x MOD y`, and `x DIV y` are undefined if `y` is zero.

In addition, the following trigonometric and exponential functions are defined:

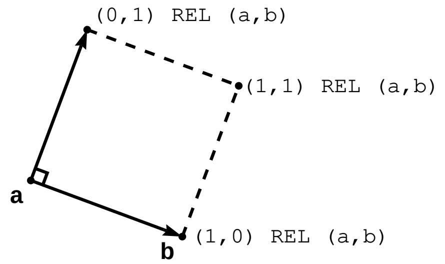
`SIN(x)`	the sine of `x`
`COS(x)`	the cosine of `x`
`ATAN(y,x)`	the angle whose tangent is `y/x`
`LN(x)`	the natural logarithm of `x`
`EXP(x)`	the exponential of `x`

All angles are in radians. The range of the `ATAN` function is the half-open interval $[-\pi, \pi)$. The sign of the result agrees with the sign of `y`. `LN(x)` is undefined if `x` is not positive.

Coordinate translation. A *point* is an ordered pair of real numbers. Since Juno-2 is intended for graphics, points and lists of points occur quite frequently. Juno-2 has a built-in infix operator for coordinate transformations on points: `p REL c`. Here `p` must be a point, say `(x,y)`, and `c` must be an ordered pair of points, say `(a,b)`. Its definition is:

`(x,y) REL (a,b)` means
the point with coordinates (`x`, `y`) in the coordinate system whose origin is at `a` and whose unit x vector ends at `b`.

Here is a figure that illustrates `REL`:



If Juno-2 had complex arithmetic, we could equivalently write the definition as follows:

`p REL (a, b) = a + p * (b - a).`

As a final example,

`(0.5, 0) REL (a, b)`

is the midpoint of the segment between `a` and `b`.

The function `p REL c` is undefined if `p` is not a point or if `c` is not a pair of points.

Text concatenation. The infix operator `&` denotes text concatenation:

`t & u` `t` concatenated with `u`

The function is undefined if either argument is not a text.

6.3 User-Defined Functions

So much for the built-in functions. You can also define new functions, as explained in Section 9.4. Defined functions can be used in terms exactly like built-in functions; they always have the standard syntax described in Section 6.1.

7 Formulas

A *formula* is a condition that is either true or false in any state of a computation. Syntactically, a formula is either a *predicate symbol* applied to one or more terms (an *atomic formula*), or a logical combination of other formulas (a *compound formula*). Like terms, formulas never have side effects.

7.1 Standard Syntax

The standard syntax for an atomic formula is

$\texttt{p}(\texttt{t}_1, \texttt{t}_2, \ldots, \texttt{t}_n)$

where `p` names a predicate and $\texttt{t}_1, \texttt{t}_2, \ldots, \texttt{t}_n$ are terms. The only atomic formulas with non-standard syntax are `TRUE`, `FALSE`, and applications of the built-in infix relations.

7.2 Formulas Are Always Defined

A pure function can be undefined, but a formula is always true or false, never undefined. This may require some getting used to. Here are four corollaries of this rule:

1. A boolean operator applied to arguments of the wrong type is false. For example,

`NIL < 3`

is false, rather than undefined.

2. An atomic formula is false if any of the terms that it contains is undefined. For example,

`1/0 = 1/0`

is false.

3. Compound formulas are defined in terms of their constituent formulas. For example, `NOT P` is the logical complement of `P`, so

```
NOT (1/0 = 1/0)
```

is true.

4. Formulas are not themselves terms. You cannot use a formula in place of a term; for example

```
(x = y) = (y = x)
```

is syntactically invalid.

7.3 Atomic Formulas

Constant Formulas.

`TRUE`	the predicate symbol that is always true
`FALSE`	the predicate symbol that is always false

Type Formulas. If `x` is any value, then

`REAL(x)`	means `x` is a real number;
`INT(x)`	means `x` is an integer;
`TEXT(x)`	means `x` is a text;
`PAIR(x)`	means `x` is an ordered pair.

In FORTRAN and Algol, integers and reals are disjoint types. But in Juno-2, as in numerical mathematics, the integers are contained within the reals. Therefore, `INT(x)` implies `REAL(x)`.

Equality. If `s` and `t` are any values, then

`s = t`	means `s` and `t` are identical;
`s # t`	means `s` and `t` are not identical.

Order. If `x` and `y` are real numbers, then

`x < y`	means `x` precedes `y`;
`x > y`	means `x` exceeds `y`;
`x <= y`	means `x` is at most `y`;
`x >= y`	means `x` is at least `y`.

These atomic formulas are false if either `x` or `y` are not numbers.

Near. The infix predicate symbol ∼, read "near", is considered to be satisfied by any pair of values, but is useful for giving hints to the Juno-2 constraint solver, as described in Section 8.9 and Appendix A.

