5. Data and Control Flow

5.1 Generalized Reference

5.1.1 Overview of Places and Generalized Reference

A *generalized reference* is the use of a *form*, sometimes called a *place*, as if it were a *variable* that could be read and written. The *value* of a *place* is the *object* to which the *place form* evaluates. The *value* of a *place* can be changed by using **setf**. The concept of binding a *place* is not defined in Common Lisp, but an *implementation* is permitted to extend the language by defining this concept.

The next figure contains examples of the use of **setf**. Note that the values returned by evaluating the *forms* in column two are not necessarily the same as those obtained by evaluating the *forms* in column three. In general, the exact *macro expansion* of a **setf** *form* is not guaranteed and can even be *implementation-dependent*; all that is guaranteed is that the expansion is an update form that works for that particular *implementation*, that the left-to-right evaluation of *subforms* is preserved, and that the ultimate result of evaluating **setf** is the value or values being stored.

```
Access function   Update Function   Update using setf
x                 (setq x datum)    (setf x datum)
(car x)           (rplaca x datum)  (setf (car x) datum)
(symbol-value x)  (set x datum)     (setf (symbol-value x) datum)
```

Figure 5-1. Examples of setf

The next figure shows *operators* relating to *places* and *generalized reference*.

```
assert                 defsetf              push
ccase                  get-setf-expansion   remf
ctypecase              getf                 rotatef
decf                   incf                 setf
define-modify-macro    pop                  shiftf
define-setf-expander   psetf
```

Figure 5-2. Operators relating to places and generalized reference.

Some of the *operators* above manipulate *places* and some manipulate *setf expanders*. A *setf expansion* can be derived from any *place*. New *setf expanders* can be defined by using **defsetf** and **define-setf-expander**.

5.1.1.1 Evaluation of Subforms to Places

The following rules apply to the *evaluation* of *subforms* in a *place*:

1. The evaluation ordering of *subforms* within a *place* is determined by the order specified by the second value returned by **get-setf-expansion**. For all *places* defined by this specification (e.g., **getf**, **ldb**, ...), this order of evaluation is left-to-right. When a *place* is derived from a macro expansion, this rule is applied after the macro is expanded to find the appropriate *place*.

   Places defined by using **defmacro** or **define-setf-expander** use the evaluation order defined by those definitions. For example, consider the following:

   ```
    (defmacro wrong-order (x y) ‘(getf ,y ,x))
   ```

   This following *form* evaluates `place2` first and then `place1` because that is the order they are evaluated in the macro expansion:

```
(push value (wrong-order place1 place2))
```

2. For the *macros* that manipulate *places* (**push**, **pushnew**, **remf**, **incf**, **decf**, **shiftf**, **rotatef**, **psetf**, **setf**, **pop**, and those defined by **define-modify-macro**) the *subforms* of the macro call are evaluated exactly once in left-to-right order, with the *subforms* of the *places* evaluated in the order specified in (1).

   push, **pushnew**, **remf**, **incf**, **decf**, **shiftf**, **rotatef**, **psetf**, **pop** evaluate all *subforms* before modifying any of the *place* locations. **setf** (in the case when **setf** has more than two arguments) performs its operation on each pair in sequence. For example, in

   ```
   (setf place1 value1 place2 value2 ...)
   ```

   the *subforms* of `place1` and `value1` are evaluated, the location specified by `place1` is modified to contain the value returned by `value1`, and then the rest of the **setf** form is processed in a like manner.

3. For **check-type**, **ctypecase**, and **ccase**, *subforms* of the *place* are evaluated once as in (1), but might be evaluated again if the type check fails in the case of **check-type** or none of the cases hold in **ctypecase** and **ccase**.
4. For **assert**, the order of evaluation of the generalized references is not specified.

Rules 2, 3 and 4 cover all *standardized macros* that manipulate *places*.

