

REPORT FOR COMPILER EXPERIMENT

**

2022 年 6 月 6 日

1. 基于递归下降子程序法的三地址代码生成器
2. 基于 LR 分析法的三地址代码生成器
3. 基于 LLVM 的类 C 语言编译器

基于递归下降子程序法的三地址代码 生成器

- Int Lookhead: //储存当前 lookahead 的状态。
 - 状态包括: NONE, IDN, IF, THEN, ELSE, WHILE, DO, INT10, REAL10, INT8, REAL8, INT16, REAL16, ADD, SUB, MUL, DIV, EQ, GT, LT, LP, RP, SEMI, ASG, WRONG
- String token: //储存当前 token 的值。
 - 起始符 P, 非终结符 L 的结构体
- string code //储存 P,L 的代码
- 非终结符 T,F,E 的结构体:
 - string code //存放 T,E,F 的代码
 - string place //存放 idn 或 int 或 real 的值
- 非终结符 S 的结构体:
 - sstring code //存放 S 的代码
 - int begin //S 的起始位置
 - int next //S 的下一步的位置
- 非终结符 C 的结构体
 - string code //存放 C 的代码
 - int afalse //存放 false 时跳转位置
 - int atrue //存放 true 时跳转位置

- Proc_P: 起始符的子程序, 处理 $P \rightarrow L$, 调用 L, 获取 L.code; 处理 $P \rightarrow LP1$, 若有用 “;” 分割的式子会递归调用自己 (P1)。
- Proc_L :L 的子程序, 处理 $L \rightarrow S$ 产生式, 会调用 S 并获取 S.code;
- Proc_S :S 的子程序, 根据 lookahead 和结果判断处理的是哪个产生式。
 - $S \rightarrow id=E$, 调用 E, 获取 E.code 和 E.place 并以三地址代码形式写入 S.code。
 - $S \rightarrow if\ C\ do\ S1\ (else\ S2)$, 调用 C 获得 C.code 然后递归调用自己 (S1)。S1 返回后根据 lookahead 是不是 else 决定是否再递归调用 S2。最终由 $C+S1$ 或 $C+S1+S2$ 生成自己的 S.code。
 - $S \rightarrow while\ C\ do\ S$, 调用 C 然后递归调用自己 (S1), 由 C, S1 生成 S.code
- Proc_C: 先调用 E1, 获得 E1.code, E1.place, 然后根据 lookahead(是 $>$ 或 $<$ 或 $=$) 判断处理的是哪个产生式。不管调用哪个产生式最后只有符号 ($>$, $<$, $=$) 不一样, 都是再调用 E2, 获得 E2.code, E2.place, 然后生成 C.code。

- Proc_E: 先调用 T1, 获得 T1.code, T1.place, 然后根据 lookahead(是 + 或-) 判断处理的是哪个产生式。不管调用哪个产生式最后只有符号 (+, -) 不一样, 都是再调用 T2, 获得 T2.code, T2.place, 然后生成 E.code。
- Proc_T: 先调用 F1, 获得 F1.code, F1.place, 然后根据 lookahead(是 * 或/) 判断处理的是哪个产生式。不管调用哪个产生式最后只有符号 (*, /) 不一样, 都是再调用 F2, 获得 F2.code, F2.place, 然后生成 T.code。
- Proc_F: 根据 lookahead 类型判断处理的是哪个产生式。F->idn: F.place = 标识符, F->int/real: F.place = 数值, F->(E), 调用 E, 获得 E.place, E.place 给 F.place, F.place。

采取的是递归子程序法，分别编写了 P,L,S,C,T,F,E 的子程序。从 P 开始深入，每一个子程序都会根据当前的 lookahead 值来确定当前读到的内容是否符合自己的产生式。是就会根据对应的产生式去调用其他的子程序，并从中获取其 code 和 place 来根据语义规则生成自己的 code。并把自己的 code 返回上级，也就是调用了自己的函数。最后一层层直到返回到 P，P 会生成最终的三地址代码，并将其输出。

基于 LR 分析法的三地址代码生成器

- Sentence 类：
 - String 类型 str: 存储文法
 - Int 类型 dot: 存储文法点的位置
 - 类函数: addDot: 给文法-> 后添加点
 - moveDot: 将文法点向后移动一位
- DFA 类：
 - Vector<string> sentence: 存储该状态的文法
 - Int num: 存储状态的 ID
 - Int fatherNum: 存储能到达该状态的标号（有多个只保留一个）
 - Vecotr<int> sonNum: 存储该状态能到达的状态标号（不包含自己）