Geometric Formulas. A *point* is a pair of real numbers. A *segment* is a pair of points, called the endpoints of the segment. If `p` and `q` are points and `s` and `t` are segments, then

`p HOR q`	means `CDR(p) = CDR(q)`;
`p VER q`	means `CAR(p) = CAR(q)`;
`s CONG t`	means `s` and `t` have the same length;
`s PARA t`	means `s` is parallel to `t`.

The atomic formulas `p HOR q` and `p VER q` are true if `p=q`. The formula `s PARA t` is true if either `s` or `t` has length zero. These formulas are false if their arguments do not have the correct type.

7.4 Compound Formulas

Propositional Connectives. If `p` and `q` are formulas, then

`p OR q`	means either `p` or `q` is true;
`p AND q`	means both `p` and `q` are true;
`NOT p`	means `p` is false.

These are listed in order of increasing binding power. Parentheses can be used to override the binding power; for example, the parentheses are necessary in

```
(x < y OR x < z) AND u < x
```

Existential Quantification. If `v` is a variable and `P` is a formula, then

```
(E v :: P)
```

is a formula. (Not any formula `P` is allowed, as we will see in Section 7.5.) In theory, this existential quantification is true if there exists some value for `v` that satisfies `P`. In practice, it is true only if Juno-2's constraint solver can find such a value. To help the solver determine a satisfying assignment for `v`, you may need to supply it with a hint for `v`'s initial value; see Appendix A.

Juno-2 provides a shorthand for introducing variables in existential quantifications with hinted or fixed values. If `t` is a term, then

`(E x = t :: P)`	means	`(E x :: x = t AND P)`
`(E x ∼ t :: P)`	means	`(E x :: x ∼ t AND P)`

The variable `x` is said to be *frozen* in the first case and *hinted* in the second case.

Furthermore, the construction

(E v_1, v_2, ..., v_n :: P)

is shorthand for

(E v_1 :: (E v_2 :: ... (E v_n :: P) ...)).

Any of the `v`'s may be frozen or hinted. However, it is a static error for the same variable to occur more than once in the list $v_1, \ldots, v_n$.

For example,

`(E x :: x*x=z)`	is equivalent to `z >= 0`;
`(E x,y :: z=(x,y))`	is equivalent to `PAIR(z)`.

7.5 Constraints

The notation `(E v :: P)` is allowed only if the formula `P` is a *constraint*. A constraint is a formula, syntactically restricted in order to make it easy to solve for its unknowns. The following functions, predicate symbols, and connectives can be used in constraints:

literals,

pair introduction: `(x, y)`,
list introduction: `[x, y, ..., z]`,
`CAR`, `CDR`,
`+`, `-`, `*`, `/`,
`SIN`, `COS`, `ATAN`, `LN`, `EXP`,
`REL`,
`TRUE`, `FALSE`,
`REAL`, `TEXT`, `PAIR`,
`=`, `~`,
`HOR`, `VER`, `CONG`, `PARA`,
`AND`,
existential quantification,
user-defined pure functions,
user-defined predicates.

The following are forbidden in constraints:

`DIV`, `MOD`,
`FLOOR`, `CEILING`, `ROUND`,
`MAX`, `MIN`, `ABS`,
`INT`,
`#`, `<`, `<=`, `>`, `>=`, `&`,
`OR`, `NOT`,
functional procedures.

Functional procedures, defined in Section 9.5 below, are not to be confused with pure functions.

For example,

```
(E z :: z * z = 2)
```

is a legal formula, since `*` and `=` are allowed in constraints. But the formulas

```
(E z :: INT(z) AND z + z = 4)
(E z :: (z & "bc" & z) = "abcbcabc")
```

are illegal, since neither `INT` nor `&` is allowed in a constraint.

There is one exceptional case: `OR` is allowed in a constraint if one of its arguments is the predicate symbol `TRUE` and its other argument is a constraint. This is a degenerate case, since `TRUE OR P` is equivalent to `TRUE`.

7.6 User-Defined Predicate Symbols

So much for the built-in formulas. You can also define new predicate symbols, as explained in Section 9.3. Defined predicate symbols can be used in atomic formulas exactly like built-in predicate symbols. They always have the standard formula syntax described in Section 7.1.

8 Commands

Executing a command produces a sequence of state transitions, the eventual outcome of which is to loop forever, cause a checked runtime error, or halt in some state. Since Juno-2's constraint solver is non-deterministic, a Juno-2 command may be non-deterministic as well. That is, the outcome of a command need not be functionally determined by the initial state.

Associated with each command is a predicate on the initial state called its *guard*. A command can be activated only in a state where its guard is true. An attempt to activate a command in a state where its guard is false is said to *fail*.

