$SPAD/src/boot Makefile

Timothy Daly

September 15, 2009

Abstract

The Scratchpad language is implemented by using a mixture of Lisp and a more convenient language for writing Lisp called "Boot". This document contains a description of the Boot language, and some details of the resulting Lisp programs. The description of the translation functions available are at the end of this file.

The main difference between Lisp and Boot is in the syntax for the application of a function to its argument. The Lisp format (F X Y Z), means, when F is a function, the application of F to its arguments X,Y, and Z, is written in Boot as F(X,Y,Z). When F is a special Lisp word it will be written in Boot by using some other syntactic construction.

Boot contains an easy method of writing expressions that denote lists, and provides an analogous method of writing patterns containing variables and constants which denote a particular class of lists. The pattern is matched against a particular list at run time, and if the list belongs to the class then its variables will take on the values of components of the list.

A second convenient feature provided by Boot is a method of writing programs that iterate over the elements of one or more lists and which either transform the state of the machine, or produce some object from the list or lists.

Contents

1 Boot

Axiom is built in layers. The first layer is contructed into an image called **bootsys**. The **bootsys** image is used to translate boot code to common lisp code. Since a boot coded interpreter is needed to translate the code for the boot coded interpreter we have a "boot-strapping" problem. In order to get the whole process to start we need certain files kept in common lisp form. This directory contains those files.

2 Boot To Common Lisp Translater

2.1 Introduction

The Scratchpad language is implemented by using a mixture of Lisp and a more convenient language for writing Lisp called "Boot". This document contains a description of the Boot language, and some details of the resulting Lisp programs. The description of the translation functions available are at the end of this file.

The main difference between Lisp and Boot is in the syntax for the application of a function to its argument. The Lisp format (F X Y Z), means, when F is a function, the application of F to its arguments X,Y, and Z, is written in Boot as F(X,Y,Z). When F is a special Lisp word it will be written in Boot by using some other syntactic construction.

Boot contains an easy method of writing expressions that denote lists, and provides an analogous method of writing patterns containing variables and constants which denote a particular class of lists. The pattern is matched against a particular list at run time, and if the list belongs to the class then its variables will take on the values of components of the list.

A second convenient feature provided by Boot is a method of writing programs that iterate over the elements of one or more lists and which either transform the state of the machine, or produce some object from the list or lists.

2.2 Lines and Commands

If the first character of a line is a closing parenthesis the line is treated as a command which controls the lines that will be passed to the translater rather than being passed itself. The command)include filename filemodifier will for example be replaced by the lines in the file filename filemodifier.

If a line starts with a closing parenthesis it will be called a command line, otherwise it will be called a plain line. The command lines are

```
name            as written

Include         )include filename filemodifier
IncludeLisp     )includelisp filename filemodifier
If              )if bootexpression
```

```
Else            )else
ElseIf          )elseif bootexpression
EndIf           )endif
Fin             )fin
Say             )say  string
Eval            )eval bootexpression
EvalStrings     )evalstrings bootexpression
Package         )package packagename

SimpleLine::= PlainLine | Include | IncludeLisp |Say | Eval | EvalStrings
              | Package
```

A PlainLine is delivered to the translater as is.

An Include delivers the lines in the file filename.filemodifier, treated as boot lines.

An IncludeLisp delivers the lines in the specified file, treated as lisp lines. The only comments allowed in lisp files that are included in this way require that the semicolon is at the beginning of the line.

A Say outputs the remainder of the line to the console, delivering nothing to the translater.

An Eval translates the reminder of the line, assumed to be written in Boot, to Lisp, and evaluates it, delivering nothing to the translater.

An EvalStrings also translates and evaluates the rest of the line but this time assumes that the Boot expression denotes a list of strings which are then delivered to the translater instead of the EvalString line. The strings are treated as Boot lines.

It is also possible to include or exclude lines based upon some condition which is the result of translating and evaluating the boot expression that follows an)if or)elseif command. This construction will be called a Conditional. A file will be composed from SimpleLines and Conditionals. A file is either terminated by the end of file or by a Fin line.

```
Components ::=(SimpleLine | Conditional)*

File ::= Components  ( Fin | empty)

A conditional is bracketed by an If and an EndIf.

Conditional ::= If Components Elselines EndIf
```

If the boot expression following the)if has value true then the Components are delivered but not the ElseLines, otherwise the Components are ignored ,and the ElseLines are delivered to the translater. In any case the lines after the EndIf are then processed.

```
ElseLines ::= Else Components | ElseIf Components ElseLines | empty
```

When the Elselines of a Conditional is being included then if an "Else Components" phrase is encountered then the following Components are included otherwise if an "ElseIf Components ElseLines" phrase is encountered then the boot expression following the)elseif is evaluated and if true the following Components are included, if false the following ElseLines is included.

2.3 Boot Syntax

2.3.1 Identifiers

The standard identifiers start with a letter (a-z or A-Z) dollar sign ($), question mark (?), or the percent sign (followed by any number of letters, digits, single quotes('), question marks, or percent signs. It is possible however, by using the escape character (_), to construct identifiers that contain any characters except the blank or newline character. The rules in this case are that an escape character followed by any non-blank character will start an identifier with that character. Once an identifier has been started either in this way or by a letter, $, or %, then it may be continued either with a letter, digit, ' , ? or %, or with an escape character followed by any non-blank character. Certain words having the form of identifiers are not classified as such, but are reserved words. They are listed below.

An identifier ends when a blank or end of line is encountered, or an escape character followed by a blank or end of line, or a character which is not a letter, digit, quote, question mark or percent sign is found. Two identifiers are equal if the strings produced by replacing each escape followed by a character by that character are equal character by character.

2.3.2 Numbers

Integers start with a digit (0-9) and are followed by any number of digits. The syntax for floating point numbers is

```
<.I | I. | I.I> <E|e> <+ | - | empty> I
```

where I is an integer.

2.3.3 Strings

Strings of characters are enclosed by double quote signs. They cannot span 2 or more lines and an escape character within a string will include the next character regardless of its nature. The meaning of a string depends somewhat on the context in which it is found, but in general a bare string denotes the interned atom making up its body whereas when it is preceded by a single quote (') it denotes the string of characters enclosed.

2.3.4 S-expressions

An s-expression is preceded by a single quote and is followed by a Lisp s-expression.

```
sexpression ::=identifier | integer | MINUS integer | float | string
            | QUOTE sexpression | parenthesized sexpression1

sexpression1 ::=sexpression (DOT sexpression | sexpression1)| empty
```

There are two ways to quote a name either 'name or "name", which both give rise to (QUOTE name). However a string that is a component of an sexpression will denote the string unless it is the sole component of the s-expression in which case it denotes a string i.e. '"name" gives rise to "name" in Lisp rather than (QUOTE "name").

2.3.5 Keywords

The table of key words follows, each is given an upper case name for use in the description of the syntax.

as written	name
and	AND
by	BY
cross	CROSS
else	ELSE
for	FOR
if	IF
in	IN
is	IS
isnt	ISNT
or	OR
repeat	REPEAT
return	RETURN
then	THEN
until	UNTIL
where	WHERE
while	WHILE
.	DOT
:	COLON
,	COMMA
;	SEMICOLON
*	TIMES
**	POWER
/	SLASH
+	PLUS
-	MINUS

```
<           LT
>           GT
<=          LE
>=          GE
=           EQ
^           NOT
^=          NE
..          SEG
#           SIZE
=>          EXIT
:=          BECOMES
==          DEF
==>         MDEF
(           OPAREN
)           CPAREN
[           OBRACK
(|          OBRACK
]           CBRACK
|)          CBRACK
 |          BAR
suchthat    BAR
```

2.3.6 Primary

```
constant::= integer | string | float | sexpression
```

The value of a constant does not depend on the context in which it is found.

```
primary::= name | constant | construct | block | tuple |  pile
```

The primaries are the simplest constituents of the language and either denote some object or perform some transformation of the machine state, or both. The statements are the largest constituents and enclosing them in parentheses converts them into a primary.

