CSPC62 - Compiler Design

Python compiler

GOKUL ADETHYA T - 106121045
RAGHAVAN BALANATHAN - 106121099
SRINIVASA RAGHAVAN P - 106121129

QUICK SORT ALGORITHM

```
def quick_sort(array, low, high):
    if(low < high):
        pivot = array[low]
        start = low + 1
        end = high
        while True:
            while start <= end and array[end] >= pivot:
                end = end - 1
            while start <= end and array[start] <= pivot:
                start = start + 1
            if start <= end:
                array[start], array[end] = array[end], array[start]
            else:
                break
        array[low], array[end] = array[end], array[low]
        idx = end
        quick_sort(array, start, idx-1)
        quick_sort(array, idx+1, end)
```

LEXICAL ANALYSIS

```
"import" {debug("IMPT"); return T_Import;}
"print"  {debug("Print"); return T_Print;}
"pass" {debug("Pass"); return T_Pass;}
"if" {debug("If"); return T_If;}
"in" {debug("In"); return T_In;}
"while" {debug("While"); return T_While;}
"break" {debug("Break"); return T_Break;}
"and" {debug("And"); return T_And;}
"def" {debug("Def"); return T_Def;}

">" {debug("GT"); return T_GT;}
"<" {debug("LT"); return T_LT;}
"return" {debug("Return"); return T_Return;}
"+" {debug("PL"); return T_PL;}
"-" {debug("MN"); return T_MN;}
"*" {debug("ML"); return T_ML;}
"/" {debug("DV"); return T_DV;}
">" {debug("GT"); return T_GT;}
"<" {debug("LT"); return T_LT;}

[0-9]+ {yylval.text = strdup(yytext); debug(yylval.text); return T_Number;}
[_a-zA-Z][_a-zA-Z0-9]* {yylval.text = strdup(yytext); debug(yylval.text); return T_ID;}
\"([^\"\n])*\" {yylval.text = strdup(yytext); debug(yylval.text); return T_String;}
\'([^\'\n])*\' {yylval.text = strdup(yytext); debug(yylval.text); return T_String;}
"#"([a-z]|[0-9]|[A-Z]|" ")* {}
{whitespace} {}
```

LEXICAL ERROR ANALYSIS

```c
static void debug(const char *X)
    {
        #ifdef DEBUG
        if(startFlag){ startFlag=0; }
        if(strcmp(X, "NL")==0) { printf("T_%s\n%d ", X, yylineno); }
        else {
            if(strlen(X) < 256){ printf("T_%s ", X);    }
            else{ printf("Max length crossed: %s, Length: %d", X, strlen(X)); exit(1);}
        }
        #endif
    }
```

- Characters that are not recognized by the lexer or are not allowed in the language.
- Sequences of characters that do not form valid tokens in the language.
- Incorrectly formatted numbers, such as "123.456.789”

Symbol Table

Scope	Name	Type	Declaration	Last Used Line
(0, 1)	array	ListTypeID	10	33
(0, 1)	quick_sort	Func_Name	13	13
(1, 3)	pivot	Identifier	15	21
(1, 3)	1	Constant	16	36
(1, 3)	start	Identifier	16	35
(1, 3)	end	Identifier	17	37
(1, 3)	True	Constant	18	18
(1, 3)	temp	Identifier	30	33
(1, 3)	temp1	Identifier	31	32
(1, 3)	idx	Identifier	34	37
(3, 5)	end	Identifier	20	27
(3, 25)	start	Identifier	22	22

Indentation

```
1  static int indent_depth(const char *K) {
2          int len = strlen(K), i, tab_count=1;
3          for(i=0; i< len ; i++) {
4              if(K[i]=='\t') tab_count++;
5              else break;
6          }
7          return tab_count;
8      }
```

Calculates the depth of indentation based on the number of tabs (\t) encountered.

```
1   [\t]*       {
2               depth = indent_depth(yytext);
3               if(depth < top()) {
4                   while (depth < top()) pop();
5                   yylval.depth = depth;
6                   debug("DD");
7                   return DD; }
8               if(depth == top()) {
9                   debug("ND");
10                  yylval.depth = depth;
11                  return ND;  }
12              if(depth > top()){
13                  push(depth);
14                  debug("ID");
15                  yylval.depth = depth;
16                  return ID; }
17          }
```

Parser

```
1   arith_exp : term {$$=$1;}
2             | arith_exp  T_PL  arith_exp {$$ = createOp("+", 2, $1, $3);}
3             | arith_exp  T_MN  arith_exp {$$ = createOp("-", 2, $1, $3);}
4             | arith_exp  T_ML  arith_exp {$$ = createOp("*", 2, $1, $3);}
5             | arith_exp  T_DV  arith_exp {$$ = createOp("/", 2, $1, $3);}
6             | T_MN arith_exp {$$ = createOp("-", 1, $2);}
7             | T_OP arith_exp T_CP {$$ = $2;} ;
```

YACC Rules for arithmetic expressions

Parser

```
1   assign_stmt : T_ID T_EQL arith_exp {insertRecord("Identifier", $1, @1
2               | T_ID T_EQL bool_exp {insertRecord("Identifier", $1, @1.
3               | T_ID  T_EQL func_call {insertRecord("Identifier", $1, @
4               | T_ID T_EQL T_OB T_CB {insertRecord("ListTypeID", $1, @1
5               | T_ID T_EQL T_OB call_args T_CB {insertRecord("ListTypeI
6               | T_ID T_OB term T_CB T_EQL term {checkLi
```

YACC Rules for assignment expressions

Parser

```
1  if_stmt : T_If bool_exp T_Cln start_suite {$$ = createOp("If", 2, $2, $4);}
2          | T_If bool_exp T_Cln start_suite elif_stmts {$$ = createOp("If", 3, $2, $4, $5);};
3
4  elif_stmts : else_stmt {$$= $1;}
5             | T_Elif bool_exp T_Cln start_suite elif_stmts {$$= createOp("Elif", 3, $2, $4, $5);};
6
7  else_stmt : T_Else T_Cln start_suite {$$ = createOp("Else", 1, $3);};
8
9  while_stmt : T_While bool_exp T_Cln start_suite {$$ = createOp("While",
```

