哈爾濱工業大學
HARBIN INSTITUTE OF TECHNOLOGY

编译原理

大作业

(2019 年度春季学期)

姓　　名　朱明彦

班　　号　计算机 9 班

学　　号　1160300314

学　　院　计算机学院

教　　师　辛明影

计算机科学与技术学院

目录

编译原理大作业

第 1 章 需求分析

在词法分析、语法分析和语义分析的实验基础之上，结合代码优化等技术，将之前的 Lexer，Parser 结合起来，形成一个完整的编译器前端程序。

第 2 章 文法设计

最终实现的文法以及语义动作的定义如下所示，其中红色字体为该产生式对应的语义动作。

1. Start -> P

2. P -> PStart D P |
   PStart S P | ϵ

3. PStart -> ϵ { env = new Env(env); offsetStack.push(offset); offset=0;}

4. D -> proc X id (M) DM P {pop(tableStack); pop(offset)} |
   record id P |
   T id A ; {enter(id.lexeme, T.type, offset);offset = offset + T.width;}

5. DM -> ϵ {table = mkTable(top(tableStack)); push(table); push(offset); offset = 0;}

6. A -> = F A | , id A | ϵ

7. M -> M , X id {enter(id.lexeme, X.type, offset); offset = offset + X.width; M.size = M1.size + 1;} |
   X id {enter(id.lexeme, X.type, offset); offset = offset + X.width; M.size = 1;}

8. T -> X {t = X.type; w = X.width;} C {T.type = C.type; T.width = C.width;}

9. X -> int {X.type = interger; X.width = 4;}|
   float {X.type = float; X.width = 8;} | bool | char

10. C -> [num] C {C.type = C1.type + '[' + num.value + ']'; C.width = num.value * C1.width;} | ϵ {C.type = t; C.width = w;}

11. S -> id = E ; {S.nextList = null; p = loopUp(id.lexeme); if p == null then error else gen(p, '=', E.addr);} |

    if (B) BM S N else BM S {backpatch(B.trueList, BM1.instr); backpatch(B.falseList, BM2.instr); temp = merge(S1.nextList, N.nextList); S.nextList = merge(temp, S2.nextList); } |

    while BM (B) BM S {backpatch(S1.nextList, BM1.instr); backpatch(B.trueList, BM2.instr); S.nextList = B.falseList; gen('goto', BM1.instr); } |

    call id (Elist) ; | return E ; | if (B) BM S {backpatch(B.trueList, BM.instr); S.nextList = merge(B.falseList, S1.nextList); } |

    L = E ; {gen(L.array, L.addr, '=', E.addr)}

12. N -> ϵ {N.nextList = makeList(nextInstr); gen('goto'); }

13. L -> L [E] {L.array = L1.array; L.type = L1.type.elem; L.width = L.type.width; t = new Temp(); L.addr = new Temp(); gen(L.addr, '=', E.addr, '*', L.width); gen(L.addr, '=', L1.addr, '+', t); } |

    id [E] {p = lookUp(id.lexeme); if p == null then error else L.array = p; L.type = id.type; L.addr = new Temp(); gen(L.addr, 'addr', E.addr, '*', L.width)}

14. E -> E + G {E.addr = newTemp(); gen(E.addr, '=', E1.addr, '+', G.addr);} |

    G {E.addr = G.addr;}

15. G -> G * F {G.addr = newTemp(); gen(G.addr, '=', G1.addr, '*', F.addr);}|

    F {G.addr = F.addr;}

16. F -> (E) {F.addr = E.addr;} |

    num {F.addr = num.value;} |

    id {F.addr = lookup(id.lexeme); if F.addr == null then error;}| real {F.addr = real.value;}| string | L {F.addr = L.array + '[' + L.addr']'}

17. B -> B || BM H {backpatch(B1.falseList, BM.instr); B.trueList = merge(B1.trueList, H.trueList); B.falstList = H.falstList;} |

    H {B.trueList = H.trueList; B.falseList = H.falseList;}

18. H -> H && BM I {backpatch(H1.trueList, BM.instr); H.trueList = I.trueList; H.falseList = merge(H1.falseList, I.falseList);} |

    I {H.trueList = I.trueList; H.falseList = I.falseList;}

```
19. I -> ! I {I.trueList = I1.falseList; I.falseList = I1.falseList;} |
         ( B ) {I.trueList = B.trueList; I.falseList = B.falseList;} |
         E Relop E {I.trueList = makeList(nextInstr); I.falseList = makeList(nextInstr
    + 1); gen('if', E1.addr, Relop.op, E2.addr, 'goto'); gen('goto');} |
         true {I.trueList = makeList(nextInstr); gen('goto');} |
         false {I.falseList = makeList(nextInstr); gen('goto');}

20. BM -> ϵ {BM.instr = nextInstr}

21. Relop -> < | <= | > | >= | == | != {Relop.op = op}

22. Elist -> Elist , E {Elist.size = Elist1.size + 1;} | E {Elist.size = 1;}
```

