Tiger Compiler

前言

Tiger Compiler是大三上编译原理的课程大作业，也是软院难度最高的课程Lab，令我至今记忆犹新😇。

在这门课中，我们实现了一个包含完整前端和后端的编译器，能将Tiger语言的代码转化为可以直接解释执行的汇编代码。其中，前端的功能包括词法分析、语法分析、构建抽象语法树、类型检查、翻译成中间代码，后端包括指令选择、数据流分析和寄存器分配。可以说是极其极其的硬核...

这个项目已经脱离了八股的范畴，平时写代码也根本不会接触到这么底层的东西。事实上，在我面实习岗位的时候，没有面试官会去深入问Tiger Compiler相关的问题，所以大可以放心把它放简历上增色（bushi）

遗憾的是，大三上学期的我太过于摆烂，各种作业都逾期了😭，所以也就没有写作为bonus的Lab7 GC，失去了一次极大提升自己代码能力和debug能力的机会。以后再弥补吧。

Lab1 Straight-line Program Interpreter

这个Lab是用来热身的，用来给C++的萌新们熟悉熟悉语言，只能说毫无压力。

Lab2 Lexical Analysis

在这个Lab中，我们会用 `flexc++` 去实现一个lexical scanner，将代码转化为Tiger Token

部分代码示例如下，此外还需要对一些特殊字符如 `\` `"` 进行处理：

```
64      /* reserved words */
65     "while" {adjust(); return Parser::WHILE;}
66     "for" {adjust(); return Parser::FOR;}
67     "to" {adjust(); return Parser::TO;}
68     "break" {adjust(); return Parser::BREAK;}
69     "let" {adjust(); return Parser::LET;}
70     "in" {adjust(); return Parser::IN;}
71     "end" {adjust(); return Parser::END;}
72     "function" {adjust(); return Parser::FUNCTION;}
73     "var" {adjust(); return Parser::VAR;}
74     "type" {adjust(); return Parser::TYPE;}
75     "array" {adjust(); return Parser::ARRAY;}
76     "if" {adjust(); return Parser::IF;}
77     "then" {adjust(); return Parser::THEN;}
78     "else" {adjust(); return Parser::ELSE;}
79     "do" {adjust(); return Parser::DO;}
80     "of" {adjust(); return Parser::OF;}
81     "nil" {adjust(); return Parser::NIL;}
```

Lab3 Syntax Analysis

在这个Lab中，我们会用 `Bisonc++` 去实现一个parser，将LALR(1)上下文无关文法转换为C++类，并由这个C++类中的成员函数完成后续parse的工作。

代码示例如下，我们需要完成grammer的转换，以及完成token priority的定义：

```
76      /* TODO: Put your lab3 code here */
77    decs:    {$$ = new absyn::DecList();}
78      | decs_nonempty  {$$ = $1;}
79      ;
80
81    decs_nonempty:  decs_nonempty_s decs_nonempty  {$$ = ($2)->Prepend($1);}
82      | decs_nonempty_s {$$ = new absyn::DecList($1);}
83      ;
84
85    decs_nonempty_s:  tydec {$$ = new absyn::TypeDec(scanner_.GetTokPos(), $1);}
86      | vardec {$$ = $1;}
87      | fundec {$$ = new absyn::FunctionDec(scanner_.GetTokPos(), $1);}
88      ;
89
90    vardec:  VAR ID ASSIGN exp {$$ = new absyn::VarDec(scanner_.GetTokPos(), $2, NULL, $4);}
91      | VAR ID COLON ID ASSIGN exp {$$ = new absyn::VarDec(scanner_.GetTokPos(), $2, $4, $6);}
92      ;
93
94    fundec:  fundec_one fundec {$$ = ($2)->Prepend($1);}
95      | fundec_one {$$ = new absyn::FunDecList($1);}
96      ;
97
98    fundec_one:  FUNCTION ID LPAREN tyfields RPAREN EQ exp {$$ = new absyn::FunDec(scanner_.GetTokPos(), $2, $4, NULL, $7);}
99      | FUNCTION ID LPAREN tyfields RPAREN COLON ID EQ exp {$$ = new absyn::FunDec(scanner_.GetTokPos(), $2, $4, $7, $9);}
100     ;
```

在这一步之后，我们便能得到一个抽象语法树(Abstract Syntax Tree, AST)，语法树中树节点的定义和实现分别在 `absyn.h` 和 `absyn.cc` 中，下图便为AST树的定义：

```
49    /**
50     * Abstract syntax tree root
51     */
52    class AbsynTree {
53    public:
54      AbsynTree() = delete;
55      AbsynTree(nullptr_t) = delete;
56      explicit AbsynTree(absyn::Exp *root);
57      AbsynTree(const AbsynTree &absyn_tree) = delete;
58      AbsynTree(AbsynTree &&absyn_tree) = delete;
59      AbsynTree &operator=(const AbsynTree &absyn_tree) = delete;
60      AbsynTree &operator=(AbsynTree &&absyn_tree) = delete;
61      ~AbsynTree();
62
63      void Print(FILE *out) const;
64    private:
65      absyn::Exp *root_;
66    };
```

> 有时间可以复习一下context-free grammar和LL,LR,LALR，了解编译器parser选型背后的逻辑

Lab4 Type Checking

在这个Lab中，我们需要编写各个C++类中的 `SemAnalyze` 函数来对抽象语法树(AST Tree)进行类型检查，并生成适当的关于类型不匹配或未声明的标识符的报错信息

语义分析阶段的一个主要工作是**符号表（symbol table, 也称为环境）**的管理，其作用是将symbol映射到它们的类型和存储位置。在处理类型、变量和函数声明时，这些symbol便与其在符号表中的“含义”相绑定。每当发现symbol的使用时，便在symbol table中查看其含义。