YACC Rules for if-elif-else, while expressions

Parser

```
1  func_def : T_Def T_ID {insertRecord("Func_Name", $2, @2.first_line, currentScope);}
2             T_OP args T_CP T_Cln start_suite
3             {$$ = createOp("Func_Name", 3, createID_Const("Func_Name", $2, currentScope), $5,
   $8);};
4
5  func_call : T_ID T_OP call_args T_CP
6             {$$ = createOp("Func_Call", 2, createID_Const("Func_Name", $1, currentScope), $3);};
```

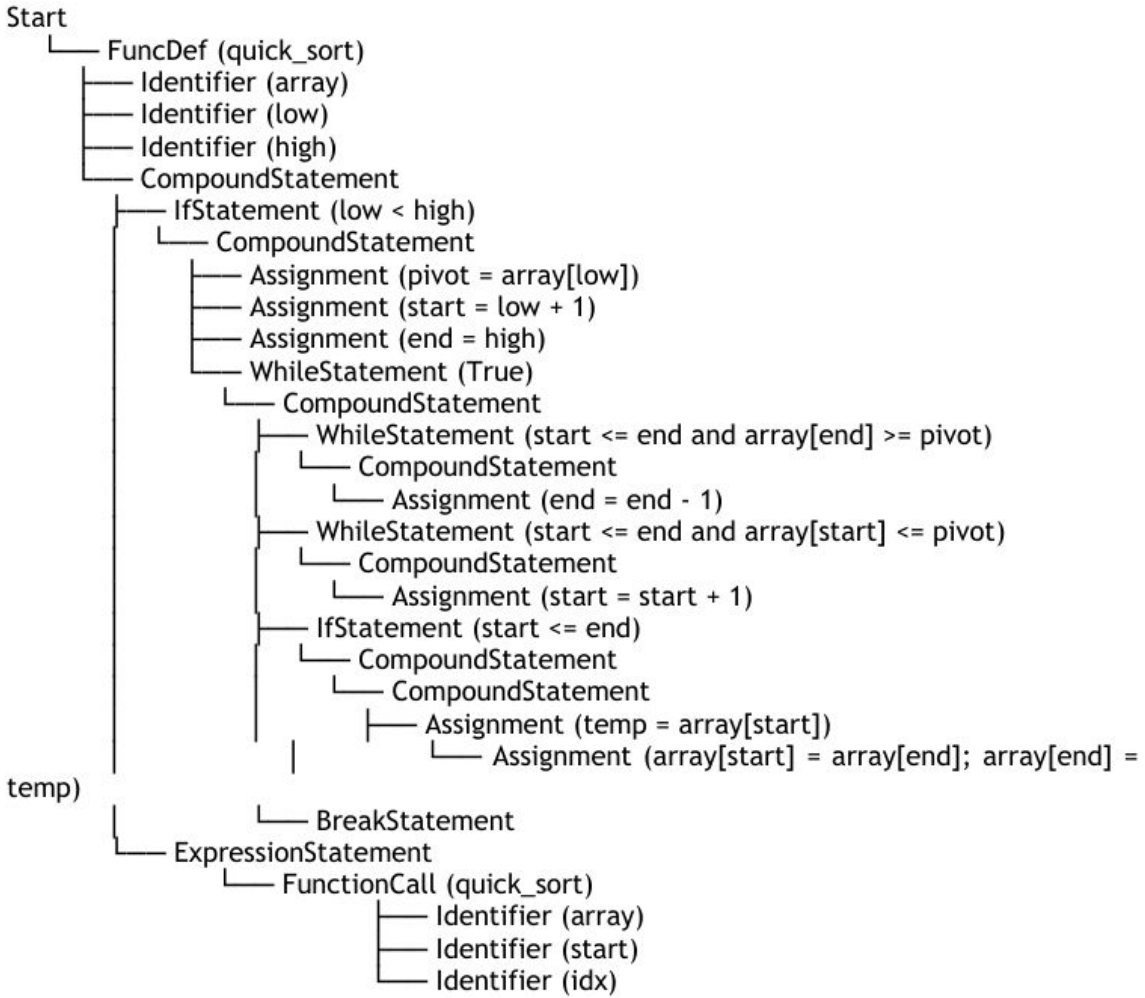
YACC Rules for function definitions and function call

Parsing error analysis

```
print 1+2
def f1():
  x=5
  while(x<10)
    if(xx==10):
    return x
  return x+2
print(f1()
```

```
----------------------------------Parsing Output-----------------
1 Syntax Error at Line 1, Column : 6
T_NL
T_NL
T_NL
3 Syntax Error at Line 4, Column : 13
T NL
4 T_If T_OP T_xx T_EQ Identifier 'xx' at line 5 Not Declared
T_NL
5 T_x Syntax Error at Line 6, Column : 4
T_NL
T_NL
Syntax Error at Line 8, Column : 10
T_EOF
```

Parse Tree for Quicksort Algorithm



Semantic Analysis

```
1  typedef struct ASTNode {
2      int nodeNo;
3      char *NType;
4      int noOps;
5      struct ASTNode** NextLevel;
6      record *id;
7  } node;
```

Each production rule defines actions associated with constructing an annotated parse tree.

