

Updates in parser code:

```
t2parser.y
1  %{\
2      /*definitions*/
3      #include <stdio.h>
4      #include<string.h>
5      #include<stdlib.h>
6      #include<ctype.h>
7      #include <limits.h>
8      // #include"lex.yy.c" // this is creating multiple definitions
9
10     // Declaration of tree
11     struct node {
12         int num_children;        // Number of children
13         struct node **children; // Array of pointers to child nodes
14         char *token;            // Token associated with the node
15     };
16
17     struct node *head;
18     struct node* mknnode(int num_children, struct node **children, char *token) ;
19     void printtree(struct node* tree);
20     void printInorder(struct node *tree);
21     void add(char);
22     void insert_type();
23     int search(char *);
24
25     void check_declaration(char *);
26     void check_return_type(char *);
27     int check_types(char *, char *);
28     char *get_type(char *);
29
30     struct dataType {
31         char * id_name;
32         char * data_type;
33         char * type;
34         int line_no;
35         int thisscope;
36         int num_params;
37     } symbol_table[40];
38
39     int count=0;
40     int q;
41     char type[10];
42     extern int countn;
43     extern int scope;
44     int curr_num_params=0;
45     int curr_num_args=0;
```

```

99 telugu_identifier:
100     TELUGU_IDENTIFIER {printf("CHECKING FOR %s\n",$1.name);check_declaration($1.name); printf("saw pure id2");$$nd = mknode(NULL, NULL, $1.name);}
101
102 telugu_function_name: // only for functions being declared
103     TELUGU_IDENTIFIER {printf("parser saw teluguFuncName");
104     $$nd = mknode(NULL, NULL, $1.name);}
105
106 telugu_function_name_call: // only for functions being called
107     TELUGU_IDENTIFIER {printf("parser saw teluguFuncNameCall");
108     $$nd = mknode(NULL, NULL, $1.name);}
109
110 telugu_identifier_declaring: // only for identifiers being declared
111     TELUGU_IDENTIFIER { printf("saw varDeclareid");add('V');$$nd = mknode(NULL, NULL, $1.name);}
112
113
114 telugu_imported_library: // only for identifiers being declared
115     TELUGU_IDENTIFIER { add('L');$$nd = mknode(NULL, NULL, $1.name);}
116
117 telugu_print:
118     TELUGU_PRINT {add('K');$$nd = mknode(NULL, NULL, $1.name);}
119
120 telugu_int:
121     TELUGU_INT {$$.nd = mknode(NULL, NULL, $1.name);add('i');}
122
123 telugu_input: // cin>> , scanf()
124     TELUGU_INPUT {add('K');$$nd = mknode(NULL, NULL, $1.name);}
125
126 telugu_float:
127     TELUGU_FLOAT {add('f');$$nd = mknode(NULL, NULL, $1.name);}
128
129 telugu_import:
130     TELUGU_IMPORT {$$.nd = mknode(NULL, NULL, $1.name);}
131
132 telugu_constant:
133     TELUGU_INT {$$.nd = mknode(NULL, NULL, $1.name);add('i');}
134     | TELUGU_FLOAT {$$.nd = mknode(NULL, NULL, $1.name);add('f');}
135     | TELUGU_STRING {$$.nd = mknode(NULL, NULL, $1.name);add('s');}
136     | TELUGU_CHARACTER {$$.nd = mknode(NULL, NULL, $1.name);add('c');}
137
138
139
140
141
142
143
144
145
146
147 ✓ telugu_datatype:
148     TELUGU_DATATYPE {insert_type();$$nd = mknode(NULL, NULL, $1.name);}
149
150
151
152
153
154
155
156
157
158
159
160
161
162
163
164
165
166
167
168
169
170
171
172
173
174
175
176
177
178
179
180
181
182
183
184
185
186
187
188
189
190
191
192
193
194
195
196
197
198
199
200
201
202
203
204
205
206
207
208
209
210
211
212
213
214
215
216
217
218
219
220
221
222
223
224
225
226
227
228
229
230
231
232
233
234
235
236
237
238
239
240
241
242
243
244
245
246
247
248
249
250
251
252
253
254
255
256
257
258
259
260
261
262
263
264
265
266
267
268
269
270
271
272
273
274
275
276
277
278
279
280
281
282
283
284
285
286
287
288
289
290
291
292
293
294
295
296
297
298
299
300
301
302
303
304
305
306
307
308
309
310
311
312
313
314
315
316
317
318
319
320
321
322
323
324
325
326
327
328
329
330
331
332
333
334
335
336
337
338
339
340
341
342
343
344
345
346
347
348
349
350
351
352
353
354
355
356
357
358
359
360
361
362
363
364
365
366
367
368
369
370
371
372
373
374
375
376
377
378
379
380
381
382
383
384
385
386
387
388
389
390
391
392
393
394
395
396
397
398
399
400
401
402
403
404
405
406
407
408
409
410
411
412
413
414
415
416
417
418
419
420
421
422
423
424
425
426
427
428
429
430
431
432
433
434
435
436
437
438
439
440
441
442
443
444
445
446
447
448
449
450
451
452
453
454
455
456
457
458
459
460
461
462
463
464
465
466
467
468
469
470
471
472
473
474
475
476
477
478
479
480
481
482
483
484
485
486
487
488
489
490
491
492
493
494
495
496
497
498
499
500
501
502
503
504
505
506
507
508
509
510
511
512
513
514
515
516
517
518
519
520
521
522
523
524
525
526
527
528
529
530
531
532
533
534
535
536
537
538
539
540
541
542
543
544
545
546
547
548
549
550
551
552
553
554
555
556
557
558
559
560
561
562
563
564
565
566
567
568
569
570
571
572
573
574
575
576
577
578
579
580
581
582
583
584
585
586
587
588
589
590
591
592
593
594
595
596
597
598
599
600
601
602
603
604
605
606
607
608
609
610
611
612
613
614
615
616
617
618
619
620
621
622
623
624
625
626
627
628
629
630
631
632
633
634
635
636
637
638
639
640
641
642
643
644
645
646
647
648
649
650
651
652
653
654
655
656
657
658
659
660
661
662
663
664
665
666
667
668
669
670
671
672
673
674
675
676
677
678
679
680
681
682
683
684
685
686
687
688
689
690
691
692
693
694
695
696
697
698
699
700
701
702
703
704
705
706
707
708
709
710
711
712
713
714
715
716
717
718
719
720
721
722
723
724
725
726
727
728
729
730
731
732
733
734
735
736
737
738
739
740
741
742
743
744
745
746
747
748
749
750
751
752
753
754
755
756
757
758
759
760
761
762
763
764
765
766
767
768
769
770
771
772
773
774
775
776
777
778
779
780
781
782
783
784
785
786
787
788
789
790
791
792
793
794
795
796
797
798
799
800
801
802
803
804
805
806
807
808
809
810
811
812
813
814
815
816
817
818
819
820
821
822
823
824
825
826
827
828
829
830
831
832
833
834
835
836
837
838
839
840
841
842
843
844
845
846
847
848
849
850
851
852
853
854
855
856
857
858
859
860
861
862
863
864
865
866
867
868
869
870
871
872
873
874
875
876
877
878
879
880
881
882
883
884
885
886
887
888
889
890
891
892
893
894
895
896
897
898
899
900
901
902
903
904
905
906
907
908
909
910
911
912
913
914
915
916
917
918
919
920
921
922
923
924
925
926
927
928
929
930
931
932
933
934
935
936
937
938
939
940
941
942
943
944
945
946
947
948
949
950
951
952
953
954
955
956
957
958
959
960
961
962
963
964
965
966
967
968
969
970
971
972
973
974
975
976
977
978
979
980
981
982
983
984
985
986
987
988
989
990
991
992
993
994
995
996
997
998
999

