

# 编译原理lab2

专业	姓名	学号
计算机科学与技术	吴浩岚	19335209

## 零、llvm相关的用法

参考文件<https://github.com/llvm/llvm-project/blob/llvmorg-11.0.1/llvm/include/llvm/Support/JSON.h>

An **Array** is a JSON array, which contains heterogeneous JSON values.

An **Object** is a JSON object, which maps strings to heterogenous JSON values.

A **Value** is an JSON value of unknown type. 其中value包括

`null,boolean,number,string,object,array` 等类型

一些常用的函数如下:

```
json::Object *getAsObject() {  
    return LLVM_LIKELY(Type == T_Object) ? &as<json::Object>() : nullptr;  
}  
json::Array *getAsArray() {  
    return LLVM_LIKELY(Type == T_Array) ? &as<json::Array>() : nullptr;  
}  
Value *get(StringRef K);  
//...
```

## 一、全局变量与函数的处理

针对文件 `001_var_defn.sysu.c` 实现全局变量的处理, 和add表达式:

```
int a = 3;  
int b = 5;  
  
int main(){  
    return a + b;  
}
```

针对有可能有多个全局变量定义和全局函数的情况, 我们需要用到以下几个产生式:

```
CompUnit: CompUnitItem  
CompUnit: CompUnit CompUnitItem  
CompUnitItem: Decl {} | FuncDef {}
```

其中, `CompUnit: CompUnitItem`, 只需要弹栈, 并加入到 `TranslationUnitDecl` 的 `inner` 即可。

```

CompUnit: CompUnitItem {
  //llvm::errs() << "CompUnit: CompUnitItem";
  auto inner = stak.back();
  stak.pop_back();
  stak.push_back(llvm::json::Object{{"kind", "TranslationUnitDecl"},
                                     {"inner", *(inner.getAsObject()-
>get("inner"))}}});
}

```

而对于 `CompUnit: CompUnit CompUnitItem`，首先需要弹两次栈，在 `CompUnit` 的 `inner` 数组中加入 `CompUnitItem` 的相关结构。

```

CompUnit: CompUnit CompUnitItem {
  //llvm::errs() << "CompUnit: CompUnit CompUnitItem";
  auto inner1 = stak.back(); //CompUnitItem
  stak.pop_back();
  auto inner2 = stak.back(); //CompUnit
  stak.pop_back();
  inner2.getAsObject()->get("inner")->getAsArray()->push_back(*
(inner1.getAsObject()->get("inner")));
  stak.push_back(inner2);
}

```

最终的分析结果如下所示：

```

1  inner:
2  - inner:
3      - kind: IntegerLiteral
4        value: 3
5      kind: VarDecl
6      name: a
7  - inner:
8      - kind: IntegerLiteral
9        value: 5
10     kind: VarDecl
11     name: b
12 - inner:
13     - inner:
14         - inner:
15             - inner:
16                 - value: b
17                 - value: a
18                 kind: BinaryOperator
19             kind: ReturnStmt
20         kind: CompoundStmt
21     kind: FunctionDecl
22     name: main
23 kind: TranslationUnitDecl

```

## 二、处理函数块中的语句

针对 `01_var_defn.sysu.c`，处理函数块中的多个语句：

```
int a;
int main(){
    a=10;
    return 0;
}
```

函数块的处理，首先将一个stmt归约为一个BlockItem，之后的stmt只需不断地加入到BlockItem里面即可。（这种左递归地实现方式，将会多次出现，之后就不再提及了）如下所示：

```
BlockItem: Stmt {
    auto stmt = stak.back();
    stak.pop_back();
    stak.push_back(llvm::json::Object{{"kind", "CompoundStmt"},
                                       {"inner", llvm::json::Array{stmt}}});
}| BlockItem Stmt {
    auto stmt = stak.back(); //stmt
    stak.pop_back();
    auto list = stak.back(); //
    list.getAsObject()->get("inner")->getAsArray()->push_back(stmt);
    stak.pop_back();
    stak.push_back(list);
}
```

### 三、处理数组的定义和函数形参表

借助 00\_arr\_defn2.sysu.c，完善变量的定义。

```
int a[10][10];
int main(){
    int a[10][10];
    return 0;
}
```

```
[-VarDecl 0x2377498 <tester/function_test2020/00_arr_defn2.sysu.c:1:1, col:13> col:5 a 'int [10][10]'
~-FunctionDecl 0x23775a8 <line:2:1, line:5:1> line:2:5 main 'int ()'
~-CompoundStmt 0x23777a0 <col:11, line:5:1>
|~-DeclStmt 0x2377758 <line:3:5, col:18>
| |~-VarDecl 0x23776f0 <col:5, col:17> col:9 a 'int [10][10]'
| |~-ReturnStmt 0x2377790 <line:4:5, col:12>
| |~-IntegerLiteral 0x2377770 <col:12> 'int' 0
```