Failure is not to be confused with abortion of the computation due to a run-time error. A computation that aborts may change the state before it aborts, but an attempt to activate a command where its guard is false fails immediately, without changing the state. To implement these semantics without backtracking, the Juno-2 grammar syntactically restricts the placement of commands that may potentially fail.

We write "`grd(S)`" to denote the guard of the command `S`. If `grd(S)=TRUE`, then `S` can be activated in any state, and we say that `S` is *total*. Otherwise, `S` is *partial*.

The remainder of this section lists the Juno-2 commands.

8.1 Abort, Skip

The command `ABORT` causes a checked runtime error. It is total.

The command `SKIP` is a no-op. It is also total.

8.2 Assignment

The assignment command has the form

$$\mathtt{v}_1,\ \mathtt{v}_2,\ \ldots,\ \mathtt{v}_n\ :=\ \mathtt{t}_1,\ \mathtt{t}_2,\ \ldots,\ \mathtt{t}_n$$

where $\mathtt{v}_1, \mathtt{v}_2, \ldots, \mathtt{v}_n$ are distinct variables and $\mathtt{t}_1, \mathtt{t}_2, \ldots, \mathtt{t}_n$ are terms. The effect is to evaluate all the terms and then set each variable to the value of the corresponding term. For example,

```
x, y := y, x
```

swaps `x` and `y`.

The assignment command is total. If any term $\mathtt{t}_i$ is undefined, the assignment is equivalent to `ABORT`.

8.3 Guard

If `P` is a formula and `S` is a command, the guarded command

```
P -> S
```

fails if P is false, and otherwise is the same as S. We have

```
grd(P -> S) = (P AND grd(S)).
```

8.4 Sequential Composition

If S is a command and T is a total command, then

```
S ; T
```

means: execute S, then execute T. The guard of "S ; T" is the guard of S. Notice that since T is total, there is no danger that T will fail after S has changed the state.

The guard arrow has weaker binding power than the semicolon; thus in

```
x < y -> max := y; min := x,
```

the guard covers both assignments.

8.5 Else Command

If S and S′ are commands, the command

```
S | S′
```

(read "S else S′") executes S if its guard is true, otherwise it executes S′. It fails only if both S and S′ fail:

```
grd(S | S′) = grd(S) OR grd(S′).
```

The else operator has weaker binding power than the guard arrow and semicolon.

8.6 Command Grouping

Curly braces can be used to override the binding power of the operators on commands. For example,

```
P -> A ; B | C
```

is equivalent to

```
{ P -> { A ; B } } | C
```

which is different from

```
P -> A ; { B | C }.
```

8.7 Iteration

If S is a command, then the total command

```
DO S OD
```

repeatedly executes S until its guard becomes false. That is, this command is equivalent to

```
{ S ; DO S OD } | SKIP.
```

For example,

```
DO n < m -> m := m - n
 | m < n -> n := n - m
OD
```

will compute the greatest common divisor of the two non-negative integers n and m.

8.8 Alternative Construct

If S is a command, then the total command

```
IF S FI
```

is equivalent to

```
S | ABORT.
```

For example,

```
IF x >= y -> m := x
 | y >= x -> m := y
FI
```

is equivalent to m := MAX(x, y). In particular, both commands abort if either x or y are not numbers: the IF command aborts because both guards are false; the assignment aborts because MAX is defined only on numbers.

8.9 Projection

If v is a variable and S is a command such that grd(S) is a constraint, then the command

```
VAR v IN S END
```

introduces v as a local and executes S. The scope of v starts at IN and ends at END. Its initial value is chosen so as to make the guard of S true; if the constraint solver cannot find such a value, the command fails. We have:

```
grd(VAR v IN S END) = (E v :: grd(S)).
```

For example, the command

```
VAR r IN
  r * r = 2 -> x := r
END
```

will set x to one of the square roots of two. If you care which square root you get, you can use the near predicate to give a hint to the solver:

```
VAR r IN
  r ~ 1 AND r * r = 2 -> x := r
END.
```

This has the same formal semantics as the previous version, but in practice will set r to the positive square root of two.

Juno-2 solves constraints by numerical methods; the near predicate provides the initial value for the numerical iteration. The hint you supply can be an arbitrary expression, and you can supply a different hint for each variable. For more information on the near predicate and Juno-2's constraint solver, see Appendix A.