```

```

180 telugu_closed_floor_bracket:
181     TELUGU_CLOSED_FLOOR_BRACKET {$$.nd = mknode(NULL, NULL, $1.name);
182     //here we need to remove all the variables declared in this scope
183     // change all of their scope to INT_MAX
184     int i;
185     for(i=count-1; i>=0; i--) {
186         if(symbol_table[i].thisscope == scope &&
187            strcmp(symbol_table[i].type, "Variable")==0) {
188             symbol_table[i].thisscope = INT_MAX;
189             printf("\nERASING %s from symbol table as its CURRENT SCOPE is FINISHED\n", symbol_table[i].id_name);
190         }
191     }
192     scope--;
193 } // decrease scope for variables
194
195 telugu_punctuation_comma:
196     TELUGU_PUNCTUATION_COMMA {$$.nd = mknode(NULL, NULL, $1.name);}
197
198 telugu_newline:
199     TELUGU_NEWLINE {$$.nd = mknode(NULL, NULL, $1.name);}
200
201 telugu_finish:
202     TELUGU_FINISH {$$.nd = mknode(NULL, NULL, $1.name);
203     //strcpy(exp_type,exp_type_empty); //this is not working
204     } // resetting exp_type string
205
206 telugu_function:
207     TELUGU_FUNCTION {add('K');$.nd = mknode(NULL, NULL, $1.name);}
208
209 telugu_return:
210     TELUGU_RETURN {add('K');$.nd = mknode(NULL, NULL, $1.name);}
211
212 telugu_character:
213     TELUGU_CHARACTER {add('c');$.nd = mknode(NULL, NULL, $1.name);}
214
215 ---
284 exp: // empty not allowed
285
286 telugu_int {
287     if(strcmp(exp_type, "")==0) {
288         strcpy(exp_type, "sankhya");
289     }
290     else if(strcmp(exp_type, "theega")==0) {
291         sprintf(errors[sem_errors], "Line %d: operation among int and string in expression not allowed\n", countn+1);
292         sem_errors++;
293     }
294     printf("Parser found int\n");
295     int num_children = 1; // Number of children
296     struct node **children = (struct node **)malloc(num_children * sizeof(struct node *));
297     children[0] = $1.nd;
298     $$nd = mknode(num_children, children, "INT");
299 }
300 telugu_float {
301     if(strcmp(exp_type, "")==0) {
302         strcpy(exp_type, "thelu");
303     }
304     else if(strcmp(exp_type, "theega")==0) {
305         sprintf(errors[sem_errors], "Line %d: operation among float and string in expression not allowed\n", countn+1);
306         sem_errors++;
307     }
308     printf("Parser found float\n");
309     int num_children = 1; // Number of children
310     struct node **children = (struct node **)malloc(num_children * sizeof(struct node *));
311     children[0] = $1.nd;
312     $$nd = mknode(num_children, children, "FLOAT");
313 }
314 telugu_character {
315     printf("Parser found character\n");
316     int num_children = 1; // Number of children
317     struct node **children = (struct node **)malloc(num_children * sizeof(struct node *));
318     children[0] = $1.nd;
319     $$nd = mknode(num_children, children, "CHAR");
320 }
321 telugu_string {
322     if(strcmp(exp_type, "")==0) {
323         strcpy(exp_type, "theega");
324     }
325     else if(strcmp(exp_type, "sankhya")==0 || strcmp(exp_type, "thelu")==0) {
326         sprintf(errors[sem_errors], "Line %d: operation among string and int/float in expression not allowed\n", countn+1);

