Table of Contents

[1. The Compiler Phases 9](#_Toc481936105)

[1.1.1. The Preprocessor 10](#_Toc481936106)

[1.1.2. Scanning and Parsing 10](#_Toc481936107)

[1.1.1. Syntax Tree Optimization 10](#_Toc481936108)

[1.1.2. Middle Code Generation 10](#_Toc481936109)

[1.1.3. Middle Code Optimization 11](#_Toc481936110)

[1.1.4. Register Preparation and Allocation 12](#_Toc481936111)

[1.1.5. Register Allocation 12](#_Toc481936112)

[1.1.6. Object Code Generation 12](#_Toc481936113)

[1.1.1. Object Code Optimization 12](#_Toc481936114)

[1.1.2. Linking 12](#_Toc481936115)

[1.1.3. The Standard Library 12](#_Toc481936116)

[2. An Introduction to Bison and JFlex 13](#_Toc481936117)

[2.1.1. The Language 13](#_Toc481936118)

[2.1.2. The Grammar 13](#_Toc481936119)

[2.1.3. Bison 15](#_Toc481936120)

[2.1.4. JFlex 19](#_Toc481936121)

[2.1.5. Main 22](#_Toc481936122)

[3. Main 24](#_Toc481936123)

[3.1. Reading the Source File 25](#_Toc481936124)

[3.2. Checking the Object File 26](#_Toc481936125)

[3.3. Reading the Object File 26](#_Toc481936126)

[3.4. The main method 26](#_Toc481936127)

[4. The Preprocessor 29](#_Toc481936128)

[4.1. The Expression Parser 29](#_Toc481936129)

[4.1.1. The Grammar 29](#_Toc481936130)

[4.1.2. The Scanner 33](#_Toc481936131)

[4.2. The Preprocessor 36](#_Toc481936132)

[4.2.1. Tri Graphs 37](#_Toc481936133)

[4.2.1. Comments, Strings, and Characters 38](#_Toc481936134)

[4.2.2. Slash Codes 40](#_Toc481936135)

[4.2.3. The Line List 43](#_Toc481936136)

[4.2.4. Lines 46](#_Toc481936137)

[4.2.5. Includes 46](#_Toc481936138)

[4.2.6. Macros 47](#_Toc481936139)

[4.2.7. Conditional Programming 51](#_Toc481936140)

[4.2.8. Macro Expansion 53](#_Toc481936141)

[4.2.9. String Merging 56](#_Toc481936142)

[5. Scanning 58](#_Toc481936143)

[5.1.1. The typedef-name Problem 58](#_Toc481936144)

[5.2. The Scanner 58](#_Toc481936145)

[6. Parsing 64](#_Toc481936146)

[6.1. Declarations 65](#_Toc481936147)

[6.1.1. Function Definitions 65](#_Toc481936148)

[6.1.2. Declarations 69](#_Toc481936149)

[6.1.3. Struct and Union 75](#_Toc481936150)

[6.1.4. Enum 77](#_Toc481936151)

[6.1.5. Declarator 78](#_Toc481936152)

[6.1.6. Pointers 82](#_Toc481936153)

[6.1.7. Parameter List 84](#_Toc481936154)

[6.1.8. Initializer 85](#_Toc481936155)

[6.1.9. Type Name 86](#_Toc481936156)

[6.1.10. Direct and Abstract Declarator 86](#_Toc481936157)

[6.2. Statements 90](#_Toc481936158)

[6.2.1. The Forward-Jump Problem (Backpatching) 90](#_Toc481936159)

[6.2.2. The if-else Problem (Open and Closed Statements) 92](#_Toc481936160)

[6.2.3. Header Rules 93](#_Toc481936161)

[6.2.4. If Statements 94](#_Toc481936162)

[6.2.5. Switch Statements 94](#_Toc481936163)

[6.2.6. Case Statements 95](#_Toc481936164)

[6.2.7. While Statements 96](#_Toc481936165)

[6.2.8. For Statements 97](#_Toc481936166)

[6.2.9. Label Statements 97](#_Toc481936167)

[6.2.10. Closed statement 98](#_Toc481936168)

[6.2.11. Do Statements 98](#_Toc481936169)

[6.2.12. Block Statements 99](#_Toc481936170)

[6.2.13. Default Statements 99](#_Toc481936171)

[6.2.14. Break Statements 100](#_Toc481936172)

[6.2.1. Continue Statements 100](#_Toc481936173)

[6.2.2. Goto Statements 100](#_Toc481936174)

[6.2.3. Return Statements 101](#_Toc481936175)

[6.2.4. Expression Statements 102](#_Toc481936176)

[6.2.1. Declaration Statements 102](#_Toc481936177)

[6.2.2. Load Register Statements 102](#_Toc481936178)

[6.2.3. Store Register Statements 103](#_Toc481936179)

[6.2.4. Clear Registers Statements 104](#_Toc481936180)

[6.2.1. Store Flagbyte Statements 104](#_Toc481936181)

[6.2.1. Jump Register Statements 104](#_Toc481936182)

[6.2.2. Interrupt Statements 105](#_Toc481936183)

[6.3. Expressions 105](#_Toc481936184)

[6.3.1. Precedence and Associativity 105](#_Toc481936185)

[6.3.2. Syntax Tree Generation Methods 106](#_Toc481936186)

[6.3.3. Comma Expression 106](#_Toc481936187)

[6.3.4. Logical Expression 107](#_Toc481936188)

[6.3.5. Integer Expressions 108](#_Toc481936189)

[6.3.6. Integral and Floating Promotion 109](#_Toc481936190)

[6.3.7. Arithmetic Conversion 109](#_Toc481936191)

[6.3.8. Assignment Expression 110](#_Toc481936192)

[6.3.9. Conditional Expression 110](#_Toc481936193)

[6.3.10. Constant Expression 111](#_Toc481936194)

[6.3.11. Logical Expression 111](#_Toc481936195)

[6.3.12. Bitwise Expression 112](#_Toc481936196)

[6.3.13. Equality Expressions 112](#_Toc481936197)

[6.3.14. Relation Expressions 113](#_Toc481936198)

[6.3.15. Shift Expression 114](#_Toc481936199)

[6.3.16. Arithmetic Expression 115](#_Toc481936200)

[6.3.17. Cast Expression 116](#_Toc481936201)

[6.3.18. Prefix Expression 116](#_Toc481936202)

[6.3.19. Unary Arithmetic Expression 117](#_Toc481936203)

[6.3.1. Prefix Increment and Decrement 117](#_Toc481936204)

[6.3.2. Logical Not 117](#_Toc481936205)

[6.3.3. Bitwise Not 118](#_Toc481936206)

[6.3.4. sizeof 118](#_Toc481936207)

[6.3.5. Address Expression 119](#_Toc481936208)

[6.3.6. Dereference Expression 119](#_Toc481936209)

[6.3.7. Postfix Expression 119](#_Toc481936210)

[6.3.8. Postfix Increment and Decrement 120](#_Toc481936211)

[6.3.9. Dot 120](#_Toc481936212)

[6.3.10. Arrow 121](#_Toc481936213)

[6.3.11. Index 121](#_Toc481936214)

[6.3.12. Function Call 122](#_Toc481936215)

[6.3.1. Parameter Argument List 123](#_Toc481936216)

[6.3.2. Primary Expression 124](#_Toc481936217)

[7. The Type System 125](#_Toc481936218)

[7.1. The Type Class 125](#_Toc481936219)

[7.1.1. Arithmetic Types 125](#_Toc481936220)

[7.1.2. Enumeration Types 126](#_Toc481936221)

[1.1.1. Bitfield Types 126](#_Toc481936222)

[7.1.3. Logical Types 126](#_Toc481936223)

[7.1.4. Pointer Types 127](#_Toc481936224)

[7.1.5. Array Types 127](#_Toc481936225)

[7.1.6. Function Types 128](#_Toc481936226)

[7.1.7. Struct and Union Types 129](#_Toc481936315)

[Type 130](#_Toc481936316)

[7.1.8. Size 130](#_Toc481936354)

[7.1.9. Completeness 131](#_Toc481936355)

[7.1.1. Constant and Volatile Types 132](#_Toc481936356)

[7.1.2. Equality 132](#_Toc481936357)

[7.1.3. Test Methods 133](#_Toc481936398)

[7.2. Type Casting 136](#_Toc481936399)

[7.3. Type Promotion 140](#_Toc481936400)

[7.4. The Value Conversion Class 142](#_Toc481936401)

[8. The Symbol Table 147](#_Toc481936402)

[8.1. The Symbol Class 147](#_Toc481936403)

[8.1.1. Symbol Storage 147](#_Toc481936404)

[8.2. The Symbol Table Class 158](#_Toc481936405)

[8.2.1. Add Symbols 162](#_Toc481936406)

[8.2.2. Looking Up Symbols 162](#_Toc481936407)

[1.1.1. Generate Offset 163](#_Toc481936408)

[1.1.1. Table Text 163](#_Toc481936409)

[9. The Syntax Tree 164](#_Toc481936410)

[10. Syntax Tree Optimization 167](#_Toc481936411)

[11. The Middle Code 183](#_Toc481936412)

[12. Middle Code Generation 194](#_Toc481936413)

[13. Initializer 216](#_Toc481936414)

[13.1.1. Static Initialization 216](#_Toc481936415)

[13.1.2. Stack Initialization 228](#_Toc481936416)

[14. Middle Code Optimization 233](#_Toc481936417)

[14.1. Optimization 233](#_Toc481936418)

[14.1.1. Next Goto Statements 234](#_Toc481936419)

[14.1.2. Next-Double Goto Statements 234](#_Toc481936420)

[14.1.3. Goto Chains 234](#_Toc481936421)

[14.1.4. Clear Unreachable Code 235](#_Toc481936422)

[14.1.5. Remove Cleared Code 236](#_Toc481936423)

[14.2. Generation 237](#_Toc481936424)

[14.2.1. Base Blocks Generation 237](#_Toc481936425)

[14.2.2. Live Sets Generation 239](#_Toc481936426)

[14.2.3. Path Sets 241](#_Toc481936427)

[14.2.4. Removal of Unused Code 244](#_Toc481936428)

[15. Object Code Preparation 245](#_Toc481936429)

[15.1. Register Preparation 245](#_Toc481936430)

[15.2. Register Allocation 265](#_Toc481936431)

[16. Object Code Generation 269](#_Toc481936432)

[16.1. The Object Operator 269](#_Toc481936433)

[16.2. The Object Code 269](#_Toc481936434)

[16.2.1. Object Code Generation 283](#_Toc481936435)

[16.3. Object Code Generation 283](#_Toc481936436)

[17. The Linker 322](#_Toc481936437)

[17.1. The Linker Class 322](#_Toc481936438)

[17.1.1. Generation 323](#_Toc481936439)

[17.1.2. Main Trace Generation 325](#_Toc481936440)

[18. Auxiliary Classes 329](#_Toc481936441)

[18.1. Error Handling 329](#_Toc481936442)

[18.2. Container Classes 330](#_Toc481936443)

[18.2.1. Ordered Pair 330](#_Toc481936444)

[18.2.2. Unordered Pair 331](#_Toc481936445)

[18.2.3. Triple 332](#_Toc481936446)

[18.2.4. Tuple 333](#_Toc481936447)

[18.3. Undirected Graph 333](#_Toc481936448)

[18.3.1. Addition and Removal of Vertices and Edges 334](#_Toc481936449)

[18.3.2. Graph Partition 335](#_Toc481936450)

[18.4. Directed Graph 336](#_Toc481936451)

[19. The Standard Library 339](#_Toc481936452)

[19.1. Integral and Floating Limits 339](#_Toc481936453)

[19.2. The Assert Macro 340](#_Toc481936454)

[19.3. Locale Data 340](#_Toc481936455)

[19.4. Character Types 342](#_Toc481936456)

[19.5. Strings 344](#_Toc481936457)

[19.6. Long Jumps 350](#_Toc481936458)

[19.7. Mathematical Functions 351](#_Toc481936459)

[19.8. Standard Input/Output 359](#_Toc481936460)

[19.8.1. Printing 359](#_Toc481936461)

[19.8.2. Scanning 371](#_Toc481936462)

[19.8.3. File Handling 381](#_Toc481936463)

[19.9. The Standard Library 391](#_Toc481936464)

[19.10. Time 396](#_Toc481936465)

[20. Main.java 404](#_Toc481936466)

[21. The C Grammar 405](#_Toc481936467)

[21.1. The Preprocessor Grammar 405](#_Toc481936468)

[21.2. The Language Grammar 405](#_Toc481936469)

[22. A Crash Course in C 412](#_Toc481936470)

[22.1. The Compiler and the Linker 412](#_Toc481936471)

[22.2. The Hello-World Program 413](#_Toc481936472)

[22.3. Comments 414](#_Toc481936473)

[1.1.1 Types and Variables 414](#_Toc481936474)

[2.1.1 Simple Types 415](#_Toc481936475)

[3.1.1 Variables 415](#_Toc481936476)

[4.1.1 Constants 416](#_Toc481936477)

[5.1.1 Output 416](#_Toc481936478)

[6.1.1 Input 417](#_Toc481936479)

[7.1.1 Enumerations 418](#_Toc481936480)

[8.1.1 Arrays 419](#_Toc481936481)

[9.1.1 Characters and Strings 419](#_Toc481936482)

[10.1.1 The ASCII Table 420](#_Toc481936483)

[11.1.1 Pointers 420](#_Toc481936484)

[12.1.1 Pointers and Dynamic Memory 421](#_Toc481936485)

[13.1.1 Defining our own types 423](#_Toc481936486)

[14.1.1 The Type Size 424](#_Toc481936487)

[22.4. Expressions and Operators 424](#_Toc481936488)

[15.1.1 Arithmetic Operators 424](#_Toc481936489)

[16.1.1 Pointer Arithmetic 425](#_Toc481936490)

[17.1.1 Increment and decrement 426](#_Toc481936491)

[18.1.1 Relational Operators 426](#_Toc481936492)

[19.1.1 Logical Operators 426](#_Toc481936493)

[20.1.1 Bitwise Operators 427](#_Toc481936494)

[21.1.1 Assignment 428](#_Toc481936495)

[22.1.1 The Condition Operator 428](#_Toc481936496)

[23.1.1 Precedence and Associatively 428](#_Toc481936497)

[22.5. Statements 429](#_Toc481936498)

[24.1.1 Selection statements 429](#_Toc481936499)

[25.1.1 Iteration statements 431](#_Toc481936500)

[26.1.1 Jump statements 433](#_Toc481936501)

[27.1.1 Expression statements 433](#_Toc481936502)

[28.1.1 Volatile 433](#_Toc481936503)

[22.6. Functions 433](#_Toc481936504)

[22.6.1. Void Functions 435](#_Toc481936505)

[22.6.2. Local and global variables 435](#_Toc481936506)

[22.6.3. Inner Blocks with Local Variables 436](#_Toc481936507)

[22.6.4. Call-by-Value and Call-by-Reference 437](#_Toc481936508)

[22.6.5. Static, Extern, and Register Variables 439](#_Toc481936509)

[22.6.6. Recursion 440](#_Toc481936510)

[22.6.7. Definition and Declaration 440](#_Toc481936511)

[22.6.8. Higher-Order Functions 441](#_Toc481936512)

[22.7. Structures and Linked Lists 443](#_Toc481936513)

[22.7.1. Stacks and Linked Lists 443](#_Toc481936514)

[22.7.2. Unions 446](#_Toc481936515)

[22.7.3. Bitfields 446](#_Toc481936516)

[22.8. Structures with function pointers 447](#_Toc481936517)

[22.9. The Preprocessor 447](#_Toc481936518)

[22.10. The Standard Library 448](#_Toc481936519)

[22.10.1. File Management 448](#_Toc481936520)

[22.10.2. Program Parameters 450](#_Toc481936521)

[22.10.3. Environment Functions 450](#_Toc481936522)

[22.10.4. Searching and Sorting 451](#_Toc481936523)

[22.10.5. Functions with Variable Number of Parameters 452](#_Toc481936524)

[22.10.6. String Management 452](#_Toc481936525)

[22.10.7. Memory Management 454](#_Toc481936526)

[22.10.8. Mathematical Functions 454](#_Toc481936527)

[22.10.9. Long Jumps 454](#_Toc481936528)

[22.10.10. Time 455](#_Toc481936529)

[22.10.11. Random Numbers 456](#_Toc481936530)

[22.10.12. Limits of Integral and Floating Types 456](#_Toc481936531)

[22.10.13. Character Functions 457](#_Toc481936532)

[22.11. Further Reading 458](#_Toc481936533)

[23. A Crash Course in Java 459](#_Toc481936534)

[Foreword 460](#_Toc481936535)

# The Compiler Phases

Register Preparator

Register Allocator

Parser

Middle Code with Register Information

Object Code Generation

Object Code Optimizer

Object Code

Optimized Object Code

Middle Code with Allocated Registers

### The Preprocessor

First, we need to perform preprocessing: comments are removed, files are included, macros (with or without parameters) are expanded, the conditional programming is evaluated.

### Scanning and Parsing

The scanner is responsible for putting together sequences of characters into tokens, which is the least significant parts of the source code. For instance, the characters ‘i’ and ‘f’ are interpreted as the key word if and the characters ‘1’, ‘2’, and ‘3’ are interpreted as the integer 123.

Every programming language has a syntax, which in some cases is defined by a grammar. The C grammar of this books is described in Chapter 21. The parser checks whether the source code applies to the grammar by requesting tokes from the scanner when needed. The C grammar can be divided into three parts: declarations and definitions, statements and expressions. When the declarations are parsed the symbol table is generated, which holds information of the declared variables, types, and functions that is used when parsing the statements and expressions. During the parsing, comprehensive type checking and type conversions are also performed to determine whether the source code is semantically correct.

The output of the parser is a syntax holding the structure of the program, with references to the symbols of the symbol table.

### Syntax Tree Optimization

The purpose of the syntax tree optimization phase is to make it as small as possible. To some extent, it reduces the tree caused by the code of the programmer. In the following example, the side effects (function call and increment) are kept, but the addition is removed.

|  |  |
| --- | --- |
| f(x) + i++; | f(x); i++; |
| (a) Before | (b) After |

However, in many cases the it is subtrees added by the parser that are removed. For instance, the C standard function printf returns an integer, which in many cases are ignored. The parser stores the value in a temporary variable, which is removed in case it is not used.

|  |  |
| --- | --- |
| temp = printf(“Hello”); | printf(“Hello”); |
| (a) Before | (b) After |

### Middle Code Generation

When the syntax tree has been appropriate optimized, is translated into middle code list. Each middle code instruction is made of three-address-code, holding one operator and three addresses. Each address may refer to a name, a symbol, a value, or a jump address.

ymbols and values. As the name implies, the instruction it is The output of sequence of middle code. That is, a simple code format suitable for further optimization and object code generation. In this book, we use three-address-code. As the name implies, the code is made up of an operator and three addresses, which are symbols or jump targets.

### Middle Code Optimization

The purpose of the middle code optimization phase is to make the middle code more effective. One example is the next-jump instruction. Naturally, an instruction that jumps to the next instruction shall be removed:

|  |  |
| --- | --- |
| 1. goto 2  2. ... | 1. empty  2. ... |
| (a) Before | (b) After |

In the same way, if a conditional instruction jump can be reduced:

|  |  |
| --- | --- |
| 1. if a < b goto 3  2. goto 10  3. ... | 1. if a >= b goto 10  2. empty  3. ... |
| (a) Before | (b) After |

Goto-chains are also reduced. In the following example, all goto instructions jumps to the final target:

|  |  |
| --- | --- |
| 1. goto 3  2. ...  3. goto 5  4. ...  5. goto 10 | 1. goto 10  2. ...  3. goto 10  4. ...  5. goto 10 |
| (a) Before | (b) After |

Finally, unreachable code is reduced. That is, every instruction that cannot be reached from the first instruction of the function is removed:

if (0) printf(“False”); else printf(“True”);

|  |  |
| --- | --- |
| 1. goto 2  2. // *call printf(“False”);*  3. goto 5  4. // *call printf(“True”);*  5. ... | 1. empty  2. // *call printf(“False”);*  3. empty  4. empty  5. ... |
| (a) Before | (b) After |

simplify the middle code. When the middle code has been generated, the functions holds code that is not reachable and variables not used, introduced both by the programmer and the scanning and parsing phase above, that need to be removed. Moreover, there are also complicated jump instruction that can be simplified.

This phase is also responsible for the generation of base blocks and live sets. A base block is the longest sequence of middle code instructions that is not the target of a jump instruction. A live set is a set connected to each middle code instruction that holds the variable accessed later on in the same base block, that information is used when generating object code in the next phase.

### Register Preparation and Allocation

Before we do the actual register allocation, we traverse the middle code list and track the use of (so far anonyms) registers. The result is a set of register usages that forms the vertex set of a graph. An edge is added for each pair of usages that overlaps. Then the registers are allocated by a graph-coloring algorithm.

### Register Allocation

A graph is a mathematical structure made up by a non-empty set of vertices, where some pairs are connected by edges. Two vertices connected by an edge are neighborhoods. A graph can be colored in such way that that no neighbors have the same color. The color number of a graph is the minimum number of colors needed to color the graph. When allocating the registers we use a graph, where the tracks are the vertices and two vertices are connected by an edge if the tracks overlaps. Then we use graph coloring to assign each track a register.

### Object Code Generation

When the registers have been allocated, we are ready to generate the object code. In this book, there are actually two versions of object code. This phase generates both 16-bits executable com-code and assembler code that can be further assembled into com-code. The reason for the generation of both assembler code and com-code is that it is virtually impossible to debug the generated com-code, it is much easier to study the assembler code instead. When generating the target code we use the registers allocated, we also use registers holding address to values.

### Object Code Optimization

When the object code has been generated, it needs to be optimized. Foremost, we need to consider the conditional jump instructions. An unconditional jump instruction may be eight of sixteen bits, while a condition jump is always eight bits. We also translates addition or subtraction with one into incremement or decrement, and we reduse pop-push-chains.

### Linking

When the source files have compiled into object files they need to be linked together into an executable file. The task of the linker is to modify all references to static and extern variables and functions as well as make sure that only the parts reachable from the main functions are included in the final executable file.

### The Standard Library

The C language comes with a set of functions, macros, and instructions for many services. It is mostly implemented in C. However, there are some additional non-standard functions for accessing registers.

# An Introduction to Bison and JFlex

In this chapter, we use the JFlex scanner generator and the Bison parser generator, and construct a simple programming language.

### The Language

A program is made up by a non-empty sequence of statements, each terminated by a semicolon. There are five kinds of statements:

1. Read. A non-empty sequence of variables are assigned values read from the input stream, with an optional prompt.
2. Write. The non-empty sequence of values are is written to the output stream, with an optional text.
3. Assign. A variable is assigned the value of an expression.

The mathematical constants pi and e are stored in the variable map when the execution starts. An expression can be made up of the four rules of arithmetic, parentheses as well as the functions sin, cos, tan, log, exp, log10, exp10, and sqrt (square root). Division by zero, invalid function arguments, or invalid input generate error messages.

Below is an example:

// The area and circumference of a circle

read "Input the radius of a circle: ", radius;

assign area = pi \* radius \* radius;

assign circumference = 2 \* pi \* radius;

write "The area and circumference are: ", area, circumference;

newline;

// Fahrenheit to Celsius

read "Input a temperature in degrees Fahrenheit: ", farenheit;

assign celsius = (farenheit - 32) / 1.8;

write "In Celsius degrees: ", celsius;

newline;

// Pythagorean Theorem

read "Input the two cathetuses of a right-angled triangle: ", a, b;

assign c = sqrt(a \* a + b \* b);

write "The hypothenuse is: ", c;

When executed (the underlined text represent input values):

Input the radius of a circle: 10

The area and circumference are: 62.83185307179586, 314.1592653589793

Input a temperature in degrees Fahrenheit: 212

In Celsius degrees: 100.0

Input the two cathetuses of a right-angled triangle: 3,4

The hypothenuse is: 5.0

### The Grammar

Every programming language has a syntax, which may be defined by a grammar. The grammar of our language is given below. A grammar is made up by a set of *rules*, among which one is the start rule (in this case statement\_list). The grammar rules for our language follows below, the vertical bar at the left of the rules can be read as “or”. The first rule statement\_list can be read as “a statement list is a statement or a statement followed by a statement list.”

statement\_list ::=

statement

| statement statement\_list

statement ::=

**assign** **identifier** EQUAL expression **;**

| **read** optional\_output identifier\_list **;**

| **write** optional\_output expression\_list **;**

| **newline** **;**

optional\_output ::=

/\* Empty. \*/

| **string** **,**

identifier\_list ::=

**identifier**

| **identifier** **,** identifier\_list

expression\_list ::=

expression

| expression **,** expression\_list

expression ::=

binary\_expression

| expression **+** binary\_expression

| expression **-** binary\_expression

binary\_expression ::=

unary\_expression

| binary\_expression **\*** unary\_expression

| binary\_expression **/** unary\_expression

unary\_expression ::=

primary\_expression

| **+** unary\_expression

| **-** unary\_expression

| **sin** unary\_expression

| **cos** unary\_expression

| **tan** unary\_expression

| **log** unary\_expression

| **exp** unary\_expression

| **log10** unary\_expression

| **exp10** unary\_expression

| **sqrt** unary\_expression

primary\_expression ::=

**(** expression **)**

| **identifier**

| **value**

The words and character in boldface is the end products of the grammar. The rules shall be applied in such a way that finally only tokens remain. The first line of the example in the previous section are made up of the following tokens:

**read string , identifier ;**

### Bison

Bison is a tool based on Yacc[[1]](#footnote-1), which is a classic parser generator. Even though it is possible to write a parser in pure Java, it is easier to use a tool like Bison. Simply put, Bison takes a grammar as input and generators Java code as output. However, it does not only check whether the input string comply with the grammar, each rule is also allowed to perform actions, defined by Java code.

The parser is divided into two parts: one part stating the rules and tokens of the grammar, and one part defining the grammar rules. The tokens written in boldface in the previous section are written in capital letters in CUP.

Parser.bison

%{

package calculator;

import java.io.\*;

import java.util.\*;

class ErrorMessage {

public static void print(String message) {

Main.ErrorStream.println(message);

System.exit(-1);

}

}

%}

%name-prefix = ""

The following tokens do not holds any attributes.

%token ASSIGN READ WRITE NEWLINE EQUAL PLUS MINUS TIMES DIVIDE

SIN COS TAN LOG EXP LOG10 EXP10 SQRT

LEFT\_PAREN RIGHT\_PAREN COMMA SEMICOLON

STRING, IDENTIFIER, and VALUE are tokens with attributes.

%token <String> STRING IDENTIFIER

%token <Double> VALUE

%type <List> identifier\_list

%type <List> expression\_list

%type <Double> expression binary\_expression unary\_expression primary\_expression

There are also the rules statement\_list, statement, and optional\_output in the rules section below. However, they do not return values and do not need to be defined in this section.

