```
Group 10
Compilers Lab
19 March 2018

150101009 Anup Agarwal
150101078 Surabhi Gupta
150101059 Satti Sai Chandan Reddy

Assignment 3 Submission

--------------------------------- grammar.ypp ---------------------------------

%token CHAR ELSE FLOAT FOR IF INT RETURN VOID WHILE PTR_OP INC_OP DEC_OP AND_OP
OR_OP LT GT LE_OP GE_OP EQ_OP NE_OP BOOL DO
%token NOT AMP TILDE STAR ASSIGN PROCESSOR LINK CLUSTER JOB MEMORY JOB_ID
FLOPS_REQUIRED DEADLINE
%token AFFINITY ALGO ISA CLOCK_SPEED MEM_REQUIRED ID MEMORY_TYPE SCHEDULER
%token NUM REAL STRING_LITERAL L1_MEMORY L2_MEMORY MEM_SIZE NAME START_POINT
END_POINT BANDWIDTH
%token CHANNEL_CAPACITY TOPOLOGY LINK_CAPACITY LINK_BANDWIDTH GET_AVAILABLE_MEMORY
GET_MEMORY IS_RUNNING SUBMIT_JOBS
%token GET_CLOCK_SPEED DISCARD_JOB RUN OPEN_BRACKET CLOSED_BRACKET OPEN_CURLY
CLOSED_CURLY OPEN_SQUARE CLOSED_SQUARE
%token SEMI COMMA DOT PLUS MINUS DIVIDE MODULUS PIPE XOR QUES COLON SCHED

%nonassoc "UIF"
%nonassoc ELSE

%{
  extern int yylex();
  extern int yyerror(char *);
%}

%%

statement_list
: statement statement_list { printf("statement_list: statement statement_list
\n"); }
| /* EPSILON */ { printf("statement_list: EPSILON\n"); }
;

statement
: expression_statement { printf("statement: expression_statement\n\n\n"); }
| selection_statement { printf("statement: selection_statement\n\n\n"); }
| iteration_statement { printf("statement: iteration_statement\n\n\n"); }
| OPEN_CURLY statement_list CLOSED_CURLY { printf("statement: OPEN_CURLY
statement_list CLOSED_CURLY\n\n\n"); }
| declaration { printf("statement: declaration\n\n\n"); }
;

declaration
: type_specifier declarator ASSIGN assignment_expression SEMI { printf
("declaration: type_specifier declarator ASSIGN assignment_expression SEMI\n"); }
;

declarator
: ID  { printf("declarator: ID\n"); }
| declarator OPEN_SQUARE assignment_expression CLOSED_SQUARE { printf("declarator:
declarator OPEN_SQUARE assignment_expression CLOSED_SQUARE\n"); }
| declarator OPEN_SQUARE CLOSED_SQUARE  { printf("declarator: declarator
OPEN_SQUARE CLOSED_SQUARE\n"); }
;

type_specifier
: VOID { printf("type_specifier: VOID\n"); }
| CHAR { printf("type_specifier: CHAR\n"); }
```

```
| INT { printf("type_specifier: INT\n"); }
| FLOAT { printf("type_specifier: FLOAT\n"); }
| BOOL { printf("type_specifier: BOOL\n"); }
| PROCESSOR { printf("type_specifier: PROCESSOR\n"); }
| LINK { printf("type_specifier: LINK\n"); }
| CLUSTER { printf("type_specifier: CLUSTER\n"); }
| JOB { printf("type_specifier: JOB\n"); }
| MEMORY { printf("type_specifier: MEMORY\n"); }
;

primary_expression
: ID { printf("primary_expression: ID\n"); }
| NUM { printf("primary_expression: NUM\n"); }
| REAL { printf("primary_expression: REAL\n"); }
| STRING_LITERAL { printf("primary_expression: STRING_LITERAL\n"); }
| OPEN_BRACKET expression CLOSED_BRACKET { printf("primary_expression:
OPEN_BRACKET expression CLOSED_BRACKET\n"); }
| constructor { printf("primary_expression: constructor \n"); }
| mem_func { printf("primary_expression: mem_func\n"); }
| OPEN_SQUARE argument_expression_list CLOSED_SQUARE { printf("primary_expression:
OPEN_SQUARE argument_expression_list CLOSED_SQUARE\n"); }
;

postfix_expression
: primary_expression { printf("postfix_expression: primary_expression\n"); }
| postfix_expression OPEN_SQUARE expression CLOSED_SQUARE { printf
("postfix_expression: postfix_expression OPEN_SQUARE expression CLOSED_SQUARE
\n"); }
| postfix_expression OPEN_BRACKET CLOSED_BRACKET { printf("postfix_expression:
postfix_expression OPEN_BRACKET CLOSED_BRACKET\n"); }