```

```

327         sem_errors++;
328     }
329     printf("Parser found string\n");
330     int num_children = 1; // Number of children
331     struct node **children = (struct node **)malloc(num_children * sizeof(struct node *));
332     children[0] = $1.nd;
333     $$nd = mknode(num_children, children, "STRING");
334 }

```

Updates in lexical analyser code:

```

12     extern int countn; // replaced new_line count
13     extern int scope=1; // starts from 1

```

```

642 variable_declaration:
643     telugu_datatype telugu_identifer_declaring telugu_assignment_operator exp {
644         //add('V'); // this is taking ';' as a variable
645         printf("Parser found datatypeId=exp\n");
646         int num_children = 4; // Number of children
647         struct node **children = (struct node **)malloc(num_children * sizeof(struct node *));
648         children[0] = $1.nd;
649         children[1] = $2.nd;
650         children[2] = $3.nd;
651         children[3] = $4.nd;
652         $$nd = mknode(num_children, children, "datatypeId=exp");
653         if(strcmp($1.name, exp_type)){
654             sprintf(errors[sem_errors], "Line %d: Data type casting not allowed in declaration\n", countn);
655             sem_errors++;
656         }
657     }

681 parameters_repeat: // can be empty 0 or more occurences
682     { $$nd = mknode(NULL, NULL, "empty"); }
683 | parameters_repeat telugu_datatype telugu_identifer_declaring telugu_punctuation_comma {
684     printf("Parser found paramRepDatatypeIdComma\n");
685     curr_num_params++;
686     int num_children = 4; // Number of children
687     struct node **children = (struct node **)malloc(num_children * sizeof(struct node *));
688     children[0] = $1.nd;
689     children[1] = $2.nd;
690     children[2] = $3.nd;
691     children[3] = $4.nd;
692     $$nd = mknode(num_children, children, "paramRepDatatypeIdComma");
693 }

694
695 parameters_line: // can be empty
696     { $$nd = mknode(NULL, NULL, "empty"); }
697 | parameters_repeat telugu_datatype telugu_identifer_declaring {
698     printf("Parser found parameters_line\n");
699     curr_num_params++;
700     int num_children = 3; // Number of children
701     struct node **children = (struct node **)malloc(num_children * sizeof(struct node *));
702     children[0] = $1.nd;
703     children[1] = $2.nd;
704     children[2] = $3.nd;
705     $$nd = mknode(num_children, children, "paramLine");
706 }
707

708 identifiers_repeat: // abc,x,y,p can be empty
709     { $$nd = mknode(NULL, NULL, "empty"); }
710 | telugu_identifer {
711     curr_num_args++;
712     printf("Parser found lastparam\n");
713     int num_children = 1; // Number of children
714     struct node **children = (struct node **)malloc(num_children * sizeof(struct node *));
715     children[0] = $1.nd;
716     $$nd = mknode(num_children, children, "paramEnd");
717 }
718 | telugu_constant {
719     curr_num_args++;
720     printf("Parser found lastparam\n");
721     int num_children = 1; // Number of children
722     struct node **children = (struct node **)malloc(num_children * sizeof(struct node *));
723     children[0] = $1.nd;
724     $$nd = mknode(num_children, children, "paramEnd");
725 }

```

```

726 | telugu_identifier telugu_punctuation_comma identifiers_repeat {
727 |     curr_num_args++;
728 |     printf("Parser found id-comma-prep\n");
729 |     int num_children = 3; // Number of children
730 |     struct node **children = (struct node **)malloc(num_children * sizeof(struct node *));
731 |     children[0] = $1.nd;
732 |     children[1] = $2.nd;
733 |     children[2] = $3.nd;
734 |     $$nd = mknode(num_children, children, "paramRep");
735 | }
736 | telugu_constant telugu_punctuation_comma identifiers_repeat {
737 |     curr_num_args++;
738 |     printf("Parser found const-comma-prep\n");
739 |     int num_children = 3; // Number of children
740 |     struct node **children = (struct node **)malloc(num_children * sizeof(struct node *));
741 |     children[0] = $1.nd;
742 |     children[1] = $2.nd;
743 |     children[2] = $3.nd;
744 |     $$nd = mknode(num_children, children, "paramRep");
745 | }
746 |
756 ~ equation:
757 ~ telugu_identifier telugu_assignment_operator { strcpy(exp_type, " "); } exp {
758 |     printf("Parser found equation\n");
759 |     int num_children = 3; // Number of children
760 |     struct node **children = (struct node **)malloc(num_children * sizeof(struct node *));
761 |     children[0] = $1.nd;
762 |     children[1] = $2.nd;
763 |     children[2] = $4.nd;
764 |     $$nd = mknode(num_children, children, "id=exp");
765 |     //check if identifier type and exp_type mismatch -> if yes then typecast is happening
766 |     if(strcmp(get_type($1.name), exp_type)){
767 |         sprintf(errors[sem_errors], "Line %d: Data type casting not allowed in equation\n", countn);
768 |         sem_errors++;
769 |     }
770 | }