An alternative method of grouping uses indentation to indicate the parenthetical structure. A number of lines whose first non-space characters are in the same column will be called a "pile". The translater first tokenizes the lines producing identifier, key word, integer, string or float tokens, and then examines the pile structure of a Boot program in order to add additional tokens called SETTAB, BACKTAB and BACKSET. These tokens may be considered as commands for creating a pile. The SETTAB starts a new line indented from the previous line and pushes the resulting column number on to a stack of tab positions. The BACKTAB will start a new line at the column position found at the head of the stack and removes it from the stack. The BACKSET has the same effect as a BACKTAB immediately followed by a SETTAB. The meaning of a sequence of tokens containing SETTAB BACKTAB and BACKSET is the same the sequence in which each SETTAB is replaced by OPAREN , each BACKTAB is replaced by CPAREN, and each BACKSET is replaced by SEMICOLON. By construction the BACKTABS and SETTABS are properly nested.

```
listof(p,s)== p | p s ... s p

parenthesized s ::=   OPAREN s CPAREN
piled         s ::=   SETTAB s BACKTAB

blockof s ::=    parenthesized (listof (s,SEMICOLON))
pileof s  ::=    piled         (listof (s,BACKSET  ))
```

A pileof s has the same meaning as a blockof s. There is however a slight difference because piling is weaker than separation by semicolons. In other words the pile items may be listof(s,SEMICOLON). In other words if statements::= listof(statement,SEMICOLON) then we can have a pileof statements which has the same meaning as the flattened sequence formed by replacing all BACKSET's by SEMICOLON's.

A blockof statement is translated to a compound statement e.g. in the absence of any exits, (a;b;c;d) is translated to (PROGN a b c d).

2.3.7 Selectors

```
selector::= leftassociative(primary, DOT)
```

A selector a.b denotes some component of a structure, and in general is translated to (ELT a b). There are some special identifiers that may be used in the b position to denote list components, of which more later. The DOT has a greater precedence than juxtaposition and is left associative, For example

```
a.b.c  is grouped as (a.b).c which is translated to
  (ELT (ELT a b) c)

application ::= selector selector ... selector
```

Application of function to argument is denoted by juxtaposition.

A sequence of selectors is right associative and so f g h x is grouped as f(g(h x)). The applications f x and f(x) mean the application of f to x and get translated to the Lisp (f x). The application of a function to the empty list is written f(), meaning the Lisp (f). f(x,y,z) gets translated to the Lisp (f x y z). Common Lisp does not permit a variable to occur in operator position, so that when f is a variable its application has to be put in argument position of a FUNCALL or APPLY. f(x,y,z) has to be replaced by FUNCALL(f,x,y) which gets translated to the Lisp (FUNCALL f x y z). In Common Lisp each symbol might refer to two objects a function and a non-function. In order to resolve this ambiguity when a function symbol appears in a context other than operator position it has to be preceded by the symbol FUNCTION. Also it is possible to produce the function type symbol from the non-function symbol by applying SYMBOL-FUNCTION to it.

Certain reserved words called infixed operators namely POWER, TIMES, SLASH, PLUS MINUS ,IS, EQ , NE , GT , GE , LT , LE , IN , AND, OR, indicate application by being placed between their 2 arguments.

Infixed application may be either right- or left-associative.

```
rightassociative(p,o)::= p o p o p o ... o p
                    ==  p o (p o (p o ... o p)))

leftassociative(p,o)::= p o p o p o ... o p
                    ==  (((p o p) o p) o ...) o p
```

```
exponent ::= rightassociative(application,POWER)
```

```
reduction ::= (infixedoperator |string | thetaname) SLASH application
```

In a reduction the application denotes a list of items and operator SLASH application accumulates the list elements from the left using the operator

```
e.g.  +/[a,b,c] means (((0+a)+b)+c)
```

Only certain operators are provided with values when the list is empty they are and, or +, *, max, min, append, union. However any function can be used as an operator by enclosing it in double quotes. In this case the reduction is not applicable to an empty list.

```
multiplication ::= rightassociative(exponent,TIMES|SLASH) | reduction

minus ::= MINUS multiplication | multiplication

arith ::= leftasscociative(minus,PLUS | MINUS)

is ::= arith | arith (IS | ISNT) pattern

comparison ::= is (EQ | NE | GT | GE | LT | LE | IN) is | is

and  ::= leftassociative (comparison,AND)

return ::= and | RETURN and

expression ::= leftassociative(return,OR)
```

The infixed operators denote application of the function to its two arguments. To summarize, the infixed operators are, in order of decreasing precedence strengths.

```
.
juxtaposition
**
* /
+ -
is
= ^= > >= < <= in
and
or
```

2.3.8 Conditionals

```
conditional ::= IF where THEN where |
                IF where THEN where ELSE where
```

```
IF a THEN b is translated to (COND (a b)) and
IF a THEN b else c is translated to (COND (a b) (T c))
```

```
statement::= conditional | loop | expression
```

2.3.9 Loops

```
loop ::= crossproduct REPEAT statement | REPEAT statement

iterator ::= forin | suchthat | until | while

iterators ::= iterator iterator ... iterator

crossproduct ::=rightassociative(iterators,CROSS)

suchthat ::= BAR where

while ::= WHILE expression

until ::= UNTIL expression

forin ::= for variable IN segment |
          for variable IN segment BY arith

segment::= arith | arith SEG arith | arith SEG
```

A loop performs an iterated transformation of the state which is specified by its statement component and its iterators. The forin construction introduces a new variable which is assigned the elements of the list which is the value of the segment in the order in which they appear in the list .

A segment of the form arith denotes a list, and segments of the form "arith SEG arith" and "arith SEG" denote terminating and non-terminating arithmetic progressions. The "BY arith" option is the step size, if omitted the step is 1.

Two or more forin's may control a loop. The associated lists are scanned in parallel and a variable of one forin may not appear in the segment expression that denotes the list in a second forin. Such a variable may however occur in the conditions for filtering or introduced by a suchthat, or for termination introduced by a while iterator, and in the statement of the loop. The forin variables are local to the statement, the conditions that follow a while or suchthat in the same list of iterators and have no meaning outside them. The loop will be terminated when one of its forin lists is null, or if the condition in a while is not satisfied. The list elements are filtered by all the suchthat conditions. The ordering of the iterators is irrelevant to the meaning, so it is best to avoid side effects within the conditions for filtering and termination.

It is possible to control a loop by using a cross-product of iterators. The iteration in the case iterators1 CROSS iterators2 is over all pairs of list items one from the list denoted by iterators1 and the other from the list denoted by iterators2. In this case the variables introduced forin statements in iterators1 may be used in iterators2.

2.3.10 Lists

Boot contains a simple way of specifying lists that are constructed by CONS and APPEND, or by transforming one list to another in a systematic manner.

```
construct ::= OBRACK construction CBRACK

construction ::= comma | comma iteratortail

iteratortail ::= REPEAT iterators | iterators
```

A construct expression denotes a list and may also have a list of controlling iterators having the same syntax as a loop. In this case the expression is enclosed in brackets and the iterators follow the expression they qualify, rather than preceding it.

In the case that there are no iterators the construct expression denotes a list by listing its components separated by commas, or by a comma followed by a colon. In the simple case in which there are no colons the Boot expression [a,b,c,d] translates to the Lisp (LIST a b c d) or (CONS a (CONS b (CONS c (CONS d NIL)))).

When elements are separated by comma colon, however, the expression that follows will be assumed to denote a list which will be appended to the following list, rather than consed. An exception to this rule is that a colon preceding the last expression is translated to the expression itself. If it immediately preceded by a CONS then it need not denote a list.

For example:

```
[] is translated to the empty list NIL
```

```
[a] is translated to the 1-list (LIST a) or (CONS a NIL)
[:a] is translated to a
[a,b] is translated to the 2-list (LIST a b) or (CONS a (CONS b NIL))
[:a,b] is translated to (APPEND a (CONS b NIL))
[a,:b] is translated to (CONS a b)
[:a,:b] is translated to (APPEND a b)
[:a,b,c] is translated to (APPEND a (CONS b (CONS c NIL)))
[a,:b,c] is translated to (CONS a (APPEND b (CONS c NIL)))
[a,b,:c] is translated to (CONS a (CONS b c))
```

If the construct expression has iterators that control the production of the list the resulting list depends on the form of the comma expression. i.e.

```
construction ::= comma iteratortail
```

If the comma expression is recognised as denoting a list by either preceding it by a colon, or having commas at top level as above, then the successive values are appended. If not then the successive values are consed. e.g.

```
[f i for i in x] denotes the list formed by applying f to each
  member of the list x.

[:f i for i in 0..n] denotes the list formed by appending the
  lists f i for each i in 0..n.
```

2.3.11 Patterns

```
is ::= arith | arith IS  pattern
```

The pattern in the proposition "arith IS pattern" has the same form as the construct phrase without iterators. In this case, however it denotes a class of lists rather than a list, and is composed from identifiers rather than expressions. The proposition is translated into a program that tests whether the arith expression denotes a list that belongs to the class. If it does then the value of the is expression is true and the identifiers in the pattern are assigned the values of the corresponding components of the list. If the list does not match the pattern the value of the is expression is false and the values of the identifier might be changed in some unknown way that reflects the partial success of the matching. Because of this uncertainty, it is advisable to use the variables in a pattern as new definitions rather than assigning to variables that are defined elsewhere.

```
pattern::= identifier | constant | [ patternlist ]
```

