Compilers Lab

THE GRAMMAR AND FLEX CODE

ANURAG RAMTEKE - 150101010

NEEL MITTAL - 150101042

PRASHANSI KAMDAR - 150101047

Start

- Function_declaration

  Every line of the program will be a function declaration

- Object_declaration

  Or an object declaration

Start

- Function_declaration

  Every line of the program will be a function declaration

- Object_declaration

  Or an object declaration

object_declaration

- Dec_specifier IDENTIFIER ASGN_OP postfix_expression SEMI_COLON

  Declaring a variable with an initialisation

- Dec_specifier IDENTIFIER ASGN_OP object_expression SEMI_COLON

  Declaring an object with the corresponding constructor

- Dec_specifier IDENTIFIER

  Declaring an object or a variable without an initialisation

- Dec_specifier array_expression ASGN_OP array_initialiser SEMI_COLON

  Declaring an array with initialisation

- Dec_specifier array_expression

  Declaring an array without initialisation

Dec_specifier

- Type_specifier pointer

  Generates pointer types

- Type_specifier

  Generates data types

array_initialiser

- LEFT_CURLY array_initialiser RIGHT_CURLY

  This rule generates the curly brackets enclosing an array initialising list

- expression COMMA array_initialiser

  This rule generates the array initialising list, keeping in mind that the array can be multidimensional

- expression

  This rule ends the array initialising list

Function_declaration

- Dec_specifier identifier LEFT_PARENTHESIS argument_list RIGHT_PARENTHESIS body_or_proto

  This generates functions with arguments.

- Dec_specifier identifier LEFT_PARENTHESIS RIGHT_PARENTHESIS body_or_proto

  This generates functions without arguments.

body_or_proto

- LEFT_CURLY statement_list RIGHT_CURLY

  This will generate the body of the function.

- SEMI_COLON

  This will generate the prototype of the function.

argument_list

- Dec_specifier identifier COMMA argument_list

  Generating the argument list

- Dec_specifier identifier

  Ending the argument list

type_specifier

- INT
- VOID
- FLOAT
- CHAR
- PROC
- CLUST
- JB
- MEM
- LNK

pointer

- MUL_OP pointer
- MUL_OP

Statement_List

- Statement Statement_List

  This gives a statement and allows the language to have more number of statement following the first statement.

- Statement

  This gives the last statement for the program.

Statement

- LEFT_CURLY Statement_List RIGHT_CURLY

  This allows the program to have a block of statements encapsulated within curly brackets.
- Expression_Statement

  Expression statement production gives algebraic and logical expressions defined in the program.
- Selection_Statement

  Allows carrying out conditional operations within the program.
- Iteration_Statement

  Allows the provision of iteration over a statement or a block of statements.
- Jump_statement

  Generates jump statements
- Object_Declaration

  Allows object declarations

jump_statement

- RETURN expression

  Generates the return statement

- CONTINUE SEMI_COLON

  Generates the continue statement

- BREAK SEMI_COLON

  Generates the break statement

Selection_Statement

- IF LEFT_PARENTHESIS expression RIGHT_PARENTHESIS LEFT_CURLY Statement_List RIGHT_CURLY else LEFT_CURLY Statement_List RIGHT_CURLY

  Generates the if statement with an else. The body of the if must be enclosed in curly brackets to solve issues like dangling else.

- IF LEFT_PARENTHESIS expression RIGHT_PARENTHESIS LEFT_CURLY Statement_List RIGHT_CURLY

  Generates the if statement without an else.

Iteration_Statement

- WHILE LEFT_PARENTHESIS expression RIGHT_PARENTHESIS statement

  program can use a while loop for iteration

- FOR LEFT_PARENTHESIS expression_statement expression_statement RIGHT_PARENTHESIS statement

  this production rule gives the user for loop to be used for iteration

- FOR LEFT_PARENTHESIS expression_statement expression_statement expression RIGHT_PARENTHESIS statement

  A different way to use the for loop

Expression_Statement

- SEMI_COLON

  this is an empty expression

- Expression SEMI_COLON

  Allows user to have an expression followed by a semi colon to signify the end of the expression.

Expression

- Assignment_expression

  gives program the independence to use conditional expressions or unary expression

- Expression COMMA Assignment_expression

  allows to have an expression followed by more assignment expressions

assignment_expression

- unary_expression ASGN_OP assignment_expression

  gives identifier values to some function or conditional statement or even a constant

- conditional_expression

  generates all kinds of algebraic and logical expressions in the program.

- object_expression

  generates the object assignment expression

Conditional_Expression

- logical_or_expression

  logical or expression signifies the OR functionality for the language and is going to be at the highest level of the parse tree and thus placing it at the least priority.

Logical_or_expression

- logical_or_expression OR_OP logical_and_expression

  The logical and expression must be evaluated before the or expression. Hence this expression has higher priority in the Language.

- logical_and_expression

  To generate the last term in the or expression having and or some other operation having higher priority.

