

Appendix B

A YACC/Bison grammar

```
%{ /* Yacc/Bison grammar for
    Ease - a language for programming concurrent systems.

    Version Beta.09
    Copyright (C) 1991, 1992 Steven Ericsson Zenith.
    Copyright (C) 1992 Science Frontiers, International.

    This program is free software; you can redistribute it and/or modify
    it under the terms of the GNU General Public License as published by
    the Free Software Foundation; either version 1, or (at your option)
    any later version.

    This program is distributed in the hope that it will be
    useful, but WITHOUT ANY WARRANTY; without even the implied
    warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR
    PURPOSE. See the GNU General Public License for more details.

    You should have received a copy of the GNU General Public License
    along with this program; if not write to the Free Software
    Foundation, Inc., 675 Mass Ave, Cambridge, MA 02139, USA.
    */
%}
/* Terminals */
%union {
    struct symbol *sym;
    struct list *list;
    char *value;
}
```

```

/* name */
%token NAME

/* keywords and key symbols */
%token BECOMES READ WRITE GET PUT      /* := ? ! ?* !* */
%token SEQUENCE TEST SELECT RESOURCE CHOICE
%token ELSE AFTER ON STOP
%token COOPERATE SUBORDINATE          /* || // */
%token WHILE DO UNTIL
%token FOR FROM BY
%token INCREMENT DECREMENT           /* ++ -- */
%token STOP SKIP

/* expression symbols */
%token MUL DIV REM EXP                /* * / % ^ */
%token SUB ADD                        /* - + */
%token EQ NE GT LT GE LE              /* = <> > < >= <= */
%token AND OR XOR
%token BITAND BITOR BITXOR           /* /\ \/ >< */
%token RSHIFT LSHIFT                 /* >> << */
%token COMPLEMENT NOT                 /* ~ */
%token TEXT CHARACTER NUMBER
%token TRUE FALSE
%token IF SIZE OF
%token OPENTO CLOSETO TO              /* (.. ..) .. */
%token COERCE ASSERT CAST             /* -> => >| */

/* declarators and types */
%token LET VAL RENAME TYPE ENUM IS AT
%token COUNTED                        /* :: */
%token CONTEXT STREAM SINGLE LO HI REPLY
%token PROCEDURE FUNCTION WHERE
%token BOOL
%token INT INT8 INT16 INT32 INT64 INT128
%token FLOAT32 FLOAT64 FLOAT128 CHAR STRING
%token UNSIGNED TRUNC ROUND

/* modules */
%token MODULE END SHOW HIDE USE

/* linebreak points */
%token NEWLINE

```

```
%start program

%%
/* processes */
program: . process .
        ;

process:  action
        | construction
        | instance
        | specification_block ':' ... process
        | error_handler
        ;

error_handler: /* Handles grammatical errors:
               disguards process or finds next semicolon */
             error process
             | error ';'
             ;

action:  assignment
        | interaction
        | STOP
        | SKIP
        ;

construction:
        sequence
        | test
        | selection
        | combination
        | choice
        | cooperation
        | subordination
        | repetition
        ;

assignment:
        element BECOMES expression
        | element INCREMENT
        | element DECREMENT
        ;
```

```
interaction:
    input
  | output
  ;

input:    read
  | get
  ;

output:   write
  | put
  ;

read:     element READ element_list
  ;

write:    element WRITE expression_list
  ;

get:      element GET reference_list
  ;

put:      element PUT reference_list
  ;

reference: NAME
  ;

reference_list:
    reference
  | reference ',' . reference_list
  ;

element_list:
    element
  | element ',' . element_list
  ;

sequence: SEQUENCE . '}'
  | SEQUENCE . subsequence '}'
  | SEQUENCE . replicator .. subsequence '}'
  | SEQUENCE . ON STOP . process .. subsequence '}'
  ;
```

```

subsequence:
    process .
    | process ... subsequence
    ;

test:    TEST . default
    | TEST . conditional_body default
    | TEST . replicator .. conditional . default
    ;

default: ';'
    | ELSE . process . ';'
    ;

conditional_body:
    conditional .
    | conditional ... conditional_body
    ;

conditional:
    expression .. process
    | test
    ;

selection:
    SELECT selector .. option_body default
    ;

selector: expression
    ;

option_body:
    option .
    | option ... option_body
    ;

option:  match_list .. process
    ;

match_list:
    expression
    | expression ',' . match_list