The rule handles assignment statements. The actions include inserting records into a symbol table, creating nodes in the parse tree to represent assignments, and updating associated values in the symbol table.

```
1  assign_stmt : T_ID  T_EQL func_call {insertRecord("Identifier", $1, @1.first_line, currentScope);
2                $$ = createOp("=", 2, createID_Const("Identifier", $1, currentScope), $3);}
```

Semantic error analysis

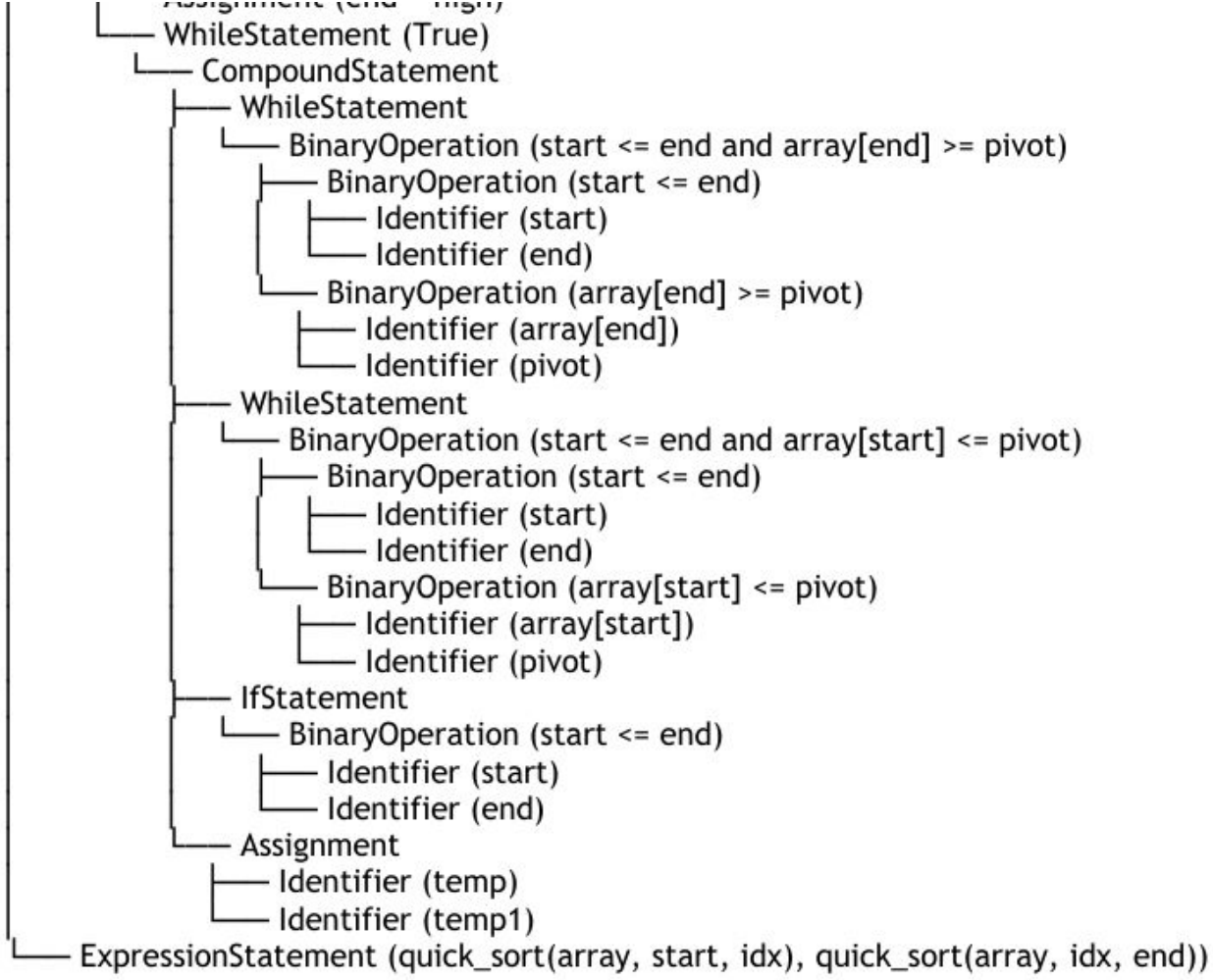
```
import random
x=10
def=5
while(xx<5):
  print(x+12)
```

```
------------------------------Semantic Output--------------------------
T_IMPT T_random
T_NL
2 T_x T_Assign T_10
T NL
T Def T Assign
Semantic Error at Line 3, Column : 4
T_NL
T_While T_OP T_xx T_LT Identifier 'xx' at line 4 Not Declared
T_NL
5 T_Print T_OP T_x T_Plus T_12 T_CP
T NL
```

Our parser detects two main categories of semantic errors during compilation:

- undeclared variables (line 4) and
- reserved identifier misuse (line 3).

Syntax Tree for Quicksort Algorithm



Intermediate Code Generation

```
1   typedef struct Quad {
2           char *R;
3           char *A1;
4           char *A2;
5           char *Op;
6           int I;
7       } Quad;
```

```
1   void makeQ(char *R, char *A1, char *A2, char *Op) {
2           allQ[qIndex].R = (char*)malloc(strlen(R)+1);
3           allQ[qIndex].Op = (char*)malloc(strlen(Op)+1);
4           allQ[qIndex].A1 = (char*)malloc(strlen(A1)+1);
5           allQ[qIndex].A2 = (char*)malloc(strlen(A2)+1);
6       }
```

BackPatching

```
void merge(List *dest, List *src1, List *src2) {
    dest->size = src1->size + src2->size;
    dest->capacity = dest->size;
    dest->array = realloc(dest->array, dest->size * sizeof(int));
    memcpy(dest->array, src1->array, src1->size * sizeof(int));
    memcpy(dest->array + src1->size, src2->array, src2->size * sizeof(int));
}
```

```
void addToFalseList(int quadIndex) {
    if (falseList.size == falseList.capacity) {
        falseList.array = realloc(falseList.array, (falseList.capacity * 2) * sizeof(int));
        falseList.capacity *= 2;
    }
    falseList.array[falseList.size++] = quadIndex;
}
```

Backpatching manages the presence of conditional expressions and loops.

It operates by maintaining True Lists and False Lists during parsing or intermediate code generation, which store the addresses where conditions evaluate to true or false.