Logical_and_expression

- logical_and_expression

  production rule to generate the last term for and expression

- bitwise_or_expression

  production rule that can be used to create bitwise or in the terms

- logical_and_expression AND_OP bitwise_or_expression

  used in the language for generation of bitwise or with the and expression an an AND operator

bitwise_or_expression

- bitwise_or_expression

  create last or more terms with bitwise or operations in the expression

- bitwise_xor_expression

  bitwise xor is higher priority than bitwise or expression and thus is produced after bitwise or expression and hence lower in the parse tree. This production rule gives bitwise xor terms added to the expressions

- bitwise_or_expression BTW_OR bitwise_xor_expression

  creating bitwise xor terms with expression having bitwise or

bitwise_xor_expression

- and_expression

  bitwise and has higher priority for the grammar defined for our language and hence follows in the production rule of bitwise xor expression

- bitwise_xor_expression XOR_OP and_expression

  generates and expression term with bitwise xor expression terms

and_expression

- equality_expression

  the equality expression allows for relational expressions to be used in the program.

- and_expression AMPERSAND equality_expression

  Generates an AND expression along with equality expression which by itself can generate relational expressions.

equality_expression

- relational_expression

  relational expression consists of combination of greater than, less than or equal to expressions to be generated in the code.

- equality_expression EQ_OP relational_expression

  This allows generation of equality expression along with an equality operand and relational expression.

- equality_expression NE_OP relational_expression

  This again allows the generation of an equality expression and relational expression but with a non-equality operand.

relational_expression

- shift_expression

  generates binary shift operation expression in the program
- relational_expression LESS_THAN shift_expression

  generates relational expression followed by less_than operand and binary shift expression(s).
- relational_expression GREATER_THAN shift_expression

  generates relational expression and shift expression(s), where the binary shift is appended after a greater than symbol which is again a relational symbol.
- relational_expression LE_OP shift_expression

  this production gives a relational expression with relational operands less_than_equal_to and followed by binary shift expression(s).
- relational_expression GE_OP shift_expression

  the production rule gives relational and shift expression together joined using a greater than or equal to operand

shift_expression

- additive_expression

  allows the user to generate addition/subtraction expressions in the program.

- shift_expression LEFT_OP additive_expression

  allows the conjuction of shift and addition expression in the code using left_shift binary operation

- shift_expression RIGHT_OP additive_expression

  allows the generation of conjunction of addition expression(which can further go to other expressions which are higher in priority than the addition/subtraction symbol) and binary shift expression using a right shift operand.

additive_expression

- multiplicative_expression

  multiplicative expression has higher priority than additive expression and hence is lower in the parse tree according to this production rule for this grammar.

- additive_expression PLUS multiplicative_expression

  generates conjuction of additive expressions with a multiplicative expression using an addition symbol.

- additive_expression MINUS multiplicative_expression

  generates additive expression in the program followed by a subtract sign and a multiplicative expression.

multiplicative_expression

- unary_expression

  unary into some data type has higher priority than multiplicative expression. Thus it is generated lower to the multiplicative production rule for the parse tree.

- multiplicative_expression MUL_OP unary_expression

  this production rule allows the conjunction of multiplicative expressions (i.e. *, /,%) with unary expression using a multiply operand.

- multiplicative_expression DIV_OP unary_expression

  this again allows conjunction of unary expressions with a multiplicative expression using a divide operand.

- multiplicative_expression MOD_OP unary_expression

  Multiplicative expression(s) followed by mod(%) and unary expression

unary_expression

- postfix_expression

  generates postfix expressions (variable++, variable--,variable etc.)

- INC_OP unary_expression

  these give prefix expressions like ++variable.

- DEC_OP unary_expression

  this again gives prefix expressions like --variable

- unary_operator unary_expression

  gives unary operators like ~, !, + etc with a unary expression.

unary_operator

- PLUS
- MINUS
- BTW_NOT
- NOT_OP
- MUL_OP
- AMPERSAND

postfix_expression

- primary_expression

  generates identifiers, constants, string_literals etc.

- array_expression

  generates arrays for the programs

- function_expression

  Function evaluation gets higher priority than every arithmetic or relational operation

- postfix_expression INC_OP

  generates postfix expressions like variable++

- postfix_expression DEC_OP

  Generates postfix decreasing expressions like variable—

array_expression

- array_expression LEFT_BRACKET expression RIGHT_EXPRESSION

  This can generate multidimensional arrays.

- Primary_expression LEFT_BRACKET expression RIGHT_BRACKET

  This will generate the array identifier.

primary_expression

- IDENTIFIER DEREF_OP IDENTIFIER
- IDENTIFIER
- CONSTANT
- STRING_LITERAL
- LEFT_PARENTHESIS expression RIGHT_PARENTHESIS

