

浙江大学

本科实验报告

课程名称：编译系统设计

姓 名：张瑞祥

学 院：计算机科学与技术

系：计算机科学与技术

专 业：计算机科学与技术

学 号：3120104198

指导教师：李莹

2015 年 7 月 2 日

一 概述

1.1 Pascal JIT Compiler

按照本次编译系统设计课程要求,我实现了基于给定文法的Pascal语言的编译器.
主要的技术栈如下所示:

编译器采用python实现

语法分析 ply.lex

语义分析 ply.yacc

中间代码 llvm IR code

代码生成 llvm

运行环境 Linux Windows MacOS

本编译支持的主要功能如下所示:

课程要求的主要的Pascal语法功能,如函数嵌套定义,递归调用,数组,枚举等等

错误恢复 可以在语法分析,语义分析阶段检查并报告多个错误

代码优化 支持用户可选择的代码优化等级,采用了常量传播,不变式外提,常量叠加,dead code elimination等等

JIT支持 除了生成本地的可执行二进制文件之外,默认按照JIT方式执行代码,利用了llvm原生支持的JIT优化

1.2 分工说明

因为组内只有我一个人,因此各部分都是自己独立完成

二 词法分析

词法分析部分，因为我选择用python来写本编译器，因此相应的就选择了 ply 来做词法分析的工具。

2.1 PLY简介

PLY是一个利用纯python实现传统编译器前端工具lex和yacc的工具，在用法和功能上保持本质上的不变。PLY同样采用LALR(1) parsing来作为parse的算法，同时提供了输入检查，错误报告等功能。

PLY包括两个独立的module,分别是lex.py和yacc.py

2.2 词法分析模块

词法分析代码见打包上传的代码文件夹内的 frontend/lex_pas.py

词法分析的主要目的是讲输入的字符串转化为token的流。比如输入是

```
x = 3 + 42 * (s - t)
```

的情况下，输入的token应该是

```
'x','=', '3', '+', '42', '*', '(', 's', '-', 't', ')'
```

根据ply.lex的语法要求，为每个token用正则表达式进行定义和捕获，并且在需要捕获之后进行一些处理的情况下，添加用 @TOKEN 修饰符进行修饰的方法即可。

下面的代码片段是捕获一个 8 进制数字常量的处理函数:

```
oct=r"[0-7]"
@TOKEN(r'0'+oct+r'*)
def t_cINTEGER_8(t):
    t.value=int(t.value,8)
    t.type='cINTEGER'
    return t
```

三 语义分析

3.1 ply.yacc简介

ply.yacc是ply工具共用来进行parse的模块.

给定CFG之后,为每一个产生式写相应的处理函数,就可以在parsing的过程中建立AST

3.2 语义分析模块

代码见代码文件夹内的 frontend/yacc_pas.py

本模块的主要功能是在有了词法分析得到的 token() 方法之后,建立一颗完整的AST

下面的代码片段说明了语义分析的主要工作的例子:

二元表达式节点的创建

```
# expr : expr PLUS term | expr MINUS term | expr OR term |
term
def p_Expr(p):
    '''expr : expr oPLUS term
           | expr oMINUS term
           | expr kOR term
           | term'''
    if len(p)==4:
        p[0]=create_stmt_node('bin_expr',p[2],p[1],p[3])
    else:
        p[0]=p[1]
```

同时可以通过ply.yacc的precedence来特别指定一些运算符的优先级和结合性.

```
precedence=(
    ('left','oPLUS','oMINUS'),
    ('left','oMUL','oDIV','kDIV','kMOD')
)
```

在语义分析结束之后,生成相应的AST.如下所示

```
def p_Start(p):
    'program : program_head routine oDOT'
    p[0]=ProgramNode(p[1],p[2])
```

三 抽象语法树

AST的代码见 frontend/AST.py

AST中各节点的设计详见代码文档 doc/ast/index.html

AST部分采用OOP的设计,所有的节点都继承自 ASTNode .

```
class ASTNode(object):
    def __init__(self):
        pass

    def codegen(self):
        pass

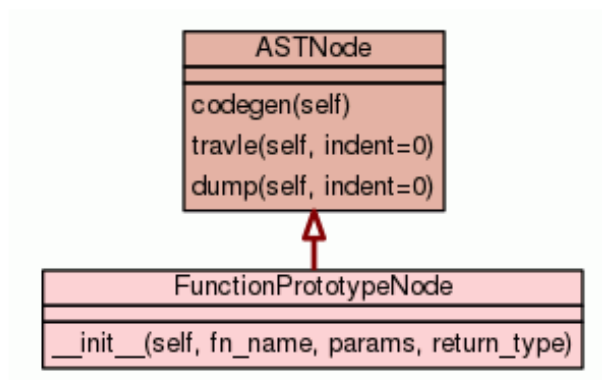
    def travle(self, indent=0):
        print self.dump(indent)
        attrs=gattrs(self)
        if isinstance(self, ListNode):
            for o in self.node_list:
                if isinstance(o, ASTNode):
                    o.travle(indent+2)
        else:
            for o in attrs:
                v=getattr(self,o)
                if v and isinstance(v,ASTNode):
                    v.travle(indent+2)

    def dump(self, indent=0):
        return '{0}{1}'.format(
            ' ' * indent, self.__class__.__name__)
```