%start statement\_list;

%%

statement\_list:

statement

| statement statement\_list;

The right-hand side are numbered from one, and the attribute values are accessable with the dollar-notation. In the rule below, the value of the identifier is stored in $2 and the value of the expression is stored in $4.

statement:

ASSIGN IDENTIFIER EQUAL expression SEMICOLON {

Main.VariableMap.put($2, $4);

}

| READ optional\_output identifier\_list SEMICOLON {

try {

String buffer = Main.BufferedReader.readLine();

String textArray[] = buffer.split(",");

if ($3.size() != textArray.length) {

ErrorMessage.print("Invalid number of values.");

}

for (int index = 0; index < $3.size(); ++index) {

String name = (String) $3.get(index), text = textArray[index];

try {

double value = Double.valueOf(text);

Main.VariableMap.put(name, value);

}

catch (Exception exception) {

ErrorMessage.print("Invalid input.");

}

}

}

catch (IOException exception) {

ErrorMessage.print(exception.getMessage());

}

}

| WRITE optional\_output expression\_list SEMICOLON {

boolean first = true;

for (Double value : (List<Double>) $3) {

Main.OutputStream.print((first ? "" : ", ") + value);

first = false;

}

Main.OutputStream.println();

}

| NEWLINE SEMICOLON {

Main.OutputStream.println();

};

optional\_output:

/\* Empty. \*/

| STRING COMMA {

Main.OutputStream.print($1);

};

The value of the rule is stored in $$. However, the $$ value is always a reference to an Object. Therefore, we need to transform it into a list before we can call the add method.

identifier\_list:

IDENTIFIER {

$$ = new LinkedList<String>();

((List<String>) $$).add($1);

}

| IDENTIFIER COMMA identifier\_list {

$$ = $3;

((List<String>) $$).add($1);

};

expression\_list:

expression {

$$ = new LinkedList<Double>();

((List<Double>) $$).add($1);

}

| expression COMMA expression\_list {

$$ = $3;

((List<Double>) $$).add($1);

};

expression:

binary\_expression {

$$ = $1;

}

| expression PLUS binary\_expression {

$$ = $1 + $3;

}

| expression MINUS binary\_expression {

$$ = $1 - $3;

};

binary\_expression:

unary\_expression {

$$ = $1;

}

| binary\_expression TIMES unary\_expression {

$$ = $1 \* $3;

}

| binary\_expression DIVIDE unary\_expression {

if ($3 == 0) {

ErrorMessage.print("Division by Zero.");

}

$$ = $1 / $3;

};

unary\_expression:

primary\_expression {

$$ = $1;

}

| PLUS unary\_expression {

$$ = $2;

}

| MINUS unary\_expression {

$$ = -$2;

}

| SIN unary\_expression {

$$ = Math.sin($2);

}

| COS unary\_expression {

$$ = Math.cos($2);

}

| TAN unary\_expression {

$$ = Math.tan($2);

}

| LOG unary\_expression {

if ($2 <= 0) {

ErrorMessage.print("Logarithm of Non-Positive Value.");

}

$$ = Math.log($2);

}

| EXP unary\_expression {

$$ = Math.exp($2);

}

| LOG10 unary\_expression {

if ($2 <= 0) {

ErrorMessage.print("Logarithm of Non-Positive Value.");

}

$$ = Math.log10($2);

}

| EXP10 unary\_expression {

$$ = Math.pow(10, $2);

}

| SQRT unary\_expression {

if ($2 < 0) {

ErrorMessage.print("Square Root of Negativ Value.");

}

$$ = Math.sqrt($2);

};

primary\_expression:

LEFT\_PAREN expression RIGHT\_PAREN {

$$ = $2;

}

| IDENTIFIER {

if (Main.VariableMap.containsKey($1)) {

$$ = Main.VariableMap.get($1);

}

else {

ErrorMessage.print("Unknown Name: \"" + $1 + "\".");

}

}

| VALUE {

$$ = $1;

};

### JFlex

For the parser of the previous section to work properly it also needs a scanner that identifies the tokens. JFlex is a scanner-generator, which task is to identify and transform groups of characters into tokens.

First we need to Token class that handles a token generated by the scanner, it holds its identity and a potential attribute value.

Token.java

package calculator;

public class Token {

public int m\_id;

public Object m\_value;

public Token(int id) {

m\_id = id;

m\_value = null;

}

public Token(int id, Object value) {

m\_id = id;

m\_value = value;

}

public int getId() {

return m\_id;

}

public Object getValue() {

return m\_value;

}

}

When it comes to complicated tokens, JFlex applies regular expressions. Some characters have special meanings in accordance with the following table. However, their special status can be canceled by preceeding it with a backslash.

|  |  |
| --- | --- |
| **Character** | **Meaning** |
| ( and ) | Grouping subexpressions |
| [ and ] | Any character within the brackets |
| [^ and ] | Any character except the characters within the brackets |
| \* | Zero or more characters |
| + | One or moree characters |
| . | Any character except new line |

Scanner.jflex

package calculator;

%%

%public

%class Scanner

%type Token

%notunix

//%unicode

%line

%char

%eofval{

return null;

%eofval}

%yylexthrow{

java.lang.Exception

%yylexthrow}

%eofval{

return null;

%eofval}

%yylexthrow{

java.lang.Exception

%yylexthrow}

%{

public Parser.Lexer toLexer() {

return (new ScannerToLexer(this));

}

%}

In the scanner code below, a line comment is two slashes followed by zero or more occurrences of any character except newline. Since both the slash has special meaning we must insert a backslash before it for it to mean just a slash.

LINE\_COMMENT = \/\/.\*

A string is a double-quote, followed by zero or more occurrences of any character (except double-quote) followed by a double-quote.

STRING = \"[^\"]\*\"

An identifier is a capital or small letter, or a underline, followed by zero or more capital or small letters, or digits, or underlines.

IDENTIFIER = [a-zA-Z\_][a-zA-Z0-9\_]\*

A value is as least one digit, potentially followed by more digits, a dot, and even more digits.

VALUE = [0-9]+|([0-9]+\.[0-9]+)

A white space is either a space, a tabulator, a return, a newline, or a form-feed.

WHITE\_SPACE = [ \t\r\n\f]

%%

"assign" { return (new Token(Parser.ASSIGN)); }

"read" { return (new Token(Parser.READ)); }

"write" { return (new Token(Parser.WRITE)); }

"newline" { return (new Token(Parser.NEWLINE)); }

"sin" { return (new Token(Parser.SIN)); }

"cos" { return (new Token(Parser.COS)); }

"tan" { return (new Token(Parser.TAN)); }

"log" { return (new Token(Parser.LOG)); }

"exp" { return (new Token(Parser.EXP)); }

"log10" { return (new Token(Parser.LOG10)); }

"exp10" { return (new Token(Parser.EXP10)); }

"sqrt" { return (new Token(Parser.SQRT)); }

"=" { return (new Token(Parser.EQUAL)); }

"+" { return (new Token(Parser.PLUS)); }

"-" { return (new Token(Parser.MINUS)); }

"\*" { return (new Token(Parser.TIMES)); }

"/" { return (new Token(Parser.DIVIDE)); }

"(" { return (new Token(Parser.LEFT\_PAREN)); }

")" { return (new Token(Parser.RIGHT\_PAREN)); }

"," { return (new Token(Parser.COMMA)); }

";" { return (new Token(Parser.SEMICOLON)); }

The yytext method return the scanned text, which is useful with identifiers and strings.

{IDENTIFIER} {

return (new Token(Parser.IDENTIFIER, yytext()));

}

{STRING} {

String text = yytext();

return (new Token(Parser.STRING, text.substring(1, text.length() - 1)));

}

{VALUE} {

return (new Token(Parser.VALUE, Double.valueOf(yytext())));

}

{LINE\_COMMENT} {

// Empty.

}

{WHITE\_SPACE} {

// Empty.

}

Finally, the last part of the scanner is the dot rule, which catch the case not caught by any of the previous tokens (equivalent to default in a switch statement). In that case, we just exit the execution with an error message.

. {

Main.ErrorStream.println("Unknown character: \'" + yytext() + "\'.");

System.exit(-1);

}

The ScannerToLexer class is necessary to make the parser work with the scanner. It works as link between the scanner and the parser and implements the Lexer interface in the Parser class. The yylex method reads the next token from the scanner and returns it in the form preferred by the parser.

ScannerToLexer.java

package calculator;

public class ScannerToLexer implements Parser.Lexer {

private Scanner m\_scanner;

private Object m\_nextValue;

public ScannerToLexer(Scanner scanner) {

m\_scanner = scanner;

}

public int yylex() throws java.io.IOException {

try {

Token nextToken = m\_scanner.yylex();

if (nextToken != null) {

m\_nextValue = nextToken.getValue();

return nextToken.getId();

}

}

catch (Exception exception) {

exception.printStackTrace();

yyerror(exception.getMessage());

}

return 0;

}

public Object getLVal () {

return m\_nextValue;

}

public void yyerror (String message) {

Main.ErrorStream.println(message);

System.exit(-1);

}

}

### Main

Bison and JFlex generates the Parser and Scanner classes. The main class creates a scanner object which is used by the parser object. The parse method is called with invoke the parsing. The input is read from BufferedReader, the output is written to OutputStream, error messages are written to ErrorStream, and VariableMap is used to keep track of the variables read and assigned by the program.

Main.java

package calculator;

import java.io.\*;

import java.util.\*;

public class Main {

private static InputStream InputStream = System.in;

public static PrintStream OutputStream = System.out;

public static PrintStream ErrorStream = System.err;

public static BufferedReader BufferedReader = null;

public static final Map<String,Double> VariableMap = new HashMap<>();

public static void main(String args[]) {

if (args.length != 1) {

ErrorStream.printf("Usage: calculator inputfile");

}

try {

InputStreamReader reader = new InputStreamReader(Main.InputStream);

BufferedReader = new BufferedReader(reader);

InputStreamReader scannerReader =

new InputStreamReader(new FileInputStream(args[0]));

Scanner scanner = new Scanner(scannerReader);

Parser parser = new Parser(scanner.toLexer()); // bison

parser.parse();

}

catch (Exception exception) {

exception.printStackTrace();

}

}

static {

Main.VariableMap.put("pi", Math.PI);

Main.VariableMap.put("e", Math.E);

}

}

Finally, below follows a small script file that generates the Java source code for the parser and scanner.

Parser.bat

@c:

@cd "C:\Users\Stefan\Documents\Calculator\_Bison\src\calculator"

@"C:\Program Files (x86)\GnuWin32\bin\bison" --language=Java -o Parser.java Parser.bison

@"C:\Program Files\Java\jre1.8.0\_73\bin\java" -jar C:\Users\Stefan\Documents\YaccForJava\JFlex\jflex-1.6.1\lib\jflex-1.6.1.jar Scanner.jflex

# Main

The Main class handles the overall compiling and linking.

Main.java

package c\_compiler;

import java.io.\*;

import java.math.\*;

import java.util.\*;

public class Main {

public static boolean Debug2 = true;

public static boolean Warning = false;

public static boolean CheckStackHeap2 = true;

public static Map<String,Macro> MacroMap = new MyMap<>();

public static Stack<File> IncludeStack = new Stack<>();

public static Set<File> IncludeSet = new MySet<>();

public static Stack<Triple<Boolean,Boolean,PreProcessor.IfStatus>>

IfStack = new Stack<>();

public static int FileId = 0;

public static String AssemblerPrompt2 = "Assembler: ";

public static final int MarginSize2 = 20;

public static int TemporaryCount = 0;

public static String CheckStackHeapFunction = "$CheckStackHeapFunction";

public static String CheckStackHeapFunctionEllipse =

"$CheckStackHeapFunctionEllipse";

public static String StackHeapError = "$StackHeapError";

public static String StackHeapErrorEllipse = "$StackHeapErrorEllipse";

public static String StackOverflowName = "$StackOverflow";

public static String StackOverflowMessage = "Stack Overflow => Exit -1.";

public static String PathName = "$Path";

public static String PathText = "Path";

public static int ObjectCodeTableMapSize = 11;

public static final String NumberId = "#";

public static final String TemporaryId = "$";

public static final String SeparatorId = "@";

public static final char CharacterId = '~';

public static final String FileMarker = "&";

public static final String PlusMarker = "~043";

public static int BlockCount = 0;

public static Stack<Symbol> SymbolStack = new Stack<>();

public static int Line = 0;

public static Symbol FuncSymbol = null;

public static String IncludePath = null, Path = null;

public static SymbolTable CurrentTable = null;

public static List<MiddleCode> MiddleCodeList = new LinkedList<>();

public static Map<String,LinkerInfo2> LinkerMap = new MyMap<>();

public static Stack<Map<BigInteger,Integer>> CaseMapStack = new Stack<>();

public static Stack<Integer> DefaultStack = new Stack<>();

public static Stack<Set<Integer>> BreakSetStack = new Stack<>(),

ContinueSetStack = new Stack<>();

public static Map<Integer,String> GotoMap = new MyMap<>();

public static Map<String,Integer> LabelMap = new MyMap<>();

public static Stack<BigInteger> EnumValueStack = new Stack<>();

## Reading the Source File

public static void readSourceFile(File sourceFile, File preproFile,

File objectFile, File middleFile)

throws Exception {

MacroMap.clear();

IncludeSet.clear();

StringBuilder buffer = new StringBuilder();

PreProcessor.doProcess(sourceFile, buffer);

Assert.error(IfStack.isEmpty(),

"#if, #ifdef, or #ifndef without matching #endif");

PrintStream preproStream = new PrintStream(preproFile);

preproStream.print(buffer);

preproStream.close();

StringReader preprocessedReader = new StringReader(buffer.toString());

Scanner scanner = new Scanner(preprocessedReader);

try {

LinkerMap.clear();

SymbolTable.TableCount = 0;

Main.CurrentTable = new SymbolTable(null, Scope.Global);

Parser parser = new Parser(scanner);

parser.parse();

Main.CurrentTable = Main.CurrentTable.parentTable();

}

catch (Exception exception) {

exception.printStackTrace();

Assert.error(null, " Syntax error");

}

PrintStream objectStream = new PrintStream(objectFile);

for (File includeFile : IncludeSet) {

objectStream.println(includeFile.getPath());

}

objectStream.println("\nSize: " + LinkerMap.size());

for (Map.Entry<String,LinkerInfo2> entry : LinkerMap.entrySet()) {

LinkerInfo2 linkerInfo = entry.getValue();

linkerInfo.save(objectStream);

}

objectStream.close();

}

## Checking the Object File

public static boolean isObjectFileUpToDate(File sourceFile,File objectFile)

{ if (sourceFile.lastModified() > objectFile.lastModified()) {

return false;

}

try {

BufferedReader objectReader = new BufferedReader

(new InputStreamReader(new FileInputStream(objectFile)));

String line = null;

while (!(line = objectReader.readLine()).isEmpty()) {

File traceFile = new File(line);

if (traceFile.lastModified() > objectFile.lastModified()) {

return false;

}

}

return true;

}

catch (Exception e) {

return false;

}

}

## Reading the Object File

public static Map<String,LinkerInfo2> readObjectFile(File objectFile)

throws Exception {

Map<String,LinkerInfo2> linkerMap = new MyMap<>();

BufferedReader objectReader =

new BufferedReader(new InputStreamReader(new FileInputStream(objectFile)));

while (!objectReader.readLine().isEmpty()) {

// Empty.

}

String tokenArray[] = objectReader.readLine().split(" ");

int globalInfoMapSize = Integer.parseInt(tokenArray[1]);

for (int index = 0; index < globalInfoMapSize; ++index) {

LinkerInfo2 linkerInfo = new LinkerInfo2();

linkerInfo.load(objectReader);

linkerMap.put(linkerInfo.getName(), linkerInfo);

}

objectReader.close();

return linkerMap;

}

## The main method

public static void main(String args[]) {

if (args.length == 0) {

System.err.println("usage: compiler <filename>");

System.exit(-1);

}

List<String> argList = new LinkedList<>();

for (String text : args) {

argList.add(text);

}

argList.add("CType");

argList.add("ErrNo");

argList.add("File");

argList.add("Locale");

argList.add("Math");

argList.add("Printf");

argList.add("Scanf");

argList.add("SetJmp");

argList.add("StdLib");

argList.add("String");

argList.add("Temp");

argList.add("Time");

boolean rebuild = argList.remove("-r"),

print = argList.remove("-p");

CheckStackHeap2 = argList.remove("-c");

Warning = argList.remove("-w");

IncludePath = System.getenv("include");

String pathName = "C:\\Users\\Stefan\\Documents\\C\_Compiler\\stdio\\";

if (IncludePath == null) {

IncludePath = pathName;

}

try {

boolean link = false;

for (String arg : argList) {

File sourceFile = new File(pathName + arg + ".c");

File objectFile = new File(pathName + arg + ".obj");

if (rebuild || !isObjectFileUpToDate(sourceFile, objectFile)) {

File preproFile = new File(pathName + arg + ".p");

File middleFile = new File(pathName + arg + ".mid");

if (print) {

System.out.println("Compiling \"" + sourceFile.getPath() +"\".");

}

BlockCount = 0;

readSourceFile(sourceFile, preproFile, objectFile, middleFile);

link = true;

}

}

if (link) {

Path = null;

Linker2 linker = new Linker2();

for (String arg : argList) {

File objectFile = new File(pathName + arg + ".obj");

if (print) {

System.out.println("Loading \"" + objectFile.getPath() + "\".");

}

linker.addAll(readObjectFile(objectFile));

}

PathText = "C:\\D\\" + argList.get(0) + ".com";

File comFile = new File(PathText);

File asmFile = new File(pathName + argList.get(0) + ".asm");

linker.generate(comFile, asmFile);

}

else if (print) {

System.out.println(pathName + argList.get(0) +".com is up-to-date.");

}

}

catch (Exception Exception) {

Exception.printStackTrace(System.err);

System.exit(-1);

}

}

}

# The Preprocessor

Basically, the preprocessor is made up of two parts, where the first part is a scanner and a parser and the second part takes care of comments, string, and characters as well as handling macros and conditional programming.

## The Expression Parser

### The Grammar

Before we consider the preprocessor itself, we need to find a way to decide the whether the expressions given in the #if or #elif directives equals zero. To do so, we need a parser and a scanner, and we use JFlex and Bison from Chapter 2. The grammar of an expression follows below.

expression ::=

logical\_or\_expression

| logical\_or\_expression **?** expression **:** expression

logical\_or\_expression ::=

logical\_and\_expression

| logical\_or\_expression **||** logical\_and\_expression

logical\_and\_expression ::=

bitwise\_ior\_expression

| bitwise\_and\_expression **&&** bitwise\_ior\_expression

bitwise\_ior\_expression ::=

bitwise\_xor\_expression

| bitwise\_ior\_expression **|** bitwise\_xor\_expression

bitwise\_xor\_expression ::=

bitwise\_and\_expression

| bitwise\_xor\_expression **^** bitwise\_and\_expression

bitwise\_and\_expression ::=

equality\_expression

| bitwise\_and\_expression **&** equality\_expression

equality\_expression ::=

relation\_expression

| equality\_expression **==** relation\_expression

| equality\_expression **!=** relation\_expression

relation\_expression ::=

shift\_expression

| relation\_expression **<** shift\_expression

| relation\_expression **<=** shift\_expression

| relation\_expression **>** shift\_expression

| relation\_expression **>=** shift\_expression

shift\_expression ::=

add\_expression

| shift\_expression **<<** add\_expression

| shift\_expression **>>** add\_expression

add\_expression ::=

multiply\_expression

| add\_expression **+** multiply\_expression

| add\_expression **-** multiply\_expression

multiply\_expression ::=

prefix\_expression

| multiply\_expression **\*** prefix\_expression

| multiply\_expression **/** prefix\_expression

| multiply\_expression **%** prefix\_expression

prefix\_expression ::=

primary\_expression

| **+** prefix\_expression

| **-** prefix\_expression

| **~** prefix\_expression

| **!** prefix\_expression

primary\_expression ::=

**identifier**

| **value**

| **defined** **identifier**

| **defined** **( identifier )**

| **(** expression **)**

The parser is divided into two parts: one part that defines the rules and tokens, and one part that defines the grammar rules. As mentioned in Section **Fel! Hittar inte referenskälla.**, the parser defined in Bison decides whether the given code agrees with the grammar. Moreover, the parser can also evaluate a resulting value. In our case, we want to calculate the integer value of the expression to decide whether it is zero. In this parser, an identifier is a string, a value is an integer and the value of each non-terminal is also an integer (real values are not used by the preprocessor).

PreParser.bison

%{

package c\_compiler;

import java.math.\*;

import java.util.\*;

%}

%name-prefix = "Pre"

%token QUESTION\_MARK COLON LEFT\_PAREN RIGHT\_PAREN

PLUS MINUS MULTIPLY DIVIDE MODULO DEFINED

EQUAL NOT\_EQUAL LESS\_THAN LESS\_THAN\_EQUAL

GREATER\_THAN GREATER\_THAN\_EQUAL LEFT\_SHIFT

RIGHT\_SHIFT LOGICAL\_OR LOGICAL\_AND LOGICAL\_NOT

BITWISE\_XOR BITWISE\_IOR BITWISE\_AND BITWISE\_NOT

%token <String> IDENTIFIER

%token <Integer> VALUE

%type <Integer> expression logical\_or\_expression

logical\_and\_expression bitwise\_ior\_expression

bitwise\_xor\_expression bitwise\_and\_expression

equality\_expression relation\_expression

shift\_expression add\_expression

multiply\_expression prefix\_expression

primary\_expression

%start expression

// --------------------------------------------------------------------------

%%

expression:

logical\_or\_expression {

$$ = $1;

PreProcessor.PreProcessorResult = $$;

}

| logical\_or\_expression QUESTION\_MARK expression

COLON expression {

$$ = ($1 != 0) ? $3 : $5;

PreProcessor.PreProcessorResult = $$;

};

logical\_or\_expression:

logical\_and\_expression {

$$ = $1;

}

| logical\_or\_expression LOGICAL\_OR logical\_and\_expression {

$$ = (($1 != 0) || ($3 != 0)) ? 1 : 0;

};

logical\_and\_expression:

bitwise\_ior\_expression {

$$ = $1;

}

| bitwise\_and\_expression LOGICAL\_AND bitwise\_ior\_expression {

$$ = (($1 != 0) && ($3 != 0)) ? 1 : 0;

};

bitwise\_ior\_expression:

bitwise\_xor\_expression {

$$ = $1;

}

| bitwise\_ior\_expression BITWISE\_IOR bitwise\_xor\_expression {

$$ = $1 | $3;

};

bitwise\_xor\_expression:

bitwise\_and\_expression {

$$ = $1;

}

| bitwise\_xor\_expression BITWISE\_XOR bitwise\_and\_expression {

$$ = $1 ^ $3;

};

bitwise\_and\_expression:

equality\_expression {

$$ = $1;

}

| bitwise\_and\_expression BITWISE\_AND equality\_expression {

$$ = $1 & $3;

};

equality\_expression:

relation\_expression {

$$ = $1;

}

| equality\_expression EQUAL relation\_expression {

$$ = ($1 == $3) ? 1 : 0;

}

| equality\_expression NOT\_EQUAL relation\_expression {

$$ = ($1 != $3) ? 1 : 0;

};

relation\_expression:

shift\_expression { $$ = $1; }

| relation\_expression LESS\_THAN shift\_expression {

$$ = ($1 < $3) ? 1 : 0;

}

| relation\_expression LESS\_THAN\_EQUAL shift\_expression {

$$ = ($1 <= $3) ? 1 : 0;

}

| relation\_expression GREATER\_THAN shift\_expression {

$$ = ($1 > $3) ? 1 : 0;

}

| relation\_expression GREATER\_THAN\_EQUAL shift\_expression {

$$ = ($1 >=$3) ? 1 : 0;

};

shift\_expression:

add\_expression {

$$ = $1;

}

| shift\_expression LEFT\_SHIFT add\_expression {

$$ = $1 << $3;

}

| shift\_expression RIGHT\_SHIFT add\_expression {

$$ = $1 >> $3;

};

add\_expression:

multiply\_expression {

$$ = $1;

}

| add\_expression PLUS multiply\_expression {

$$ = $1 + $3;

}

| add\_expression MINUS multiply\_expression {

$$ = $1 - $3;

};

multiply\_expression:

prefix\_expression {

$$ = $1;

}

| multiply\_expression MULTIPLY prefix\_expression {

$$ = $1 \* $3;

}

| multiply\_expression DIVIDE prefix\_expression {

$$ = $1 / $3;

}

| multiply\_expression MODULO prefix\_expression {

$$ = $1 % $3;

};

prefix\_expression:

primary\_expression {

$$ = $1;

}

| PLUS prefix\_expression {

$$ = $2;

}

| MINUS prefix\_expression {

$$ = -$2;

}

| BITWISE\_NOT prefix\_expression {

$$ = ~$2;

}

An expression is cosidered to be true if it does not equal zero.

| LOGICAL\_NOT prefix\_expression {

$$ = ($2 == 0) ? 1 : 0;

};

In the preprocessor directive, every identifier is treated as the value zero.

primary\_expression:

IDENTIFIER {

$$ = 0;

}

| VALUE {

$$ = $1;

}

The defined directive works both with or without parenthesis.

|  |  |
| --- | --- |
| #if defined (x) | #if defined x |
| With parenthesis | Without parenthesis |

| DEFINED IDENTIFIER {

$$ = Main.MacroMap.containsKey($2) ? 1 : 0;

}

| DEFINED LEFT\_PAREN IDENTIFIER RIGHT\_PAREN {

$$ = Main.MacroMap.containsKey($3) ? 1 : 0;

}

| LEFT\_PAREN expression RIGHT\_PAREN {

$$ = $2;

};

### The Scanner

As mention in Section 2.1.4, tokens are the smallest parts of the language. However, the parser cannot know that, for instance, a sequence of letters is an identifier, or that a sequence of digits is an integer value. Instead, that is the task of a scanner.

PreScanner.jflex

package c\_compiler;

import java.math.\*;

%%

%public

%class PreScanner

%type Token

%notunix

//%unicode

%line

%char

%eofval{

return null;

%eofval}

%yylexthrow{

java.lang.Exception

%yylexthrow}

%eofval{

return null;

%eofval}

%yylexthrow{

java.lang.Exception

%yylexthrow}

%{

public PreParser.Lexer toLexer() {

return (new PreScannerToPreLexer(this));

}

%}

OCTAL\_VALUE = [0][0-7]\*([uU]|[sSlL]|[uU][sSlL]|[sSlL][uU])?

DECIMAL\_VALUE = [1-9][0-9]\*([uU]|[sSlL]|[uU][sSlL]|[sSlL][uU])?

HEXADECIMAL\_VALUE = [0][xX][0-9a-fA-F]+([uU]|[sSlL]|[uU][sSlL]|[sSlL][uU])?

