Purpose

To provide a convenient extension to C syntax for the C-for-Inform language, which is likely never to be used for any program other than the Inform 7 compiler.

§**1. Unit testing.** The following provides a simple test of the extended syntax, and is run by using the `-test-extensions` switch at the command line. It gives some examples of what we'll be compiling below, anyway.

```
sub full_test_double_squares {
    test_ds("[[w1, w2 == ...]]");
    test_ds("[[w1, w2 == golly]]");
    test_ds("[[word w1 == zowie]]");
    test_ds("[[word w1 == wow/huh/zowie]]");
    test_ds("[[w1, w2 == by the pool]]");
    test_ds("[[w1, w2 == moreover ***]]");
    test_ds("[[w1, w2 == *** by the pool]]");
    test_ds("[[w1, w2 == by the pool ***]]");
    test_ds("[[w1, w2 == golly gosh/moses mrs smith]]");
    test_ds("[[w1, w2 == so mr ### we meet again]]");
    test_ds("[[w1, w2 == hello there ...]]");
    test_ds("[[w1, w2 == ... the memory of water]]");
    test_ds("[[w1, w2 <-- something]]");
    test_ds("[[w1, w2 == ... nantucket ...]]");
    test_ds("[[w1, w2 == ... when ... : w]]");
    test_ds("[[w1, w2 == hot porridge ... when ... : w]]");
    test_ds("[[w1, w2 == ... when ... breakfast is served]]");
    test_ds("[[w1, w2 == ... when ... and ... breakfast is served]]");
    test_ds("[[w1, w2 == ... when ... and ... breakfast is served : w]]");
    test_ds("[[w1, w2 == ... when ... and ... breakfast is served : w, x]]");
    test_ds("[[w1, w2 == ... when ... and ... breakfast is served : w, x, y]]");
    test_ds("[[w1, w2 == ... when ... breakfast is served : w, z, x]]");
    test_ds("[[w1, w2 == bacon ... when ... breakfast is served : w]]");
    test_ds("[[w1, w2 == and now ... --> now1, now2]]");
    test_ds("[[w1, w2 == ... until then --> then1, then2]]");
    test_ds("[[w1, w2 == ... when ... : w --> w1, w2 ... when1, when2]]");
    test_ds("[[w1, w2 == ... when ... : w --> when1, when2 ... w1, w2]]");
    test_ds("[[w1, w2 == ... when ... : w --> when1, w2 ... when2, w1]]");
    test_ds("[[w1, w2 == ... OPENBRACKET for ... only CLOSEBRACKET --> x1, x2 ... vm1, vm2]]");
}

sub test_ds {
    my $original = $_[0];
    print "inweb: testing expansion ", $original, "\n\n";
    print expand_double_squares(" ".$original), "\n\n";
}
```

§**2. The expander.** We are given what might, or might not, be a valid command. If we can make sense of it, we return its expansion; if not, we return an error message with a query ? in column 1.

```
sub expand_double_squared_command {
    my $comm = $_[0];
    my $operator = 0;
    my $lhs;
    my $rhs;
    my $w1;
    my $w2;
    my $single_word_flag = 0;
    ⟨Identify the command as either a test or an assignment 3⟩;
    ⟨Parse the left-hand side into C expressions for a word range 4⟩;
    if ($operator == 1) {
        if ($single_word_flag == 1) ⟨Expand a single word test 6⟩
        else ⟨Expand a word range test 7⟩;
    }
    if ($operator == 2) ⟨Expand an assignment command 5⟩;
    return '?: This should not happen';
}
```

§**3.** We split the line as RHS, operator, LHS; there are only two valid operators.

⟨Identify the command as either a test or an assignment 3⟩ ≡

```
    if ($comm =~ m/^(.*?)\s*\=\=\s*(.*?)\s*$/) {
        $operator = 1; $lhs = $1; $rhs = $2;
    } else {
        if ($comm =~ m/^(.*?)\s*\<\-\-\s*(.*?)\s*$/) {
            $operator = 2; $lhs = $1; $rhs = $2;
        }
    }
    if ($operator == 0) {
        return '?: [[ ... ]] with neither <-- nor == operators';
    }
```

This code is used in §2.