这里简单以几个节点为例说明设计

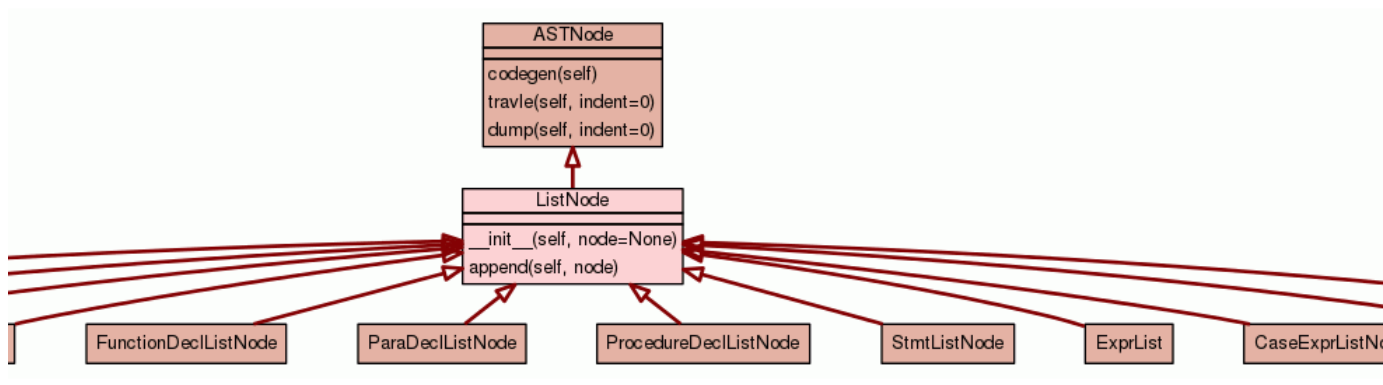
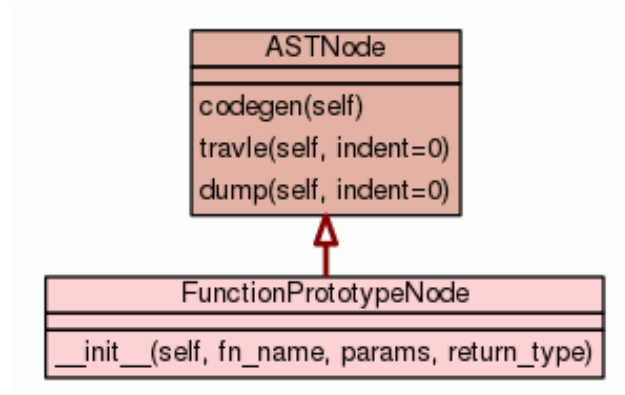
Class FunctionPrototypeNode

[source code](#)



Class *FunctionPrototypeNode*

[source code](#)



四 中间代码生成

代码生成模块的代码详见 `backend/codegen.py` .

4.1 llvm简介

LLVM提供了完整编译系统的中间层，它会将中间语言（Intermediate form，IF）从编译器取出与最优化，最优化后的IF接着被转换及链结到目标平台的汇编语言。LLVM可以接受来自GCC工具链所编译的IF，包含它底下现存的编译器。

LLVM也可以在编译时期、链结时期，甚至是运行时期产生可重新定位的代码（Relocatable Code）。

LLVM支持与语言无关的指令集架构及类型系统。每个在静态单赋值形式（SSA）的指令集代表着，每个变数（被称为具有类型的寄存器）仅被赋值一次，这简化了变数间相依性的分析。LLVM允许代码被静态的编译，包含在传统的GCC系统底下，或是类似JAVA等后期编译才将IF编译成机器码所使用的实时编译（JIT）技术。它的类型系统包含基本类型（整数或是浮点数）及五个复合类型（指针、数组、矢量、结构及函数），在LLVM具体语言的类型建制可以以结合基本类型来表示，

举例来说，C++所使用的class可以被表示为结构、函数及函数指针的数组所组成。

4.2 llvmlite简介

项目主页见[llvmlite](#)

llvmlite是用python对llvm进行了一层包装的API库。

llvmlite库非常年轻,是最近一年才开始形成和发展的,因此现在网络上用llvmlite开发的编译器除了numba为python做的jit之后很少见,资料也比较少.因此本次编译器的开发是在阅读llvmlite的源码的同时进行开发的.

4.3 符号表设计

符号表代码在 `backend/codegen.py` .

符号表代码文档在 `doc/symbletable/index.html` .

符号表的设计采用了 Hash + Stack 的方式进行实现,同时设计了一个作用域的计数器,每当进入一个新的作用域的时候,计数器加一,然后将所有的变量,类型`,常量等声明进行变量的allocate,然后将类型,地址等信息存到符号表中. 当一个作用域失效的时候,根据其对应的作用域计数从符号表中删除.

SymbleTable
<pre><code>__init__(self) add_var(self, var_name, addr, scope_id, var_type=None) add_fn(self, fn_name, fn_block, scope_id) add_type(self, name, type_def, scope_id) fetch_var_addr(self, var_name) fetch_var_addr_type(self, var_name) fetch_fn_block(self, fn_name) fetch_type(self, type_name) remove_var(self, var_name) remove_scope(self, scope_id)</code></pre>

4.4 代码生成模块

采用Visitor Modo进行代码生成,为了将AST和代码生成的代码完全分离开来.

`_codegen()` 方法以一个AST的子类的作为参数,来做这个节点的代码生成工作.

下面以若干个典型的程序结构进行代码生成模块设计的说明

赋值操作:

```
#allocate address and add to symble table
def _codegen_new_var(self,var,var_type,builder):
    with builder.goto_entry_block():
        tp=var_type if isinstance(var_type,ir.Type) else
self._helper_get_type(var_type)
        addr=builder.alloca(tp,name=var)
        self.sym_table.add_var(var,addr,CodeGenerator.scope_cnt,tp)
    return addr
```

用 IRBuilder 类的 alloca 方法为变量申请相应的内存,然后进行绑定,并更新符号表

数组成员

```
def _codegen_ArrayMemberNode(self,node,builder):
    var_addr,var_type=self.sym_table.fetch_var_addr_type(node.var_name)
    # index=node.indices[0].value
    index=[]
    for i in node.indices:
        if isinstance(i,VariableNode):
            val=self._codegen(i,builder)
        else:
            val=builder.constant(ir.IntType(32),i.value)
        index.append(val)
    indices=index
    # indices=
[builder.constant(ir.IntType(32),index),builder.constant(ir.IntType(32),
0)]
    indices.append(builder.constant(ir.IntType(32),0))
    addr=builder.gep(var_addr,indices)
    return builder.load(addr,"arraymember")
```

如上代码中,从AST节点中拿到index的所有信息,然后利用 gep 方法,得到内存中的地址,然后用 load 操作得到数组变量的值.