IDENTIFIER = [a-zA-Z\_][a-zA-Z0-9\_]\*

WHITE\_SPACE = [ \t\r\n\f\0]

%%

"defined" { return (new Token(PreParser.DEFINED)); }

"?" { return (new Token(PreParser.QUESTION\_MARK)); }

":" { return (new Token(PreParser.COLON)); }

"||" { return (new Token(PreParser.LOGICAL\_OR)); }

"&&" { return (new Token(PreParser.LOGICAL\_AND)); }

"!" { return (new Token(PreParser.LOGICAL\_NOT)); }

"&" { return (new Token(PreParser.BITWISE\_AND)); }

"^" { return (new Token(PreParser.BITWISE\_XOR)); }

"|" { return (new Token(PreParser.BITWISE\_IOR)); }

"~" { return (new Token(PreParser.BITWISE\_NOT)); }

"==" { return (new Token(PreParser.EQUAL)); }

"!=" { return (new Token(PreParser.NOT\_EQUAL)); }

"<" { return (new Token(PreParser.LESS\_THAN)); }

"<=" { return (new Token(PreParser.LESS\_THAN\_EQUAL)); }

">" { return (new Token(PreParser.GREATER\_THAN)); }

">=" { return (new Token(PreParser.GREATER\_THAN\_EQUAL)); }

"<<" { return (new Token(PreParser.LEFT\_SHIFT)); }

">>" { return (new Token(PreParser.RIGHT\_SHIFT)); }

"+" { return (new Token(PreParser.PLUS)); }

"-" { return (new Token(PreParser.MINUS)); }

"\*" { return (new Token(PreParser.MULTIPLY)); }

"/" { return (new Token(PreParser.DIVIDE)); }

"%" { return (new Token(PreParser.MODULO)); }

"(" { return (new Token(PreParser.LEFT\_PAREN)); }

")" { return (new Token(PreParser.RIGHT\_PAREN)); }

In case of an octal, decimal, or hexadecimal value, we just remove the letters u (unsigned), s (short), l (long), and x (hexadecimal value marker) from the text and let the Integer class interpret the value.

{IDENTIFIER} {

return (new Token(PreParser.IDENTIFIER, yytext()));

}

{OCTAL\_VALUE} {

String text = yytext().toLowerCase().replace("x", "").replace("u", "").

replace("s", "").replace("l", "");

int value = Integer.valueOf(text, 8);

return (new Token(PreParser.VALUE, value));

}

{DECIMAL\_VALUE} {

String text = yytext().toLowerCase().replace("x", "").replace("u", "").

replace("s", "").replace("l", "");

int value = Integer.valueOf(text, 10);

return (new Token(PreParser.VALUE, value));

}

{HEXADECIMAL\_VALUE} {

String text = yytext().toLowerCase().replace("x", "").replace("u", "").

replace("s", "").replace("l", "");

int value = Integer.valueOf(text, 16);

return (new Token(PreParser.VALUE, value));

}

It is important to detect white spaces, otherwise the user would not be allowed to add any spaces to the code.

{WHITE\_SPACE} {

// Empty.

}

If we detect a character we do not recognize, we just stop the execution with an error message.

. { Assert.error(yytext(), "unknown character"); }

## The Preprocessor

The preprocessor has several tasks:

* **Tri Graphs**. When C was originally introduced, some keyboards had a limited set of keys. Therefore, a special set of double question mark character sequence was introduced, which are replaced by their modern equivalents.
* **Comments**. Each block comment or line comment is replaced by blank characters.
* **Slashes in strings and characters**. Each backslash is processed and transformed into a regular character. Thereafter, to ease the macro expansion later we transform each character into the format of a backslash followed by three octal digits.
* **Include files**. The system include files (encapsulated by ‘<’ and ‘>’) and internal include files (encapsulated by quotes) are read and included in the final code.
* **Conditional programming**. The #if, #ifdef, #ifndef, #elif, and #endif directives are processed.
* **Macro expansion**. The source code is traversed and each macro is expanded. Since the contents of each character and string constant has been translated into slash codes we do not need to check whether the macro to be expanded is part of a character of string.
* **String sequences**. Finally, every sequence of string is concatenated into one string. For instance, the sequence "ab" "cd" "ef" is concatenated into "abcded".

The first phase of the preprocessor is to read the source file and place in a text buffer (a StringBuilder object), and then replace the tri graph sequences, remove the comments and replace every character in string and characters with its equivalent backslash code. Then the buffer is divided into lines and each line starting with a preprocessor directive is evaluated and every line is search for macros to expand.

Preprocessor.java

package c\_compiler;

import java.io.\*;

import java.util.\*;

public class PreProcessor {

public static enum IfStatus {If, Else};

public static void doProcess(File file, StringBuilder buffer)

throws Exception {

readFile(file, buffer);

triGraphs(buffer);

traverse(buffer);

Main.Path = file.getPath();

Main.Line = 0;

buffer.insert(0, Main.SeparatorId + Main.Path + ",0$\n");

int stackSize = Main.IfStack.size();

List<String> lineList = generateLineList(buffer);

lineList = modifyLineList(lineList);

traverseLineList(lineList);

Assert.error(Main.IfStack.size() == stackSize,

"unbalanced #if and #endif strucure");

buffer.replace(0, buffer.length(), "");

for (String line : lineList) {

buffer.append(line + "\n");

}

mergeStrings(buffer);

replaceAll(buffer, "\0", "");

}

private static void readFile(File file, StringBuilder buffer)

throws IOException {

FileInputStream fileStream = new FileInputStream(file);

InputStreamReader streamReader = new InputStreamReader(fileStream);

BufferedReader reader = new BufferedReader(streamReader);

String line;

while ((line = reader.readLine()) != null) {

buffer.append(line.trim() + "\n");

}

reader.close();

}

### Tri Graphs

The following table shows the tri graphs character sequences and their modern equivalences:

|  |  |
| --- | --- |
| **Tri-Graph Character Sequence** | **Modern Equivalence** |
| ??= | # |
| ??/ | \ |
| ??´ | ^ |
| ??( | [ |
| ??) | ] |
| ??! | | |
| ??< | { |
| ??> | } |
| ??- | ~ |

The tri graphs sequences are easy to replace, since we do not need to take any consideration in whether the tri graphs are placed inside strings, characters, or comments.

private static void triGraphs(StringBuilder buffer) {

replaceAll(buffer, "?=", "#");

replaceAll(buffer, "?/", "\\");

replaceAll(buffer, "?'", "^");

replaceAll(buffer, "?(", "[");

replaceAll(buffer, "?)", "]");

replaceAll(buffer, "?!", "|");

replaceAll(buffer, "?<", "{");

replaceAll(buffer, "?>", "}");

replaceAll(buffer, "?-", "~");

}

private static void replaceAll(StringBuilder buffer,

String oldText, String newText) {

int oldSize = oldText.length(), index;

while ((index = buffer.indexOf(oldText)) != -1) {

buffer.replace(index, index + oldSize, newText);

}

}

### Comments, Strings, and Characters

The next step is to take care of block comments (/\* to \*/) and line comments (// to the end of the line), strings, and characters. Since they may be nested, they need to be evaluated in the same phase.

The idea is that we go through the buffer, and when we find the beginning of a line comment, a block comment, a string, or a character we call the corresponding method, which modify the buffer and returns the index of the next character to be inspected.

To make the inspection of the source code buffer easier we start by adding three zero characters, which are removed at the end of the method.

public static void traverse(StringBuilder buffer) {

buffer.append("\0\0\0");

for (int index = 0; buffer.charAt(index) != '\0'; ++index) {

if (buffer.substring(index, index + 2).equals("//")) {

index = traverseLineComment(index, buffer);

}

else if (buffer.substring(index, index + 2).equals("/\*")) {

index = traverseBlockComment(index, buffer);

}

else if (buffer.charAt(index) == '\"') {

index = traverseString(index + 1, buffer, true);

}

else if (buffer.charAt(index) == '\'') {

index = traverseCharacter(index + 1, buffer, true);

}

else {

Assert.error(buffer.charAt(index) != '$',

Main.SeparatorId, "invalid character");

}

}

replaceAll(buffer, "\0", "");

}

The traverseLineComment method is quite easy, we just continue until we find the end of the line (‘\n’) or the end of the buffer (‘\0’), and replace every preceding character with a blank.

private static int traverseLineComment(int index, StringBuilder buffer) {

while ((buffer.charAt(index) != '\n') &&

(buffer.charAt(index) != '\0')) {

buffer.setCharAt(index++, ' ');

}

return index;

}

The traverseBlockComment method is a little bit more complicated. We continue until we find the end of the comment (\*/). If we instead find the end of the buffer (‘\0’) the comment is unterminated, which result in an error message. Each character is replaced by a space character. However, newlines (‘\n’) are not replaced, in order for the line count to work properly.

private static int traverseBlockComment(int index, StringBuilder buffer) {

while (true) {

if (buffer.charAt(index) == '\0') {

Assert.error("Unfinished commant block.");

}

else if (buffer.substring(index, index + 2).equals("\*/")) {

buffer.setCharAt(index, ' ');

buffer.setCharAt(index + 1, ' ');

return (index + 1);

}

else if (buffer.charAt(index) == '\n') {

buffer.setCharAt(index++, '\n');

}

else {

buffer.setCharAt(index++, ' ');

}

}

}

The traverseString method needs to keep track of the end of the string (‘\”’), the end of the buffer (‘\0’), or newline in the string (‘\n’). The end of the buffer or newline result in error messages. Moreover, when we find a backslash (‘\\’) we call doSlash, that transforms the slash code into a regular character. Finally, when the string has been traversed each regular character is modified into a backslash followed by three octal digits, in order to ease the macro replacement process in a later phase.

public static int traverseString(int index, StringBuilder buffer, boolean octal) {

while (true) {

switch (buffer.charAt(index)) {

case '\"':

return index;

case '\n':

Assert.error("new line in string.");

break;

case '\0':

Assert.error("unfinished string.");

break;

case '\\':

slashToChar(index, buffer);

break;

}

if (octal) {

charToOctal(index, buffer);

index += 4;

}

else {

++index;

}

}

}

The traverseCharacter method is similar to traverseString above, it generates error messages for end-of-line or newline, calls doSlash for each backslash, and finally changes each regular character into a backslash followed by three octal digits.

public static int traverseCharacter(int index, StringBuilder buffer,

boolean octal) {

int charCount = 0;

while (true) {

switch (buffer.charAt(index)) {

case '\'':

Assert.error(charCount == 1, "invalid character sequence");

return index;

case '\n':

Assert.error("new line in character.");

break;

case '\0':

Assert.error("unfinished character.");

break;

case '\\':

slashToChar(index, buffer);

break;

}

if (octal) {

charToOctal(index, buffer);

index += 4;

}

else {

++index;

}

++charCount;

}

}

### Slash Codes

The slashToChar method inspect the character succeeding the slash.

If the characters following the slash are a small or capital x and one or two hexadecimal characters, or three octal the following two characters are inspected, and their value calculated and the two hexadecimal digits are replaced three octal digits. However, if the two following characters are not hexadecimal digits, an error message occurs.

public static void slashToChar(int index, StringBuilder buffer) {

char char1 = buffer.charAt(index + 1),

char2 = buffer.charAt(index + 2),

char3 = buffer.charAt(index + 3);

If the three, two, or one characters following the slash are octal digits, the value of the digits is calculated and the slash sequence is replaced by the character with the ASCII value of the octal digits. If the value of the digits exceeds 255, an error is reported.

if (isOctal(char1) && isOctal(char2) && isOctal(char3)) {

int octValue = 64 \* charToOctal(char1) +

8 \* charToOctal(char2) +

charToOctal(char3);

Assert.error(octValue < 256, "invalid octal sequence");

buffer.replace(index, index + 4, String.valueOf((char) octValue));

}

else if (isOctal(char1) && isOctal(char2)) {

int octValue = 8 \* charToOctal(char1) + charToOctal(char2);

buffer.replace(index, index + 3, String.valueOf((char) octValue));

}

else if (isOctal(char1)) {

int octValue = charToOctal(char1);

buffer.replace(index, index + 2, String.valueOf((char) octValue));

}

If the character following the slash is a small or capital x and two or one hexadecimal digits, the value of the digits is calculated and the slash sequence is replaced by the character with the ASCII value of the hexadecimal digits. A small or capital x not followed by at least one hexadecimal digit results in an error being reported.

else if ((char1 == 'x') || (char1 == 'X')) {

if (isHex(char1) && isHex(char2)) {

int hexValue = 16 \* charToHex(char1) + charToHex(char2);

buffer.replace(index, index + 3, String.valueOf((char) hexValue));

}

else if (isHex(char1)) {

int hexValue = charToHex(char1);

buffer.replace(index, index + 2, String.valueOf((char) hexValue));

}

else {

Assert.error("invalid hexadecimal sequence");

}

}

If the slash is not followed by an x of an octal digit, it may be followed by a special character. The m\_slashMap map is used to look up the corresponding character, which replaces the slash sequence.

else if (m\_slashMap.containsKey(char1)) {

buffer.replace(index, index + 2, String.valueOf(m\_slashMap.get(char1)));

}

If none of the cases above applies, an error is reported.

else {

Assert.error(buffer.charAt(index + 1), "invalid slash sequence");

}

}

private static final Map<Character,Character> m\_slashMap = new ListMap<>();

static {

m\_slashMap.put('0','\0');

m\_slashMap.put('n','\n');

m\_slashMap.put('t','\t');

m\_slashMap.put('v', (char) 11);

m\_slashMap.put('b','\b');

m\_slashMap.put('r','\r');

m\_slashMap.put('f','\f');

m\_slashMap.put('a',(char) 7);

m\_slashMap.put('\\','\\');

m\_slashMap.put('?','?');

m\_slashMap.put('\'','\'');

m\_slashMap.put('"','\"');

}

private static boolean isOctal(char c) {

return "01234567".contains(String.valueOf(c));

}

private static boolean isHex(char c) {

return "0123456789abcdefABCDEF".contains(String.valueOf(c));

}

private static int charToOctal(char c) {

return "01234567".indexOf(c);

}

private static int charToHex(char c) {

return "0123456789abcdef".indexOf(Character.toLowerCase(c));

}

When the character or string has been translated from slash codes to regular characters, charToOctal is called to translate them to octal slash codes.

public static void charToOctal(int index, StringBuilder buffer) {

int asciiValue = buffer.charAt(index);

char firstChar = "01234567".charAt(asciiValue / 64),

secondChar = "01234567".charAt((asciiValue % 64) / 8),

thirdChar = "01234567".charAt(asciiValue % 8);

String text = "\\" + String.valueOf(firstChar) +

String.valueOf(secondChar) +

String.valueOf(thirdChar) ;

buffer.replace(index, index + 1, text);

}

The octalToChar method is called by the scanner to tralform the octal shalch codes into regular characters.

public static String octalToChar(String text) {

StringBuilder buffer = new StringBuilder(text);

for (int index = (text.length() - 1); index >= 0; --index) {

if (buffer.charAt(index) == '\\') {

PreProcessor.slashToChar(index, buffer);

}

}

return buffer.toString();

}

### The Line List

When the buffer has been modified in accordance with the methods above, it is time to transform it into a list of lines to process the preprocessors directives. The transformation is simple, we just use the split method. We add an extra blank line in the last line is a macro definition that ends with two slashes (“\\”).

private static List<String> generateLineList(StringBuilder buffer) {

String lineArray[] = buffer.toString().split("\n");

List<String> lineList = new LinkedList<>();

for (String line : lineArray) {

lineList.add(line.trim());

}

lineList.add("");

return lineList;

}

However, we also need to merge lines that ends with two backslashes together:

|  |  |
| --- | --- |
| #define min(x,y) \\  (((x) < (y)) ? \\  (x) : (y)) | #define min(x,y) (((x) < (y)) ? (x) : (y)) |
| (a) Before | (b) After |

If we encounter lines ending with two backslashes, we continue to append the lines to lineBuffer. Since an empty line was added at the end of the line list in generateLineList above, there is no risk that that the last line ends with a backslash.

private static List<String> modifyLineList(List<String> lineList) {

List<String> resultList = new LinkedList<>();

for (int index = 0; index < lineList.size(); ++index) {

String line = lineList.get(index);

StringBuilder lineBuffer = new StringBuilder();

while (line.endsWith("\\\\")) {

lineBuffer.append(line.substring(0, line.length() - 2) + " ");

line = lineList.get(++index);

}

resultList.add(lineBuffer.toString() + line + "\0");

}

return resultList;

}

When traversing the lines, we check whether the line starts with a sharp (‘#’). If it does, we consider the word following the sharp sign (#). If it is a preprocessor directive, we call the corresponding method. If the line does not start with a sharp and we are in a visible part of source code, we call searchForMacros, which expands macros. If we are not in a visible part of the source code, the line is replaced by an empty line.

public static void traverseLineList(List<String> lineList)throws Exception{

for (int index = 0; index < lineList.size(); ++index) {

String line = lineList.get(index);

if (line.startsWith("$")) {

// Empty.

}

else if (line.startsWith("#")) {

line = trimLeft(line.substring(1));

if (line.startsWith("line")) {

String rest = line.substring("line".length());

lineList.set(index, doLine(trimLeft(rest)));

}

else if (isVisible() && line.startsWith("include")) {

String rest = line.substring("include".length());

String result = doInclude(rest.trim());

lineList.set(index, result);

}

else if (isVisible() && line.startsWith("error")) {

String message = line.substring("error".length()).trim();

System.err.println(message);

System.exit(-1);

}

else if (isVisible() && line.startsWith("define")) {

String rest = line.substring("define".length());

doDefine(trimLeft(rest));

lineList.set(index, "");

}

else if (isVisible() && line.startsWith("undef")) {

String rest = line.substring("undef".length());

doUndef(trimLeft(rest));

lineList.set(index, "");

}

else if (line.startsWith("ifdef")) {

String rest = line.substring("ifdef".length());

doIfDefined(trimLeft(rest));

lineList.set(index, "");

}

else if (line.startsWith("ifndef")) {

String rest = line.substring("ifndef".length());

doIfNotDefined(trimLeft(rest));

lineList.set(index, "");

}

else if (line.startsWith("if")) {

String rest = line.substring("if".length());

doIf(trimLeft(rest));

lineList.set(index, "");

}

else if (line.startsWith("elif")) {

String rest = line.substring("elif".length());

doElseIf(trimLeft(rest));

lineList.set(index, "");

}

else if (line.startsWith("else")) {

String rest = line.substring("else".length());

doElse(trimLeft(rest));

lineList.set(index, "");

}

else if (line.startsWith("endif")) {

String rest = line.substring("endif".length());

doEndIf(trimLeft(rest));

lineList.set(index, "");

}

else {

lineList.set(index, "");

}

}

else if (isVisible()) {

String result = searchForMacros(line, new Stack<String>());

lineList.set(index, result);

}

else {

lineList.set(index, "");

}

Main.Line += countChar(line, '\n') + 1;

}

}

The isVisible method checks whether we are in a visible part of the source code; that is, a part that has not been earlier omitted by a #if, #ifdef, #ifndef, #elif, or #else preprocessor directive. We use the Main.IfStack stack since conditional programming can be nested. If the source code is not visible at any nesting level (that is, anywhere in the stack), the area is not visible and we return false.

private static boolean isVisible() {

for (Triple<Boolean,Boolean,IfStatus> ifTriple : Main.IfStack) {

boolean currTrue = ifTriple.getFirst();

if (!currTrue) {

return false;

}

}

return true;

}

The trimLeft and trimRight methods simple removes all white-spaces to the left or right of the string.

private static String trimLeft(String text) {

int index = 0;

while ((index < text.length()) &&

Character.isWhitespace(text.charAt(index))) {

++index;

}

return text.substring(index);

}

private static String trimRight(String text) {

int index = text.length() - 1;

while ((index >= 0) && Character.isWhitespace(text.charAt(index))) {

--index;

}

return text.substring(0, index + 1);

}

The countChar method simple counts the number of occurrences of the character.

private static int countChar(String text, char c) {

int count = 0;

for (char c2 : text.toCharArray()) {

if (c == c2) {

++count;

}

}

return count;

}

### Lines

The doLine method handles the #line directive by setting the Main.Path and Main.Line fields and returns a text with the path and line that starts and ends with dollar signs (‘$’).

|  |  |  |
| --- | --- | --- |
| #line 100 C:\Temp\Test.c |  |  |
| $C:\Temp\Test.c,100$ |  |  |

private static String doLine(String line) {

String lineText, pathText;

int spaceIndex = line.indexOf(" ");

if (spaceIndex != -1) {

lineText = line.substring(0, spaceIndex);

pathText = line.substring(spaceIndex + 1).trim();

}

else {

lineText = line;

pathText = null;

}

try {

Main.Line = Integer.parseInt(lineText);

}

catch (NumberFormatException exception) {

Assert.error(lineText, "invalid line number");

}

if (pathText != null) {

Main.Path = pathText;

}

return Main.SeparatorId + Main.Path + "," + Main.Line + Main.SeparatorId;

}

### Includes

There are two kinds of include files: system files (‘<’ and ‘>’) and internal files (‘\”’). The systems files are included from the path given by Main.IncludePath, and the internal files are included locally.

The Main.IncludeSet set is used to track the included file so that the source file include the files shall be recompiled if it or any of its included files has been changed after the object file. The Main.IncludeStack is used to make sure that there is no circular inclusion.

The Main.Path and Main.Line fields are temporary replaced by the path and line of the included file. After the include line has been read, the original path and line is added to the included source code, in the dollar sign format (for example: “$C:\Temp\Test.c,100$”). The method returns the source code of the included file, which will replace the original include line.

private static String doInclude(String line) throws Exception {

File includeFile = null;

if (line.startsWith("<") && line.endsWith(">")) {

includeFile = new File(Main.IncludePath +

line.substring(1, line.length() - 1));

}

else if (line.startsWith("\"") && line.endsWith("\"")) {

String name = line.substring(1, line.length() - 1);

includeFile = new File(octalToChar(name));

}

else {

Assert.error(line, "invalid include format");

}

Assert.error(!Main.IncludeStack.contains(includeFile),

"nestled include directives: " + includeFile.getPath());

Main.IncludeStack.push(includeFile);

Main.IncludeSet.add(includeFile);

String oldPath = Main.Path;

int oldLine = Main.Line;

StringBuilder buffer = new StringBuilder();

doProcess(includeFile, buffer);

Main.Line = oldLine;

Main.Path = oldPath;

buffer.append(Main.SeparatorId + Main.Path + "," +

Main.Line + Main.SeparatorId);

Main.IncludeStack.pop();

return buffer.toString();

}

### Macros

The Macro class keep track of a macro, a macro has a possible empty list of parameters and a text.

Macro.java

package c\_compiler;

public class Macro {

private int m\_parameters;

private String m\_body;

public Macro(int parameters, String body) {

m\_parameters = parameters;

m\_body = body;

}

public Macro(String body) {

m\_parameters = 0;

m\_body = body;

}

public int parameters() {

return m\_parameters;

}

public String body() {

return m\_body;

}

public boolean equals(Object object) {

if (object instanceof Macro) {

Macro macro = (Macro) object;

return m\_body.equals(macro.m\_body) &&

(m\_parameters == macro.m\_parameters);

}

return false;

}

}

The doDefine method splits the line into the name of the macro, a potential parameter list, and a body. When the first character that is not a letter, digit, or an underline is reached, we check whether the line so far is a proper identifier. Then we have three cases:

1. We have reached the end of the line, in which case we have a macro with a name, but without parameters or body.

2. We have reached a space, in which case we have a macro with a name and a body, but no parameters.

3. We have reached a left parenthesis, in which case we have a list of parameters and we call doParameterDefine.

4. If none of the above applies, we generate an error message.

Preprocessor.java

public static void doDefine(String line) {

int index;

for (index = 0; Character.isLetterOrDigit(line.charAt(index)) ||

(line.charAt(index) == '\_'); ++index) {

// Empty.

}

String name = line.substring(0, index);

checkIdentifier(name);

switch (line.charAt(index)) {

case '\0': {

Macro oldMacro = Main.MacroMap.get(name),

newMacro = new Macro("");

Assert.error((oldMacro == null) || oldMacro.equals(newMacro),

name, "macro already defined");

Main.MacroMap.put(name, newMacro);

}

break;

case ' ': {

String text = line.substring(index + 1).trim();

Macro oldMacro = Main.MacroMap.get(name),

newMacro = new Macro(text);

Assert.error((oldMacro == null) || oldMacro.equals(newMacro),

name, "macro already defined");

Main.MacroMap.put(name, newMacro);

}

break;

case '(': {

String text = line.substring(index);

doParameterDefine(name, text);

}

break;

default: {

Assert.error(line, "invalid macro definition");

}

break;

}

}

The doParameterDefine method look into macros with parameters in two steps. First we extract the parameters by calling scanParameters. Then we go through the parameter list and check that the parameters are identifiers and that no parameters is repeated.

private static void doParameterDefine(String name, String text) {

List<String> paramList = new LinkedList<>();

StringBuilder buffer = new StringBuilder(text);

int lastIndex = scanParameters(buffer, 0, paramList);

buffer.delete(0, lastIndex + 1);

Set<String> paramSet = new ListSet<>();

for (int index = 0; index < paramList.size(); ++index) {

String param = paramList.get(index).trim();

checkIdentifier(param);

Assert.error(paramSet.add(param), param, "repeated macro parameter");

paramList.set(index, param);

}

Then we extract the identifiers from the body and replace each occurrence of a parameters with the text “$parameter\_index$”.

List<Pair<String,Integer>> identList =

getIdentifierList(buffer.toString());

for (int index = (identList.size() - 1); index >= 0; --index) {

Pair<String,Integer> identPair = identList.get(index);

String ident = identPair.getFirst();

int identIndex = identPair.getSecond();

int paramIndex = paramList.indexOf(ident);

if (paramIndex != -1) {

buffer.replace(identIndex, identIndex + ident.length(),

Main.SeparatorId + paramIndex + Main.SeparatorId);

}

}

Finally, we look for the merge operator ##. When we find one, we remove the operator and trim the text to its left and right.

int index;

String body = buffer.toString();

while ((index = body.indexOf("##")) != -1) {

String left = body.substring(0, index),

right = body.substring(index + 2);

body = left.trim() + right.trim();

}

It is allowed to redefine a macro if it is has the same parameters list and macro.