§**4.**

⟨Parse the left-hand side into C expressions for a word range 4⟩ ≡

```
    if ($lhs =~ m/^\s*(\S+?)\s*\,\s*(\S+)\s*$/) {
        $w1 = $1; $w2 = $2;
    } else {
        if ($lhs =~ m/^\s*(\S+)\s*$/) {
            $w1 = $1."->word_ref1"; $w2 = $1."->word_ref2";
        } else {
            if ($lhs =~ m/^\s*word\s+(\S+)\s*$/) {
                $w1 = $1; $w2 = $w1; $single_word_flag = 1;
            } else {
                return '?: [[ ... ]] without comma-separated word pair';
            }
        }
    }
```

This code is used in §2.

§**5.** There are three basic commands, of which one is easy:

⟨Expand an assignment command 5⟩ ≡
```
    if ($single_word_flag == 1) {
        return '?: [[ ... ]] cannot assign a single word variable';
    }
    return $w1 . " = " . $rhs . "->word_ref1; ". $w2 . " = " . $rhs . "->word_ref2;";
```
This code is used in §2.

§**6.** And one is nearly easy:

⟨Expand a single word test 6⟩ ≡
```
    if ($rhs =~ m/^\s*(\S+?)\s*$/) {
        my $list = $1;
        my $cond = '((.$w1.'>=0) && (';
        my $ct = 1;
        while ($list =~ m/^([^\/]+)(.*)$/) {
            if ($ct > 1) { $cond .= ' || '; }
            $ct++;
            $cond .= compare_word_p($w1, 0, $1);
            $list = $2; $list =~ s/^\///;
        }
        $cond .= '))';
        return $cond;
    }
    return '?: [[ ... ]] cannot test a single word variable to other than a single word';
```
This code is used in §2.

§**7.** While the third is the big one. We are now comparing a word range against what may be a complicated pattern.

⟨Expand a word range test 7⟩ ≡
```
    my $write_positions = "";
    my $write_ranges = "";
    ⟨Split off optional portions of the pattern indication what positions or ranges to write 8⟩;
    my $right_unanchored = 0;
    my $left_unanchored = 0;
    ⟨Determine whether the pattern is anchored to the right or left edges 9⟩;
    my $nw = 0;
    ⟨Split the pattern into an array of words 10⟩;
    my $nr = 0;
    ⟨Divide this up into ranges with ellipses between 11⟩;
    ⟨Work out how these ranges are anchored 12⟩;
    if ($nr == 0) { return "(FALSE)"; }
    my $cond = "(";
    ⟨Work out a C condition to test the pattern 13⟩;
    $cond .= ")";
    if ($write_ranges ne "") { return '?: [[ ... ]] has too many write pairs of variables'; }
    if ($write_positions ne "") { return '?: [[ ... ]] has too many : variables'; }
    return $cond;
```
This code is used in §2.

§**8.**

⟨Split off optional portions of the pattern indication what positions or ranges to write 8⟩ ≡

```
if ($rhs =~ m/^(.*)\s*\-\-\>\s*(.*?)\s*$/) {
    $write_ranges = $2; $rhs = $1;
}
if ($rhs =~ m/^(.*)\s*\:\s*(.*?)\s*$/) {
    $rhs = $1; $write_positions = $2;
}
if (($rhs.' '.$write_positions) =~ m/\-\-\>/) { return '?: [[ ... ]] has too many -->s'; }
$rhs =~ m/^\s*(.*?)\s*$/; $rhs = $1;
$write_positions =~ m/^\s*(.*?)\s*$/; $write_positions = $1;
$write_ranges =~ m/^\s*(.*?)\s*$/; $write_ranges = $1;
```

This code is used in §7.

§**9.** If either end of the pattern is `***`, this means "match all words here right up to the edge"; we'll call the pattern "unanchored", because the word position where the match will start or finish is somewhere loose inside the word range we're matching.

⟨Determine whether the pattern is anchored to the right or left edges 9⟩ ≡

```
if ($rhs =~ m/^(.*)\s+\*\*\*\s*$/) {
    $rhs = $1;
    $right_unanchored = 1;
}
if ($rhs =~ m/^\*\*\*\s+(.*)$/) {
    $rhs = $1;
    $left_unanchored = 1;
}
```

This code is used in §7.