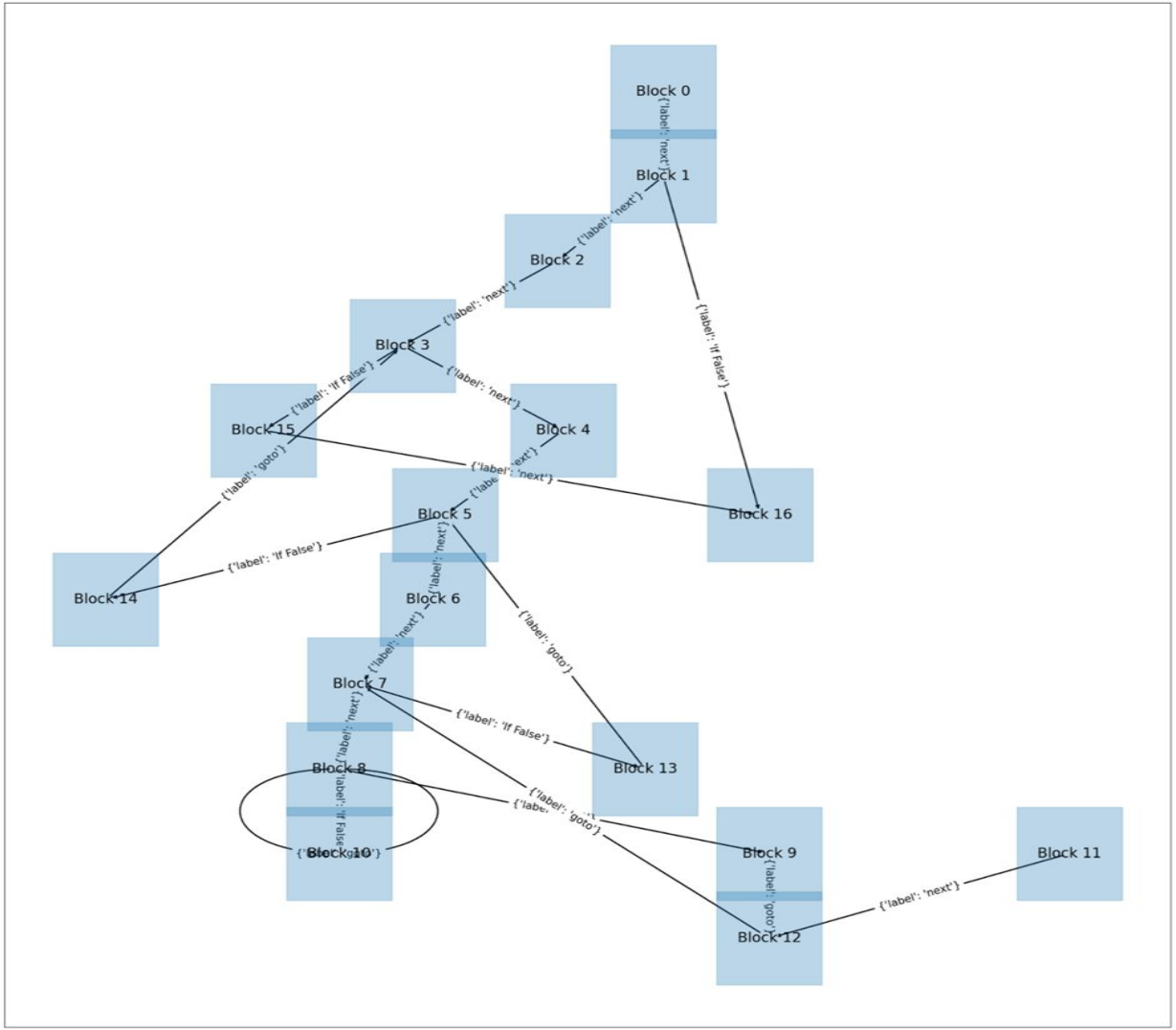
3 address codes - Quadruples For Quicksort

```
89      ListAssign      low     temp1   array
90      ListAssign      end     temp    array
95      Param   array   -       -
96      Param    array  -       -
97      Param    start  -       -
98      Call    quick_sort      3       T27
103     Param   array   -       -
104     Param    array  -       -
105     Param    idx    -       -
106     Call    quick_sort      3       T34
107     return  -       -       -
108     goto    -       -       L1
109     Label   -       -       L2
110     Label   -       -       L0
111     EndF    quick_sort      -       -
112     Param   array   -       -
113     Param    array  -       -
114     Param    0      -       -
115     Call    quick_sort      3       T37
```