Vector<DFA> LR: 存储状态

Vector<Sentence> input: 输入的文法

r[20][10]: 存储分析表的 acc 和 r (+ 状态号)

rr[20][10]: 存储分析表的状态号

- Init(): 生成第一个状态 S0, 把 S' -> 第一个非终结符加入到状态中。
- addSentence(DFA &dfa): 如果-> 后出现的第一个是非终结符, 将该非终结符开始的文法先 addDot, 然后添加到状态中。
- findSon(): 从 S0 开始, 给状态 S0 添加文法。再将每个文法移动 dot 后的文法作为下个状态。重复这个过程, 直到不再有新的状态产生。(每次添加状态前, 先判断是否重复)
- 状态生成完, 先重新编号, 再循环一遍找到每个状态的上一个状态和下一个状态。
- Match1(a,b,len): 判断状态是不是结束状态, 是否是终结文法
- output1(string str): 输入非终结符号和终结符号, 输出 LR 表,

基于 LLVM 的类 C 语言编译器

基于 LLVM 的类 C 语言编译器

ACHITECTURE DESIGN

- Use ANTLR4 as lexer and parser generator
 - Supports automatic conversion of left-recursive grammars
- Use Rust to write the compiler itself
 - Tagged unions
- Use LLVM IR as output
 - Produces industrial-quality binaries
 - Focus on COMPILER itself, not on implementation details on target platform

基于 LLVM 的类 C 语言编译器

BRIEF INTRODUCTION OF TOOLS USED

- ANTLR: ANother Tool for Language Recognition
- Supports ALL*(Adaptive-LL) grammars
 - Produces human-readable code
- Error recovery during lex and parse
- Attributes and Actions
- Automatic conversion of left-recursive grammars
- Even more!
 - Semantic predicate
 - Customized error recovery
 - ...

- A fairly new language designed by Mozilla
- Tagged Union

```
pub enum Type{  
    Int(size, signed),  
    Pointer(Box<Type> elem),  
    ...  
}
```

- Safe
 - no more segfaults!

- LLVM
- Focus on compile itself
 - ARCH-VENDOR-OS-ENV
 - x86_64-unknown-linux-gnu
 - Infinite number of registers
 - SSA(Static Single Assignment) values
 - Intrinsics
 - Most optimised for target platform
- Large eco-system
 - Clang – The most modern C compiler
 - Rust
 - Nvidia Cuda Compiler

```
# return a + b
define dso_local i32 @add(i32
  ↪ %0, i32 %1) {
%3 = alloca i32, align 4
%4 = alloca i32, align 4
store i32 %0, i32* %3, align 4
store i32 %1, i32* %4, align 4

%5 = load i32, i32* %3, align
  ↪ 4
%6 = load i32, i32* %4, align
  ↪ 4
%7 = add nsw i32 %5, %6
ret i32 %7
}
```

基于 LLVM 的类 C 语言编译器

LANGUAGE DESIGN

- Based on ANSI C
- With a bit modification
- `int arr[5] -> int[5] arr`
- remove union and enum
- ...

- Key points

- Key points
- Full support of multi-dimensional array
- Full support of structs
- Full support of Pointers
- Some support of function pointers
- And arbitrary combination of types above

```
int a = 0x1234;
int b = 'a';
char c = 3 + 235;
extern int printf(char*, ...);
char * str = "___Hello world from Cb lang!\n";
struct test{
    int foo;
    char bar;
};
int add(int a, int[] b){
    return a + b[0];
}
int main(int argc, char** argv){
```

```
int[5][2] array;
int[5]* array_ptr = array;
struct test ss;
struct test* ss_pointer = &ss;
ss.foo = 1234;
printf(&str[3]);
printf(str + 3);
printf("This is %d %d\n", 2 + 1238796, a);
printf("Argc: %d\n", argc);
array[0][0] = 0x123;
printf("array[0][0] = %d\n", array[0][0]);
printf("array_ptr[0][0] = %d\n", array_ptr[0][0]);
if(argc == 1) {
    printf("Then\n");
}
```

```
    return -1;
} else {
    printf("Else\n");
}
printf("%d\n", ss.bar);
{
    int* pointer = &ss.bar;
    *pointer = 0x1234;
    printf("%d\n", ss.bar);
    ss_pointer->foo = 0;
    printf("%d\n", ss.foo);
}
printf("A + B = %d\n", add(2, array[0]));
printf("argv: %s\n", argv[0]);
```



```

1 && printf("not_reached");
0 && printf("good\n");
0 || printf("not_reached");
1 || printf("good\n");
(0,1) || printf("good\n");
printf("1+2*3=%d\n", 1 + 2 * 3 * 3 / 3);
if (!1) {
    printf("not reached");
} else {
    printf("good\n");
}
{
    int i = 0;
    for(i = 0; i < argc; ++ i){