Macro oldMacro = Main.MacroMap.get(name),

newMacro = new Macro(paramList.size(), body);

Assert.error((oldMacro == null) || oldMacro.equals(newMacro),

name, "macro already defined");

Main.MacroMap.put(name, newMacro);

}

An identifier is a text starting with a letter or an underline and continuing with letters, digits, or underlines. If the given name is not an identifier, an error occurs.

private static void checkIdentifier(String name) {

Assert.error(!name.isEmpty() && (Character.isLetter(name.charAt(0)) ||

(name.charAt(0) == '\_')),

name, "invalid identifier");

for (int index = 1; index < name.length(); ++index) {

Assert.error(Character.isLetterOrDigit(name.charAt(index)) ||

(name.charAt(index) == '\_'),

name, "invalid identifier");

}

}

The getIdentifierList method extracts identifiers from a text and returns a list holding the identifiers and their indexes in the text.

private static List<Pair<String,Integer>> getIdentifierList(String text) {

StringBuilder buffer = new StringBuilder(text);

List<Pair<String,Integer>> identList = new LinkedList<>();

if (!text.isEmpty()) {

int index = 0;

while (buffer.charAt(index) != '\0') {

char c = buffer.charAt(index);

if (Character.isLetter(buffer.charAt(index)) ||

(buffer.charAt(index) == '\_')) {

int nextIndex = index;

while (Character.isLetterOrDigit(buffer.charAt(nextIndex)) ||

(buffer.charAt(nextIndex) == '\_')) {

++nextIndex;

}

String ident = buffer.substring(index, nextIndex);

identList.add(new Pair<>(ident, index));

index = nextIndex;

}

else {

++index;

}

}

}

return identList;

}

The doUndef method removes a macro from the MacroMap map. If the macro does not exists a warning is given.

public static void doUndef(String line) {

String name = line;

checkIdentifier(name);

Assert.warning(Main.MacroMap.remove(name) != null,

name, "macro not defined");

}

### Conditional Programming

With conditional programming, it is possible to exclude part of the source code. The doIf method uses the parser of Section 4.1. It calls parseExpression, which creates a scanner and parser, parse the line, and returns a Boolean value that is true in case of a non-zero value.

private static void doIf(String line) {

boolean result = parseExpression(line);

Main.IfStack.push(new Triple<>(result, result, IfStatus.If));

}

public static Object PreProcessorResult;

private static boolean parseExpression(String line) {

StringReader reader = new StringReader(line);

int result = 0;

try {

PreScanner preScanner = new PreScanner(reader);

PreParser preParser = new PreParser(preScanner.toLexer());

Assert.error(preParser.parse(), "pre parser");

result = (int) PreProcessorResult;

}

catch (Exception exception) {

Assert.error(line, "invalid expression");

}

reader.close();

return (result != 0);

}

The doIfDefined and doIfNotDefined checks if the macro is defined or not, respectively, and add the result to the Main.IfStack stack.

private static void doIfDefined(String line) {

String name = line.trim();

checkIdentifier(name);

boolean result = Main.MacroMap.containsKey(name);

Main.IfStack.push(new Triple<>(result, result, IfStatus.If));

}

private static void doIfNotDefined(String line) {

String name = line.trim();

checkIdentifier(name);

boolean result = !Main.MacroMap.containsKey(name);

Main.IfStack.push(new Triple<>(result, result, IfStatus.If));

}

The doIfElseIf method is a little bit more complicated. It checks whether there is a preceding #if directive (Main.IfStack is not empty). Thereafter, it looks up the status of the if-stack. The stack value is a triple, the first value indicates whether the current if-status is true, and the second value gives whether there has been an earlier true if-status. The third value is of type IfStatus, that can hold the value If and Else. If the value is Else, and #elseif has already occurred, and another on is not allowed. Finally, if this if-chain has ever been true (lastTrue is true) is does not matter whether the condition of directive is true, the new status is false. If not, we consider if this value is true. In both cases, we push the result on the stack.

private static void doElseIf(String line) {

boolean result = parseExpression(line);

Assert.error(!Main.IfStack.isEmpty(),

"#elif without preceeding #if, #ifdef, or #ifndef");

Triple<Boolean,Boolean,IfStatus> ifTriple = Main.IfStack.pop();

boolean currTrue = ifTriple.getFirst(),

lastTrue = ifTriple.getSecond();

IfStatus lastIf = ifTriple.getThird();

Assert.error(lastIf == IfStatus.If,

"#elif after #else if not allowed");

if (!lastTrue) {

Main.IfStack.push(new Triple<>(result, result, IfStatus.If));

}

else {

Main.IfStack.push(new Triple<>(false, true, IfStatus.If));

}

}

The doElse method is a little bit easier. It checks whether there has ever been a true part of the if-chain. If not, it pushes true on the stack.

private static void doElse(String line) {

Assert.error(!Main.IfStack.isEmpty(),

"#else without preceeding #if, #ifdef, or #ifndef");

Triple<Boolean,Boolean,IfStatus> ifTriple = Main.IfStack.pop();

boolean lastTrue = ifTriple.getSecond();

IfStatus lastIf = ifTriple.getThird();

Assert.error(lastIf == IfStatus.If,

"#else after #else if not allowed");

Main.IfStack.push(new Triple<>(!lastTrue, true /\* doesn't matter \*/,

IfStatus.Else));

}

Finally, the doEndIf just pops the if-stack, subject to it is not already empty.

private static void doEndIf(String line) {

Assert.error(!Main.IfStack.isEmpty(),

"#endif without preceeding #if, #ifdef, or #ifndef");

Main.IfStack.pop();

}

### Macro Expansion

In the regular source code (line not starting with ‘#’), we need to replace macro calls with their resulting bodies. We also need to recursively replace macro calls in the bodies. However, unlike function, recursive macro calls are not allowed.

We go through the text, and look for identifiers, we do not have to worry about identifier in strings or characters, since every letter or underline (an identifier must begin with a letter or underline) in strings or characters has been changed to a three-octal-digit backslash code by slashToChar in Section 4.2.2.

When we find and identifier that is a stored macro name (either in m\_specialMacroSet or MacroMap), we scan the parameters if the name is followed by a left parenthesis, each parameter is recursively searched for macros. We then look up the macro body and replace its marked parameter locations with the parameters. Finally, we replace the macro call with its body. The nameStack stack is used to prevent recursive macro calls.

private static final Set<String> m\_specialMacroSet = new ListSet<>();

static {

m\_specialMacroSet.add("\_\_STDC\_\_");

m\_specialMacroSet.add("\_\_FILE\_\_");

m\_specialMacroSet.add("\_\_LINE\_\_");

m\_specialMacroSet.add("\_\_DATE\_\_");

m\_specialMacroSet.add("\_\_TIME\_\_");

}

private static String searchForMacros(String text,Stack<String> nameStack){

StringBuilder buffer = new StringBuilder(text + "\0");

for (int index = 0; buffer.charAt(index) != '\0'; ++index) {

if (Character.isLetter(buffer.charAt(index)) ||

(buffer.charAt(index) == '\_')) {

int nextIndex = index;

while (Character.isLetterOrDigit(buffer.charAt(nextIndex)) ||

(buffer.charAt(nextIndex) == '\_')) {

++nextIndex;

}

String name = buffer.substring(index, nextIndex);

if (!nameStack.contains(name) && (Main.MacroMap.containsKey(name) ||

m\_specialMacroSet.contains(name))){

nameStack.push(name);

List<String> paramList = new LinkedList<>();

if (buffer.charAt(nextIndex) == '(') {

nextIndex = scanParameters(buffer, nextIndex, paramList) + 1;

}

for (int paramIndex = 0; paramIndex < paramList.size();

++paramIndex) {

String oldParam = paramList.get(paramIndex) + "\0";

String newParam = searchForMacros(oldParam, nameStack);

paramList.set(paramIndex, newParam);

}

String macroText = lookupMacro(name, paramList);

macroText = searchForMacros(macroText, nameStack);

buffer.replace(index, nextIndex, macroText);

index += macroText.length();

nameStack.pop();

}

else {

index = nextIndex - 1;

}

}

}

return buffer.toString().replace("\0", "");

}

The scanParameters extracts the actual parameters of a macro call. The parameters are separated by commas and the list is terminated by a right parenthesis. It becomes a bit complicated since the parameters themselves can hold parenthesis and commas. The field paranCount counts the level of parentheses nesting and only takes parameter into consideration when the nesting level is zero.

private static int scanParameters(StringBuilder buffer, int index,

List<String> paramList) {

int parenCount = 0;

StringBuilder paramBuffer = new StringBuilder();

for (; true; ++index) {

char c = buffer.charAt(index);

switch (c) {

case '(':

if (++parenCount > 1) {

paramBuffer.append(c);

}

break;

case ')':

if (--parenCount == 0) {

paramList.add(paramBuffer.toString());

return index;

}

else {

paramBuffer.append(c);

}

break;

case ',':

if (parenCount == 1) {

paramList.add(paramBuffer.toString());

paramBuffer.delete(0, paramBuffer.length());

}

else {

paramBuffer.append(c);

}

break;

case '\0':

Assert.error("unfinished macro parameter sequence");

break;

default:

paramBuffer.append(c);

break;

}

}

}

The lookupMacro method looks up a given macro. If it stored in MacroMap, the number of parameters is checked and then the parameter locations in the macro body are replaced by the actual parameters.

private static String lookupMacro(String name, List<String> paramList) {

if (Main.MacroMap.containsKey(name)) {

Macro macro = Main.MacroMap.get(name);

Assert.error(paramList.size() == macro.parameters(),

"invalid number of parameters");

StringBuilder buffer = new StringBuilder(macro.body());

for (int index = 0; index < macro.parameters(); ++index) {

replaceAll(buffer, "#$" + index + Main.SeparatorId,

"\"" + paramList.get(index) + "\"");

replaceAll(buffer, Main.SeparatorId + index + Main.SeparatorId,

paramList.get(index));

}

return buffer.toString();

}

If the macro name is not stored in MacroMap, it may be one of the predefined special macros. The \_\_STDC\_\_ macro is replaced by the one integer value since the compiler of this book supports standard C.

else if (name.equals("\_\_STDC\_\_")) {

Assert.error(paramList.isEmpty(), "\_\_STDC\_\_",

"invalid macro parameter list");

return "1";

}

The \_\_FILE\_\_ macro is replaced by the current path (Main.Path), the \_\_LINE\_\_ macro is replaced by the current line number (Main.Line).

else if (name.equals("\_\_FILE\_\_")) {

Assert.error(paramList.isEmpty(), "\_\_FILE\_\_",

"invalid macro parameter list");

StringBuilder fileBuffer =

new StringBuilder("\"" + Main.Path.replace("\\", "\\\\") + "\"\0");

traverseString(1, fileBuffer, true);

return fileBuffer.toString().replace("\0", "");

}

else if (name.equals("\_\_LINE\_\_")) {

Assert.error(paramList.isEmpty(), "\_\_LINE\_\_",

"invalid macro parameter list");

return Integer.toString(Main.Line);

}

The \_\_DATE\_\_ macro is replaced by the current date on the format “Jan 1, 1970” and the \_\_TIME\_\_ macro is replaced by the current time on the format “01:02:03”.

else if (name.equals("\_\_DATE\_\_")) {

Assert.error(paramList.isEmpty(), "\_\_DATE\_\_",

"invalid macro parameter list");

Calendar calendar = new GregorianCalendar();

String monthArray[] = {"Jan", "Feb", "Mar", "Apr", "May", "Jun",

"Jul", "Aug", "Sep", "Oct", "Nov", "Dec"};

return (String.format("\"%s %02d %04d\"",

monthArray[calendar.get(Calendar.MONTH)],

calendar.get(Calendar.DAY\_OF\_MONTH),

calendar.get(Calendar.YEAR)));

}

else if (name.equals("\_\_TIME\_\_")) {

Assert.error(paramList.isEmpty(), "\_\_TIME\_\_",

"invalid macro parameter list");

Calendar calendar = new GregorianCalendar();

return (String.format("\"%02d:%02d:%02d\"",

calendar.get(Calendar.HOUR\_OF\_DAY),

calendar.get(Calendar.MINUTE),

calendar.get(Calendar.SECOND)));

}

If the macro is not stored in the macro map and is not a special macro, null is returned.

else {

return null;

}

}

### String Merging

When the all the macros have been expanded, it is finally time to check whether there are pairs of string constants that need to be merged:

|  |  |  |
| --- | --- | --- |
| "Hello" "World" | "HelloWorld" |  |
| (a) Before | (b) After |  |

private static void mergeStrings(StringBuilder buffer) {

int lastIndex = -1, index = buffer.length() - 1;

while (index >= 0) {

char c = buffer.charAt(index);

if (c == '\"') {

if (lastIndex != -1) {

buffer.replace(index, lastIndex + 1, "");

}

--index;

while ((index >= 0) && (buffer.charAt(index) != '\"')) {

--index;

}

lastIndex = index--;

}

else {

if (!Character.isWhitespace(c)) {

lastIndex = -1;

}

--index;

}

}

}

}

# Scanning

### The typedef-name Problem

Let us take look at the tow source code lines below. Intuitively, the first line looks like an expression statement where x and y are integer or floating variables. Admittedly, the expression lacks side effects and is therefore meaningless, but it is still an expression statement. On the other hand, the second line looks like a pointer declaration where T is a type defined by typedef and p is a pointer to that type.

x \* y;

T\* p;

However, syntactically it is the same thing: an identifier followed by an asterisk, another identifier and a semicolon. The parser cannot distinguish between the two cases, which is a problem. The only reasonably solution is to give the scanner access to the symbol table, so it can look up the whether the identifier is a typedef. If it is, the TYPEDEF token is returned; otherwise, the IDENTIFIER token is returned. In that way, the parser can distinguish between the two cases.

## The Scanner

The scanner of this section is similar to the scanners of Sections 2.1.4 and 4.1.2. However, there are more keyword and operators as well as more complicated regular expressions.

Scanner.jflex

package c\_compiler;

import java.math.\*;

%%

%public

%class Scanner

%type Token

%notunix

//%unicode

%line

%char

%eofval{

return null;

%eofval}

%yylexthrow{

java.lang.Exception

%yylexthrow}

%eofval{

return null;

%eofval}

%yylexthrow{

java.lang.Exception

%yylexthrow}

%{

public Parser.Lexer toLexer() {

return (new ScannerToLexer(this));

}

private static final int OctalBase = 8, DecimalBase = 10,

HexadecimalBase = 16;

private Sort getIntegralSort(StringBuilder buffer) {

Sort sort = Sort.Signed\_Int;

int shortIndex = buffer.indexOf("s");

if (shortIndex != -1) {

sort = Sort.Signed\_Short\_Int;

buffer.replace(shortIndex, shortIndex + 1, "");

}

int longIndex = buffer.indexOf("l");

if (longIndex != -1) {

sort = Sort.Signed\_Long\_Int;

buffer.replace(longIndex, longIndex + 1, "");

}

int unsignedIndex = buffer.indexOf("u");

if (unsignedIndex != -1) {

sort = Sort.valueOf(sort.name().replace("Signed", "Unsigned"));

buffer.replace(unsignedIndex, unsignedIndex + 1, "");

}

return sort;

}

private Sort getFloatingSort(StringBuilder buffer) {

Sort sort = Sort.Double;

int floatIndex = buffer.indexOf("f");

if (floatIndex != -1) {

sort = Sort.Float;

buffer.replace(floatIndex, floatIndex + 1, "");

}

int longIndex = buffer.indexOf("l");

if (longIndex != -1) {

sort = Sort.Long\_Double;

buffer.replace(longIndex, longIndex + 1, "");

}

return sort;

}

%}

However, the regular expressions for values are a bit more complicated. There are several kinds of values:

* Octal. An octal value starts with a zero that is followed by zero or more digits between zero and eight, inclusive.
* Decimal. A decimal value is made up by one or more digits.
* Hexadecimal. A hexadecimal value starts with zero, followed by a small or capital x and one of more hexadecimal digits.
* Floating. A floating value starts with one or more digits followed by a dot and zero or more digits or starts with a dot followed by one or more digits. It is possible to add an exponent part that starts with a small or capital e, followed by a potential plus or minus sign and one or more digits. Finally, a floating value can also be made up by one or more digits without a dot, followed by a small or capital e, a possible plus or minus sign, and one or more digits.

The decimal values can be appended by a small or capital u to indicate an unsigned value and a small or capital s or l to indicate a short or long value. In the same way, a floating value can be appended by a small or capital f or l to indicate a float or long double. Note that the values cannot start with a plus or minus sign; instead, the unary operators in that parser takes care of that.

OCTAL\_VALUE = [0][0-7]\*([uU]|[sSlL]|[uU][sSlL]|[sSlL][uU])?

DECIMAL\_VALUE = [1-9][0-9]\*([uU]|[sSlL]|[uU][sSlL]|[sSlL][uU])?

HEXADECIMAL\_VALUE = [0][xX][0-9a-fA-F]+([uU]|[sSlL]|[uU][sSlL]|[sSlL][uU])?

FLOATING\_VALUE = (([0-9]+"."[0-9]\*|"."[0-9]+)([eE][\+\-]?[0-9]+)?|[0-9]+[eE][\+\-]?[0-9]+)([fF]|[lL])?

A character starts with single-quote followed by one or more occurrences of everything except a single-quote and is terminated by a double-quote. In the same way, a string starts with a double-quote followed by zero or more occurrences of anything except a double-quote and is terminated by a double-quote.

CHAR\_VALUE = \'[^\']+\'

STRING\_VALUE = \"[^\"]\*\"

An identifier starts with a letter or underline, followed by zero or more letters, digits, under underlines. A path line is a dollar sign followed by anything except newline (the dot represent every character except newline), and a white space is either a space, a tabulator, a return, a newline, or a formfeed.

IDENTIFIER = [a-zA-Z\_][a-zA-Z0-9\_]\*

PATH\_LINE = \$.\*\$

WHITE\_SPACE = [ \t\r\n\f]

%%

"auto" { return (new Token(Parser.AUTO)); }

"break" { return (new Token(Parser.BREAK)); }

"case" { return (new Token(Parser.CASE)); }

"char" { return (new Token(Parser.CHAR)); }

"clear\_registers" { return (new Token(Parser.CLEAR\_REGISTERS)); }

"const" { return (new Token(Parser.CONSTANT)); }

"continue" { return (new Token(Parser.CONTINUE)); }

"default" { return (new Token(Parser.DEFAULT)); }

"do" { return (new Token(Parser.DO)); }

"double" { return (new Token(Parser.DOUBLE)); }

"else" { return (new Token(Parser.ELSE)); }

"enum" { return (new Token(Parser.ENUM)); }

"extern" { return (new Token(Parser.EXTERN)); }

"float" { return (new Token(Parser.FLOAT)); }

"for" { return (new Token(Parser.FOR)); }

"goto" { return (new Token(Parser.GOTO)); }

"int" { return (new Token(Parser.INT)); }

"interrupt" { return (new Token(Parser.INTERRUPT)); }

"if" { return (new Token(Parser.IF)); }

"jump\_register" { return (new Token(Parser.JUMP\_REGISTER)); }

"load\_register" { return (new Token(Parser.LOAD\_REGISTER)); }

"long" { return (new Token(Parser.LONG)); }

"register" { return (new Token(Parser.REGISTER)); }

"return" { return (new Token(Parser.RETURN)); }

"short" { return (new Token(Parser.SHORT)); }

"signed" { return (new Token(Parser.SIGNED)); }

"sizeof" { return (new Token(Parser.SIZE\_OF)); }

"static" { return (new Token(Parser.STATIC)); }

"store\_register" { return (new Token(Parser.STORE\_REGISTER)); }

"store\_flagbyte" { return (new Token(Parser.STORE\_FLAGBYTE)); }

"struct" { return (new Token(Parser.STRUCT)); }

"switch" { return (new Token(Parser.SWITCH)); }

"typedef" { return (new Token(Parser.TYPEDEF)); }

"union" { return (new Token(Parser.UNION)); }

"unsigned" { return (new Token(Parser.UNSIGNED)); }

"wchar\_t" { return (new Token(Parser.WCHAR\_T)); }

"while" { return (new Token(Parser.WHILE)); }

"void" { return (new Token(Parser.VOID)); }

"volatile" { return (new Token(Parser.VOLATILE)); }

";" { return (new Token(Parser.SEMICOLON)); }

":" { return (new Token(Parser.COLON)); }

"," { return (new Token(Parser.COMMA)); }

"." { return (new Token(Parser.DOT)); }

"->" { return (new Token(Parser.ARROW)); }

"..." { return (new Token(Parser.ELLIPSE)); }

"(" { return (new Token(Parser.LEFT\_PAREN)); }

")" { return (new Token(Parser.RIGHT\_PAREN)); }

"{" { return (new Token(Parser.LEFT\_BLOCK)); }

"}" { return (new Token(Parser.RIGHT\_BLOCK)); }

"[" { return (new Token(Parser.LEFT\_SQUARE)); }

"]" { return (new Token(Parser.RIGHT\_SQUARE)); }

"\*" { return (new Token(Parser.ASTERRISK)); }

"?" { return (new Token(Parser.CONDITION)); }

"||" { return (new Token(Parser.LOGICAL\_OR)); }

"&&" { return (new Token(Parser.LOGICAL\_AND)); }

"!" { return (new Token(Parser.LOGICAL\_NOT)); }

"&" { return (new Token(Parser.AMPERSAND)); }

"^" { return (new Token(Parser.BITWISE\_XOR)); }

"|" { return (new Token(Parser.BITWISE\_IOR)); }

"~" { return (new Token(Parser.BITWISE\_NOT)); }

"==" { return (new Token(Parser.EQUAL)); }

"!=" { return (new Token(Parser.NOT\_EQUAL)); }

"<" { return (new Token(Parser.LESS\_THAN)); }

"<=" { return (new Token(Parser.LESS\_THAN\_EQUAL)); }

">" { return (new Token(Parser.GREATER\_THAN)); }

">=" { return (new Token(Parser.GREATER\_THAN\_EQUAL)); }

"<<" { return (new Token(Parser.LEFT\_SHIFT)); }

">>" { return (new Token(Parser.RIGHT\_SHIFT)); }

"+" { return (new Token(Parser.PLUS)); }

"-" { return (new Token(Parser.MINUS)); }

"/" { return (new Token(Parser.DIVIDE)); }

"%" { return (new Token(Parser.MODULO)); }

"++" { return (new Token(Parser.INCREMENT)); }

"--" { return (new Token(Parser.DECREMENT)); }

"=" { return (new Token(Parser.ASSIGN)); }

"+=" { return (new Token(Parser.ADD\_ASSIGN)); }

"-=" { return (new Token(Parser.SUBTRACT\_ASSIGN)); }

"\*=" { return (new Token(Parser.MULTIPLY\_ASSIGN)); }

"/=" { return (new Token(Parser.DIVIDE\_ASSIGN)); }

"%=" { return (new Token(Parser.MODULO\_ASSIGN)); }

"<<=" { return (new Token(Parser.LEFT\_SHIFT\_ASSIGN)); }

">>=" { return (new Token(Parser.RIGHT\_SHIFT\_ASSIGN)); }

"&=" { return (new Token(Parser.AND\_ASSIGN)); }

"^=" { return (new Token(Parser.XOR\_ASSIGN)); }

"|=" { return (new Token(Parser.IOR\_ASSIGN)); }

{IDENTIFIER} {

String name = yytext();

name = name.equals("abs") ? "abs$" : name;

Symbol symbol = Main.CurrentTable.lookupSymbol(name);

if ((symbol != null) && symbol.isTypedef()) {

//System.out.println("typedef name: " + symbol.name());

return (new Token(Parser.TYPEDEF\_NAME, symbol.getType()));

}

else {

//System.out.println("identifier: " + name);

return (new Token(Parser.IDENTIFIER, name));

}

}

{OCTAL\_VALUE} {

StringBuilder buffer = new StringBuilder(yytext().toLowerCase());

Type type = Type.createArithmeticType(getIntegralSort(buffer));

BigInteger value = new BigInteger(buffer.toString(), OctalBase);

Symbol symbol = Symbol.createValueSymbol(type, value);

return (new Token(Parser.VALUE, symbol));

}

{DECIMAL\_VALUE} {

StringBuilder buffer = new StringBuilder(yytext().toLowerCase());

Type type = Type.createArithmeticType(getIntegralSort(buffer));

BigInteger value = new BigInteger(buffer.toString(), DecimalBase);

Symbol symbol = Symbol.createValueSymbol(type, value);

return (new Token(Parser.VALUE, symbol));

}

{HEXADECIMAL\_VALUE} {

String text = yytext().substring(2).toLowerCase();

StringBuilder buffer = new StringBuilder(text);

Type type = Type.createArithmeticType(getIntegralSort(buffer));

BigInteger value = new BigInteger(buffer.toString(), HexadecimalBase);

Symbol symbol = Symbol.createValueSymbol(type, value);

return (new Token(Parser.VALUE, symbol));

}

{FLOATING\_VALUE} {

StringBuilder buffer = new StringBuilder(yytext().toLowerCase());

Type type = Type.createArithmeticType(getFloatingSort(buffer));

BigDecimal value = new BigDecimal(buffer.toString());

Symbol symbol = Symbol.createValueSymbol(type, value);

return (new Token(Parser.VALUE, symbol));

}

{CHAR\_VALUE} {

Type type = Type.createArithmeticType(Sort.Signed\_Char);

String text = yytext().substring(1, yylength() - 1);

int value = (int) PreProcessor.octalToChar(text).charAt(0);

Symbol symbol = Symbol.createValueSymbol(type, BigInteger.valueOf(value));

return (new Token(Parser.VALUE, symbol));

}

{STRING\_VALUE} {

Type type = Type.createArithmeticType(Sort.String);

String text = yytext().substring(1, yylength() - 1);

Symbol symbol = Symbol.createValueSymbol(type, PreProcessor.octalToChar(text));

return (new Token(Parser.VALUE, symbol));

}

{PATH\_LINE} {

String text = yytext().substring(1, yylength() - 1);

int index = text.indexOf(',');

Main.Path = text.substring(0, index).replace("+", " ");

Main.Line = Integer.parseInt(text.substring(index + 1));

}

{WHITE\_SPACE} {

if (yytext().equals("\n")) {

++Main.Line;

}

}

. { Assert.error(yytext(), "unknown character"); }

# Parsing

This chapter describes the parser of the compiler, how it parses the source code to test whether is complies to the grammar. However, the parser does also performs type checking, store symbols in the symbol table and generates middle code.

Parser.bison

%{

package c\_compiler;

import java.math.\*;

import java.util.\*;

%}

%name-prefix = ""

%token AUTO REGISTER STATIC EXTERN TYPEDEF CONSTANT VOLATILE

SIGNED UNSIGNED CHAR WCHAR\_T SHORT INT LONG

FLOAT DOUBLE VOID ENUM STRUCT UNION

PLUS MINUS DIVIDE MODULO INCREMENT DECREMENT

EQUAL NOT\_EQUAL LESS\_THAN LESS\_THAN\_EQUAL GREATER\_THAN

GREATER\_THAN\_EQUAL LEFT\_SHIFT RIGHT\_SHIFT

ASSIGN ADD\_ASSIGN SUBTRACT\_ASSIGN MULTIPLY\_ASSIGN

DIVIDE\_ASSIGN MODULO\_ASSIGN LEFT\_SHIFT\_ASSIGN

RIGHT\_SHIFT\_ASSIGN AND\_ASSIGN IOR\_ASSIGN XOR\_ASSIGN

COLON COMMA SEMICOLON ELLIPSE DOT ARROW ASTERRISK AMPERSAND

LEFT\_PAREN RIGHT\_PAREN LEFT\_BLOCK RIGHT\_BLOCK

LEFT\_SQUARE RIGHT\_SQUARE SIZE\_OF

LOGICAL\_OR LOGICAL\_AND LOGICAL\_NOT BITWISE\_XOR BITWISE\_IOR

BITWISE\_NOT CONDITION IF ELSE SWITCH CASE DEFAULT

FOR WHILE DO CONTINUE BREAK RETURN GOTO INTERRUPT

LOAD\_REGISTER STORE\_REGISTER CLEAR\_REGISTERS

STORE\_FLAGBYTE JUMP\_REGISTER

%token <String> IDENTIFIER;

%token <Type> TYPEDEF\_NAME;

%token <Symbol> VALUE;

%type <Object> declaration\_specifier

%type <String> optional\_identifier

%type <Type> struct\_union\_specifier

%type <Scope> struct\_union

%type <Type> enum\_specifier

%type <List> enum\_list

%type <Symbol> enum

%type <List> declaration declarator\_list init\_bitfield\_declarator

%type <Declarator> declarator direct\_declarator

%type <List> optional\_pointer\_list pointer\_list

%type <Type> pointer

%type <Pair> optional\_parameter\_ellipse\_list parameter\_ellipse\_list

%type <List> parameter\_list

%type <Symbol> parameter\_declaration

%type <List> optional\_identifier\_list identifier\_list initializer\_list

%type <Object> initializer

%type <Type> type\_name

%type <Declarator> abstract\_declarator direct\_abstract\_declarator

%type <List> optional\_statement\_list

%type <SyntaxTree> statement open\_statement closed\_statement

%type <MiddleOperator> equality\_operator relation\_operator

shift\_operator binary\_add\_operator assignment\_operator

multiply\_operator prefix\_add\_operator increment\_operator

%type <SyntaxTree> optional\_constant\_expression optional\_expression

constant\_expression expression

assignment\_expression conditional\_expression

logical\_or\_expression logical\_and\_expression

bitwise\_ior\_expression bitwise\_xor\_expression

bitwise\_and\_expression equality\_expression

relation\_expression shift\_expression

add\_expression multiply\_expression cast\_expression

prefix\_expression postfix\_expression primary\_expression

%type <List> optional\_argument\_expression\_list argument\_expression\_list

## Declarations

The translation\_unit rule is the start of the grammar.

