# COSMOS

Shashank Shanbhag - CS20BTECH11061
Ayush Jha-CS20BTECH11006
Avula Dinesh - CS20BTECH11005
Yuvraj Singh Shekhawat- CS20BTECH11057
Rohan Atkurkar - CS20BTECH11041

September 20, 2022

## Contents

## 1 About COSMOS:-

### 1.1 Motivation

The word COSMOS means universe. Universe has a variety of astronomical problems mostly related to astrophysics and astrometry. During problem solving, There are often many variables and different units. For example, the Universe has sizes ranging from absolutely nothing to nearly infinite so for distance alone we have multiple units like meters(m), Astronomical Unit(A.U), light year(ly), parsec(pc) e.t.c. It is often tiresome and confusing to learn so many formulas and keep track of units and dimensions at each point during calculation.
We aim to eliminate these drawbacks by our language COSMOS.

### 1.2 Brief Description

COSMOS is predominantly built for space scientists and learning students. It is a statically typed programming language, and it serves as a scientific calculator for astronomical problems.
It aims at assisting people with no background in programming. Various familiar data types and functions implementing a variety of formulas make COSMOS a user friendly language.

### 1.3 Uses

Users don't need to be too wary of units and store bookish formulas in their memory cells.
It supports computations related to basic Gravitation, binary star systems, star magnitudes, astronomical objects, blackbodies, special relativity, light waves, and unit conversion.
Since Astronomy usually deals with huge numbers, the built-in scientific notation constant of COSMOS comes in handy. It can handle numbers ranging from $10^{-32768}$ to $10^{32767}$ with 6 digit mantissa precision.
We will be using Flex for lexical analysis and Yacc for syntax analysis.

# 2 How to use COSMOS:-

## 2.1 Constants and measurement metrics:-

This is a table that contains the details of some constants and measurement metrics that are used in the language.

| **Measuring Metric** | **Unit & How to use the Function in the language** |
|---|---|
| Mass | Kg |
| Density | $\frac{kg}{m^3}$ |
| Distance | m |
| Time | s |
| Speed | m/s |
| Acceleration | $\frac{m}{s^2}$ |
| Temperature | k |
| Force | N or $\frac{kgm}{s^2}$ |
| Energy | J or $\frac{kgm^2}{s^4}$ |
| Momentum | kg m/s |

Table 1: Measuring metrics

| **Constant** | **Unit & How to use the Constant in the language** |
|---|---|
| Solar mass | $1.989\times10^{30}$ kg |
| $R_e$ Radius of Earth | $6.371\ \times10^{3}$m |
| $M_e$ | $5.972\ \times10^{24}$m |
| $R_s$ | $1.989\times10^{30}$m |
| Astronomical Unit | $1.496\times10^{11}$m |
| Light year | $9.460\times10^{15}$m |
| Gravitational Constant(G) | $6.674e\times10^{-11}$ |
| speed of light | $2.997\ ^{8}$ m/s |

Table 2: Constants

## 2.2 Operators:-

| Operators | Use of operator | Associativity |
|---|---|---|
| + | addition | left-right |
| - | subtraction | |
| * | multiplication | |
| / | division | |
| ∧ | power | |
| > | Greater than | |
| < | Less than | |
| >= | Greater than or equal to | |
| <= | Less than or equal to | |
| == | Test equivalence | |
| ! = | Test inequality | |
| && | AND Left | |
| \|\| | OR Left | |
| % | modulo | |
| . | access (used for structs) | |
| = | Assignment | right-left |
| ! | logical not | |

Table 3: Operators used in the language

## 2.3 Operator Precedence :

| Order Of precedence: |
|---|
| . ( ) [ ] |
| ! |
| ∧ |
| * / % |
| + - |
| < <= > >= |
| == != |
| && |
| \|\| |
| = |

Table 4: Operator precedence

## 2.4 Comments

Comments are of the form : '#/...../#' . Comment line starts with a # and ends with another #, the part in between is considered as a comment.

### 2.5 Keywords used in the language:-

| Keyword | | | | |
|---|---|---|---|---|
| bool | char | short | int | float |
| string | auto | astrobject | mass | density |
| dist | time | speed | acc | energy |
| force | freq | ly | AU | solar_mass |
| parsec | temp | arcsec | repeat | break |
| continue | if | else | return | void |
| struct | proc | | | |

### 2.6 Data Types:-

We are using both primitive and non-primitive data types.

#### 2.6.1 Primitive data types:-

There are 2 types of primitive data types we are using:-

1. Boolean(as bool)
   A boolean data type can only take 2 values: true(1) or false(0).
2. Numeric
   There will be 2 parts under numeric data type:
   (a) Character(as char)
   A character variable can take all ASCII values ranging from 0-127.
   (b) Integral
   There will be two parts under integral data type:
   - Integer
     Here we can split into 2 parts:
     i. short
     Short data type is used to represent exponent in scientific notation.
     ii. int
     Int data type is used to represent integer numbers.
   - Floating point(as float) This is used to represent decimal numbers or floating point numbers. It will be used to represent mantissa in scientific notation.

#### 2.6.2 Non-Primitive data types:

All non-primitive data types will be represented in the format of scientific notation . Format:
Scientific notation- A pair consisting of $< float > e < shortint >$
With precision upto 6 decimals

Astronomical Data Types:-

1. mass(in kg)
   mass data type represents the mass of any planet, star or astronomical object.

2. density(in kg/m3)
   density data type represents the density of any planet, star or astronomical object.

3. dist(in m)
   dist data type represents the distance between astronomical objects.

4. speed(in m/s)
   speed data type represents the speed of the astronomical object.It might be used to represent escape velocity, orbital velocity, etc.

5. acc(in m/s2)
   acc data type represents the acceleration of the astronomical object. It might be used to represent acceleration due to gravity(g).

6. time(in s)
   time data type represents the time taken into consideration while solving the problem.

7. force(in N)
   force data type usually represents the gravitational force of attraction between astronomical objects.

8. energy(in J)
   energy data type represents kinetic energy, potential energy, binding energy, etc. for astronomical objects.

9. ly
   ly(Light year) is the distance light travels in one year. We will use this data type to represent extremely large distances.

10. AU
    AU(Astronomical unit) is roughly the distance from earth and sun. We will use this data type to represent extremely large distances.

11. solar_mass
    solar_mass(Stellar mass) is a phrase that is used by astronomers to describe the mass of a star. We will use this data type to represent large masses.

12. parsec
    parsec is equal to about 3.26 light years . One parsec corresponds to the distance at which the mean radius of the earth's orbit subtends an angle of one second of arc. We will use this data type to represent extremely large distances.

13. temp(in K)
    temp data type will be used to represent the temperature of any star, or the average temperature of empty space between celestial bodies.

14. arcsec
    arcsec is the data type that will be used to represent angle. arcsec equals 1/3600 of a degree.

    Data type formed from combination of primitive data types

15. string
    string data type will be used to represent a combination of characters.

    Struct data type

16. astrobject
    astroject(Astronomical_object) data type is used to represent spherical astronomical objects. It will be a struct with mass and radius of the astronomical object as its members.

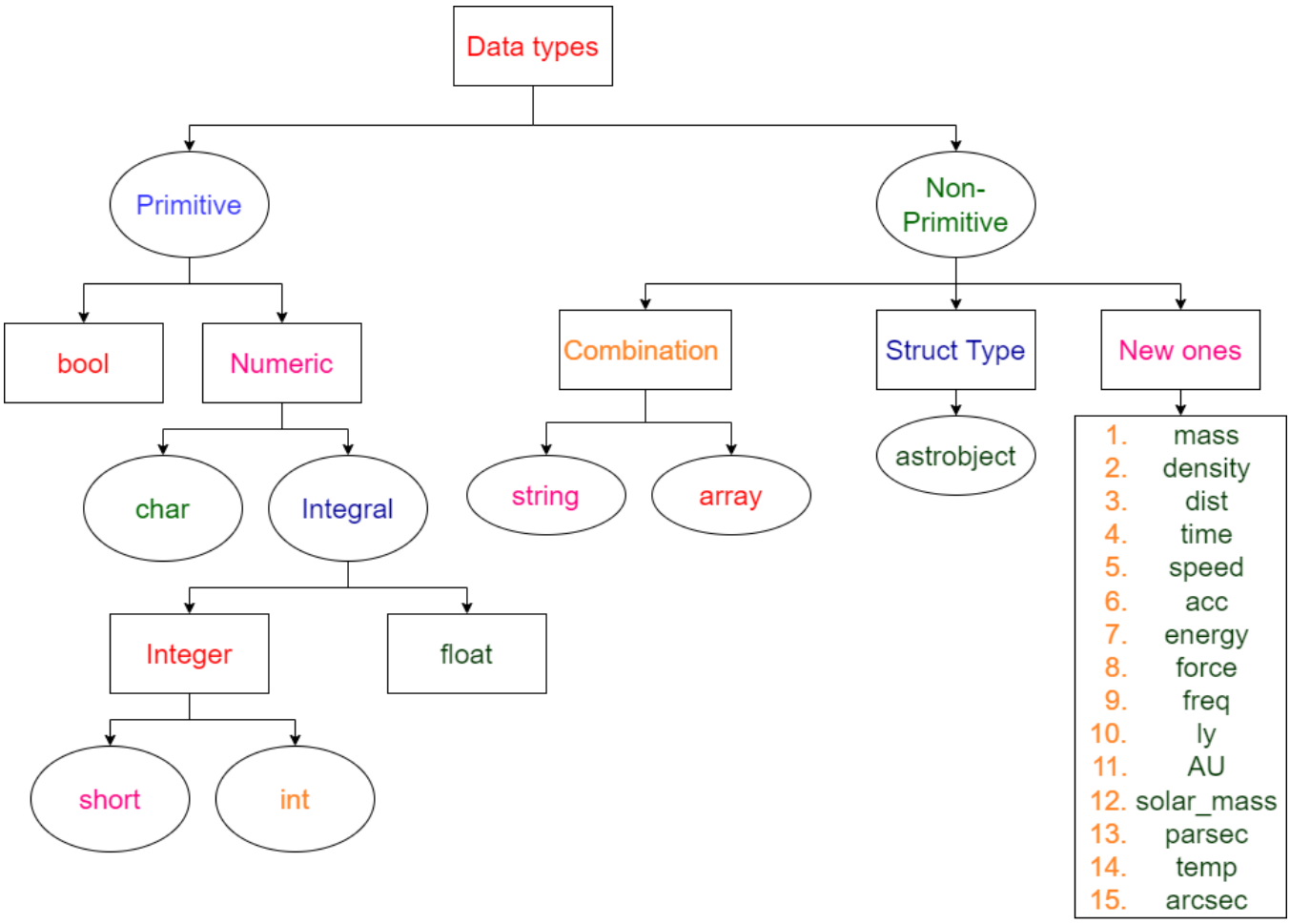


Figure 1: Flowchart of data types

## 2.7 Syntax:-

Examples of syntax are

1. syntax for repeat
   repeat statement is used for looping.
   Syntax:
   repeat( initialize; condition; update){
   #statements for execution#
   }

2. syntax for if-else
   If-else is used for creating conditional statements.
   Syntax:
   if( expression )
   {
   #statements for execution if expression is true#
   }
   else
   {
   #statements for execution if expression is false#
   }

3. syntax of defining function
   keyword "proc" is used before declaring the function.
   Syntax:-
   Proc data_type func_name(data_type1 arg1,data_type arg2....)
   {
   # Function body #
   }