5.1.1.1.1 Examples of Evaluation of Subforms to Places

```
 (let ((ref2 (list '())))
   (push (progn (princ "1") 'ref-1)
         (car (progn (princ "2") ref2))))
>>  12
=>  (REF1)

 (let (x)
    (push (setq x (list 'a))
          (car (setq x (list 'b))))
     x)
=>  (((A) . B))
```

push first evaluates `(setq x (list 'a)) => (a)`, then evaluates `(setq x (list 'b)) => (b)`, then modifies the *car* of this latest value to be `((a) . b)`.

5.1.1.2 Setf Expansions

Sometimes it is possible to avoid evaluating *subforms* of a *place* multiple times or in the wrong order. A *setf expansion* for a given access form can be expressed as an ordered collection of five *objects*:

List of temporary variables
: a list of symbols naming temporary variables to be bound sequentially, as if by **let***, to *values* resulting from value forms.

List of value forms
: a list of forms (typically, *subforms* of the *place*) which when evaluated yield the values to which the corresponding temporary variables should be bound.

List of store variables
: a list of symbols naming temporary store variables which are to hold the new values that will be assigned to the *place*.

Storing form
: a form which can reference both the temporary and the store variables, and which changes the *value* of the *place* and guarantees to return as its values the values of the store variables, which are the correct values for **setf** to return.

Accessing form
: a *form* which can reference the temporary variables, and which returns the *value* of the *place*.

The value returned by the accessing form is affected by execution of the storing form, but either of these forms might be evaluated any number of times.

It is possible to do more than one **setf** in parallel via **psetf**, **shiftf**, and **rotatef**. Because of this, the *setf expander* must produce new temporary and store variable names every time. For examples of how to do this, see **gensym**.

For each *standardized* accessor function *F*, unless it is explicitly documented otherwise, it is *implementation-dependent* whether the ability to use an *F form* as a **setf** *place* is implemented by a *setf expander* or a *setf function*. Also, it follows from this that it is *implementation-dependent* whether the name `(setf` *F*`)` is *fbound*.

5.1.1.2.1 Examples of Setf Expansions

Examples of the contents of the constituents of *setf expansions* follow.

For a variable *x*:

```
()              ;list of temporary variables
()              ;list of value forms
(g0001)         ;list of store variables
(setq x g0001)  ;storing form
x               ;accessing form
```

Figure 5-3. Sample Setf Expansion of a Variable

For `(car` *exp*`)`:

```
(g0002)                             ;list of temporary variables
(exp)                               ;list of value forms
(g0003)                             ;list of store variables
(progn (rplaca g0002 g0003) g0003)  ;storing form
(car g0002)                         ;accessing form
```

Figure 5-4. Sample Setf Expansion of a CAR Form

For `(subseq` *seq s e*`)`:

```
(g0004 g0005 g0006)         ;list of temporary variables
(seq s e)                   ;list of value forms
(g0007)                     ;list of store variables
(progn (replace g0004 g0007 :start1 g0005 :end1 g0006) g0007)
                            ;storing form
(subseq g0004 g0005 g0006)  ; accessing form
```

Figure 5-5. Sample Setf Expansion of a SUBSEQ Form

In some cases, if a *subform* of a *place* is itself a *place*, it is necessary to expand the *subform* in order to compute some of the values in the expansion of the outer *place*. For `(ldb` *bs* `(car` *exp*`))`:

```
(g0001 g0002)            ;list of temporary variables
(bs exp)                 ;list of value forms
(g0003)                  ;list of store variables
(progn (rplaca g0002 (dpb g0003 g0001 (car g0002))) g0003)
                         ;storing form
(ldb g0001 (car g0002))  ; accessing form
```

Figure 5-6. Sample Setf Expansion of a LDB Form

5.1.2 Kinds of Places

Several kinds of *places* are defined by Common Lisp; this section enumerates them. This set can be extended by *implementations* and by *programmer code*.

5.1.2.1 Variable Names as Places

The name of a *lexical variable* or *dynamic variable* can be used as a *place*.

5.1.2.2 Function Call Forms as Places

A *function form* can be used as a *place* if it falls into one of the following categories:

* A function call form whose first element is the name of any one of the functions in the next figure.