§**10.**

⟨Split the pattern into an array of words 10⟩ ≡

```
while ($rhs =~ m/^(\S*?)\s+(.*)$/) {
    $words[$nw++] = $1; $rhs = $2;
}
$words[$nw++] = $rhs;
```

This code is used in §7.

§**11.**

⟨Divide this up into ranges with ellipses between 11⟩ ≡

```
my $i;
my $range_start = -1;
for ($i=0; $i < $nw; $i++) {
    if ($words[$i] eq '...') {
        if ($range_start >= 0) {
            $range_from[$nr] = $range_start; $range_to[$nr++] = $i-1;
            $range_start = -1;
        }
    } else {
        if ($range_start == -1) { $range_start = $i; }
    }
}
if ($range_start >= 0) {
    $range_from[$nr] = $range_start; $range_to[$nr++] = $nw-1;
}
```

This code is used in §7.

§**12.**

⟨Work out how these ranges are anchored 12⟩ ≡

```
my $i;
for ($i=0; $i<$nr; $i++) {
    $range_length[$i] = $range_to[$i]-$range_from[$i]+1;
    $range_anchor[$i] = -1;
    $range_anchor_direction[$i] = 0;
    $range_from_fixed[$i] = 0;
    $range_to_fixed[$i] = 0;
    if (($range_from[$i] == 0) && ($left_unanchored == 0)) {
        $range_anchor[$i] = 0;
        $range_anchor_direction[$i] = 1;
        $range_from_fixed[$i] = 1;
    }
    if (($range_to[$i] == $nw-1) && ($right_unanchored == 0)) {
        $range_anchor[$i] = $nw-1;
        $range_anchor_direction[$i] = -1;
        $range_to_fixed[$i] = 1;
    }
}
```

This code is used in §7.

§**13.**

⟨Work out a C condition to test the pattern 13⟩ ≡

⟨Begin with a check that the word range is valid and includes enough words 14⟩;

```
my $i;
for ($i=0; $i<$nr; $i++) { $range_done[$i] = 0; }
$left_inset = 0;
$right_inset = 0;
$left_extent = 0;
$right_extent = 0;
```

⟨First check all of the ranges with fixed endpoints 15⟩;
⟨Then check all of the ranges with floating endpoints 16⟩;

```
for ($i=0; $i<$nr; $i++) {
    if ($range_done[$i] == 0) { return '?: [[ ... ]] has intractable range'; }
}
```

⟨Add always-true conditions with the side effect of writing the word ranges 17⟩;

This code is used in §7.

§**14.** Word ranges inside the Inform compiler are pairs of values (w_1, w_2) where either $w_1 = w_2 = -1$, meaning "no text", or $0 \le w_1 \le w_2 < N$, where N is the number of words read in by the lexer. Here we are testing a word range where w_1 is represented by the C expression in `$w1`, and similarly w_2 by `$w2`. So we first check that $w_1 \ge 0$, and then that not only is $w_2 \ge w_1$, but also that it's larger by sufficient to include all of the words we're trying to match. Thus

```
[[x1, x2 == unseeded ... grapes]]
```

will certainly fail if the word range contains fewer than three words.

⟨Begin with a check that the word range is valid and includes enough words 14⟩ ≡

```
$cond .= "(". $w1 . ">=0)";
my $need = 0;                                   # minimum number of words needed
my $exact = 1;                                  # is this the exact number of words we must have?
for ($i=0; $i<$nr; $i++) {
    $need = $need + $range_length[$i];
    if ($range_from_fixed[$i] == 0) { $exact = 0; }
    if ($range_to_fixed[$i] == 0) { $exact = 0; }
}
for ($i=0; $i<$nw; $i++) {
    if ($words[$i] eq '...') { $need++; }
}
$need--;
if ($exact == 1) { $op = "=="; } else { $op = ">="; }
$cond .= " && (". $w2 . $op . var_offset($w1, $need) . ")";
```

This code is used in §13.