  Brackets get the highest priority

object_expression

- link_object
- memory_object
- job_object
- cluster_object

cluster_object

- CLUSTER LEFT_PARENTHESIS proc_arr_arg COMMA topology_arg COMMA link_band_arg COMMA link_cap_arg narp
- processor_object

proc_arr_arg

- PROCESSORS ASGN_OP IDENTIFIER
- PROCESSORS ASGN_OP LEFT_BRACKET cluster_list RIGHT_BRACKET
- IDENTIFIER
- LEFT BRACKET cluster_list RIGHT_BRACKET

cluster_list

- cluster_object COMMA cluster_list
- cluster_object

topology_arg

- TOPOLOGY ASGN_OP STRING_LITERAL
- STRING_LITERAL

link_band_arg

- LINK_BANDWIDTH ASGN_OP conditional_expression
- conditional_expression

link_cap_arg

- LINK_CAPACITY ASGN_OP conditional_expression
- conditional_expression

narp

- COMMA conditional_expression RIGHT_PARENTHESIS
- COMMA NAME ASGN_OP conditional_expression
- RIGHT_PARENTHESIS

processor_object

- PROCESSOR LEFT_PARENTHESIS isa_args COMMA clock_args COMMA mem_args narp

Isa_args

- ISA ASGN_OP PROC_TYPE
- PROC_TYPE

Clock_args

- CLOCK_SPEED ASGN_OP CONSTANT
- CONSTANT

mem_args

- MEM1 ASGN_OP memory COMMA MEM2 ASGN_OP memory
- MEM1 ASGN_OP memory COMMA memory
- MEM1 ASGN_OP memory
- memory COMMA MEM2 ASGN_OP memory
- memory COMMA memory
- memory

memory

- memory_object
- IDENTIFIER

link_object

- LINK LEFT_PARENTHESIS start_args COMMA end_args COMMA link_band_args COMMA link_cap_args narp
- IDENTIFIER

start_args

- START_POINT ASGN_OP conditional_expression
- conditional_expression

end_args

- END_POINT ASGN_OP conditional_expression
- conditional_expression

memory_object

- MEMORY LEFT_PARENTHESIS mem_type_args COMMA mem_size_args narp

mem_type_args

- MEMORY_TYPE ASGN_OP MEM_TYPES
- MEM_TYPES

mem_size_args

- MEMORY_SIZE ASGN_OP conditional_expression
- conditional_expression

job_object

- JOB LEFT_PARENTHESIS job_id_args COMMA flops_args COMMA deadline_args COMMA mem_required_args COMMA affinity_args RIGHT_PARENTHESIS

job_id_args

- JOB_ID ASGN_OP conditional_expression
- conditional_expression

flops_args

- FLOPS_REQUIRED ASGN_OP conditional_expression
- conditional_expression

deadline_args

- DEADLINE ASGN_OP conditional_expression
- conditional_expression

mem_required_args

- MEM_REQUIRED ASGN_OP conditional_expression
- conditional_expression

affinity_args

- AFFINITY ASGN_OP LEFT_BRACKET list RIGHT_BRACKET
- LEFT_BRACKET list RIGHT_BRACKET

list

- CONSTANT
- CONSTANT COMMA list

function_expression

- RUN LEFT_PARENTHESIS cluster_list RIGHT_PARENTHESIS
- WAIT LEFT_PARENTHESIS RIGHT_PARENTHESIS
- DISCARD_JOB LEFT_PARENTHESIS jon_list RIGHT_PARENTHESIS
- STOP LEFT_PARENTHESIS IDENTIFIER RIGHT_PARENTHESIS
- IDENTIFIER LEFT_PARENTHESIS RIGHT_PARENTHESIS
- IDENTIFIER LEFT_PARENTHESIS argument_expression_list RIGHT_PARENTHESIS
- IDENTIFIER DOT object_function.

object_function

- processor_function
- job_function
- link_function
- memory_function

memory_function

- GET_AVAILABLE_MEMORY LEFT_PARENTHESIS RIGHT_PARENTHESIS

Job_function

- GET_JOB_AFFINITY LEFT_PARENTHESIS RIGHT_PARENTHESIS
- GET_JOB_MEMORY LEFT_PARENTHESIS RIGHT_PARENTHESIS
- GET_FLOPS LEFT_PARENTHESIS RIGHT_PARENTHESIS
- GET_DEADLINE LEFT_PARENTHESIS RIGHT_PARENTHESIS

These production rules are added to give user a better simulation. The functions when made into the language will let user get the information about the job.

processor_function

- IS_RUNNING LEFT_PARENTHESIS RIGHT_PARENTHESIS
- SUBMIT_JOBS LEFT_PARENTHESIS job_list RIGHT_PARENTHESIS
- GET_CLOCK_SPEED LEFT_PARENTHESIS RIGHT_PARENTHESIS
- GET_PROC_TYPE LEFT_PARENTHESIS RIGHT_PARENTHESIS
- IS_PROCESSOR LEFT_PARENTHESIS RIGHT_PARENTHESIS

The last two production rules are again added to the language for letting the user create his own scheduling algorithms in the language creating a better simulation.