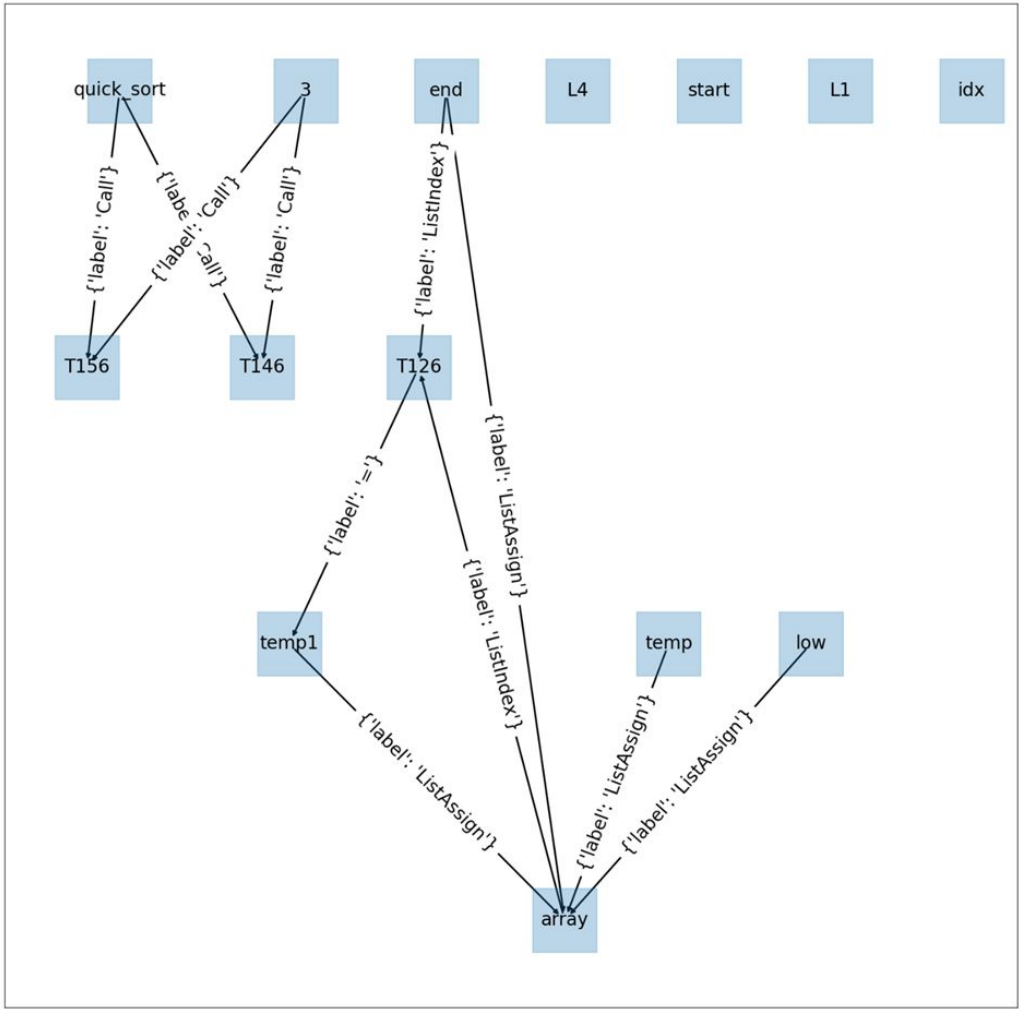
Basic Blocks - For Quicksort recursive call

```
('Block 14:\n'
 "['80', 'Label', None, None, 'L4']\n" - leader
 "['81', 'ListIndex', 'array', 'end', 'T126']\n"
 "['82', '=', 'T126', None, 'temp1']\n"
 "['83', 'ListAssign', 'low', 'temp1', 'array']\n"
 "['84', 'ListAssign', 'end', 'temp', 'array']\n"
 "['89', 'Param', 'array', None, None]\n"
 "['90', 'Param', 'array', None, None]\n"
 "['91', 'Param', 'start', None, None]\n"
 "['92', 'Call', 'quick_sort', '3', 'T146']\n"
 "['97', 'Param', 'array', None, None]\n"
 "['98', 'Param', 'array', None, None]\n"
 "['99', 'Param', 'idx', None, None]\n"
 "['100', 'Call', 'quick_sort', '3', 'T156']\n"
 "['101', 'return', None, None, None]\n"
 "['102', 'goto', None, None, 'L1']")
==============
"Block 15:\n['103', 'Label', None, None, 'L2']" - leader
==============
('Block 16:\n'
 "['104', 'Label', None, None, 'L0']\n" - leader
 "['105', 'EndF', 'quick_sort', None, None]")
==============
```

Control Flow Graph for Quicksort



DAG for Quicksort Recursive call



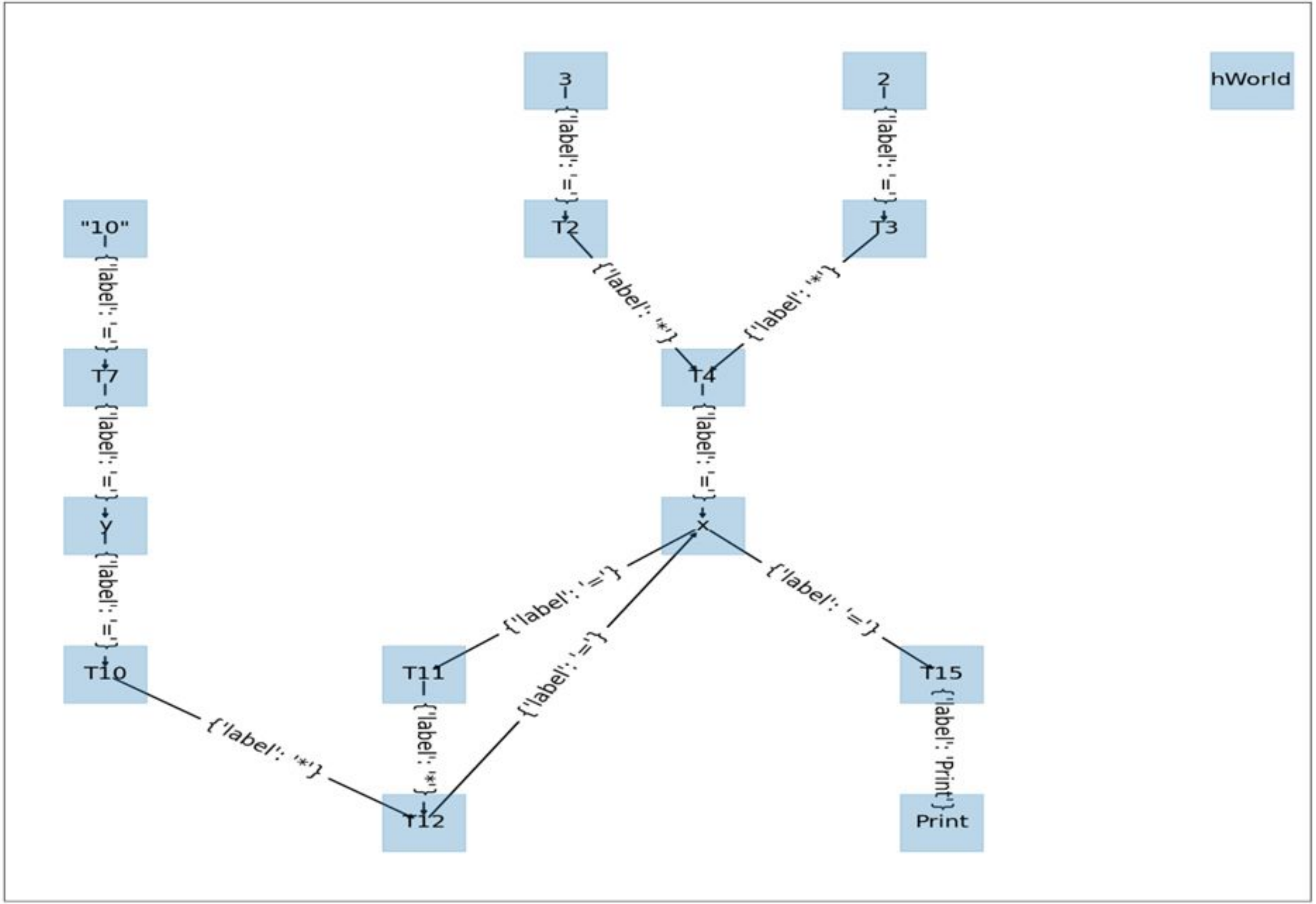
Code optimization

```
import hWorld
x=3*2
y="10"
x=y*x
print(x)
```

