Step 4 Report

刘轩奇 2018011025

2020年10月5日

1 工作内容

本人选择不使用辅助工具 ANTLR 因此自己实现了 lexer 和 parser。

1.1 文件说明

`montLexer.h/cpp` 词法分析器；

`montParser.h/cpp` 语法分析器；

`montConceiver.h/cpp` 产生中间代码；

`montAssembler.h/cpp` 从中间代码产生汇编代码；

`montLog.h` 记录编译错误信息；

`montCompiler.cpp` MiniDecaf 编译器，包含主函数，编译成功返回 0 否则返回 -1，并将错误信息输出到 `std::cerr`。

1.2 本步骤完成的工作

1 词法分析 添加了以下新的 Token：

* `EQUAL` 等号
* `NOT_EQUAL` 不等号
* `GREATER` 大于号
* `GREATER_EQUAL` 大于等于号
* `LESS` 小于号
* `LESS_EQUAL` 小于等于号
* `LAND` 逻辑与
* `LOR` 逻辑或

2 句法分析 修改了 `expression` 节点的生成逻辑，并添加了以下新的 AST 节点，对应于指导书给出的非终结符：

* `logical_and`
* `logical_or`
* `equality`

* relational

3 产生中间代码 按照指导书上的说明，将对应符号分别生成对应的中间代码：

* EQ, NEQ
* LT, GT, LE, GE
* LAND, LOR

4 产生汇编代码 新的中间代码将生成对应的汇编代码（省略了载入操作数和存储结果的语句）：

* [EQ] sub t1,t1,t2; seqz t1,t1;
* [NEQ] sub t1,t1,t2; snez t1,t1;
* [LE] sgt t1,t1,t2; xori t1,t1,1;
* [LT] slt t1,t1,t2;
* [GE] slt t1,t1,t2; xori t1,t1,1;
* [GT] sgt t1,t1,t2;
* [LAND] snez t1,t1; snez t2,t2; and t1,t1,t2;
* [LOR] or t1,t1,t2; snez t1,t1;

2 思考题

1 在表达式计算时，对于某一步运算，是否一定要先计算出所有的操作数的结果才能进行运算？

答 并非如此。例如计算逻辑或时，若首个操作数为真，则结果必为真，无需再计算第二个操作数的值。但在本次实验中，并未要求实现这种短路功能。

2 在 MiniDecaf 中，我们对于短路求值未做要求，但在包括 C 语言的大多数流行的语言中，短路求值都是被支持的。为何这一特性广受欢迎？你认为短路求值这一特性会给程序员带来怎样的好处？

答 一方面，短路特性能够提高程序运行的效率，当第一个操作数已经能决定运算结果时，不用再计算第二个操作数的值；另一方面，可以利用这种特性进行程序流程控制，使得程序更加简洁。例如希望第一个过程执行成功时不再执行第二个过程，这时就可以利用逻辑或的短路特性，这正是给程序员带来的便利之处。