```

```

        ;

combination:
    call
    | reply
    ;

call:    write GET reference
    | put  GET reference
    ;

reply:   RESOURCE . get . WRITE expression
    | RESOURCE . get .. process . WRITE expression
    | RESOURCE . get .. process . PUT  reference
    ;

choice:  CHOICE . default_choice
    | CHOICE . alternative_body default_choice
    | CHOICE . replicator .. alternative . default_choice
    ;

default_choice:
    ';'
    | ELSE . process . ';'
    | ELSE . AFTER time .. process . ';'
    ;

alternative_body:
    alternative .
    | alternative ... alternative_body
    ;

alternative:
    input .. process
    | boolean .. input .. process
    | reply
    | boolean .. reply
    | '|' . process
    | boolean '|' . process
    | choice
    ;

time:    float

```

```

;

cooperation:
    cooperate . ';'
    | cooperate . cooperation
;

cooperate:
    COOPERATE . process
    | COOPERATE . replicator .. process
;

subordination:
    subordinate . ';'
    | subordinate . subordination
;

subordinate:
    SUBORDINATE . process
    | SUBORDINATE . replicator .. process
    /* Process placement */
    | SUBORDINATE . ON node .. process
    | SUBORDINATE . ON replicator .. process
;

repetition:
    WHILE boolean .. process
    | DO . process . UNTIL boolean
;

replicator:
    replicant FOR . count
    | replicant FOR . count FROM . base
    | replicant FOR . count FROM . base BY step
;

replicant:
    NAME
;

instance:
    procid '(' argument_list ')'
;

```

```
/* specifications */
specification_block:
    specification
    | specification specification_block
    ;

specification:
    declaration .
    | definition
    ;

definition:
    procedure
    | function
    | typedef ...
    | module ...
    ;

declaration:
    LET specifier
    | ENUM name_list
    | ENUM FROM base . name_list
    | USE modulid
    ;

specifier:
    NAME EQ expression
    | name_list BECOMES allocation
    | NAME RENAME element
    ;

allocation:
    expression
    | expression ON node
    ;

node:
    integer
    | integer AT integer
    ;

name_list:
    NAME
    | NAME ',' . name_list
```

```
        ;

procedure:
    PROCEDURE procid '(' pformal_list ')' . process .
    ;

procid:  NAME
    ;

pformal_list:
    /* EMPTY */
    | pformal
    | pformal ',' . pformal_list
    ;

pformal:  NAME
    | compliant NAME
    | VAL NAME
    | VAL compliant NAME
    ;

function:
    FUNCTION funcid '(' fformal_list ')' . function_expression
    ;

funcid:  NAME
    ;

function_expression:
    EQ . expression .
    | EQ . expression . WHERE . process .
    ;

fformal_list:
    /* EMPTY */
    | fformal
    | fformal ',' . fformal_list
    ;

fformal:  NAME
    | compliant NAME
    ;
```

```
compliant:
    type
    | '[' ']' type
    ;

/* modules */
module:  MODULE modulid ... module_body END MODULE
        ;

modulid: NAME
        ;

module_body:
    show hide
    | hide show
    ;

show:    SHOW . specification_block
        ;

hide:    HIDE . specification_block
        ;

/* types */
type:    primitive_type
    | tuple_type
    | array_type
    | integer_type '[' limit ']' COUNTED type
    | typeid
    | TYPE OF expression
    ;

typedef: TYPE typeid IS type
    | TYPE typeid CONTEXT . bag_list
    ;

typeid:  NAME
        ;

bag_list: bag
    | bag_list '|' . bag
    ;
```

```
bag:      bag_type
        | priority .. bag_type
        ;

bag_type:
        unordered
        | singleton
        | stream
        | exchange
        ;

unordered:
        type          ;

singleton:
        SINGLE type
        | '[' integer ']' . singleton
        | '[' integer ']' . stream
        ;

stream:   STREAM type
        ;

exchange: type REPLY type
        ;

priority:
        LO
        | HI
        | LO integer
        | HI integer
        ;

primitive_type:
        BOOL
        | integer_type
        | UNSIGNED integer_type
        | float_type
        ;

integer_type:
        INT
        | CHAR
```

```
    | INT8
    | INT16
    | INT32
    | INT64
    | INT128
    ;

float_type:
    FLOAT32
    | FLOAT64
    | FLOAT128
    ;

array_type:
    '[' integer ']' . type
    | STRING
    ;

tuple_type:
    '(' type ',' . type_list ')'
    ;

type_list:
    type
    | type ',' . type_list
    ;

/* elements and expressions */
literal: CHARACTER
    | string
    | NUMBER
    | TRUE
    | FALSE
    | '_'
    ;

string: TEXT
    | TEXT '\\\ ' ... string
    ;

tuple: '(' expression ',' . expression_list ')'
    ;
```

```

table:    '[' expression_list ']'
        ;

element:  NAME
        | tuple
        | table
        | element '[' . subscript ']'
        | element '[' . base CLOSETO
        | element OPENTO limit ']'
        | element '[' . base FOR . count ']'
        | element '[' . base TO . limit ']'
        ;

lambda:   NAME '(' argument_list ')'
        | lambda_binding . '(' . function_expression ')'
        ;

argument_list:
        /* EMPTY */
        | expression_list
        ;

lambda_binding:
        declaration . ':'
        | declaration . lambda_binding
        ;

merge:    string '(' expression_list ')'
        ;

primary:  literal
        | merge
        | element
        | lambda
        | '(' expression . ')'
        ;

monadic:  primary
        | ADD primary
        | SUB primary
        | NOT primary
        | COMPLEMENT primary