变量定义

VarDef → Ident { '[' ConstExp ']' }

| Ident { '[' ConstExp ']' } '=' InitVal

把后面 [10][10] 看作 ArrList。之后产生式如下：

```
VarDecl: BType VarList T_SEMI
ArrList: T_L_SQUARE Exp T_R_SQUARE | ArrList T_L_SQUARE Exp T_R_SQUARE
```

这里采用的方法也是产生式的递归操作，就不再赘述了。

而如果是对多维数组进行了初始化操作的，在赋值过程中又会有所不同，更加复杂一些：

```
int main(){
    int a[4][2]={1,2,3,4,5,6,7,8};
    int b[4][2]={{a[0][0],a[0][1]},{3,4},{5,6},{7,8}};
    return 0;
}
```

```
-DeclStmt 0x9a4238 <line:3:5, col:54>
  ~VarDecl 0x9a3b10 <col:5, col:53> col:9 b 'int [4][2]' cinit
    ~InitListExpr 0x9a4028 <col:17, col:53> 'int [4][2]'
      ~InitListExpr 0x9a4098 <col:18, col:34> 'int [2]'
        ~ImplicitCastExpr 0x9a40e8 <col:19, col:25> 'int' <LValueToRValue>
          ~ArraySubscriptExpr 0x9a3c98 <col:19, col:25> 'int' lvalue
            ~ImplicitCastExpr 0x9a3c80 <col:19, col:22> 'int *' <ArrayToPointerDecay>
              ~ArraySubscriptExpr 0x9a3c08 <col:19, col:22> 'int [2]' lvalue
                ~ImplicitCastExpr 0x9a3bf0 <col:19> 'int (*)[2]' <ArrayToPointerDecay>
                  ~DeclRefExpr 0x9a3b78 <col:19> 'int [4][2]' lvalue Var 0x9a36b8 'a' 'int [4][2]'
                  ~IntegerLiteral 0x9a3b98 <col:21> 'int' 0
                ~IntegerLiteral 0x9a3c28 <col:24> 'int' 0
              ~ImplicitCastExpr 0x9a4100 <col:27, col:33> 'int' <LValueToRValue>
                ~ArraySubscriptExpr 0x9a3d68 <col:27, col:33> 'int' lvalue
                  ~ImplicitCastExpr 0x9a3d50 <col:27, col:30> 'int *' <ArrayToPointerDecay>
                    ~ArraySubscriptExpr 0x9a3d10 <col:27, col:30> 'int [2]' lvalue
                      ~ImplicitCastExpr 0x9a3cf8 <col:27> 'int (*)[2]' <ArrayToPointerDecay>
                        ~DeclRefExpr 0x9a3cb8 <col:27> 'int [4][2]' lvalue Var 0x9a36b8 'a' 'int [4][2]'
                        ~IntegerLiteral 0x9a3cd8 <col:29> 'int' 0
                      ~IntegerLiteral 0x9a3d30 <col:32> 'int' 1
                    ~InitListExpr 0x9a4128 <col:36, col:40> 'int [2]'
                      ~IntegerLiteral 0x9a3dd8 <col:37> 'int' 3
                      ~IntegerLiteral 0x9a3df8 <col:39> 'int' 4
                    ~InitListExpr 0x9a4188 <col:42, col:46> 'int [2]'
                      ~IntegerLiteral 0x9a3e68 <col:43> 'int' 5
                      ~IntegerLiteral 0x9a3e88 <col:45> 'int' 6
                    ~InitListExpr 0x9a41e8 <col:48, col:52> 'int [2]'
                      ~IntegerLiteral 0x9a3ef8 <col:49> 'int' 7
                      ~IntegerLiteral 0x9a3f18 <col:51> 'int' 8
                ~ReturnStmt 0x9a4270 <line:4:5, col:12>
                  ~IntegerLiteral 0x9a4250 <col:12> 'int' 0
```

使用的产生式如下：

```
VarDef: Ident ArrList T_EQUAL InitVal{...}
InitVal: T_L_BRACE InitValList T_R_BRACE{...} | T_L_BRACE T_R_BRACE{...} |
Exp{...}
InitValList: InitVal{...} | InitValList T_COMMA InitVal {...} //也采用了
左递归
Exp:
LVal: ArrExp{...}
```