| postfix_expression OPEN_BRACKET argument_expression_list CLOSED_BRACKET { printf
("postfix_expression: postfix_expression OPEN_BRACKET argument_expression_list
CLOSED_BRACKET\n"); }
| postfix_expression DOT ID { printf("postfix_expression: postfix_expression DOT ID
\n"); }
| postfix_expression DOT mem_func { printf("postfix_expression: postfix_expression
DOT mem_func\n"); }
| postfix_expression PTR_OP ID { printf("postfix_expression: postfix_expression
PTR_OP ID\n"); }
| postfix_expression INC_OP { printf("postfix_expression: postfix_expression INC_OP
\n"); }
| postfix_expression DEC_OP { printf("postfix_expression: postfix_expression DEC_OP
\n"); }
;

argument_expression_list
: assignment_expression  { printf("argument_expression_list: assignment_expression
\n"); }
| argument_expression_list COMMA assignment_expression { printf
("argument_expression_list: argument_expression_list COMMA assignment_expression
\n"); }
;

unary_expression
: postfix_expression { printf("unary_expression: postfix_expression\n"); }
| INC_OP unary_expression { printf("unary_expression: INC_OP unary_expression
\n"); }
| DEC_OP unary_expression { printf("unary_expression: DEC_OP unary_expression
\n"); }
| unary_operator unary_expression { printf("unary_expression: unary_operator
unary_expression\n"); }
;

unary_operator
: AMP { printf("unary_operator: AMP\n"); }
| STAR { printf("unary_operator: STAR\n"); }
```

```
| PLUS { printf("unary_operator: PLUS\n"); }
| MINUS { printf("unary_operator: MINUS\n"); }
| TILDE { printf("unary_operator: TILDE\n"); }
| NOT { printf("unary_operator: NOT\n"); }
;

multiplicative_expression
: unary_expression { printf("multiplicative_expression: unary_expression\n"); }
| multiplicative_expression STAR unary_expression { printf
("multiplicative_expression: multiplicative_expression STAR unary_expression\n"); }
| multiplicative_expression DIVIDE unary_expression { printf
("multiplicative_expression: multiplicative_expression DIVIDE unary_expression
\n"); }
| multiplicative_expression MODULUS unary_expression { printf
("multiplicative_expression: multiplicative_expression MODULUS unary_expression
\n"); }
;

additive_expression
: multiplicative_expression { printf("additive_expression:
multiplicative_expression\n"); }
| additive_expression PLUS multiplicative_expression { printf
("additive_expression: additive_expression PLUS multiplicative_expression\n"); }
| additive_expression MINUS multiplicative_expression { printf
("additive_expression: additive_expression MINUS multiplicative_expression\n"); }
;

relational_expression
: additive_expression { printf("relational_expression: additive_expression\n"); }
| relational_expression LT additive_expression { printf("relational_expression:
relational_expression LT additive_expression\n"); }
| relational_expression GT additive_expression { printf("relational_expression:
relational_expression GT additive_expression\n"); }
| relational_expression LE_OP additive_expression { printf("relational_expression:
relational_expression LE_OP additive_expression\n"); }
| relational_expression GE_OP additive_expression { printf("relational_expression:
relational_expression GE_OP additive_expression\n"); }
;

equality_expression
: relational_expression { printf("equality_expression: relational_expression\n"); }
| equality_expression EQ_OP relational_expression { printf("equality_expression:
equality_expression EQ_OP relational_expression\n"); }
| equality_expression NE_OP relational_expression { printf("equality_expression:
equality_expression EQ_OP relational_expression\n"); }
;

and_expression
: equality_expression { printf("and_expression: equality_expression\n"); }
| and_expression AMP equality_expression { printf("and_expression: and_expression
AMP equality_expression\n"); }
;

exclusive_or_expression
: and_expression { printf("exclusive_or_expression: and_expression\n"); }
| exclusive_or_expression XOR and_expression { printf("exclusive_or_expression:
exclusive_or_expression XOR and_expression\n"); }
;

inclusive_or_expression
: exclusive_or_expression { printf("inclusive_or_expression:
exclusive_or_expression\n"); }
| inclusive_or_expression PIPE exclusive_or_expression { printf
("inclusive_or_expression: inclusive_or_expression PIPE exclusive_or_expression
\n"); }
;
```

```
logical_and_expression
: inclusive_or_expression { printf("logical_and_expression: inclusive_or_expression
\n"); }
| logical_and_expression AND_OP inclusive_or_expression { printf
("logical_and_expression: logical_and_expression AND_OP inclusive_or_expression
\n"); }
;

logical_or_expression
: logical_and_expression { printf("logical_or_expression: logical_and_expression
\n"); }
| logical_or_expression OR_OP logical_and_expression { printf
("logical_or_expression: logical_or_expression OR_OP logical_and_expression\n"); }
;

conditional_expression
: logical_or_expression { printf("conditional_expression: logical_or_expression
\n"); }
| logical_or_expression QUES expression COLON conditional_expression { printf
("conditional_expression: logical_or_expression QUES expression COLON
conditional_expressionn"); }
;

assignment_expression
: conditional_expression { printf("assignment_expression: conditional_expression\n
\n"); }
| unary_expression ASSIGN assignment_expression { printf("assignment_expression:
unary_expression ASSIGN assignment_expression\n\n"); }
;

expression
: assignment_expression { printf("expression: assignement_expression\n"); }
| expression COMMA assignment_expression { printf("expression: expression COMMA
assignment_expression\n"); }
;

iteration_statement
: WHILE OPEN_BRACKET expression CLOSED_BRACKET statement { printf
("iteration_statement: WHILE OPEN_BRACKET expression CLOSED_BRACKET statement
\n"); }
| DO statement WHILE OPEN_BRACKET expression CLOSED_BRACKET SEMI { printf
("iteration_statement: DO statement WHILE OPEN_BRACKET expression CLOSED_BRACKET
SEMI\n"); }
| FOR OPEN_BRACKET expression_statement expression_statement CLOSED_BRACKET
statement { printf("iteration_statement: FOR OPEN_BRACKET expression_statement
expression_statement CLOSED_BRACKET statement\n"); }
| FOR OPEN_BRACKET expression_statement expression_statement expression
CLOSED_BRACKET statement { printf("iteration_statement: FOR OPEN_BRACKET
expression_statement expression_statement expression CLOSED_BRACKET statement
\n"); }
| FOR OPEN_BRACKET declaration expression_statement CLOSED_BRACKET statement
{ printf("iteration_statement: FOR OPEN_BRACKET declaration expression_statement
CLOSED_BRACKET statement\n"); }
| FOR OPEN_BRACKET declaration expression_statement expression CLOSED_BRACKET
statement { printf("iteration_statement: FOR OPEN_BRACKET declaration
expression_statement expression CLOSED_BRACKET statement\n"); }
;

selection_statement
: IF OPEN_BRACKET expression CLOSED_BRACKET statement ELSE statement { printf
("selection_statement: IF OPEN_BRACKET expression CLOSED_BRACKET statement ELSE
statement\n"); }
| IF OPEN_BRACKET expression CLOSED_BRACKET statement %prec "UIF" { printf("IF