`aref`	`cdadr`	`get`
`bit`	`cdar`	`gethash`
`caaaar`	`cddaar`	`logical-pathname-translations`
`caaadr`	`cddadr`	`macro-function`
`caaar`	`cddar`	`ninth`
`caadar`	`cdddar`	`nth`
`caaddr`	`cddddr`	`readtable-case`
`caadr`	`cdddr`	`rest`
`caar`	`cddr`	`row-major-aref`
`cadaar`	`cdr`	`sbit`
`cadadr`	`char`	`schar`
`cadar`	`class-name`	`second`
`caddar`	`compiler-macro-function`	`seventh`
`cadddr`	`documentation`	`sixth`
`caddr`	`eighth`	`slot-value`
`cadr`	`elt`	`subseq`
`car`	`fdefinition`	`svref`
`cdaaar`	`fifth`	`symbol-function`
`cdaadr`	`fill-pointer`	`symbol-plist`
`cdaar`	`find-class`	`symbol-value`
`cdadar`	`first`	`tenth`
`cdaddr`	`fourth`	`third`

Figure 5-7. Functions that setf can be used with---1

In the case of **subseq**, the replacement value must be a *sequence* whose elements might be contained by the sequence argument to **subseq**, but does not have to be a *sequence* of the same *type* as the *sequence* of which the subsequence is specified. If the length of the replacement value does not equal the length of the subsequence to be replaced, then the shorter length determines the number of elements to be stored, as for **replace**.

* A function call form whose first element is the name of a selector function constructed by **defstruct**. The function name must refer to the global function definition, rather than a locally defined *function*.
* A function call form whose first element is the name of any one of the functions in the next figure, provided that the supplied argument to that function is in turn a *place* form; in this case the new *place* has stored back into it the result of applying the supplied "update" function.

Function name	Argument that is a place	Update function used
`ldb`	`second`	`dpb`
`mask-field`	`second`	`deposit-field`
`getf`	`first`	`implementation-dependent`

Figure 5-8. Functions that setf can be used with---2 During the **setf** expansion of these *forms*, it is necessary to call **get-setf-expansion** in order to figure out how the inner, nested generalized variable must be treated.

The information from **get-setf-expansion** is used as follows.

ldb

In a form such as:

`(setf (ldb` *byte-spec place-form*`)` *value-form*`)`

the place referred to by the *place-form* must always be both *read* and *written*; note that the update is to the generalized variable specified by *place-form*, not to any object of *type* **integer**.

Thus this **setf** should generate code to do the following:

1. Evaluate *byte-spec* (and bind it into a temporary variable).
2. Bind the temporary variables for *place-form*.
3. Evaluate *value-form* (and bind its value or values into the store variable).
4. Do the *read* from *place-form*.
5. Do the *write* into *place-form* with the given bits of the *integer* fetched in step 4 replaced with the value from step 3.

If the evaluation of *value-form* in step 3 alters what is found in *place-form*, such as setting different bits of *integer*, then the change of the bits denoted by *byte-spec* is to that altered *integer*, because step 4 is done after the *value-form* evaluation. Nevertheless, the evaluations required for *binding* the temporary variables are done in steps 1 and 2, and thus the expected left-to-right evaluation order is seen. For example:

```
 (setq integer #x69) =>  #x69
 (rotatef (ldb (byte 4 4) integer)
          (ldb (byte 4 0) integer))
 integer =>  #x96
;;; This example is trying to swap two independent bit fields
;;; in an integer.  Note that the generalized variable of
;;; interest here is just the (possibly local) program variable
;;; integer.
```

mask-field

This case is the same as **ldb** in all essential aspects.

getf

In a form such as:

`(setf (getf` *place-form ind-form*`)` *value-form*`)`

the place referred to by *place-form* must always be both *read* and *written*; note that the update is to the generalized variable specified by *place-form*, not necessarily to the particular *list* that is the property list in question.