cluster_function

- GET_PROCESSOR LEFT_PARENTHESIS RIGHT_PARENTHESIS
- GET_PROCESSOR LEFT_PARENTHESIS primary_expression RIGHT_PARENTHESIS
  gives the list of processor of the cluster, a function in the language that will be checked by the lexer
- IS_PROCESSOR LEFT_PARENTHESIS RIGHT_PARENTHESIS
  this is a string in the language that allows functionality of checking a cluster or a processor
- SUBMIT_JOBS LEFT_PARENTHESIS job_list RIGHT_PARENTHESIS

job_list

- IDENTIFIER COMMA job_list
- IDENTIFIER
- job_object COMMA job_list
- job_object

```
/* table.h */

#define         BREAK                           0
#define     CHAR                                1
#define     CONTINUE                            2
#define     ELSE                                3
#define     FLOAT                               4
#define     FOR                                 5
#define     IF                                  6
#define     INT                                 7
#define     RETURN                              8
#define     VOID                                9
#define     WHILE                               10
#define     PROC                                11
#define     LNK                                 12
#define     JB                                  13
#define     CLUST                               14
#define     CLUSTER                             15
#define     PROCESSOR                           16
#define     ISA                                 17
#define     PROC_TYPE                           18
#define     CLOCK_SPEED                         19
#define     MEM1                                20
#define     MEM2                                21
#define     NAME                                22
#define     TOPOLOGY                            23
#define     LINK_BANDWIDTH                      24
#define     LINK_CAPACITY                       25
#define     LINK                                26
#define     START_POINT                         27
#define     END_POINT                           28
```

```
#define   MEMORY_TYPE                    29
#define   MEM_TYPE                       30
#define   MEMORY_SIZE                    31
#define   JOB                            32
#define   JOB_ID                         33
#define   FLOPS_REQUIRED                 34
#define   DEADLINE                       35
#define   MEM_REQUIRED                   36
#define   AFFINITY                       37
#define   RUN                            38
#define   WAIT                           39
#define   DISCARD_JOB                    40
#define   STOP                           41
#define   GET_AVAILABLE_MEMORY           42
#define   GET_JOB_AFFINITY               43
#define   GET_JOB_MEMORY                 44
#define   GET_FLOPS                      45
#define   GET_DEADLINE                   46
#define   IS_RUNNING                     47
#define   SUBMIT_JOBS                    48
#define   GET_FLOPS_SPEED                49
#define   GET_PROC_TYPE                  50
#define   IS_PROCESSOR                   51
#define   GET_PROCESSOR                  52
#define   MEM                            53
#define   IDENTIFIER                     54
#define   CONSTANT                       55
#define   STRING_LITERAL                 56
#define   RIGHT_OP                       57
#define   LEFT_OP                        58
#define   INC_OP                         59
#define   DEC_OP                         60
```

```
#define   DREF_OP                        61
#define   AND_OP                         62
#define   OR_OP                          63
#define   LE_OP                          64
#define   GE_OP                          65
#define   EQ_OP                          66
#define   NE_OP                          67
#define   SEMI_COLON                     68
#define   LEFT_CURLY                     69
#define   RIGHT_CURLY                    70
#define   COMMA                          71
#define   ASGN_OP                        72
#define   LEFT_PARENTEHSIS               73
#define   RIGHT_PARENTHESIS              74
#define   LEFT_BRACKET                   75
#define   RIGHT_BRACKET                  76
#define   DOT                            77
#define   AMPERSAND                      78
#define   NOT_OP                         79
#define   BTW_NOT                        80
#define   MINUS                          81
#define   PLUS                           82
#define   MUL_OP                         83
#define   DIV_OP                         84
#define   MOD_OP                         85
#define   LESS_THAN                      86
#define   GREATER_THAN                   87
#define   XOR_OP                         88
#define   BTW_OR                         89
#define   INVALID                        90
#define   MEMORY                         91
#define   PROCESSORS                     92
```

```
/* grammar.lex*/


D           [0-9]
L           [a-zA-Z_]
H           [a-fA-F0-9]
E           [Ee][+-]?{D}+
FS          (f|F|l|L)
IS          (u|U|l|L)*

%{
#include <stdio.h>
#include "table.h"

void count();
%}

%%

"break"                  { printf("<"); count(); printf(",%s> ","BREAK"); return(BREAK);}
"char"                  { printf("<"); count(); printf(",%s> ","CHAR"); return(CHAR);}
"continue"              { printf("<"); count(); printf(",%s> ","CONTINUE"); return(CONTINUE);}
"else"                  { printf("<"); count(); printf(",%s> ","ELSE"); return(ELSE);}
"float"                 { printf("<"); count(); printf(",%s> ","FLOAT"); return(FLOAT);}
"for"                   { printf("<"); count(); printf(",%s> ","FOR"); return(FOR);}
"if"                     { printf("<"); count(); printf(",%s> ","IF"); return(IF);}
"int"                   { printf("<"); count(); printf(",%s> ","INT"); return(INT);}
"return"                { printf("<"); count(); printf(",%s> ","RETURN"); return(RETURN);}
"void"                  { printf("<"); count(); printf(",%s> ","VOID"); return(VOID);}
"while"                  { printf("<"); count(); printf(",%s> ","WHILE"); return(WHILE);}
"proc"                   { printf("<"); count(); printf(",%s> ","PROC"); return(PROC);}
"lnk"                    { printf("<"); count(); printf(",%s> ","LNK"); return(LNK);}
```