OPEN_BRACKET expression CLOSED_BRACKET statement\n"); }
;
```

```
expression_statement
: SEMI { printf("expression_statement: SEMI\n"); }
| expression SEMI { printf("expression_statement: expression SEMI\n"); }
;

assign_colon
: ASSIGN { printf("assign_colon: ASSIGN\n"); }
| COLON { printf("assign_colon: COLON\n"); }
;

constructor
: PROCESSOR OPEN_BRACKET param_list_proc CLOSED_BRACKET { printf("constructor:
PROCESSOR OPEN_BRACKET param_list_proc CLOSED_BRACKET\n"); }
| MEMORY OPEN_BRACKET param_list_mem CLOSED_BRACKET { printf("constructor: MEMORY
OPEN_BRACKET param_list_mem CLOSED_BRACKET\n"); }
| JOB OPEN_BRACKET param_list_job CLOSED_BRACKET { printf("constructor: JOB
OPEN_BRACKET param_list_job CLOSED_BRACKET\n"); }
| LINK OPEN_BRACKET param_list_link CLOSED_BRACKET { printf("constructor: LINK
OPEN_BRACKET param_list_link CLOSED_BRACKET\n"); }
| CLUSTER OPEN_BRACKET param_list_cluster CLOSED_BRACKET { printf("constructor:
CLUSTER OPEN_BRACKET param_list_cluster CLOSED_BRACKET\n"); }
| SCHEDULER OPEN_BRACKET param_list_scheduler CLOSED_BRACKET { printf
("constructor: SCHEDULER OPEN_BRACKET param_list_scheduler CLOSED_BRACKET\n"); }
;

param_list_job
: job_param1 COMMA job_param2 COMMA job_param3 COMMA job_param4 COMMA job_param5
{ printf("param_list_job: job_param1 COMMA job_param2 COMMA job_param3 COMMA
job_param4 COMMA job_param5\n"); }
;

job_param1
: JOB_ID assign_colon assignment_expression { printf("job_param1: JOB_ID
assign_colon assignment_expression\n"); }
| assignment_expression { printf("job_param1: assignment_expression\n"); }
;

job_param2
: FLOPS_REQUIRED assign_colon assignment_expression { printf("job_param2:
FLOPS_REQUIRED assign_colon assignment_expression\n"); }
| assignment_expression { printf("job_param2: assignment_expression\n"); }
;

job_param3
: DEADLINE assign_colon assignment_expression { printf("job_param3: DEADLINE
assign_colon assignment_expression\n"); }
| assignment_expression { printf("job_param3: assignment_expression\n"); }
;

job_param4
: MEM_REQUIRED assign_colon assignment_expression { printf("job_param4:
MEM_REQUIRED assign_colon assignment_expression\n"); }
| assignment_expression { printf("job_param4: assignment_expression\n"); }
;

job_param5
: AFFINITY assign_colon assignment_expression { printf("job_param5: AFFINITY
assign_colon assignment_expression\n"); }
| assignment_expression { printf("job_param5: assignment_expression\n"); }
;

param_list_proc
: proc_param1 COMMA proc_param2 COMMA proc_param3 { printf("param_list_proc:
proc_param1 COMMA proc_param2 COMMA proc_param3\n"); }
;
```

```
proc_param1
: ISA assign_colon assignment_expression { printf("proc_param1: ISA assign_colon
assignment_expression\n"); }
| assignment_expression { printf("proc_param1: assignment_expression\n"); }
;

proc_param2
: CLOCK_SPEED assign_colon assignment_expression { printf("proc_param2:
CLOCK_SPEED assign_colon assignment_expression\n"); }
| assignment_expression { printf("proc_param2: assignment_expression\n"); }
;

proc_param3
: L1_MEMORY assign_colon assignment_expression {printf("proc_param3: L1_MEMORY
assign_colon assignment_expression\n"); }
| assignment_expression {printf("proc_param3: assignment_expression\n"); }
| L1_MEMORY assign_colon assignment_expression COMMA opt_proc_param4 {printf
("proc_param3: L1_MEMORY assign_colon assignment_expression COMMA opt_proc_param4
\n"); }
| assignment_expression COMMA opt_proc_param4 {printf("proc_param3:
assignment_expression COMMA opt_proc_param4\n"); }
;

opt_proc_param4
: L2_MEMORY assign_colon assignment_expression {printf("opt_proc_param4: L2_MEMORY