```

```
    | SIZE OF element
    ;

exponent: monadic
    | monadic EXP . exponent
    ;

multiplicative:
    exponent
    | multiplicative MUL . exponent
    | multiplicative DIV . exponent
    | multiplicative REM . exponent
    ;

additive: multiplicative
    | additive ADD . multiplicative
    | additive SUB . multiplicative
    ;

shift:    additive
    | shift LSHIFT . additive
    | shift RSHIFT . additive
    ;

relational:
    shift
    | relational LT . shift
    | relational GT . shift
    | relational LE . shift
    | relational GE . shift
    ;

equality: relational
    | equality EQ . relational
    | equality NE . relational
    ;

bitwise:  equality
    | bitwise BITAND . equality
    | bitwise BITOR  . equality
    | bitwise BITXOR . equality
    ;
```

```
logical: bitwise
        | logical AND . bitwise
        | logical OR  . bitwise
        | logical XOR . bitwise
        ;

constraint:
        logical
        | logical COERCE type
        | logical COERCE TRUNC type
        | logical COERCE ROUND type
        | logical ASSERT type
        | logical CAST  type
        ;

expression:
        constraint
        | IF boolean .. expression . ELSE . expression
        ;

expression_list:
        expression
        | expression ',' . expression_list
        ;

/* semantic limiters */
subscript: integer
        ;

base:     integer
        ;

count:    integer
        ;

limit:    integer
        ;

step:     integer
        ;

boolean:  expression
        ;
```

```
integer: expression
        ;

float:   expression
        ;

/* Newlines and colon */
.:      /* EMPTY */
        | ...
        ;

...:    NEWLINE
        | ... NEWLINE
        ;

...:    ...
        | ':' ' '
        ;

%%
```

Appendix C

A FLEX lexer

```
/* Ease Compiler - Copyright (C) 1991, 1992 Steven Ericsson Zenith
```

```

A flex specified lexer for
Ease - a language for programming concurrent systems
Copyright (C) 1991, 1992 Steven Ericsson Zenith.
Copyright (C) 1992 Science Frontiers, International.
#HPC01 Computer Science Project.
```

```
This lexer accompanies the grammar "ease.y".
```

```
Version Beta .09 April 1992.
```

```
*/
%{
```

```
#include <stdio.h>
#include "ease_tab.h" /* bison generated definitions */
#include "ease.h"
#include "version.h"
```

```
int lineno = 1;
%}
```

```
Name          [a-zA-Z][0-9a-zA-Z_']*
Whitespace    [ \t]+
Newline       [\n\f\r]
Integer       [0-9]*|"#[0-9A-Fa-f]*
Float         [0-9]+("."[0-9]+)?[eE][+-]?[0-9]+|[0-9]+"."[0-9]*
```

```

Text          \"(\\.|[^\\""])*\"
Character     ' . | '\\ . '
%%

"/*"         comment();

{Whitespace} ;

{Newline}    {
    ++lineno;
    return(NEWLINE);
}

test         { return(TEST);      }
select      { return(SELECT);    }
resource    { return(RESOURCE);  }
choice      { return(CHOICE);    }
else        { return(ELSE);      }
after       { return(AFTER);     }
on          { return(ON);        }
while       { return(WHILE);     }
do          { return(DO);        }
until       { return(UNTIL);    }
for         { return(FOR);       }
from        { return(FROM);     }
by          { return(BY);       }

and         { return(AND);       }
or          { return(OR);        }
xor         { return(XOR);       }
not         { return(NOT);       }
true        { return(TRUE);     }
false       { return(FALSE);    }
if          { return(IF);       }
size        { return(SIZE);     }
of          { return(OF);       }

let         { return(LET);       }
val         { return(VAL);       }
rename      { return(RENAME);   }
type        { return(TYPE);     }
enum        { return(ENUM);     }
is          { return(IS);       }