Three address code (Before optimization):

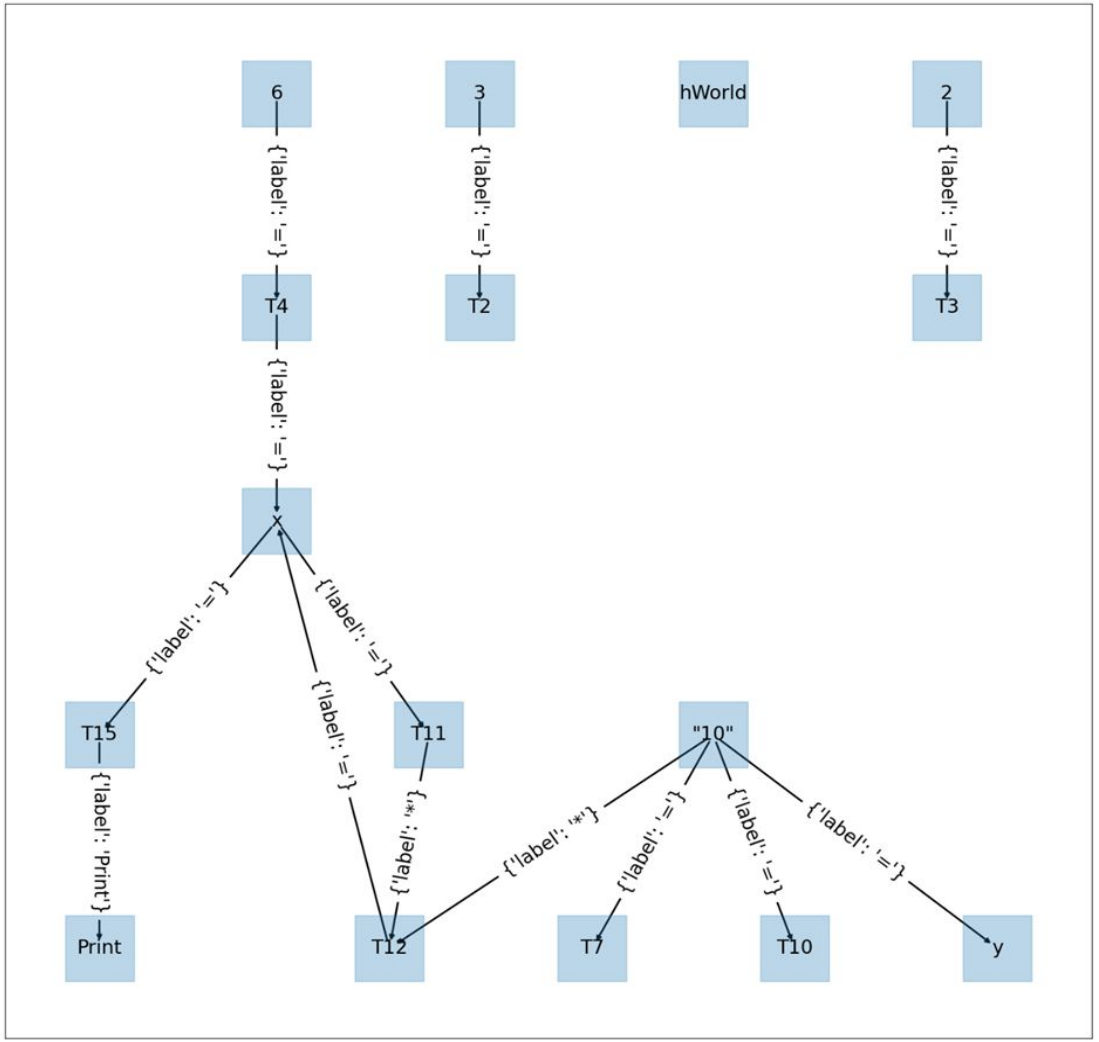
----------All Quads----------

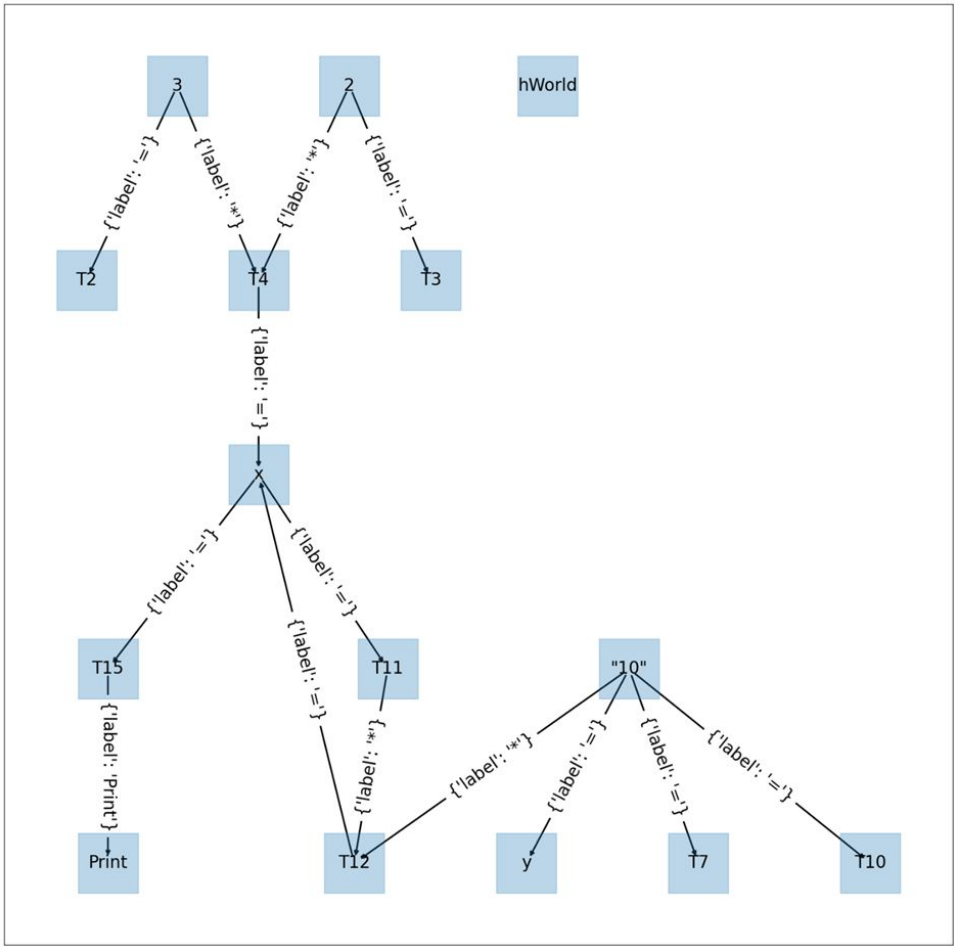
0	import	hWorld	-	-
1	=	3	-	T2
2	=	2	-	T3
3	*	T2	T3	T4
4	=	T4	-	x
5	=	"10"	-	T7
