编译原理小组作业指南与评分标准2023

by 詹奇

实验简介

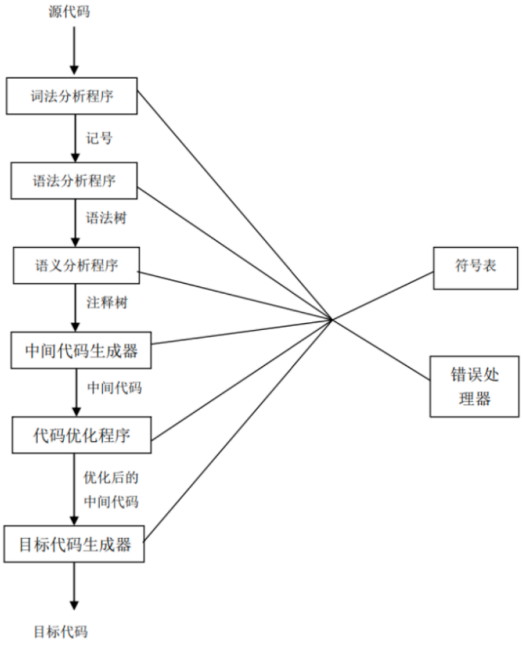
既然这门课程叫做编译原理, 那么我们的大作业自然就是实现一个编译器, 即一个将high-level的源代码转换成low-level代码的程序. 要求实现Tiger语言或一个类C或类PASCAL语言的编译器, 也可以自己创造一个功能等价的新语言.

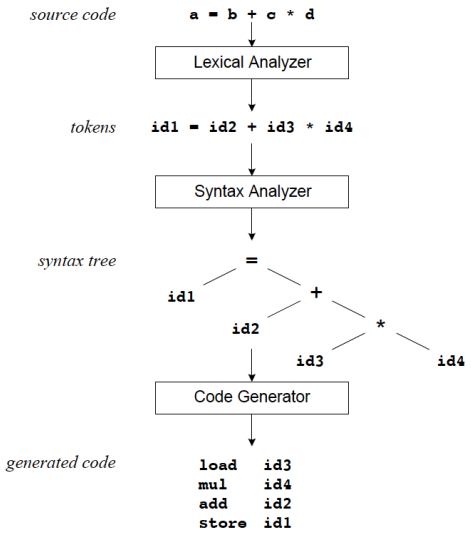
重要信息：

- **组队：该实验由1-3人组队完成, 不同人数的最低要求也是不同的. 小组由同学们自行组成，并在第三次课开始之前上报组队名单.**

- **截止日期：夏学期第五周5月28日晚上22:00.**
- **分工：如果是多人合作完成, 小组成员必须在实验报告中明确说明分工情况, 每位成员必须有coding的部分.**

一般来说, 编译器由词法分析, 语法分析, 语义分析, 代码优化与代码生成几个步骤组成, 下面我们逐一简要介绍.





词法分析

词法分析的目的是将源代码(一个个字符)解析成一个个token. Token 就是词法分析的最小单元, 表示一个程序有意义的最小分割. 下表为常见的token类型和样例:

类型	样例
标识符	x, sum, i
关键字	if, while, return
分隔符	}, (,), ;
运算符	+, < , =
字面量	true, 666, "hello world"
注释	// this is a comment.

当然这些只是例子, 在你实现的语言中, 并不需要一定支持上述样例, 比如你的注释不需要是`//`.

```
x = a + b * 2;
```

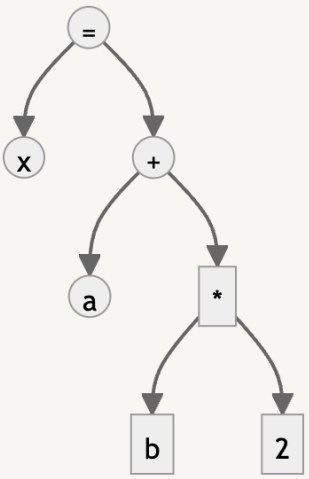
上面的简单代码会被解析为:
`[(identifier, x), (operator, =), (identifier, a), (operator, +), (identifier, b), (operator, *), (literal, 2), (separator, ;)]`.

