School of Mathematical and Computational Sciences

Indian Association for the Cultivation of Science

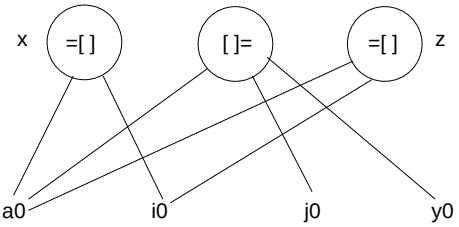
Compiler Construction: COM 5202

Tutorial XI (09 April, 2025)

M. Sc Semester IV: 2024-2025 *Instructor:* Goutam Biswas

Exercise 1. *The DAG for the following code sequence is given.*

```
x = a[i]
a[j] = y
z = a[i]
```



Identify the common sub-expression.

Ans. *There is no common sub-expression as j may be equal to i. The assignment* `a[j]=y` *kills* `a[i]`

Exercise 2. *Consider the following matrix multiplication program. The declared arrays are of type* `int a[20][20], b[20][20], c[20][20]`, `sizeof(int)` *is 4.*

```
for(i=0; i<n; ++i)
    for(j=0; j<n; ++j)
        c[i][j] = 0;
for(i=0; i<n; ++i)
    for(j=0; j<n; ++j)
        for(k=0; k<n; ++k)
            c[i][j] = c[i][j] + a[i][k]*b[k][j];
```

1. *Translate the program into a sequence of 3-address statements.*
2. *Draw the DAG corresponding to the biggest block.*
3. *Eliminate common subexpressions, perform copy propagation, and eliminate dead code.*

Exercise 3. **[20]**

In this assignment we shall put main computation in the body of our recursive descent parser of **Tutorial 9, Ex 3**. *We know from* **Tutorial VII, Ex. 3** *the equivalent grammar (after removal of left-recursion). It is as follows:*

$$
\begin{array}{rcl}
P & \rightarrow & \{\ DL\ :\ CL\ \} \\
DL & \rightarrow & D\ DL \mid \varepsilon \\
D & \rightarrow & TY\ VL \\
TY & \rightarrow & \texttt{zah} \mid \texttt{flt} \\
VL & \rightarrow & id\ VL' \\
VL' & \rightarrow & id\ VL' \mid \varepsilon \\
CL & \rightarrow & C\ CL' \\
CL' & \rightarrow & CL \mid \varepsilon \\
C & \rightarrow & PC \mid RC \mid AC
\end{array}
\qquad
\begin{array}{rcl}
PC & \rightarrow & \texttt{print}\ E \\
RC & \rightarrow & \texttt{input}\ id \\
AC & \rightarrow & id\ =\ E \\
E & \rightarrow & T\ E' \\
E' & \rightarrow & +\ T\ E' \mid -\ T\ E' \mid \varepsilon \\
T & \rightarrow & F\ T' \\
T' & \rightarrow & *\ F\ T' \mid /\ F\ T' \mid \varepsilon \\
F & \rightarrow & ic \mid fc \mid id \mid (\ E\ )
\end{array}
$$

Augment necessary C code to your recursive descent parser (with the symbol table) of the **Tutorial IX, Ex. 3**. *We already had discussion about how to pass synthesized and inherited attributes as parameters to the functions corresponding to non-terminals. As an example The prototype for the function corresponding to E' may be* `int Eprime(valType *ivP, valType *svP)`, *where* `valType` *is the type of data,* `ivP` *is a pointer to inherited attribute and* `svP` *is a pointer to synthesized attribute.*

A change is necessary regarding the access of the input file by the scanner. The scanner generator flex, by default, takes its input from `stdin`. *In last two assignments the input data, the program to parse, was written in a file (*`data`*), and it was redirected to the parser as* `$ ./a.out < data`.
But now our compiled code of the calculator will also take its data from `stdin`. *Naturally it will not work as the* `stdin` *redirected.*
To avoid this problem we supply the file name of the input file (`data`*) as a command line argument to the main program of the recursive descent parser-interpreter. It opens the file using* `fopen()` *call, and assign the file pointer (*`File *`*) to the global variable* `yyin` *defined by lex (*`extern FILE *yyin;`*). After this* `lex.yy.c` *will take its data directly from the file* `data`.

Use the `Makefile` *of* **Tutorial 9, Ex. 3** *to control the whole process.*
Finally prepare a `.tar` *file using the following command.*
`$ tar cvf <roll no>.11.tar lex.l recursiveDescent.c Makefile` *If you use any other header file of your own include that as well. Send the* `.tar` *file to* `goutamamartya@gmail.com`.

Following are a few sample of Input/Output. Note that while reading the input, the variable name e.g. `a0:` *is a prompt of my code. The actual input is say 10* `a0:  10`. **Input**

```
// Syntactically correct program
{
 : print 2*10/3+10
}
```

Output

```
$ ./a.out D0
Value: 16
```

Input

```
// Syntactically correct program
{
 : print 2.0*10.5/3.7+11.2
}
```

Output

```
$ ./a.out D1
Value: 16.875676
```

Input

```
// Syntactically correct program
{
 : print 2.0*10/3+11.2
}
```

Output

```
$ ./a.out D2
Value: 17.866667
```

Input

```
// Syntactically correct program
{
   zah  a0
   :
   input a0
   print a0
}
```

Output

```
$ ./a.out D3
a0: 10
Value: 10
```

Input

```
// Syntactically correct program
{
 flt  a0
 :
 input a0
 print a0
}
```

Output

```
$ ./a.out D4
a0: 12
Value: 12.000000
```

Input

```
// Syntactically correct program
{
 zah  a0 b0
 :
 input a0
 b0 = a0
 print b0
}
```

Output

```
$ ./a.out D5
a0: 12
Value: 12
```

Input

```
// Syntactically correct program
{
 zah  a0 b0
 :
 input a0
 b0 = 2*a0-5+11/3
 print b0
}
```

Output

```
$ ./a.out D6
a0: 20
Value: 38
```

Input

```
// Syntactically correct program
{
 flt  a0
 zah  b0
 :
 input a0
 input b0
 b0 = 3*a0 - b0/2
 print b0
}
```

Output

```
$ ./a.out D7
a0: 10
b0: 20
Var-Data type mismatch: b0 (9)
Value: 20
```