Thus this **setf** should generate code to do the following:

1. Bind the temporary variables for *place-form*.
2. Evaluate *ind-form* (and bind it into a temporary variable).
3. Evaluate *value-form* (and bind its value or values into the store variable).
4. Do the *read* from *place-form*.
5. Do the *write* into *place-form* with a possibly-new property list obtained by combining the values from steps 2, 3, and 4. (Note that the phrase "possibly-new property list" can mean that the former property list is somehow destructively re-used, or it can mean partial or full copying of it. Since either copying or destructive re-use can occur, the treatment of the resultant value for the possibly-new property list must

proceed as if it were a different copy needing to be stored back into the generalized variable.)
If the evaluation of *value-form* in step 3 alters what is found in *place-form*, such as setting a different named property in the list, then the change of the property denoted by *ind-form* is to that altered list, because step 4 is done after the *value-form* evaluation. Nevertheless, the evaluations required for *binding* the temporary variables are done in steps 1 and 2, and thus the expected left-to-right evaluation order is seen.

For example:

```
 (setq s (setq r (list (list ’a 1 ’b 2 ’c 3)))) =>  ((a 1 b 2 c 3))
 (setf (getf (car r) ’b)
       (progn (setq r nil) 6)) =>  6
 r =>  NIL
 s =>  ((A 1 B 6 C 3))
;;; Note that the (setq r nil) does not affect the actions of
;;; the SETF because the value of R had already been saved in
;;; a temporary variable as part of the step 1. Only the CAR
;;; of this value will be retrieved, and subsequently modified
;;; after the value computation.
```

5.1.2.3 VALUES Forms as Places

A **values** *form* can be used as a *place*, provided that each of its *subforms* is also a *place* form.

A form such as

`(setf (values` *place-1* `...`*place-n*`)` *values-form*`)`

does the following:

1. The *subforms* of each nested *place* are evaluated in left-to-right order.
2. The *values-form* is evaluated, and the first store variable from each *place* is bound to its return values as if by **multiple-value-bind**.
3. If the *setf expansion* for any *place* involves more than one store variable, then the additional store variables are bound to **nil**.
4. The storing forms for each *place* are evaluated in left-to-right order.

The storing form in the *setf expansion* of **values** returns as *multiple values*[2] the values of the store variables in step 2. That is, the number of values returned is the same as the number of *place* forms. This may be more or fewer values than are produced by the *values-form*.

5.1.2.4 THE Forms as Places

A **the** *form* can be used as a *place*, in which case the declaration is transferred to the *newvalue* form, and the resulting **setf** is analyzed. For example,

```
 (setf (the integer (cadr x)) (+ y 3))
```

is processed as if it were

```
 (setf (cadr x) (the integer (+ y 3)))
```

5.1.2.5 APPLY Forms as Places

The following situations involving **setf** of **apply** must be supported:

* `(setf (apply #'aref` *array subscript** *more-subscripts*`)` *new-element*`)`
* `(setf (apply #'bit` *array subscript** *more-subscripts*`)` *new-element*`)`
* `(setf (apply #'sbit` *array subscript** *more-subscripts*`)` *new-element*`)`

In all three cases, the *element* of *array* designated by the concatenation of *subscripts* and *more-subscripts* (i.e., the same *element* which would be *read* by the call to *apply* if it were not part of a **setf** *form*) is changed to have the *value* given by *new-element*. For these usages, the function name (**aref**, **bit**, or **sbit**) must refer to the global function definition, rather than a locally defined *function*.

No other *standardized function* is required to be supported, but an *implementation* may define such support. An *implementation* may also define support for *implementation-defined operators*.

If a user-defined *function* is used in this context, the following equivalence is true, except that care is taken to preserve proper left-to-right evaluation of argument *subforms*:

```
(setf (apply #'name arg*) val)
==  (apply #'(setf name) val arg*)
```

5.1.2.6 Setf Expansions and Places

Any *compound form* for which the *operator* has a *setf expander* defined can be used as a *place*. The *operator* must refer to the global function definition, rather than a locally defined *function* or *macro*.

5.1.2.7 Macro Forms as Places

A *macro form* can be used as a *place*, in which case Common Lisp expands the *macro form* as if by **macroexpand-1** and then uses the *macro expansion* in place of the original *place*. Such *macro expansion* is attempted only after exhausting all other possibilities other than expanding into a call to a function named `(setf` *reader*`)`.

5.1.2.8 Symbol Macros as Places

A reference to a *symbol* that has been *established* as a *symbol macro* can be used as a *place*. In this case, **setf** expands the reference and then analyzes the resulting *form*.

5.1.2.9 Other Compound Forms as Places

For any other *compound form* for which the *operator* is a *symbol f*, the **setf** *form* expands into a call to the *function* named `(setf` *f*`)`. The first *argument* in the newly constructed *function form* is *newvalue* and the remaining *arguments* are the remaining *elements* of *place*. This expansion occurs regardless of whether *f* or `(setf` *f*`)` is defined as a *function* locally, globally, or not at all. For example,

`(setf (`*f arg1 arg2* `...)` *new-value*`)`

expands into a form with the same effect and value as

```
 (let ((#:temp-1 arg1)          ;force correct order of evaluation
       (#:temp-2 arg2)
       ...
       (#:temp-0 new-value))
   (funcall (function (setf f)) #:temp-0 #:temp-1 #:temp-2...))
```

A *function* named `(setf` *f*`)` must return its first argument as its only value in order to preserve the semantics of **setf**.

5.1.3 Treatment of Other Macros Based on SETF

For each of the "read-modify-write" *operators* in the next figure, and for any additional *macros* defined by the *programmer* using **define-modify-macro**, an exception is made to the normal rule of left-to-right evaluation of arguments. Evaluation of *argument forms* occurs in left-to-right order, with the exception that for the *place argument*, the actual *read* of the "old value" from that *place* happens after all of the *argument form evaluations*, and just before a "new value" is computed and *written* back into the *place*.

Specifically, each of these *operators* can be viewed as involving a *form* with the following general syntax:

```
 (operator preceding-form* place following-form*)
```

The evaluation of each such *form* proceeds like this:

1. *Evaluate* each of the *preceding-forms*, in left-to-right order.
2. *Evaluate* the *subforms* of the *place*, in the order specified by the second value of the *setf expansion* for that *place*.
3. *Evaluate* each of the *following-forms*, in left-to-right order.
4. *Read* the old value from *place*.
5. Compute the new value.
6. Store the new value into *place*.

decf	pop	pushnew
incf	push	remf

Figure 5-9. Read-Modify-Write Macros

5.2 Transfer of Control to an Exit Point

When a transfer of control is initiated by **go**, **return-from**, or **throw** the following events occur in order to accomplish the transfer of control. Note that for **go**, the *exit point* is the *form* within the **tagbody** that is being executed at the time the **go** is performed; for **return-from**, the *exit point* is the corresponding **block** *form*; and for **throw**, the *exit point* is the corresponding **catch** *form*.

1. Intervening *exit points* are "abandoned" (i.e., their *extent* ends and it is no longer valid to attempt to transfer control through them).
2. The cleanup clauses of any intervening **unwind-protect** clauses are evaluated.
3. Intervening dynamic *bindings* of **special** variables, *catch tags*, *condition handlers*, and *restarts* are undone.
4. The *extent* of the *exit point* being invoked ends, and control is passed to the target.

The extent of an exit being "abandoned" because it is being passed over ends as soon as the transfer of control is initiated. That is, event 1 occurs at the beginning of the initiation of the transfer of control. The consequences are undefined if an attempt is made to transfer control to an *exit point* whose *dynamic extent* has ended.

Events 2 and 3 are actually performed interleaved, in the order corresponding to the reverse order in which they were established. The effect of this is that the cleanup clauses of an **unwind-protect** see the same dynamic *bindings* of variables and *catch tags* as were visible when the **unwind-protect** was entered.

Event 4 occurs at the end of the transfer of control.