词法分析显然是件很简单的事情, 本质上就是一个有限状态机(正则文法)的匹配. 只需要简单的遍历源代码中的字符, 根据字符的值分别判断即可.

语法分析

由词法分析得到的一个个token在语法分析阶段进行进一步的解析. 具体来说, 需要:

1. 对于一串合法的tokens, 生成语法树.
2. 对于的一串**不合法**的token, 检测到可能的错误并报告给用户.
   如下图所示, 上面例子中的一串token就可以解析为一个合法的语法树:



上下文无关文法回顾

如果说词法分析由正则文法为基础, 语法分析则有上下文无关文法作为基石. 实际编程中你往往需要为你的语言设计一个文法, 举例来说四则运算的表达式就可以利用下面的文法

表示.

```
E → int
E → E Op E
E → (E)
Op → + | - | * | /
```

词法分析和语法分析并不困难, 我们不再赘述.

> 很多语言都有支持词法分析和语法分析的库/包, 只需要你按照某种格式(BNF)写出词法和文法就可以直接生成对应的 lexer, parser. 你当然可以使用它们. 例如往年使用最多的也是个人作业中介绍的Lex Yacc, 或者 Antlr, lark 等等.
> 但值得指出的是, 自己手写一个词法分析器和语法分析器并没有那么困难, 比如你会发现语法分析时采用朴素的递归下降就足以解决绝大部分语言的文法了.可参考craftinginterpreter.
> **你可以直接使用现成的文法(BNF), 但不要直接使用他人提供的解析文法的代码.**

语义分析

> 与词语法分析的定式相比, 从语义分析和代码优化的自由度极大增加, 实现的内容和方式都可以自主选择.
> 下面的内容往往是举几个重要的例子或者列举几种主流的方法供大家参考, **并不是硬性标准.**

语义分析即基于语法分析的结果(语法树), 对程序的语义进行分析, 在语法树上增添语义信息, 建立符号表(symbol table)等. 我简单的将语义分析分为两个类型: 检查与推断. 以下列出一些你可以进行的(有些是必须要进行的)的语义分析:

语义推断

- 变量绑定

```
int x = 1; // x1
int y = 0;
{
  int x = 3; // x2
  y += x; // x2
}
y += x; // x1
```

对于上述代码, 容易发现 `y+=x` 加的两个 `x` 来自于两个不同的定义, 那么编译器在生成代码的时候, 仅仅看 `y+=x` 是判断不出来该选择哪个变量的, 这就是语义分析所需要解决的问题之一, 为这些使用的变量提供额外的语义信息. 这一过程实际上部分对应了符号表的建立.
如果你还实现如闭包等特性, 变量的绑定很可能会变的更加复杂.

在变量的指向确定后, 你同时也可能发现一些语义错误, 例如使用的变量是未定义的(找不到指向), 变量有重复定义, 这时你可以选择向使用者报错.

- 类型推断

另外, 很多静态类型语言允许开发者不显式的指定变量的类型而交由编译器来推断, 例如 `auto` 关键字的使用. 这时你可能需要自行推断变量的类型, 在无法推断或者其他你认为有错误的情况报错. 当然如果你的语言没有这种东西也就无所谓了.

语义检查

语义检查, 顾名思义就是检查语义有没有错误.
如果你实现的是静态类型语言, 那么一个非常重要的检查即为**类型检查(type checking)**.

```
int a() {
  return 100;
}
string b = "hello";
int c = a() / b;
```

例如上述代码, 你需要推断得到 `a()` 返回值是int, 而 `b` 的类型是string, 这两个类型是无法进行除法的, 也就产生了类型错误. 这时你会发现变量绑定是十分重要的, 你需要知道这里使用的变量/函数究竟指向哪一个定义.

除此之外, 函数调用的参数类型, 个数等也都是应该检查的.