```

```

context      { return(CONTEXT);      }
stream      { return(STREAM);        }
single      { return(SINGLE);         }
lo           { return(LO);            }
hi           { return(HI);            }
reply       { return(REPLY);         }

procedure   { return(PROCEDURE);      }
function    { return(FUNCTION);      }
where       { return(WHERE);         }

bool        { return(BOOL);          }

int         { return(INT);            }
int[8]      { return(INT8);           }
int[1][6]   { return(INT16);         }
int[3][2]   { return(INT32);         }
int[6][4]   { return(INT64);         }
int[1][2][8] { return(INT128);       }

float[3][2] { return(FLOAT32);       }
float[6][4] { return(FLOAT64);       }
float[1][2][8] { return(FLOAT128);   }

string      { return(STRING);        }
char        { return(CHAR);          }

module      { return(MODULE);        }
end         { return(END);            }
show       { return(SHOW);           }
hide       { return(HIDE);           }
use        { return(USE);            }

stop       { return(STOP);           }
skip       { return(SKIP);           }

round      { return(ROUND);          }
trunc     { return(TRUNC);           }

at         { return(AT);             }

{Name}     {

```

```

        yyval.value = (char *)strdup(yytext);
        return(NAME);
    }

    {Integer}      {
        default_type = "int";
        yyval.value = (char *)strdup(yytext);
        return(NUMBER);
    }

    {Float}       {
        default_type = "float32";
        yyval.value = (char *)strdup(yytext);
        return(NUMBER);
    }

    {Text}        {
        default_type = "string";
        yyval.value = (char *)strdup(yytext);
        return(TEXT);
    }
    }{Character}  {
        default_type = "char";
        yyval.value = (char *)strdup(yytext);
        return(CHARACTER);
    }

    "{"           { return(SEQUENCE);    }
    "||"         { return(COOPERATE);    }
    "//"         { return(SUBORDINATE);  }

    "++"         { return(INCREMENT);    }
    "--"         { return(DECREMENT);    }

    ":@"         { return(BECOMES);     }
    "?*"         { return(GET);         }
    "!*"         { return(PUT);         }
    "?"         { return(READ);         }
    "!"         { return(WRITE);       }

    "^"         { return(EXP);         }
    "*"         { return(MUL);         }
    "/"         { return(DIV);         }
    "%"         { return(REM);         }

```

```

"- "      { return(SUB);      }
"+"      { return(ADD);      }

"="      { return(EQ);      }
"<>"     { return(NE);      }
">"      { return(GT);      }
"<"      { return(LT);      }
">="     { return(GE);      }
"<="     { return(LE);      }

"/\\"     { return(BITAND);   }
"\\/\"    { return(BITOR);   }
"><"     { return(BITXOR);  }
">>"     { return(RSHIFT);  }
"<<"     { return(LSHIFT);  }
"~"      { return(COMPLEMENT); }

"(.."    { return(OPENTO);   }
"..)"    { return(CLOSETO);  }
".."     { return(TO);      }

"=>"     { return(ASSERT);   }
"->"     { return(COERCE);   }
">|"     { return(CAST);     }

"::"     { return(COUNTED);  }

"}"      { return('}');     }
":"      { return(':');     }
";"      { return(';');     }

"("      { return('(');     }
")"      { return(')');     }

"["      { return('[');     }
"]"      { return(']');     }

"|"      { return('|');     }

","      { return(',');     }

"_ "     { return('_');     }

```

```

"\"          { return('\');      }

.           {
char m[40];
sprintf(m,"illegal character '\\%o'", (int)yytext[0]);
yyerror(m);
}

<<EOF>>     {
if (diagnose) printf("/* Lexer: end of file */\n");
yyterminate();
}
%%

int comment()
/* For now we won't take care of nested comments or premature EOF
*/
{
    char c ;

    while ((c = input()) != '/') {
        if (c == '\n') ++lineno ;
        while ((c = input()) != '*') if (c == '\n') lineno++ ;
    }
}

yyerror(char *s)
{
    printf("Ease %s: at line %d = %s (%s)\n",
          EASE_VERSION, lineno, s, yytext);
}

```