%start translation\_unit

%%

translation\_unit:

external\_declaration

| translation\_unit external\_declaration;

external\_declaration:

function\_definition

| declaration;

### Function Definitions

A function definition is made up by declarator, possible preceded by a declaration specifier list, followed by an optional declaration list, and a block with an optional statement list.

Parser.bison

function\_definition:

declarator {

GenerateDeclaration.generateFunctionHeader(null, $1);

}

optional\_declaration\_list {

GenerateDeclaration.checkFunctionDefinition();

}

LEFT\_BLOCK optional\_statement\_list RIGHT\_BLOCK {

SyntaxTree stat = SyntaxTreeGenerator.generateBlockStatement($6);

SyntaxTreeGenerator.checkGotoSet();

GenerateDeclaration.generateFunctionCode(stat);

}

| declaration\_specifier\_list declarator {

GenerateDeclaration.generateFunctionHeader(Main.SpecifierStack.pop(), $2);

}

optional\_declaration\_list {

GenerateDeclaration.checkFunctionDefinition();

}

LEFT\_BLOCK optional\_statement\_list RIGHT\_BLOCK {

SyntaxTree stat = SyntaxTreeGenerator.generateBlockStatement($7);

SyntaxTreeGenerator.checkGotoSet();

GenerateDeclaration.generateFunctionCode(stat);

};

optional\_declaration\_list:

/\* Empty. \*/

| optional\_declaration\_list declaration;

GenerateDeclaration.java

package c\_compiler;

import java.math.\*;

import java.util.\*;

The generateFunctionHeader is called before the middle code of the function has been generated.

package c\_compiler;

import java.io.\*;

import java.math.\*;

import java.util.\*;

public class GenerateDeclaration {

public static void generateFunctionHeader(Specifier specifier,

Declarator decl) {

Triple<Boolean,Storage,Type> specTriple;

if (specifier == null) {

specTriple = new Triple<>(true, Storage.Extern, Type.SignedIntegerType);

}

else {

specTriple = specifier.generate();

}

A general function declarator may be unnamed; however, a function definition must be named and of course be a function declaration.

boolean external = specTriple.getFirst();

Storage storage = specTriple.getSecond();

Type specType = specTriple.getThird();

decl.add(specType);

The Main.FuncSymbol is set, which will be used in both the middle and object code generation.

String funcName = decl.name();

Type funcType = decl.type();

Assert.error(funcName != null, "unnamed function definition");

Assert.error(funcType.isFunction(), funcName, "not a function");

Type returnType = funcType.getReturnType();

Assert.error((storage == Storage.Static) || (storage == Storage.Extern),

funcName, "a function must be static or extern");

Main.FuncSymbol = Symbol.createVariableSymbol(funcName, external,

storage, funcType, null);

Main.CurrentTable.addSymbol(Main.FuncSymbol);

if (isMain(Main.FuncSymbol)) {

Assert.error(returnType.isVoid() || returnType.isInteger(),

"main", "main function must return void or integer");

}

Main.CurrentTable = new SymbolTable(Main.CurrentTable, Scope.Function);

}

If the function misses storage, it is given extern storage.

If the function is the main function is has to return void or integer.

The middle code list is cleared, the symbol table is pushed with function scope, and the function start middle code instruction is added to the list.

It is not allowed to mix the old and new style function definition style in that way that a new-style function definition have a declaration list. In the old-style function definition, the parameter identifier list must match the declaration list.

public static void checkFunctionDefinition() {

Type funcType = Main.FuncSymbol.getType();

if (funcType.isOldStyleFunction()) {

List<String> nameList = funcType.getNameList();

Map<String,Object> entryMap = Main.CurrentTable.getEntryMap();

Assert.error(nameList.size()==entryMap.size(),

Main.FuncSymbol.getName(), "unmatched number of parameters"

+ " in old-style function definition");

for (String name : nameList) {

Assert.error(entryMap.containsKey(name), name, "undefined parameter");

}

}

else {

Assert.error(Main.CurrentTable.getEntryMap().isEmpty(),

"new and old style mixed function definition");

for (Symbol paramSymbol : funcType.getParameterList()) {

Main.CurrentTable.addSymbol(paramSymbol);

}

}

}

The function is the main function if it is named main and have static or extern storage.

public static boolean isMain(Symbol symbol) {

Storage storage = symbol.getStorage();

return (symbol.getName().equals("main") &&

((storage == Storage.Static) || (storage == Storage.Extern)));

}

The main function must have an empty parameter list or the (int argc, char\* argv[]) parameter list.

public static void checkMain(Symbol symbol) {

List<Type> typeList = symbol.getType().getTypeList();

Assert.error((typeList == null) || typeList.isEmpty() ||

isMainArgs(symbol), "main", "invalid parameter list");

}

public static boolean isMainArgs(Symbol symbol) {

List<Type> typeList = symbol.getType().getTypeList();

return (typeList != null) && (typeList.size() == 2) &&

typeList.get(0).isInteger() &&

typeList.get(1).isPointer() &&

typeList.get(1).getPointerType().isPointer() &&

typeList.get(1).getPointerType().getPointerType().isChar();

}

The generateFunctionCode is called after the middle code of the function has been generated. Its task is to genereate the assembler and object code of the function. If the function returns void and is the main function, the Exit middle code is added to middle code list. If the it returns void and is not the main, the Return middle code instruction is added.

public static void generateFunctionCode(SyntaxTree stat) {

List<MiddleCode> middleCodeList = new LinkedList<>();

MiddleCodeGenerator generateCode =

new MiddleCodeGenerator(middleCodeList);

generateCode.addMiddleCode(MiddleOperator.FunctionStart);

Main.CurrentTable.setFunctionOffset();

generateCode.generateStatement(stat);

generateCode.backpatchGoto();

if (Main.FuncSymbol.getType().getReturnType().isVoid()) {

if (isMain(Main.FuncSymbol)) {

Type signedShortType =

Type.createArithmeticType(Sort.Signed\_Short\_Int);

Symbol zeroSymbol =

Symbol.createValueSymbol(signedShortType, BigInteger.ZERO);

generateCode.addMiddleCode(MiddleOperator.Exit, null, zeroSymbol);

}

else {

generateCode.addMiddleCode(MiddleOperator.Return);

}

}

generateCode.addMiddleCode(MiddleOperator.FunctionEnd, Main.FuncSymbol);

Main.CurrentTable.checkUnreferredSymbol();

List<ObjectCode> objectCodeList = new LinkedList<>();

ObjectCodeGenerator objectCodeGenerator =

new ObjectCodeGenerator(middleCodeList, objectCodeList);

if (isMain(Main.FuncSymbol)) {

checkMain(Main.FuncSymbol);

objectCodeGenerator.addInitCode();

if (isMainArgs(Main.FuncSymbol)) {

objectCodeGenerator.addArgsCode();

Main.FuncSymbol.setEntryPoint(ObjectCode.MainInitSize +

ObjectCode.MainInitArgsSize);

}

else {

Main.FuncSymbol.setEntryPoint(ObjectCode.MainInitSize);

}

}

objectCodeGenerator.generateObjectCodeList();

objectCodeGenerator.generateJumpInfo();

List<Byte> byteList = new LinkedList<Byte>();

Map<Integer,String> accessMap = new ListMap<>(),

callMap = new ListMap<>();

Map<Integer,Integer> byteToTextMap = new ListMap<>();

Set<Integer> returnSet = new ListSet<>();

List<String> textList = new LinkedList<>();

objectCodeGenerator.generateTargetByteList(byteList, accessMap, callMap,

returnSet, byteToTextMap, textList);

Main.FuncSymbol.setByteList(byteList);

Main.FuncSymbol.setAccessMap(accessMap);

Main.FuncSymbol.setCallMap(callMap);

Main.FuncSymbol.setReturnSet(returnSet);

Main.FuncSymbol.setByteToTextMap(byteToTextMap);

Main.FuncSymbol.setTextList(textList);

Main.GlobalSet.add(Main.FuncSymbol);

if (isMain(Main.FuncSymbol) && Main.CheckStackHeap) {

objectCodeGenerator.generateStackHeapCheckFunction();

}

Main.CurrentTable = Main.CurrentTable.getParentTable();

Main.FuncSymbol = null;

}

Then the middle code is being optimized and a list of base blocks is generated.

The object code is actually generated twice, since we want to know which symbols are not used and can be removed from the symbol table.

If the function is the main function we have to modify the entry point of the function.

Finally, the function is added to the linker map and the symbol table is popped.

The generateAssemblerText method generates the assembler code for the function.

When a function containing goto statements has been parsed, the addresses need to be set with the help of the label map.

### Declarations

Parser.bison

declaration:

declaration\_specifier\_list SEMICOLON {

Main.SpecifierStack.pop().generate();

$$ = new LinkedList<>();

}

| declaration\_specifier\_list declarator\_list SEMICOLON {

Main.SpecifierStack.pop().generate();

$$ = $2;

};

declaration\_specifier\_list:

declaration\_specifier {

Specifier specifier = new Specifier();

specifier.add($1);

Main.SpecifierStack.push(specifier);

}

| declaration\_specifier\_list declaration\_specifier {

Specifier specifier = Main.SpecifierStack.peek();

specifier.add($2);

};

declaration\_specifier:

CONSTANT { $$ = Mask.Constant; }

| VOLATILE { $$ = Mask.Volatile; }

| AUTO { $$ = Mask.Auto; }

| REGISTER { $$ = Mask.Register; }

| STATIC { $$ = (Main.CurrentTable.getScope() == Scope.Global)

? Mask.Private : Mask.Static; }

| EXTERN { $$ = Mask.Extern; }

| TYPEDEF { $$ = Mask.Typedef; }

| VOID { $$ = Mask.Void; }

| CHAR { $$ = Mask.Char; }

| WCHAR\_T { $$ = Mask.WChar\_T; }

| SHORT { $$ = Mask.Short; }

| INT { $$ = Mask.Int; }

| LONG { $$ = Mask.Long; }

| FLOAT { $$ = Mask.Float; }

| DOUBLE { $$ = Mask.Double; }

| SIGNED { $$ = Mask.Signed; }

| UNSIGNED { $$ = Mask.Unsigned; }

| struct\_union\_specifier { $$ = $1; }

| enum\_specifier { $$ = $1; }

| TYPEDEF\_NAME { $$ = $1; };

The Mask class is used to distinguish between the different type specifiers and used by the Specifier class below.

Mask.java

package c\_compiler;

import java.util.\*;

public class Mask {

public static final int StorageMask = 0x0000FFF;

public static final int Auto = 0x0000010;

public static final int Register = 0x0000020;

public static final int Static = 0x0000040;

public static final int Extern = 0x0000080;

public static final int Typedef = 0x0000100;

public static final int QualifierMask = 0x000F000;

public static final int Constant = 0x0001000;

public static final int Volatile = 0x0002000;

public static final int Private = 0x0004000;

public static final int SortMask = 0xFFF0000;

public static final int Signed = 0x0010000;

public static final int Unsigned = 0x0020000;

public static final int Char = 0x0040000;

public static final int WChar\_T = 0x0080000;

public static final int Short = 0x0100000;

public static final int Int = 0x0200000;

public static final int Long = 0x0400000;

public static final int Float = 0x0800000;

public static final int Double = 0x1000000;

public static final int Void = 0x2000000;

public static final int SignedChar = Signed | Char;

public static final int UnsignedChar = Unsigned | Char;

public static final int ShortInt = Short | Int;

public static final int SignedShort = Signed | Short;

public static final int SignedShortInt = Signed | Short | Int;

public static final int UnsignedShort = Unsigned | Short;

public static final int UnsignedShortInt = Unsigned | Short | Int;

public static final int SignedInt = Signed | Int;

public static final int UnsignedInt = Unsigned | Int;

public static final int LongInt = Long | Int;

public static final int SignedLong = Signed | Long;

public static final int SignedLongInt = Signed | Long | Int;

public static final int UnsignedLong = Unsigned | Long;

public static final int UnsignedLongInt = Unsigned | Long | Int;

public static final int LongDouble = Long | Double;

public static final Map<Integer,String> MaskToNameMap = new ListMap<>();

static {

MaskToNameMap.put(Auto, "auto");

MaskToNameMap.put(Register, "register");

MaskToNameMap.put(Static, "static");

MaskToNameMap.put(Extern, "extern");

MaskToNameMap.put(Typedef, "typedef");

MaskToNameMap.put(Constant, "constant");

MaskToNameMap.put(Volatile, "volatile");

MaskToNameMap.put(Signed, "signed");

MaskToNameMap.put(Unsigned, "unsigned");

MaskToNameMap.put(Char, "char");

MaskToNameMap.put(WChar\_T, "wchar\_t");

MaskToNameMap.put(Short, "short");

MaskToNameMap.put(Int, "int");

MaskToNameMap.put(Long, "long");

MaskToNameMap.put(Float, "float");

MaskToNameMap.put(Double, "double");

MaskToNameMap.put(Void, "void");

}

}

The Specifier class is used to generate the type specified by a list of specifiers. First of all, we need the maskToStorageMap and maskToSortMap to map between specifiers and storage and sorts.

Specifier.java

package c\_compiler;

import java.util.\*;

public class Specifier {

private static final Map<Integer,Storage> m\_maskToStorageMap=new ListMap<>();

private static final Map<Integer,Sort> m\_maskToSortMap = new ListMap<>();

private List<Object> m\_specList = new LinkedList<>();

static {

m\_maskToStorageMap.put(Mask.Auto, Storage.Auto);

m\_maskToStorageMap.put(Mask.Register, Storage.Register);

m\_maskToStorageMap.put(Mask.Extern, Storage.Extern);

m\_maskToStorageMap.put(Mask.Static, Storage.Static);

m\_maskToStorageMap.put(Mask.Typedef, Storage.Typedef);

m\_maskToSortMap.put(Mask.Void, Sort.Void);

m\_maskToSortMap.put(Mask.Char, Sort.Signed\_Char);

m\_maskToSortMap.put(Mask.SignedChar, Sort.Signed\_Char);

m\_maskToSortMap.put(Mask.UnsignedChar, Sort.Unsigned\_Char);

m\_maskToSortMap.put(Mask.Short, Sort.Signed\_Short\_Int);

m\_maskToSortMap.put(Mask.ShortInt, Sort.Signed\_Short\_Int);

m\_maskToSortMap.put(Mask.SignedShort, Sort.Signed\_Short\_Int);

m\_maskToSortMap.put(Mask.SignedShortInt, Sort.Signed\_Short\_Int);

m\_maskToSortMap.put(Mask.UnsignedShort, Sort.Unsigned\_Short\_Int);

m\_maskToSortMap.put(Mask.UnsignedShortInt, Sort.Unsigned\_Short\_Int);

m\_maskToSortMap.put(Mask.Int, Sort.Signed\_Int);

m\_maskToSortMap.put(Mask.Signed, Sort.Signed\_Int);

m\_maskToSortMap.put(Mask.SignedInt, Sort.Signed\_Int);

m\_maskToSortMap.put(Mask.Unsigned, Sort.Unsigned\_Int);

m\_maskToSortMap.put(Mask.UnsignedInt, Sort.Unsigned\_Int);

m\_maskToSortMap.put(Mask.Long, Sort.Signed\_Long\_Int);

m\_maskToSortMap.put(Mask.LongInt, Sort.Signed\_Long\_Int);

m\_maskToSortMap.put(Mask.SignedLong, Sort.Signed\_Long\_Int);

m\_maskToSortMap.put(Mask.SignedLongInt, Sort.Signed\_Long\_Int);

m\_maskToSortMap.put(Mask.UnsignedLong, Sort.Unsigned\_Long\_Int);

m\_maskToSortMap.put(Mask.UnsignedLongInt, Sort.Unsigned\_Long\_Int);

m\_maskToSortMap.put(Mask.Float, Sort.Float);

m\_maskToSortMap.put(Mask.Double, Sort.Double);

m\_maskToSortMap.put(Mask.LongDouble, Sort.Long\_Double);

}

The first part of the type generation process is to inspect the specification list. The list is made up by masks (see the Mask class above) or at most one type.

public void add(Object spec) {

m\_specList.add(spec);

}

public Triple<Boolean,Storage,Type> generate() {

int totalMask = 0;

Type compoundType = null;

if (m\_specList != null) {

for (Object object : m\_specList) {

if (object instanceof Integer) {

int mask = (Integer) object;

Assert.error((totalMask & mask) == 0, Mask.MaskToNameMap.get(mask),

"keyword defined twice");

totalMask |= mask;

}

else if (object instanceof Type) {

Assert.error(compoundType == null,"invalid type specifier sequence");

compoundType = (Type) object;

}

}

}

We check whether the mask holds the constant or volatile qualifiers.

boolean isConstant = ((totalMask & Mask.QualifierMask) & Mask.Constant)!=0;

boolean isVolatile = ((totalMask & Mask.QualifierMask) & Mask.Volatile)!=0;

boolean isPrivate = ((totalMask & Mask.QualifierMask) & Mask.Private) !=0;

If the storage mask does not match any of the members of the m\_maskToStorageMap, two or more storage specifiers have been stated, which is not allowed.

Storage storage = null;

int storageMask = totalMask & Mask.StorageMask;

boolean external = (Main.CurrentTable.getScope() == Scope.Global) &&

!isPrivate;

if (storageMask != 0) {

Assert.error(!isPrivate, "invalid storage specifier sequence");

Assert.error(m\_maskToStorageMap.containsKey(storageMask),

"invalid storage specifier sequence");

storage = m\_maskToStorageMap.get(storageMask);

}

If there is a compound type and if it is an enumeration, we need to add its members as signed integers to the symbol table.

if ((compoundType != null) && compoundType.isEnumeration()) {

List<Symbol> symbolList = compoundType.symbolList();

for (Symbol itemSymbol : symbolList) {

itemSymbol.setStorage(storage);

Main.CurrentTable.addSymbol(itemSymbol);

if (storage == Storage.Static) {

GenerateInitializer.generateStatic(itemSymbol);

}

}

}

else if (Main.ParameterCount > 0) {

storage = Storage.Auto;

}

else if (Main.CurrentTable.getScope() == Scope.Global) {

storage = Storage.Static;

}

else {

storage = Storage.Auto;

}

Finally, we extract the sort mask and test whether it represent a valid type.

Assert.error(storage != null, "storage");

if ((compoundType != null) && compoundType.isEnumeration()) {

List<Symbol> symbolList = compoundType.getEnumerationList();

for (Symbol itemSymbol : symbolList) {

itemSymbol.setStorage(storage);

}

}

Sort sort = null;

int sortMask = totalMask & Mask.SortMask;

if (sortMask != 0) {

Assert.error(m\_maskToSortMap.containsKey(sortMask),

"invalid type specifier sequence");

sort = m\_maskToSortMap.get(sortMask);

}

When defining the final type, there are four cases:

1. If the compound type is not null and the sort is null, the result type is the compound type, with the possible addition of constant or volatile qualifiers.

2. If the compound type is null and the sort is not null, the result type is based on the sort.

3. The compound type and sort are both null, which means that there is no type (which may happen in function definitions). In that case, the type shall be a signed integer.

4. If both the compound type and the sort are not null, the type specification is invalid.

if ((compoundType != null) && (sort == null)) {

compoundType.setConstant(compoundType.isConstant() || isConstant);

compoundType.setVolatile(compoundType.isVolatile() || isVolatile);

return (new Triple<>(external, storage, compoundType));

}

else if ((compoundType == null) && (sort != null)) {

Type simpleType = Type.createArithmeticType(sort);

simpleType.setConstant(isConstant);

simpleType.setVolatile(isVolatile);

return (new Triple<>(external, storage, simpleType));

}

else if ((compoundType == null) && (sort == null)) {

Type simpleType = Type.createArithmeticType(Sort.Signed\_Int);

simpleType.setConstant(isConstant);

simpleType.setVolatile(isVolatile);

return (new Triple<>(external, storage, simpleType));

}

else {

Assert.error("invalid type specification sequence");

return null;

}

}

The generateQuailifers method is called when a pointer is declared. Its task is to exam whether the pointer is constant or volatile.

public Type generateQualifiers() {

Type type = Type.createPointerType(null);

if (m\_specList != null) {

int totalMask = 0;

for (Object object : m\_specList) {

if (object instanceof Integer) {

int mask = (Integer) object;

Assert.error((totalMask & mask) == 0, "keyword defined twice");

Assert.error((mask & Mask.QualifierMask) != 0,

"invalid specification in pointer qualifier list");

totalMask |= mask;

}

else {

Assert.error("invalid specification in pointer qualifier list");

}

}

type.setConstant((totalMask & Mask.Constant) != 0);

type.setVolatile((totalMask & Mask.Volatile) != 0);

}

return type;

}

}

### Struct and Union

A struct or union may be defined with an optional name tag and an optional member list. However, it may not lack both the name tag and member list.

Parser.bison

struct\_union\_specifier:

struct\_union optional\_identifier {

Scope scope = ($1 == Sort.Struct) ? Scope.Struct : Scope.Union;

GenerateDeclaration.generateStructUnionHeader($2, $1);

Main.CurrentTable = new SymbolTable(Main.CurrentTable, scope);

}

LEFT\_BLOCK declaration\_list RIGHT\_BLOCK {

$$ = GenerateDeclaration.generateStructUnionSpecifier($2, $1);

Main.CurrentTable = Main.CurrentTable.getParentTable();

}

| struct\_union IDENTIFIER {

$$ = GenerateDeclaration.lookupStructUnionSpecifier($2, $1);

};

struct\_union:

STRUCT { $$ = Sort.Struct; }

| UNION { $$ = Sort.Union; };

optional\_identifier:

/\* Empty. \*/ { $$ = null; }

| IDENTIFIER { $$ = $1; };

declaration\_list:

declaration

| declaration\_list declaration;

We extract the member list of the struct or union by accessing the current symbol table. If it is non-empty, the struct or union is complete (it has members) and we load the symbol list. If it is empty, the strucrt or union is incomplete and the symbol list is null.

If the struct or union has name, we need to check if there is another struct with the same name. If it is, first we need to check that they both are structs or unions (that they have the same sort) and that at least one of them is incomplete (the symbol list is null).

GenerateDeclaration.java

public static void generateStructUnionHeader(String optName, Sort sort){

if ((optName != null) && (Main.CurrentTable.lookupTag(optName) == null)){

Type newType = Type.createStructOrUnionType(sort, null, true);

Main.CurrentTable.addTag(optName, newType);

}

}

public static Type generateStructUnionSpecifier(String optName, Sort sort) {

Map<String,Symbol> memberMap = null;

if (!Main.CurrentTable.getEntryMap().isEmpty()) {

memberMap = new ListMap<>();

for (Object entry : Main.CurrentTable.getEntryMap().values()) {

Symbol symbol = (Symbol) entry;

memberMap.put(symbol.getName(), symbol);

}

}

if (optName != null) {

Type oldType = Main.CurrentTable.lookupTag(optName);

if (oldType != null) {

Assert.error(oldType.getSort() == sort,

optName, "tag already defined");

Assert.error((oldType.getMemberMap() == null) || (memberMap == null),

optName, "duplicate symbol");

if (oldType.getMemberMap() == null) {

oldType.setMemberMap(memberMap);

}

return oldType;

}

else {

Type newType = Type.createStructOrUnionType(sort, memberMap, true);

Main.CurrentTable.getParentTable().addTag(optName, newType);

return newType;

}

}

else {

return Type.createStructOrUnionType(sort, memberMap, true);

}

}

When accessing a named struct or union, we look it up in the tag table (not the symbol table) and make sure is has the right sort (struct, union, or enum).

public static Type lookupStructUnionSpecifier(String name, Sort sort) {

Type type = Main.CurrentTable.lookupTag(name);

if (type != null) {

type = Type.createStructOrUnionType(sort, type.getMemberMap(), true);

Assert.error(type.getSort() == sort, name, "invalid type");

}

else {

type = Type.createStructOrUnionType(sort, null, true);

Main.CurrentTable.addTag(name, type);

}

return type;

}

### Enum

When generating enums we use the Main.EnumValueStack, to keep track of the current enum value. We also push the symbol table.

Parser.bison

enum\_specifier:

ENUM optional\_identifier {

Main.EnumValueStack.push(BigInteger.ZERO);

}

LEFT\_BLOCK enum\_list RIGHT\_BLOCK {

$$ = GenerateDeclaration.generateEnumSpecifier($2, $5);

Main.EnumValueStack.pop();

}

| ENUM IDENTIFIER {

$$ = GenerateDeclaration.lookupEnum($2);

};

enum\_list:

enum {

$$ = new LinkedList<>();

((List) $$).add($1);

}

| enum\_list COMMA enum {

$1.add($3);

$$ = $1;

};

enum:

IDENTIFIER {

$$ = GenerateDeclaration.generateEnumItem($1, null);

}

| IDENTIFIER ASSIGN constant\_expression {

$$ = GenerateDeclaration.generateEnumItem($1, $3);

};

GenerateDeclaration.java

public static Symbol generateEnumItem(String itemName,

SyntaxTree optInitTree) {

Type itemType = Type.createArithmeticType(Sort.Signed\_Int);

itemType.setConstant(true);

BigInteger bigValue;

if (optInitTree != null) {

Main.EnumValueStack.pop();

bigValue = (BigInteger) optInitTree.getSymbol().getValue();

}

else {

bigValue = Main.EnumValueStack.pop();

}

Symbol itemSymbol = Symbol.createVariableSymbol(itemName, true,

Storage.Extern, itemType, bigValue);

Main.CurrentTable.addSymbol(itemSymbol);

Main.EnumValueStack.push(bigValue.add(BigInteger.ONE));

return itemSymbol;

}

public static Type generateEnumSpecifier(String optName,

List<Symbol> symbolList) {

Type type = Type.createEnumType(symbolList);

if (optName != null) {

Main.CurrentTable.addTag(optName, type);

}

return type;

}

public static Type lookupEnum(String name) {

Type type = Main.CurrentTable.lookupTag(name);

Assert.error(type != null, name, "unknown enum");

Assert.error(type.isEnum(), name, "not an enum");

return Type.createEnumType(type.getEnumerationList());

}

### Declarator

A declarator can be initialized with a value or marked as a bitfield (but not both).