6	=	T7	-	y
7	=	y	-	T10
8	=	x	-	T11
9	*	T10	T11	T12
10	=	T12	-	x
11	=	x	-	T15
12	Print	T15	-	-

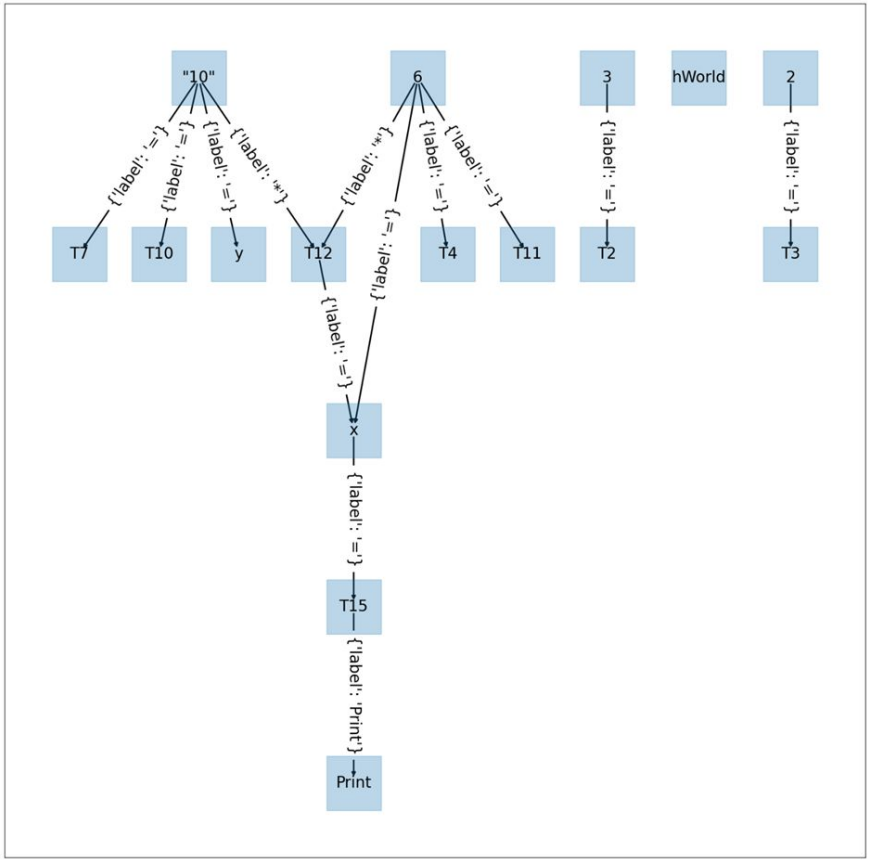


3
2
hWorld
"10"
T2
T3
T7
T4
Y
X
T10
T11
T15
T12
Print
{'label': '='}
{'label': '*'}
{'label': 'Print'}