assign_colon assignment_expression\n"); }
| L2_MEMORY assign_colon assignment_expression COMMA opt_proc_param5 { printf
("opt_proc_param4: L2_MEMORY assign_colon assignment_expression COMMA
opt_proc_param5 \n"); }
| assignment_expression { printf("opt_proc_param4: assignment_expression\n"); }
| assignment_expression COMMA opt_proc_param5 { printf("opt_proc_param4:
assignment_expression COMMA opt_proc_param5\n"); }
| SCHED assign_colon assignment_expression { printf("opt_proc_param5: SCHED
assign_colon assignment_expression\n"); }
| SCHED assign_colon assignment_expression COMMA opt_proc_param6 { printf
("opt_proc_param5: SCHED assign_colon assignment_expression COMMA opt_proc_param6
\n"); }
| NAME assign_colon assignment_expression { printf("opt_proc_param6: NAME
assign_colon assignment_expression\n"); }
;

opt_proc_param5
: SCHED assign_colon assignment_expression { printf("opt_proc_param5: SCHED
assign_colon assignment_expression\n"); }
| assignment_expression { printf("opt_proc_param5: assignment_expression\n"); }
| SCHED assign_colon assignment_expression COMMA opt_proc_param6 { printf
("opt_proc_param5: SCHED assign_colon assignment_expression COMMA opt_proc_param6
\n"); }
| assignment_expression COMMA opt_proc_param6 { printf("opt_proc_param5:
assignment_expression COMMA opt_proc_param6\n"); }
| NAME assign_colon assignment_expression { printf("opt_proc_param6: NAME
assign_colon assignment_expression\n"); }
;

opt_proc_param6
: NAME assign_colon assignment_expression { printf("opt_proc_param6: NAME
assign_colon assignment_expression\n"); }
| assignment_expression { printf("opt_proc_param6: assignment_expression\n"); }
;

param_list_mem
: mem_param1 COMMA mem_param2 { printf("param_list_mem: mem_param1 COMMA mem_param2
\n"); }
;
```

```
mem_param1
: MEMORY_TYPE assign_colon assignment_expression { printf("mem_param1: MEMORY_TYPE
assign_colon assignment_expression\n"); }
| assignment_expression { printf("mem_param1: assignment_expression\n"); }
;

mem_param2
: MEM_SIZE assign_colon assignment_expression { printf("mem_param2: MEM_SIZE
assign_colon assignment_expression\n"); }
| assignment_expression { printf("mem_param2: assignment_expression\n"); }
| MEM_SIZE assign_colon assignment_expression COMMA opt_mem_param3 { printf
("mem_param2: MEM_SIZE assign_colon assignment_expression COMMA opt_mem_param3
\n"); }
| assignment_expression COMMA opt_mem_param3 { printf("mem_param2:
assignment_expression COMMA opt_mem_param3\n"); }
;

opt_mem_param3
: NAME assign_colon assignment_expression { printf("opt_mem_param3: NAME
assign_colon assignment_expression\n"); }
| assignment_expression { printf("opt_mem_param3: assignment_expression\n"); }
;

param_list_link
: link_param1 COMMA link_param2 COMMA link_param3 COMMA link_param4 { printf
("param_list_link: link_param1 COMMA link_param2 COMMA link_param3 COMMA
link_param4\n"); }
;

link_param1
: START_POINT assign_colon assignment_expression { printf("link_param1:
START_POINT assign_colon assignment_expression\n"); }
| assignment_expression { printf("link_param1: assignment_expression\n"); }
;

link_param2
: END_POINT assign_colon assignment_expression { printf("link_param2: END_POINT
assign_colon assignment_expression\n"); }
| assignment_expression { printf("link_param2: assignment_expression\n"); }
;

link_param3
: BANDWIDTH assign_colon assignment_expression { printf("link_param3: BANDWIDTH
assign_colon assignment_expression\n"); }
| assignment_expression { printf("link_param3: assignment_expression\n"); }
