hw1实验过程记录

参考材料

Task 1 (爬树) 完全参考课件week1.pptx给出的伪代码,代码风格参考lib/utils/printer/slpast_xmlprinter.c。

• Task 1中哈希表的哈希函数参考知乎: 实操案例:字符串哈希表操作-知平 (zhihu.com)

Task 2 参考hw2/docs/lab2.md中对Yacc规则部分的陈述。

实验过程

Task 1: 递归遍历AST, 生成LLVM中间代码

生成代码的过程实际上就是爬树,即递归地对遍历到的节点应用处理函数,具体如下:

• 输入: 变量表和Optional[id]

• 处理:向表中添加必要的表项,输出必要的LLVM IR指令

• 返回: 新的变量表和Optional[string/integer]

• 尽管伪代码要求我们返回一个新表,实际代码可通过传入指向变量表的指针来直接修改(添加表项)

变量表

首先关注变量表的实现:由于提供的table.h/c很难处理key为string的情形,故包装了strtable.h/c,仅修改了

• 哈希函数: 改为字符串哈希

• key的比较:将比较地址改为比较字符串内容(strcmp())

最终使用了更新的symbol.h/c。

生成LLVM code

考虑处理过程,可根据节点类型细分为:

- 1. 处理Statement List
- 2. 处理Statement
- 3. 处理Expression

Statement List的处理很简单,只需先call左边的Statement,然后递归处理右边的Statement List,无需返回值。

Expression的处理比较复杂:

- 首先, Expression需要返回对应的常数或者变量索引值, 按虎书的程序设计风格定义了
 - 结构体 AS_expResult_: 存放常数或id值
 - o 这里直接用索引值而不用id (string):在LLVM IR中没有变量id的概念,只有%x (x为正数)即索引值的概念
- Expression需要记录计算得到的结果
 - 。 或者是记录在传入的id中,或者是记录在临时变量中,其中记录是指添加一条表项记录id到索引值的映射关系
 - o 两种记录都需要根据变量表 (LLVMIR_num) 获取新的索引值
- 对不同的Expression, 处理方式不同:
 - o numConst: 直接返回该num
 - 。 idExp: 在变量表中查找到id对应的索引值, 然后返回
 - minusExp: 对 -e, 递归地处理Expression e, 然后输出

```
1 \mid %x = sub i64 0, e
```

- o escExp: 先call左边的Statement List, 然后递归处理右边的Expression并返回右边的结果
- o opExp: 先call左边的Expression, 再call右边的Expression, 最后输出

```
1 \mid %x = op i64 e1, e2
```

Statement的处理根据其类型的不同可分为:

- assign
 - 。 直接获取左边的id, 然后call右边的Expression
 - o 检查Expression的返回值
 - 如果是字面常量,输出

```
1 \mid %x = add i64 0, numConst
```

- 如果是索引值,只需要添加对应的表项,无需输出LLVM IR代码
 - 原因:在Expression中已经输出过了
- putint
 - o call右边的Expression
 - 输出

```
1 call void @putint(i64 e)
2 call void @putch(i64 10)
```

- putch
 - o call右边的Expression
 - 。 输出

Task 2: 增加对 putint(e1, e2) 语法的支持

• 目标: 支持 putint(e1, e2) 的解析,它解释为 putint(e1); putint(e2);

Q: putint(e1, e2); 解释为 putint(e1); putint(e2); 和 先计算 e1, e2 再打印的区别?

- 由于FDMJ中的Expression有副作用,如果 e2 里包含 putint 的输出语句就会出错
- 匹配 putint(e1, e2); : 为Statement的生成式增加一条规则

```
1 | PUTINT '(' EXP ',' EXP ')' ';'
```

- 解析 putint(e1, e2);
 - 。 由于返回值的类型必须是 A_stm , 故不能直接转换为一个Statement List

```
1 A_stmList(A_putint($1, $3), A_stmList(A_putint($1, $5), NULL))
```

考虑 Escape Expression 的副作用,可以将 putint(e1, e2); 转换为 putint({putint(e1);}e2),故动作为

```
1  PUTINT '(' EXP ',' EXP ')' ';' {
2    $$ = A_Putint($1, A_EscExp($1, A_StmList(A_Putint($1, $3), NULL),
    $5));
3  };
```

测试结果

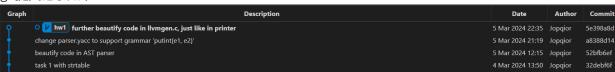
特别地,对于test4.fdmj:

```
public int main() {
   val=5;
   val=val+({putint(val);val=3;} val-1)*val;
   putint(val);
}
```

输出结果为 5\n11\n, 因为转换成AST的过程总是先处理左边的Expression, 再处理右边的Expression。如果先处理右边的Expression,再处理左边的Expression,结果就会不同 (由于Escape Expression有副作用)。

开发过程

git提交记录如下:



● 第一次提交完成了llvmgen.c的修改,并添加了utils/dsa/strtable.h/c

```
task 1 with strtable

Commit: 32debf6f8708bb95419ea2cc1ff558c9ffd1e572

Parents: fb9abf13be94807d46f91ceef30461641b568d2e

Author: Jopqior <imzwan@163.com>
Committer: Jopqior <imzwan@163.com>
Date: Mon Mar 04 2024 13:50:21 GMT+0800 (GMT+08:00)

task 1 with strtable

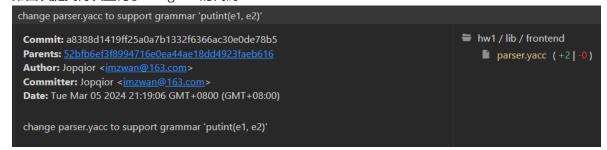
include / utils / dsa
```

• 第二次提交重构了llvmgen.c的代码

• 第三次提交修改了parser.yacc, 支持 putint(e1, e2)



• 第四次提交再次重构了llvmgen.c的代码



最后添加symbol.h/c,更改代码:

