参考编译器介绍

参考编译器为2020级yjk学长的代码结构

总体结构

编译器总体分为前端，中端，后端

前端：

实现词法分析、语法分析、语义分析

中端：

实现错误处理、基于llvm的符号表设计、llvm中间代码生成

后端：

实现mips代码生成，mips符号表和寄存器表设计

接口设计

由于整个编译器接口数量几百个，这里只说明核心接口

前端：

1. `TokenType`，词法分析的核心，为所有token标定类型并返回
2. `TokenLexer`，词法分析器，输入源代码文件，利用自己设计的迭代器解析所有token并保存，最终返回一个tokenList
3. `CompUnitParser`，核心编译单元，语法分析的起点，根据提供的tokenList，自顶向下解析
4. 所有文法里面定义的非终结符都设计成独立的接口，以及相对应的解析器，完成对整个文法的语法树的构建
5. `SyntaxNode`，所有分析器都要实现的输出接口，用于将分析器内容转换为字符串输出到文件

中端：

1. `Error`，错误处理的核心，为所有的错误类型和处理方法提供接口
2. `ErrorType`，定义所有错误类型
3. `Symbol`，属于前端和中端的符号，定义特殊的结构
4. `SymbolTable`，符号表和作用域对应，维护所有符号的作用域
5. `IrType`，llvm的核心类型，所有其它类型都从继承llvmType
6. `IrModule`，从`CompUnit`构建，相互对应
7. `IrBuilder`，llvm核心构建程序，将输入的`CompUnit`解析并返回`IrModule`
8. `IrValue`，所有llvm语句都继承自value，为所有语句提供公共的方法和属性
9. 其余的比如`IrBasicBlock, IrFunction, IrConstant, IrGlobalVariable`等接口都是符合llvm架构设计的
10. `IrNode`，所有value都要实现的接口，提供输出函数

后端：

1. `MipsBuilder`，后端构建的核心接口，将输入的`IrModule`转换为一个`mipsModule`返回
2. `MipsModule`，后端构建完成返回的核心类，包含后端mips的整体结构
3. `MipsSymbol`，mips独有的符号，建立从llvm虚拟寄存器的映射
4. `MipsSymbolTable`，每个函数维护一个表，和llvm的符号表相对应
5. `RegisterFile`，寄存器表，和mipsSymbolTable对应，维护当前函数的寄存器，提供寄存器的分配和回收机制

6. 其余的`MipsBasicBlock, MipsFunction, MipsInstruction`提供的接口和llvm差不多，都是提供解析与返回
7. `MipsNode`，所有上述mips类都要实现的输出接口

文件组织

文件组织结构涉及的文件众多，因此只展示部分核心文件

```
--Compiler
--/frontend
    --/lexer
        Token
        TokenLexer
        TokenType
    --/parser
        --/declaration
        --/expression
        --/function
        --/statement
        --/terminal
        CompUnit
        CompUnitParser
--/middle
    --/error
        ErrorTable
        Error
        ErrorType
    --/llvmir
        --/type
        --/value
        IrBuilder
        IrModule
        IrValue
    --/symbol
        Symbol
        SymbolTable
        SymbolType
--/backend
    --/basicblock
    --/function
    --/instruction
    --/symbol
        MipsSymbol
        MipsSymbolTable
    MipsBuilder
    MipsModule
    RegisterFile
```

编译器总体设计

总体结构

总体结构分为前端、中端和后端

前端：

实现词法分析，语法分析，语义分析和错误处理

中端：

实现符号表设计和llvm中间代码生成

后端：

实现mips生成

接口设计

接口数量太多，仅展示部分核心接口

前端

1. `Token`，词法分析的核心设计token类
2. `TokenType`，根据匹配规则，为所有token返回类型
3. `TokenLexer`，词法分析核心，将输入的源代码文件按行解析，返回tokenList
4. `CompUnitParser`，语法分析核心，语法分析的起点，将tokenList自顶向下解析构造语法树
5. `CompUnit`，语法分析的返回值，语法树的根
6. `MainFunDef`，main函数独有的函数定义接口
7. `MainFuncDefParser`，解析main部分
8. `parserOutput`，所有解析器都要实现的输出接口
9. `globalParm`，维护所有解析器之间要使用的公共全局变量和表

中端

1. `Error`，提供错误处理的规范性接口
2. `ErrorType`，定义错误的类型
3. `Symbol`，为llvm以及语义分析定义符号数据结构
4. `SymbolTable`，存储符号，每个作用域维护一个符号表
5. `llvmBuilder`，llvm构建核心文件，将输入的语义分析语法树转换为llvm结构
6. `llvmModule`，llvm的模块，和编译单元CompUnit对应
7. `llvmValue`，所有的value都要继承自llvmValue，提供公共的属性和方法
8. `llvmType`，所有类型都继承自llvmType
9. `llvmOutput`，为所有的llvmValue提供输出的接口

后端

优化前：

1. `mipsGenModule`，将输入的llvm语句逐句翻译，构建内存分配表，翻译成mips指令

优化后：

1. `mipsModuleBuilder`，mips构建核心接口，将llvmModule转换为mips结构
2. `mipsModule`，mips的结构定义，和llvmModule相对应
3. `mipsSymbol`，mips的符号定义，维护mips符号和llvm的虚拟寄存器的关系
4. `mipsSymbolTable`，每个函数维护一个符号表，用于存储符号，寄存器，虚拟寄存器的映射关系
5. `mipsRegister`，每个符号表对应一个寄存器表，用于寄存器的分配和回收
6. 其它mips结构接口，比如`mipsFunc, mipsBasicBlock, mipsInstruction`
7. `mipsOutput`，所有mips的结构接口都要实现的输出接口

文件组织

文件数量过多，仅展示核心文件

```
--Compiler
--/frontend
    --/lexer
        Token
        TokenLexer
        TokenType
    --/parser
        --/Decl
        --/Exp
        --/FuncDef
        --/Stmt
        --/Terminal
        CompUnit
        CompUnitParser
        MainFuncDef
        MainFuncDefParser
        parserOutput
    GlobalParm
--/middle
    --/error
        Error
        ErrorType
    --/symbol
        Symbol
        SymbolTable
        SymbolType
    --/llvm
        --/constant
        --/type
        --/value
        llvmBuilder
        llvmModule
        llvmValue
        llvmOutput
--/backend2
    MipsGenModule
    Pair
--/backend
    --/mipsBasicBlock
    --/mipsFunction
    --/mipsInstruction
    --/symbol
        mipsSymbol
        mipsSymbolTable
    mipsModule
    mipsModuleBuilder
    mipsRegister
    mipsOutput
```

词法分析设计

编码前设计

1. `Token`类的设计，这是词法分析的基础，token应该包含有类型，行号，值
2. `TokenType`，没有过多涉及，根据文法给出的token表，设计为枚举类型
3. `TokenLexer`，一个解析翻译程序，读取一个文本文件，对每一行的字符串进行分析，采用状态机进行匹配token，将得到的token存到一个表里面
4. 错误处理，暂时没有标准的错误处理类，在`TokenLexer`里面对两个特殊错误进行特判输出

编码后修改

语法分析设计

编码前设计

对于非终结符的设计：

参照学长的设计，为每一个非终结符都设计一个分析器，同时设计一个自己的类

比如对于`CompUnit`这个非终结符，产生了`CompUnit`和`CompUnitParser`类

每个非终结符的`Parser`类的内容都根据其产生式右侧的结构进行设计

比如对于`AddExp → MulExp | AddExp ('+' | '−') MulExp`

`AddExpParser`就需要一个`ArrayList<MulExp>`

对于终结符的设计：

终结符不需要`Parser`，只需要有自身的类即可，这样整个语法树的构建就出现了终点

`GlobalParm`，这里还设计了一个通用的全局变量类，可以理解为所有公共的方法，比如获取一个token，回退一个token。这里还设计了错误的列表，将词法分析和语法分析的错误处理都放进这个错误表里面。这个类可以理解为一个公共的迭代器，工具类