The value of arith IS identifier is true and the value of arith is assigned to the identifier. (PROGN (SETQ identifier arith) T) The expression arith IS constant is translated to (EQUAL constant arith). The expression arith IS [pattenlist] produces a program which tests whether arith denotes a list of the right length and that each patternitem matches the corresponding list component.

```
patternitem ::= EQ application  | DOT | pattern | name := pattern
```

If the patternitem is EQ application then the value is true if the component is EQUAL to the value of the application expression. If the patternitem is DOT then the value is true regardless of the nature of the component. It is used as a place-holder to test whether the component exists. If the patternitem is pattern then the component is matched against the pattern as above. If the patternitem is name:=pattern then the component is matched against the pattern as above, and if the value is true the component is assigned to the name. This last provision enables both a component and its components to be given names.

```
patternlist ::= listof(patternitem,COMMA)|
                listof(patternitem,COMMA) COMMA patterntail
                patterntail

patterncolon ::= COLON patternitem

patterntail ::= patterncolon |
               patterncolon COMMA listof(patternitem,COMMA)
```

The patternlist may contain one colon to indicate that the following patternitem can match a list of any length. In this case the matching rule is to construct the expression with CONS and APPEND from the pattern as shown above and then test whether the list can be constructed in this way, and if so deduce the components and assign them to identifiers.

The effect of a pattern that occurs as a variable in a for iterator is to filter the list by the pattern.

```
forin ::= for pattern IN segment
```

is translated to two iterators

```
for g IN segment | g IS pattern
```

where g is an invented identifier.

```
forin ::= for (name:=pattern) IN segment
```

is translated to two iterators

```
for name IN segment BAR name IS pattern
```

in order to both filter the list elements, and name both elements and their components.

2.3.12 Assignments

A pattern may also occur on the left hand side of an assignment statement, and has a slightly different meaning. The purpose in this case is to give names to the components of the list which is the value of the right hand side. In this case no checking is done that the list matches the pattern precisely and the only effect is to construct the selectors that correspond to the identifiers in the pattern, apply them to the value of the right hand side and assign the selected components to the corresponding identifiers. The effect of applying CAR or CDR to arguments to which they are not applicable will depend on the underlying Lisp system.

```
assignment::= assignvariable BECOMES assignment| statement

assignvariable := OBRACK patternlist CBRACK | assignlhs
```

The assignment having a pattern as its left hand side is reduced as explained above to one or more assignments having an identifier on the left hand side. The meaning of the assignment depends on whether the identifier starts with a dollar sign or not, if it is and whether it is followed by :local or :fluid. If the identifier does not start with a dollar sign it is treated as local to the body of the function in which it occurs, and if it is not already an argument of the function, a declaration to that effect is added to the Lisp code by adding a PROG construction at top level within the body of the function definition.

If such an identifier assignment does not occur in the body of a function but in a top level expression then it is also treated as a local. The sole exception to this rule is when the top level expression is an assignment to an identifier in which case it is treated as global.

If the left hand side of an assignment is an identifier that starts with a dollar sign it will not be classified as a local but will be treated as non-local. If it is also followed by :local then it will be treated as a declaration of a FLUID (VMLisp) or SPECIAL variable (Common Lisp) which will be given an initial value which is the value of the right hand side of the assignment statement. The FLUID or SPECIAL variables may be referred to or assigned to by functions that are applied in the body of the declaration.

If the left hand side of an assignment statement is an identifier that does not start with a dollar sign followed by :local then it will also be treated as a FLUID or SPECIAL declaration, however it may only be assigned to in the body of the function in which the assignment it occurs.

```
assignment::= assignvariable BECOMES assignment | statement

assignvariable := OBRACK patternlist CBRACK | assignlhs

assignlhs::= name | name COLON local |
     name DOT primary DOT ... DOT primary
```

If the left hand side of an assignment has the form

```
name DOT primary DOT ... DOT primary
```

the assignment statement will denote an updating of some component of the value of name. In general name DOT primary := statement will get translated to (SETELT name primary statement) or (SETF (ELT name primary) statement) There are however certain identifiers that denote components of a list which will get translated to statements that update that component (see appendix) e.g.

```
a.car:=b is translated to (SETF (CAR a) b) in Common Lisp.
```

The iterated DOT is used to update components of components and e.g

```
a.b.c:=d is translated to (SETF (ELT (ELT a b)c) d)
```

```
exit::= assignment | assignment EXIT where
```

The exit format "assignment EXIT where" is used to give a value to a blockof or pileof statements in which it occurs at top level.

The expression

```
(a =>b;c) will be translated to if a then b else c or
(COND (a b) (T c))
```

If the exit is not a component of a blockof or pileof statements then

```
a=>b will be translated to (COND (a b))
```

2.3.13 Definitions

Functions may be defined using the syntax

```
functiondefinition::= name DEF where | name variable DEF where


variable ::= parenthesized variablelist | pattern

variableitem ::=
     name| pattern | name BECOMES pattern | name IS pattern

variablelist ::= variableitem | COLON name |
             variableitem COMMA variablelist
```

Function definitions may only occur at to level or after a where. The name is the name of the function being defined, and the most frequently used form of the variable is either a single name or a parenthesized list of names separated by commas. In this case the translation to Lisp is straightforward, for example:

```
f x == E  or f(x)==E is translated to (DEFUN f (x) TE)
f (x,y,z)==E is translated to (DEFUN f (x y z) TE)
f ()==E is translated to (DEFUN f () TE)
```

where TE is the translation of E. At top level

```
f==E is translated to (DEFUN f () TE)
```

The function being defined is that which when applied to its arguments produces the value of the body as result where the variables in the body take on the values of its arguments.

A pattern may also occur in the variable of a definition of a function and serves the purpose, similar to the left hand side of assignments, of naming the list components. The phrase

```
 name pattern DEF where
is translated to
        name g DEF (pattern:=g;where)
```

similarly

```
 name1 name2 := pattern  DEF where  or name1 name2 is pattern  DEF where

are both translated to
        name1 name2 DEF (pattern:=name2;where)
```

similarly for patterns that occur as components of a list of variables. order

```
variablelist ::=
  variableitem | COLON name | variableitem COMMA variablelist
```

The parenthesized variablelist that occurs as a variable of a function definition can contain variables separated by commas but can also have a comma colon as its last separator.

This means that the function is applicable to lists of different sizes and that only the first few elements corresponding to the variables separated by commas are named, and the last name after the colon denotes the rest of the list.

Macros may be defined only at top level, and must always have a variable

```
macrodefinition::=  name variable MDEF where
```

The effect of a macrodefinition is to produce a Lisp macro which is applied to arguments that are treated as expressions, rather than their values, and whose result if formed by first substituting the expressions for occurrences of the variables within the body and then evaluating the resulting expression.

2.3.14 Where Clauses

Expressions may be qualified by one or more function definitions using the syntax

```
where ::= exit | exit WHERE qualifier

qualifier ::= functiondefinition |
      pileof (functiondefinition) | blockof functiondefinition
```

The functions may only be used within the expression that is qualified. This feature has to be used with some care, however, because a where clause may only occur within a function body, and the component functions are extruded, so to speak, from their contexts renamed, and made into top level function definitions. As a result the variables of the outer function cannot be referred to within the inner function. If a qualifying function has the format ”name DEF where” then the where phrase is substituted for all occurences of the name within the expression qualified. If an expression is qualified by a phrase that is not a function definition then the result will be a compound statement in which the qualifying phrase is followed by the qualified phrase.