函数调用

```
def __codegen_FunctionDecl(self,node,builder,gen_type):
    CodeGenerator.scope_cnt+=1
    fn_params_name,fn_params_type=[],[]
    fn_return_type_l=[]
```



```

proto=self._codegen_FunctionPrototype(node.prototype,builder,fn_params_
type,fn_params_name,fn_return_type_l,gen_type)
    fn_name=node.prototype.name

    self.sym_table.add_fn(fn_name,proto,CodeGenerator.scope_cnt)

    bb_entry=proto.append_basic_block('entry')
    newBuilder=ir.IRBuilder(bb_entry)
    for i,arg in enumerate(proto.args):
        arg.name=fn_params_name[i]
        addr=self._codegen_new_var(arg.name,fn_params_type[i],newBuilder)
        newBuilder.store(arg,addr)

    fn_return_type=fn_return_type_l[0]
    return_addr=self._codegen_new_var(fn_name,fn_return_type,newBuilder)

    res=self._codegen(node.body,newBuilder)
    if gen_type=='fun':
        return_val=newBuilder.load(return_addr,fn_name)
        newBuilder.ret(return_val)
    else:
        newBuilder.ret_void()

    # for i,arg in enumerate(proto.args):
    #     self.sym_table.remove_var(arg.name)
    # self.sym_table.remove_var(fn_name)
    self.sym_table.remove_scope(CodeGenerator.scope_cnt)

    CodeGenerator.scope_cnt-=1
    return proto

def
__codegen_FunctionPrototype(self,proto,builder,fn_params_type,fn_params_
name,fn_return_type_l,gen_type):
    fn_name=proto.name
    fn_return_type=self._helper_get_type(proto.return_type.type_name if
gen_type == 'fun' else 'void')
    if proto.params:
        for para_type_list in proto.params.node_list:
            tp=self._helper_get_type(para_type_list.type.type_name)
            fn_params_type+=[tp]*len(para_type_list.var_list.node_list)
            fn_params_name+=para_type_list.var_list.node_list

    fn_type=ir.FunctionType(fn_return_type,fn_params_type)
    func=ir.Function(self.module,fn_type,fn_name)
    fn_return_type_l.append(fn_return_type)
    return func

scope_cnt=0

```

```

def _codegen_FunctionDeclNode(self,node,builder):
    return self._codegen_FunctionDecl(node,builder,'fun')

def _codegen_ProcedureDeclNode(self,node,builder):
    return self._codegen_FunctionDecl(node,builder,'pro')

```

上述代码生成函数的定义的代码, 首先根据函数原型, 做参数的空间申请和赋值工作,然后根据函数body内的代码进行代码生成工作.

二元表达式

```

# = >= > <= < = + - or * div mod and
def _codegen_BinaryExpr(self,node,builder):
    if node.op == ':=':
        if isinstance(node.lhs,ArrayMemberNode):
            lhs_val=self._codegen_ArrayMemberLhsNode(node.lhs,builder)
            rhs_val=self._codegen(node.rhs,builder)
            self._codegen_do_assign(lhs_val,rhs_val,builder)
            return rhs_val
        else:
            assert(isinstance(node.lhs,VariableNode))
            var_addr=self.sym_table.fetch_var_addr(node.lhs.name)
            rhs_val=self._codegen(node.rhs,builder)
            # if isinstance(node.rhs,ArrayMemberNode):
            #     rhs_val=builder.load(rhs_val,"array_member_value")
            self._codegen_do_assign(var_addr,rhs_val,builder)
            return rhs_val

    lhs=self._codegen(node.lhs,builder)
    rhs=self._codegen(node.rhs,builder)
    lhs_type=self._helper_parse_type_to_id(node.lhs)
    rhs_type=self._helper_parse_type_to_id(node.rhs)
    if lhs_type>rhs_type:
        self._codegen_type_cast(rhs,lhs_type,builder)
    elif lhs_type<rhs_type:
        self._codegen_type_cast(lhs,rhs_type,builder)
    node.type=max(lhs_type,rhs_type)

    op=node.op
    op=='=' if op == '=' else op
    if op in ('==','>=','>','<=','<'):
        if node.type == _Type_Float:
            return builder.fcmp_ordered(op,lhs,rhs)
        else:
            return builder.icmp_signed(op,lhs,rhs)
    elif op == '+':

```

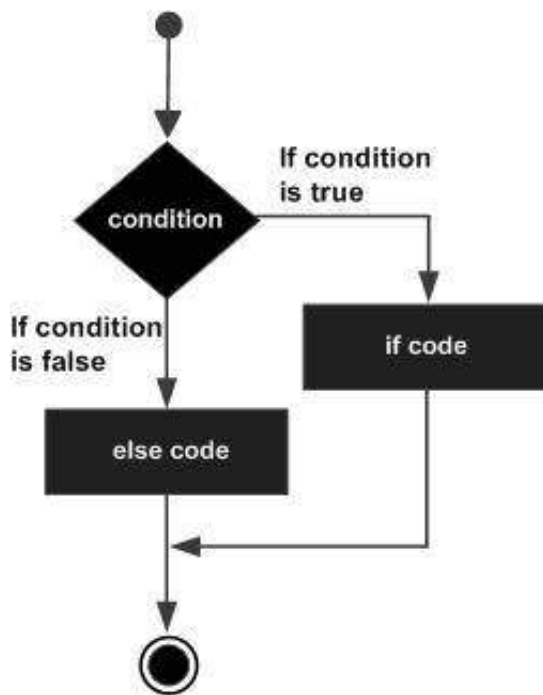
```

    if node.type == _Type_Float:
        return builder.fadd(lhs,rhs,name='float_add')
    else:
        return builder.add(lhs,rhs,name='integer_add')
elif op == '-':
    if node.type == _Type_Float:
        return builder.fsub(lhs,rhs,name='float_sub')
    else:
        return builder.sub(lhs,rhs,name='integer_sub')
elif op == 'or':
    if node.type == _Type_Float:
        raise CodegenError('Floating-point number or operation')
    else:
        return builder.or_(lhs,rhs,name='integer_or')
elif op in ('*', 'mul'):
    if node.type == _Type_Float:
        return builder.fmul(lhs,rhs,name='float_mul')
    else:
        return builder.mul(lhs,rhs,name='integer_mul')
elif op == 'div':
    if node.type == _Type_Float:
        return builder.fdiv(lhs,rhs,name='float_div')
    else:
        return builder.sdiv(lhs,rhs,name='integer_div')
elif op == 'mod':
    if node.type == _Type_Float:
        return builder.frem(lhs,rhs,name='float_mod')
    else:
        return builder.srem(lhs,rhs,name='integer_mod')
elif op == 'and':
    if node.type == _Type_Float:
        raise CodegenError('Floating-point number or operation')
    else:
        return builder.and_(lhs,rhs,name='integer_and')
else:
    raise CodegenError('invalid bin_expr operator')