代码优化

代码优化是指编译器在生成目标代码的过程中, 通过改变源代码的组织结构或改进生成的目标代码, 以使程序运行更快或占用更少的内存.
我们介绍两个代码优化技术, 其他的留给感兴趣的同学自行了解.

常量传播

程序中如果进行运算的都是常量的话, 我们就知道它们最后赋值传播到变量上的值也就会是常量.

- 简单的常量传播:

```
x = 3;
y = x + 4;
    |
    |
    |
    \\/
x = 3;
y = 7
```

- 不那么简单的常量传播: 如果函数的参数是常量, 函数运行也是常量, 那么函数的返回值也是常量.

```
x = 3;               int add_ten(int x) {
y = add_ten(x);          return x + 10;
    |                }
    |
    |
    \\/
x = 3;
y = 13;
```

死代码消除

在程序中, 不可到达的代码和对程序状态没有影响的代码是我们可以删去的, 这样可以减小生成代码的大小或加快代码运行速度. 例如:

```
int global;
void f ()
{
  int i;
  i = 1;          /* dead store */
  global = 1;     /* dead store */
```

```
  global = 2;
  return;
  global = 3;     /* unreachable */
}
```

显然函数`f`中只有`global = 2`这句话是有用的. 所以我们可以将函数转化为:

```
int global;
void f ()
{
  global = 2;
  return;
}
```

...

我们这里给出的几个例子实际上都是比较简单的, 都是简单的基于变量赋值和运算的优化. 当你能支持的语法越来越多的时候, 优化就变得更加困难. 如果你**很想**对你的程序进行不平凡的优化, 那么就需要对程序进行更深刻的分析. 更多的关于程序分析和优化技术的知识, 可参考Tai-e. 当然这需要很多额外的时间与功夫, 显然以上不是本课程实验的主要目的, 仅供有兴趣的同学实现.

另外值得指出的是, 代码优化是可以在编译器的多个流程进行的, 例如在语法分析阶段根据语法树你就可以对表达式进行简化, 下一节代码生成中更是有许多代码优化技术可以实现.

代码生成

代码生成即在AST或者IR(中间表示)的基础上, 生成你所需要的目标代码.
我们在本节介绍3种在本实验中比较常见的代码生成技术.

利用LLVM基于AST生成

> **不推荐**2-3人小组选择此方案。

LLVM API	目的
`Builder->CreateFAdd(L, R, "addtmp");`	生成左侧和右侧表达式后, 进行浮点数加法.
`Builder->CreateCall(CalleeF, ArgsV, "calltmp");`	函数调用
`Function::Create(FT, ..., Name, TheModule.get());`	生成函数

LLVM 是一套编译器工具链, C/C++, Rust等语言都有使用其作为后端代码生成的编译器. 我们只需要实现前端, 后端的目标代码直接调用LLVM提供的API来完成即可.
所以我们可以采用的一种标准的, 同样相对简单的代码生成方式是利用LLVM生成目标代码, 其实质就是遍历你的AST, 根据节点的类型调用LLVM提供的相应API来生成IR, 我们举几个简单的例子.

具体的生成LLVM IR入门内容可参考LLVM tutorial. 初次接触LLVM必定需要时间和精力来学习(甚至环境配置也不是一件易事), 请多阅读手册和教程(大部分是英文的). 而在生成IR之后, 简单的LLVM API 就能直接帮助我们生成目标代码, 可参考Compiling to Object Code.

利用LLVM基于中间表示生成

> **不推荐**3人小组选择此方案。

如果你用过上文基于LLVM的方法, 你会发现其实LLVM最后会根据我们调用的API生成中间表示IR, 比如下面的例子.

```
@.str = private constant [13 x i8] c"Hello World!\\00", align 1 ;

define i32 @main() ssp {
entry:
 %retval = alloca i32
 %0 = alloca i32
 %"alloca point" = bitcast i32 0 to i32
 %1 = call i32 @puts(i8* getelementptr inbounds ([13 x i8]*@.str, i64 0, i64 0))
 store i32 0, i32* %0, align 4
 %2 = load i32* %0, align 4
 store i32 %2, i32* %retval, align 4
 br label %return
return:
 %retval1 = load i32* %retval
 ret i32 %retval1
}

declare i32 @puts(i8*)
```

这样的中间表示有几个优势:

- 独立于机器
- 通常易于分析(例如代码优化就经常基于中间表示来完成)

- 易于翻译成汇编代码
  所以你其实可以不用第一种方法在AST上直接调用LLVM的API, 而是生成LLVM的IR, 然后再调用LLVM的接口, 也就是我们的第二种途径.