2.3.15 Tuples

Although a tuple may appear syntactically in any position occupied by a primary it will only be given meaning when it is the argument to a function. To denote a list it has to be enclosed in brackets rather than parentheses. A tuple at top level is treated as if its components appeared at top level in the order of the list.

```
tuple::=   parenthesized (listof (where,COMMA))
```

2.3.16 Blocks and Piles

```
block::=   parenthesized (listof (where,SEMICOLON))
pile::=    piled (listof (listof(where,SEMICOLON),BACKSET))
A block or a pile get translated to a compound statement or PROGN
```

2.3.17 Top Level

```
toplevel ::=  functiondefinition | macrodefinition | primary
```

2.3.18 Translation Functions

```
(boottocl "filename")
translates the file "filename.boot" to
the common lisp file "filename.clisp"
```

```
(bootclam "filename")
translates the file "filename.boot" to
the common lisp file "filename.clisp"
```

producing, for each function a hash table to store previously computed values indexed by argument list. The function first looks in the hash table for the result if there returns it, if not computes the result and stores it in the table.

```
(boottoclc "filename")
translates the file "filename.boot" to
the common lisp file "filename.oclisp"
with the original boot code as comments
```

```
(boot "filename")
translates the file "filename.boot" to
the common lisp file "filename.clisp",
compiles it to the file  "filename.bbin"
and loads  the bbin file.

(bo "filename")
translates the file "filename.boot"
and prints the result at the console

(stout "string") translates the string "string"
and prints the result at the console

(sttomc "string") translates the string "string"
to common lisp, and compiles the result.

(fc "functionname" "filename")
attempts to find the boot function
functionname in the file filename,
if found it translates it to common
lisp, compiles and loads it.

BOOT_-COMPILE_-DEFINITION_-FROM_-FILE(fn,symbol)
 is similar to fc, fn is the file name but symbol is the  symbol
 of the function name rather than the string.
(fn,symbol)

BOOT_-EVAL_-DEFINITION_-FROM_-FILE(fn,symbol)
attempts to find the definition of symbol in file fn, but this time
translation is followed by EVAL rather than COMPILE

(defuse "filename")
Translates the file filename, and writes a report of the
functions defined and not used, and used and not defined in the
file filename.defuse

(xref "filename")
Translates the file filename, and writes a report of the
names  used, and  where used to the file filename.xref
```

3 The Makefile

There is a directory called IN that contains the boot, clisp and lisp code. This code gets copied into the intermediate directory `${INT}/boot` so that the compiler has a writeable directory to change. This intermediate code is then compiled into the `${OUT}` directory (which is in the object directory for the type of machine code we are constructing (rt, ps2, rios)) then, when all of the .o files are produced we construct a lisp image that contains the boot translator, called bootsys, which lives in the mnt/sys/bin directory. this bootsys image is critical for the rest of the makefiles to succeed.

There are two halves of this file. the first half compiles the .lisp files that live in the src/boot directory. the second half compiles the .clisp files (which are generated from the .boot files). It is important that the .clisp files are kept in the src/boot directory for the boot translator as they cannot be recreated without a boot translator (a bootstrap problem).

An important subtlety is that files in the boot translator depend on the file npextras. there are 3 macros in npextras that must be in the lisp workspace (`|shoeOpenInputFile| |shoeOpenOutputFile| memq`).

3.1 Directories

⟨*virtual directories*⟩≡

```
IN=${SRC}/boot
MID=${INT}/boot
OUT=${OBJ}/${SYS}/boot
DOC=${MNT}/${SYS}/doc/src/boot
```

3.2 Environment

3.2.1 Lisp Images

We will use create and use several lisp images during the build process. We name them here for convenience.

The **LISPSYS** image we will use to compile the common lisp code to machine code.

The **BOOTSYS** image we need to run to translate boot files to common lisp files.

The **LOADSYS** image is a clean lisp image that we load with compiled files to be saved.

The **SAVESYS** image is the final name we will use as the result of this Makefile.

Since this is a boot-strapping process we are in one of two possible states. Either **BOOTSYS** exists and we can use it immediately or we need to construct it and then use it. You'll notice that **BOOTSYS** and **SAVESYS** correspond to these two cases.

⟨*environment*⟩≡

```
LISPSYS= ${OBJ}/${SYS}/bin/lisp
BOOTSYS= ${OBJ}/${SYS}/bin/bootsys
LOADSYS= ${OBJ}/${SYS}/bin/lisp
SAVESYS= ${OBJ}/${SYS}/bin/bootsys
```

3.3 The list of files

This is a list of all of the files that must be loaded to construct the boot translator image. Note that the order is important as earlier files will contain code needed by later files.

Note that the **OBJS1** list differs from the **OBJS** list only in that the first has each name surrounded by quotes. The **OBJS1** list is fed to lisp and the **OBJS** list is used by make. Keep these exactly in sync, including ordering, unless you have a *very* good reason.

⟨*environment*⟩+≡

```
OBJS1=  "${OUT}/boothdr.${O}" "${OUT}/exports.${O}" \
        "${OUT}/npextras.${O}" "${OUT}/ptyout.${O}" \
        "${OUT}/btincl2.${O}" "${OUT}/btscan2.${O}" \
        "${OUT}/typrops.${O}" "${OUT}/btpile2.${O}" \
        "${OUT}/typars.${O}" "${OUT}/tyextra.${O}" \
        "${OUT}/tytree1.${O}"
OBJS=   ${OUT}/boothdr.${O} ${OUT}/exports.${O} \
        ${OUT}/npextras.${O} ${OUT}/ptyout.${O} \
        ${OUT}/btincl2.${O} ${OUT}/btscan2.${O} \
        ${OUT}/typrops.${O} ${OUT}/btpile2.${O} \
        ${OUT}/typars.${O} ${OUT}/tyextra.${O} \
        ${OUT}/tytree1.${O}
```

3.3.1 The Bootstrap files

BOOTS is a list of the boot file targets. If you modify a boot file you'll have to explicitly build the clisp files and merge the generated code back into the pamphlet by hand. The assumption is that if you know enough to change the fundamental bootstrap files you know how to migrate the changes back. This process, by design, does not occur automatically (though it could).

The files in the **BOOTS** list are needed to bootstrap the boot parser. They are written in boot. Invoking this Makefile with

```
make boot BOOTSYS=(path to bootsys) BYE=(lisp exit fn)
```

will run notangle on the pamphlet files to create the raw boot files. The raw boot files are translated in a **bootsys** image using the **boottocl** function to create the raw clisp files.

⟨*environment*⟩+≡

```
BOOTS=ptyout.boot btincl2.boot typrops.boot btpile2.boot \
      typars.boot tyextra.boot trtree1.boot
```

4 Proclaim optimization

GCL, and possibly other common lisps, can generate much better code if the function argument types and return values are proclaimed.

In theory what we should do is scan all of the functions in the system and create a file of proclaim definitions. These proclaim definitions should be loaded into the image before we do any compiles so they can allow the compiler to optimize function calling.

GCL has an approximation to this scanning which we use here.

The first step is to build a version of GCL that includes gcl_collectfn. This file contains code that enhances the lisp compiler and creates a hash table of structs. Each struct in the hash table describes information that about the types of the function being compiled and the types of its arguments. At the end of the compile-file this hash table is written out to a ".fn" file.

The second step is to build axiom images (depsys, interpsys, AXIOMsys) which contain the gcl_collectfn code.

The third step is to build the system. This generates a .fn file for each lisp file that gets compiled.

The fourth step is to build the proclaims.lisp files. There is one proclaims.lisp file for boot (boot-proclaims.lisp), interp (interp-proclaims.lisp), and algebra (algebra-proclaims.lisp).

To build the proclaims file (e.g. for interp) we:

```
(a) cd to obj/linux/interp
(b) (yourpath)/axiom/obj/linux/bin/lisp
(c) (load "sys-pkg.lsp")
(d) (mapcar #'load (directory "*.fn"))
(e) (with-open-file (out "interp-proclaims.lisp" :direction :output)
      (compiler::make-proclaims out))
```

Note that step (c) is only used for interp, not for boot.

The fifth step is to copy the newly constructed proclaims file back into the src/interp diretory (or boot, algebra).

In order for this information to be used during compiles we define

⟨*environment*⟩+≡
```
PROCLAIMS=(load "${IN}/boot-proclaims.lisp")
```

5 Special Commands

We are working in a build environment that combines Makefile technology with Lisp technology. Instead of invoking a command like **gcc** and giving it arguments we will be creating Lisp S-expressions and piping them into a Lisp image. The Lisp image starts, reads the S-expression from standard input, evaluates it, and finding an end-of-stream on standard input, exits.

There are two special S-expressions that we collect here. The first, **DEPS** will load all of the macros and functions needed to compile a common lisp file that was translated from boot. These are collected into the **npextras.lisp** [6]) file.

Note that the ' symbol should not appear in this S-expression because the **DEPS** command will be expanded into a shell echo command and it will be surrounded by single quotes at the expansion. Adding a single quote symbol will break this expansion.

⟨*environment*⟩+≡
```
DEPS=   (load (quote $(patsubst %, "%", ${MID}/npextras.lisp)))
```

The **CMD0** S-expression is the command line we use to create the final **SAVESYS** image. This S-expression will be given to a clean lisp image, loaded with the compiled files, and saved.

Note that the ' symbol should not appear in this S-expression because the **CMD0** command will be expanded into a shell echo command and it will be surrounded by single quotes at the expansion. Adding a single quote symbol will break this expansion.