```

上面的代码进行二元表达式的工作, 特殊处理了赋值表达式

if表达式



```

def _codegen_IfExpr(self,node,builder):
    cond_val=self._codegen(node.cond_expr,builder)
    cond_expr_type=self._helper_parse_type_to_id(node.cond_expr.type)

    cmp_res=builder.icmp_signed('!','=',cond_val,builder.constant(ir.IntType(1),0))

    then_bb=builder.function.append_basic_block('then_bb')
    else_bb=builder.function.append_basic_block('else_bb')
    merge_bb=builder.function.append_basic_block('merge_bb')
    builder.cbranch(cmp_res,then_bb,else_bb)

    builder.position_at_start(then_bb)
    then_val=self._codegen(node.then_expr,builder)
    builder.branch(merge_bb)

    then_bb=builder.block
    builder.position_at_start(else_bb)
    else_val=self._codegen(node.else_expr,builder)
    else_bb=builder.block
    builder.branch(merge_bb)

    builder.position_at_start(merge_bb)
  
```

上面的代码首先判断条件的解析代码,然后根据是否等于0来进行相应的跳转 then_bb , else_bb , merge_bb

五 错误恢复

本编译器的错误恢复操作主要集中在词法分析和语义分析上.

通过修改CFG, 添加 error 项来进行错误的判断, 并进行相应的处理.

举例如下所示

在赋值表达式中, 正确的语法应该是

```
def p_Assign_stmt_1(p):
    'assign_stmt : yNAME oASSIGN expression'
    # print '$$$$ ', p.lineno(1), p.lexpos(1)
    var=VariableNode(p[1])
    p[0]=create_stmt_node(':=', var, p[3])
```

通过添加 error 项, 进行如下所示错误的判断:

```
def p_Assign_stmt_1_error(p):
    'assign_stmt : yNAME error expression'
    process_syntax_error(p, 2, ':=')
    p[0]=None
```

如上所示通过添加生成式 assign_stmt : yNAME error expression, 就可以在中间的符号 := 出现错误的时候进行此类型的错误处理.

同时yacc在parse的时候遇到了第一个错误的时候, 不会立即停止报告错误, 而是通过上面的方式恢复出来然后继续进行parse, 尽可能的找出来更多的错误.

六 代码优化

本编译器支持用户指定程度的代码优化工作, 主要利用了llvm的代码优化, 并且支持自定义的 llvm pass 进行优化工作.

下面以常量折叠和dead code elimination 来举例说明代码优化工作

对于下面所示的Pascal代码,

```
program test_1;

var i,a,b: integer;

begin
    a:=3;
```

```
b:=1+2+a;
end.
```

在不加任何的优化的情况下,本编译器生成的IR code如下所示

```
define void @"codegen_global_func"()
{
codegen_global_block:
  %"i" = alloca i32
  %"a" = alloca i32
  %"b" = alloca i32
  store i32 3, i32* %"a"
  %"integer_add" = add i32 1, 2
  %"a.1" = load i32* %"a"
  %"integer_add.1" = add i32 %"integer_add", %"a.1"
  store i32 %"integer_add.1", i32* %"b"
  tail call void @writeln(i32 b)
  ret void
}
```

而在加了优化之后,做了常量折叠操作之后生成的中间代码如下所示

```
define void @codegen_global_func() {
codegen_global_block:
  tail call void @writeln(i32 6)
  ret void
}
```

可以看到在编译时刻就能确定的常量的情况下全部做了优化操作,同时把在之后没有用的dead code `a:=3` 删除了.

七 程序演示

下面的截图演示了本编译器的主要功能,与验收时的展示相对应

本编译器的执行文件为 `ppc`,意思是 Python Pascal Compiler

7.1 编译器用法说明

首先是程序用法说明:

```
sodabeta@ayaya: ~/CS/CC/prj
File Edit View Search Terminal Help
(p2ll)→ prj ./ppc
usage: ppc [-h] [-O OPTIMIZATION_LEVEL] [-i IR_FILE] [-m MACHINECODE_FILE]
        input_file
ppc: error: too few arguments
(p2ll)→ prj ./ppc -h
usage: ppc [-h] [-O OPTIMIZATION_LEVEL] [-i IR_FILE] [-m MACHINECODE_FILE]
        input_file

positional arguments:
  input_file            specify the input file

optional arguments:
  -h, --help            show this help message and exit
  -O OPTIMIZATION_LEVEL, --optimization_level OPTIMIZATION_LEVEL
                        specify the optimization level
  -i IR_FILE, --ir_file IR_FILE
                        generate ir code
  -m MACHINECODE_FILE, --machinecode_file MACHINECODE_FILE
                        generate machine code
(p2ll)→ prj □
```