Similar to the syntax for existential quantification, Juno-2 provides a shorthand for introducing variables in projections with hinted or fixed values. If `t` is a term, then

```
VAR x = t IN S END  means
  VAR x IN x = t -> S END
VAR x ~ t IN S END  means
  VAR x IN x ~ t -> S END
```

Furthermore, the command

```
VAR v1, v2, ..., vn IN S END
```

is shorthand for

```
VAR v1 IN
  ...
    VAR vn IN
      S
    END
  ...
END.
```

As before, any of the `v`'s may be frozen or hinted, and it is a static error for the same variable to occur more than once in the list $v_1, \ldots, v_n$.

By definition, we have

```
grd(VAR v1, v2, ..., vn IN S END) =
  (E v1, v2, ..., vn :: grd(S)).
```

As another example, here is a program that reverses the list `p`, using an auxiliary variable `q`:

```
q := NIL;
DO
  VAR u, v IN
    p = (u, v) ->
      q := (u, q);
      p := v
  END
OD;
p := q
```

As a final example,

```
VAR a, b IN
  INT(a) AND INT(b)
  AND a * b = 89801 ->
    x, y := a, b
END
```

is illegal, since `INT` is not allowed in constraints.

8.10 Save Command

If `M` is the name of a module and `T` is a total command, the total command

```
SAVE M IN T END
```

is shorthand for

```
M.Save(); T; M.Restore().
```

Notice that since `T` follows a semi-colon in this translation, it must be total.

This is useful for temporarily changing various aspects of the program state, as illustrated in the built-in PostScript module `PS`.

8.11 Procedure Call

A procedure call is a total command. See Section 9.5 for the syntax and semantics of a procedure call.

9 Definitions

You can extend Juno-2 by defining your own constants, global variables, predicates, pure functions, and procedures. These definitions are grouped into modules, as described in Section 11.

Each definition is either public or private; private definitions are preceeded by the optional keyword `PRIVATE`. Private definitions are hidden from clients, and they are elided in public views of the module.

Constants, global variables, and procedures are imperative programming constructs, so except for the initialization order of constants and global variables, their definition order within a module does not matter. Predicate and pure function definitions, on the other hand, express relations in first-order logic (where recursive definitions are disallowed), so there are restrictions on their definition order.

9.1 Constants

If $id_1, \ldots, id_n$ are names and $t_1, \ldots, t_n$ are terms, the definition

```
CONST id1 = t1, ..., idn = tn;
```

defines each id_i as a read-only name for the value of the term t_i. Here are some examples:

```
CONST Pi = 3.14159, Zero = SIN(Pi);
CONST Data = [("Jan", 2.3), ("Feb", 5)];
CONST Len = Text.Length("Juno-2");
```

It is an unchecked error for any t_i to depend on the initial value of a constant or global variable declared at or after id_i in the same module. For example, the definition

```
CONST x = 3 * x - 2;
```

is an unchecked error because it is self-referential.

9.2 Global Variables

If $v_1, \ldots, v_n$ are names and $t_1, \ldots, t_n$ are terms, the definition

```
VAR v1 [:= t1], ..., vn [:= tn];
```

introduces $v_1, \ldots, v_n$ as global variables with initial values $t_1, \ldots, t_n$. If any t_i is omitted, the corresponding v_i's initial value is arbitrary. It is an unchecked error for any t_i to depend on the initial value of a constant or global variable declared at or after v_i in the same module.

9.3 Predicates

A predicate definition has the form

```
PRED id(args) IS C END;
```

where `C` is a constraint and `args` is a list of distinct identifiers. It defines `id(args)` to be equivalent to `C`. A defined predicate can be used in a formula just like a built-in predicate.

Here are some example predicate definitions:

```
PRED R3(p) IS
  (E x,y,z ::
    p = [x, y, z] AND REAL(x) AND
    REAL(y) AND REAL(z))
END;

PRED Colinear(a, b, c) IS
  (a, b) PARA (a, c)
END;
```

In a predicate definition, the constraint `C` may not refer to any procedures, nor to any constants, global variables, predicates, or pure functions defined at or after `id` in the same module.

9.4 Pure Functions

A pure function definition has the form

```
FUNC res = id(args) IS C END;
```

where `res` and `id` are identifiers, `args` is a list of identifiers, there are no duplicates among the identifiers `res` and `args`, and `C` is a constraint. The effect is to define `id` to be the function with the property that `res = id(args)` is logically equivalent to `C`. A defined pure function can be used in a term just like a built-in function.

Here are some example pure function definitions:

```
FUNC n = Half(m) IS m = n + n END;

FUNC y = Cadr(l) IS
  (E x, tail :: l = (x, (y, tail)))
END;
```

In a pure function definition, the constraint `C` may not refer to any procedures, nor to any constants, global variables, predicates, or pure functions defined at or after `id` in the same module.