The environment variable used to read:

```
CMD0=   (progn (mapcar (function (lambda (x) (load  x))) (quote (${OBJS1}))) (system::save-system "${SAVESYS}"))
```

but has been changed to allow dynamic linking. The issue is that some versions of the loader on some systems cannot save dynamically loaded code. The new `CMD0` string is rumored to work with both loaders.

In fact, the new `CMD0` reads:

```
CMD0=   (compiler::link (quote (${OBJS1})) "${SAVESYS}" (format nil "(let ((*load-path* (cons ~S *load-path*))(si::*load-types* ~S)) (compiler::emit-fn t)) (when (fboundp (quote si::sgc-on)) (si::sgc-on t)) (setq compiler::*default-system-p* t)" si::*system-directory* (quote  (list ".lsp"))))
```

and it does not work. The failure is a series of messages:

```
/home/axiom--main--1--patch-33/obj/linux/boot/exports.o(.text+0x0): In function ‘init_code’:
: multiple definition of ‘init_code’
/home/axiom--main--1--patch-33/obj/linux/boot/boothdr.o(.text+0x0): first defined here
```

and the problem appears to be that each `.o` file has `init_code`.

Until this is fixed we need to continue to use the old scheme.

⟨*environment*⟩+≡
```
CMD0=   (progn (mapcar (function (lambda (x) (load  x))) (quote (${OBJS1}))) (system::save-system "${SAVESYS}"))
```

5.1 boothdr.lisp [1]

⟨*boothdr.o (OUT from MID)*⟩≡
```
${OUT}/boothdr.${O}: ${MID}/boothdr.lisp
        @ echo 1 making ${OUT}/boothdr.${O} from ${MID}/boothdr.lisp
        @ ( cd ${MID} ; \
          if [ -z "${NOISE}" ] ; then \
           echo '(progn ${PROCLAIMS} (compile-file "boothdr.lisp" :output-file "${OUT}/boothdr.${O}") (${BYE}))' | ${LISPSYS} ; \
          else \
           echo '(progn ${PROCLAIMS} (compile-file "boothdr.lisp" :output-file "${OUT}/boothdr.${O}") (${BYE}))' | ${LISPSYS} >${TMP}/trace ; \
          fi )
```

⟨*boothdr.lisp (MID from IN)*⟩≡
```
${MID}/boothdr.lisp: ${IN}/boothdr.lisp.pamphlet
        @echo 2 making ${MID}/boothdr.lisp from ${IN}/boothdr.lisp.pamphlet
        @${TANGLE} boothdr.lisp.pamphlet >${MID}/boothdr.lisp
```

⟨*boothdr.lisp.dvi (DOC from IN)*⟩≡
```
${DOC}/boothdr.lisp.dvi: ${IN}/boothdr.lisp.pamphlet ${DOC}/axiom.sty
        @echo 3 making ${DOC}/boothdr.lisp.dvi from ${IN}/boothdr.lisp.pamphlet
        @(cd ${DOC} ; \
        cp ${IN}/boothdr.lisp.pamphlet ${DOC} ; \
        ${DOCUMENT} ${NOISE} boothdr.lisp ; \
        rm -f ${DOC}/boothdr.lisp.pamphlet ; \
        rm -f ${DOC}/boothdr.lisp.tex ; \
        rm -f ${DOC}/boothdr.lisp )
```

5.2 btincl2.lisp [2]

Notice that the progn special form that wraps the compile-file function call executes **DEPS** first. The **DEPS** variable expands into an s-expression that will load **npextras.lisp**. This file contains macros needed at compile time.

⟨*btincl2.o (OUT from MID)*⟩≡
```
${OUT}/btincl2.${O}: ${MID}/btincl2.lisp
        @ echo 4 making ${OUT}/btincl2.${O} from ${MID}/btincl2.lisp
        @ ( cd ${MID} ; \
          if [ -z "${NOISE}" ] ; then \
           echo '(progn ${PROCLAIMS} ${DEPS} (compile-file "btincl2.lisp" :output-file "${OUT}/btincl2.${O}") (${BYE}))' | ${LISPSYS} ; \
          else \
           echo '(progn ${PROCLAIMS} ${DEPS} (compile-file "btincl2.lisp" :output-file "${OUT}/btincl2.${O}") (${BYE}))' | ${LISPSYS} >${TMP}/trace ; \
          fi )
```

⟨*btincl2.lisp (MID from IN)*⟩≡

```
${MID}/btincl2.lisp: ${IN}/btincl2.boot.pamphlet
        @echo 5 making ${MID}/btincl2.lisp from ${IN}/btincl2.boot.pamphlet
        @${TANGLE} -Rbtincl2.clisp btincl2.boot.pamphlet >${MID}/btincl2.lisp
```

⟨*btincl2.boot.dvi (DOC from IN)*⟩≡

```
${DOC}/btincl2.boot.dvi: ${IN}/btincl2.boot.pamphlet ${DOC}/axiom.sty
        @echo 6 making ${DOC}/btincl2.lisp.dvi from \
           ${IN}/btincl2.boot.pamphlet
        @(cd ${DOC} ; \
        cp ${IN}/btincl2.boot.pamphlet ${DOC} ; \
        ${DOCUMENT} ${NOISE} btincl2.boot ; \
        rm -f ${DOC}/btincl2.boot.pamphlet ; \
        rm -f ${DOC}/btincl2.boot.tex ; \
        rm -f ${DOC}/btincl2.boot )
```

⟨*btincl2.boot (IN from IN)*⟩≡

```
btincl2.boot: btincl2.boot.pamphlet
        @echo 7 making btincl2.boot from btincl2.boot.pamphlet
        @( ${TANGLE} btincl2.boot.pamphlet >btincl2.boot ; \
          if [ -z "${NOISE}" ] ; then \
           echo '(progn (boottran::boottocl "btincl2.boot") (${BYE}))' | ${BOOTSYS} ; \
          else \
           echo '(progn (boottran::boottocl "btincl2.boot") (${BYE}))' | ${BOOTSYS} >${TMP}/trace ; \
          fi )
```

5.3 btpile2.boot [3]

Notice that the progn special form that wraps the compile-file function call executes **DEPS** first. The **DEPS** variable expands into an s-expression that will load **npextras.lisp**. This file contains macros needed at compile time.

⟨*btpile2.o (OUT from MID)*⟩≡

```
${OUT}/btpile2.${O}: ${MID}/btpile2.lisp
        @ echo 8 making ${OUT}/btpile2.${O} from ${MID}/btpile2.lisp
        @ ( cd ${MID} ; \
          if [ -z "${NOISE}" ] ; then \
           echo '(progn ${PROCLAIMS} ${DEPS} (compile-file "btpile2.lisp" :output-file "${OUT}/btpile2.${O}") (${BYE}))' | ${LISPSYS} ; \
          else \
           echo '(progn ${PROCLAIMS} ${DEPS} (compile-file "btpile2.lisp" :output-file "${OUT}/btpile2.${O}") (${BYE}))' | ${LISPSYS} >${TMP}/trace ; \
          fi )
```

⟨*btpile2.lisp (MID from IN)*⟩≡

```
${MID}/btpile2.lisp: ${IN}/btpile2.boot.pamphlet
	@echo 9 making ${MID}/btpile2.lisp from ${IN}/btpile2.boot.pamphlet
	@${TANGLE} -Rbtpile2.clisp btpile2.boot.pamphlet >${MID}/btpile2.lisp
```

⟨*btpile2.boot.dvi (DOC from IN)*⟩≡

```
${DOC}/btpile2.boot.dvi: ${IN}/btpile2.boot.pamphlet ${DOC}/axiom.sty
	@echo 10 making ${DOC}/btpile2.boot.dvi from \
	   ${IN}/btpile2.boot.pamphlet
	@(cd ${DOC} ; \
	cp ${IN}/btpile2.boot.pamphlet ${DOC} ; \
	${DOCUMENT} ${NOISE} btpile2.boot ; \
	rm -f ${DOC}/btpile2.boot.pamphlet ; \
	rm -f ${DOC}/btpile2.boot.tex ; \
	rm -f ${DOC}/btpile2.boot )
```

⟨*btpile2.boot (IN from IN)*⟩≡

```
btpile2.boot: btpile2.boot.pamphlet
	@echo 11 making btpile2.boot from btpile2.boot.pamphlet
	@( ${TANGLE} btpile2.boot.pamphlet >btpile2.boot ; \
	  if [ -z "${NOISE}" ] ; then \
	   echo '(progn (boottran::boottocl "btpile2.boot") (${BYE}))' | ${BOOTSYS} ; \
	  else \
	   echo '(progn (boottran::boottocl "btpile2.boot") (${BYE}))' | ${BOOTSYS} >${TMP}/trace ; \
	  fi )
```

5.4 btscan2.boot [4]

Notice that the progn special form that wraps the compile-file function call executes **DEPS** first. The **DEPS** variable expands into an s-expression that will load **npextras.lisp**. This file contains macros needed at compile time.