§**15.** Suppose we have a pattern such as

```
[[x1, x2 == peeled or ... perhaps ... unpeeled mangos]]
```

This gives us three ranges to match, of lengths 2, 1, 2. We start by testing the two end ranges, because we can do so quickly – if there's going to be a match, we know exactly at what word positions they must occur. Thus, "peeled" must be at `x1`, "mangos" must be at `x2`, and so on. The reason for doing this is that it will be slower to find "perhaps", whose position is ambiguous, so we don't want to look unless the pattern otherwise matches – this enables us to reject non-matching word ranges more quickly.

⟨First check all of the ranges with fixed endpoints 15⟩ ≡

```
my $j;
for ($i=0; $i<$nr; $i++) {
    if ($range_done[$i] == 1) { next; }
    if (($range_from[$i] == 0) && ($range_from_fixed[$i] == 1)) {
        for ($j=0; $j<$range_length[$i]; $j++) {
            $cond .= compare_word($w1, $j, $words[$j]);
        }
        $range_done[$i] = 1;
        $left_extent = $range_length[$i];
        $left_inset += $range_length[$i] - 1;
        next;
    }
    if (($range_to[$i] == $nw-1) && ($range_to_fixed[$i] == 1)) {
        for ($j=$range_length[$i]-1; $j>=0; $j--) {
            $cond .= compare_word($w2, 0-$j, $words[$nw-1-$j]);
        }
        $range_done[$i] = 1;
        $right_inset += $range_length[$i] - 1;
        $right_extent = $range_length[$i];
        next;
    }
}
```

This code is used in §13.

§**16.** A floating range can consist of only a single word, at present. (An occasional nuisance when coding Inform, but basically an acceptable compromise.) We detect this with a call to Inform's routine `is_word_intermediate`, which determines whether a given word occurs at a position p such that $w_1 < p < w_2$, and returns the minimum value of p if it does, -1 if it does not. If there's only one floating range like this, we can just check the return value to see if it's non-negative. But if there are two or more floating ranges, thus:

```
[[x1, x2 == peeled and ... seeded ... sliced ... in syrup : i, j]]
```

then we need to store the position of "seeded", since we must not only check that it exists but also use that as the left end of the range of words inside which we look for "sliced". To do this, we extract the name of a C variable to store the information in, finding it in the `$write_positions` part of the pattern. Thus on a successful match, the example text would store the word position of "seeded" in `i`, and of "sliced" in `j`.

⟨Then check all of the ranges with floating endpoints 16⟩ ≡

```
my $left_var = $w1;
my $right_var = $w2;
$unwritten_var_exists = 0;
$no_written_vars = 0;
my $i;
for ($i=0; $i<$nr; $i++) {
```

```
        if ($range_done[$i] == 1) { next; }
        if ($range_length[$i] == 1) {
            $cond .= " && (";
            if ($write_positions ne "") {
                if ($write_positions =~ m/^\s*(.*?)\s*\,\s*(.*)$/) {
                    $write_var = $1; $write_positions = $2;
                } else { $write_var = $write_positions; $write_positions = ""; }
                $cond .= "(".$write_var."=";
            } else {
                if ($unwritten_var_exists == 1) {
                    return '?: [[ ... ]] has insufficient : variables';
                }
                $unwritten_var_exists = 1;
            }
            if ($write_var eq "") { $cond.= "("; }
            $cond .=
                "Text__is_word_intermediate(".$words[$range_from[$i]]."_V,"
                . var_offset($left_var, $left_inset)
                . ","
                . var_offset($right_var, 0-$right_inset).("))";
            if ($write_var ne "") {
                $cond .= ",(".$write_var.">=0))";
                $left_var = $write_var;
                $written_vars[$no_written_vars++] = $write_var;
            } else {
                $cond.= " >= 0)";
            }
            $range_done[$i] = 1;
            next;
        }
    }
```

This code is used in §13.

§**17.** That's it for testing the condition: now for the tricky part, storing the word ranges matched on a successful outcome. Note that nothing is written unless the test has succeeded. To see the difficulty here, consider

```
[[w1, w2 == ... OPENBRACKET ... CLOSEBRACKET : i --> w1, w2 ... p1, p2]]
```

which looks for a bracketed clause at the end of the word range and splits it off into (p_1, p_2). We are clearly going to want to set p_2 to w_2, but to the old value of w_2, not the new one, which is going to be just before the open-bracket. So it is important to assign these variables in the right order, so that we don't change w_2 before using it.

