=> Introduction

Imagine that you are in a Space Sciences class, and your professor tries to convince you that the statment, "If there is life on Mars, it must be very primitive" is true. He might make a statement like, "Suppose there is life on Mars," and go on to talk about the environment it would have to endure, its possible effects on the planet, and so forth. For the next few minutes he would go on talking as if a recent space probe had discovered life, and convince you that the life must be very primitive. But when the class was over, you wouldn't rush over to your friends and ask them if they had heard that life was discovered on Mars. You accepted the assumption for the sake of discussion, and when the discussion was over, the assumption was dropped.

Here is the template for the => introduction rule:

```
P => Q BY INTRO,
PROOF;
    [ASSUME P;]
    .
    .
    .
    Q ...
QED;
```

Brackets ([]) in a template indicate an optional part. When you are using the => introduction template, you don't have to write out the ASSUME P part if you don't want to. As before, the vertical dots indicate where proof lines go, and the horizontal dots stand for any kind of justification.

The hypothesis to the => introduction rule is a little different from others we have seen. Instead of certain assertions, it requires a whole proof. This proof has a special part called the assumption. Any proof lines you write between the PROOF and the QED can use P as a hypothesis. If, by assuming P, you can write out a proof of Q, then P => Q must also be true. Each line in the proof can get its hypotheses from any previous proof line, or from the ASSUME line (even if you decided not to write it). The proof can't use P => Q as a hypothesis though; that hasn't been proved yet. Here is an example of the use of the => introduction rule:

```
I<=J => I+1<=J+1 BY INTRO,
PROOF;
    ASSUME I<=J;
    1<=1 BY ARITH;
    I+1 <= J+1 BY ARITH, I<=J, +, 1<=1;
QED;
```

Proof lines written after the QED can get their hypotheses from any lines written before the PROOF (including the P => Q line) or after the QED. However, a line that comes after the QED cannot reach inside, between the PROOF and QED for a hypothesis. That would be like after Space Science class telling your friends there is life on Mars.

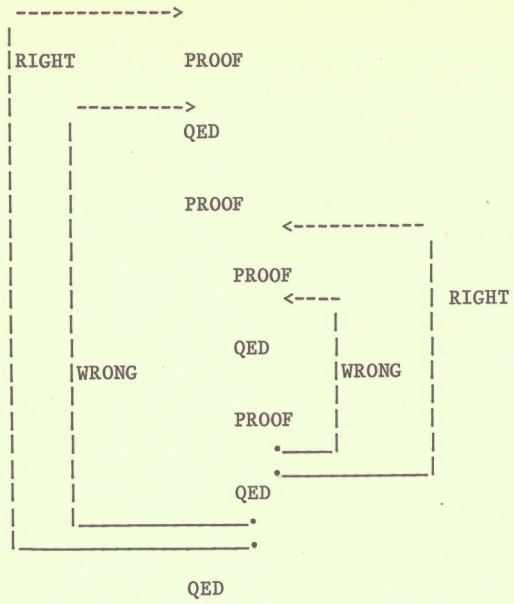

Figure 3 Hypotheses and Proof Blocks

The reason we write the hypothesis-proof for the => introduction rule after the line we are trying to prove is for ease in reading. Once we see that the prover wishes to show an implication, we can read the following lines keeping in mind the assumption being made (P) and the conclusion we are trying to reach (Q).

Any kind of proof whatsoever can be written where the vertical dots are in the template, even a proof that uses the => introduction rule. Here is an example of such a use.

```
I<=J => (J<=I => I=J) BY INTRO,
PROOF;
    ASSUME I<=J;
    J<=I => I=J BY INTRO,
    PROOF;
        ASSUME J<=I;
        I=J BY ARITH, I<=J, J<=I;
    QED;
QED;
```

The part of the proof between a PROOF and its corresponding QED is called a "proof" block. The above proof blocks are said to be "nested," because one is inside the other. The rule for reaching back and getting hypotheses in the presence of nested proof blocks is that you can always reach around a proof block, but you can never reach into one. Figure 3 shows examples of

right and wrong ways of reaching back for hypotheses. If a proof line is allowed to reach back to a previous proof line for a hypothesis, the assertion on the previous line is said to be "accessible" to the later line. Later we will see more restrictions on which assertions are accessible to proof lines.