# 3 Built-in Features Of Cosmos:-

## 3.1 Gravitational formulae:-

| **Name of Law or formulae** | **Law or formula** & how to call in the language |
|---|---|
| Force between 2 masses | $F = \frac{GMm}{R^2}$ |
| Kepler's Law 3 | $T^2 = \frac{4\pi^2}{GM} r^3$ |
| Acceleration due to gravity | $a = \frac{Gm}{r^2}$ |
| Gravity below the surface of a homogeneous planet | $\frac{GMr}{R^3}$ |
| Gravity above the surface of a planet | $\frac{GM}{r^2}$ r is the distance from the center of the planet |
| Gravitational Potential Energy | $\frac{GMm}{r}$ |
| Escape velocity | $\sqrt{2GM/r}$ or $\sqrt{2gR}$ |
| Orbital velocity | $\sqrt{\frac{Gm}{r}}$ |
| Time period for satellite | $\frac{2\pi}{(GM)^{1/2}} a^{3/2}$ |
| Binding Energy | GMm/2r |
| Satellite orbit height | $(GmT^2/4\pi^3)^{3/2}$ |

Table 5: Laws and formulae

Note:- here r is the distance between centres of two bodies

1. Gravitational Force between 2 masses :$F = \frac{GMm}{r^2}$
   This function accepts mass of body 1, mass of body 2 and the distance between them and returns the force of gravitation in Newtons.

   Ex. Force of earth's gravitation on a body of mass 1kg at the surface of earth. Earth 's mass= Me, Mass of body =m, Radius of earth =Re
   Calling the function
   gravitational_force(Me, m, Re );

2. Kepler's Law 3 : $T^2 = \frac{4\pi^2}{GM} r^3$
   This function is used to calculate the time period of one satellite given the time period of another satellite and orbital radius of both the satellites.

   Ex. Titan, the largest moon of Saturn, has a mean orbital radius of $1.22 \times 109m$. The orbital period of Titan is 15.95 days. Hyperion, another moon of Saturn, orbits at a mean radius of $1.48 \times 109m$. Use Kepler's third law of planetary motion to predict the orbital period of Hyperion in days. $rT = 1.22 \times 109m$, $TT = 15.95 days rH = 1.48 \times 109m$ . And we need to find out Th.

   Function call: keplers_law3(rT, rH, TT)

3. Acc due to gravity : $\frac{Gm}{r^2}$ This function returns the acceleration due to gravity at a distance r from the center of the planet.

Ex. Calculate acceleration due to gravity at the surface of a planet with mass M, radius R.
Function call: calculate_g(M, R)

4. Gravity below the homogeneous surface: This function returns acceleration due to gravity at a distance r from the center of the planet( $< R$). M is the mass of planet

   Function call: g_below(M,R,r);

5. Gravitational Potential Energy : $\frac{GMm}{r}$
   This function returns the gravitational potential energy of an object of mass m and a distance r from the center of planet of mass M

   Function call: calculate_gpe(M,m, r);

6. Escape Velocity $(2GM/R)^{1/2} or (2gR)^{1/2}$ : This function accepts two arguments. It has 2 declarations:

   (a) The first prototype accepts an astronomical object (obj) and returns escape velocity of the object wrt that astronomical object.
   Function call: escape_velocity_obj (Obj )

   (b) The second one accepts radius R of the planet and the acceleration due to gravity (g) and returns the escape velocity of the object wrt that planet.
   Function call: escape_velocity(g,R)

7. Orbital velocity :$(2Gm/r)^{1/2}$ This function accepts an astronomical object and the orbital distance and returns the required orbital velocity of a satellite.

   Function call: orbital_velocity( Planet, r)

8. Time period for satellite : $(2\pi/(GM)^{1/2})r^{3/2}$ This function accepts an astronomical object and the orbital distance and returns the time period of a satellite (in days).

   Function call : time_period(Planet, r)

9. Satellite orbit height : $(GmT^2/4\pi^3)^{3/2}$ This function accepts an astronomical object and the time period (in days) of the satellite and returns the orbital height of a satellite from the center of the astronomical object.

   Function call : orbital_height(Planet, T)

10. Binding Energy : GMm/2r This function accepts an astronomical object, mass of the body, and distance from the center of the astronomical object (r )and returns the binding energy of the body.