Parser.bison

declarator\_list:

init\_bitfield\_declarator {

$$ = $1;

}

| declarator\_list COMMA init\_bitfield\_declarator {

MiddleCodeGenerator.appendMiddleCodeList($1, $3);

$$ = $1;

};

init\_bitfield\_declarator:

declarator {

GenerateDeclaration.generateDeclarator(Main.SpecifierStack.peek(), $1);

$$ = new LinkedList<>();

}

| declarator ASSIGN initializer {

$$ = new LinkedList<>();

GenerateDeclaration.generateAssignmentDeclarator

((List) $$, Main.SpecifierStack.peek(), $1, $3);

}

| declarator COLON constant\_expression {

GenerateDeclaration.generateBitfieldDeclarator

(Main.SpecifierStack.peek(), $1, $3);

$$ = new LinkedList<>();

}

| COLON constant\_expression {

GenerateDeclaration.generateBitfieldDeclarator

(Main.SpecifierStack.peek(), null, $2);

$$ = new LinkedList<>();

};

The generateDeclaration method takes the optional declarator list and for each declarator calls generateAssignmentDeclarator, generateBitfieldDeclarator, or generateDeclarator depending on whether the declarator comes with an assignment, a bitfield, or neither.

If the the declarator list is null, we check whether the specifiers type is enum or has a tag, if not it is meaningsless and a warning occur.

GenerateDeclaration.java

public static void generateDeclarator(Specifier specifier,

Declarator decl) {

Triple<Boolean,Storage,Type> specTriple = specifier.generate();

boolean external = specTriple.getFirst();

Storage storage = specTriple.getSecond();

Type specType = specTriple.getThird();

decl.add(specType);

String name = decl.getName();

Type type = decl.type();

if (type.isFunction()) {

Assert.error((storage == Storage.Static) || (storage == Storage.Extern),

null, "only static or extern storage allowed for function");

storage = Storage.Extern;

}

Assert.error(((Main.CurrentTable.getScope() != Scope.Struct) &&

(Main.CurrentTable.getScope() != Scope.Union)) ||

((storage == Storage.Auto)||(storage == Storage.Register)),

null, "only auto or register storage allowed in struct or union");

if (type.isFunction()) {

Symbol oldSymbol = Main.CurrentTable.lookupSymbol(name);

if (oldSymbol != null) {

Type oldType = oldSymbol.getType();

Assert.error(oldType.getReturnType().equals(type.getReturnType()),

name,"different return type in function redeclaration");

Assert.error(oldType.isEllipse() == type.isEllipse(), name,

"mixing ellipse function parameter in redeclaration");

List<Type> oldTypeList = oldType.getTypeList(),

newTypeList = type.getTypeList();

Assert.error(((oldTypeList == null) && (oldTypeList == null)) ||

((oldTypeList != null) && (oldTypeList != null) &&

oldTypeList.equals(newTypeList)), name,

"different parameter lists in function redeclaration");

}

}

Symbol symbol =

Symbol.createVariableSymbol(name, external, storage, type, null);

Main.CurrentTable.addSymbol(symbol);

}

The generateAssignmentDeclarator method is called for a declarator with an assignment. We check that the declarator is not located inside a struct or union (the scope of the symbol table is struct or union), since it is not allowed to initialize struct or union members. We also check that the declarator is not a function, since functions cannot be initialized. If the declarator is defined in the global scope and does not have the static storage it has external linkage and is visible from other source files, the linker will make it visible in a later phase.

An extern or typedef declarator cannot be initialized. However, if it is auto or register, we call GenerateInitializer.generateStack that generates a suitable series of assignment instructions. If it static, we instead call ValueConversion.storeStaticSymbol that will store the initialized value (that is always constant) in a block of bytes, that will be paced in the final execution file by the linker.

public static void generateAssignmentDeclarator

(List<MiddleCode> middleCodeList, Specifier specifier,

Declarator decl, Object init) {

Triple<Boolean,Storage,Type> specTriple = specifier.generate();

boolean external = specTriple.getFirst();

Storage storage = specTriple.getSecond();

Type specType = specTriple.getThird();

decl.add(specType);

String name = decl.getName();

Type type = decl.type();

Assert.error(!type.isFunction(), null, "function cannot be initialized");

Assert.error(storage != Storage.Typedef, name,

"typedef cannot be initialized");

Assert.error(storage != Storage.Extern, name,

"extern cannot be initialized");

Assert.error((Main.CurrentTable.getScope() != Scope.Struct) &&

(Main.CurrentTable.getScope() != Scope.Union),

name, "struct or union field cannot be initialized");

switch (storage) {

case Auto:

case Register: {

Assert.error(Main.CurrentTable.getScope() != Scope.Global, name,

"auto or register storage in global scope");

init = GenerateInitializerAuto.match(type, init);

init = GenerateInitializerAuto.generateAutoValueList(init);

Symbol symbol =

Symbol.createVariableSymbol(name, false, storage, type, null);

if (init instanceof SyntaxTree) {

SyntaxTree symbolTree =

new SyntaxTree(MiddleOperator.Variable, symbol);

SyntaxTree assignTree =

SyntaxTreeGenerator.generateBinaryTree(MiddleOperator.Assign,

type, symbolTree, (SyntaxTree)init);

MiddleCodeGenerator generateCode =

new MiddleCodeGenerator(middleCodeList);

generateCode.generateExpression(assignTree);

}

else {

GenerateInitializerAuto.generateAuto(symbol, middleCodeList, type,

init,name, Main.CurrentTable.getOffset());

}

Main.CurrentTable.addSymbol(symbol);

}

break;

case Static: {

init = GenerateInitializerStatic.generateValueExpression(init);

init = GenerateInitializerStatic.staticConvert(type, init);

init = GenerateInitializerStatic.generateAddressExpression(init);

if ((init instanceof SyntaxTree) &&

(((SyntaxTree) init).getOperator() == MiddleOperator.Variable)){

Main.GlobalSet.add(((SyntaxTree) init).getSymbol());

}

else {

List<Byte> byteList = new LinkedList<>();

Map<Integer,String> accessMap = new ListMap<>();

List<String> textList = new LinkedList<>();

GenerateInitializerStatic.generateStatic

(type, init, byteList, accessMap, textList);

Symbol symbol =

Symbol.createInitSymbol(name, external, storage, type,

byteList, accessMap, textList);

Main.CurrentTable.addSymbol(symbol);

}

}

break;

}

}

The generateBitfieldDeclarator method is called for a declarator with a bitfield. Opposite to the assignment case above, bitfields are only allowed in structs. Moreover, a struct member must be auto or register, if has no storage it is given auto storage. The bitfield value must be constant, and the number of bits is not allowed to exceed eight times the size of declaratory. Finally, it is poosible to define just a bitfield, without a declaratory. In that case, we just check that the bitfield value does not exceed eitgh times the size of a long integer.

The generateDeclarator method is called for a declarator without an assignment or a bitfield. If the declaratory is define in global space and is not static, it has external linkage. If is located in a struct or union, is has to be auto or register. If it is a function, it can only be extern or static, and if it has been declared earlier we check that the new declaratory has the same return type, parameter list, and ellipse status.

public static void generateBitfieldDeclarator(Specifier specifier,

Declarator decl, SyntaxTree bitsTree) {

Triple<Boolean,Storage,Type> specTriple = specifier.generate();

boolean external = specTriple.getFirst();

Storage storage = specTriple.getSecond();

Type specType = specTriple.getThird();

Assert.error(Main.CurrentTable.getScope() == Scope.Struct,

bitsTree, "bitfields only allowed on structs");

Assert.error((storage == Storage.Auto) || (storage == Storage.Register),

null, "only auto or register storage allowed in struct or union");

Object bitsValue = bitsTree.getSymbol().getValue();

int bits = ((BigInteger) bitsValue).intValue();

if (decl != null) {

decl.add(specType);

Type type = decl.type();

Assert.error(type.isIntegral(), type, "non-integral bits expression");

Assert.error((bits >= 1) && (bits <= (8 \* type.getSize())), bitsValue,

"bits value out of range");

type.setBitfieldMask(bits);

Symbol symbol = Symbol.createVariableSymbol(decl.getName(), external,

storage, type, null);

Main.CurrentTable.addSymbol(symbol);

}

else {

Assert.error((bits >= 1) && (bits <= (8 \* Type.LongSize)), bitsValue,

"bits value out of range");

}

}

### Pointers

A declarator can be repeatedly annotated with a pointer specifier, which can be constant or volatile.

Parser.bison

declarator:

optional\_pointer\_list direct\_declarator {

$$ = GenerateDeclaration.generatePointerDeclarator($1, $2);

};

direct\_declarator:

IDENTIFIER {

$$ = new Declarator($1);

}

| LEFT\_PAREN declarator RIGHT\_PAREN {

$$ = $2;

}

| direct\_declarator

LEFT\_SQUARE optional\_constant\_expression RIGHT\_SQUARE {

$$ = GenerateDeclaration.generateArrayType($1, $3);

}

| direct\_declarator

LEFT\_PAREN parameter\_ellipse\_list RIGHT\_PAREN {

$$ = GenerateDeclaration.generateNewFunctionDeclaration($1, $3);

}

| direct\_declarator

LEFT\_PAREN optional\_identifier\_list RIGHT\_PAREN {

$$ = GenerateDeclaration.generateOldFunctionDeclaration($1, $3);

};

optional\_pointer\_list:

/\* Empty. \*/ {

$$ = new LinkedList<>();

}

| pointer\_list {

$$ = $1;

};

pointer\_list:

pointer {

$$ = new LinkedList<>();

((List) $$).add($1);

}

| pointer\_list pointer {

$1.add($2);

$$ = $1;

};

pointer:

ASTERRISK {

$$ = Type.createPointerType(null);

}

| ASTERRISK declaration\_specifier\_list {

$$ = Main.SpecifierStack.pop().generateQualifiers();

};

Generator.java

public static Declarator generatePointerDeclarator(List<Type> typeList,

Declarator decl) {

if (decl == null) {

decl = new Declarator(null);

}

for (Type type : typeList) {

Type pointerType = Type.createPointerType(null);

pointerType.setConstant(type.isConstant());

pointerType.setVolatile(type.isVolatile());

decl.add(pointerType);

}

return decl;

}

public static Declarator generatePointerListDeclarator

(List<Pair<Boolean,Boolean>> pointerList, Declarator decl) {

for (Pair<Boolean,Boolean> pair : pointerList) {

Type pointerType = Type.createPointerType(null);

boolean isConstant = pair.getFirst(),

isVolatile = pair.getSecond();

pointerType.setConstant(isConstant);

pointerType.setVolatile(isVolatile);

decl.add(pointerType);

}

return decl;

}

### Parameter List

A parameter is a declaration with at most one declaratory, which has no initializer or bitfield.

Parser.bison

optional\_parameter\_ellipse\_list:

/\* Empty. \*/ { $$ = null; }

| parameter\_ellipse\_list { $$ = $1; };

parameter\_ellipse\_list:

parameter\_list {

$$ = new Pair<List<Symbol>,Boolean>($1, false);

}

| parameter\_list COMMA ELLIPSE {

$$ = new Pair<List<Symbol>,Boolean>($1, true);

};

parameter\_list:

{

++Main.ParameterCount;

}

parameter\_declaration {

--Main.ParameterCount;

$$ = new LinkedList<>();

((List) $$).add($2);

}

| parameter\_list COMMA {

++Main.ParameterCount;

}

parameter\_declaration {

--Main.ParameterCount;

$1.add($4);

$$ = $1;

};

parameter\_declaration:

declaration\_specifier\_list {

$$ = GenerateDeclaration.generateParameter

(Main.SpecifierStack.pop(), null);

}

| declaration\_specifier\_list declarator {

$$ = GenerateDeclaration.generateParameter

(Main.SpecifierStack.pop(), $2);

}

| declaration\_specifier\_list abstract\_declarator {

$$ = GenerateDeclaration.generateParameter

(Main.SpecifierStack.pop(), $2);

};

optional\_identifier\_list:

/\* Empty. \*/ {

$$ = new LinkedList<>();

}

| identifier\_list {

$$ = $1;

};

identifier\_list:

IDENTIFIER {

$$ = new LinkedList<>();

((List) $$).add($1);

}

| identifier\_list COMMA IDENTIFIER {

$1.add($3);

$$ = $1;

};

GenerateDeclaration.java

public static Symbol generateParameter(Specifier specifier,

Declarator decl) {

Triple<Boolean,Storage,Type> specTriple = specifier.generate();

Storage storage = specTriple.getSecond();

Type specType = specTriple.getThird();

Assert.error((storage == Storage.Auto) || (storage == Storage.Register),

"parameter storage must be auto or register");

if (decl != null) {

decl.add(specType);

Type type = toParameterType(decl.type());

return (Symbol.createParameterSymbol(decl.getName(), storage, type));

}

else {

Type type = toParameterType(specType);

return (Symbol.createParameterSymbol(null, storage, type));

}

}

private static Type toParameterType(Type type) {

switch (type.getSort()) {

case Array:

return Type.createPointerType(type.getArrayType());

case Function:

return Type.createPointerType(type);

default:

return type;

}

}

### Initializer

A declaratory can be initialized with a single value, or a (possible nested) list.

Parser.bison

initializer:

assignment\_expression {

$$ = $1;

}

| LEFT\_BLOCK initializer\_list optional\_comma RIGHT\_BLOCK {

$$ = $2;

};

optional\_comma:

/\* Empty. \*/

| COMMA;

initializer\_list:

initializer {

$$ = new LinkedList<>();

((List) $$).add($1);

}

| initializer\_list COMMA initializer {

$1.add($3);

$$ = $1;

};

### Type Name

A type name is a declaratory specifier list, with or without an abstract declaratory.

Parser.bison

type\_name:

declaration\_specifier\_list {

$$ = GenerateDeclaration.generateTypeName(Main.SpecifierStack.pop(), null);

}

| declaration\_specifier\_list abstract\_declarator {

$$ = GenerateDeclaration.generateTypeName(Main.SpecifierStack.pop(), $2);

};

GenerateDeclaration.java

public static Type generateTypeName(Specifier specifier,

Declarator decl) {

Triple<Boolean,Storage,Type> specTriple = specifier.generate();

Type specType = specTriple.getThird();

if (decl != null) {

decl.add(specType);

return decl.type();

}

else {

return specType;

}

}

}

### Direct and Abstract Declarator

A declaratory is a direct declarator with an optional pointer list, and there are three kinds of direct declarators:

* Arrays.
* Old-style Functions.
* New-style Functions.

Parser.bison

declarator:

optional\_pointer\_list direct\_declarator {

$$ = GenerateDeclaration.generatePointerDeclarator($1, $2);

};

direct\_declarator:

IDENTIFIER {

$$ = new Declarator($1);

}

| LEFT\_PAREN declarator RIGHT\_PAREN {

$$ = $2;

}

| direct\_declarator

LEFT\_SQUARE optional\_constant\_expression RIGHT\_SQUARE {

$$ = GenerateDeclaration.generateArrayType($1, $3);

}

| direct\_declarator

LEFT\_PAREN parameter\_ellipse\_list RIGHT\_PAREN {

$$ = GenerateDeclaration.generateNewFunctionDeclaration($1, $3);

}

| direct\_declarator

LEFT\_PAREN optional\_identifier\_list RIGHT\_PAREN {

$$ = GenerateDeclaration.generateOldFunctionDeclaration($1, $3);

};

An abstract declarator is a declaratory, with the difference that it does not have to have a name.

abstract\_declarator:

pointer\_list {

$$ = GenerateDeclaration.generatePointerDeclarator($1, null);

}

| optional\_pointer\_list direct\_abstract\_declarator {

$$ = GenerateDeclaration.generatePointerDeclarator($1, $2);

};

direct\_abstract\_declarator:

LEFT\_PAREN abstract\_declarator RIGHT\_PAREN {

$$ = $2;

}

| LEFT\_SQUARE optional\_constant\_expression RIGHT\_SQUARE {

$$ = GenerateDeclaration.generateArrayType(null, $2);

}

| direct\_abstract\_declarator

LEFT\_SQUARE optional\_constant\_expression RIGHT\_SQUARE {

$$ = GenerateDeclaration.generateArrayType($1, $3);

}

| LEFT\_PAREN optional\_parameter\_ellipse\_list RIGHT\_PAREN {

$$ = GenerateDeclaration.generateNewFunctionDeclaration(null, $2);

}

| direct\_abstract\_declarator

LEFT\_PAREN optional\_parameter\_ellipse\_list RIGHT\_PAREN {

$$ =

GenerateDeclaration.generateNewFunctionDeclaration($1, $3);

};

If the array size is given, we have to check it out. It shall be a constant integral with a value greater than zero. If the size is not given, we set it to zero; in that case, the type is incomplete.

GenerateDeclaration.java

public static Declarator generateArrayType(Declarator decl,

SyntaxTree optSizeTree) {

if (decl == null) {

decl = new Declarator(null);

}

if (optSizeTree != null) {

int arraySize = ((BigInteger)

optSizeTree.getSymbol().getValue()).intValue();

Assert.error(arraySize > 0, arraySize, "non-positive array size");

Type arrayType = Type.createArrayType(arraySize, null);

decl.add(arrayType);

}

else {

Type arrayType = Type.createArrayType(0, null);

decl.add(arrayType);

}

return decl;

}

In case of a new-style function declaratory, we need to check its elliptic status. If it is elliptic, it has to have at least one parameter. If it has one parameter that is void, the parameter list is regarded as empty. In that case, it cannot be elliptic.

Otherwise, if the parameter list is not empty and not just holding a void, we have to look into each parameter. If the parameter is named, we check that the name is not repeated in the parameter list. Except for the case above, a parameter cannot be void. A function parameter is changed to a pointer to the function, and an array parameter is changed to a pointer to the array type.

public static Declarator generateNewFunctionDeclaration(Declarator decl,

Pair<List<Symbol>,Boolean> paramPair) {

if (decl == null) {

decl = new Declarator(null);

}

List<Symbol> paramList = paramPair.getFirst();

boolean ellipse = paramPair.getSecond();

if (paramList.isEmpty()) {

Assert.error(!ellipse, "...",

"an elliptic function must have at least one parameter");

paramList = null;

}

else if ((paramList.size() == 1) && paramList.get(0).getType().isVoid()) {

Assert.error(paramList.get(0).getName() == null, paramList.get(0).getName(),

"a void parameter cannot be named");

Assert.error(!ellipse, "...",

"an elliptic function cannot have a void parameter");

paramList.clear();

}

else {

Set<String> nameSet = new ListSet<>();

for (Symbol symbol : paramList) {

String name = symbol.getName();

if (name != null) {

Assert.error(nameSet.add(name),

name, "repeated name in parameter list");

}

Type type = symbol.getType();

if (type.isVoid()) {

Assert.error(name, "invalid void parameter");

}

else if (type.isArray()) {

Type pointerType = Type.createPointerType(type.getArrayType());

pointerType.setConstant(true);

symbol.setType(pointerType);

}

else if (type.isFunction()) {

Type pointerType = Type.createPointerType(type);

pointerType.setConstant(true);

symbol.setType(pointerType);

}

}

}

if (paramList != null) {

decl.add(Type.createNewStyleFunctionType(null, paramList, ellipse));

}

else {

List<String> nameList = new LinkedList<String>();

decl.add(Type.createOldStyleFunctionType(null, nameList));

}

return decl;

}

An old-style function is simpler than a new-style function, we just must check that there is no repeated name in the parameter list.

public static Symbol generateParameter(Specifier specifier,

Declarator decl) {

Triple<Boolean,Storage,Type> specTriple = specifier.generate();

Storage storage = specTriple.getSecond();

Type specType = specTriple.getThird();

Assert.error((storage == Storage.Auto) || (storage == Storage.Register),

"parameter storage must be auto or register");

if (decl != null) {

decl.add(specType);

Type type = toParameterType(decl.type());

return (Symbol.createParameterSymbol(decl.getName(), storage, type));

}

else {

Type type = toParameterType(specType);

return (Symbol.createParameterSymbol(null, storage, type));

}

}

## Statements

### The Forward-Jump Problem (Backpatching)

When generating middle code instructions for expressions or statements, a common situation is that we generate forward jump instructions without yet knowing the target address. One way to solve the problem is to work with backpatching, which means that we use sets to keep track of jump instructions with yet unknown target addresses that eventually becomes filled in when the target addresses have become known.

For instance, let us look at the while statement to the left. The parsing of expression a < b will generate the middle code in the middle and the sets to the right. The middle code line will a have undefined targets since we at the moment does not know where to jump if the expression is true or false.

|  |  |  |
| --- | --- | --- |
| while (a < b)  ++a; | 1. if a < b goto ?  2. goto ? | a < b: true set = {1}  false set = {2} |

When parsing the while statement, the sets of the expression becomes backpatched: If the a < b expression is true, line 1 shall jump to line 3, and if it is false, line 2 shall jump to line 5. Whatever comes after the while statement.

|  |  |  |
| --- | --- | --- |
| 1. if a < b goto 3  2. goto 5  3. ++a  4. goto 1  5. ... |  |  |

If we instead let the while statement be surrounded by an if-else statement, we have both the true and false sets of the if expression x < y and the sets of the while expression a < b:

|  |  |  |
| --- | --- | --- |
| if (x < y)  while (a < b)  ++a;  else  ++b; | 1. if x < y goto ?  2. goto ?  3. if a < b goto ?  4. goto ?  5. ++a  6. goto 3  7. goto 9  8. ++b  9. ... | x < y: true set: {1}  false set: {2}  a < b: true set: {3}  false set: {4} |

When the if and while statements become parsed the following call will occur:

backpatch({1}, 3);

backpatch({2}, 8);

backpatch({3}, 5);

backpatch({4}, 7);

|  |  |  |
| --- | --- | --- |
| 1. if x < y goto 3  2. goto 8  3. if a < b goto 5  4. goto 7  5. ++a  6. goto 3  7. goto 9  8. ++b  9. ... |  |  |

The generated middle code above may appear ineffective, but the middle code optimizer of Chapter 14 will change it into:

|  |  |  |
| --- | --- | --- |
| 1. if x >= y goto 5  2. if a >= b goto 6  3. ++a  4. goto 2  5. ++b  6. ... |  |  |

Another example is the evaluation of a logical-and expression. Since C takes advantage of lazy evaluation, we have to insert jump instructions in the middle code.

|  |  |  |
| --- | --- | --- |
| if ((a < b) && (c < d))  ++a;  else  ++b; | 1. if a < b goto ?  2. goto ?  3. if c < d goto ?  4. goto ?  5. ++a  6. goto 7  7. ++b  8. ... |  |

When the first expression (a < b) is evaluated, its true set is {1} and its false set is {2}. When the second expression is evaluated the true set of the first expression is backpatched to the beginning of the second expression. This means that if the first expression is true we continue to evaluate the second expression. However, if the first expression is false the second expression will never be evaluated since it not necessary, the total expression will be false regardless of the value of the second expression. The result will be that the true set of the total expression is the one of the second expression and the false set is the union of the falses sets of both expressions.

|  |  |  |
| --- | --- | --- |
| 1. if a < b goto 3  2. goto ?  3. if c < d goto ?  4. goto ?  5. ++a  6. goto 7  7. ++b | true set: {3}  false set: {2, 4} |  |

In a similar way, the true set of an expression with logical-or will be the union of both true sets while the false set will the false set of the second expression.

|  |  |  |
| --- | --- | --- |
| if ((a < b) || (c < d))  ++a;  else  ++b; | 1. if a < b goto ?  2. goto 3  3. if c < d goto ?  4. goto ?  5. ++a  6. goto 7  7. ++b  8. ... | true set: {1, 3}  false set: {4} |

The Backpatch class holds a set of static methods that backpatch a single address or a set of addresses. The first two methods call the last two methods with the current size of ther middle code list as target address. In this way, the target address becomes the next middle code instruction to become generated. More specifically, the backpatching is performed by setting the third address of the middle code, since the third address is always the jump address in conditional and unconditional jump instructions.

Backpatch.java

### The if-else Problem (Open and Closed Statements)

The if-else problem[[2]](#footnote-2) is the problem of syntactically interpret the leftmost source code below. Semantically, the middle interpretation of the left statement is the correct one, each else shall be connected to the latest preceding if.

|  |  |  |
| --- | --- | --- |
| if (a < b)  if (c < d)  e = 1;  else  f = 2; |  |  |

Below is a simple set of statement rules. Unfortunately, they are ambiguous in that way that the an else does not have to be connected to the latest preceding if, resulting in both the middle and rightmost semantically interpretation above, depending in which order the rules are applied.

statement ::=

IF LEFT\_PAREN logical\_expression RIGHT\_PAREN statement

| IF LEFT\_PAREN logical\_expression RIGHT\_PAREN statement ELSE statement

| ...

To solve the problem, we need a more complicated set of rules that works with open and closed statements. The following set is unambiguous in that way that it always connects each else with the latest preceding if, with compiles to the semantics of C. A theoretical explanation of the open-closed statement solution is beyond the scope of this book, but I recommend the Dragon Book by Aho et al. for a closer look.

statement ::=

open\_statement

| closed\_statement;

open\_statement ::=

IF LEFT\_PAREN logical\_expression RIGHT\_PAREN statement

| IF LEFT\_PAREN logical\_expression RIGHT\_PAREN closed\_statement

ELSE open\_statement

| SWITCH LEFT\_PAREN expression RIGHT\_PAREN open\_statement

| CASE constant\_expression COLON open\_statement

| WHILE LEFT\_PAREN logical\_expression RIGHT\_PAREN open\_statement

| FOR LEFT\_PAREN optional\_expression SEMICOLON optional\_logical\_expression

SEMICOLON optional\_expression RIGHT\_PAREN open\_statement

| IDENTIFIER COLON open\_statement;

closed\_statement ::=

IF LEFT\_PAREN logical\_expression RIGHT\_PAREN closed\_statement

ELSE closed\_statement

| SWITCH LEFT\_PAREN expression RIGHT\_PAREN closed\_statement

| WHILE LEFT\_PAREN logical\_expression RIGHT\_PAREN closed\_statement

| FOR LEFT\_PAREN optional\_expression SEMICOLON optional\_logical\_expression

SEMICOLON optional\_expression RIGHT\_PAREN closed\_statement

| DO statement WHILE LEFT\_PAREN logical\_expression RIGHT\_PAREN SEMICOLON

| CASE constant\_expression COLON closed\_statement

| LEFT\_BLOCK optional\_statement\_list RIGHT\_BLOCK

| DEFAULT COLON closed\_statement

| RETURN optional\_expression SEMICOLON

| BREAK SEMICOLON

| CONTINUE SEMICOLON

| GOTO IDENTIFIER SEMICOLON

| optional\_expression SEMICOLON

| declaration;

Parser.bison

statement:

open\_statement { $$ = $1; }

| closed\_statement { $$ = $1; };

optional\_statement\_list:

/\* Empty. \*/ {

$$ = new LinkedList<SyntaxTree>();

}

| optional\_statement\_list statement {

$1.add($2);

$$ = $1;

};

SyntaxTreeGenerator.java

package c\_compiler;

import java.math.\*;

import java.util.\*;

public class SyntaxTreeGenerator {

public static SyntaxTree generateBlockStatement(List<SyntaxTree> list) {

SyntaxTree blockTree = new SyntaxTree(MiddleOperator.Block);

blockTree.getChildList().addAll(list);

return blockTree;

}

### Header Rules

The open statements include the compound statements if, if-else, switch, case, while, for, and label statements. The switch, while, and for statements need to do some preparation before the parsing, which means that they need one header rule each. Each rule calls its corresponding method in the Generate class that does the actual work of type checking and middle code generation.