不依赖LLVM生成汇编

> 任何小组都可以选择此方案。

第三种方案是不允许直接或间接地使用LLVM提供的LLVM IR转汇编的功能, 需要自己基于AST或IR生成某种目标架构（推荐RISC-V）的汇编代码/机器码. 你需要根据你生成的汇编选择**汇编模拟器**如venus, BRISCV来检验你的成果. 而对于我们的评测而言, 一种可行的方法是生成汇编然后利用汇编器来帮你生成可执行文件, 最后在QEMU上运行我们的测试代码.
对比编译成中间表示, 生成汇编代码还有许多难点, 你需要面临寄存器分配, 平台相关的特性, 通过系统调用与操作系统交互等等困难. 当然, 收获和困难也是成正比的.

对于Tiger语言，教材每章的程序设计作业以及教材官方网站上提供的框架代码能够帮助你循序渐进地实现一个不依赖LLVM的编译器。

> **另辟蹊径: 如果你有其他想法, 欢迎在与助教和老师交流后实现.**

建议

- 语言的选择: 你可以选择一个主流编程语言的子集(例如C)并加以变化, 可以选择开源社区中流行的语言, 也可以自己从头设计一个新的语言. 如果你不知道该选择什么语言, 可以考虑 SysY. **如果你对语言进行了裁减/变化, 需要在验收时以及实验报告中显式地指出相应的变化.**
- 你可以从对简单语法的支持开始(比如算术表达式和基本的变量赋值), 实现一个完整流程. 然后在测试后迭代, 逐渐往简单的编译器上添加新的功能.
- 我需要经常遍历语法树进行各种操作(类型检查与推断, 生成IR或者生成汇编代码), 而对于每一种类型的节点又要有不同的操作, 有什么好的设计方式(模式)吗? visitor pattern.
- 如果你毫无头绪, 不妨从学长们的项目开始看起. (或者问问chatGPT? 但要小心它的回答:)

评分标准

实验报告10% 实验验收90%.

测试程序	测试内容	测试方法	分值(百分制)
1 判断素数	基本控制流语句与算术	统一测试	10
2 汉诺塔	递归函数定义与调用	统一测试	15
3 矩阵乘法	数组的定义与使用	统一测试	20
4 最长公共子序列	字符串操作	统一测试	20
5 整数四则运算	综合测试	统一测试	25
语法报错	语法分析阶段识别的错误, 如括号不匹配	由小组提供样例	5
语义报错	语义阶段识别的错误, 如使用未定义的变量, 错误的类型	由小组提供样例	5