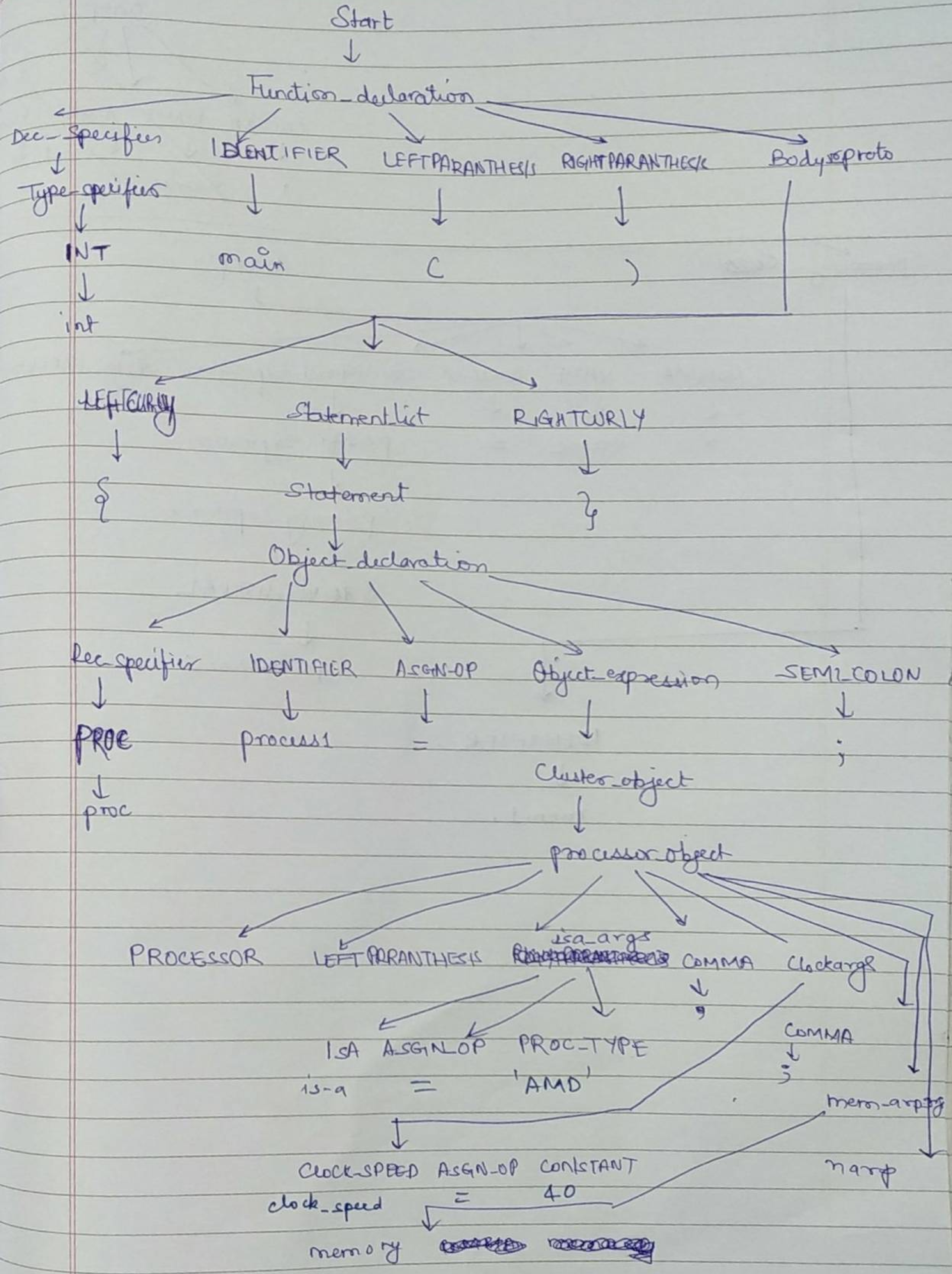
```
"jb"                                  { printf("<"); count(); printf(",%s> ","JB"); return(JB);}
"clust"                               { printf("<"); count(); printf(",%s> ","CLUST"); return(CLUST);}
"Cluster"                            { printf("<"); count(); printf(",%s> ","CLUSTER"); return(CLUSTER);}
"Processor"                           { printf("<"); count(); printf(",%s> ","PROCESSOR"); return(PROCESSOR);}
"processors"                  { printf("<"); count(); printf(",%s> ","PROCESSORS"); return(PROCESSORS);}
"isa"                                 { printf("<"); count(); printf(",%s> ","ISA"); return(ISA);}
('ARM')|('AMD')|('CDC')|('MIPS')     { printf("<"); count(); printf(",%s> ","PROC_TYPE"); return(PROC_TYPE);}
"clock_speed"                         { printf("<"); count(); printf(",%s> ","CLOCK_SPEED"); return(CLOCK_SPEED);}
"l1_memory"                   { printf("<"); count(); printf(",%s> ","MEM1"); return(MEM1);}
"l2_memory"                   { printf("<"); count(); printf(",%s> ","MEM2"); return(MEM2);}
"name"                                { printf("<"); count(); printf(",%s> ","NAME"); return(NAME);}
"topology"                            { printf("<"); count(); printf(",%s> ","TOPOLOGY"); return(TOPOLOGY);}
"Link_bandwidth"                       { printf("<"); count(); printf(",%s> ","LINK_BANDWIDTH"); return(LINK_BANDWIDTH);}
"link_capacity"                        { printf("<"); count(); printf(",%s> ","LINK_CAPACITY"); return(LINK_CAPACITY);}
"Link"                                { printf("<"); count(); printf(",%s> ","LINK"); return(LINK);}
"start_point"                          { printf("<"); count(); printf(",%s> ","START_POINT"); return(START_POINT);}
"end_point"                           { printf("<"); count(); printf(",%s> ","END_POINT"); return(END_POINT);}
"memory_type"                        { printf("<"); count(); printf(",%s> ","MEMORY_TYPE"); return(MEMORY_TYPE);}
('primary')|('secondary')|('cache')  { printf("<"); count(); printf(",%s> ","MEM_TYPE"); return(MEM_TYPE);}
"mem_size"                            { printf("<"); count(); printf(",%s> ","MEMORY_SIZE"); return(MEMORY_SIZE);}
"Job"                                 { printf("<"); count(); printf(",%s> ","JOB"); return(JOB);}
"job_id"                             { printf("<"); count(); printf(",%s> ","JOB_ID"); return(JOB_ID);}
"flops_required"                     { printf("<"); count(); printf(",%s> ","FLOPS_REQUIRED"); return(FLOPS_REQUIRED);}
"deadline"                            { printf("<"); count(); printf(",%s> ","DEADLINE"); return(DEADLINE);}
"mem_required"                       { printf("<"); count(); printf(",%s> ","MEM_REQUIRED"); return(MEM_REQUIRED);}
"affinity"                           { printf("<"); count(); printf(",%s> ","AFFINITY"); return(AFFINITY);}
"run"                                 { printf("<"); count(); printf(",%s> ","RUN"); return(RUN);}
"wait"                                { printf("<"); count(); printf(",%s> ","WAIT"); return(WAIT);}
"discard_job"                         { printf("<"); count(); printf(",%s> ","DISCARD_JOB"); return(DISCARD_JOB);}
"stop"                               { printf("<"); count(); printf(",%s> ","STOP"); return(STOP);}
"Get_available_memory"                { printf("<"); count(); printf(",%s> ","GET_AVAILABLE_MEMORY"); return(GET_AVAILABLE_MEMORY);}
"get_job_affinity"                    { printf("<"); count(); printf(",%s> ","GET_JOB_AFFINITY"); return(GET_JOB_AFFINITY);}
```

