Department of Computer Science
Ashoka University

Programming Language Design and Implementation (PLDI): CS-1319-1

Quiz - 3: Offline — *Marks: 100*
Date: *December 03, 2022* — Time: *16:30 - 18:00*

Instructions:

1. The quiz can be taken from anywhere.
2. The quiz comprises one question (totalling 100 marks) and one bonus question (for 10 marks). Each question has multiple parts with marks shown for each.
3. The quiz is Open book, Open notes, and Open Internet.
4. Any copy from peers will be dealt with zero tolerance - both to get zero in the question.
5. The quiz will be available from 16:25.
6. The quiz must be submitted within 18:10.
7. No late or email or physical submission will be entertained.
8. You may choose between two options to prepare your answers:
   (a) Write the answers on paper. Take snaps and prepare a single PDF.
   (b) Alternately, you may type your answer in an editor (Notepad / WORD etc.) and save as PDF.
   - Naturally, you may use a mixed style too (like drawing a diagram on paper and include its image in the file you are typing).
   - Whatever way you prepare the answers, all should be made into a single PDF file.
9. Remember to write Name & Roll Number on every page and put pages in right order.
10. The PDF should be named with roll number and uploaded.
11. No question or doubt will be entertained. If you have any query, make your own assumptions, state them clearly in your answer and proceed.
12. Write in clear handwriting and in an unambiguous manner. If TAs have difficulty reading / understanding your answer, they will make assumptions at their best capacity to evaluate. You would not get an opportunity for explanation or rebuttal.

1. Consider the following program to find the integer square root of an integer by Babylonian iteration.

```
int recur(int a, int b);
int abs(int a, int b);

int main() {
    int n, x, y;
    n = 625;
    x = n;
    y = x + 2;
    while (abs(y, x) > 1) {
        y = x;
        x = recur(x, n);
    }
    return 0;
}
int recur(int a, int b) {
    return (a + b / a) / 2;
}
int abs(int a, int b) {
    int r;
    if (a < b)
        r = b - a;
    else
        r = a - b;
    return r;
}
```

Translate the above program to three address codes using the grammar and semantic actions discussed in Module 7. Specifically, write the following:

(a) Global Symbol Table showing the symbol name, data type, category, size, and offset. Mark appropriate parent / child pointers to build the tree of symbol tables. **[5]**

BEGIN SOLUTION

ST.glb				Parent: *Null*
Name	**Type**	**Category**	**Size**	**Offset**
recur	int× int → int	func	0	ST.recur
abs	int× int → int	func	0	ST.abs
main	void → int	func	0	ST.main

END SOLUTION

(b) Array of quad codes starting at index 100. **[20 + 10 + 10 = 40]**

i. For function `main`.

BEGIN SOLUTION

```
// Function main
[100]: t00 = 625
[101]: n = t00
[102]: x = n
[103]: t01 = 2
[104]: t02 = x + t01
[105]: y = t02
[106]: param x
[107]: param y
[108]: t03 = call abs, 2
[109]: t04 = 1
[110]: if t03 > t04 goto 112
[111]: goto 118
[112]: y = x
[113]: param n
[114]: param x
[115]: t05 = call recur, 2
[116]: x = t05
[117]: goto 106
[118]: ret 0
```

END SOLUTION

ii. For function `recur`.

BEGIN SOLUTION

```
// Function recur
[100]: t00 = b / a
[101]: t01 = a + t00
[102]: t02 = 2
[103]: t03 = t01 / t02
[104]: ret t03
```

END SOLUTION

iii. For function `abs`.

BEGIN SOLUTION

```
// Function abs
[100]: if a < b goto 102
[101]: goto 105
[102]: t00 = b - a
[103]: r = t00
[104]: goto 107
[105]: t01 = a - b
[106]: r = t01
[107]: ret r
```

END SOLUTION

(c) Symbol Tables for functions `main`, `recur`, and `abs` showing the symbol name, data type, category, size, and offset. Mark appropriate parent / child pointers to build the tree of symbol tables.

[4 + 3 + 3 = 10]

BEGIN SOLUTION

ST.main				Parent: *ST.glb*
Name	**Type**	**Category**	**Size**	**Offset**
n	int	local	4	-4
x	int	local	4	-8
y	int	local	4	-12
t00	int	temp	4	-16
t01	int	temp	4	-20
t02	int	temp	4	-24
t03	int	temp	4	-28
t04	int	temp	4	-32
t05	int	temp	4	-36

ST.recur				Parent: *ST.glb*
Name	**Type**	**Category**	**Size**	**Offset**
b	int	param	4	+12
a	int	param	4	+8
t00	int	temp	4	-4
t01	int	temp	4	-8
t02	int	temp	4	-12
t03	int	temp	4	-16

ST.abs				Parent: *ST.glb*
Name	**Type**	**Category**	**Size**	**Offset**
b	int	param	4	+12
a	int	param	4	+8
r	int	local	4	-4
t00	int	temp	4	-8
t01	int	temp	4	-12

END SOLUTION

2. We want to generate a better code when a constant occurs in an expression. For example, we presently generate

```
[100]: t00 = 2
[101]: t01 = 3
[102]: t02 = 4
[103]: t03 = t01 * t02
[104]: t04 = t00 + t03
[105]: a = t04
```

for `a = 2 + 3 * 4`. Clearly temporary variables `t00`, `t01`, and `t02` are useless. It would be better to translate as:

```
[100]: t03 = 3 * 4
[101]: t04 = 2 + t03
[102]: a = t04
```

(a) Modify the required actions appropriately to achieve this. [**10**]

