
系统设计文档

The Architecture Design of this Project

Boyi Lee & Mark Young

Linux Programming 终期报告 • 大二 • 2013年6月28日



目录



理论背景	2
完整的编译器前端	2
源语言语法 <i>language grammar</i>	2
<i>statement</i> 产生式:	2
<i>Linux Programming</i> 运用	4
程序设计框架	5
词法分析器	5
符号表和类型	5
表达式的中间代码	5
语句的中间代码	5
语法分析器	5
测试分析	6

理论背景

Theory Background of the Project

完整的编译器前端

项目为一个完整的编译器前端，是基于 Alfred V. Aho, Monica S. Lam, Sethi, Ullman 编写的《Compilers》第二版中 2.5~2.8 节中非正式描述的简单编译器编写的。不同的是，能够为 Bool 表达式生成跳转代码。下面先给出源语言的语法。

这个语言的一个程序由一个块 Block 组成，该块中包含可选的声明 declarations 和语句 statements. 语法符号 basic 表示类型。

源语言语法 language grammar

```
program  →  block
block    →  { decls stmts }
decls    →  decls decl | ε
decl     →  type id ;
type     →  type [ num ] | basic
stmts    →  stmts stmt | ε
```

statement 产生式：

```
stmt     →  loc = bool ;
          |  if ( bool ) stmt
          |  if ( bool ) stmt else stmt
          |  while ( bool ) stmt
          |  do stmt while ( bool );
          |  break ;
          |  block
loc       →  loc [ bool ] | id
```

表达式的产生式处理了运算符的结合性和优先级。对每个优先级级别都使用一个非终结符号。

```
bool  →  bool || join | join
join  →  join && equality | equality
equality → equality == rel | equality != rel | rel
rel    →  expr < expr | expr <= expr | expr >= expr | expr > expr |
          expr
expr   →  expr + term | expr - term | term
term   →  term * unary | term / unary | unary
unary  →  ! unary | - unary | factor
factor →  (bool) | loc | num | real | true | false
```

根据源语言的语法，结合编译原理的相关知识，利用Linux编程环境，实现一个可同时编译多个源文件的编译器前端，并且，它有友好的界面。

结合编译原理，构建如下程序框架：



Linux Programming 运用

文件I/O

整个系统的输入及输出设计文件操作，其中有源程序的读取以及中间代码的生成结果。

多进程

采用多进程在用户编译等待的同时终端有geek风格动画的呈现，采用fork实现，父进程进行动画输出，子进程进行多线程编译工作。同时对僵尸进程进行了处理，测试过程中确保没有遗留僵尸进程的发生。

多线程

鉴于Linux本身没有提供较好地多线程接口，采用POSIX标准下的多线程机制，目的在于同时进行多个文件的同步编译，提高系统的吞吐量。因为不同文件编译时间是不定的，所以在线程创建之后进行了阻塞，直至最后一个线程执行完毕。

信号

子进程的执行中分为多线程，

程序设计框架

Architecture of Programming Design

结合编译器结构图，从 `main` 函数入口创建词法分析器 `Lexer` 与一个语法分析器 `Parser`，然后调用语法分析器 `Parser` 中的方法 `program`。

词法分析器

主要方法为 `scan()` 扫描函数，返回符号单元 `Token` 类对象。符号表用 `map` 实现，创建 `Lexer` 对象的同时用 `map` 保留关键字，如：`int`, `true`, `break`, `while`, `if`, `else`...在 `scan()` 函数中，`switch` 分支语句和 `if` 判断语句简单地实现了确定有限自动机。

符号表和类型

类 `Env` 把字符串单元映射为类 `Id` 的对象。把类 `Type` 定义为类 `Word` 的子类，因为像 `int` 这样的基本类型名字就是保留字，词法分析器将其映射为适当的对象。对应于基本类型的对象是 `Type::int`, `Type::float`, `Type::char`, `Type::bool`。这些对象从基类中继承了 `tag`，值为 `Tag::Basic`。静态函数 `numeric & max` 可用于类型转换。编译器支持数组，因此在计算数组元素地址时不可避免要用到类型宽度。

表达式的中间代码

语法树的节点类的基类为 `Node`，`Node` 共两个子类，分别是表达式节点 `Expr` 和语句节点 `Stmt` 类。在多线程同时编译多个源文件时，我们作了输出文件名到 `label` 值以及代码行号 `lexline` 的映射，以便输出和报错。`gen` 函数生成项，`reduce` 规约。

语句的中间代码

每个语句都由一个类实现，对应字段为子类的对象。如：`While` 对应于测试表达式的字段和一个子语句字段。`gen` 确定语句的开始和结束 `label`。

语法分析器

语法分析器读入一个由词法单元组成的流，并调用合适的构造函数，构建一颗抽象语法树。对符号表的处理采用向前链表，其实也可以用栈实现。

测试分析

Test & Analysis

见测试文档