The notion of assumptions and proof blocks alters our idea of the meaning of assertions in the proofs we write. For assertions at "top level" (i.e. not within any proof block) the meaning is unchanged from that described in section I. But in general we must say of an assertion in our proofs that if all the assumptions accessible to it are true, then its meaning as described in section I is true. (Of course, top level assertions have no assumptions.) The meaning of an assertion with free variables is that any assignment of values to those free variables that makes all accessible assumptions true will also make the assertion itself true.

The Cases Rule

The | elimination rule is also called the CASES rule. Here is the idea behind the CASES rule. Suppose you know that Q or R is true, though you don't know which one, and you want to show that P follows. If you first assume Q, and show that P follows, and then assume R, and show that P follows, then P must be true. Here is the template for the CASES rule.

```
P BY CASES, Q|R,
PROOF;
CASE Q;
    .
    .
    .
    P ...
CASE R;
    .
    .
    .
    P ...
QED;
```

The assertion Q|R is a hypothesis of this rule that has to be written out, just like the hypotheses of the ARITH rule. Each CASE line introduces an assumption, just like the ASSUME line of the => introduction rule. The part of the proof between CASE Q and CASE R is a proof block, and so is the part of the proof between CASE R and QED. The same accessibility rules that apply to the => introduction proof blocks apply for these proof blocks as well: it is all right to reach around a proof block for a hypothesis, but you are not allowed to reach into a proof block.

The above template is for a hypothesis with a two-place |. There are similar templates for an | operator with three or more places. You simply include a CASE line and a proof block for each operand. If you applied the | elimination rule to P | Q | R, you would have three cases: P, Q, and R. However, if you applied the rule to the hypothesis (P | Q) | R, you would have only two cases: P | Q and R. The cases must be split in the same manner as they appear in the hypothesis.

```
ABS(I)>J => I>J | -I>J BY INTRO,
PROOF;
    ASSUME ABS(I) > J;
    C: I<=0 | I>0 BY ARITH; /# SPLITTING #/
    I>J | -I>J BY CASES, C,
    PROOF;
    CASE I <= 0;
        I<=0 => ABS(I) = -I BY FUNCTION, ABS(I);
        ABS(I) = -I; /# => ELIMINATION #/
        -I > J;      /# SUBSTITUTE INTO ABS(I) > J #/
        I>J | -I>J;  /# | INTRODUCTION #/
    CASE I>0;
        0 < I BY ARITH, I > 0;                  /# COMPARISON     #/
        0 <= I-1 BY ARITH, 0 < I;               /# COMPARISON     #/
        0 <= I-1+1 BY ARITH, 0 <= I-1;          /# WEAKENING      #/
        I-1 = I+(-1) BY ARITH;                  /# SUBTRACTION    #/
        I-1+1 = I+(-1)+1;                       /# SUBSTITUTION   #/
        I+(-1)+1 = I+((-1)+1) BY ARITH;         /# ASSOCIATIVITY  #/
        I-1+1 = I+((-1)+1);                     /# SUBSTITUTION   #/
        -1+1 = 0 BY ARITH;                      /# COMPUTATION    #/
        I-1+1 = I+0;                            /# SUBSTITUTION   #/
        I+0 = I BY ARITH;                       /# IDENTITY       #/
        I-1+1 = I;                              /# SUBSTITUTION   #/
        0 <= I;                                 /# SUBSTITUTION   #/
        I >= 0 BY ARITH, 0 <= I;                /# COMPARISON     #/

        I>=0 => ABS(I)=I BY FUNCTION, ABS(I);
        ABS(I) = I;     /# => ELIMINATION #/
        I > J;          /# SUBSTITUTE INTO ABS(I) > J #/
        I>J | -I>J;     /# | INTRODUCTION #/
    QED;
QED;
```

Figure 4 A Sample Proof

An Example

We now have seen enough rules to write out a proof of an (at least slightly) interesting assertion, which appears in Figure 4. The parts enclosed in /# and #/ are not part of the proof; they are there only to indicate which template has been used. Note the C: that is put before the conclusion of the proof line on the fourth line of the proof. When a variable is followed by a colon, it is called a "label". A label can be put before an assertion that is the conclusion of a proof line, or it can be put before the assumption on an ASSUME or CASE line. On subsequent lines, whenever that variable is written, it means "insert the assertion labeled by this variable here". There is such a use of the label C on the fifth line of the proof.