7.2 测试1

第一个测试是递归计算fibonacci数字, 测试了函数定义, 函数递归, for循环, if表达式, 数组, 输出功能

测试程序位`test/test_fib.pas`

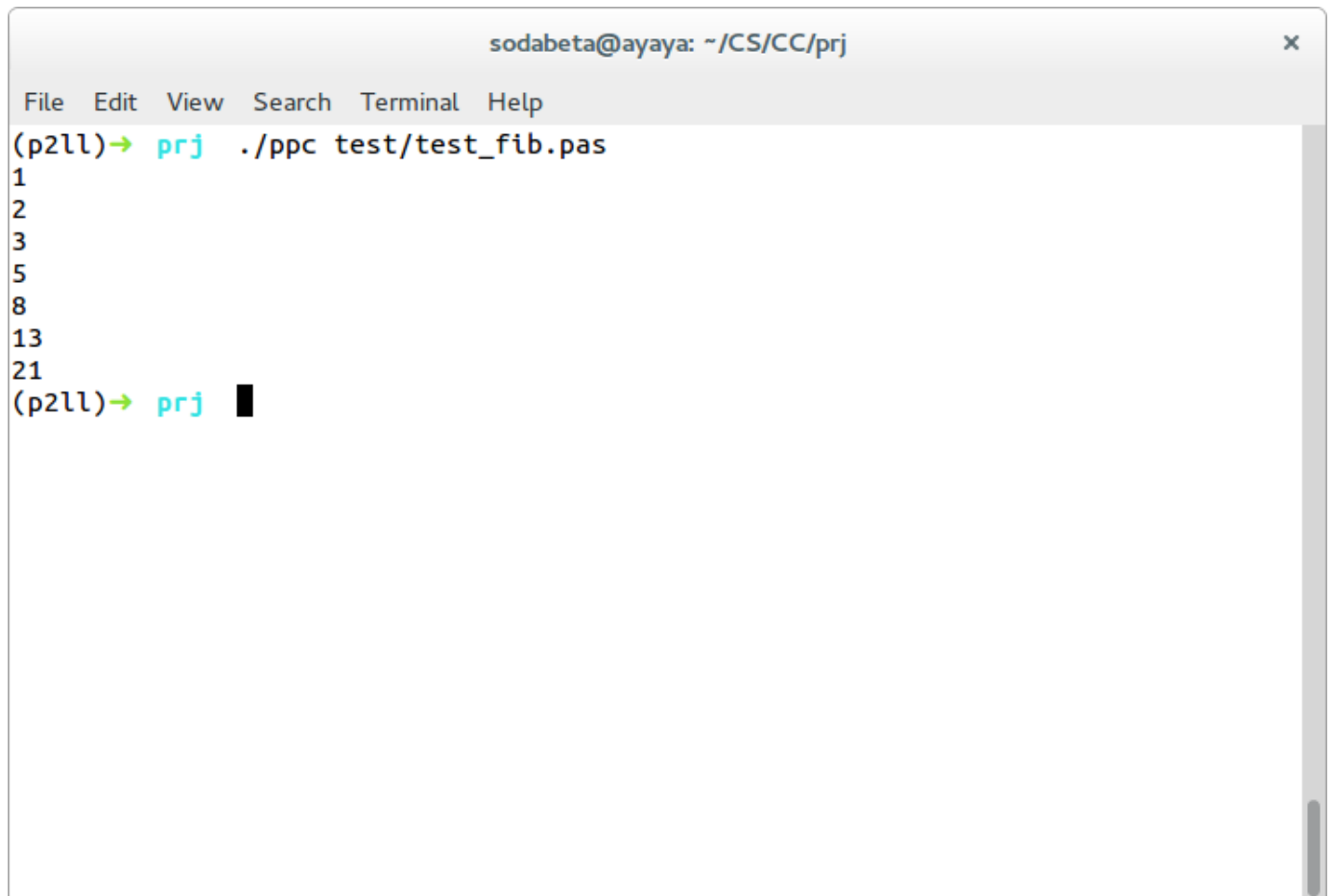
```
program test_1;
type
  arr = array [1..50] of integer;
var i,a: integer;
    f:arr;

function fib(x:integer):integer;
begin
  if ((x = 0) or (x = 1)) then
    fib:=1
  else
    fib:=fib(x - 2) + fib(x - 1);
end;

begin
```

```
for i:=1 to 7 do begin
    f[i]:=fib(i);
end;
for i:=1 to 7 do begin
    writeln(f[i]);
end;
end.
```

程序运行如下所示



The screenshot shows a terminal window titled "sodabeta@ayaya: ~/CS/CC/prj". The terminal has a menu bar with "File", "Edit", "View", "Search", "Terminal", and "Help". The command prompt is "(p2ll)→ prj". The user has entered the command "./ppc test/test_fib.pas". The output of the program is displayed on the next lines: "1", "2", "3", "5", "8", "13", and "21". The prompt "(p2ll)→ prj" is followed by a black cursor block.

```
sodabeta@ayaya: ~/CS/CC/prj
File Edit View Search Terminal Help
(p2ll)→ prj ./ppc test/test_fib.pas
1
2
3
5
8
13
21
(p2ll)→ prj █
```

要生成中间代码, 加上 -i 操作即可, 如下所示


```
sodabeta@ayaya: ~/CS/CC/prj
File Edit View Search Terminal Help
(p2ll)→ prj ./ppc -i test/fib_icode test/test_fib.pas
1
2
3
5
8
13
21
(p2ll)→ prj ls test/
fib_icode test4.pas test_fib.pas tp
test1.pas test_error.pas test_scope.pas
(p2ll)→ prj █
```