我们在这个Lab中管理了两个这样的symbol table：

1. `venv`：变量环境，管理变量名到类型的映射
2. `tenv`：类型环境，管理类型名到类型的映射

以下为一个C++类的 `SemAnalyze` 示例：

```
194     type::Ty *AssignExp::SemAnalyze(env::VEnvPtr venv, env::TEnvPtr tenv,
195                                     int labelcount, err::ErrorMsg *errormsg) const {
196       /* TODO: Put your lab4 code here */
197       if (typeid(*var_) == typeid(SimpleVar)) {
198         sym::Symbol *var_sym = ((SimpleVar *) this->var_)->sym_;
199         env::EnvEntry *var_entry = venv->Look(var_sym);
200         if (typeid(*var_entry) == typeid(env::VarEntry)) {
201           bool flag = ((env::VarEntry *) var_entry)->readonly_;
202           if (flag) errormsg->Error(this->pos_, "loop variable can't be assigned");
203         }
204       }
205       type::Ty *assign_type = this->var_->SemAnalyze(venv, tenv, labelcount, errormsg);
206       type::Ty *exp_type = this->exp_->SemAnalyze(venv, tenv, labelcount, errormsg);
207       if (typeid(*assign_type) == typeid(type::VoidTy)) {
208         return type::VoidTy::Instance();
209       }
210       else if (typeid(*assign_type) != typeid(*exp_type)) {
211         errormsg->Error(this->pos_, "unmatched assign exp");
212         return type::VoidTy::Instance();
213       }
214     }
```

Lab5 Part1: Escape Analysis

在这个Lab中，我们将完成对**逃逸变量（escape，传地址、内层嵌套函数访问）**的检测。

我们创建了一个不同于 `venv` 的环境 `esc::EscEnv env`，将变量映射至相应的 `esc::EscapeEntry`（包含 `depth` 和 `escape` 等信息）用于进行逃逸检测。**如果一个变量使用时所处的depth大于变量定义时所处的depth，则我们认为这个变量是逃逸的。**

以下为一个类型检查函数的示例：

```
153  void FunctionDec::Traverse(esc::EscEnvPtr env, int depth) {
154    /* TODO: Put your lab5 code here */
155    std::list<FunDec *> funDecList = this->functions_->GetList();
156    for (FunDec *dec : funDecList) {
157      env->BeginScope();
158      std::list<Field *> fieldList = dec->params_->GetList();
159      for (Field *f : fieldList) {
160        f->escape_ = false;
161        env->Enter(f->name_, new esc::EscapeEntry(depth + 1, &f->escape_));
162      }
163      dec->body_->Traverse(env, depth + 1);
164      env->EndScope();
165    }
166  }
```

Lab5 Part2: Translation

在这个Lab中，我们需要完成 `x64 stack frame` 、 `IR tree translation` 以及 `code generation` 的设计与实现。这是整个Tiger Comiler中代码量最大、难度最高、bug最隐蔽的一部分，需要大量时间和精力以及足够的耐心去完成。

1. `x64 stack frame` ：设计Tiger Compiler的栈帧结构，处理寄存器、栈分配等相关事务
2. `IR tree translation` ：将AST转换为**中间表示树（IR Tree，是一种抽象机器语言，可以表示目标机器的操作无需太多涉及机器相关的细节）**，一些树节点的示例：
   a. `CONST(i)` ：整型常数i
   b. `NAME(n)` ：符号常数n
   c. `TEMP(t)` ：临时变量t
   d. `BINOP(o, e1, e1)` ：对操作数e1、e2施加二元操作符o表示的操作
   e. `MEM(e)` ：开始于存储器地址e的 `wordSize` 个字节的内容（ `wordSize` 是在Frame模块中定义的）
   f. `CALL(f, l)` ：过程调用，以参数列表l调用函数f
   g. `ESEQ(s, e)` ：先计算语句s以形成其side-effect，然后计算e作为此表达式的结果
   h. `MOVE(TEMP t, e)` ：计算e并将结果送入临时单元t
   i. `MOVE(MEM(e1), e2)` ：计算e1生成地址a，然后计算e2，并将计算结果存储到从地址a开始的 `wordSize` 个字节的存储单元中
   j. `EXP(e)` ：计算e但是忽略结果
   k. `JUMP(e, labs)` ：将控制流转移到地址e，标号labs指出表达式e可能计算出所有的目标地址
   l. `CJUMP(o, e1, e2, t, f)` ：依次计算e1、e2，用比较符o比较e1、e2，若结果为true则

跳到地址t；否则跳转到f

m. `SEQ(s1, s2)`：语句s1之后跟随语句s2

n. `LABEL(n)`：定义名字n的常数值为当前机器代码的地址

3. `Code generation`：使用指令选择算法（如`Maximal Munch`、动态规划、树文法、快速匹配）生成汇编指令（寄存器现在尚未进行分配）

Lab6 Register Allocation

在这个Lab中，我们需要完成`liveness analysis`和`register allocation`两部分

1. `liveness analysis`：生成程序的控制流图(control flow graph)，并根据活跃分析的数据流方程计算出每一个节点入边和出边上活跃的变量，用来构造冲突图(conflict graph)。
2. `register allocation`：根据`liveness analysis`中得出的冲突图进行寄存器分配（图着色，NP-Complete问题）。对于溢出的变量，我们选择将其放到内存中(栈)。

> 数据流方程：
>
> in[n] = use[n] U (out[n] – def[n])
>
> out[n] = U in[s] （s为n的直接后继节点）

Lab7 Garbage Collection

遗憾...没有写，希望将来能够补上