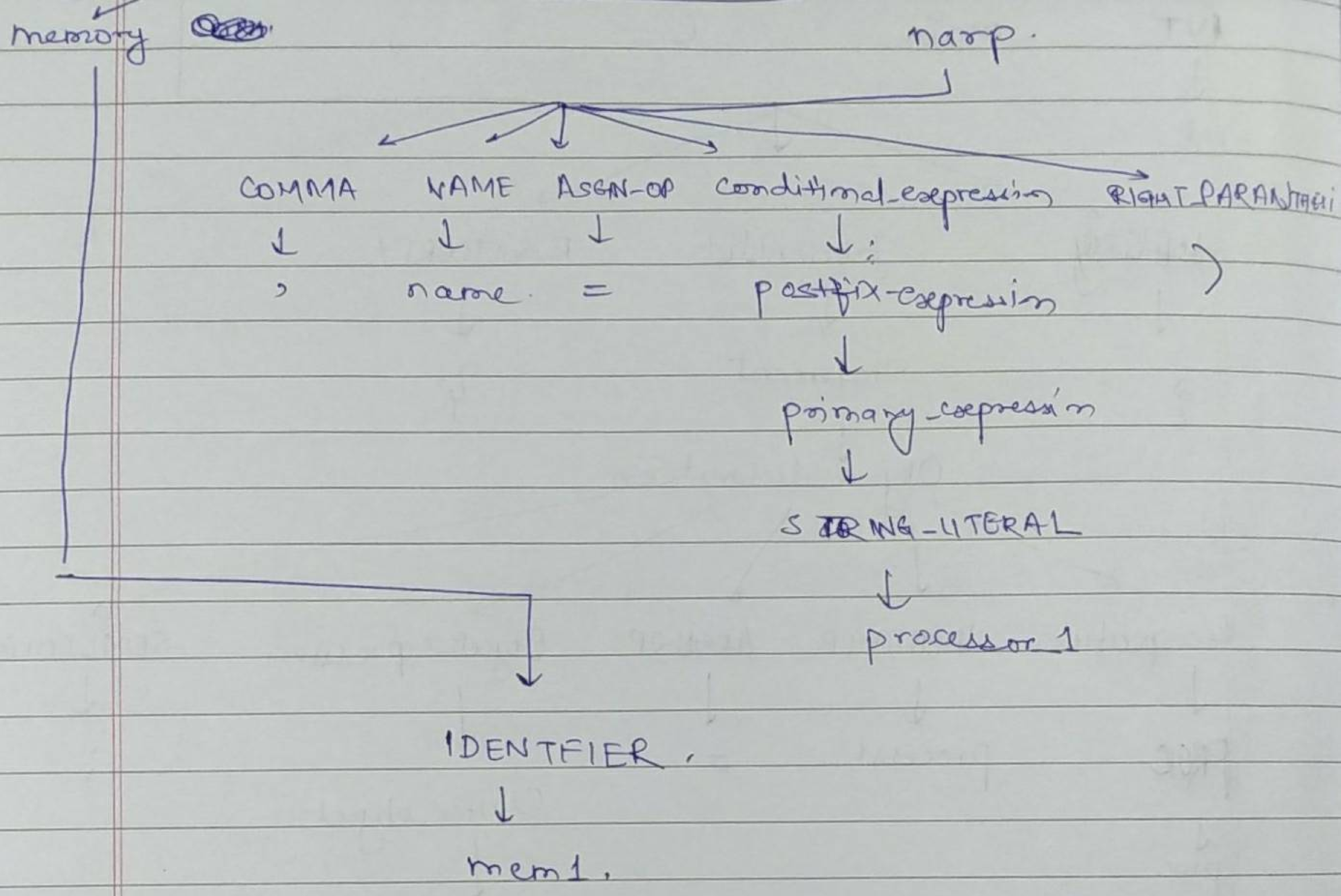
```
"get_memory"                 { printf("<"); count(); printf(",%s> ","GET_JOB_MEMORY"); return(GET_JOB_MEMORY);}
"get_flops"                  { printf("<"); count(); printf(",%s> ","GET_FLOPS"); return(GET_FLOPS);}
"get_deadline"               { printf("<"); count(); printf(",%s> ","GET_DEADLINE"); return(GET_DEADLINE);}
"is_running"                 { printf("<"); count(); printf(",%s> ","IS_RUNNING"); return(IS_RUNNING);}
"submit_jobs"                { printf("<"); count(); printf(",%s> ","SUBMIT_JOBS"); return(SUBMIT_JOBS);}
"get_flops_speed"            { printf("<"); count(); printf(",%s> ","GET_FLOPS_SPEED"); return(GET_FLOPS_SPEED);}
"get_proc_type"              { printf("<"); count(); printf(",%s> ","GET_PROC_TYPE"); return(GET_PROC_TYPE);}
"is_processor"               { printf("<"); count(); printf(",%s> ","IS_PROCESSOR"); return(IS_PROCESSOR);}
"get_processor"              { printf("<"); count(); printf(",%s> ","GET_PROCESSOR"); return(GET_PROCESSOR);}
"Memory"                     { printf("<"); count(); printf(",%s> ","MEMORY"); return(MEMORY);}
"mem"                        { printf("<"); count(); printf(",%s> ","MEM"); return(MEM);}

{L}({L}|{D})*                { printf("<"); count(); printf(",%s> ","IDENTIFIER"); return(IDENTIFIER);}

0[xX]{H}+{IS}?               { printf("<"); count(); printf(",%s> ","CONSTANT"); return(CONSTANT);}
0{D}+{IS}?                   { printf("<"); count(); printf(",%s> ","CONSTANT"); return(CONSTANT);}
{D}+{IS}?                    { printf("<"); count(); printf(",%s> ","CONSTANT"); return(CONSTANT);}
L?'(\\.|[^\\'])+'            { printf("<"); count(); printf(",%s> ","CONSTANT"); return(CONSTANT);}

{D}+{E}{FS}?                 { printf("<"); count(); printf(",%s> ","CONSTANT"); return(CONSTANT);}
{D}*"."{D}+({E})?{FS}?       { printf("<"); count(); printf(",%s> ","CONSTANT"); return(CONSTANT);}
{D}+"."{D}*({E})?{FS}?       { printf("<"); count(); printf(",%s> ","CONSTANT"); return(CONSTANT);}

\"(\\.|[^\\"])*\"             { printf("<"); count(); printf(",%s> ","STRING_LITERAL"); return(STRING_LITERAL);}

">>"                         { printf("<"); count(); printf(",%s> ","RIGHT_OP"); return(RIGHT_OP);}
"<<"                         { printf("<"); count(); printf(",%s> ","LEFT_OP"); return(LEFT_OP);}
"++"                         { printf("<"); count(); printf(",%s> ","INC_OP"); return(INC_OP);}
"--"                         { printf("<"); count(); printf(",%s> ","DEC_OP"); return(DEC_OP);}
"->"                         { printf("<"); count(); printf(",%s> ","DREF_OP"); return(DREF_OP);}
"&&"                        { printf("<"); count(); printf(",%s> ","AND_OP"); return(AND_OP);}
"||"                         { printf("<"); count(); printf(",%s> ","OR_OP"); return(OR_OP);}
```

```
"<="                                { printf("<"); count(); printf(",%s> ","LE_OP"); return(LE_OP);}