中间代码如下所示

```
; ModuleID = "Test"
target triple = "unknown-unknown-unknown"
target datalayout = ""

define void @"writeln"(i32 %"x")
{
entry:
    %"f" = inttoptr i64 139761332760592 to void (i32)*
    call void (i32)* %"f"(i32 %"x")
    ret void
}

define i32 @"readln"(i32 %"x")
{
entry:
    %"f" = inttoptr i64 139761332760648 to i32 (i32)*
    %".2" = call i32 (i32)* %"f"(i32 %"x")
    ret i32 %".2"
}

define void @"codegen_global_func"()
{
```

```

codegen_global_block:
    %"i" = alloca i32
    %"a" = alloca i32
    %"f" = alloca [50 x i32]
    %"i.1" = alloca i32
    store i32 1, i32* %"i.1"
    %"i.2" = load i32* %"i.1"
    %".2" = icmp sle i32 %"i.2", 7
    %"i.6" = alloca i32
    br i1 %".2", label %"loop", label %"after_bb"
loop:
    %"i.3" = load i32* %"i.1"
    %".4" = getelementptr [50 x i32]* %"f", i32 %"i.3", i32 0
    %"i.4" = load i32* %"i.1"
    %"call_fn" = call i32 (i32)* @"fib"(i32 %"i.4")
    store i32 %"call_fn", i32* %".4"
    %"i.5" = load i32* %"i.1"
    %".6" = add i32 %"i.5", 1
    store i32 %".6", i32* %"i.1"
    %".8" = icmp sle i32 %".6", 7
    br i1 %".8", label %"loop", label %"after_bb"
after_bb:
    store i32 1, i32* %"i.6"
    %"i.7" = load i32* %"i.6"
    %".11" = icmp sle i32 %"i.7", 7
    br i1 %".11", label %"loop.1", label %"after_bb.1"
loop.1:
    %"i.8" = load i32* %"i.6"
    %".13" = getelementptr [50 x i32]* %"f", i32 %"i.8", i32 0
    %"arraymember" = load i32* %".13"
    call void (i32)* @"writeln"(i32 %"arraymember")
    %"i.9" = load i32* %"i.6"
    %".14" = add i32 %"i.9", 1
    store i32 %".14", i32* %"i.6"
    %".16" = icmp sle i32 %".14", 7
    br i1 %".16", label %"loop.1", label %"after_bb.1"
after_bb.1:
    ret void
}

define i32 @"fib"(i32 %"x")
{
entry:
    %"x.1" = alloca i32
    store i32 %"x", i32* %"x.1"
    %"fib.1" = alloca i32
    %"x.2" = load i32* %"x.1"
    %".3" = icmp eq i32 %"x.2", 0
    %"x.3" = load i32* %"x.1"

```

```

%"x.4" = icmp eq i32 %"x.3", 1
%"integer_or" = or i1 %"x.3", %"x.4"
%"x.5" = icmp ne i1 %"integer_or", 0
br i1 %"x.5", label %"then_bb", label %"else_bb"
then_bb:
    store i32 1, i32* %"fib.1"
    br label %"merge_bb"
else_bb:
    %"x.4" = load i32* %"x.1"
    %"integer_sub" = sub i32 %"x.4", 2
    %"call_fn" = call i32 @fib(i32 %"integer_sub")
    %"x.5" = load i32* %"x.1"
    %"integer_sub.1" = sub i32 %"x.5", 1
    %"call_fn.1" = call i32 @fib(i32 %"integer_sub.1")
    %"integer_add" = add i32 %"call_fn", %"call_fn.1"
    store i32 %"integer_add", i32* %"fib.1"
    br label %"merge_bb"
merge_bb:
    %"fib.2" = load i32* %"fib.1"
    ret i32 %"fib.2"
}

```

要生成本地的汇编代码, 加上 -m 参数即可. 如下所示

The screenshot shows a terminal window titled "sodabeta@ayaya: ~/CS/CC/prj". The terminal contains the following commands and output:

```

File Edit View Search Terminal Help
(p2ll)→ prj ./ppc -m fib_machinecode test/test_fib.pas
1
2
3
5
8
13
21
(p2ll)→ prj 

```

汇编代码如下所示

```

    .text
    .file    "<string>"
    .globl   writeln
    .align   16, 0x90
    .type    writeln,@function
writeln:
    .cfi_startproc
    pushq    %rax
.Ltmp0:
    .cfi_def_cfa_offset 16
    movabsq  $139746104143888, %rax
    callq    *%rax
    popq     %rax
    retq
.Ltmp1:
    .size    writeln, .Ltmp1-writeln
    .cfi_endproc

    .globl   readln
    .align   16, 0x90
    .type    readln,@function
readln:
    .cfi_startproc
    pushq    %rax
.Ltmp2:
    .cfi_def_cfa_offset 16
    movabsq  $139746104143944, %rax
    callq    *%rax
    popq     %rdx
    retq
.Ltmp3:
    .size    readln, .Ltmp3-readln
    .cfi_endproc

    .globl   codegen_global_func
    .align   16, 0x90
    .type    codegen_global_func,@function
codegen_global_func:
    .cfi_startproc
    pushq    %r14
.Ltmp4:
    .cfi_def_cfa_offset 16
    pushq    %rbx
.Ltmp5:
    .cfi_def_cfa_offset 24
    subq     $216, %rsp
.Ltmp6:
    .cfi_def_cfa_offset 240

```

```

.Ltmp7:
    .cfi_offset %rbx, -24
.Ltmp8:
    .cfi_offset %r14, -16
    movl    $1, 4(%rsp)
    xorl    %eax, %eax
    testb   %al, %al
    jne     .LBB2_3
    movabsq $fib, %r14
    .align  16, 0x90
.LBB2_2:
    movslq  4(%rsp), %rdi
    imulq   $200, %rdi, %rbx
    callq   *%r14
    movl    %eax, 8(%rsp,%rbx)
    movl    4(%rsp), %eax
    incl    %eax
    movl    %eax, 4(%rsp)
    cmpl    $8, %eax
    jl      .LBB2_2
.LBB2_3:
    movl    $1, (%rsp)
    xorl    %eax, %eax
    testb   %al, %al
    jne     .LBB2_6
    movabsq $writeln, %rbx
    .align  16, 0x90
.LBB2_5:
    movslq  (%rsp), %rax
    imulq   $200, %rax
    movl    8(%rsp,%rax), %edi
    callq   *%rbx
    movl    (%rsp), %eax
    incl    %eax
    movl    %eax, (%rsp)
    cmpl    $8, %eax
    jl      .LBB2_5
.LBB2_6:
    addq    $216, %rsp
    popq    %rbx
    popq    %r14
    retq
.Ltmp9:
    .size    codegen_global_func, .Ltmp9-codegen_global_func
    .cfi_endproc

    .globl   fib
    .align  16, 0x90
    .type    fib,@function

```

```

fib:
    .cfi_startproc
    pushq    %r14
.Ltmp10:
    .cfi_def_cfa_offset 16
    pushq    %rbx
.Ltmp11:
    .cfi_def_cfa_offset 24
    pushq    %rax
.Ltmp12:
    .cfi_def_cfa_offset 32
.Ltmp13:
    .cfi_offset %rbx, -24
.Ltmp14:
    .cfi_offset %r14, -16
    movl     %edi, 4(%rsp)
    testl    %edi, %edi
    sete     %al
    cmpl     $1, %edi
    sete     %cl
    orb %al, %cl
    movzbl   %cl, %eax
    cmpl     $1, %eax
    jne .LBB3_2
    movl     $1, (%rsp)
    jmp .LBB3_3
.LBB3_2:
    movl     4(%rsp), %edi
    addl     $-2, %edi
    movabsq  $fib, %r14
    callq    *%r14
    movl     %eax, %ebx
    movl     4(%rsp), %edi
    decl     %edi
    callq    *%r14
    addl     %ebx, %eax
    movl     %eax, (%rsp)
.LBB3_3:
    movl     (%rsp), %eax
    addq     $8, %rsp
    popq     %rbx
    popq     %r14
    retq
.Ltmp15:
    .size    fib, .Ltmp15-fib
    .cfi_endproc

.section     ".note.GNU-stack","",@progbits

```

7.3 测试2

本测试用意是测试函数的递归定义和调用,测试代码在 test/test_scope.pas .

测试代码如下所示

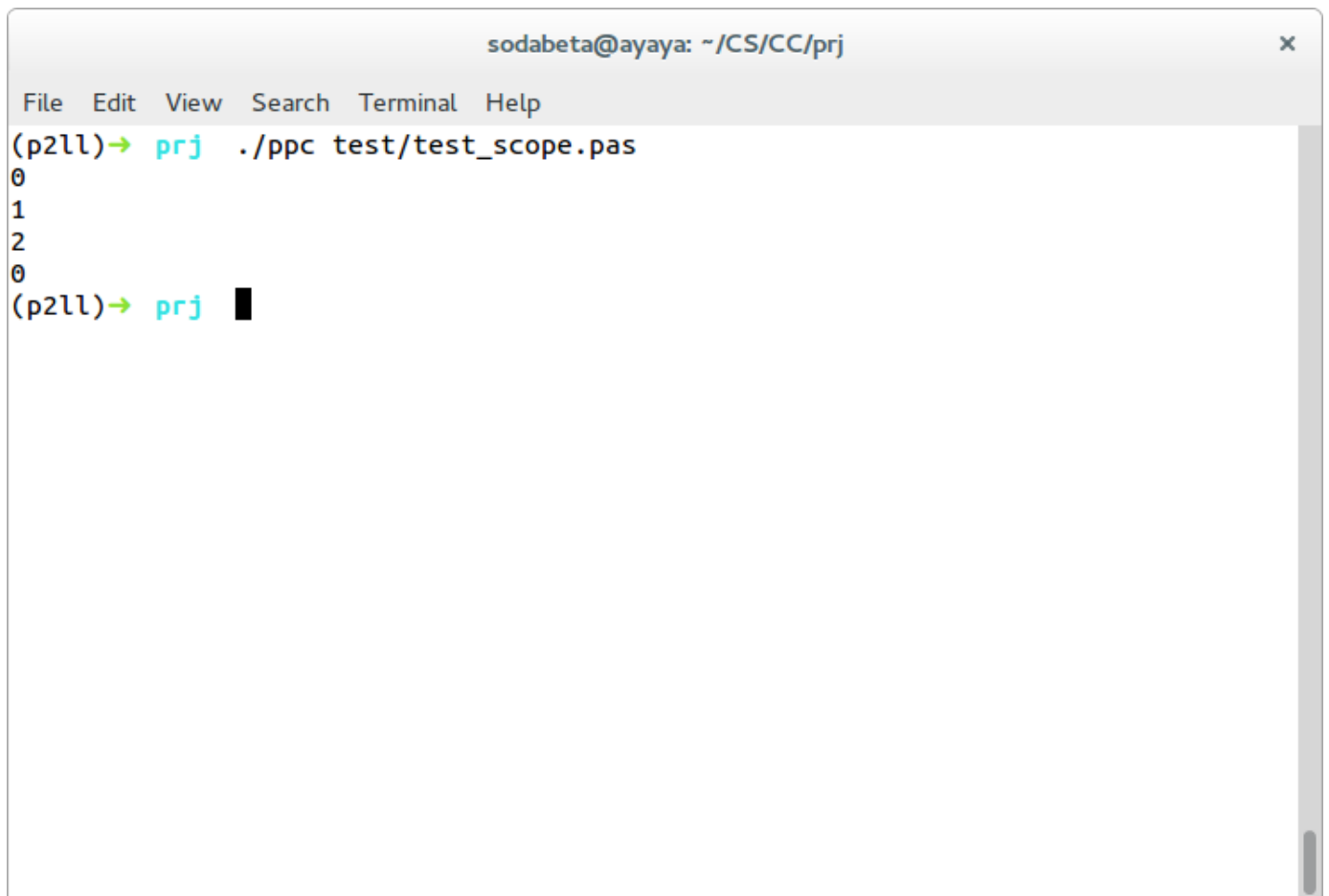
```
program test_2;
type
    arr = array [1..50] of integer;
var i,a,b,c,s: integer;
    f:arr;

function g1(x:integer):integer;
    var s:integer;
    function g2(x:integer):integer;
        var s:integer;
    begin
        s:=2;
        writeln(s);
        g2:=1;
    end;
begin
    s:=1;
    writeln(s);
    g2(12);
    g1:=1;
end;

begin
    s:=0;
    writeln(s);
    g1(3);
    writeln(s);
end.
```

可以看到在主程序里,首先定义了函数 g1 ,然后在 g1 内定义了内层的函数 g2 ,然后这三部分都定义了一个同名的变量 s ,并且赋予不同的值,预期的输出应该是 0 1 2 0

运行结果如下所示

A terminal window titled 'sodabeta@ayaya: ~/CS/CC/prj' with a menu bar (File, Edit, View, Search, Terminal, Help). The terminal shows the execution of a Pascal program 'test/test_scope.pas' using the 'prj' command. The output of the program is '0', '1', '2', and '0' on separate lines. The prompt '(p2ll)→ prj' is followed by a black cursor.

```
sodabeta@ayaya: ~/CS/CC/prj
File Edit View Search Terminal Help
(p2ll)→ prj ./ppc test/test_scope.pas
0
1
2
0
(p2ll)→ prj █
```

7.4 测试3

本测试主要测试错误恢复,测试代码是 test/test_error.pas .

测试代码如下所示

```
program test_1;
type
    arr = array [0..50] of integer;
var i,a,b,k,x,y,sum,testfunc : integer;
    aa:arr;
function fib(x:integer):integer;
begin
    if ((x = 0) or (x = 1)) then
        fib:=1
    else
        fib:=fib(x - 2) + fib(x - 1);
end;

function gao(x:integer):integer;
begin
    gao:=7;
end;
```

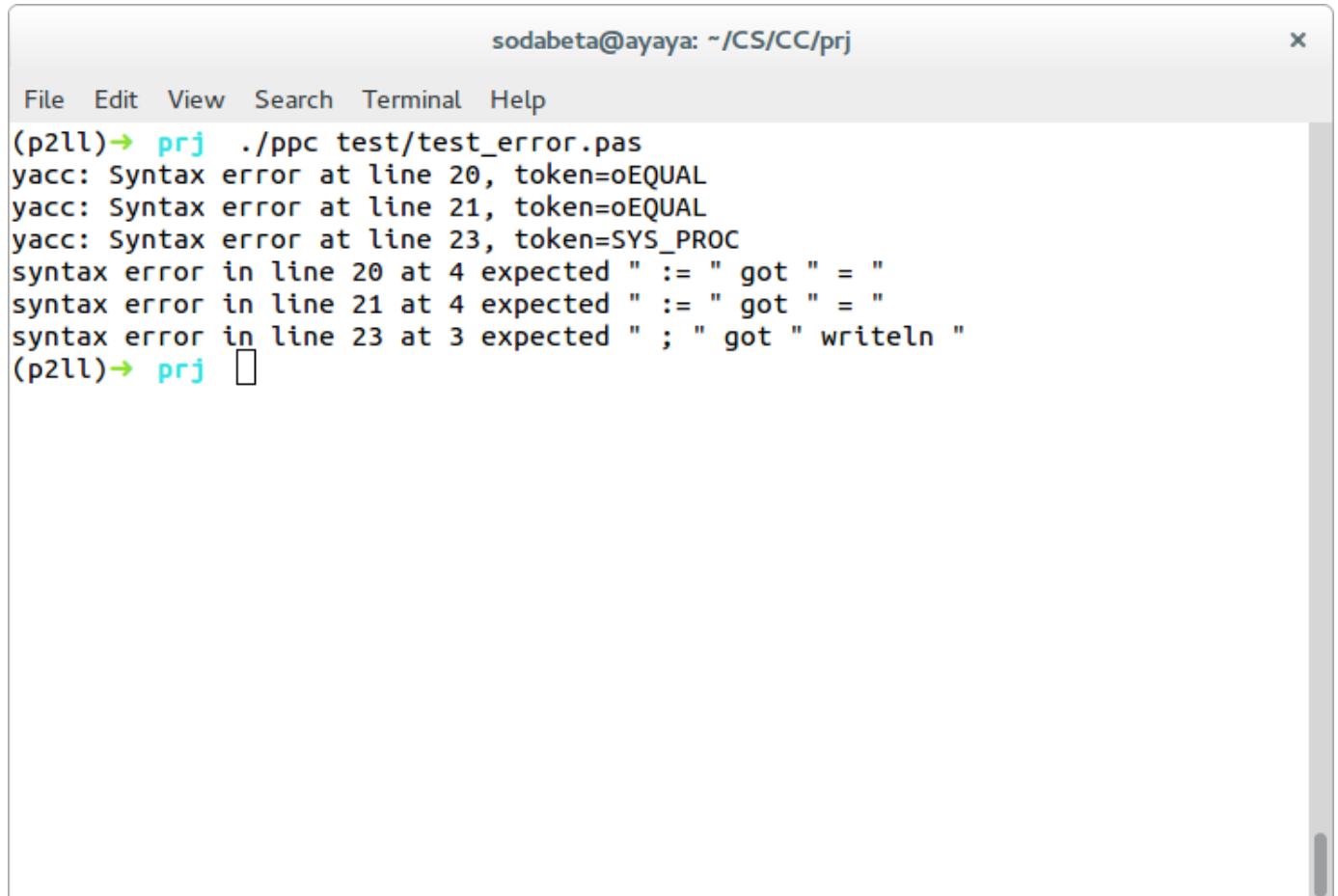


```
begin
  i=10;
  a=fib(10);
  b:=123456
  writeln(b);

  aa[3]:=10;
  a:=aa[3];
  aa[10]:=10;
  b:=fib(aa[10]);
  writeln(b);
  writeln(gao(8));
end.
```

可以看到程序在20,21,23行都有错误

下面是运行结果



The screenshot shows a terminal window titled "sodabeta@ayaya: ~/CS/CC/prj". The terminal displays the command to compile a Pascal program and the resulting syntax errors:

```
(p2ll)→ prj ./ppc test/test_error.pas
yacc: Syntax error at line 20, token=oEQUAL
yacc: Syntax error at line 21, token=oEQUAL
yacc: Syntax error at line 23, token=SYS_PROC
syntax error in line 20 at 4 expected " := " got " = "
syntax error in line 21 at 4 expected " := " got " = "
syntax error in line 23 at 3 expected " ; " got " writeln "
(p2ll)→ prj
```

可以看到编译器报告了多处错误.