```

```
        if (i == 3){
            printf("Break!");
            break;
        } else if(i == 2){
            printf("Continue!");
            continue;
        }
        printf("for %d\n", i);
    }
}
```

基于 LLVM 的类 C 语言编译器

LEX

- Use ANTLR as lexer generator
- Optionally prints token stream for debugging

```
53  'int'          at 1:0
77  IDENTIFIER     at 1:4
1   '='           at 1:6
78  INTEGER        at 1:8
3   ';'           at 1:14
53  'int'          at 2:0
77  IDENTIFIER     at 2:4
1   '='           at 2:6
80  CHAR_LITERAL   at 2:8
3   ';'           at 2:11
51  'char'         at 3:0
77  IDENTIFIER     at 3:5
```

基于 LLVM 的类 C 语言编译器

SYNTAX ANALYSIS AND SEMANTIC ANALYSIS

- Do syntax analysis and semantic analysis in one step
 - Thanks to ANTLR

```
addSubExpr
returns[
    Option<Box<dyn ExprNode>> e
]: mulDivExpr {
    $e = $mulDivExpr.e;
} ( '+' mulDivExpr {
    let lhs = (&$e).clone().unwrap();
    let rhs = ($mulDivExpr.e).clone().unwrap();
    let location = $start.start as usize .. recog.get_current_token().stop as u
    $e = Some(Box::new(
        report_or_unwrap!(
            BinaryExprNode::new_add(lhs, rhs, location)
            ,recog)
        ) as Box<dyn ExprNode>);
...

```

All errors discovered during syntax and semantic analysis is reported, and sometimes, recovered.

```
(win/x64) → ~/code/compiler-exp git:(master) x ./target/debug/compiler-exp tests/error.cb
error: mismatched input ')' expecting {'void', 'char', 'short', 'int', 'long', 'struct', 'unsigned'}
  tests/error.cb:1:10
1  int mian();
   ^ mismatched input ')' expecting {'void', 'char', 'short', 'int', 'long', 'struct', 'unsigned'}

error: missing '{' at ';'
  tests/error.cb:1:11
1  int mian();
   ^ missing '{' at ';'

error: missing '}' at '<EOF>'
  tests/error.cb:1:12
1  int mian();
   ^ missing '}' at '<EOF>'
```

图 1

- 18 types of semantic errors
 - TypeNotFound
 - VariableNotFound
 - EntityNameConflict
 - ...
- All pretty-printed

```
error: Duplicate field a
tests/error.cb:3:9
2   int a;
   - Previously used here
3   int a;
   ^ Duplicate field a
```

图 2

```
error: Duplicate field a
tests/error.cb:4:9
2   int a;
   - Previously used here
3   char c;
4   int a;
   ^ Duplicate field a
```

图 3


```
error: Field bar not found of struct foo
  tests/error.cb:8:10
1   struct foo{
2       int a;
3       char c;
4   };
5
6   Struct foo defined here
7
8   test.bar;
    ^^^ Field bar not found of struct foo
```

图 4

```
error: Type mismatch, expected Pointer of Struct, found struct foo
tests/error.cb:8:5
8      test->bar += "!@#";
      ^^^^^ Type mismatch, expected Pointer of Struct, found struct foo
```

图 5

```
( C , Integer { signed: true, size: 8 }], location: 0..38 }, llvm
error: Type mismatch, expected Array, found struct foo
tests/error.cb:8:5
8      test[123] = 2;
      ^^^^^ Type mismatch, expected Array, found struct foo

thread 'main' panicked at 'called `Result::unwrap()` on an `Err` v
```

图 6

```
error: Type mismatch, expected Integer, found struct foo
  tests/error.cb:9:9
9   arr[test] = 2;
    ^^^^ Type mismatch, expected Integer, found struct foo
```

图 7

基于 LLVM 的类 C 语言编译器

CODE GENERATION

- Based on LLVM builder
 - another layer of type checking
- Exports LLVM IR for LLVM compiler

- Support shortcut for logical operators
 - `&&` and `||`
 - Uses Φ node in SSA graph
- Support compile-time constants
 - `int[1+2*3] a;`

基于 LLVM 的类 C 语言编译器

LIMITATION

- Incomplete support for function pointers
- No support for recursive types
 - Linked Lists