实验过程

本次 parser2 的实验在难度上有所提升。实验推导用的语法主要参考 SYsY 提供的文档，同时补充了对于字符串的推导。在实验过程中还需要学习使用 llvm 的 json 库，学会如何使用 llvm 提供的方法来构建和操作 json 库。在大体上了解了实验的总体结构和 llvm 库的使用之后，剩下的实验就开始偏向体力活了。

实验中遇到的问题&解决方法

C++ initializer_list 的用法与右值引用

在对 llvm::json 里面对象的初始化中，我们经常可以看见一下的写法

```
llvm::json::Object tmp{{"kind", "TranslationUnitDecl"},
                       {"range", },
                       {"inner", llvm::json::Array{inner}}};
```

这里是使用了 C++ 的 initializer_list 语法来对一个对象进行初始化

llvm::json 中对象的成员函数接受的参数大多都是右值。如果我们想将某一个变量作为参数传递给某个成员函数，通常会出错（成员函数具体的参数类型我们可以参考 llvm/Support/JSON.h）。这里我们可以使用 `std::move` 来做一个类型转换

```
stak.push_back(std::move(tmp));
```

文法的翻译问题

SYsY 参考语法采用拓展的 Backus 范式。使用符号 `[...]` 表示方括号内包含的为可选项。符号 `{...}` 表示花括号内包含的为可重复 0 次或多次的项

在转换成语法生成式的时候，对一些正则表达式我们要进行适当的处理。以下用某些文法作为例子来说明我们如何对含有 `[...]` 和 `{...}` 的文法。

处理 `[...]`

例如文法 $CompUnit \rightarrow [\ CompUnit\ ]\ (\ Decl \mid FuncDef\ )$，我们首先将其划分为两条文法：$CompUnit \rightarrow\ CompUnit\ \ (\ Decl \mid FuncDef\ )$ 和 $CompUnit \rightarrow\ (\ Decl \mid FuncDef\ )$

最终，我们得到四条推导规则

```
CompUnit : FuncDef {}
         | VarDecl {}
         | CompUnit VarDecl {}
         | CompUnit FuncDef {}
```

处理 `{...}`

例如文法 $VarDecl \rightarrow BType\ \ VarDef\ \{','VarDef\}$，我们添加一个符号 $VarDefs$ 来表示 $VarDef\ \{','VarDef\}$，然后利用左递归的形式来表示 `{...}`：
$VarDecl \rightarrow BType\ \ VarDefs$ 和 $VarDefs \rightarrow VarDef$ 、
$VarDefs \rightarrow VarDefs\ \ ','\ \ VarDef$

最终，我们可以得到以下推导规则

```
VarDecl : BType VarDefs {}

VarDefs : VarDef {}
        | VarDefs T_COMMA VarDef {}
```

内存管理问题

在完成本次实验的时候，本人经常会遇到 Segmentation Fault 的问题或者遇到一些看起来十分诡异的问题（比如说程序在解析的时候会在json中写入一些奇怪的字符）。在经过一系列的 debug 之后，本人发现绝大部分的问题是由于内存管理有误导致的。而导致内存出现问题大都是由于我们取出 `stak` 栈顶元素不规范导致的，这其中也涉及到了深浅拷贝的问题。

```
auto Ident = stak.back();
stak.pop_back();

// 将 Ident 作为 llvm::json::Object 使用
Ident.getAsObject()->get("...");
```

当我们直接使用 `back()` 函数来取出栈顶对象时，此时进行的是深拷贝。但是，当我们试图偷懒，在取出栈顶元素并将其作为 `llvm::json::Object` 使用的时候，内存问题就出现了。

```
auto Ident = stak.back().getAsObject();
stak.pop_back();
```

此时进行的就是浅拷贝了。我们查阅头文件 `llvm/Support/JSON.h` 中 `llvm::json::Array` 的成员函数，如下：

```
Value &back() { return V.back(); }
const Value &back() const { return V.back(); }
```

`back()` 返回的是一个应用，而 `llvm::json::Value` 中的 `getAsObject()` 定义如下：

```
const json::Object *getAsObject() const {
  return LLVM_LIKELY(Type == T_Object) ? &as<json::Object>() : nullptr;
}
json::Object *getAsObject() {
  return LLVM_LIKELY(Type == T_Object) ? &as<json::Object>() : nullptr;
}
```

因此，当使用 `stak.back().getAsObject()` ，并随后就调用 `stack.pop_back()` 之后，就会产生一个野指针，最终导致内存管理的问题。

处理隐式类型转换的问题

在 llvm 生成的语法树中，通常会进行一系列的隐式类型转换。为了处理这个问题，我们需要记录每一个变量的类型。因此虽然 parser-1 并不需要我们推导 `type` ，但是我们仍然试图推导 AST 树中每一个节点的 `type` 。

为了解决这个问题，我们建立了一个符号表来记录变量的类型，并对类型的优先级进行了一些规定。当在一个表达式中存在类型不同的元素时，我们将会根据元素的优先级进行一个隐式类型转换；或者在函数调用时函数的参数类型与实参类型不对时，我们也需要进行一个隐式类型转换。