⟨*btscan2.o (OUT from MID)*⟩≡

```
${OUT}/btscan2.${O}: ${MID}/btscan2.lisp
	@ echo 12 making ${OUT}/btscan2.${O} from ${MID}/btscan2.lisp
	@ ( cd ${MID} ; \
	  if [ -z "${NOISE}" ] ; then \
	   echo '(progn ${PROCLAIMS} ${DEPS} (compile-file "btscan2.lisp" :output-file "${OUT}/btscan2.${O}") (${BYE}))' | ${LISPSYS} ; \
	  else \
	   echo '(progn ${PROCLAIMS} ${DEPS} (compile-file "btscan2.lisp" :output-file "${OUT}/btscan2.${O}") (${BYE}))' | ${LISPSYS} >${TMP}/trace ; \
	  fi )
```

⟨*btscan2.lisp (MID from IN)*⟩≡
```
${MID}/btscan2.lisp: ${IN}/btscan2.boot.pamphlet
	@echo 13 making ${MID}/btscan2.lisp from ${IN}/btscan2.boot.pamphlet
	@${TANGLE} -Rbtscan2.clisp btscan2.boot.pamphlet >${MID}/btscan2.lisp
```

⟨*btscan2.boot.dvi (DOC from IN)*⟩≡
```
${DOC}/btscan2.boot.dvi: ${IN}/btscan2.boot.pamphlet ${DOC}/axiom.sty
	@echo 14 making ${DOC}/btscan2.boot.dvi from \
	   ${IN}/btscan2.boot.pamphlet
	@(cd ${DOC} ; \
	cp ${IN}/btscan2.boot.pamphlet ${DOC} ; \
	${DOCUMENT} ${NOISE} btscan2.boot ; \
	rm -f ${DOC}/btscan2.boot.pamphlet ; \
	rm -f ${DOC}/btscan2.boot.tex ; \
	rm -f ${DOC}/btscan2.boot )
```

⟨*btscan2.boot (IN from IN)*⟩≡
```
btscan2.boot: btscan2.boot.pamphlet
	@echo 15 making btscan2.boot from btscan2.boot.pamphlet
	@( ${TANGLE} btscan2.boot.pamphlet >btscan2.boot ; \
	  if [ -z "${NOISE}" ] ; then \
	   echo '(progn (boottran::boottocl "btscan2.boot") (${BYE}))' | ${BOOTSYS} ; \
	  else \
	   echo '(progn (boottran::boottocl "btscan2.boot") (${BYE}))' | ${BOOTSYS} >${TMP}/trace ; \
	  fi )
```

5.5 exports.lisp [5]

⟨*exports.o (OUT from MID)*⟩≡
```
${OUT}/exports.${O}: ${MID}/exports.lisp
	@ echo 16 making ${OUT}/exports.${O} from ${MID}/exports.lisp
	@ ( cd ${MID} ; \
	  if [ -z "${NOISE}" ] ; then \
	   echo '(progn ${PROCLAIMS} (compile-file "exports.lisp" :output-file "${OUT}/exports.${O}") (${BYE}))' | ${LISPSYS} ; \
	  else \
	   echo '(progn ${PROCLAIMS} (compile-file "exports.lisp" :output-file "${OUT}/exports.${O}") (${BYE}))' | ${LISPSYS} >${TMP}/trace ; \
	  fi )
```

⟨*exports.lisp (MID from IN)*⟩≡
```
${MID}/exports.lisp: ${IN}/exports.lisp.pamphlet
	@echo 17 making ${MID}/exports.lisp from ${IN}/exports.lisp.pamphlet
	@${TANGLE} exports.lisp.pamphlet >${MID}/exports.lisp
```

⟨*exports.lisp.dvi (DOC from IN)*⟩≡

```
${DOC}/exports.lisp.dvi: ${IN}/exports.lisp.pamphlet ${DOC}/axiom.sty
	@echo 18 making ${DOC}/exports.lisp.dvi from \
	    ${IN}/exports.lisp.pamphlet
	@(cd ${DOC} ; \
	cp ${IN}/exports.lisp.pamphlet ${DOC} ; \
	${DOCUMENT} ${NOISE} exports.lisp ; \
	rm -f ${DOC}/exports.lisp.pamphlet ; \
	rm -f ${DOC}/exports.lisp.tex ; \
	rm -f ${DOC}/exports.lisp )
```

5.6 npextras.lisp [6]

⟨*npextras.o (OUT from MID)*⟩≡

```
${OUT}/npextras.${O}: ${MID}/npextras.lisp
	@ echo 19 making ${OUT}/npextras.${O} from ${MID}/npextras.lisp
	@ ( cd ${MID} ; \
	  if [ -z "${NOISE}" ] ; then \
	   echo '(progn ${PROCLAIMS} (compile-file "npextras.lisp" :output-file "${OUT}/npextras.${O}") (${BYE}))' | ${LISPSYS} ; \
	  else \
	   echo '(progn ${PROCLAIMS} (compile-file "npextras.lisp" :output-file "${OUT}/npextras.${O}") (${BYE}))' | ${LISPSYS} >${TMP}/trace ; \
	  fi )
```

⟨*npextras.lisp (MID from IN)*⟩≡

```
${MID}/npextras.lisp: ${IN}/npextras.lisp.pamphlet
	@echo 20 making ${MID}/npextras.lisp from ${IN}/npextras.lisp.pamphlet
	@${TANGLE} npextras.lisp.pamphlet >${MID}/npextras.lisp
```

⟨*npextras.lisp.dvi (DOC from IN)*⟩≡

```
${DOC}/npextras.lisp.dvi: ${IN}/npextras.lisp.pamphlet ${DOC}/axiom.sty
	@echo 21 making ${DOC}/npextras.lisp.dvi from \
	       ${IN}/npextras.lisp.pamphlet
	@(cd ${DOC} ; \
	cp ${IN}/npextras.lisp.pamphlet ${DOC} ; \
	${DOCUMENT} ${NOISE} npextras.lisp ; \
	rm -f ${DOC}/npextras.lisp.pamphlet ; \
	rm -f ${DOC}/npextras.lisp.tex ; \
	rm -f ${DOC}/npextras.lisp )
```

5.7 ptyout.lisp [7]

Notice that the progn special form that wraps the compile-file function call executes **DEPS** first. The **DEPS** variable expands into an s-expression that will load **npextras.lisp**. This file contains macros needed at compile time.

⟨*ptyout.o (OUT from MID)*⟩≡
```
${OUT}/ptyout.${O}: ${MID}/ptyout.lisp
	@ echo 22 making ${OUT}/ptyout.${O} from ${MID}/ptyout.lisp
	@ ( cd ${MID} ; \
	  if [ -z "${NOISE}" ] ; then \
	   echo '(progn ${PROCLAIMS} ${DEPS} (compile-file "ptyout.lisp" :output-file "${OUT}/ptyout.${O}") (${BYE}))' | ${LISPSYS} ; \
	  else \
	   echo '(progn ${PROCLAIMS} ${DEPS} (compile-file "ptyout.lisp" :output-file "${OUT}/ptyout.${O}") (${BYE}))' | ${LISPSYS} >${TMP}/trace ; \
	  fi )
```

⟨*ptyout.lisp (MID from IN)*⟩≡
```
${MID}/ptyout.lisp: ${IN}/ptyout.boot.pamphlet
	@echo 23 making ${MID}/ptyout.lisp from ${IN}/ptyout.boot.pamphlet
	@${TANGLE} -Rptyout.clisp ptyout.boot.pamphlet >${MID}/ptyout.lisp
```

⟨*ptyout.boot.dvi (DOC from IN)*⟩≡
```
${DOC}/ptyout.boot.dvi: ${IN}/ptyout.boot.pamphlet ${DOC}/axiom.sty
	@echo 24 making ${DOC}/ptyout.boot.dvi from ${IN}/ptyout.boot.pamphlet
	@(cd ${DOC} ; \
	cp ${IN}/ptyout.boot.pamphlet ${DOC} ; \
	${DOCUMENT} ${NOISE} ptyout.boot ; \
	rm -f ${DOC}/ptyout.boot.pamphlet ; \
	rm -f ${DOC}/ptyout.boot.tex ; \
	rm -f ${DOC}/ptyout.boot )
```

⟨*ptyout.boot (IN from IN)*⟩≡
```
ptyout.boot: ptyout.boot.pamphlet
	@echo 25 making ptyout.boot from ptyout.boot.pamphlet
	@( ${TANGLE} ptyout.boot.pamphlet >ptyout.boot ; \
	  if [ -z "${NOISE}" ] ; then \
	   echo '(progn (boottran::boottocl "ptyout.boot") (${BYE}))' | ${BOOTSYS} ; \
	  else \
	   echo '(progn (boottran::boottocl "ptyout.boot") (${BYE}))' | ${BOOTSYS} >${TMP}/trace ; \
	  fi )
```

5.8 tyextra.lisp [8]

Notice that the progn special form that wraps the compile-file function call executes **DEPS** first. The **DEPS** variable expands into an s-expression that will load **npextras.lisp**. This file contains macros needed at compile time.