If a pure function has more than one possible result for some input, the result computed for that function is nondeterministic. For example,

```
FUNC y = Sqrt(x) IS y * y = x END;
```

is unadvised, since `y` is not determined uniquely from `x`. However, such a function can be used reliably if the functional result is given a suitable hint — for example:

```
a ~ 1 AND a = Sqrt(2).
```

It is perfectly all right for a pure function to be partial; both `Half` and `Cadr` above are partial, since `Half` is defined only on numerical arguments, and `Cadr` is defined only on lists with at least two elements.

9.5 Procedures

A procedure definition has the form

```
PROC [outs :=] [(inouts):] id(ins) IS
  S
END;
```

where `outs`, `inouts`, and `ins` are lists of identifiers (collectively called the *formal parameters* of the procedure), `id` is an identifier (the name of the procedure) and `S` is a total command (the body of the procedure). The "`:=`" must be omitted if `outs` is empty, and the initial "`()`" and "`:`" must be omitted if `inouts` is empty; however, the second "`()`" are required even if `ins` is empty. Additionally, the initial "`()`" may be omitted if `inouts` contains exactly one identifier.

Procedure Call. A call to the procedure has the form

```
[outActs :=] [(inoutActs):] id(inActs)
```

where `outActs` and `inoutActs` are lists of variables and `inActs` is a list of terms (collectively called the *actual parameters* of the call). The "`:=`" must be omitted if `outActs` is empty, and the initial "`()`" and "`:`" must be omitted if `inoutActs` is empty; however, the latter "`()`" are required even if `inActs` is empty. Additionally, the initial

"()" may be omitted if inoutActs contains exactly one variable.

The meaning of the call is the same as the following block

```
VAR outs, inouts, ins IN
  inouts, ins := inoutActs, inActs;
  S;
  outActs, inoutActs := outs, inouts
END
```

Any formal variable that occurs in any actual must be renamed before applying this rule. (Since the formals are dummies, renaming them consistently does not affect the meaning of the procedure definition.) Note that by definition of the assignment command (see Section 8.2), the call is equivalent to ABORT if any of the inActs is undefined.

For example, after the definitions

```
PROC st:Push(x) IS st := (x, st) END;

PROC x := st:Pop() IS
  x, st := CAR(st), CDR(st)
END;
```

the command

```
st := NIL;
st:Push(1); st:Push(2);
x := st:Pop()
```

would leave st = [1] and x = 2.

Functional Procedures. A procedure is *functional* if it has exactly one out parameter, like the Pop procedure above. Such a procedure can be used as a function, with the obvious meaning. For example, here is a recursive definition of a procedure for appending two lists:

```
PROC res := Append(y, z) IS
    y = NIL -> res := z
  | res := (CAR(y), Append(CDR(y), z))
END;
```

Juno-2's framework is geared toward pure functions, but we include functional procedures for their notational convenience. However, they should be used with care. Functional procedures cannot be used within constraints. Functional procedures may have side-effects, but it is dangerous to use such procedures functionally, since Juno-2 may evaluate the subexpressions of an expression in any order. Finally, note that a functional procedure whose body aborts is semantically different from an undefined pure function.

10 Closures

Flexible drawing libraries often require user-supplied procedures as parameters. For example, a scatter-plot package can accept a user-supplied procedure for drawing the points in the plot, or an animation package can accept a procedure for drawing each frame of an animation. For this reason, the Juno-2 language supports closures.

A *closure* is a procedure (called the *body* of the closure) paired with a list of values (called the *environment* of the closure).

Any built-in or user-defined procedure p is considered a closure with body p and an empty environment.

10.1 Closure Introduction

The built-in functional procedure CLOSE creates closures with non-empty environments. The command

```
CLOSE(cl, t1, ..., tn)
```

evaluates to a closure whose body is the body of the closure cl and whose environment is cl's enviroment extended at the right by the list of values $t_1, \ldots, t_n$.

10.2 Closure Application

The built-in procedure APPLY applies a closure. The effect of the command

```
[outs]:=[(inouts):] APPLY(cl, t1, ..., tn)
```

is to apply the body of the closure cl to the given out parameters, the given inout parameters, and the in parameters formed by concatenating the list of values in cl's environment with the values of the actual arguments $t_1, \ldots, t_n$. The out and inout arguments are optional, just as for an ordinary procedure call.

For example,

```
VAR
  cl := CLOSE(Text.Cat, "Hello, ");
  nm := APPLY(cl, "Juno!");
```

initializes nm to "Hello, Juno!".

The environment of a closure provides values only for the in parameters of its body, not for any out or inout parameters. Actual out and inout parameters can be supplied when the closure is applied.

Argument counts are checked dynamically at APPLY time. For out and inout parameters, the number of actuals must equal the number of formals of the closure's body. For in parameters, the number of values in the closure environment together with the number of actuals must sum to the number of the body's formal in parameters.