Parser.bison

switch\_header:

SWITCH {

Main.CaseSetStack.push(new ListSet<BigInteger>());

Main.DefaultStack.push(false);

};

while\_header:

WHILE {

++Main.LoopCount;

};

for\_header:

FOR {

++Main.LoopCount;

};

### If Statements

The if statement backpatches the true set to the beginning of the statement and the false set to the middle code instruction following the statement. The if-else statement backpatches the true set to the beginning of the true statement and the false set to the beginning of the false statement.

Parser.bison

open\_statement:

IF LEFT\_PAREN expression RIGHT\_PAREN statement {

$$ = SyntaxTreeGenerator.generateIfStatement($3, $5);

}

| IF LEFT\_PAREN expression RIGHT\_PAREN

closed\_statement ELSE open\_statement {

$$ = SyntaxTreeGenerator.generateIfElseStatement($3, $5, $7);

}

GenerateDeclaration.java

public static SyntaxTree generateIfStatement(SyntaxTree expr,

SyntaxTree stat) {

SyntaxTree ifTree = new SyntaxTree(MiddleOperator.If);

ifTree.addChild(toLogicalExpression(expr));

ifTree.addChild(stat);

return ifTree;

}

public static SyntaxTree generateIfElseStatement(SyntaxTree expr,

SyntaxTree trueStat, SyntaxTree falseStat){

SyntaxTree ifElseTree = new SyntaxTree(MiddleOperator.IfElse);

ifElseTree.addChild(toLogicalExpression(expr));

ifElseTree.addChild(trueStat);

ifElseTree.addChild(falseStat);

return ifElseTree;

}

### Switch Statements

The prepareSwitchStatement method pushes the case map stack with an empty map and the default stack with minus one. The minus one value means that there has been no default statement so far.

Parser.bison

| switch\_header LEFT\_PAREN expression RIGHT\_PAREN open\_statement {

$$ = SyntaxTreeGenerator.generateSwitchStatement

($3,$5, Main.CaseSetStack.pop(),Main.DefaultStack.pop());

}

The generateSwitchStatement method generates middle code jump instructions for each of the case statements, which are stored in the Main.CaseMapStack stack. Each entry in the stack holds an integer value (each case expression value must be possible to evaluate in compile-time) and a line number. If the original switch expression equals the constant case expression, we jump to the start line of the statement following the case expression.

If there is a default statements, we finally jump to its start line if the switch statement does not match any of the case statements. Note that the default statement does not have to be placed after the case statements, even though I can see no good reason not to do so.

GenerateDeclaration.java

public static SyntaxTree generateSwitchStatement(SyntaxTree expr,

SyntaxTree stat, Set<BigInteger> caseSet,

boolean isDefault) {

Assert.error(expr.getType().isIntegralOrPointer(),

"not a pointer or integral type in switch statement");

Pair<Set<BigInteger>,Boolean> pair = new Pair<>(caseSet, isDefault);

SyntaxTree switchTree = new SyntaxTree(MiddleOperator.Switch, pair);

switchTree.addChild(expr);

switchTree.addChild(stat);

return switchTree;

}

private static Object generateValue(SyntaxTree expr) {

if (MiddleCode.isUnary(expr.getOperator())) {

Object subValue = generateValue(expr.getChild(0));

return ValueConversion.unaryCast(expr.getOperator(), subValue);

}

else if (MiddleCode.isBinary(expr.getOperator())) {

Object leftValue = generateValue(expr.getChild(0)),

rightValue = generateValue(expr.getChild(1));

return ValueConversion.binaryCast(expr.getOperator(),

leftValue, rightValue);

}

return null;

}

### Case Statements

The case statement must comply with the following demands:

* Main.CaseMapStack must not be empty. If it is empty, the case statements misses a surrounding switch statements.
* The case expression must be integral and constant and thereby possible to evaluate by the compiler.
* Since Main.CaseMapStack is not empty, there is at least one surrounding switch statement. We call peek on the stack to obtain the map holding the case expression value of the closest surrounding switch statement, add the case value with the line number of the case statement, and check that there is no earlier cases expression with the same value (put returns the old value if the key is already present in the map, null if is not[[3]](#footnote-3)).

Parser.bison

| CASE constant\_expression:caseSymbol COLON

line\_marker:statLine open\_statement {:

Generate.generateCaseStatement(caseSymbol, statLine);

:}

GenerateDeclaration.java

public static SyntaxTree generateCaseStatement(SyntaxTree valueTree,

SyntaxTree stat) {

Assert.error(!Main.CaseSetStack.empty(), valueTree, "unsurronded case");

BigInteger caseValue = (BigInteger) valueTree.getSymbol().getValue();

Assert.error(Main.CaseSetStack.peek().add(caseValue),

caseValue, "repeated case value");

SyntaxTree caseTree =

new SyntaxTree(MiddleOperator.Case, valueTree.getSymbol());

caseTree.addChild(stat);

return caseTree;

}

public static SyntaxTree generateDefaultStatement(SyntaxTree stat) {

Assert.error(!Main.DefaultStack.isEmpty(), "unsurronded default");

Assert.error(!Main.DefaultStack.pop(), "repeated default");

Main.DefaultStack.push(true);

SyntaxTree defaultTree = new SyntaxTree(MiddleOperator.Default);

defaultTree.addChild(stat);

return defaultTree;

}

### While Statements

The while header prepares the while statement by pushing break and continue stacks with integers sets. The continue set will eventually be backpatched to the beginning of the while statement and break set will be backpatched to the statement following the while statement.

The true set of the while expression is backpatched to the beginning of the statement (or the first statement of a block statement) inside the while loop while the false set is backpatched to the statement following the while statement, similar to the break set. In order to make sure that the while statement is followed by jump statement, the jump\_marker rule adds a jump statement at the end of the statement surrounded by the while statement, which is backpatched to the beginning of the while expression, similar to the continue set.

Parser.bison

| while\_header LEFT\_PAREN expression RIGHT\_PAREN open\_statement {

$$ = SyntaxTreeGenerator.generateWhileStatement($3, $5);

--Main.LoopCount;

}

GenerateDeclaration.java

public static SyntaxTree generateWhileStatement(SyntaxTree expr,

SyntaxTree stat) {

SyntaxTree whileTree = new SyntaxTree(MiddleOperator.While);

whileTree.addChild(toLogicalExpression(expr));

whileTree.addChild(stat);

return whileTree;

}

### For Statements

The for statement is a little bit more complicated than the while statement. To start with the continue and break stack are pushed, like the while case.

The for statement holds three optional expression: the initialization expression, the test expression, and the increment expression. The true and false sets of the initialization and increment expressions are all backpatched to beginning of the test expression. Like the while case, in order to make sure that the for statement is followed by jump statement the jump\_marker rule add a jump statement at the end of the statement surrounded by the for statement, which is also backpatched to the beginning of the test expression. There is also a jump line inserted after the test optional expression, which is backpatched to the beginning of the for statements to make sure that the for loop works properly even if the test expression has been omitted. An omitted test expression is equivalent to an infinite loop.

Finally, like the while case, the continue set is backpatched to the beginning of the test expression and the break set is backpatched to the beginning of the statement following the while statement.

Parser.bison

| for\_header LEFT\_PAREN optional\_expression SEMICOLON

optional\_expression SEMICOLON

optional\_expression RIGHT\_PAREN open\_statement {

$$ = SyntaxTreeGenerator.generateForStatement($3, $5, $7, $9);

--Main.LoopCount;

}

SyntaxTreeGeneration.java

public static SyntaxTree generateForStatement(SyntaxTree initExpr,

SyntaxTree testExpr, SyntaxTree nextExpr,

SyntaxTree stat) {

SyntaxTree forTree = new SyntaxTree(MiddleOperator.For);

forTree.addChild(initExpr);

forTree.addChild(toLogicalExpression(testExpr));

forTree.addChild(nextExpr);

forTree.addChild(stat);

return forTree;

}

### Label Statements

The label statement is quite simple, we just add the name of the label together with the line number of the beginning of the statement following the label to Main.LabelMap and check that the label has not been added already. The labels are used as targets of goto statements[[4]](#footnote-4).

Parser.bison

| IDENTIFIER COLON open\_statement {

$$ = SyntaxTreeGenerator.generateLabelStatement($1, $3);

};

GenerateDeclaration.java

public static SyntaxTree generateLabelStatement(String label,

SyntaxTree stat) {

Assert.error(Main.LabelSet.add(label), label, "repeated label");

SyntaxTree labelTree = new SyntaxTree(MiddleOperator.Label, label);

labelTree.addChild(stat);

return labelTree;

}

### Closed statement

The rules of the closed statements are similar to the open statements and the same Generate methods are called: generateIfElseStatement, generateSwitchStatement, generateWhileStatement, generateForStatement, and generateCaseStatement.

Parser.bison

closed\_statement:

IF LEFT\_PAREN expression RIGHT\_PAREN closed\_statement

ELSE closed\_statement {

$$ = SyntaxTreeGenerator.generateIfElseStatement($3, $5, $7);

}

| switch\_header LEFT\_PAREN expression RIGHT\_PAREN closed\_statement {

$$ = SyntaxTreeGenerator.generateSwitchStatement($3, $5, Main.CaseSetStack.pop(), Main.DefaultStack.pop());

}

| while\_header LEFT\_PAREN expression RIGHT\_PAREN closed\_statement {

$$ = SyntaxTreeGenerator.generateWhileStatement($3, $5);

--Main.LoopCount;

}

| for\_header LEFT\_PAREN optional\_expression SEMICOLON

optional\_expression SEMICOLON

optional\_expression RIGHT\_PAREN closed\_statement {

$$ = SyntaxTreeGenerator.generateForStatement($3, $5, $7, $9);

--Main.LoopCount;

}

| DO {

++Main.LoopCount;

} statement WHILE LEFT\_PAREN expression RIGHT\_PAREN SEMICOLON {

$$ = SyntaxTreeGenerator.generateDoStatement($6, $3);

--Main.LoopCount;

}

| CASE constant\_expression COLON closed\_statement {

$$ = SyntaxTreeGenerator.generateCaseStatement($2, $4);

}

### Do Statements

The do statement is a weaker version of the while statement. The difference is that the while expression may be false from the beginning resulting in zero iterations while the do expression is located at the end, resulting in at least one iteration. However, the do statement may in some cases present a simpler and more intuitive solution.

Like the while and for cases, the do statement is prepared by pushing the continue and break stacks. The true set of the do expression is backpatched to the beginning of the statement surrounded by the do statement, the false set is backpatched to the statement followed the do statement and finally the continue and break stacks are popped.

Parser.bison

| DO {

++Main.LoopCount;

} statement WHILE LEFT\_PAREN expression RIGHT\_PAREN SEMICOLON {

$$ = SyntaxTreeGenerator.generateDoStatement($6, $3);

--Main.LoopCount;

}

GenerateDeclaration.java

public static SyntaxTree generateDoStatement(SyntaxTree expr,

SyntaxTree stat) {

SyntaxTree doTree = new SyntaxTree(MiddleOperator.Do);

doTree.addChild(toLogicalExpression(expr));

doTree.addChild(stat);

return doTree;

}

### Block Statements

The block statement simple, a block is just a list of statements surrounded by brackets. Since a two variables with the same name can be defined in different blocks, we push the symbol table before the statement list.

Parser.bison

| LEFT\_BLOCK {

Main.CurrentTable = new SymbolTable(Main.CurrentTable, Scope.Block);

}

optional\_statement\_list RIGHT\_BLOCK {

$$ = SyntaxTreeGenerator.generateBlockStatement($3);

Main.CurrentTable = Main.CurrentTable.getParentTable();

}

GenerateDeclaration.java

public static SyntaxTree generateBlockStatement(List<SyntaxTree> list) {

SyntaxTree blockTree = new SyntaxTree(MiddleOperator.Block);

blockTree.getChildList().addAll(list);

return blockTree;

}

### Default Statements

The default statement have to comply with the following demands:

* Main.DefaultStack must not be empty. If it is empty, the default statements misses a surrounding switch statements.
* Since Main.DefaultStack is not empty, there is at least one surrounding switch statement. We call pop to see if the top value is minus one. If it is not, there has already been a default statement in the closest surrounding switch statement. If it is minus one, there has not been a earlier default statement and we push the line number at the beginning of the statement following the default statement. Since the line numbers are numbered from zero, it cannot be minus one.

Parser.bison

| DEFAULT COLON closed\_statement {

$$ = SyntaxTreeGenerator.generateDefaultStatement($3);

}

GenerateDeclaration.java

public static SyntaxTree generateDefaultStatement(SyntaxTree stat) {

Assert.error(!Main.DefaultStack.isEmpty(), "unsurronded default");

Assert.error(!Main.DefaultStack.pop(), "repeated default");

Main.DefaultStack.push(true);

SyntaxTree defaultTree = new SyntaxTree(MiddleOperator.Default);

defaultTree.addChild(stat);

return defaultTree;

}

### Break Statements

The break statement checks that Main.BreakSetStack is not empty, which would indicate that we do not have a surronding while, for, do, or switch statement, and add the line number of the beginning of the next statement to the set on the top of the stack similar to the continue case above. Note that the break statement unlike the continue statement may be surrondered by a switch statement.

Parser.bison

| BREAK SEMICOLON {

$$ = SyntaxTreeGenerator.generateBreakStatement();

}

GenerateDeclaration.java

public static SyntaxTree generateBreakStatement() {

Assert.error(!Main.CaseSetStack.empty() || (Main.LoopCount > 0),

"unsurronded break");

return (new SyntaxTree(MiddleOperator.Break));

}

### Continue Statements

The continue statement checks that Main.ContinueSetStack is not empty, which would indicate that we do not have a surrounding while, for, or do statement, and add the line number of the beginning of the next statement to the set on the top of the stack.

Parser.bison

| CONTINUE SEMICOLON {

$$ = SyntaxTreeGenerator.generateContinueStatement();

}

GenerateDeclaration.java

public static SyntaxTree generateContinueStatement() {

Assert.error(Main.LoopCount > 0, "unsurronded continue");

return (new SyntaxTree(MiddleOperator.Continue));

}

### Goto Statements

The goto statement stores the name of the label and the line number of the beginning of the following statement in Main.GotoMap. The gotos will eventlually be matched to the lables in Main.LabelMap.

Parser.bison

| GOTO IDENTIFIER SEMICOLON {

$$ = SyntaxTreeGenerator.generateGotoStatement($2);

}

GenerateDeclaration.java

public static SyntaxTree generateGotoStatement(String label) {

Main.GotoSet.add(label);

return (new SyntaxTree(MiddleOperator.Goto, label));

}

### Return Statements

The generateReturnStatement method is surprisely complicated, which due to the fact the return statement shall be interpreted as an exit statement in case of the main function.

Parser.bison

| RETURN optional\_expression SEMICOLON {

$$ = SyntaxTreeGenerator.generateReturnStatement($2);

}

If the return expression is not null, we that the surrounding function does not return void. Then we compare the expression type with the type of the surrounding function’s return type.

GenerateDeclaration.java

public static SyntaxTree generateReturnStatement(SyntaxTree optExpr) {

if (GenerateDeclaration.isMain(Main.FuncSymbol)) {

SyntaxTree exitTree = new SyntaxTree(MiddleOperator.Exit);

if (optExpr != null) {

optExpr = TypeCast.implicitCast

(optExpr, Main.FuncSymbol.getType().getReturnType());

exitTree.addChild(optExpr);

}

else {

Assert.error(Main.FuncSymbol.getType().getReturnType().isVoid(),

"return value of void function");

exitTree.addChild(null);

}

return exitTree;

}

else {

SyntaxTree returnTree = new SyntaxTree(MiddleOperator.Return);

if (optExpr != null) {

optExpr = TypeCast.implicitCast

(optExpr, Main.FuncSymbol.getType().getReturnType());

returnTree.addChild(optExpr);

}

else {

Assert.error(Main.FuncSymbol.getType().getReturnType().isVoid(),

"return value of void function");

returnTree.addChild(null);

}

return returnTree;

}

}

If the surrounding function returns a pointer to a function and the return expression is a or a pointer to a function, we check the they are equal; that is, that they return equal types and have equal parameter lists.

If the sorrounding function returns a pointer and the expression type is array, we compare the pointer type with array type. If the return expression is a string we check that the surrounding function’s return type is a pointer to a (signed or unsigned) character. Otherwise, we just cast the return expression to the surrounding function’s return type.

If the surrounding function is the main function, we cast the return expression to a signed short integer, since we already checked that main does not return void, it has to return an integral type. If the surrounding function is not main, we generate middle code instructions for the return value and function return. Note that there are two different instructions, where the first one sets the return value and the second one performs the return jump.

On the other hand, if there is no return expression we check that the surrounding function returns void. If it is the main function, we add the exit middle code instruction with argument zero. If it is not, we just add the return middle code instruction without precede it with the setting of a return value.

### Expression Statements

An expression statement is an optional expression followed by a semicolon. The expression is evaluated and its true and false sets are backpatched to the beginning of the next statement. Note that we do not use the result of the expression, the only interested part is its potential side effects. If it has no side effects, the statements will be removed by the middle code optimizer in Chapter 14.

Parser.bison

| optional\_expression SEMICOLON {

$$ = SyntaxTreeGenerator.generateExpressionStatement($1);

}

GenerateDeclaration.java

public static SyntaxTree generateExpressionStatement(SyntaxTree expr) {

SyntaxTree exprTree = new SyntaxTree(MiddleOperator.Expression);

exprTree.addChild(expr);

return exprTree;

}

### Declaration Statements

Declaration statements is an extension of the C standard, which states that the declarations are placed before the statements. With declaration statements it is possible to mix declarations and statements.

Parser.bison

| declaration {

$$ = SyntaxTreeGenerator.generateDeclarationStatement($1);

}

GenerateDeclaration.java

public static SyntaxTree generateDeclarationStatement

(List<MiddleCode> middleCodeList) {

return (new SyntaxTree(MiddleOperator.Declaration, middleCodeList));

}

### Load Register Statements

The closed statements do also include a set of internal statements; that is, statements not included in standard C, but necessary for the standard library functionality. The first statement is the load register statement, which stores a value in a register. The operands are the name of a register and an expression of integral or pointer type with same size as the register.

Parser.bison

| LOAD\_REGISTER LEFT\_PAREN IDENTIFIER

COMMA expression RIGHT\_PAREN SEMICOLON {

$$ = SyntaxTreeGenerator.generateLoadToRegisterStatement($3, $5);

}

GenerateDeclaration.java

public static SyntaxTree generateLoadToRegisterStatement(String name,

SyntaxTree expr) {

Assert.error(expr.getType().isIntegralOrPointer() ||

expr.getType().isArray(), expr,

"not an integral or a pointer type");

Register register = generateRegister(name);

int size = expr.getType().isArray() ? Type.PointerSize

: expr.getType().getSize();

Assert.error(ObjectCode.registerSize(register) == size,

expr, "incorrect expression type size");

SyntaxTree loadTree =

new SyntaxTree(MiddleOperator.LoadToRegister, register);

loadTree.addChild(expr);

return loadTree;

}

private static Register generateRegister(String registerName) {

try {

return Register.valueOf(registerName);

}

catch (IllegalArgumentException exception) {

Assert.error(registerName, "invalid register name");

return null;

}

}

### Store Register Statements

The store register statement takes as operands the name of a register and an expression of integral or pointer type with same size as the register. Moreover, since the purpose of this statement is to store the value of a register in a variable, the expression must be a left-value.

Parser.bison

| STORE\_REGISTER LEFT\_PAREN IDENTIFIER

COMMA expression RIGHT\_PAREN SEMICOLON {

$$ = SyntaxTreeGenerator.generateSaveFromRegisterStatement($3, $5);

}

GenerateSyntaxTree.java

public static SyntaxTree generateSaveFromRegisterStatement(String name,

SyntaxTree expr){

Assert.error(isLeftValue(expr), expr, "not a left-value");

Assert.error(expr.getType().isIntegralOrPointer(), expr,

"not an integral or a pointer type");

Register register = generateRegister(name);

Assert.error(ObjectCode.registerSize(register) ==

expr.getType().getSize(), expr,

"incorrect expression type size");

SyntaxTree saveTree =

new SyntaxTree(MiddleOperator.SaveFromRegister, register);

saveTree.addChild(expr);

return saveTree;

}

### Clear Registers Statements

The clear register statement simple clears the values of all register. However, it does not physically clear the values, but rather inform the object code generator of Chapter 15 that the values of the all registers shall be discarded.

Parser.bison

| CLEAR\_REGISTERS LEFT\_PAREN RIGHT\_PAREN SEMICOLON {

$$ = SyntaxTreeGenerator.generateClearRegistersStatement();

}

GenerateSyntaxTree.java

public static SyntaxTree generateClearRegistersStatement() {

return (new SyntaxTree(MiddleOperator.ClearRegisters));

}

### Store Flagbyte Statements

Parser.bison

| STORE\_FLAGBYTE LEFT\_PAREN expression RIGHT\_PAREN SEMICOLON {

$$ = SyntaxTreeGenerator.generateSaveFlagByteRegisterStatement($3);

}

GenerateSyntaxTree.java

public static SyntaxTree generateSaveFlagByteRegisterStatement

(SyntaxTree expr) {

Assert.error(isLeftValue(expr), expr, "not a left-value");

Assert.error(expr.getType().isIntegral(), expr, "not an integral type");

Assert.error(expr.getType().getSize() == Type.ShortSize, expr,

"incorrect expression type size");

SyntaxTree storeTree = new SyntaxTree(MiddleOperator.SaveFromFlagByte);

storeTree.addChild(expr);

return storeTree;

}

### Jump Register Statements

When performing a call to a function which address is stored in a pointer variables we need to store the address in a register and jump to that register.

Parser.bison

| JUMP\_REGISTER LEFT\_PAREN IDENTIFIER RIGHT\_PAREN SEMICOLON {

$$ = SyntaxTreeGenerator.generateJumpRegisterStatement($3);

}

GenerateSyntaxTree.java

public static SyntaxTree generateJumpRegisterStatement(String name) {

Register register = generateRegister(name);

return (new SyntaxTree(MiddleOperator.JumpRegister, register));

}

### Interrupt Statements

When making system calls, an interrupt occurs. The operand is an integral value of short size (1 byte).

Parser.bison

| INTERRUPT LEFT\_PAREN constant\_expression RIGHT\_PAREN SEMICOLON {

$$ = SyntaxTreeGenerator.generateInterruptStatement($3);

};

GenerateSyntaxTree.java

public static SyntaxTree generateInterruptStatement

(SyntaxTree interruptTree) {

Object value = interruptTree.getSymbol().getValue();

int intValue = ((BigInteger) value).intValue();

Assert.error((intValue >= 0) && (intValue <= 255),

"interrupt value out of range");

return (new SyntaxTree(MiddleOperator.Interrupt,

interruptTree.getSymbol()));

}

## Expressions

C has a large set of operators.

### Precedence and Associativity

C has fifteen levels of precedence. Note that the binary versions of addition and subtraction have lower precedence than their unary forms. Also, note that we consider the logical and/or operators not to be communicative, even that they in theory are communicative. The reason is that C apply lazy evaluation, which means that the expression is only evaluated to the degree necessary to determine its value. If the left operand is true in the or operator, the overall expression is evaluated to true without the right operand being evaluated, In the same way, if the left operand is false in the and operator, the overall expression is evaluated to false without the right operand being evaluated.

|  |  |  |  |
| --- | --- | --- | --- |
| **Operator** | **Precedence** | **Associativity** | **Communicative** |
| , | Lowest | Left | No |
| = += -= \*= /= %= &= ^= |= <<= >>= |  | Right | No |
| ? : |  | Left | No |
| || |  | Left | No (lazy evaluation) |
| && |  | Left | No (lazy evaluation) |
| | |  | Left | No |
| ^ |  | Left | No |
| & |  | Left | No |
| == != |  | Left | Yes |
| < <= > >= |  | Left | No |
| << >> |  | Left | No |
| +  - |  | Left  Left | Yes  No |
| \*  / % |  | Left  Left | Yes  No |
| ! ~ ++ -- + - \* & (type) sizeof |  | Right | No |
| () -> .  [] | Highest | Left  Left | No  Yes |

### Syntax Tree Generation Methods

GenerateSyntaxTree.java

public static SyntaxTree generateUnaryTree(MiddleOperator operator,

Type type, SyntaxTree childTree) {

SyntaxTree unaryTree = new SyntaxTree(operator, type);

unaryTree.addChild(childTree);

return unaryTree;

}

public static SyntaxTree generateBinaryTree(MiddleOperator operator,

Type type, SyntaxTree leftTree,

SyntaxTree rightTree) {

SyntaxTree binaryTree = new SyntaxTree(operator, type);

binaryTree.addChild(leftTree);

if (MiddleCode.isShift(operator)) {

rightTree = TypeCast.implicitCast(rightTree, Type.SignedCharType);

}

binaryTree.addChild(rightTree);

return binaryTree;

}

public static SyntaxTree generateValueTree(Symbol symbol) {

return (new SyntaxTree(MiddleOperator.Value, symbol));

}

public static SyntaxTree generateValueTree(Type type, Object value) {

Symbol symbol = Symbol.createValueSymbol(type, value);

return (new SyntaxTree(MiddleOperator.Value, symbol));

}

### Comma Expression

The comma operator has the lowest precedence. Both operands are evaluated and the result is the right operand, which means that the result of the left operand is discarded. The only meaningful application of the comma operator is to give the left operand a side effect. For instance:

++i, i + 1

Parser.bison

optional\_expression:

/\* Empty. \*/ { $$ = null; }

| expression { $$ = $1; };

expression:

assignment\_expression {

$$ = $1;

}

| expression COMMA assignment\_expression {

$$ = SyntaxTreeGenerator.generateCommaExpression($1, $3);

};

The only thing to do generateCommaExpression is to backpatch the true and false set of the left expression to the beginning of the right expression.

GenerateSyntaxTree.java

public static SyntaxTree generateCommaExpression(SyntaxTree leftTree,

SyntaxTree rightTree) {

return generateBinaryTree(MiddleOperator.Comma, rightTree.getType(),

leftTree, rightTree);

}

### Logical Expression

As stated in Section 0, there is no logical type in C. However, there are a number on occasions (if, while, do, while statements) when we need to test whether a value is true or false or. If the expression equals zero it is considered false, otherwise it is true. Since C supports lazy evaluation we need to generate two statements that perform jumps to two different addresses depending on whether the expression is true or false. If the expression is constant we can detmine its value alrady at compile-time and only generate one jump statement.