```

```

887 function_declaration:
888     telugu_function telugu_function_name { add('F'); } telugu_open_curly_bracket parameters_line telugu_closed_curly_bracket telugu_open_floor_bracket function_content telugu_closed_floor_bracket {
889 |     printf("Parser found equation\n");
890 |     int num_children = 8; // Number of childrenfunction_call
891 |     struct node **children = (struct node **)malloc(num_children * sizeof(struct node *));
892 |     children[0] = $1.nd;
893 |     children[1] = $2.nd;
894 |     children[2] = $4.nd;
895 |     children[3] = $5.nd;
896 |     children[4] = $6.nd;
897 |     children[5] = $7.nd;
898 |     children[6] = $8.nd;
899 |     children[7] = $9.nd;
900 |     $$nd = mknode(num_children, children, "func-id-(param){content}");
901 |     symbol_table[count-curr_num_params-1].num_params= curr_num_params;
902 |     curr_num_params=0;
903 | }
904 |
905 function_call:
906     telugu_identifier { check_declaration($1.name); } telugu_open_curly_bracket identifiers_line telugu_closed_curly_bracket {
907 |     printf("Parser found id(idLine)Finish\n");
908 |     int num_children = 4; // Number of children
909 |     struct node **children = (struct node **)malloc(num_children * sizeof(struct node *));
910 |     children[0] = $1.nd;
911 |     children[1] = $3.nd;
912 |     children[2] = $4.nd;
913 |     children[3] = $5.nd;
914 |     $$nd = mknode(num_children, children, "id(idLine)Finish");
915 |
916 |     for(int i=0;i<count;i++){
917 |         if(strcmp(symbol_table[i].id_name,$1.name)==0){ // found the corresponding function
918 |             if(symbol_table[i].num_params!=curr_num_args){
919 |                 printf("ERROR: Number of parameters do not match\n");
920 |                 sprintf(errors[sem_errors], "Line %d: need %d arguments but found %d args\n", countn+1,symbol_table[i].num_params,curr_num_args);
921 |                 sem_errors++;
922 |                 break;
923 |             }
924 |         }
925 |     }
926 |
927 |     curr_num_args=0;
928 | }

```

```

990  //////////////////////////////////////// SYMBOL TABLE & SEMANTIC ANALYSIS PART
991
992  int search(char *type) {
993      int i;
994      for(i=count-1; i>=0; i--) {
995          if(strcmp(symbol_table[i].id_name, type)==0) {
996              return symbol_table[i].thisscope;
997              break;
998          }
999      }
1000      return 0;
1001  }
1002
1003  void check_declaration(char *c) {
1004      q = search(c);
1005      if(!q) {
1006          sprintf(errors[sem_errors], "Line %d: Variable \"%s\" not declared before usage!\n", countn+1, c);
1007          sem_errors++;
1008      }
1009  }
1010
1011  char *get_type(char *var){
1012      for(int i=0; i<count; i++) {
1013          // Handle case of use before declaration
1014          if(!strcmp(symbol_table[i].id_name, var)) {
1015              return symbol_table[i].type;
1016          }
1017      }
1018  }
1019
1020  void add(char c) {
1021      if(c == 'V'){ // variable
1022          for(int i=0; i<reserved_count; i++){
1023              if(!strcmp(reserved[i], strdup(yy_text))){
1024                  sprintf(errors[sem_errors], "Line %d: Variable name \"%s\" is a reserved keyword!\n", countn+1, yy_text);
1025                  sem_errors++;
1026                  return;
1027              }
1028          }
1029      }