11 Modules

Larger Juno-2 programs are organized into *modules*, which are collections of declarations. The module `A` must *import* the module `B` to refer to the names declared in `B`. The qualified identifier `B.N` is used within `A` to refer to the entity in `B` named `N`.

Within a module, each name can be declared either *public* or *private*. Only public names are visible to importers.

11.1 Import

The `IMPORT` statement has the following form:

```
IMPORT M1, ..., Mn;
```

where the M_i are module names.

`IMPORT` makes the names of the modules M_1, ..., M_n visible. To refer to an entity named `N` in any M_i, the importer must use the qualified identifier M_i`.N`. Importing a module provides access to the names it declares, but not to those it imports. The import relation must be acyclic.

11.2 Modules

A module has the form:

```
MODULE id ";" { Imports } { Defns }
```

where `id` is an identifier that names the module, `{Imports}` is a sequence of import statements, and `{Defns}` is a sequence of definitions (or comments).

All defined names in the module `id` are visible without qualification in the module. The names defined in the module and the visible imported names must be distinct.

11.3 Comments

Comments may appear in a module wherever a `MODULE` header, `IMPORT` statement, or definition is legal. Comments can appear at top-level only; this is to simplify the pretty-printer. Comments may be nested. Public and private comments have the following forms:

```
(* public comment *)
/* private comment */
```

Private comments are elided from public module views.

A Constraint Solving

In principal, the semantics of Juno-2 depend only on the semantics constaints, and not on the implementation of the constraint solver. But in practice, if you are using Juno-2 you will sometimes need to know something about how the solver works. This section is intended to provide some helpful information.

There are three steps in solving a system of constraints. The solver (1) propagates known values, (2) propagates hints, and (3) uses the numeric solver.

The numeric solver uses Newton-Raphson iteration. This is quite reliable (even for the consistent but over-constrained case) *provided that adequate hints are given*. But if any variable involved in a non-linear constraint is unhinted, the solver is unlikely to succeed. Thus, if the solver isn't working properly there is probably a problem with the initial hints or with their propagation.

Initial hints. You specify the initial hints with the *near* constraint. If `s` and `t` are terms, then

```
s ~ t
```

is semantically `TRUE`, but it supplies a hint to the solverr for one or more unknowns.

Propagating known values. The solver's first step uses equalities to eliminate unknowns by propagating known values. That is, if `x=E` or `E=x` is a constraint, where `x` is an unknown and `E` is an expression all of whose variables have known values, then `x` is reclassified as known instead of unknown, and given the known value of `E`. For example, in

```
VAR x,y,z IN
  x=5 AND x=y AND y*y=z -> ...
END
```

the known value `5` for `x` is propagated to `y` and then the known value `25` of `y*y` is propagated to `z`. In this simple case, propagation of known values has solved all the constraints, but in general, some unknowns and constraints will remain.

In propagating known values, the solver does not use algebraic facts about arithmetic. For example, from `x=2 AND x+y=4` it is unable to propagate the known value `2` to `y`. Therefore the propogation phase leaves `y` as an unknown, and the solver uses the numeric solver on the system `2+y=4` with unknown `y`.

Propagating hints. The solver's second step uses equalities and near constraints to propagate hints to as many unknowns as possible. That is, if `x=E` or `E=x` or `x~E` or `E~x` is a constraint, where `x` is an unhinted unknown and `E` is an expression all of whose variables either have known values or hints, then `x` is given as a hint the value of `E`, where the unknowns in `E` are considered to have their hinted values. For example, in

```
VAR x,y,z IN
  x=1 AND x~y AND
  y+0.5=z AND z*z=2 -> ...
END
```

the known value 1 for x is propagated to become the hinted value for y, and then this hint is used to give the hint 1.5 to z. Thus the numeric solver is invoked on the equation z*z=2 with the initial value z=1.5.

In propagating hints, the solver does not use algebraic facts about arithmetic. But hints for ordered pairs are used to produce hints for their components. For example, consider the command

```
VAR p IN
  p ~ (0.5, 0) REL ((0, 0), (2, 2)) AND
  CAR(p) * CAR(p) = 2 -> ...
END
```

The solver will use the value of the REL expression (which is (1,1)) as a hint for p. This will cause the solver to use 1 as the hint for CAR(p). Therefore, the Newton-Raphson solver will solve x*x=2 with 1 as the initial value of x, producing a final solution of p = (1.414, 1).

Errors in hints. If the propagation of hints terminates without making use of one of the user's near constraints, the Juno compiler gives the error message *unused near constraint* and refuses to compile the program. The alternative would be to invoke the numeric solver from an initial value that ignored one or more of the user's near constraints, which would be likely to be fail in some confusing way.