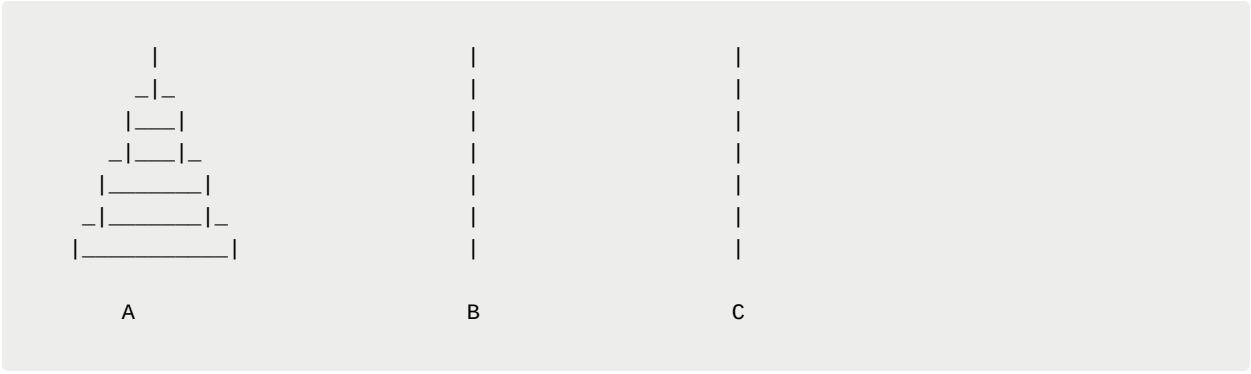
- 测试1 判断素数

一个简单的判断素数的程序. 输入为一个整数, 你只需要对素数输出1, 其他输出0即可.

Input	Output
2	1
3	1
4	0

- 测试2 汉诺塔

汉诺塔问题相信大家已经很熟悉了, 不再赘述.



输入: 盘子的数量n

输出: 将n个盘子从A柱(借助B柱)移动到C柱的最简过程.

```
# input                 # input
3                       1
# output                # output
A->C                    A->C
A->B
C->B
A->C
B->A
B->C
A->C
```

> **该程序必须使用递归完成.**

- 测试3 矩阵乘法

输入:两个矩阵 *A* 和 *B* 。每个矩阵输入的第一行是单个空格隔开的三个正整数M, N, P, 即

A = M x N , B = N x P

接下来2行，第一行MxN个整数, 第二行NxP个整数, 整数之间使用一个空格隔开.

输出: 按照矩阵行列输出

```
# input
4 1 4
123 345 567 789
-123 345 -567 789
# output
-15129 42435 -69741 97047
-42435 119025 -195615 272205
-69741 195615 -321489 447363
-97047 272205 -447363 622521
```

- 测试4 最长公共子序列

最长公共子序列（LCS）是一个经典的字符串处理问题, 它用于找到两个字符串中最长的共同子序列. 想必大家也已经很熟悉了, 我们不再赘述.

输入: 两个字符串, 以空格隔开

输出: 最长子序列的长度

Input	Output

Input	Output
ABCDGH AEDFHR	3
AGGTAB GXTXAYB	4

- 测试5 整数四则运算

在个人作业中我们借助lex和yacc实现了一个简单的计算器, 而现在我们要求你使用你的编程语言和你的编译器完成这一点.

- 需要支持加减乘除和括号, 运算结果为整数
- 整数的表达方式为十进制
- 不需要考虑语法错误的情况, 也不需要支持不合法的表达式
- 其中优先级和结合性与我们平时习惯相同.
- 我们保证输入没有空格, 建议用类似于scanf的函数读入.
- 为了简单起见, 你只需要支持如下文法:

```
exp-> exp + exp
    | exp – exp
    | exp * exp
    | exp / exp
    | NUM
```

输入为一行表达式, 输出为表达式计算的结果.

Input	Output
3+4+57	64
3/2+2	3
2-4-10	-12

> 对于上面5个测试, 你只需要像做程序设计练习题使用`scanf, printf`一样类似的标准输入和输出即可.

- 语法报错与语义报错

该测试的意义在于检查你的语法分析与语义分析实现. 由于语言和分析技术是自定义的, 测试也就自然是自定义的. **由小组提供样例在验收和实验报告中展示.**

评测环境

与个人作业类似, 我们提供了评测程序, 具体见学在浙大相应文件.

我们同样建议你在Linux环境下完成实验.

如果你的环境和实现未被支持, 请及时与助教联系.

操作系统	支持架构
Linux	x86_64(amd64), arm64, riscv64
MacOS	arm64
Windows	x86_64(amd64)

请注意!!!

- 不同于个人实验, 通过以上的测试并不保证你会获得任何分数, 这只是作为一个标准, 说明你可以来验收了.
- 在验收时, 助教会进行一系列操作来确认你的实现是正确的, 包括但不限于:
  - 检查是否能够现场本地编译运行上面的几个固定测试和小组提供的测试.
  - 问一些与源代码相关的问题.
  - 要求你现场使用你的编译器正确编译并执行一些简单程序.
- 如果你们组的成员数目和你们实现的内容不匹配, 可能不能拿到全部的分数.
- 验收时, 所有成员务必到场.

