中国科学技术大学
University of Science and Technology of China

PL0 编译实验

报告人：xxx xxx xxx
指导教师：xxxx

20xx 年 xx 月 xx 日

大纲

大纲

声明语法:

- 1. Id dimDeclaration
- 2. dimDeclaration ::= [const] dimDeclaration
- 3. dimDeclaration ::= ε

使用语法:

- 1. Id DimUse
- 2. dimUse ::= [expression] dimUse
- 3. dimUse ::= ε

要点：

- 1. 修改 vardeclaration 和 factor 函数，使其支持数组声明和使用
- 2. 增加数组类型的数据结构，用于存储数组的信息，包括维度和总大小
- 3. 增加 array_analysis 函数，用于在使用数组时生成将相对数组首地址偏移压入栈中的代码，采用了老师 ppt 上的方法
- 4. 详情演示见后

大纲

print 语法

使用语法:

- 1. print (*expression, expression, ...*)
- 2. print ("*string*")

要点:

- 1. 修改 statement 函数，实现 print 分析
- 2. 分析时，生成将多个表达式的值或字符串的字符压入栈中的代码
- 3. 详情演示见后

大纲

使用语法：

- 1. scanf (*leftval, leftval, …*)
- 2. 左值是标识符或数组元素，拥有存储空间。

要点：

- 1. 修改 statement 函数，实现 scanf 分析
- 2. 分析时，生成将对应空间的地址压入栈中的代码
- 3. 详情演示见后

大纲

使用语法:

- 1. for (*var ID* : (*expression, expression, expression*))
- 2. for (*ID* : (*expression, expression, expression*))
- 3. for (*ID* : (*expression, expression*))
- 4. for (*var ID* : (*expression, expression*))
- 5. var 缺省时，ID 为已经定义的变量，否则为新定义的变量，step 缺省时默认为 1

具体实现

要点：

- 1. 修改 statement 函数，实现 for 分析
- 2. 分析时，生成将循环变量的初始值压入栈中的代码，并将循环变量的地址压入栈中
- 3. 生成赋值语句，将循环变量的初始值赋给循环变量
- 4. 生成将循环变量的当前值与终止值比较的代码，生成 JPC 指令，同时记录当前指令的位置
- 5. 生成循环体的代码
- 6. 生成将循环变量的当前值加上步长的代码
- 7. 生成循环变量的赋值代码
- 8. 生成 JMP 指令，跳转位置为比较代码的位置
- 9. 回填 JPC 指令的跳转位置，跳转位置为循环体的下一条指令
- 10. 详情演示见后

大纲

使用语法:

- 1. if expression then statement else statement
- 2. if expression then statement
- 3. 就近匹配

具体实现

要点：

- 1. 修改 statement 函数，实现 if else 分析
- 2. 分析时，生成将表达式的值压入栈中的代码
- 3. 生成 JPC 指令，同时记录当前指令的位置
- 4. 生成 if 语句块的代码
- 5. 生成 JMP 指令，同时记录当前指令的位置
- 6. 回填 JPC 指令的跳转位置，跳转位置为 else 语句块的第一条一条指令
- 7. 生成 else 语句块的代码
- 8. 回填 JMP 指令的跳转位置，跳转位置为 else 语句块最后一条语句的下一条指令
- 9. 详情演示见后

大纲

break 语法

使用语法:

- 1. statement ::= break
- 2. break 语句只能出现在循环语句 ("*while*" *or* "*for*") 中

具体实现

要点:

- 1. 在 complier 类里添加 is_for 和，is_while 变量和 break_arr 变量，用于标记当前是否在循环语句中以及记录 break 生成 JMP 的位置
- 2. 分析时，直接生成 JMP 指令，同时记录当前指令的位置 $(break\_arr)$
- 3. 在 for 或 while 分析开始时，
  $save\_is\_for\ while = is\_for\ while$，$is\_for\ while = true$,
  $save\_break\_arr = break\_arr$
- 4. 在 for 或 while 分析结束时，
  $is\_for\ while = save\_is\_for\ while$
- 5. 如果 $break\_arr\ != save\_break\_arr$, 则回填 break_arr 的跳转位置，跳转位置为 for 或 while 语句块的下一条指令，
  $break\_arr = save\_break\_arr$
- 6. 详情演示见后