```

```

1030 q=search(yy_text);
1031 if(!q) { // insert into symbol table only if not already present
1032     if(c == 'H') { //header
1033         symbol_table[count].id_name=strdup(yy_text);
1034         symbol_table[count].data_type=strdup(type);
1035         symbol_table[count].line_no=countn;
1036         symbol_table[count].type=strdup("Header");
1037         symbol_table[count].thisscope=scope;
1038         symbol_table[count].num_params=0;
1039         count++;
1040     }
1041     else if(c == 'K') { //keyword
1042         symbol_table[count].id_name=strdup(yy_text);
1043         symbol_table[count].data_type=strdup("N/A");
1044         symbol_table[count].line_no=countn;
1045         symbol_table[count].type=strdup("Keyword\t");
1046         symbol_table[count].thisscope=scope;
1047         symbol_table[count].num_params=0;
1048         count++;
1049     }
1050     else if(c == 'V') { //variable
1051         printf("yytext: %s\n", yy_text);
1052         symbol_table[count].id_name=strdup(yy_text);
1053         symbol_table[count].data_type=strdup(type);
1054         symbol_table[count].line_no=countn;
1055         symbol_table[count].type=strdup("Variable");
1056         symbol_table[count].thisscope=scope;
1057         symbol_table[count].num_params=0;
1058         count++;
1059     }
1060     else if(c == 'C') { //constant sankhya
1061         symbol_table[count].id_name=strdup(yy_text);
1062         symbol_table[count].data_type=strdup("CONST");
1063         symbol_table[count].line_no=countn;
1064         symbol_table[count].type=strdup("constantx");
1065         symbol_table[count].thisscope=scope;
1066         symbol_table[count].num_params=0;
1067         count++;
1068     }
1069     else if(c == 'i') { //constant sankhya
1070         symbol_table[count].id_name=strdup(yy_text);
1071         symbol_table[count].data_type=strdup("CONST");
1072         symbol_table[count].line_no=countn;
1073         symbol_table[count].type=strdup("sankhya");
1074         symbol_table[count].thisscope=scope;

```



```

1075         symbol_table[count].num_params=0;
1076         count++;
1077     }
1078     else if(c == 'f') { //constant float thelu
1079         symbol_table[count].id_name=strdup(yy_text);
1080         symbol_table[count].data_type=strdup("CONST");
1081         symbol_table[count].line_no=countn;
1082         symbol_table[count].type=strdup("thelu");
1083         symbol_table[count].thisscope=scope;
1084         symbol_table[count].num_params=0;
1085         count++;
1086     }
1087     else if(c == 'c') { //constant character aksharam
1088         symbol_table[count].id_name=strdup(yy_text);
1089         symbol_table[count].data_type=strdup("CONST");
1090         symbol_table[count].line_no=countn;
1091         symbol_table[count].type=strdup("aksharam");
1092         symbol_table[count].thisscope=scope;
1093         symbol_table[count].num_params=0;
1094         count++;
1095     }
1096     else if(c == 's') { //constant string theega
1097         symbol_table[count].id_name=strdup(yy_text);
1098         symbol_table[count].data_type=strdup("CONST");
1099         symbol_table[count].line_no=countn;
1100         symbol_table[count].type=strdup("theega");
1101         symbol_table[count].thisscope=scope;
1102         symbol_table[count].num_params=0;
1103         count++;
1104     }
1105     else if(c == 'F') {
1106         symbol_table[count].id_name=strdup(yy_text);
1107         symbol_table[count].data_type=strdup(type);
1108         symbol_table[count].line_no=countn;
1109         symbol_table[count].type=strdup("Function");
1110         symbol_table[count].thisscope=scope;
1111         printf("\nSETTING %s's params to %d\n", symbol_table[count-curr_num_params].id_name, curr_num_params);
1112         symbol_table[count-curr_num_params].num_params=curr_num_params;
1113         curr_num_params=0;
1114         count++;
1115     }
1116     else if(c == 'L') {
1117         symbol_table[count].id_name=strdup(yy_text);

```

```

1118         symbol_table[count].data_type=strdup(type);
1119         symbol_table[count].line_no=countn;
1120         symbol_table[count].type=strdup("Library");
1121         symbol_table[count].thisscope=scope;
1122         symbol_table[count].num_params=0;
1123         count++;
1124     }
1125 }
1126 }
1127 else if(c == 'V' && q) {
1128     if(q != INT_MAX){
1129         sprintf(errors[sem_errors], "Line %d: Multiple declarations of \"%s\" not allowed!\n", countn+1, yy_text);
1130         sem_errors++;
1131     }
1132 }
1133 else{ // its scope is already destroyed, now it can be redeclared again into the symbol table with current scope
1134     // search again for that symbol table value
1135     int i;
1136     for(i=count-1; i>=0; i--) {
1137         if(strcmp(symbol_table[i].id_name, type)==0) {
1138             symbol_table[i].thisscope = scope;
1139             symbol_table[count].line_no=countn;
1140             symbol_table[count].num_params=0;
1141             printf("\nReinserted %s because its previous scope is finished\n", type);
1142             break;
1143         }
1144     }
1145 }
1146 }
1147 }
1148 }
1149 void insert_type() {
1150     strcpy(type, yy_text);
1151 }

```

SAMPLE INPUT:

≡ input3.txt

```

1  thechko numpy;
2  sankhya a=10;
3  a=a-5;
4  theega y="sdf";
5  y=y+"scf";
6  okavela(a peddadi 2){
7      sankhya x=5;
8      x=x+2;
9  }

```

SAMPLE OUTPUT:

PS C:\Users\Karthik Chittoor\Desktop\compiler design lab> Get-Content input3.txt | my_program.exe