    Function call : binding_energy(planet, r )

### 3.2 Black-Body Radiation formulae:-

1. Stefan's Law
   This function takes the temperature of the blackbody as an argument and returns its Radiant flux.
   $$J^* = \sigma T^4$$
   Function Call: radiant_flux(T)

2. Wien's Displacement Law
   This function takes the temperature of the blackbody as an argument and returns the wavelength for which the blackbody-radiance curve peaks (maximum spectral energy density).
   $$\lambda_{max} = b/T$$
   Function Call: peak_wavelength(T)

3. Rayleigh-Jeans Law
   This function takes wavelength $\lambda$ and temperature T of the blackbody as arguments and returns its spectral radiance.
   $$B_\lambda(T) = \frac{2ck_BT}{\lambda^4}$$
   Function Call: spectral_radiance($\lambda$,T)

### 3.3 Relativity Formulas

1. Lorentz factor
   Lorentz factor is widely used in special relativity. It is used to calculate relative mass, relative energy in an inertial frame. This function takes the speed (v) of an inertial frame as argument and computes its lorentz factor according to the equation below.
   $$\gamma = \frac{1}{\sqrt{1-\frac{v^2}{c^2}}}$$
   Function Call: lorentz_factor(v)

2. Energy-Mass Equivalence
   This function takes the rest mass m of body as argument and returns its rest energy.
   $$E = mc^2$$
   Function call: emc(m)

### 3.4 Light Formulas

1. Energy of lightwave
   This function returns the energy of electromagnetic lightwave. It has two signatures:

   - The first takes frequency $\nu$ as an argument.
     Function Call: efreq($\nu$)

   - The other takes wavelength $\lambda$ as argument.
     Function Call: elambda($\lambda$)

2. Redshift
   This function takes the recessional speed(v) of a galaxy as an argument and returns its red-

shift. Our function first compares recessional speed to speed of light and operates accordingly.
Function Call: redshift(v)

3. Parallax
This function takes the parallax angle p of a star in parsec and returns its distance from earth.
$$d = 1/p$$
Function Call: parallax(p)

### 3.5 Star Magnitude Formulas

1. Bolometric/Absolute Magnitude This function takes distance d of the star and its apparent magnitude m as arguments and returns its bolometric/absolute magnitude.
$m - 2.5 \times log[(d/10)^2]$ Function Call: abs_mag(m,d)

2. Star Luminosity
This function takes temperature T and radius r of the star as arguments and returns its luminosity (Power of star).
$$L = 4\pi r^2 \sigma T^4$$
Function Call: luminosity(r,T)

3. Distance Determination
This function takes absolute and apparent magnitude as arguments and returns its distance.
$$D = (10pc) \times 10^{(m-Mv)/5}$$
Function Call: star_dist(m,Mv)

4. Apparent Magnitude
This function takes the brightness ratio r of two stars as argument and returns the absolute difference of their magnitudes.
The star having greater brightness will have less magnitude.
$$m2 - m1 = -2.5 \times log(B2/B1)$$
Function Call: apparent_mag(r)

### 3.6 Binary Star Formulas

This library will have functions which take binary stars s1, s2 (both having astronomical object data type) as inputs and could compute related parameters. Example

- COM orbit
- Time Period of system

### 3.7 Other Formulas

1. Hubble's Law
This function takes the proper distance(d) of a galaxy as argument and returns its recessional speed(v).
$$v = H_0 D$$
Function Call: hubble(d)

2. Schwarzschild Radius
   This function takes blackhole b as argument and returns its Schwarzschild radius.
   $$R_s = 2GM/c2$$
   Function Call: schwarzschild(b)

### 3.8 Conversion

This library includes all conversion functions between 2 datatypes of the same kind. Example parsec,AU,ly,dist are all distances so there will be functions to convert each unit into another.
Example: convert_ly_to_AU(ly) converts the given light year distance into astronomical units(AU) and returns the answer.
To convert x times unit 1 into unit 2:
Function Call: convert_unit1_to_unit2(x)

### 3.9 man():

This is a utility function specifically designed for informative purposes. It accepts a string containing the keywords related to which the information is required. It also provides a brief description of various libraries provided inside the language. Example

Function call: man("Solar System") will print the information related to solar system on the output stream.