⟨*tyextra.o (OUT from MID)*⟩≡
```
${OUT}/tyextra.${O}: ${MID}/tyextra.lisp
	@ echo 26 making ${OUT}/tyextra.${O} from ${MID}/tyextra.lisp
	@ ( cd ${MID} ; \
	  if [ -z "${NOISE}" ] ; then \
	   echo '(progn ${PROCLAIMS} ${DEPS} (compile-file "tyextra.lisp" :output-file "${OUT}/tyextra.${O}") (${BYE}))' | ${LISPSYS} ; \
	  else \
	   echo '(progn ${PROCLAIMS} ${DEPS} (compile-file "tyextra.lisp" :output-file "${OUT}/tyextra.${O}") (${BYE}))' | ${LISPSYS} >${TMP}/trace ; \
	  fi )
```

⟨*tyextra.lisp (MID from IN)*⟩≡
```
${MID}/tyextra.lisp: ${IN}/tyextra.boot.pamphlet
	@echo 27 making ${MID}/tyextra.lisp from ${IN}/tyextra.boot.pamphlet
	@${TANGLE} -Rtyextra.clisp tyextra.boot.pamphlet >${MID}/tyextra.lisp
```

⟨*tyextra.boot.dvi (DOC from IN)*⟩≡
```
${DOC}/tyextra.boot.dvi: ${IN}/tyextra.boot.pamphlet ${DOC}/axiom.sty
	@echo 28 making ${DOC}/tyextra.boot.dvi from \
	  ${IN}/tyextra.boot.pamphlet
	@(cd ${DOC} ; \
	cp ${IN}/tyextra.boot.pamphlet ${DOC} ; \
	${DOCUMENT} ${NOISE} tyextra.boot ; \
	rm -f ${DOC}/tyextra.boot.pamphlet ; \
	rm -f ${DOC}/tyextra.boot.tex ; \
	rm -f ${DOC}/tyextra.boot )
```

⟨*tyextra.boot (IN from IN)*⟩≡
```
tyextra.boot: tyextra.boot.pamphlet
	@echo 29 making tyextra.boot from tyextra.boot.pamphlet
	@( ${TANGLE} tyextra.boot.pamphlet >tyextra.boot ; \
	  if [ -z "${NOISE}" ] ; then \
	   echo '(progn (boottran::boottocl "tyextra.boot") (${BYE}))' | ${BOOTSYS} ; \
	  else \
	   echo '(progn (boottran::boottocl "tyextra.boot") (${BYE}))' | ${BOOTSYS} >${TMP}/trace ; \
	 fi )
```

5.9 typars.lisp [9]

Notice that the progn special form that wraps the compile-file function call executes **DEPS** first. The **DEPS** variable expands into an s-expression that will load **npextras.lisp**. This file contains macros needed at compile time.

⟨*typars.o (OUT from MID)*⟩≡
```
${OUT}/typars.${O}: ${MID}/typars.lisp
	@ echo 30 making ${OUT}/typars.${O} from ${MID}/typars.lisp
	@ ( cd ${MID} ; \
	  if [ -z "${NOISE}" ] ; then \
	   echo '(progn ${PROCLAIMS} ${DEPS} (compile-file "typars.lisp" :output-file "${OUT}/typars.${O}") (${BYE}))' | ${LISPSYS} ; \
	  else \
	   echo '(progn ${PROCLAIMS} ${DEPS} (compile-file "typars.lisp" :output-file "${OUT}/typars.${O}") (${BYE}))' | ${LISPSYS} >${TMP}/trace ; \
	  fi )
```

⟨*typars.lisp (MID from IN)*⟩≡
```
${MID}/typars.lisp: ${IN}/typars.boot.pamphlet
	@echo 31 making ${MID}/typars.lisp from ${IN}/typars.boot.pamphlet
	@${TANGLE} -Rtypars.clisp typars.boot.pamphlet >${MID}/typars.lisp
```

⟨*typars.boot.dvi (DOC from IN)*⟩≡
```
${DOC}/typars.boot.dvi: ${IN}/typars.boot.pamphlet ${DOC}/axiom.sty
	@echo 32 making ${DOC}/typars.boot.dvi from ${IN}/typars.boot.pamphlet
	@(cd ${DOC} ; \
	cp ${IN}/typars.boot.pamphlet ${DOC} ; \
	${DOCUMENT} ${NOISE} typars.boot ; \
	rm -f ${DOC}/typars.boot.pamphlet ; \
	rm -f ${DOC}/typars.boot.tex ; \
	rm -f ${DOC}/typars.boot )
```

⟨*typars.boot (IN from IN)*⟩≡
```
typars.boot: typars.boot.pamphlet
	@echo 33 making typars.boot from typars.boot.pamphlet
	@( ${TANGLE} typars.lisp.pamphlet >typars.boot ; \
	  if [ -z "${NOISE}" ] ; then \
	   echo '(progn (boottran::boottocl "typars.boot") (${BYE}))' | ${BOOTSYS} ; \
	  else \
	   echo '(progn (boottran::boottocl "typars.boot") (${BYE}))' | ${BOOTSYS} >${TMP}/trace ; \
	  fi )
```

5.10 typrops.lisp [10]

Notice that the progn special form that wraps the compile-file function call executes **DEPS** first. The **DEPS** variable expands into an s-expression that will load **npextras.lisp**. This file contains macros needed at compile time.

⟨*typrops.o (OUT from MID)*⟩≡
```
${OUT}/typrops.${O}: ${MID}/typrops.lisp
	@ echo 34 making ${OUT}/typrops.${O} from ${MID}/typrops.lisp
	@ ( cd ${MID} ; \
	  if [ -z "${NOISE}" ] ; then \
	   echo '(progn ${PROCLAIMS} ${DEPS} (compile-file "typrops.lisp" :output-file "${OUT}/typrops.${O}") (${BYE}))' | ${LISPSYS} ; \
	  else \
	   echo '(progn ${PROCLAIMS} ${DEPS} (compile-file "typrops.lisp" :output-file "${OUT}/typrops.${O}") (${BYE}))' | ${LISPSYS} >${TMP}/trace ; \
	  fi )
```

⟨*typrops.lisp (MID from IN)*⟩≡
```
${MID}/typrops.lisp: ${IN}/typrops.boot.pamphlet
	@echo 35 making ${MID}/typrops.lisp from ${IN}/typrops.boot.pamphlet
	@${TANGLE} -Rtyprops.clisp typrops.boot.pamphlet >${MID}/typrops.lisp
```

⟨*typrops.boot.dvi (DOC from IN)*⟩≡
```
${DOC}/typrops.boot.dvi: ${IN}/typrops.boot.pamphlet ${DOC}/axiom.sty
	@echo 36 making ${DOC}/typrops.boot.dvi from \
	   ${IN}/typrops.boot.pamphlet
	@(cd ${DOC} ; \
	cp ${IN}/typrops.boot.pamphlet ${DOC} ; \
	${DOCUMENT} ${NOISE} typrops.boot ; \
	rm -f ${DOC}/typrops.boot.pamphlet ; \
	rm -f ${DOC}/typrops.boot.tex ; \
	rm -f ${DOC}/typrops.boot )
```

⟨*typrops.boot (IN from IN)*⟩≡
```
typrops.boot: typrops.boot.pamphlet
	@echo 37 making typrops.boot from typrops.boot.pamphlet
	@( ${TANGLE}  typrops.boot.pamphlet >typrops.boot ; \
	  if [ -z "${NOISE}" ] ; then \
	   echo '(progn (boottran::boottocl "typrops.boot") (${BYE}))' | ${BOOTSYS} ; \
	  else \
	   echo '(progn (boottran::boottocl "typrops.boot") (${BYE}))' | ${BOOTSYS} >${TMP}/trace ; \
	  fi )
```

5.11 tytree1.lisp [11]

Notice that the progn special form that wraps the compile-file function call executes **DEPS** first. The **DEPS** variable expands into an s-expression that will load **npextras.lisp**. This file contains macros needed at compile time.