Two common problems lead to the *unused near constraint* error. One of them is to accidentally produce a cycle of hints, as in

```
VAR x,y IN
  x ~ (y,5) AND y ~ CAR(x) -> ...
END
```

The second is to write near constraints that Juno can't use because it doesn't use algebraic identities to propagate NEAR constraints, as for example in:

```
VAR x, y IN
  x=2 AND y+x~5 AND y*y=10 -> ...
END
```

Such systems can be solved by rewriting the NEAR constraints so the unknowns appear alone on one side of the ~, like this:

```
VAR x, y IN
  x=2 AND y~5-x AND y*y=10 -> ...
END
```

B Syntax

In this appendix, we describe the Juno-2 syntax: its operators, keywords, reserved words, grammar, and tokens.

B.1 Operators and Keywords

Here are Juno-2's operators:

```
; .  , : ( ) { } [ ] := :: |
-> ~ = # < > >= <= + - * / &
```

Here are Juno-2's keywords:

```
MODULE IMPORT PRIVATE CONST VAR PRED FUNC PROC
IS SKIP ABORT IF FI DO OD SAVE IN END NIL TRUE
FALSE OR AND NOT CONG PARA HOR VER E REL DIV MOD
```

Here are Juno-2's reserved identifiers, which cannot be redefined:

```
FLOOR CEILING ROUND MAX MIN ABS SIN COS ATAN LN
EXP CAR CDR REAL TEXT PAIR INT CLOSE APPLY
```

Identifiers beginning with the underscore character are also reserved.

B.2 EBNF Grammar

The following grammar uses an extended BNF notation in which `{S}` is a sequence of zero or more occurrences of `S`, and `[S]` is zero or one occurrence of `S`. Tokens are written as all upper-case identifiers, as double-quoted strings, or surrounded by angle brackets. Each production list ends with a period.

B.2.1 Compilation Unit Productions

```
Module          = [ ModDecl ] { Import } { Decl }.
ModDecl         = MODULE <Id> ";".
Import          = IMPORT IdList ";".
Decl            = [ PRIVATE ] Declaration ";".
```

Comments may appear anywhere at the top level of a module.

B.2.2 Declaration Productions

```
Declaration     = CONST ConstDecls
                | VAR VarDecls
                | PRED PredDecl
                | FUNC FuncDecl
                | PROC ProcDecl.
ConstDecls      = ConstDecl { "," ConstDecl }.
ConstDecl       = <Id> "=" Expr.
VarDecls        = VarDecl { "," VarDecl }.
VarDecl         = <Id> [ ":=" Expr ].
PredDecl        = <Id> "(" IdList ")" IS Constraint END.
FuncDecl        = <Id> "=" <Id> "(" IdList ")" IS Constraint END.
ProcDecl        = ProcHead IS TotalCmd END.
ProcHead        = [ IdList ":=" ] [ IdList2 ":" ] <Id> "(" [ IdList ] ")".
```

B.2.3 Command Productions

```
TotalCmd         = Cmd.
Cmd              = SKIP | ABORT
                 | QIdList ":=" ExprList
                 | Cmd ";" TotalCmd
                 | Formula "->" Cmd
                 | Cmd "|" Cmd
                 | DO Cmd OD
                 | IF Cmd FI
                 | VAR NearVarList IN Cmd END
                 | SAVE <Id> IN TotalCmd END
                 | ProcedureCall
                 | "{" Cmd "}".
ProcedureCall    = [ QIdList ":=" ] [ QIdList2 ":" ] QId "(" [ExprList] ")".
```

The infix command operators (namely, "`:=`", "`;`", "`->`", and "`|`") are listed in *decreasing* order of binding power.

The grammar defines a `TotalCmd` simply as a `Cmd`, but there are syntactic restrictions on `TotalCmds`. The commands `SKIP`, `ABORT`, assignment, `DO ... OD`, `IF ... FI`, `SAVE ... IN ... END`, and procedure call are always `TotalCmds`. In addition, the commands `T;A`, `A|T`, `{ T }`, and `VAR NearVarList IN T END` are each `TotalCmds` when the command `T` is a `TotalCmd` and none of the variables in `NearVarList` has a hint. Any other command is considered partial, even if its guard is semantically `TRUE`.

B.2.4 Formula Productions

```
Constraint       = Formula.
Formula          = AndFormula [ OR Formula ].
AndFormula       = NotFormula [ AND AndFormula ].
NotFormula       = SimpleFormula | NOT NotFormula.
SimpleFormula    = TRUE | FALSE
                 | "(" Formula ")"
                 | RelationFormula
                 | PredicateUse
                 | SolveFormula.
RelationFormula  = Expr RelationOp Expr.
RelationOp       = "~" | "=" | "#" | "<" | ">" | "<=" | ">="
                 | CONG | PARA | HOR | VER.
PredicateUse     = QId "(" [ ExprList ] ")".
SolveFormula     = "(" E NearVarList "::" Constraint ")".
```

The grammar defines a `Constraint` simply as a `Formula`, but `Constraints` must conform to the restrictions listed in Section 7.5.