编码后修改

对于有些情况，比如`Lval = exp;` `exp;`这两种情况会发生公共前缀冲突，预读会出问题，因此对这里进行了一个回溯的修改，确保不会产生问题

语义分析设计

编码前设计

语义主要在语法的基础上添加符号表，只需要在语法分析的所有的类里面添加一个符号表参数

所有类都有符号表的参数，符号表的来源分为继承和自己新创建

对于所有的`Func`，函数的入口需要自己新创建符号表

遇到`if`和`for`和`block`语句都会创建新符号表

然后读到对应的定义语句就会把符号添加到当前符号表里面

只设计了`symbol`类

设计通用的错误处理类，将词法分析，语法分析，语义分析的所有错误处理统一化

编码后修改

先前的符号设计会出现`if, for`后面不是`block`的情况而创建了符号表

因此推翻之前的新符号表创建条件，将符号表的创建统一放到`Block`上，只要有block就说明应该创建符号表

同时symbol类不足以清晰描述所有符号

扩展出symbol的如下子类：

1. `SymbolConst`，所有常量都使用这个类构建
2. `SymbolVar`，所有变量使用这个类构建
3. `SymbolFunc`，为函数的定义设计一个符号，同时能存储函数的形式参数

代码生成设计

编码前设计

类型设计，所有llvm的类型都继承自`llvmType`

1. `llvmFuncType`，函数类型
2. `llvmIntegerType`，只有char和int两种类型，所以设计了`i8, i32`
3. `llvmVoidType`，函数void的返回类型
4. `llvmPointerType`，指针类型
5. `llvmArrayType`，数组类型

整体架构设计

采用`llvmModuleBuilder`来构建，获取`CompUnit`，从语法树构建llvm结构

`llvmModule`包含`llvmGlobalVariable`和`llvmFunc`

`llvmFunc`包含很多的`llvmBasicBlock`

`llvmBasicBlock`包含各种从语法树翻译来的`llvmInstruction`

指令设计

1. binary指令，比如add, sub, sge, sle，计算指令和比较计算指令
2. 内存操作指令，alloca, load, store, gep
3. 其它指令, br, call, trunc, zext, ret

核心在于指令的翻译，对每一个stmt进行指令翻译

在`llvmInstructionBuilder`里面再次构建一个和语义分析一样的语法树，通过语法树的递归结构来解析

跳转指令比如`break, continue, br`先不考虑

所有指令涉及的虚拟寄存器的命名，指令构建的时候即命名分配序号

编码后修改

由于涉及cond表达式，因此添加`i1`类型

所有的基本块都需要有标签，因此创建`llvmLabel`，其也作为一种指令

淡化`llvmBasicBlock`的概念，所有的基本块的划分并不由`llvmBasicBlock`确定，这里的basicBlock只是作为指令的容器，并不作为实际基本块的划分，实际的划分依据为`llvmLabel`

由于先前的涉及提前为每个虚拟寄存器分配了序号，`if`语句的情况让序号报错，因此清除所有指令构建时的序号分配，统一在指令翻译完成，遍历所有的value，按序分配序号

由于有`break, continue`的存在，在指令构建的时候，需要维护当前的`for`语句的开始和结束标签

最后产生的llvm指令会出现ret语句和br语句相邻情况

同时秉持着basicBlock作为容器的思想，最后将一个func里的所有指令全部直接放到func里面，不经过中间的basicblcok的包裹，同时由于`ret, br`相邻冲突问题，删除所有`ret, br`到下一个`label`的指令解决冲突

代码优化设计

生成普通mips

只写了一个类`mipsGenModule`

直接根据每一条llvm指令进行强行翻译，所有的值直接在栈上分配内存并写入

没有寄存器分配，几乎所有指令都在使用`$t0, $t1, $t2`

为每个函数单独创建一个内存表的映射关系

优化设计

编码前设计

设立mips的符号表类`mipsSymbolTable`和`mipsSymbol`，符号存储llvm里虚拟寄存器的映射关系

设计`mipsRegister`寄存器表，对应每一个mipsSymbolTable， 调度当前符号表下，寄存器分配和回收

将所有的putch输出的常量转换成putstr，并添加asciiz到全局

优化设计中主要就是做了寄存器的分配，利用了符号表映射

由于llvm中的所有虚拟寄存器的名字在不同函数可能一样，在mips中会发生错误，因此修改所有的label标签，其名字前添加当前所在函数的名字来满足mips需求

编码后修改