PHASE 1: LEXICAL ANALYSIS

thechko is import keyword (line: 1, column: 1)
numpy is an identifier (line: 1, column: 9)
; is the end of a statement (line: 1, column: 14)

is a new line (line: 1)
sankhya is a data type (line: 2, column: 1)
Parser found eol-input
Parser found import-lib;-input
a is an identifier (line: 2, column: 9)
saw varDeclareidyytext: a
= is an assignment operator (line: 2, column: 10)
10 is a integer3 (line: 2, column: 11)
Parser found int
; is the end of a statement (line: 2, column: 11)
Parser found datatypeId=exp

is a new line (line: 2)
a is an identifier (line: 3, column: 1)
CHECKING FOR a
saw pure id2= is an assignment operator (line: 3, column: 2)
a is an identifier (line: 3, column: 3)
CHECKING FOR a
saw pure id2 - is an arithmetic operator(line: 3, column: 5)
Parser found identifier
5 is an integer1 (line: 3, column: 4)
Parser found int
; is the end of a statement (line: 3, column: 4)
Parser found exp-arithmeticOp-exp
Parser found equation
type of identifier: Variable XXXXXXXXXXXXXXXXXXXX exp_type=sankhya

is a new line (line: 3)
theega is a data type (line: 4, column: 1)
y is an identifier (line: 4, column: 8)
saw varDeclareidyytext: y
= is an assignment operator (line: 4, column: 9)
"sdf" is a string (line: 4, column: 10)
Parser found string
; is the end of a statement (line: 4, column: 15)

Parser found datatypeId=exp

is a new line (line: 4)
y is an identifier (line: 5, column: 1)
CHECKING FOR y
saw pure id2= is an assignment operator (line: 5, column: 2)
y is an identifier (line: 5, column: 3)
CHECKING FOR y
saw pure id2+ is an arithmetic operator (line: 5, column: 4)
Parser found identifier
"scf" is a string (line: 5, column: 5)
Parser found string
; is the end of a statement (line: 5, column: 10)
Parser found exp-arithmeticOp-exp
Parser found equation
type of identifier: Variable XXXXXXXXXXXXXXXXXX exp_type=theega

is a new line (line: 5)
okavela is a telugu_if statement (line: 6, column: 1)
(is an open curly bracket (line: 6, column: 8)
a is an identifier (line: 6, column: 9)
CHECKING FOR a
saw pure id2peddadi is a comparison operator (line: 6, column: 11)
Parser found identifier
2 is an integer1 (line: 6, column: 19)
Parser found int
) is an closed curly bracket (line: 6, column: 19)
Parser found exp-compareOp-exp
{ is an open flower bracket (line: 6, column: 20)

is a new line (line: 6)
sankhya is a data type (line: 7, column: 5)
x is an identifier (line: 7, column: 13)
saw varDeclareidyytext: x
= is an assignment operator (line: 7, column: 14)
5 is a integer3 (line: 7, column: 15)
Parser found int
; is the end of a statement (line: 7, column: 15)

```

    is a new line (line: 7)
x is an identifier (line: 8, column: 5)
CHECKING FOR x
saw pure id2= is an assignment operator (line: 8, column: 6)
x is an identifier (line: 8, column: 7)
CHECKING FOR x
saw pure id2 + is an arithmetic operator(line: 8, column: 9)
Parser found identifier
2 is an integer1 (line: 8, column: 8)
Parser found int
; is the end of a statement (line: 8, column: 8)
Parser found exp-arithmeticOp-exp
Parser found equation
type of identifier: Variable XXXXXXXXXXXXXXXXXXXX exp_type=sankhya

```

```

    is a new line (line: 8)
} is an closed flower bracket (line: 9, column: 1)
Parser found EOL-bunch
Parser found bunch-equation-finish
Parser found EOL-bunch
Parser found bunch-varDeclare-finish
Parser found EOL-bunch

```

```

ERASING x from symbol table as its CURRENT SCOPE is FINISHED
Parser found if(cond){bunch}

```

```

    is a new line (line: 9)

```

```

    is a new line (line: 10)
Parser found eol elif_repeat
Parser found eol elif_repeat
Parser found ifElseLadder
Parser found bunch_of_statement if_else_ladder bunch
Parser found EOL-bunch
Parser found bunch-equation-finish
Parser found EOL-bunch
Parser found bunch-varDeclare-finish
Parser found EOL-bunch
Parser found bunch-equation-finish
Parser found EOL-bunch
Parser found bunch-varDeclare-finish
Parser found input-bunch_of_stmts-input

```

SYMBOL TABLE:

SYMBOL	DATATYPE	TYPE	LineNUMBER	SCOPE	numParams
numpy		Library	1	0	
a	sankhya	Variable	2	1	0
10	CONST	sankhya	2	0	
5	CONST	sankhya	3	0	
y	theega	Variable	4	1	0
"sdf"	CONST	theega	4	0	
"scf"	CONST	theega	5	0	
okavela	N/A	Keyword	6	1	0
2	CONST	sankhya	6	0	
x	sankhya	Variable	7	2147483647	0