简单解释一下，其中 `{a[0][0],a[0][1]}` 属于 `LVal` 左值，归约为 `Exp` 的一种，又可归约为 `InitVal`。

这里涉及到一个比较麻烦的东西——左值。可以看一下以下几种情况：

```

-FunctionDecl 0x1d0f3c8 <tester/function_test2020/00_arr_defn2.sysu.c:2:1, line:7:1> line:2:5 main 'int ()'
  -CompoundStmt 0x1d0f940 <col:11, line:7:1>
    -DeclStmt 0x1d0f6e0 <line:3:5, col:23>
      -VarDecl 0x1d0f558 <col:5, col:22> col:9 used b 'int [3]' cinit
        -InitListExpr 0x1d0f688 <col:16, col:22> 'int [3]'
          -IntegerLiteral 0x1d0f5c0 <col:17> 'int' 1
          -IntegerLiteral 0x1d0f5e0 <col:19> 'int' 2
          -IntegerLiteral 0x1d0f600 <col:21> 'int' 3
          int b[3] = {1,2,3};
    -DeclStmt 0x1d0f840 <line:4:5, col:17>
      -VarDecl 0x1d0f710 <col:5, col:16> col:9 a 'int' cinit
        -ImplicitCastExpr 0x1d0f828 <col:13, col:16> 'int' <LValueToRValue>
          -ArraySubscriptExpr 0x1d0f808 <col:13, col:16> 'int' lvalue
            -ImplicitCastExpr 0x1d0f7f0 <col:13> 'int *' <ArrayToPointerDecay>
              -DeclRefExpr 0x1d0f778 <col:13> 'int [3]' lvalue Var 0x1d0f558 'b' 'int [3]'
              -IntegerLiteral 0x1d0f798 <col:15> 'int' 1
            int a = b[1];
        -BinaryOperator 0x1d0f8f0 <line:5:5, col:12> 'int' '='
          -ArraySubscriptExpr 0x1d0f8b0 <col:5, col:8> 'int' lvalue
            -ImplicitCastExpr 0x1d0f898 <col:5> 'int *' <ArrayToPointerDecay>
              -DeclRefExpr 0x1d0f858 <col:5> 'int [3]' lvalue Var 0x1d0f558 'b' 'int [3]'
              -IntegerLiteral 0x1d0f878 <col:7> 'int' 2
            b[2] = 6;
          -IntegerLiteral 0x1d0f8d0 <col:12> 'int' 6
        -ReturnStmt 0x1d0f930 <line:6:5, col:12>
          -IntegerLiteral 0x1d0f910 <col:12> 'int' 0

-FunctionDecl 0x1d4f3c8 <tester/function_test2020/00_arr_defn2.sysu.c:2:1, line:7:1> line:2:5 main 'int ()'
  -CompoundStmt 0x1d4f940 <col:11, line:7:1>
    -DeclStmt 0x1d4f6e0 <line:3:5, col:23>
      -VarDecl 0x1d4f558 <col:5, col:22> col:9 used a 'int' cinit
        -InitListExpr 0x1d4f688 <col:16, col:22> 'int [3]'
          -IntegerLiteral 0x1d4f5c0 <col:17> 'int' 1
          -IntegerLiteral 0x1d4f5e0 <col:19> 'int' 2
          -IntegerLiteral 0x1d4f600 <col:21> 'int' 3
          int a = {1,2,3};
    -DeclStmt 0x1d4f840 <line:4:5, col:17>
      -VarDecl 0x1d4f710 <col:5, col:16> col:9 b 'int' cinit
        -ImplicitCastExpr 0x1d4f828 <col:13, col:16> 'int' <LValueToRValue>
          -ArraySubscriptExpr 0x1d4f808 <col:13, col:16> 'int' lvalue
            -ImplicitCastExpr 0x1d4f7f0 <col:13> 'int *' <ArrayToPointerDecay>
              -DeclRefExpr 0x1d4f778 <col:13> 'int [3]' lvalue Var 0x1d4f558 'a' 'int [3]'
              -IntegerLiteral 0x1d4f798 <col:15> 'int' 1
            int b = a[1];
        -BinaryOperator 0x1d4f8f0 <line:5:5, col:12> 'int' '='
          -ArraySubscriptExpr 0x1d4f8b0 <col:5, col:8> 'int' lvalue
            -ImplicitCastExpr 0x1d4f898 <col:5> 'int *' <ArrayToPointerDecay>
              -DeclRefExpr 0x1d4f858 <col:5> 'int [3]' lvalue Var 0x1d4f558 'a' 'int [3]'
              -IntegerLiteral 0x1d4f878 <col:7> 'int' 2
            a[2] = 6;
          -IntegerLiteral 0x1d4f8d0 <col:12> 'int' 6
        -ReturnStmt 0x1d4f930 <line:6:5, col:12>
          -IntegerLiteral 0x1d4f910 <col:12> 'int' 0