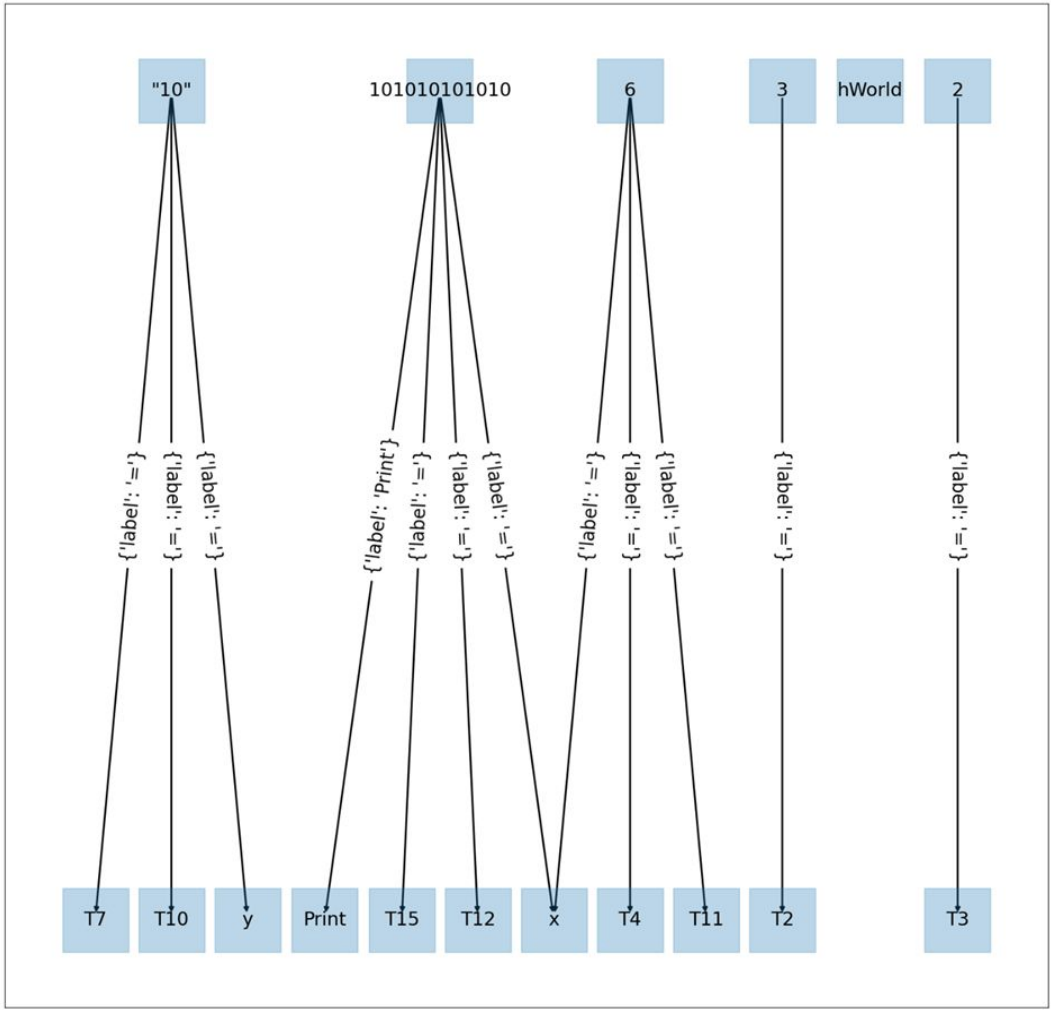
Copy propagation with induction variable elimination




3
2
hWorld
{'label': '='}
{'label': '*'}
{'label': '*'}
{'label': '='}
T2
T4
T3
{'label': '='}
x
{'label': '='}
{'label': '='}
{'label': '='}
T15
T11
"10"
{'label': 'Print'}
{'label': '*'}
{'label': '='}
{'label': '*'}
{'label': '='}
{'label': '='}
Print
T12
y
T7
T10


"10"
6
3
hWorld
2
{'label': '='}
{'label': '='}
{'label': '='}
{'label': '*'}
{'label': '*'}
{'label': '='}
{'label': '='}
{'label': '='}
{'label': '='}
T7
T10
y
T12
T4
T11
T2
T3
{'label': '='}
{'label': '='}
x
{'label': '='}
T15
{'label': 'Print'}
Print

Constant folding with copy propagation

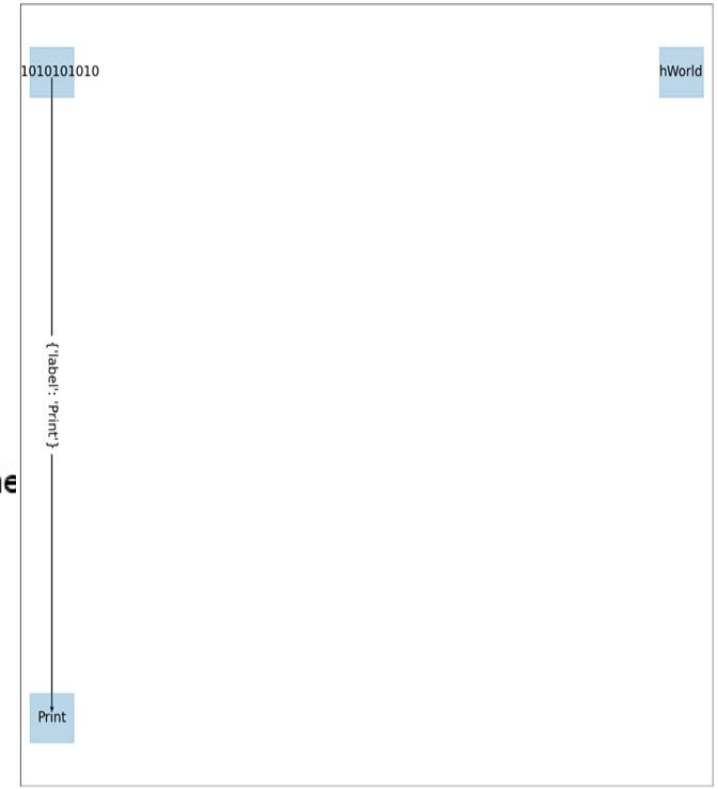


Dead Code Elimination with Peephole Optimization

Three address code (After optimization):

[[0: import, hWorld, None, None, 12: Print, 101010101010, None, None

```
import hWorld
print("101010101010")
```



Features implemented in our compiler

- **Lexical analysis**
  - Token identification
  - Lexical error detection
  - Symbol Table
- **Parser**
  - Syntax declaration
  - Indentation and syntactic error detection
  - Parse Tree
  - Abstract Syntax Tree
- **Semantic Analysis**
  - SDD + SDT
  - Annotated Parse Tree
  - Semantic Error detection

Features implemented in our compiler

- **ICG**
  - 3 address code - Quadruples
  - Backpatching
- **Code Optimization**
  - Basic blocks
  - DAG, CFG
  - Constant Folding + Copy Propagation
  - Dead Code Elimination
  - Peephole Optimization