BEGIN SOLUTION

$L \rightarrow L\ S\ \backslash\text{n}$
{}
$L \rightarrow S\ \backslash\text{n}$
{}
$S \rightarrow id = E$
{
$emit(id.loc = E.loc);$
}// No new temporary, copy code
$E \rightarrow E_1 + E_2$
{
$E.loc = gentemp();$
$emit(E.loc = E_1.loc + E_2.loc);$
}
$E \rightarrow E_1 - E_2$
{
$E.loc = gentemp();$
$emit(E.loc = E_1.loc - E_2.loc);$
}
$E \rightarrow E_1 * E_2$
{
$E.loc = gentemp();$
$emit(E.loc = E_1.loc * E_2.loc);$
}
$E \rightarrow E_1\ /\ E_2$
{
$E.loc = gentemp();$
$emit(E.loc = E_1.loc\ /\ E_2.loc);$
}
$E \rightarrow (E_1)$
{
$E.loc = E_1.loc;$
}// No new temporary, copy code
$E \rightarrow -E_1$
{
$E.loc = gentemp();$
$emit(E.loc = -E_1.loc);$
}
$E \rightarrow num$
{
$emit(E.loc = num.val);$
}// No new temporary, copy code - Modification here: E.loc = gentemp(); is removed
$E \rightarrow id$
{
$E.loc = id.loc;$
}// No new temporary, copy code

END SOLUTION

(b) Show the working of the modified scheme (symbol table and quads) for the following segment of `main` in Q 1: **[2 + 3 = 5]**

```
n = 625;
x = n;
y = x + 2;
```

BEGIN SOLUTION
Quads:

```
[100]: n = 625
[101]: x = n
[102]: t00 = x + 2
[103]: y = t00
```

Symbol Table:

ST.main				Parent: *ST.glb*
Name	**Type**	**Category**	**Size**	**Offset**
n	int	local	4	-4
x	int	local	4	-8
y	int	local	4	-12
t00	int	temp	4	-16

END SOLUTION

3. We want to add the support for treating an expression followed by semicolon (`;`) as a statement and an isolated semicolon (`;`) as a null statement. So with this, `a;`, `a+2;`, `recur(5, 3);`, `a+2;;`, etc. would be valid statements.

   Further, we want to add the support for assignment operator `=` in expression. With this, the value of the assigned (result) expression would be the value assigned to the left hand side from the right hand side. That is,

   ```
   a = 1;
   b = a;
   c = b;
   ```

   can be equivalently expressed as

   ```
   c = b = a = 1;
   ```

   Naturally, assignment operator `=` is right associative.

   For these supports, you need to remove the assignment statement rule:

   $$S \rightarrow \texttt{id} = E\ ;$$

   and add the rules:

   $$\begin{aligned} E &\rightarrow E = E \\ S &\rightarrow E\ ; \\ S &\rightarrow\ ; \end{aligned}$$

   (a) Write actions for these rules to get the desired translation. Be careful not to create any side-effect so that actions in other rules might break. **[5 + 3 + 2 = 10]**

BEGIN SOLUTION

$$
\begin{aligned}
E &\rightarrow E_1 = E_2; \\
&\quad \{ \\
&\qquad E.loc = E_1.loc; \\
&\qquad emit(E.loc' =' E_1.loc\ ' +'\ E_2.loc); \\
&\quad \} \\
S &\rightarrow E\ ; \\
&\quad \{ \\
&\qquad S.nextlist = null; \\
&\quad \} \\
S &\rightarrow\ ; \\
&\quad \{ \\
&\qquad S.nextlist = null; \\
&\quad \}
\end{aligned}
$$

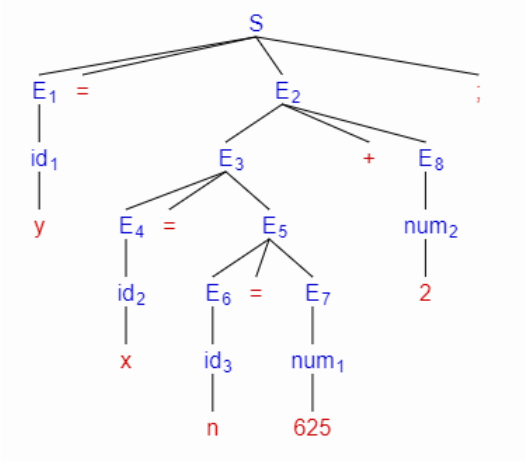
END SOLUTION

(b) Show the working of the modified scheme (parse tree and generated quads) for the following rewrite **[5 + 10 = 15]**

```
y = (x = n = 625) + 2;
```

BEGIN SOLUTION

Parse Tree:




```
L100: t00 = 625
L101: n = t00
L102: x = n
L103: t01 = 2
L104: t02 = x + t01
L105: y = t02
```

END SOLUTION

in `main` in Q 1 for the segment

```
n = 625;
x = n;
y = x + 2;
```

(c) Compare your current translation with the earlier one. [**5**]

BEGIN SOLUTION

The present translation matches the earlier one:

```
L100: t00 = 625
L101: n = t00
L102: x = n
L103: t01 = 2
L104: t02 = x + t01
L105: y = t02
```

So we get better semantic flexibility without any loss of performance in translation or quality in translated codes.

Besides the above chain assignments, this also allows us some convenient syntax like:

```
FILE * fp;

// Several statements

if (fp = fopen("Input.txt", "r")) {
    // Use fp
}
else {
    // Mark input error
}
```

Without the above modification, we would have probably written:

```
FILE * fp = fopen("Input.txt", "r");

// Several statements

if (fp) { // Which file fp should talk about
    // Use fp
}
else {
    // Mark input error
}
```

END SOLUTION