```

- 左值是可寻址的变量，有持久性；
- 右值一般是不可寻址的常量，或在表达式求值过程中创建的无名临时对象，短暂性的。

比如，第三条中的6为右值，b[2]为左值。此外，a=10;和b[2]=6;和其他左值的解析方式也不太一样。

语句 Stmt → LVal '=' Exp ';'

左值表达式 LVal → Ident {' Exp '}

基本表达式 PrimaryExp → '(' Exp ')' | LVal | Number

所以我们可以将其分为两种情况来讨论：


```

LVal: Ident{...} | ArrExp{...}
EqualLVal: Ident{...} | ArrExp{...}
Stmt: EqualLVal T_EQUAL Exp T_SEMI{...}

```

对于函数的形参表，和数组的处理过程是有些类似的，同样要通过递归的方式来处理形参。所以就不再赘述了。

## 四、关于隐式类型转换


Yorkking 18 days ago




100\_int\_literal.sysu.c 的问题，这里涉及到类型转换的问题，unsigned int 和 int 转换的问题。观察了一下 clang 的语法树得出结论：

- i. a = b + c: b 和 c 类型不一致时，往类型大的转，比如 int -> unsigned int
- ii. a 和 b + c 返回的结果类型不一致时，强行往 a 的类型转。

个人的解决方法是，给每个节点引入一个额外的 type 属性，比如问题中的 const int k2 = 0x80000000 + 1;，首先 0x80000000 加入 "type:unsigned int" 属性，1 加入 "type:int" 属性。然后在二元表达式运算的时候，比较两个的 type 是否相同，然后按照上述的规则进行类型转换，即插入 ImplicitCastExpr。然后变量声明的时候也做类似的检查 type 操作。

另外，实际上只有这个例子会涉及到复杂的类型转换，只需要针对该例子修改和 const int k2 = 0x80000000 + 1; 相关的文法即可。


Marked as answer

 3

 1

0 replies

处理起来比较麻烦，所以只好直接针对该样例进行处理。

## 五、处理八进制和十六进制数据

参考网上的教程，有如下方式，可以使用字符串流来方便地转化八进制和十六进制：

```
#include <sstream>
int x;
stringstream ss;
ss << std::hex << "1A"; //std::oct (八进制)、std::dec (十进制)
ss >> x;
cout << x<<endl;
```

所以，我们可以在识别到 `num` token 的时候，进行十六/八进制转换为十进制的操作。

```
if (t == "numeric_constant")
{
    lex_str = string(s);
    if(lex_str[1]=='x' || lex_str[1] == 'X'){
        stringstream ss;
        ss << std::hex << lex_str;
        ss >> num;
        lex_str = to_string(num);
    }else if(lex_str[0] == '0'){
        stringstream ss;
        ss << std::oct << lex_str;
        ss >> num;
        lex_str = to_string(num);
    }
    return T_NUMERIC_CONSTANT;
}
```

## 六、实验结果

最终，`parser1` 可以完全通过。

```
wuhlan3@PC-202001212347:~/lab2/SysU-lang$ CTEST_OUTPUT_ON_FAILURE=1 cmake --build ~/sysu/build -t test
[0/1] Running tests...
Test project /home/wuhlan3/sysu/build
Start 1: lexer-0
1/10 Test #1: lexer-0 ..... Passed    0.37 sec
Start 2: lexer-1
2/10 Test #2: lexer-1 ..... Passed   91.25 sec
Start 3: lexer-2
3/10 Test #3: lexer-2 ..... Passed   91.18 sec
Start 4: lexer-3
4/10 Test #4: lexer-3 ..... Passed   90.34 sec
Start 5: parser-0
5/10 Test #5: parser-0 ..... Passed    0.40 sec
Start 6: parser-1
6/10 Test #6: parser-1 ..... Passed  130.69 sec
Start 7: parser-2
7/10 Test #7: parser-2 .....***Failed    0.45 sec
[0/352] /home/wuhlan3/lab2/SysU-lang/tester/function_test2020/00_arr_defn2.sysu.c
```