PHASE 2: SYNTAX ANALYSIS

Inorder traversal of the Parse Tree:

```

program, input-bunch-input, import-lib-;-input, thechko, numpy, ;; eol-input, newline, empty, bunch-varDeclare-;-bunch, empty, datatypeId=exp, sankhya, a, =, INT, 10, ;; eol-bunch, newline, bunch-equation-;-bunch, empty, id=exp, a, =, AthematicOp, ID, a, -5;
theega y="sdf";
y=y+"scf";
okavela(a peddadi 2){
    sankhya x=5;
    x=x+2;
}

, INT, 5, ;; eol-bunch, newline, bunch-varDeclare-;-bunch, empty, datatypeId=exp, theega, y, =, STRING, "sdf", ;; eol-bunch, newline, bunch-equation-;-bunch, empty, id=exp, y, =, AthematicOp, ID, y, +, STRING, "scf", ;; eol-bunch, newline, bunch-IfElse-bunch, empty, ifElseLadder, if(cond){bunch}, if, (, condition, ID, a, peddadi, INT, 2, ), {, eol-bunch, newline, bunch-varDeclare-;-bunch, empty, datatypeId=exp, sankhya, x, =, INT, 5, ;; eol-bunch, newline, bunch-equation-;-bunch, empty, id=exp, x, =, AthematicOp, ID, x, +2;
}

, INT, 2, ;; eol-bunch, newline, empty, }, EOL-elifrepeat, newline, EOL-elifrepeat, newline, empty, empty, empty, empty,

```

PHASE 3: SEMANTIC ANALYSIS

Semantic analysis completed with no errors

SEMANTIC ERRORS:

1. Multiple Declarations of Variables/Functions:

On encountering an identifier in **variable_declaration** production, the semantic analyser looks up in the symbol table entries to check if any symbol is already declared with **symbol_table[i].id_name == \$.name**

Input:

```
1  sankhya a=3;
2  sankhya a=15;
```

Output:

PHASE 1: LEXICAL ANALYSIS

```
sankhya is a data type (line: 1, column: 1)
a is an identifier (line: 1, column: 9)
saw varDeclareidyytext: a
= is an assignment operator (line: 1, column: 10)
3 is a integer3 (line: 1, column: 11)
Parser found int
; is the end of a statement (line: 1, column: 11)
Parser found datatypeId=exp

is a new line (line: 1)
sankhya is a data type (line: 2, column: 1)
a is an identifier (line: 2, column: 9)
saw varDeclareid= is an assignment operator (line: 2, column: 10)
15 is a integer3 (line: 2, column: 11)
Parser found int
; is the end of a statement (line: 2, column: 11)
Parser found datatypeId=exp

is a new line (line: 2)

is a new line (line: 3)
Parser found EOL-bunch
Parser found EOL-bunch
Parser found bunch-varDeclare-finish
Parser found EOL-bunch
Parser found bunch-varDeclare-finish
Parser found input-bunch_of_stmts-input
```

SYMBOL	DATATYPE	TYPE	LineNUMBER	SCOPE	numParams
a	sankhya	Variable	1	1	0
3	CONST	sankhya 1	1	0	

PHASE 2: SYNTAX ANALYSIS

Inorder traversal of the Parse Tree:

```
program, input-bunch-input, empty, bunch-varDeclare-;-bunch, empty, datatypeId=exp, sankhya, a, =, INT, 3, ;, eol-bunch, n
ewline, bunch-varDeclare-;-bunch, empty, datatypeId=exp, sankhya, a, =, INT, 15, ;, eol-bunch, newline, eol-bunch, newline
, empty, empty,
```

PHASE 3: SEMANTIC ANALYSIS

```
Semantic analysis completed with 1 errors
- Line 3: Multiple declarations of "a" not allowed!
```

2.UNDECLARED VARIABLES:

On encountering an identifier in **exp** or **equation** or **bunch_of_statements** productions, the semantic analyser looks up in the symbol table entries to check if this symbol is already declared with **symbol_table[i].id_name = \$\$name**

Input:

```
1  sankhya b=a;
```

Output:

PHASE 3: SEMANTIC ANALYSIS

```
Semantic analysis completed with 1 errors  
- Line 2: Variable "a" not declared before usage!
```

3. TYPECASTING:

When parsing **equation/variable_declaration** , the semantic analyser evaluates the data type of the resulting expression in RHS of the equation. If RHS's datatypes does not match the datatype of the LHS (looks up in the symbol table for this), then semantic error is reported.

Input:

```
1  sankhya a;  
2  a="hello trichy!";
```


Output:

SYMBOL	DATATYPE	TYPE	LineNUMBER	SCOPE	numParams
a	sankhya	Variable	1	1	0
"hello trichy!"	CONST	theega	2	1	0

PHASE 2: SYNTAX ANALYSIS

Inorder traversal of the Parse Tree:

program, input-bunch-input, empty, bunch-varDeclare-;-bunch, empty, datatypeId, sankhya, a, ;; eol-bunch, newline, bunch-equation-;-bunch, empty, id=exp, a, =, STRING, "hello trichy!", ;; eol-bunch, newline, empty, empty,

PHASE 3: SEMANTIC ANALYSIS

Semantic analysis completed with 1 errors
- Line 2: Data type casting not allowed in equation

4.INVALID OPERATIONS ON DATATYPES:

The semantic analyser is designed to report invalid operations on certain datatypes associated with typecasting.

