Построение промежуточного кода для фрагмента на языке C

1. Исходный код (F6)

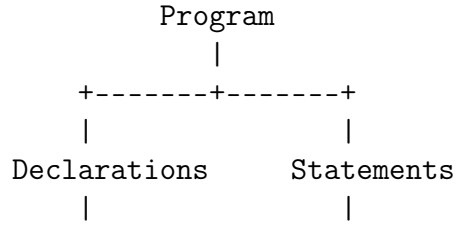
```
char[] str = "string";
int a, b, c;
a = 3;
if (str[a] == '\n') {
    b = str[a-1]; c = 0;
} else {
    b = str[a+1]; c = 1;
}
str[b] = 0;
printf("%s", str);
```

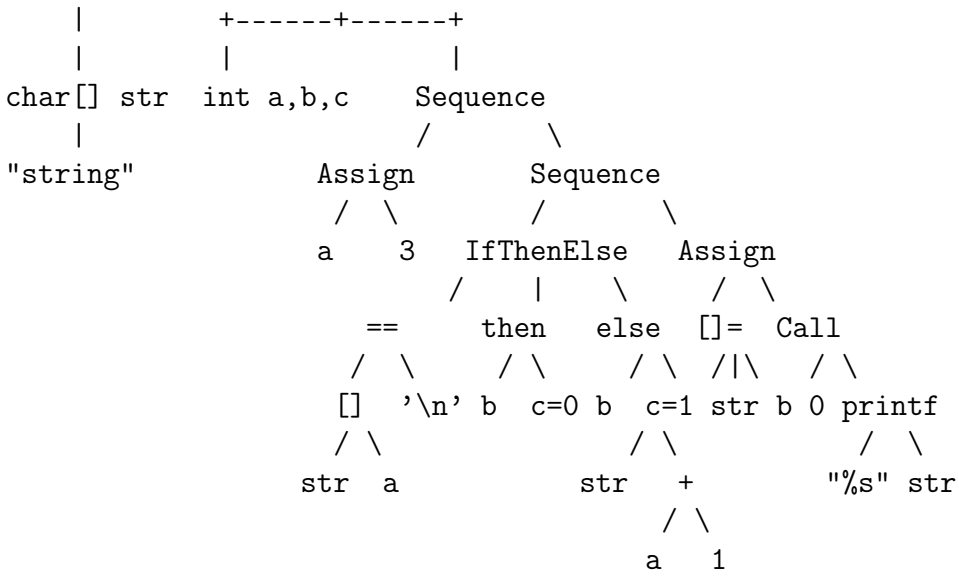
2. Типология команд промежуточного кода

Для трехадресного кода в форме четверок используем следующие типы команд:

- **Присваивание:** (:=, аргумент1, -, результат)
- **Арифметические операции:** (операция, аргумент1, аргумент2, результат)
  - +, -, *, /, %
- **Операции с массивами:**
  - Получение элемента: ([], массив, индекс, результат)
  - Запись в элемент: ([]=, значение, массив, индекс)
- **Операции сравнения:** (операция, аргумент1, аргумент2, результат)
  - ==, !=, <, >, <=, >=
- **Условный переход:** (if, условие, -, метка)
- **Безусловный переход:** (goto, -, -, метка)
- **Метка:** (label, -, -, имя_метки)
- **Вызов функции:** (call, функция, аргументы, результат)
- **Возврат:** (return, значение, -, -)

3. Синтаксическое дерево (C1)





4. Трехадресный код в форме четверок (С3)

№	Операция	Аргумент1	Аргумент2	Результат
1	:=	"string"	-	str
2	:=	3	-	a
3	[]	str	a	t1
4	==	t1	10 (символ ”)	t2
5	if	t2	-	L1
6	goto	-	-	L2
7	label	-	-	L1
8	-	a	1	t3
9	[]	str	t3	b
10	:=	0	-	c
11	goto	-	-	L3
12	label	-	-	L2
13	+	a	1	t4
14	[]	str	t4	b
15	:=	1	-	c
16	label	-	-	L3
17	[]=	0	str	b
18	param	str	-	-
19	param	"%s"	-	-
20	call	printf	2	-

5. Оптимизация промежуточного кода

Протокол оптимизирующих действий:

1. **Свёртка констант (Constant Folding):**
   - Строка 2: a = 3 (константа)
   - Строка 8: t3 = a - 1 = 3 - 1 = 2
   - Строка 13: t4 = a + 1 = 3 + 1 = 4
2. **Распространение констант (Constant Propagation):**

- Заменяем все вхождения 'a' на значение 3
- Строка 3: t1 = str[3] (вместо str[a])

3. **Удаление общих подвыражений (Common Subexpression Elimination):**
   - В данном примере нет повторяющихся вычислений
4. **Удаление мёртвого кода (Dead Code Elimination):**
   - Переменная 'c' не используется после присваивания
   - Удаляем строки 10 и 15
5. **Упрощение условных переходов (Branch Optimization):**
   - Вычисляем условие на этапе компиляции: str[3] == 10?
   - Символ 'string'[3] = 'i' (код 105), что не равно 10
   - Условие всегда ложно, удаляем ветку then

Оптимизированный код:

№	Операция	Аргумент1	Аргумент2	Результат
1	:=	"string"	-	str
2	[]	str	3	t1
3	if	t1	10	L1
4	goto	-	-	L2
5	label	-	-	L1
Удалено как недостижимый код				
11	goto	-	-	L3
12	label	-	-	L2
13	[]	str	4	b
16	label	-	-	L3
17	[]=	0	str	b
18	param	str	-	-
19	param	"%s"	-	-
20	call	printf	2	-

После удаления недостижимых меток и переходов:

№	Операция	Аргумент1	Аргумент2	Результат
1	:=	"string"	-	str
2	[]	str	4	b
3	[]=	0	str	b
4	param	str	-	-
5	param	"%s"	-	-
6	call	printf	2	-

6. Преобразование обратно в C (C3 → C)

```c
#include <stdio.h>

int main() {
    char str[] = "string";
    int b;

    //
    b = str[4];       // str[4] = 'n'
    str[b] = 0;       // b = 110 (       'n'), str[110] -
                      !

    printf("%s", str);
    return 0;
}
```

7. Обнаруженная проблема

При анализе обнаруживается серьезная ошибка в исходном коде:

- str[4] = 'n' (код ASCII 110)
- str[b] = str[110] - попытка записи за пределами массива
- Это приводит к неопределенному поведению

8. Исправленный оптимизированный код

Учитывая семантику исходной программы, исправляем код:

```c
#include <stdio.h>

int main() {
    char str[] = "string";
    int b;

    if (str[3] == '\n') {        // str[3] = 'i' = 105
        b = str[2];              // 'r' = 114
    } else {
        b = str[4];              // 'n' = 110
    }

    //
    if (b >= 0 && b < 6) {
        str[b] = 0;
    } else {
        //
        printf("Error: index %d out of bounds\n", b);
        return 1;
    }

    printf("%s", str);
    return 0;
}
```

9. Выводы

1. Построен трехадресный код в форме четверок для исходной программы
2. Определена типология команд промежуточного кода
3. Проведена оптимизация: свертка констант, распространение констант, удаление мёртвого кода
4. Обнаружена потенциальная ошибка выхода за границы массива
5. Оптимизированный код преобразован обратно в C с учетом исправлений
6. Оптимизация позволила:
    - Удалить условный переход (ветка then никогда не выполняется)
    - Упростить вычисления (константные выражения)
    - Удалить неиспользуемые переменные (c)