;

link_param4
: CHANNEL_CAPACITY assign_colon assignment_expression { printf("link_param4:
CHANNEL_CAPACITY assign_colon assignment_expression\n"); }
| assignment_expression { printf("link_param4: assignment_expression\n"); }
| CHANNEL_CAPACITY assign_colon assignment_expression COMMA opt_link_param5
{ printf("link_param4: CHANNEL_CAPACITY assign_colon assignment_expression COMMA
opt_link_param5\n"); }
| assignment_expression COMMA opt_link_param5 { printf("link_param4:
assignment_expression COMMA opt_link_param5\n"); }
;

opt_link_param5
: NAME assign_colon assignment_expression { printf("opt_link_param5: NAME
assign_colon assignment_expression\n"); }
| assignment_expression { printf("opt_link_param5: assignment_expression\n"); }
;

param_list_cluster
: cluster_param1 COMMA cluster_param2 COMMA cluster_param3 COMMA cluster_param4
```

```
{ printf("param_list_cluster: cluster_param1 COMMA cluster_param2 COMMA
cluster_param3 COMMA cluster_param4\n"); }
;

cluster_param1
: assignment_expression { printf("cluster_param1: assignment_expression\n"); }
;

cluster_param2
: TOPOLOGY assign_colon assignment_expression { printf("cluster_param2: TOPOLOGY
assign_colon assignment_expression\n"); }
| assignment_expression { printf("cluster_param2: assignment_expression\n"); }
;

cluster_param3
: LINK_BANDWIDTH assign_colon assignment_expression { printf("cluster_param3:
LINK_BANDWIDTH assign_colon assignment_expression\n"); }
| assignment_expression { printf("cluster_param3: assignment_expression\n"); }
;

cluster_param4
: LINK_CAPACITY assign_colon assignment_expression { printf("cluster_param4:
LINK_CAPACITY assign_colon assignment_expression\n"); }
| assignment_expression { printf("cluster_param4: assignment_expression\n"); }
| LINK_CAPACITY assign_colon assignment_expression COMMA opt_cluster_param5
{ printf("cluster_param4: LINK_CAPACITY assign_colon assignment_expression COMMA
opt_cluster_param5\n"); }
| assignment_expression COMMA opt_cluster_param5 { printf("cluster_param4:
assignment_expression COMMA opt_cluster_param5\n"); }
;

opt_cluster_param5
: SCHED assign_colon assignment_expression { printf("opt_cluster_param5: SCHED
assign_colon assignment_expression\n"); }
| assignment_expression { printf("opt_cluster_param5: assignment_expression\n"); }
| SCHED assign_colon assignment_expression COMMA opt_cluster_param6 { printf
("opt_cluster_param5: SCHED assign_colon assignment_expression COMMA
opt_cluster_param6\n"); }
| assignment_expression COMMA opt_cluster_param6 { printf("opt_cluster_param5:
assignment_expression COMMA opt_cluster_param6\n"); }
| NAME assign_colon assignment_expression { printf("opt_cluster_param6: NAME
assign_colon assignment_expression\n"); }
;

opt_cluster_param6
: NAME assign_colon assignment_expression { printf("opt_cluster_param6: NAME
assign_colon assignment_expression\n"); }
| assignment_expression { printf("opt_cluster_param6: assignment_expression\n"); }
;

param_list_scheduler
: schedule_param1 { printf("param_list_scheduler: schedule_param1
opt_schedule_param2\n"); }
;

schedule_param1
: ALGO assign_colon assignment_expression { printf("schedule_param1: ALGO
assign_colon assignment_expression\n"); }
| assignment_expression { printf("schedule_param1: assignment_expression \n"); }
| ALGO assign_colon assignment_expression COMMA opt_schedule_param2 { printf
("schedule_param1: ALGO assign_colon assignment_expression COMMA\n"); }
| assignment_expression COMMA opt_schedule_param2 { printf("schedule_param1:
assignment_expression COMMA\n"); }