Examples: string + string //should be allowed because its concatenation

string – string // not allowed as '-' is not defined for strings

Input:

```
1 theega s="hello"+54;
```

Output:

SYMBOL	DATATYPE	TYPE	LineNUMBER	SCOPE	numParams
s	theega	Variable	1	1	0
"hello"	CONST	theega 1	1	0	
54	CONST	sankhya 1	1	0	

PHASE 2: SYNTAX ANALYSIS

Inorder traversal of the Parse Tree:

```
program, input-bunch-input, empty, bunch-varDeclare-;-bunch, empty, datatypeId=exp, theega, s, =, AthematicOp, STRING, "hello", +54;
, INT, 54, ;; eol-bunch, newline, empty, empty,
```

PHASE 3: SEMANTIC ANALYSIS

Semantic analysis completed with 1 errors
 - Line 2: operation among int and string in expression not allowed

5. SCOPE OF VARIABLES:

The symbol table stores the scope of each variable as an integer. Starting from 1, the scope increases by 1 on seeing a '{' and decreases by 1 on seeing a '}'.

Lower scope value -> higher the globality of that variable

All the variables within {} will be marked as out of scope/destroyed by changing their scope to +infinity. Now any variable which is already defined can be defined again in another scope after verifying its scope=infinity.

Input 1:

```
1  sankhya a=3;
2  okavela(a samanam 5){
3      sankhya a;
4      okavela(a chinnadi 6){
5          sankhya a=7;
6      }
7  }
8
```

Output 1:

SYMBOL	DATATYPE	TYPE	LineNUMBER	SCOPE	numParams
a	sankhya	Variable	1	1	0
3	CONST	sankhya 1	1	0	
okavela	N/A	Keyword	2	1	0
5	CONST	sankhya 2	1	0	
6	CONST	sankhya 4	2	0	
7	CONST	sankhya 5	3	0	

PHASE 2: SYNTAX ANALYSIS

Inorder traversal of the Parse Tree:

```
program, input-bunch-input, empty, bunch-varDeclare-;-bunch, empty, datatypeId=exp, sankhya, a, =, INT, 3, ;; eol-bunch, n
ewline, bunch-IfElse-bunch, empty, ifElseLadder, if(cond){bunch}, if, (, condition, ID, a, samanam, INT, 5, ), {, eol-bunc
h, newline, bunch-varDeclare-;-bunch, empty, datatypeId, sankhya, a, ;; eol-bunch, newline, bunch-IfElse-bunch, empty, ifE
lseLadder, if(cond){bunch}, if, (, condition, ID, a, chinnadi, INT, 6, ), {, eol-bunch, newline, bunch-varDeclare-;-bunch,
empty, datatypeId=exp, sankhya, a, =, INT, 7, ;; eol-bunch, newline, empty, }, EOL-elifrepeat, newline, empty, empty, emp
ty, }, EOL-elifrepeat, newline, empty, empty, empty, empty,
```

PHASE 3: SEMANTIC ANALYSIS

Semantic analysis completed with 2 errors
- Line 4: Multiple declarations of "a" not allowed!
- Line 6: Multiple declarations of "a" not allowed!

Input 2: // in this input scope of the variable is only within the '{ }' . so we are allowed to declare the same sankhya multiple times without triggering semantic errors.

```

1   sankhya a=3;
2   ✓ okavela(a samanam 5){
3   |     sankhya b;
4   |   }
5   ✓ lekaokavela(a samanam 6){
6   |     sankhya b;
7   |   }
8   ✓ lekapothe{
9   |     sankhya b;
10  }

```

Output2:

SYMBOL	DATATYPE	TYPE	LineNUMBER	SCOPE	numParams
a	sankhya	Variable	1	1	0
3	CONST	sankhya 1	1	0	
okavela	N/A	Keyword	2	1	0
5	CONST	sankhya 2	1	0	
b	sankhya	Variable	3	2147483647	0
lekaokavela	N/A	Keyword	5	1	0
6	CONST	sankhya 5	1	0	
lekapothe	N/A	Keyword	8	1	0

PHASE 2: SYNTAX ANALYSIS

Inorder traversal of the Parse Tree:

program, input-bunch-input, empty, bunch-varDeclare-;-bunch, empty, datatypeId=exp, sankhya, a, =, INT, 3, ;; eol-bunch, newline, bunch-IfElse-bunch, empty, ifElseLadder, if(cond){bunch}, if, (, condition, ID, a, samanam, INT, 5,), {, eol-bunch, newline, bunch-varDeclare-;-bunch, empty, datatypeId, sankhya, b, ;; eol-bunch, newline, empty, }, EOL-elifrepeat, newline, elif(cond){bunch}, empty, elif, (, condition, ID, a, samanam, INT, 6,), {, eol-bunch, newline, bunch-varDeclare-;-bunch, empty, datatypeId, sankhya, b, ;; eol-bunch, newline, empty, }, EOL-elifrepeat, newline, empty, else{bunch}, else, {, eol-bunch, newline, bunch-varDeclare-;-bunch, empty, datatypeId, sankhya, b, ;; eol-bunch, newline, empty, }, eol-bunch, newline, empty, empty,

PHASE 3: SEMANTIC ANALYSIS

Semantic analysis completed with no errors