The amount of work expended to get I>=0 from I>0 is intimidating. However, we will soon learn about some short cuts that can be used to reduce that deduction to one line.

```
'0'B introduction      '0'B elimination      ~Introduction

P ...                  '0'B...                ~P BY INTRO,
.                      .                      PROOF;
.                      .                         [ASSUME P;]
.                      .                          .
~P...                  P;                         .
.                                                 .
.                                                 '0'B...
.                                              QED;
'0'B;
```

Figure 5 Rules using '0'B

Rules for '0'B

Figure 5 shows the introduction and elimination rules for '0'B. When first encountered, '0'B introduction sounds like a crazy idea. How can we prove '0'B, an assertion that is, by definition, false? Remember though, that we understand that an assertion in a proof is true only if all of the assumptions accessible to it are true. If the collection of assumptions accessible at some point contains two of the form P and ~P, then it can never happen that every member of the collection is true. We can therefore correctly assert even '0'B at that point. It should be clear , in fact, that we can justifiably assert anything at such a point, since the meaning of the assertion (in the sense of section I) is irrelevant. This observation is exactly the '0'B elimination rule.

The best way to understand the use of '0'B is to realize that proving it is the formal way of saying that one has made an inconsistent set of assumptions. If it is then necessary to draw the conclusion of an implication or CASE, it may be asserted without further proof. The fact that '0'B has been concluded shows that the implication is vacuous, or the CASE impossible.

There is another way of using the derivation of '0'B to gain infomation about hypotheses. Suppose we have proven A|~A for some assertion A. Then if we assume ~A and reach a contradiction, we know that A must be true by cases. The form of this argument is:

```
A BY CASES, A|~A,
 PROOF;
   CASE ~A;
     .
     .
     .
    '0'B;
     A;
   CASE A;
 QED;
```

```
~(P&Q) => (~P | ~Q) BY INTRO,
PROOF;
    ASSUME ~(P&Q);
    (~P | ~Q) BY CONTRA,
    PROOF;
        ASSUME ~(~P | ~Q);
        ~P BY INTRO,
        PROOF;
            ASSUME P;
            ~Q BY INTRO,
            PROOF;
                ASSUME Q;
                P & Q;
                'O'B;
            QED;
            ~P | ~Q;
            'O'B;
        QED;
        ~P | ~Q;
        'O'B;
    QED;
QED;
```

```
~~(I=I) BY INTRO,
PROOF;
   ASSUME ~(I=I);
   I = I;
   'O'B;
 QED;
```

Figure 6 Using the ~ Introduction and Contradiction Rules

Such a form is abbreviated by the CONTRADICTION rule:

```
A BY CONTRA,
 PROOF;
  [ASSUME ~A;]
     .
     .
     .
   'O'B
  QED;
```

Note that this is more than just an abbreviation of the previous form. It has no reference to the assertion A|~A, and does not need it as a hypothesis. Classical logic makes the assumption of A|~A for every assertion, and so proof by contradiction is a rule of classical logic. However, there are many assertions for which we cannot prove A|~A in PL/CV without using this rule. Systems of logic which do not assume A|~A for every assertion are called constructive. PL/CV is a constructive system as long as the rule of contradiction is not used.

The difference between constructive and non-constructive systems and their relevance is an issue of mathematical philosophy. It seems that all the mathematics needed for programs can be done using constructive logic, and we will not use the rule of contradiction in this manual. It is included for those who may want to do classical mathematics in PL/CV. Figure 6 shows an example of the use of each of these rules. Proofs that use the contradiction rule are often convoluted, and the one in Figure 6 is no exception.

```
ALL introduction                ALL Elimination

ALL X type. P BY INTRO,         P{E//I} BY ALLEL, ALL I type. P, E;
PROOF;
    [ARB[ITRARY] X type;]
    .
    .
    .
    P ...
QED;
```

Figure 7 Introduction and Elimination Rules for ALL

Rules for ALL

Figure 7 shows templates for the ALL introduction and elimination rules. Let us first consider the ALL introduction rule. When you are using the rule, substitute either FIXED or BIT for "type" in the template. Recall that brackets in a template mean that the item is optional. The nested brackets indicate that we can use ARB or ARBITRARY, or leave the line out altogether. You may substitute any variable for X in the template. Although it isn't shown explicitly in the template, the assertion you substitute for P will almost certainly contain the variable you substitute for X. The part of the proof between the PROOF and QED is a proof block, and follows all the normal accessibility rules.