;

opt_schedule_param2
```

```
: NAME assign_colon assignment_expression { printf("opt_schedule_param2: NAME
assign_colon assignment_expression\n"); }
| assignment_expression { printf("opt_schedule_param2: assignment_expression"); }
;

mem_func
: GET_AVAILABLE_MEMORY OPEN_BRACKET CLOSED_BRACKET { printf("mem_func:
GET_AVAILABLE_MEMORY OPEN_BRACKET CLOSED_BRACKET\n"); }
| GET_MEMORY OPEN_BRACKET CLOSED_BRACKET { printf("mem_func: GET_MEMORY
OPEN_BRACKET CLOSED_BRACKET\n"); }
| IS_RUNNING OPEN_BRACKET CLOSED_BRACKET {printf("mem_func: IS_RUNNING
OPEN_BRACKET CLOSED_BRACKET\n"); }
| SUBMIT_JOBS OPEN_BRACKET assignment_expression CLOSED_BRACKET {printf("mem_func:
SUBMIT_JOBS OPEN_BRACKET assignment_expression\n"); }
| GET_CLOCK_SPEED OPEN_BRACKET CLOSED_BRACKET { printf("mem_func: GET_CLOCK_SPEED
OPEN_BRACKET CLOSED_BRACKET\n"); }
| RUN OPEN_BRACKET assignment_expression CLOSED_BRACKET { printf("mem_func: RUN
OPEN_BRACKET assignment_expression CLOSED_BRACKET\n"); }
| DISCARD_JOB OPEN_BRACKET assignment_expression CLOSED_BRACKET { printf
("mem_func: DISCARD_JOB OPEN_BRACKET assignment_expression CLOSED_BRACKET\n"); }
;

%%
#include <stdio.h>

extern char yytext[];

int yyerror(char *s){
    fflush(stdout);
    printf("%s\n", s);
}

int main(){
  //yydebug = 1;
  return yyparse();
}


---------------------------------- lexer.l ----------------------------------

D                               [0-9]
L                               [a-zA-Z_]
WS                              [ \t\n\v\f]

%{
#include "grammar.tab.hpp"
%}

%%
Processor                       { return(PROCESSOR); }
Link                            { return(LINK); }
Cluster                         { return(CLUSTER); }
Job                             { return(JOB); }
Memory                          { return(MEMORY); }
Scheduler                       { return(SCHEDULER); }
sched                           { return(SCHED); }
job_id                          { return(JOB_ID); }
flops_required                  { return(FLOPS_REQUIRED); }
deadline                        { return(DEADLINE); }
affinity                        { return(AFFINITY); }
algo                            { return(ALGO); }
isa                             { return(ISA); }
clock_speed                     { return(CLOCK_SPEED); }
mem_required                    { return(MEM_REQUIRED); }
l1_memory                       { return(L1_MEMORY); }
l2_memory                       { return(L2_MEMORY); }
```

```
memory_size                     { return(MEM_SIZE); }
name                            { return(NAME); }
start_point                     { return(START_POINT); }
end_point                       { return(END_POINT); }
bandwidth                       { return(BANDWIDTH); }
channel_capacity                { return(CHANNEL_CAPACITY); }
topology                        { return(TOPOLOGY); }
link_capacity                   { return(LINK_CAPACITY); }
link_bandwidth                  { return(LINK_BANDWIDTH); }
get_available_memory            { return(GET_AVAILABLE_MEMORY); }
get_memory                      { return(GET_MEMORY); }
is_running                      { return(IS_RUNNING); }
submit_jobs                     { return(SUBMIT_JOBS); }
get_clock_speed                 { return(GET_CLOCK_SPEED); }
discard_job                     { return(DISCARD_JOB); }
run                             { return(RUN); }
char                            { return(CHAR); }
else                            { return(ELSE); }
float                           { return(FLOAT); }