第 3 章 系统设计

3.1 编译器框架

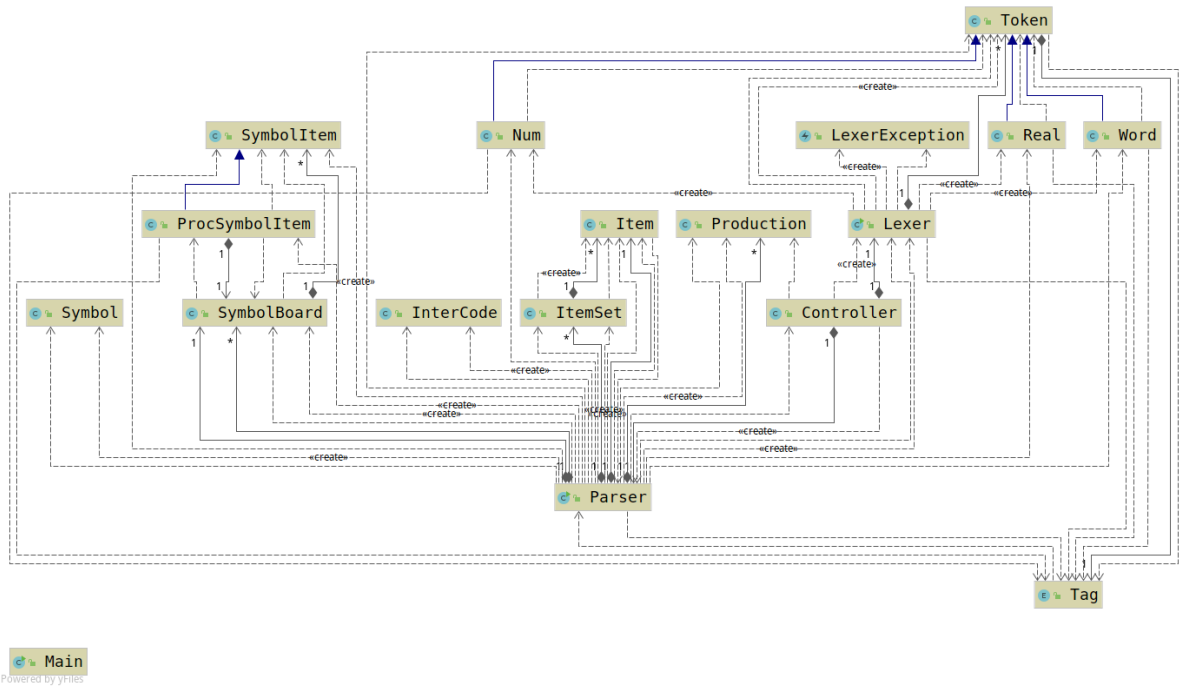

图 3.1: 编译器框架图

对于大作业最终实现的编译器，其 UML 图如3.1所示，下面分别对其中不同的实现进行阐述。

- `Lexer` 为整个编译器的词法分析器部分，其可以将输入的类 C 语言源文件转化为 Token 序列，并作为下一步语法分析器的输入。
- `Parser` 为整个编译器的语法分析器部分，在本项目中实现了 LR(1) 分析法，并在语法分析的同时进行语义动作，即所有的语义动作作用的属性均为 S 属性 [1]，最终结果为中间代码。
- `Controller` 为整个编译器 GUI 的控制部分，其使用 `JavaFX` 实现。
- `Main` 为整个编译器的主程序部分，启动 GUI 形式的编译器，调用 `Main` 对应的主函数即可。
- `Token, Real, Num, Word` 分别为 `Lexer` 输出的 Token 的父类、浮点型常数类、整数常数类以及标识符和字符串型常数类的父类。
- `Tag` 为标注 Token 类别的枚举类型。
- `LexerException` 为发现词法分析错误抛出的异常类。
- `Production` 为语法中的产生式类，分别记录产生式左部和右部。
- `Item, ItemSet` 分别为项目类和项目集类，其中项目类即在进行 LR(1) 语法分析时，对应的“项目”概念的类，即文法中的一个产生式和位于它的右部中某处的点组成；项目集类，则是项目的集合。
- `InterCode` 为 `Parser` 最终结果对应的中间代码类。
- `SymbolItem, ProcSymbolItem` 分别对应符号表中表项的父类和符号表中方法表项类，其区别为方法表项中包含其对应的局部符号表。
- `SymbolBoard` 为符号表类，其包含的表项均为 `SymbolItem` 类。
- `Symbol` 为产生式中的符号类，或者称为非终结符类，其在产生式中的右部时可能会包含 S 属性，方便语义分析时的语义动作处理。

3.2 核心数据结构

在本项目的所实现的编译器中，所用的数据结构有栈、队列、集合、`Map` 和多维数组，其中栈为最重要的数据结构。下面分别阐述各个数据结构在编译器中起到的作用。

- **栈 (Stack)**，作为核心数据结构，在 `Parser` 中使用栈记录调用关系，包括记录 offset 和符号表。
- **队列 (Queue)**，在 `Lexer` 中作为输入缓冲实现，其作用是进行一些有二义符号，如 >, >=。

- **集合 (Set)**，其多次出现在本项目的实现中，分别作为 `Lexer` 中各种符号的记录，`Parser` 中产生式、终结符、非终结符以及项目集族的记录。
- **Map**，其多次出现在本项目实现的 Compiler 中，如记录标识符与其对应的符号表条目、LR(1) 分析法中记录非终结符的 First 集以及每个项目闭包的 GOTO 表。
- **Array**，主要用于记录 LR(1) 分析表。

3.3 主要函数功能

3.4 核心流程图

第 4 章 系统实现

4.1 源文件

4.2 语法分析表

4.3 产生式序列

4.4 中间代码

4.5 附加：代码优化

4.6 GUI 展示

A 参考文献

参考文献

[1] Aho A V, Sethi R, Ullman J D. Compilers, principles, techniques[J]. Addison wesley, 1986, 7(8): 9.