">="                               { printf("<"); count(); printf(",%s> ","GE_OP"); return(GE_OP);}
"=="                                { printf("<"); count(); printf(",%s> ","EQ_OP"); return(EQ_OP);}
"!="                                { printf("<"); count(); printf(",%s> ","NE_OP"); return(NE_OP);}
";"                                 { printf("<"); count(); printf(",%s> ","SEMI_COLON"); return(SEMI_COLON);}
("{"|"<%")                          { printf("<"); count(); printf(",%s> ","LEFT_CURLY"); return(LEFT_CURLY);}
("}"|"%>")                          { printf("<"); count(); printf(",%s> ","RIGHT_CURLY"); return(RIGHT_CURLY);}
","                                { printf("<"); count(); printf(",%s> ","COMMA"); return(COMMA);}
"="                                 { printf("<"); count(); printf(",%s> ","ASGN_OP"); return(ASGN_OP);}
":"                                 { printf("<"); count(); printf(",%s> ","ASGN_OP"); return(ASGN_OP);}
"("                                 { printf("<"); count(); printf(",%s> ","LEFT_PARENTEHSIS"); return(LEFT_PARENTEHSIS);}
")"                                 { printf("<"); count(); printf(",%s> ","RIGHT_PARENTHESIS"); return(RIGHT_PARENTHESIS);}
("["|"<:")                          { printf("<"); count(); printf(",%s> ","LEFT_BRACKET"); return(LEFT_BRACKET);}
("]"|":>")                          { printf("<"); count(); printf(",%s> ","RIGHT_BRACKET"); return(RIGHT_BRACKET);}
"."                                 { printf("<"); count(); printf(",%s> ","DOT"); return(DOT);}
"&"                                { printf("<"); count(); printf(",%s> ","AMPERSAND"); return(AMPERSAND);}
"!"                                 { printf("<"); count(); printf(",%s> ","NOT_OP"); return(NOT_OP);}
"~"                                 { printf("<"); count(); printf(",%s> ","BTW_NOT"); return(BTW_NOT);}
"-"                                { printf("<"); count(); printf(",%s> ","MINUS"); return(MINUS);}
"+"                                 { printf("<"); count(); printf(",%s> ","PLUS"); return(PLUS);}
"*"                                 { printf("<"); count(); printf(",%s> ","MUL_OP"); return(MUL_OP);}
"/"                                 { printf("<"); count(); printf(",%s> ","DIV_OP"); return(DIV_OP);}
"%"                                 { printf("<"); count(); printf(",%s> ","MOD_OP"); return(MOD_OP);}
"<"                                 { printf("<"); count(); printf(",%s> ","LESS_THAN"); return(LESS_THAN);}
">"                                 { printf("<"); count(); printf(",%s> ","GREATER_THAN"); return(GREATER_THAN);}
"^"                                 { printf("<"); count(); printf(",%s> ","XOR_OP"); return(XOR_OP);}
"|"                                 { printf("<"); count(); printf(",%s> ","BTW_OR"); return(BTW_OR);}

[ \t\v\n\f]                         { count();}
.                                  { printf("<"); count(); printf(",%s> ","INVALID"); return(INVALID);}

%%
```