for                             { return(FOR); }
if                              { return(IF); }
int                             { return(INT); }
return                          { return(RETURN); }
void                            { return(VOID); }
while                           { return(WHILE); }
do                              { return(DO); }
bool                            { return(BOOL); }
mem_size                        { return(MEM_SIZE); }
memory_type                     { return(MEMORY_TYPE); }
"->"                            { return(PTR_OP); }
"&"                             { return(AMP); }
"~"                             { return(TILDE); }
"+"                             { return(PLUS); }
"-"                             { return(MINUS); }
"*"                             { return(STAR); }
"/"                             { return(DIVIDE); }
"%"                             { return(MODULUS); }
"++"                            { return(INC_OP); }
"--"                            { return(DEC_OP); }
"&&"                            { return(AND_OP); }
"||"                            { return(OR_OP); }
"<"                             { return(LT); }
">"                             { return(GT); }
"<="                            { return(LE_OP); }
">="                            { return(GE_OP); }
"=="                            { return(EQ_OP); }
"!="                            { return(NE_OP); }
"!"                             { return(NOT); }
"="                             { return(ASSIGN); }
":"                             { return(COLON); }
"("                             { return(OPEN_BRACKET); }
")"                             { return(CLOSED_BRACKET); }
"{"                             { return(OPEN_CURLY); }
"}"                             { return(CLOSED_CURLY); }
L?\"(\\.|[^\\"])*\"             { return(STRING_LITERAL); }
L?\'(\\.|[^\\'])*\'             { return(STRING_LITERAL); }
"]"                             { return(CLOSED_SQUARE); }
"["                             { return(OPEN_SQUARE); }
[0-9]*"."[0-9]+                 { return(REAL); }
[0-9]+                          { return(NUM);}
{L}({L}|{D})*                   { return(ID); }
";"                             { return(SEMI);}
","                             { return(COMMA); }
"."                             { return(DOT); }
"^"                             { return(XOR); }
"|"                             { return(PIPE); }
```

```
"?"                             { return(QUES); }
{WS}+                           {}


%%

int yywrap(void)
{
    return(1);
}


------------------------------ sample_test.c ----------------------------------

job_1 = Job(job_id=1, flops_required = 100, deadline = 200,
            mem_required = 1024, affinity = [0.2,0.5,1,2]);

job_2 = Job(job_id=2, flops_required = 5, deadline = 20,
            mem_required = 64, affinity = [0.2,0.5,1,2]);

mem1 = Memory(memory_type= 'cache', mem_size=1);
mem2 = Memory(memory_type= 'cache', mem_size=2);
mem3 = Memory(memory_type= 'cache', mem_size=2);

proc_1 = Processor(isa = 'ARM', clock_speed : 40, l1_memory = mem1,
                   sched = Scheduler(algo = "SJF", name = "my_sched_sjf"));

proc_2 = Processor(isa = 'AMD', clock_speed : 78, l1_memory = mem2);
proc_3 = Processor(isa = 'AMD', clock_speed : 78, l1_memory = mem3);

mono_sched = Scheduler(algo = "Monolithic");
cluster_1 = Cluster(processors=[proc_2, proc_3],
                    topology = "star", 100, 80, sched = mono_sched, name =
"cluster1");

run(proc_1);
run(cluster_1);
```