⟨Add always-true conditions with the side effect of writing the word ranges 17⟩ ≡

```
$no_assignments = 0;
⟨Work out which variables need to be assigned, and to what 18⟩;
for ($i=0; $i<$no_assignments; $i++) { $assignment_done[$i] = 0; }
⟨Iteratively make all possible safe assignments until all have been assigned 19⟩;
```

This code is used in §13.

§**18.** Let's call the variables to be written $v_1, v_2, v_3, ..., v_n$, where n will always be an even number. (In the example above, $n = 4$.)

In the following, we generate triples (v_i, v_k, x) to represent that an assignment $v_i \mapsto v_k + x$. Each triple is set up by a call to the routine `assign_set_var`.

⟨Work out which variables need to be assigned, and to what 18⟩ ≡

```
my $j = 0;
my $i;
for ($i=0; $i<$nw; $i++) {
    if ($words[$i] eq '...') {
        if ($j == 0) {
            $from_var = $w1;
            $from_offset = $left_extent;
        } else {
            $from_var = $written_vars[$j-1];
            $from_offset = 1;
        }
        if ($j<$no_written_vars) {
            $to_var = $written_vars[$j];
            $to_offset = -1;
            $j++;
        } else {
            $to_var = $w2;
            $to_offset = 0-$right_extent;
        }
        if ($write_ranges =~ m/^\s*(\S+)\s*\,\s*(\S+)\s*(.*?)\s*$/) {
            $write_from_var = $1;
            $write_to_var = $2;
            $write_ranges = $3;
            assign_set_var($write_from_var, $from_var, $from_offset);
            assign_set_var($write_to_var, $to_var, $to_offset);
            if ($write_ranges ne "") {
                if ($write_ranges =~ m/^\s*\.\.\.\s*(.*)$/) {
                    $write_ranges = $1;
                } else {
                    return '?: [[ ... ]] pairs of write vars should be divided by ...';
                }
            }
        } else {
            if ($write_ranges ne "") {
                return '?: [[ ... ]] write vars should be in pairs divided by ...';
            }
        }
    }
}
```

This code is used in §17.

§19. Now we have a mass of triples (v_i, v_k, x), each representing that an assignment $v_i \mapsto v_k + x$ must be made. But we can't make this assignment until v_i no longer appears in any unassigned triple (v_j, v_i, y), because to do so would invalidate the value of v_i used in that assignment.

We solve this iteratively, at each stage looking for the safe triples to assign. For each variable occurring in the middle position, we calculate `$necessity_count{$v}` as the number of other variables whose assignment is to `$v` plus or minus some offset. Thus, it's safe to assign to `$v` if and only if the necessity count is 0.

A case we have to be careful about is (v_i, v_i, x), where the assignment we'll make has the effect of adding x to v_i. This would be impossible ever to carry out if we regarded it as a dependency of v_i upon itself, which is why such a triple doesn't contribute to the necessity count of v_i.

⟨Iteratively make all possible safe assignments until all have been assigned 19⟩ ≡

```
my $no_done = 0;
my $safety_check = 0;                                   in case of a bug in this code, really
while ($no_done < $no_assignments) {
    ⟨Calculate the necessity count for each unassigned variable 20⟩;
    ⟨Assign all unassigned variables with necessity count 0 21⟩;
    if ($safety_check++ == 1000) ⟨Panic! Something has gone terribly wrong 22⟩;
}
```

This code is used in §17.

§20.

⟨Calculate the necessity count for each unassigned variable 20⟩ ≡

```
my $i;
for ($i=0; $i<$no_assignments; $i++) {
    $necessity_count{$var_to_read[$i]} = 0;
    $necessity_count{$var_to_assign[$i]} = 0;
}
for ($i=0; $i<$no_assignments; $i++) {
    if ($assignment_done[$i] == 0) {
        if ($var_to_read[$i] ne $var_to_assign[$i]) {
            $necessity_count{$var_to_read[$i]}++;
        }
    }
}
```

This code is used in §19.