## 4 Grammar :

```
%token IDENTIFIER CONSTANT STRING_LITERAL
%token AUTO
%token INC_OP DEC_OP LE_OP GE_OP EQ_OP NE_OP
%token CHAR SHORT INT FLOAT VOID BOOL STRING
%token STRUCT ELLIPSIS
%token IF ELSE REPEAT CONTINUE BREAK RETURN
%token DIST LY AU MASS SOLAR_MASS DENSITY DIST TIME SPEED ACC ENERGY
        FORCE FREQ PARSEC TEMP ARCSEC
%token ASTROBJECT
%token PROC
%start translation_unit
    primary_expression
      : IDENTIFIER
       |CONSTANT
       |STRING_LITERAL
       |'(' expression ')'
;
postfix_expression
      : primary_expression
       |postfix_expression '[' expression ']'
       |postfix_expression '(' ')'
       |postfix_expression '(' argument_expression_list ')'
       |postfix_expression '.' IDENTIFIER
```

```
        |postfix_expression INC_OP
        |postfix_expression DEC_OP
;
argument_expression_list
        : assignment_expression
        |argument_expression_list ',' assignment_expression
;
unary_expression
        : postfix_expression
        |INC_OP unary_expression
        |DEC_OP unary_expression
        |unary_operator cast_expression
;
unary_operator
        : '+'
        |'-'
;
cast_expression
        : unary_expression
        |'(' type_name ')' cast_expression
;
multiplicative_expression
        : cast_expression
        |multiplicative_expression '*' cast_expression
        |multiplicative_expression '/' cast_expression
        |multiplicative_expression '%' cast_expression
;
additive_expression
        : multiplicative_expression
        |additive_expression '+' multiplicative_expression
        |additive_expression '-' multiplicative_expression
;
relational_expression
        : additive_expression
        |relational_expression '¡' additive_expression
        |relational_expression '¿' additive_expression
        |relational_expression LE_OP additive_expression
        |relational_expression GE_OP additive_expression
;
equality_expression
        : relational_expressionv        |equality_expression EQ_OP relational_expression
        |equality_expression NE_OP relational_expression
;
logical_and_expression
        : equality_expression
        |logical_and_expression AND_OP equality_expression
;
logical_or_expression
```

```
        : logical_and_expression
        |logical_or_expression OR_OP logical_and_expression
;
assignment_expression
        : logical_or_expression
        |unary_expression assignment_operator assignment_expression
;
assignment_operator
        : '='
;
expression
        : assignment_expression
        |expression ',' assignment_expression
;
constant_expression
        : logical_or_expression
;
declaration
        : declaration_specifiers ';'
        |declaration_specifiers init_declarator_list ';'
;
declaration_specifiers
        : storage_class_specifier
        |storage_class_specifier declaration_specifiers
        |type_specifier
        |type_specifier declaration_specifiers
        |type_qualifier
        |type_qualifier declaration_specifiers
;
init_declarator_list
        : init_declarator
        |init_declarator_list ',' init_declarator
;
init_declarator
        : declarator
        |declarator '=' initializer
;
storage_class_specifier
        : AUTO
;
type_specifier
        : VOID
        |CHAR
        |SHORT
        |INT
        |FLOAT
        |struct_specifier
        |TYPE_NAME
```

```
        |DIST
        |LY
        |AU
        |MASS
        |SOLAR_MASS
        |DENSITY
        |DIST
        |TIME
        |SPEED
        |ACC
        |ENERGY
        |FORCE
        |FREQ
        |PARSEC
        |TEMP
        |ARCSEC
;
struct_specifier
        : struct_or_union IDENTIFIER '' struct_declaration_list ''
        |struct_or_union '' struct_declaration_list ''
        |struct_or_union IDENTIFIER
;
struct_declaration_list
        : struct_declaration
        |struct_declaration_list struct_declaration
;
struct_declaration
        : specifier_qualifier_list struct_declarator_list ';'
;
specifier_qualifier_list
        : type_specifier specifier_qualifier_list
        |type_specifier