B.2.5 Expression Productions

```
Expr             = Expr1 [ REL Expr1 ].
Expr1            = Expr2 [ AddOp Expr1 ].
Expr2            = Expr3 [ MultOp Expr2 ].
Expr3            = Expr4 | "-" Expr3.
Expr4            = QId
                 | NIL
                 | Literal
                 | "(" Expr ")"
                 | "(" Expr "," Expr ")"
```

```
                | "[" ExprList "]"
                | FunctionCall.
AddOp           = "+" | "-" | "&".
MultOp          = "*" | "/" | DIV | MOD.
Literal         = <Number> | <TextLiteral>.
FunctionCall    = [ QIdList2 ":" ] QId "(" [ ExprList ] ")".
```

B.2.6 Miscellaneous Productions

```
IdList          =    <Id> { "," <Id> }.
IdList2         =    <Id> | "(" IdList ")".
QId             =    <Id> [ "." <Id> ].
QIdList         =     QId { "," QId }.
QIdList2        =     QId | "(" QIdList ")".
ExprList        =    Expr { "," Expr }.
NearVarList     = NearVar { "," NearVar }.
NearVar         =    <Id> [ ("~" | "=") Expr ].
```

B.3 EBNF Syntax For Tokens

To read a token, first skip all blanks, tabs, newlines, carriage returns, vertical tabs, and form feeds. Then read the longest sequence of characters that forms an `Operator`, `Id`, `Number`, `TextLiteral`, or comment. The syntax of comments is described in Section 11.3.

An Id is a case-significant sequence of letters, digits, and underscores that begins with a letter. An Id is a keyword or reserved identifier if it appears in the corresponding list above, and an ordinary identifier otherwise.

In the following grammar, terminals are characters surrounded by double-quotes and the special terminal DQUOTE represents double-quote itself.

```
Operator        = ";"  | "::" | "." | ":=" | ":" | "(" | ")" | "[" | "]"
                | "{"  | "}"  | "|" | "->" | "=" | "#" | "<" | ">" | ">="
                | "<=" | ","  | "+" | "-"  | "*" | "/" | "&" | "~".

Id              = ( Letter | "_" ) { Letter | Digit | "_" }.

Number          = FloVal [ Exponent ]
FloVal          = { Digit } ( Digit | Digit "." | "." Digit ) { Digit }
Exponent        = ( "E" | "e" ) [ "+" | "-" ] Digit { Digit }.

TextLiteral     = DQUOTE { PrintingChar | Escape } DQUOTE.
Escape          = "\" "n" | "\" "t" | "\" "r" | "\" "f" | "\" "\"
                | "\" DQUOTE | "\" OctalDigit OctalDigit OctalDigit.
PrintingChar    = Letter | Digit | OtherChar.

OctalDigit      = "0" | "1" | ... | "7".
Digit           = OctalDigit | "8" | "9".
Letter          = "A"  | "B"  | ... | "Z"  | "a"  | "b"  | ... | "z".
OtherChar       = " " | "!" | "#" | "$" | "%" | "&" | "'" | "(" | ")" | "*"
                | "+" | "," | "-" | "." | "/" | ":" | ";" | "<" | "=" | ">"
                | "?" | "@" | "[" | "]" | "^" | "_" | "`" | "{" | "|" | "}"
                | "~" | ExtendedChar.

ExtendedChar    = any char with ISO-Latin-1 code in [8_240..8_377].
```

References

[Ado90] Adobe Systems Incorporated. *PostScript Language Reference Manual*. Addison Wesley, second edition, 1990.

[Dij76] Edsger W. Dijkstra. *A Discipline of Programming*. Series in Automatic Computation. Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1976.

[HN94] Allan Heydon and Greg Nelson. The Juno-2 Constraint-Based Drawing Editor. Research Report 131a, Digital Systems Research Center, 130 Lytton Avenue, Palo Alto, CA 94301, December 1994.

[Nel85] Greg Nelson. Juno, a Constraint-Based Graphics System. *ACM Proceedings on Computer Graphics (SIGGRAPH)*, 19(3):235–243, July 1985. San Francisco.

[Nel87] Greg Nelson. A Generalization of Dijkstra's Calculus. Technical Report 16, Digital Systems Research Center, 130 Lytton Avenue, Palo Alto, CA 94301, April 1987.

[Wir89] Niklaus Wirth. From modula to oberon. Technical report, ETH Zürich, September 1989.