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


int column = 0;

void count()
{
    int i;

    for (i = 0; yytext[i] != '\0'; i++)
            if (yytext[i] == '\n')
                        column = 0;
            else if (yytext[i] == '\t')
                        column += 8 - (column % 8);
            else
                        column++;

    ECHO;
}
```

```
/* example*/
int main()
{
    proc process1=PROCESSOR(isa='AMD',clock_speed=40,mem1,name="processor1");
}

/*lexical analysis of the above example*/


<int,INT>  <main,IDENTIFIER> <(,LEFT_PARENTEHSIS> <),RIGHT_PARENTHESIS>

<{,LEFT_CURLY>

    <proc,PROC>  <process1,IDENTIFIER> <=,ASGN_OP> <PROCESSOR,IDENTIFIER> <(,LEFT_PARENTEHSIS> <isa,ISA>
<=,ASGN_OP> <'AMD',PROC_TYPE> <,,COMMA> <clock_speed,CLOCK_SPEED> <=,ASGN_OP> <40,CONSTANT> <,,COMMA>
<mem1,IDENTIFIER> <,,COMMA> <name,NAME> <=,ASGN_OP> <"processor1",STRING_LITERAL> <),RIGHT_PARENTHESIS>
<;,SEMI_COLON>

<},RIGHT_CURLY>
```


Start
Function_declaration
Dec_specifier
Type_specifier
INT
int
IDENTIFIER
main
LEFTPARANTHESIS
(
RIGHTPARANTHESIS
)
Body_or_proto
LEFTCURLY
{
Statement_list
Statement
Object_declaration
RIGHTCURLY
}
Dec_specifier
PROC
proc
IDENTIFIER
process1
ASGN-OP
=
Object_expression
Cluster_object
processor_object
SEMI_COLON
;
PROCESSOR
LEFTPARANTHESIS
isa_args
COMMA
,
Clockargs
ISA
is-a
ASGN_OP
=
PROC_TYPE
'AMD'
COMMA
;
mem_args
narg
CLOCK_SPEED
clock_speed
ASGN_OP
=
CONSTANT
40
memory


memory
narp.
COMMA
NAME
ASGN-OP
conditional_expression
RIGHT_PARANTHESI
,
name
=
)
postfix-expression
primary_expression
STRING-LITERAL
processor_1
IDENTFIER.
mem1.