⟨*tytree1.o (OUT from MID)*⟩≡
```
${OUT}/tytree1.${O}: ${MID}/tytree1.lisp
	@ echo 38 making ${OUT}/tytree1.${O} from ${MID}/tytree1.lisp
	@ ( cd ${MID} ; \
	  if [ -z "${NOISE}" ] ; then \
	   echo '(progn ${PROCLAIMS} ${DEPS} (compile-file "tytree1.lisp" :output-file "${OUT}/tytree1.${O}") (${BYE}))' | ${LISPSYS} ; \
	  else \
	   echo '(progn ${PROCLAIMS} ${DEPS} (compile-file "tytree1.lisp" :output-file "${OUT}/tytree1.${O}") (${BYE}))' | ${LISPSYS} >${TMP}/trace ; \
	  fi )
```

⟨*tytree1.lisp (MID from IN)*⟩≡
```
${MID}/tytree1.lisp: ${IN}/tytree1.boot.pamphlet
	@echo 39 making ${MID}/tytree1.lisp from ${IN}/tytree1.boot.pamphlet
	@${TANGLE} -Rtytree1.clisp tytree1.boot.pamphlet >${MID}/tytree1.lisp
```

⟨*tytree1.boot.dvi (DOC from IN)*⟩≡
```
${DOC}/tytree1.boot.dvi: ${IN}/tytree1.boot.pamphlet ${DOC}/axiom.sty
	@echo 40 making ${DOC}/tytree1.boot.dvi from \
	     ${IN}/tytree1.boot.pamphlet
	@(cd ${DOC} ; \
	cp ${IN}/tytree1.boot.pamphlet ${DOC} ; \
	${DOCUMENT} ${NOISE} tytree1.boot ; \
	rm -f ${DOC}/tytree1.boot.pamphlet ; \
	rm -f ${DOC}/tytree1.boot.tex ; \
	rm -f ${DOC}/tytree1.boot )
```

⟨*tytree1.boot (IN from IN)*⟩≡
```
tytree1.boot: tytree1.boot.pamphlet
	@echo 41 making tytree1.boot from tytree1.boot.pamphlet
	@( ${TANGLE} tytree1.boot.pamphlet >tytree1.boot ; \
	  if [ -z "${NOISE}" ] ; then \
	   echo '(progn (boottran::boottocl "tytree1.boot") (${BYE}))' | ${BOOTSYS} ; \
	  else \
	   echo '(progn (boottran::boottocl "tytree1.boot") (${BYE}))' | ${BOOTSYS} >${TMP}/trace ; \
	  fi )
```

6 axiom.sty

⟨*axiom.sty (OUT from IN)*⟩≡
```
${DOC}/axiom.sty: ${SRC}/doc/axiom.sty.pamphlet
        @ echo 618a making ${DOC}/axiom.sty from ${SRC}/doc/axiom.sty.pamphlet
        @(cd ${DOC} ; \
        cp ${SRC}/doc/axiom.sty.pamphlet ${DOC} ; \
        ${TANGLE} -Raxiom.sty axiom.sty.pamphlet >axiom.sty ; \
        rm -f ${DOC}/axiom.sty.pamphlet )
```

7 Making the documentation

⟨*environment*⟩+≡
```
DOCFILES=${DOC}/boothdr.lisp.dvi ${DOC}/btincl2.boot.dvi \
         ${DOC}/btpile2.boot.dvi ${DOC}/btscan2.boot.dvi \
         ${DOC}/exports.lisp.dvi ${DOC}/npextras.lisp.dvi \
         ${DOC}/ptyout.boot.dvi ${DOC}/tyextra.boot.dvi \
         ${DOC}/typars.boot.dvi ${DOC}/typrops.boot.dvi \
         ${DOC}/tytree1.boot.dvi
```

⟨*document*⟩≡
```
document: ${DOCFILES}
        @ echo 42 making tex and dvi files in ${DOC}
```

8 Cleanup

⟨*clean*⟩≡
```
clean:
        @echo 43 cleaning ${OUT}
        @rm -rf ${OUT}
```

9 The Makefile

⟨*⟩≡
  ⟨*virtual directories*⟩
  ⟨*environment*⟩

```
# this stanza will create the final bootsys image

${SAVESYS}:     ${OBJS} ${LOADSYS} ${DOCFILES}
	@ echo 44 invoking make in `pwd` with parms:
	@ echo SYS= ${SYS}
	@ echo LSP= ${LSP}
	@ echo PART= ${PART}
	@ echo SPAD= ${SPAD}
	@ echo SRC= ${SRC}
	@ echo INT= ${INT}
	@ echo OBJ= ${OBJ}
	@ echo MNT= ${MNT}
	@ (cd ${OBJ}/${SYS}/bin ; echo '${CMD0}' | ${LOADSYS} >${TMP}/console )
	@ echo 45 ${SAVESYS} created

boot: ${BOOTS}
```

  ⟨*boothdr.o (OUT from MID)*⟩
  ⟨*boothdr.lisp (MID from IN)*⟩
  ⟨*boothdr.lisp.dvi (DOC from IN)*⟩

  ⟨*exports.o (OUT from MID)*⟩
  ⟨*exports.lisp (MID from IN)*⟩
  ⟨*exports.lisp.dvi (DOC from IN)*⟩

  ⟨*npextras.o (OUT from MID)*⟩
  ⟨*npextras.lisp (MID from IN)*⟩
  ⟨*npextras.lisp.dvi (DOC from IN)*⟩

  ⟨*ptyout.o (OUT from MID)*⟩
  ⟨*ptyout.lisp (MID from IN)*⟩
  ⟨*ptyout.boot.dvi (DOC from IN)*⟩
  ⟨*ptyout.boot (IN from IN)*⟩

  ⟨*btincl2.o (OUT from MID)*⟩
  ⟨*btincl2.lisp (MID from IN)*⟩
  ⟨*btincl2.boot.dvi (DOC from IN)*⟩
  ⟨*btincl2.boot (IN from IN)*⟩

  ⟨*typrops.o (OUT from MID)*⟩

⟨*typprops.lisp (MID from IN)*⟩
⟨*typprops.boot.dvi (DOC from IN)*⟩
⟨*typprops.boot (IN from IN)*⟩

⟨*btpile2.o (OUT from MID)*⟩
⟨*btpile2.lisp (MID from IN)*⟩
⟨*btpile2.boot.dvi (DOC from IN)*⟩
⟨*btpile2.boot (IN from IN)*⟩

⟨*btscan2.o (OUT from MID)*⟩
⟨*btscan2.lisp (MID from IN)*⟩
⟨*btscan2.boot.dvi (DOC from IN)*⟩
⟨*btscan2.boot (IN from IN)*⟩

⟨*typars.o (OUT from MID)*⟩
⟨*typars.lisp (MID from IN)*⟩
⟨*typars.boot.dvi (DOC from IN)*⟩
⟨*typars.boot (IN from IN)*⟩

⟨*tyextra.o (OUT from MID)*⟩
⟨*tyextra.lisp (MID from IN)*⟩
⟨*tyextra.boot.dvi (DOC from IN)*⟩
⟨*tyextra.boot (IN from IN)*⟩

⟨*tytree1.o (OUT from MID)*⟩
⟨*tytree1.lisp (MID from IN)*⟩
⟨*tytree1.boot.dvi (DOC from IN)*⟩
⟨*tytree1.boot (IN from IN)*⟩

⟨*document*⟩
⟨*axiom.sty (OUT from IN)*⟩
⟨*clean*⟩

References

[1] $SPAD/src/boot/boothdr.lisp.pamphlet

[2] $SPAD/src/boot/btincl2.boot.pamphlet

[3] $SPAD/src/boot/btpile2.boot.pamphlet

[4] $SPAD/src/boot/btscan2.boot.pamphlet

[5] $SPAD/src/boot/exports.lisp.pamphlet

[6] $SPAD/src/boot/npextras.lisp.pamphlet

[7] $SPAD/src/boot/ptyout.boot.pamphlet

[8] $SPAD/src/boot/tyextra.boot.pamphlet

[9] $SPAD/src/boot/typars.boot.pamphlet

[10] $SPAD/src/boot/typrops.boot.pamphlet

[11] $SPAD/src/boot/tytree1.boot.pamphlet