处理 if else 推导时导致的ast栈问题

对于 if else 代码块，我们采用以下语句进行推导

```
Stmt
: T_IF T_L_PATEN Cond T_R_PAREN Stmt {}
| T_IF T_L_PAREN Cond T_R_PAREN Stmt T_ELSE Stmt {}
```

在实际推导到 `T_IF T_L_PAREN Cond T_R_PAREN Stmt` 的时候，为了确定当前状态到底是执行规约还是移进操作，yacc 还需要再执行一遍 `yylex()` ，读入一个符号进行判断。如果读入的符号是 T_ELSE ，则根据下面那条产生式进行规约；否则执行上面的产生式进行规约。假设读入的符号不是 T_ELSE，我们在对 `T_IF T_L_PAREN Cond T_R_PAREN Stmt` 进行规约的时候，我们自己维护的栈 `stak` 有可能会被压入一些奇怪的东西，因为此时 yacc 已经多执行了一次 `yylex()` 。比如如下的代码

```
if (i == 0) {
    a = 0;
}
b = 1;
```

yacc 多执行的一次 `yylex()` 会读入 `b` ，解析为 `T_IDENTIFIER` ，不属于 `T_ELSE` ，所以执行规约操作。但是在我们实现的 `yylex()` 中，当解析到 `T_IDENTIFIER` 时，我们会在 `stak` 中压入一个 `llvm::json::Object` 。这就导致了我们在执行 `Stmt : T_IF T_L_PAREN Cond T_R_PAREN Stmt` 规约时， `stak` 栈顶多了一个 `T_IDENTIFIER` 对应的 `llvm::json::Obeject` 。因此我们需要进行一次特判，如果在规约 `Stmt : T_IF T_L_PAREN Cond T_R_PAREN Stmt` 规则的时候，我们需要检查一下 `stak` 这栈顶是否有一些奇怪的东西，如果有，则要将其弹出并保存起来。

在规约完 `Stmt : T_IF T_L_PAREN Cond T_R_PAREN Stmt` 的时候，由于此前 yacc 已经执行多一次 `yylex()` 将 `b` 读进去了，因此下一个读取到的应该是 `=` 。这对 yacc 自己内部的解析栈而言是没有问题的，因为对于 yacc 而言， `b` 对应的信息已经保存好了。但是对于我们维护的 `stak` 而言，却缺少了 `b` 对应的信息。这是因为在规约 `Stmt : T_IF T_L_PAREN Cond T_R_PAREN Stmt` 的时候，我们将 `stak` 栈顶的元素弹出了。为了解决这个问题，如果我们在规约 if 语句的过程中从 `stak` 栈顶有弹出元素的话，我们作一次记录；当解析到下一个符号的时候，我们再将弹出的元素再次压回到栈中，将 `stak` 恢复。

如下图，我们进行了一次检查，如果此前从 `stak` 中弹出了元素，我们再执行下一次 `yylex()` 之间想要把 `stak` 栈恢复，将弹出的元素重新压回栈中。

```
auto yylex() {
  if (!isElse) {
    stak.push_back(tmp);
    isElse = true;
  }
```

实验评测结果

```
4 warnings generated.
[338/352] tester/h_functional/104_long_array.sysu.c
[339/352] tester/h_functional/105_long_array2.sysu.c
[340/352] tester/h_functional/106_long_code.sysu.c
[341/352] tester/h_functional/108_many_params.sysu.c
[342/352] tester/h_functional/109_many_params2.sysu.c
[343/352] tester/h_functional/110_many_params3.sysu.c
[344/352] tester/h_functional/111_many_globals.sysu.c
[345/352] tester/h_functional/112_many_locals.sysu.c
[346/352] tester/h_functional/113_many_locals2.sysu.c
[347/352] tester/h_functional/114_register_alloc.sysu.c
[348/352] tester/h_functional/115_nested_calls.sysu.c
[349/352] tester/h_functional/116_nested_calls2.sysu.c
[350/352] tester/h_functional/117_nested_loops.sysu.c
[351/352] tester/mizuno_ai/mizuno_ai.sysu.c
bigbigyu@DESKTOP-O4TTDUP:/mnt/g/CompileLab/SYsU-lang$
```

通过parser-1检测

```
-- Build files have been written to: /home/bigbigyu/sysu/build
[0/1] Running tests...
Test project /home/bigbigyu/sysu/build
      Start  1: lexer-0
 1/10 Test  #1: lexer-0 .............................   Passed    0.35 sec
      Start  2: lexer-1
 2/10 Test  #2: lexer-1 .............................   Passed   88.10 sec
      Start  3: lexer-2
 3/10 Test  #3: lexer-2 .............................   Passed   86.24 sec
      Start  4: lexer-3
 4/10 Test  #4: lexer-3 .............................   Passed   86.69 sec
      Start  5: parser-0
 5/10 Test  #5: parser-0 ............................   Passed    0.38 sec
      Start  6: parser-1
 6/10 Test  #6: parser-1 ............................   Passed  104.66 sec
      Start  7: parser-2
 7/10 Test  #7: parser-2 ............................***Failed    1.56 sec
[0/352] /mnt/g/CompileLab/SYsU-lang/tester/function_test2020/00_arr_defn2.sysu.c
[1/352] /mnt/g/CompileLab/SYsU-lang/tester/function_test2020/00_main.sysu.c
```