实验报告要求

我们对于实验报告的格式和页数没有要求, 中英文皆可, 但你至少应该包含以下部分:

- 你使用的语言, 工具链和实验环境.
- 你实现的语言的语法.
- 介绍你的编译器是如何将源代码转换为目标代码的. 具体而言, 对于上文提到的编译器编译过程, 逐个介绍你的实现并解释相应的技术决策, 可以重点介绍你认为有趣的部

分.

- 展示你的编译器在验收测试中的表现. 特别是语法报错与语义报错及其他自行准备的内容.
- 小组的分工情况：详细罗列每个成员负责的内容.
- 介绍实验中遇到的问题和挑战与解决问题.

> 实验源码单独打包成一个文件随实验报告上传.

> 你可以使用任何编程语言来实现你的编译器, 但是你需要在报告中明确说明.

Just For Fun

如果你学有余力或者对这门课很感兴趣，在完成上述要求后，你还可以考虑实现下面的一个或多个内容来打造更强大的编译器。**以下内容并非必须的，不影响最终实验分数，Just for fun :)**

- 支持更多编程范式, 例如面向对象(继承, 封装, 多态)或者函数式编程(高阶函数, 柯里化).
- 实现代码优化如过程间的常量传播检测与优化.
- 强大的类型系统与类型推断.
- 支持更多复杂的数据类型, 比如字典, 集合等.
- 手写词法分析和语法分析器
- 错误恢复
- 帮助改善这门课，降低学弟学妹的实验难度：
  - 更丰富的测试用例
  - 中间步骤的测试用例
  - 对实验指导的改进/内容补充
- 虚拟机与解释执行
  在上面的生成IR过程中, 一个自然的想法是: 我们既然都选择生成IR了, 那何必一定要

生成LLVM的IR呢? 你可以自己设计IR, 然后再通过AST生成它. 那下一个问题就是怎么运行这个IR呢? 那就是在虚拟机(这里的虚拟机不是我们配环境的虚拟机)里解释执行, 其代表语言和虚拟机实现就是大家所熟知的Java和JVM, 而这里的中间表示就是Java字节码(bytecode).
对于虚拟机的设计, 一种经典且常用的方法是**stack machine**. 比如对于如下的简单Java代码:

```
public static void main(String[] args) {
    int a = 1;
    int b = 2;
    int c = a + b;
}
```

它生成的字节码的意思也很明显.

```
0: iconst_1  // 生成常量int 1
1: istore_1  // 将栈上int保存到第一个局部变量, 即 a
2: iconst_2  // 同上
3: istore_2  // 同上, 保存到 b
4: iload_1   // 将局部变量 a 放到栈上
5: iload_2   // 同上
6: iadd      // 栈上两个元素相加 a+b
7: istore_3  // 将结果保存到 c
8: return
```

在这里我们将字节码的执行放在一个栈上(stack machine)上进行. 例如 `iadd` 等指令就是直接将栈顶两个数字拿出来, 进行运算然后将结果放回栈顶. 限于篇幅, 我们没有办法介绍关于Java字节码的更多内容如frame等重要概念. 如果你很感兴趣, 可以自行RTFM.

如果你想了解更多的内容, 比如在stack machine上控制流意味着什么, 函数又该如何实现, **尤其是如何实现一个简单的字节码编译器和虚拟机**, 而不是像Java这么庞大的体系, craftinginterpreter 后半部分的内容会很适合你.

如果你实现了一个很强的功能, 比如较完善的面向对象或者利用有趣的程序分析技术进行优化), **非常欢迎**准备好测试用例展示给助教看。

以下行为成绩记零分

- 抄袭(小组作业成绩记为0)

- 无贡献(小组作业成绩记为0）

如果对于以上内容有疑问或者意见, ***尤其是你发现该实验的测试有错误或者程序无法正常工作时***, 请通过钉钉, 邮箱(qizhan@zju.edu.cn)或者线下实验课等方式及时联系我.

祝你好运!