大纲

- 在词法分析中，将 // 和 /**/ 注释全部过滤掉

- 1. sort1
- 2. sort2
- 3. for

大纲

setjmp 和 longjmp 语法

使用语法：

- 1. setjmp(*array*)：保存当前的执行环境到数组中，第一次调用返回 0
- 2. longjmp(*array, expression*)：恢复数组中保存的执行环境，val 为 setjmp 的返回值
- 3. 数组需要在程序中定义，且数组一维数组，长度需要大于等于 4

具体实现

要点:

- 1. 将 setjmp 看成单目运算符，修改 factor 函数，使之其能分析 setjmp 语句
- 2. 在 setjmp 语句分析时，生成把对应上下文存入数组的代码，同时生成 把$array[3]$ 取出压栈的代码，作为返回值
- 3. 在本次实现中，在已实现的数组语法中，数组被定义时会默认全部初始化为 0，因此第一次调用 setjmp 时，$array[3]$ 为 0，返回 0
- 4. array 前三个为 pc, ebp, esp
- 5. 在 longjmp 语句分析时，生成把数组的值取出分别赋值给 pc, ebp, esp 的代码
- 6. 同时生成把 expression 的值压栈的代码，赋值给 $array[3]$
- 7. 详情演示见后

- 1. setjmp1
- 2. setjmp2

大纲

使用语法:

- 1. $expression\_without\_condition \ ::= \ term\{addop\ term\}$
- 2. $expression\_without\_condition \ ::= \ leftval \ := \ expression$
- 3. $term \ ::= \ factor \ \{mulop\ factor\}$
- 4. $factor \ ::= (expression) \ | number \ | identifier \ | setjmp(array) \ | function\_call$
- 5. $mulop \ ::= \ * | \ /$
- 6. $addop \ ::= + | -$

- 7. *comparative_expression* ::= *expression_without_condition* {*compop expression_without_condition*}
- 8. *compop* ::= < | <= | > | >= | = | <>
- 9. *conditional_expression_AND* ::= *comparative_expression* {*AND comparative_expression*}
- 10. *conditional_expression_OR* ::= *conditional_expression_AND* {*OR conditional_expression_AND*}
- 11. *expression* ::= *odd conditional_expression_OR* | *conditional_expression_OR*

要点：

- 1. 将原本的 expression 函数修改为 expression_without_condition 函数
- 2. 增加 comparative_expression 函数
- 3. 增加 conditional_expression_AND 函数
- 4. 增加 conditional_expression_OR 函数
- 5. 增加 expression 函数
- 6. 在 expression_without_condition 函数中，增加对赋值表达式的分析
- 7. 修改解释器的 STO 和 STOA 指令，赋值后不弹栈
- 8. 栈优化，增加POP指令，在expression等使用结束后，生成POP指令，把无用数据弹出。

- 1. expression1
- 2. expression2

大纲

定义语法:

- 1. *function* ::= *procedure function_head* ; *function_body*
- 2. *function_head* ::= *function_name* (*parameter_list*) | *function_name*
- 3. *function_name* ::= *identifier*
- 4. *parameter_list* ::= *parameter* {, *parameter*}
- 5. *parameter* ::= *identifier*
- 6. *function_body* ::= *block*
- 7. *statement* ::= *return val*
- 8. *val* ::= ε | *expression*

具体实现

要点:

- 1. 增加 parameter 函数，分析函数的形参，将其加入变量表，dx 从-1 开始
- 2. 增加 transmit_parameters 函数，分析试，生成将参数压入栈的代码
- 3. 修改 statement 函数，增加分析 return 的语言，将 expression 的值或 0 压入栈
- 4. 修改 block 函数，在 statement 后，生成 return 0 语句的代码，无论 statement 里是否有 return，确保函数正常返回
- 5. 修改 factor 函数，把函数调用作为因子处理
- 5. 实现传参和返回值，递归的实现是自然而然的
- 6. call 的语法没有变化，兼容原本语法，只能 call 无参函数
- 7. 使用语法同 C，$f(a,b)$ 之类

- 1. function1
- 2. function2

大纲

总结

- 1. 实现了 setjmp 和 longjmp 函数,print 和 scanf 函数，和注释
- 2. 实现了赋值表达式，条件表达式和普通表达式的统一
- 3. 实现了数组和 for 循环，以及 break
- 4. 实现了函数传参和返回值，以及递归
- 5. 增加了LEA，STOA，LODA，POP指令，call指令的扩充
- 6. 代码整个由C++重构，增加注释，提高了可读性

总结

- 1. print 和 scanf 函数，ppt（xxx）
- 2. 表达式，注释，测试（xxx）
- 3. 其他部分（xxx）

谢谢!