§21.

⟨Assign all unassigned variables with necessity count 0 21⟩ ≡

```
my $i;
for ($i=0; $i<$no_assignments; $i++) {
    if (($assignment_done[$i] == 0) && ($necessity_count{$var_to_assign[$i]} == 0)) {
        $cond .= set_var($var_to_assign[$i], $var_to_read[$i], $var_to_offset[$i]);
        $assignment_done[$i] = 1;
        $no_done++;
    }
}
```

This code is used in §19.

§**22.** Well, perhaps this is overdramatising it. A simple case here might be

```
[[w1, w2 == possibly ... --> w2, w1]]
```

where we have to make the assignments $w_2 \mapsto w_1 + 1$, $w_1 \mapsto w_2$. Each variable depends on the other, and the log-jam cannot be broken without use of some additional storage (which C offers no convenient way to allocate, within a condition). No doubt with more labour we could contrive a way around this, but it's not worth it: such cases never in practice arose in the whole history of writing Inform.

⟨Panic! Something has gone terribly wrong 22⟩ ≡

```
my $i;
for ($i=0; $i<$no_assignments; $i++) {
    if ($assignment_done[$i] == 0) {
        inweb_error("  ".$var_to_assign[$i]." = ".$var_to_read[$i]." + ".$var_to_offset[$i]);
    }
}
return '?: [[ ... ]] no safe way to write these pairs of variables';
```

This code is used in §19.

§**23.** And that's it for the main routine. We needed two routines for dealing with these variable-assignment triples; first, to create one:

```
sub assign_set_var {
    my $write_var = $_[0];
    my $from_var = $_[1];
    my $from_offset = $_[2];
    if (($from_offset == 0) && ($write_var eq $from_var)) { return ""; }
    $var_to_assign[$no_assignments] = $write_var;
    $var_to_read[$no_assignments] = $from_var;
    $var_to_offset[$no_assignments] = $from_offset;
    $no_assignments++;
}
```

§**24.** And then to compile the assignment asked for:

```
sub set_var {
    my $write_var = $_[0];
    my $from_var = $_[1];
    my $from_offset = $_[2];
    if (($from_offset == 0) && ($write_var eq $from_var)) { return ""; }
    return " && (".$write_var."=".var_offset($from_var, $from_offset).")";
}
```

§**25.** These both need the following useful routine, which compiles a C expression consisting of a variable number plus a constant offset. For legibility, we compile to, say, `q1-5` rather than `q1+-5` in the case of a negative offset.

```
sub var_offset {
    my $var = $_[0];
    my $offset = $_[1];
    if ($offset == 0) { return $var; }
    if ($offset > 0) { return $var."+".$offset; }
    return $var."-".(0-$offset);
}
```

§**26.** Finally, a routine to generate the condition for testing a single word. Here we know we want to test the word whose number is stored in `$var` plus a constant `$with`; for instance, word `w2-1`, the penultimate word in the range being tested. And we want to test that this word is the one in `$with`.

The slight complication here is that a word in the form

```
apple/pear/peach
```

means "any of apple, pear or peach", so that we have to compile a compound condition.

```
sub compare_word {
    my $var = $_[0];
    my $offset = $_[1];
    my $with = $_[2];
    my $this;
    my $rest;
    my $cond;
    $cond = " && ";
    if ($with =~ m/\//) {
        $cond .= "(";
        while ($with =~ m/^(.*?)\/(.*)$/) {
            $this = $1; $with = $2;
            $cond .= compare_word_p($var, $offset, $this);
            $cond .= " || ";
        }
        $cond .= compare_word_p($var, $offset, $with);
        $cond .= ")";
    } else {
        $cond .= compare_word_p($var, $offset, $with);
    }
    return $cond;
}
```

§**27.** So it's actually *this* routine which generates the test of a single word. There's one final language feature to implement: the special word `###` matches any single word.

```
sub compare_word_p {
    my $var = $_[0];
    my $offset = $_[1];
    my $with = $_[2];
    if ($with eq '###') { return "(TRUE)"; }
    return "(compare_word(" . var_offset($var, $offset) . ", " . $with . "_V))";
}
```