The simplest way of explaining the ALL introduction rule is with an example. Suppose we wish to prove the assertion ALL J FIXED. J-1 < J. An informal argument would be, "Let J be any FIXED value. No matter what J is, we know by the rules of arithmetic that J-1 is less than J. This argument works for any J." The PL/CV version of this argument is given in figure 8. The purpose of the ARBITRARY line is to introduce a new variable into the proof. We will call these variables "logic variables." Every free variable in each assertion we write must be a logic variable. We can think of ARBITRARY lines as assumptions which are required by any proof lines that use the logic variables they introduce, and which are true for any value of those variables. Just as with any other hypothesis, a proof line may not reach into a proof block to get an ARBITRARY line for its free variables. Alternatively, we can think of an "ARBITRARY J" line as a kind of quantifier which says that the following block is a correct proof for any value substituted for J. We can then think of the proof block as the scope of the ARBITRARY line, in which all free occurrences of J are bound by the "quantifier".

This latter interpretation helps us understand a small confusion that can arise. What if someone tries to introduce the logic variable J at a point where J is already introduced? This problem is entirely equivalent to understanding what an assertion like

```
ALL J FIXED.( (J<0|J>=0) & ALL J FIXED.J<J+1 )
```

means, and in fact would occur in the proof of this assertion. A quantifier binds all <u>free</u> occurrences of its variable inside its scope. The

```
ALL J FIXED. J-1<J BY INTRO,
PROOF;
    ARBITRARY J FIXED;
    J=J;
    J<=J BY ARITH, J=J;                    /# weakening #/
    1<=1 BY ARITH;                         /# computation #/
    J-1<=J-1 BY ARITH, J<=J,-,1=1;         /# relation subtraction #/
    J-1<J BY ARITH, J-1<=J-1;              /# comparison #/
QED;
```

Figure 8. Example of ALL-introduction rule

meaning of the above assertion is: any value for J will make J<0|J>=0 true and ALL J FIXED.(J<J+1) is true. Note that this meaning does not explicitly involve substituting for the J's bound by the inner quantifier. That comes up only in determining the truth of that innermost quantified assertion. Similarly, an ARBITRARY "quantifier" binds only the free occurrences of its variable inside its scope. Therefore, of all the ARBITRARY lines which are accessible to some occurrence of a logic variable, it will be that of the smallest or innermost proof block which binds it.

This situation only rarely happens in actual proofs (usually long ones). It can always be avoided by careful choice of variable names. (Remember that ALL J FIXED.J<J+1 means the same thing as ALL FOO FIXED.FOO<FOO+1.) Nevertheless, to make sure our proofs are correct in the cases where it does happen, we must add the following restriction to the accessibility rules: if point A is inside a proof block with an "ARBITRARY X type" line, then no assertion containing free X outside of that block is accessible from A.

Example:

```
ALL J FIXED. J-1<J BY INTRO,
 PROOF;
    ARB J FIXED;
    .
    .
    .
     J-1<J ...
 QED;
J-1 <= J BY ARITH, J-1<J;
```

If this section of proof is not inside a proof block with an ARBITRARY J FIXED line, then the bottom line is incorrect, since it has no accessible introduction of its free variable. If there is such an enclosing block, then the inner proof of J-1<J cannot use as hypotheses any assertions with free J that come from outside (because in a sense, it is a "different" J inside this proof block).

It now should be clear how to find out the type of any variable in a proof. If the variable is bound, look at the quantifier to which it is bound. If the variable is free, then look at the ARBITRARY line that introduced it.

(The ARBITRARY line is optional; if it weren't written, just use the ARBITRARY line you would have to write if you were going to write one.) If you think of the ARBITRARY line as a quantifier then every variable we write in our proofs is bound, and its type is determined by its binding.