        |type_qualifier specifier_qualifier_list
        |type_qualifier
;
struct_declarator_list
        : struct_declarator
        |struct_declarator_list ',' struct_declarator
;
struct_declarator
        : declarator
        |':' constant_expression
        |declarator ':' constant_expression
;
type_qualifier
        : CONST
;
declarator
```

```
        : pointer direct_declarator
;
direct_declarator
        : IDENTIFIER
        |'(' declarator ')'
        |direct_declarator '[' constant_expression ']'
        |direct_declarator '[' ']'
        |direct_declarator '(' parameter_type_list ')'
        |direct_declarator '(' identifier_list ')'
        |direct_declarator '(' ')'
;
type_qualifier_list
        : type_qualifier
        |type_qualifier_list type_qualifier
;
parameter_type_list
        : parameter_list
        |parameter_list ',' ELLIPSIS
;
parameter_list
        : parameter_declaration
        |parameter_list ',' parameter_declaration
;
parameter_declaration
        : declaration_specifiers declarator
        |declaration_specifiers abstract_declarator
        |declaration_specifiers
;
identifier_list
        : IDENTIFIER
        |identifier_list ',' IDENTIFIER
;
type_name
        : specifier_qualifier_list
        |specifier_qualifier_list abstract_declarator
;
abstract_declarator
        : direct_abstract_declarator
;
direct_abstract_declarator
        : '(' abstract_declarator ')'
        |'[' ']'
        |'[' constant_expression ']'
        |direct_abstract_declarator '[' ']'
        |direct_abstract_declarator '[' constant_expression ']'
        |'(' ')'
        |'(' parameter_type_list ')'
        |direct_abstract_declarator '(' ')'
```

```
        |direct_abstract_declarator '(' parameter_type_list ')'
;
initializer
        : assignment_expression
        |'' initializer_list ''
        |'' initializer_list ',' ''
;
initializer_list
        : initializer
        |initializer_list ',' initializer
;
statement
        : compound_statement
        |expression_statement
        |selection_statement
        |iteration_statement
        |jump_statement
;
compound_statement
        : '' ''
        |'' statement_list ''
        |'' declaration_list ''
        |'' declaration_list statement_list ''
;
declaration_list
        : declaration
        |declaration_list declaration
;
statement_list
        : statement
        |statement_list statement
;
expression_statement
        : ';'
        |expression ';'
;
selection_statement
        : IF '(' expression ')' statement
        |IF '(' expression ')' statement ELSE statement
;
iteration_statement
        : REPEAT '(' expression_statement expression_statement ')' statement
        |REPEAT'(' expression_statement expression_statement expression ')' statement
;
jump_statement
        : CONTINUE ';'
        |BREAK ';'
        |RETURN ';'
```

```
        |RETURN expression ';'
;
translation_unit
        : external_declaration
        |translation_unit external_declaration
;
external_declaration
        : function_definition
        |declaration
;
function_definition
        : declaration_specifiers declarator declaration_list compound_statement
        |declaration_specifiers declarator compound_statement
        |declarator declaration_list compound_statement
        |declarator compound_statement
;
```

**Note:-** We have used ANSI YACC as reference for the grammar.

## 5 Example

Q. The apparent and absolute magnitude of a star from earth are 18 and -4!
Find its parallax angle observed from earth?

Ans. This could be solved very simply using COSMOS.

```
# Preprocessor directives and libraries must be included before main procedure#
proc int main()
{
        int m = 18, Mv = -4;
        # We can calculate distance of star from earth using star_dist() procedure #
        # Declaring a variable of type parsec to store distance #
        parsec d = star_dist(m,Mv);
        # Now we can calculate parallax using parallax() procedure #
        arc_sec ans;
        ans = parallax(d);
        output(ans);
}
```