All the sample proofs shown so far have been incomplete, since they have all contained free variables and no ARBITRARY lines. They have actually been the parts that go inside a proof block of an ALL introduction rule.

Next, look at the ALL elimination rule. In the ALLEL template, I stands for a variable, and E stands for any expression of the appropriate type.

The idea behind the ALL elimination rule is quite simple. Suppose we have proved the assertion ALL J FIXED. J-1 < J. Then we know that the assertion J-1 < J holds for <u>any</u> fixed value given to J. The ALL elimination rule lets us prove that assertion for any particular expression, such as K+2:

```
(K+2)-1 < K+2 BY ALLEL, ALL J FIXED. J-1 < J, K+2;
```

Rules for SOME

Figure 9 contains the templates for the SOME introduction and elimination rules. Both these templates use the new substitution notation introduced with the ALL elimination rule. In the introduction template, I stands for a variable, and E for any expression of the same type. In the elimination rule, both I and J stand for variables.

Suppose we wish to prove as assertion like SOME K FIXED. K>5. This assertion says that there is a FIXED value greater than 5. People from Missouri will like the SOME introduction rule; in essence, it says "show me". In this case, a value for K that will work is 6. The proof is:

```
6 > 5 BY ARITH;
SOME K FIXED. K>5 BY INTRO, 6;
```

On the other hand, suppose we have a proof that SOME K FIXED. K*K=J. This assertion states that J is a perfect square. The SOME elimination rule lets us assign a name for "the square root of J", and use that name in subsequent proof lines. (This name must not be the same as one from the ARBITRARY line of the smallest proof block containing it, if there is one, nor may it be the same as one used by a previous CHOOSE line in the same block.) For example:

```
SOME K FIXED. K*K=J ...
CHOOSE ROOTJ FIXED WHERE ROOTJ*ROOTJ = J;
```

The CHOOSE line introduces a new logic variable in the same way that the ARBITRARY line does. Subsequent proof lines can use ROOTJ as a free variable, and ROOTJ*ROOTJ = J as a hypothesis. Naturally, proof lines cannot reach into a proof block to get at ROOTJ.

As with ARBITRARY lines, the CHOOSE line can be thought of as a quantifier. It means that any value for ROOTJ that makes the WHERE assertion true will make the rest of the proof block correct. The scope of a CHOOSE line is

```
SOME introduction                SOME elimination

P{E//I} ...                      SOME I type. P ...
.                                .
.                                .
.                                .
SOME I type. P BY INTRO, E;      CHOOSE J type WHERE P{J//I};
```

Figure 9 Introduction and Elimination Rules for SOME

not the whole proof block, as with ARBITRARY lines, but only that part of the block that follows the CHOOSE. Of all the CHOOSE ROOTJ or ARBITRARY ROOTJ lines that are accessible to an instance of ROOTJ, it is the one of smallest (innermost) scope that binds it. Assertions with free occurrences of ROOTJ are not accessible to any point that must "reach through" such a scope.

Extended Quantifiers

There are some abbreviations you can use when writing quantifiers. Rather than

```
ALL I FIXED. ALL J FIXED. (I<J => I+1<J+1)
```

you can write

```
ALL (I, J) FIXED. (I<J => I+1<J+1)
```

instead. These two assertions mean the same thing, in the same way that I<J<K means the same as I<J & J<K & K<J. You can collapse together a string of any number of ALLs, or a string of any number of SOMEs, but not a mixed string. For example, in

```
ALL I FIXED. SOME J FIXED. I<J
```

there is no way to collapse the quantifiers. The introduction and elimination rules can be used on extended quantifiers. Use the examples in figure 10 as a guide.

All introduction:

```
ALL (X, Y) FIXED. (X<=Y | Y>X) BY INTRO,
PROOF;
    ARBITRARY (X, Y) FIXED;
    X<=Y | Y>X BY ARITH;     /# splitting #/
QED;
```

ALL elimination:

```
X<=X+1 & X+1<=Z => X<=Z BY ALLEL,
    ALL (I, J, K) FIXED. (I<=J & J<=K => I<=K), X, X+1, Z;
```