The generateLogicalExpression method makes sure the expression already is logical (the m\_sort field is Logical) or transforms into a logical expression. Every expression of integral, pointer, or floating type can be transferred into a logical expression. So can functions and arrays as they are implicitly transformed into pointers. The only types not possible to transform into logical expressions are structs and unions.

If the symbol is a function or array, it shall be true if its address does not equal zero. However, we do not need to actually compare the address since functions and arrays are always located at memory addresses greater than zero. Therefore, in the case of function or array, we consider them to always be true and add the next middle code address to the true set and let the false set be null.

When generating logical expression for integral, pointer, or floating expressions, we have tree cases to consider:

1. The expression is constant and equals zero. We generate an unconditional jump middle code instruction and add its address (the address is given by the current size of the Main.MiddleCodeList list) to the resulting false set, and set the result true set to null.
2. It is constant but does not equal zero. We do the opposite: we generate an unconditional jump middle code instruction and add its address to the resulting true set, and set the result false set to null.
3. It is not constant and therefore its value cannot be determined in compile-time. We add a conditional jump middle code instruction, where we compare the expression to zero, and add its address to the true set. We then add an unconditional jump statement and adds its address to the false set.

The jump instructions will later be backpatched to jump to suitable addresses depending on whether the expression is true or false.

Finally, if the symbol type is none of the above, it must be a struct or union, which are not possible to transform into logical type.

public static SyntaxTree toLogicalExpression(SyntaxTree tree) {

if (tree == null) {

return null;

}

Type type = tree.getType();

Assert.error(!type.isVoid(), "void",

"cannot be converted to logical type");

Assert.error(!type.isStruct(), "struct",

"cannot be converted to logical type");

Assert.error(!type.isUnion(), "union",

"cannot be converted to logical type");

if (type.isFunctionOrArray()) {

return generateValueTree(Type.LogicalType, Boolean.TRUE);

}

else if (type.isArithmeticOrPointer()) {

return generateUnaryTree(MiddleOperator.ArithmeticToLogical,

Type.LogicalType, tree);

}

else if (type.isLogical()) {

return tree;

}

else {

Assert.error("toLogical");

return null;

}

}

public static SyntaxTree checkLogicalExpression(SyntaxTree tree) {

if (tree.getType().isLogical()) {

return generateUnaryTree(MiddleOperator.LogicalToIntegral,

Type.SignedIntegerType, tree);

}

return tree;

}

### Integer Expressions

The generateIntegerExpression method can be considered the opposite of generateLogicalExpression above. There are a number on occasions where we need to transform a logical expression into an integer expression. In the example below the resulting symbol will be the temporary symbol $1.

|  |  |  |
| --- | --- | --- |
| 1. if a < b goto ?  2. goto ? | true set: {1}  false set: {2} | 1. if a < b goto 3  2. goto 5  3. $1 = 1  4. goto 6  5. $2 = 0  6. ... |

We have three cases to consider:

* If the false set is null, its logical expression is always true, and is transformed into a value symbol with the integer value one.
* In the same way, if the true set is null (both the true and false cannot be null) the logical expression is always false, and is transformed into a value symbol with the integer value zero.
* If both the true and false sets are not null, the logical expression is not constant and we need to generate code as the example above.

### Integral and Floating Promotion

If the type of both operands of a binary arithmetic expression are smaller than, both of them are converted to integers. This means that it is not possible to add two characters. They are both converted to integers, the resulting type is also integer. The goes for integral extra parameters in elliptic functions calls: each parameter with type less than integer is converted to integer. Note that this only applies to extra parameters, not regular parameters[[5]](#footnote-5).

In the same way, if a float value is given as extra parameter to an elliptic function, it is converted to double. Note that there is no equivalent floating promotion for float values in binary expressions. It is quite possible to add two float values without any of them being converted to double, with the resulting value also being float.

### Arithmetic Conversion

Simply put, arithmetic conversion is the process of converting the smaller type to the larger type in binary arithmetic expressions. More specifically, if the two operands have different types:

* If either operand is long double, the other operand is converted to long double.
* If either operand is double, the other operand is converted to double.
* If either operand is float, the other operand is converted to float.
* If both operands are integral, integral promotions is performed on both operands.
* If either operand is unsigned long, the other operand is converted to unsigned long.[[6]](#footnote-6)
* If either operand is signed long, the other operand is converted to signed long.
* If either operand is unsigned int, the other operand is converted to unsigned int.
* Otherwise, both operands have type signed int.

### Assignment Expression

There are two kinds of assignment: simple and compound. In the case of compound assignment, we first evaluate the corresponding binary expression (addition in the example below), assign its result to a temporary variable that we then assign to original left symbol.

|  |  |  |
| --- | --- | --- |
| x += y; | $1 = x + y  x = $1 |  |

Parser.bison

assignment\_expression:

conditional\_expression {

$$ = $1;

}

| prefix\_expression assignment\_operator assignment\_expression {

$$ = SyntaxTreeGenerator.generateAssignmentExpression($2, $1, $3);

};

assignment\_operator:

ASSIGN { $$ = MiddleOperator.Assign; }

| ADD\_ASSIGN { $$ = MiddleOperator.BinaryAddAssign; }

| SUBTRACT\_ASSIGN { $$ = MiddleOperator.BinarySubtractAssign; }

| MULTIPLY\_ASSIGN { $$ = MiddleOperator.MultiplyAssign; }

| DIVIDE\_ASSIGN { $$ = MiddleOperator.DivideAssign; }

| MODULO\_ASSIGN { $$ = MiddleOperator.ModuloAssign; }

| AND\_ASSIGN { $$ = MiddleOperator.BitwiseAndAssign; }

| IOR\_ASSIGN { $$ = MiddleOperator.BitwiseIOrAssign; }

| XOR\_ASSIGN { $$ = MiddleOperator.BitwiseXOrAssign; }

| LEFT\_SHIFT\_ASSIGN { $$ = MiddleOperator.LeftShiftAssign; }

| RIGHT\_SHIFT\_ASSIGN { $$ = MiddleOperator.RightShiftAssign; };

GenerateSyntaxTree.java

public static SyntaxTree generateAssignmentExpression

(MiddleOperator middleOp, SyntaxTree leftTree,

SyntaxTree rightTree) {

Assert.error(isLeftValue(leftTree), leftTree, "not a left-value");

rightTree = checkFunctionArrayStringLogicalExpression(rightTree);

rightTree = TypeCast.implicitCast(rightTree, leftTree.getType());

return generateBinaryTree(middleOp, leftTree.getType(),

leftTree, rightTree);

}

### Conditional Expression

The conditional operator applies lazy evaluation, which means that only on of the true or false expression will be evaluated.

Parser.bison

conditional\_expression:

logical\_or\_expression {

$$ = $1;

}

| logical\_or\_expression CONDITION expression COLON conditional\_expression {

$$ = SyntaxTreeGenerator.generateConditionExpression($1, $3, $5);

};

GenerateSyntaxTree.java

public static SyntaxTree generateConditionExpression

(SyntaxTree testTree, SyntaxTree trueTree,

SyntaxTree falseTree) {

testTree = toLogicalExpression(testTree);

Type maxType = TypeCast.maxType(trueTree.getType(), falseTree.getType());

trueTree = TypeCast.implicitCast(trueTree, maxType);

falseTree = TypeCast.implicitCast(falseTree, maxType);

SyntaxTree condTree = new SyntaxTree(MiddleOperator.Conditional, maxType);

condTree.addChild(testTree);

condTree.addChild(trueTree);

condTree.addChild(falseTree);

return condTree;

}

### Constant Expression

On several occasions in the grammar, a constant expression is required. Therefore, the constant\_expression rule returns a regular expression after checking that it actually is constant.

Parser.bison

optional\_constant\_expression:

/\* Empty. \*/ { $$ = null; }

| constant\_expression { $$ = $1; };

constant\_expression:

conditional\_expression {

SyntaxTree tree = SyntaxTreeOptimizer.optimizeExpression($1);

Assert.error(tree.getType().isIntegralOrPointer(),

tree, "non-integral expression");

Assert.error((tree.getSymbol() != null) && tree.getSymbol().hasValue(),

tree, "non-constant expression");

$$ = tree;

};

### Logical Expression

As described in Section 6.2.1, the logical-or expression backpatch the false set of the first operand to the beginning of the right operand. This can be interpreted as if the left expression is false, we try the right expression instead. If at least one of the expressions is true, the result is true. However, if the left expression is true, the result is determined to be true without evaluating the right expression.

Parser.bison

logical\_or\_expression:

logical\_and\_expression {

$$ = $1;

}

| logical\_or\_expression LOGICAL\_OR logical\_and\_expression {

$$ = SyntaxTreeGenerator.generateLogicalExpression

(MiddleOperator.LogicalOr, $1, $3);

};

GenerateSyntaxTree.java

public static SyntaxTree generateLogicalExpression(MiddleOperator operator,

SyntaxTree leftTree, SyntaxTree rightTree){

leftTree = toLogicalExpression(leftTree);

rightTree = toLogicalExpression(rightTree);

return generateBinaryTree(operator, Type.LogicalType,

leftTree, rightTree);

}

The logical-and expression backpatch the true set of the first operand to the beginning of the right operand. This can be interpreted as if the left expression is true, we try the right expression instead. If both the expressions are true, the result is true. However, if the left expression is false, the result is determined to be false without evaluating the right expression.

### Bitwise Expression

Unlike the logical expressions in the previous section, the inclucise-or, exclusive-or and and bitwise expression does not support lazy evaluation, which means that the right expressions is always evaluated regardless of the value of the left expression. Both the operands has to be integral and the result is integral.

Parser.bison

bitwise\_ior\_expression:

bitwise\_xor\_expression {

$$ = $1;

}

| bitwise\_ior\_expression BITWISE\_IOR bitwise\_xor\_expression {

$$ = SyntaxTreeGenerator.generateBitwiseExpression

(MiddleOperator.BitwiseIOr, $1, $3);

};

bitwise\_xor\_expression:

bitwise\_and\_expression {

$$ = $1;

}

| bitwise\_xor\_expression BITWISE\_XOR bitwise\_and\_expression {

$$ = SyntaxTreeGenerator.generateBitwiseExpression

(MiddleOperator.BitwiseXOr, $1, $3);

};

bitwise\_and\_expression:

equality\_expression {

$$ = $1;

}

| bitwise\_and\_expression AMPERSAND equality\_expression {

$$ = SyntaxTreeGenerator.generateBitwiseExpression

(MiddleOperator.BitwiseAnd, $1, $3);

};

GenerateSyntaxTree.java

public static SyntaxTree generateBitwiseNotExpression(SyntaxTree tree) {

tree = checkFunctionArrayStringLogicalExpression(tree);

Type type = tree.getType();

Assert.error(type.isIntegralOrPointer(), type,

"not allowed in bitwise not expression");

return generateUnaryTree(MiddleOperator.BitwiseNot, type, tree);

}

### Equality Expressions

All types except structs or unions can be compared with equal (==) and not equal (!=) operator. All arithmetic values can be compared and the arithmetic conversion of Section 6.3.7 is applied.

Parser.bison

equality\_expression:

relation\_expression {

$$ = $1;

}

| equality\_expression equality\_operator relation\_expression {

$$ = SyntaxTreeGenerator.generateEqualityExpression($2, $1, $3);

};

equality\_operator:

EQUAL { $$ = MiddleOperator.Equal; }

| NOT\_EQUAL { $$ = MiddleOperator.NotEqual; };

GenerateSyntaxTree.java

public static SyntaxTree generateEqualityExpression(MiddleOperator operator,

SyntaxTree leftTree, SyntaxTree rightTree) {

leftTree = checkFunctionArrayStringLogicalExpression(leftTree);

rightTree = checkFunctionArrayStringLogicalExpression(rightTree);

Type leftType = leftTree.getType(),

rightType = rightTree.getType();

Assert.error(leftType.isArithmeticOrPointer() ||

leftType.isStructOrUnion(), leftType,

"not allowed in relation expression");

Assert.error(rightType.isArithmeticOrPointer() ||

leftType.isStructOrUnion(), rightType,

"not allowed in relation expression");

Type maxType = TypeCast.maxType(leftType, rightType);

leftTree = TypeCast.implicitCast(leftTree, maxType);

rightTree = TypeCast.implicitCast(rightTree, maxType);

return generateBinaryTree(operator, Type.LogicalType,

leftTree, rightTree);

}

### Relation Expressions

All arithmetic and pointer types can be compared with relation operators.

Parser.bison

relation\_expression:

shift\_expression {

$$ = $1;

}

| relation\_expression relation\_operator shift\_expression {

$$ = SyntaxTreeGenerator.generateRelationalExpression($2, $1, $3);

};

relation\_operator:

LESS\_THAN { $$ = MiddleOperator.LessThan; }

| LESS\_THAN\_EQUAL { $$ = MiddleOperator.LessThanEqual; }

| GREATER\_THAN { $$ = MiddleOperator.GreaterThan; }

| GREATER\_THAN\_EQUAL { $$ = MiddleOperator.GreaterThanEqual; };

GenerateSyntaxTree.java

public static SyntaxTree generateRelationalExpression

(MiddleOperator operator,

SyntaxTree leftTree, SyntaxTree rightTree) {

leftTree = checkFunctionArrayStringLogicalExpression(leftTree);

rightTree = checkFunctionArrayStringLogicalExpression(rightTree);

Type leftType = leftTree.getType(),

rightType = rightTree.getType();

Assert.error(leftType.isArithmeticOrPointer(), leftType,

"not allowed in relation expression");

Assert.error(rightType.isArithmeticOrPointer(), rightType,

"not allowed in relation expression");

Type maxType = TypeCast.maxType(leftType, rightType);

leftTree = TypeCast.implicitCast(leftTree, maxType);

rightTree = TypeCast.implicitCast(rightTree, maxType);

return generateBinaryTree(operator, Type.LogicalType,

leftTree, rightTree);

}

### Shift Expression

Shift expressions are allowed on integral operands and applies the arithmetic conversion of Section 6.3.7.

Parser.bison

shift\_expression:

add\_expression {

$$ = $1;

}

| shift\_expression shift\_operator add\_expression {

$$ = SyntaxTreeGenerator.generateShiftExpression($2, $1, $3);

};

shift\_operator:

LEFT\_SHIFT { $$ = MiddleOperator.LeftShift; }

| RIGHT\_SHIFT { $$ = MiddleOperator.RightShift; };

GenerateSyntaxTree.java

public static SyntaxTree generateShiftExpression(MiddleOperator operator,

SyntaxTree leftTree, SyntaxTree rightTree){

leftTree = checkFunctionArrayStringLogicalExpression(leftTree);

rightTree = checkFunctionArrayStringLogicalExpression(rightTree);

Type leftType = leftTree.getType(),

rightType = rightTree.getType();

Assert.error(leftType.isIntegralOrPointer(), leftType,

"not allowed in bitwise expression");

Assert.error(rightType.isIntegralOrPointer(), rightType,

"not allowed in bitwise expression");

Type charType = rightType.isSigned() ? Type.SignedCharType

: Type.UnsignedCharType;

rightTree = TypeCast.explicitCast(rightTree, charType);

return generateBinaryTree(operator, leftTree.getType(),

leftTree, rightTree);

}

### Arithmetic Expression

Addition, subtraction, multiplications, and division are allowed for integral and arithmetic types. However, the modulo operator is only allowed for integral types. There are also two special cases: an integral value added to or subtracted from a pointer and subtraction between two pointers.

Parser.bison

add\_expression:

multiply\_expression {

$$ = $1;

}

| add\_expression binary\_add\_operator multiply\_expression {

$$ = SyntaxTreeGenerator.generateArithmeticExpression($2, $1, $3);

};

binary\_add\_operator:

PLUS { $$ = MiddleOperator.BinaryAdd; }

| MINUS { $$ = MiddleOperator.BinarySubtract; };

multiply\_expression:

cast\_expression {

$$ = $1;

}

| multiply\_expression multiply\_operator cast\_expression {

$$ = SyntaxTreeGenerator.generateArithmeticExpression($2, $1, $3);

};

multiply\_operator:

ASTERRISK { $$ = MiddleOperator.Multiply; }

| DIVIDE { $$ = MiddleOperator.Divide; }

| MODULO { $$ = MiddleOperator.Modulo; };

GenerateSyntaxTree.java

public static SyntaxTree generateArithmeticExpression

(MiddleOperator operator,

SyntaxTree leftTree, SyntaxTree rightTree) {

leftTree = checkFunctionArrayStringLogicalExpression(leftTree);

rightTree = checkFunctionArrayStringLogicalExpression(rightTree);

Type leftType = leftTree.getType(),

rightType = rightTree.getType();

if (operator == MiddleOperator.Modulo) {

Assert.error(leftType.isIntegralOrPointer(), leftType,

"not allowed in arithmetic expression");

Assert.error(rightType.isIntegralOrPointer(), rightType,

"not allowed in arithmetic expression");

}

else {

Assert.error(leftType.isArithmeticOrPointer(), leftType,

"not allowed in arithmetic expression");

Assert.error(rightType.isArithmeticOrPointer(), rightType,

"not allowed in arithmetic expression");

}

Type maxType = TypeCast.maxType(leftType, rightType);

leftTree = TypeCast.implicitCast(leftTree, maxType);

rightTree = TypeCast.implicitCast(rightTree, maxType);

return generateBinaryTree(operator, maxType, leftTree, rightTree);

}

When subtracting two pointers, they must point at values of the same type. Note that the result is number of values between the two pointers, not the number of bytes. This means that we must divide the pointer difference with the size of the pointer type (if it is greater than one).

When adding a value to a pointer, or subtracting a value from a pointer, we have the opposite problem compared to the pointer subtraction above: we need to multiply the integral value with the size of the pointer type (if it is greater than one), since the result shall be the number of values given by the integral value, not the number of bytes.

When it comes to regular arithmetic expressions not involving pointers.

### Cast Expression

In C, it is possible to cast expressions between integral, floating, and pointer expressions.

Parser.bison

cast\_expression:

prefix\_expression {

$$ = $1;

}

| LEFT\_PAREN type\_name RIGHT\_PAREN cast\_expression {

$$ = SyntaxTreeGenerator.generateCastExpression($4, $2);

};

GenerateSyntaxTree.java

public static SyntaxTree generateCastExpression(SyntaxTree fromTree,

Type toType) {

fromTree = checkFunctionArrayStringLogicalExpression(fromTree);

return TypeCast.explicitCast(fromTree, toType);

}

### Prefix Expression

In C, there are several prefix expressions:

* Unary add and minus
* Logical not
* Bitwise not
* The sizeof operator
* The address operator
* The derefence operator

Parser.bison

prefix\_add\_operator:

PLUS { $$ = MiddleOperator.UnaryAdd; }

| MINUS { $$ = MiddleOperator.UnarySubtract; };

increment\_operator:

INCREMENT { $$ = MiddleOperator.Increment; }

| DECREMENT { $$ = MiddleOperator.Decrement; };

prefix\_expression:

postfix\_expression {

$$ = $1;

}

### Unary Arithmetic Expression

The unary arithmetic operators (plus and minus) support all arithmetic types.

Parser.bison

| prefix\_add\_operator cast\_expression {

$$ = SyntaxTreeGenerator.generateUnaryExpression($1, $2);

}

GenerateSyntaxTree.java

public static SyntaxTree generateUnaryExpression(MiddleOperator operator,

SyntaxTree tree){

tree = checkFunctionArrayStringLogicalExpression(tree);

Type type = tree.getType();

Assert.error(type.isArithmeticOrPointer(), type,

"not allowed in unary expression");

return generateUnaryTree(operator, type, tree);

}

### Prefix Increment and Decrement

The prefix increment and decrement operators does first change the value and return the new value. The operand must be a left-value of arithmetic or pointer type.

Parser.bison

| increment\_operator prefix\_expression {

$$ = SyntaxTreeGenerator.generatePrefixIncrementExpression($1, $2);

}

GenerateSyntaxTree.java

public static SyntaxTree generatePrefixIncrementExpression

(MiddleOperator operator, SyntaxTree tree){

Assert.error(isLeftValue(tree), tree, "not left-value");

Type type = tree.getType();

Assert.error(type.isArithmeticOrPointer(), tree,

"not allowed in prefix increment/decrement");

if (operator == MiddleOperator.Increment) {

operator = MiddleOperator.PrefixIncrement;

}

else {

operator = MiddleOperator.PrefixDecrement;

}

return generateUnaryTree(operator, type, tree);

}

### Logical Not

The logical-not operator just swaps the true and false sets of its operand.

Parser.bison

| LOGICAL\_NOT cast\_expression {

$$ = SyntaxTreeGenerator.generateLogicalNotExpression($2);

}

GenerateSyntaxTree.java

public static SyntaxTree generateLogicalNotExpression(SyntaxTree tree) {

tree = toLogicalExpression(tree);

return generateUnaryTree(MiddleOperator.LogicalNot,

Type.LogicalType, tree);

}

### Bitwise Not

The bitwise-not operator applies to integral types.

Parser.bison

| BITWISE\_NOT cast\_expression {

$$ = SyntaxTreeGenerator.generateBitwiseNotExpression($2);

}

GenerateSyntaxTree.java

public static SyntaxTree generateBitwiseNotExpression(SyntaxTree tree) {

tree = checkFunctionArrayStringLogicalExpression(tree);

Type type = tree.getType();

Assert.error(type.isIntegralOrPointer(), type,

"not allowed in bitwise not expression");

return generateUnaryTree(MiddleOperator.BitwiseNot, type, tree);

}

### sizeof

The sizeof-operator takes an expression or a type inside parenthesis as operand. The operator is not allowed on function or bitfields.

Parser.bison

| SIZE\_OF prefix\_expression {

$$ = SyntaxTreeGenerator.generateSizeOfExpression($2);

}

| SIZE\_OF LEFT\_PAREN type\_name RIGHT\_PAREN {

$$ = SyntaxTreeGenerator.generateSizeOfType($3);

}

GenerateSyntaxTree.java

public static SyntaxTree generateSizeOfExpression(SyntaxTree tree) {

Type type = tree.getType();

Assert.error(!type.isFunction(), "sizeof function not allowed");

Assert.error(!type.isBitfield(), "sizeof bitfield not allowed");

return generateValueTree(Type.SignedIntegerType,

BigInteger.valueOf(type.getSize()));

}

public static SyntaxTree generateSizeOfType(Type type) {

Assert.error(!type.isFunction(), "sizeof function not allowed");

Assert.error(!type.isBitfield(), "sizeof bitfield not allowed");

return generateValueTree(Type.SignedIntegerType,

BigInteger.valueOf(type.getSize()));

}

### Address Expression

The address operator takes the address of its operand and the return type is a pointer to the operand type, unless it is an array, in which case the result is a pointer to the array type. The operand cannot have register storage.

Parser.bison

| AMPERSAND cast\_expression {

$$ = SyntaxTreeGenerator.generateAddressExpression($2);

}

GenerateSyntaxTree.java

public static SyntaxTree generateAddressExpression(SyntaxTree tree) {

Assert.error(isAddressable(tree), tree, "not addressable");

Type type = tree.getType();

Assert.error(!type.isBitfield(), tree, "address of bitfield not allowed");

Type baseType = type.isArray() ? type.getArrayType() : type;

Type pointerType = Type.createPointerType(baseType);

return generateUnaryTree(MiddleOperator.Address, pointerType, tree);

}

### Dereference Expression

The dereference operator takes a pointer or array as operand.

Parser.bison

| ASTERRISK cast\_expression {

$$ = SyntaxTreeGenerator.generateDerefExpression($2);

};

GenerateSyntaxTree.java

public static SyntaxTree generateDerefExpression(SyntaxTree tree) {

Type type = tree.getType();

Assert.error(type.isPointerOrArray(), tree, "not a pointer");

return generateUnaryTree(MiddleOperator.Deref,

type.getPointerOrArrayType(), tree);

}

### Postfix Expression

There are five kinds of postfix operators:

1. Postfix increment and decrement
2. Dot
3. Arrow
4. Index
5. Function Call

Parser.bison

postfix\_expression:

primary\_expression {

$$ = $1;

}

### Postfix Increment and Decrement

The postfix increment and decrement change the operand and return the original value. The operand has to be a left-value of arithmetic or pointer type.

Parser.bison

| postfix\_expression increment\_operator {

$$ = SyntaxTreeGenerator.generatePostfixIncrementExpression($2, $1);

}

GenerateSyntaxTree.java

public static SyntaxTree generatePostfixIncrementExpression

(MiddleOperator operator, SyntaxTree tree){

Assert.error(isLeftValue(tree), tree, "not left-value");

Type type = tree.getType();

Assert.error(type.isArithmeticOrPointer(), tree,

"not allowed in postfix increment/decrement");

if (operator == MiddleOperator.Increment) {

operator = MiddleOperator.PostfixIncrement;

}

else {

operator = MiddleOperator.PostfixDecrement;

}

return generateUnaryTree(operator, type, tree);

}

### Dot

The dot operator generates different result is the operand has and address symbol. If it has, it is the result of the arrow, dereference, or index operator. In that case, it inherits the address symbol and increase the address offset with the member offset.

If the operand does not have an address symbol, it is a regular variable. Then we just create a temporary symbol with the same storage and increase the offset with the member offset.

Parser.bison

| postfix\_expression DOT IDENTIFIER {

$$ = SyntaxTreeGenerator.generateDotExpression($1, $3);

}

GenerateSyntaxTree.java

public static SyntaxTree generateDotExpression(SyntaxTree tree, String name){

Type type = tree.getType();

Assert.error(type.isStructOrUnion(), tree, "not a struct or union");

Symbol memberSymbol = type.lookupMember(name);

Assert.error(memberSymbol != null, name, "not a member");

SyntaxTree memberTree =

new SyntaxTree(MiddleOperator.Variable, memberSymbol);

return generateBinaryTree(MiddleOperator.Dot, memberSymbol.getType(),

tree, memberTree);

}

### Arrow

The arrow operator takes a pointer to a struct or union as operand. The result symbol has the operand as address symbol and the member offset as offset.

Parser.bison

| postfix\_expression ARROW IDENTIFIER {

$$ = SyntaxTreeGenerator.generateArrowExpression($1, $3);

}

GenerateSyntaxTree.java

public static SyntaxTree generateArrowExpression(SyntaxTree tree,

String name) {

Type type = tree.getType();

Assert.error(type.isPointer() && type.getPointerType().isStructOrUnion(),

tree, "not a pointer to struct or a union");

Symbol memberSymbol = type.getPointerType().lookupMember(name);

Assert.error(memberSymbol != null, name, "not a member");

SyntaxTree memberTree = new SyntaxTree(MiddleOperator.Variable,

memberSymbol);

return generateBinaryTree(MiddleOperator.Arrow, memberSymbol.getType(),

tree, memberTree);

}

### Index

The index operator takes a pointer or array and an integral. We have three cases:

1. The index is constant, in which case we treat the operator as if it was the arrow operator.
2. The index is not constant and the array or pointer type size is one, in which case we generate that add the index to the pointer or array.
3. The index is not constant and the array or pointer type size is greater than, in which case we also need to generate code that multiply the index with the type size.

Parser.bison

| postfix\_expression LEFT\_SQUARE expression RIGHT\_SQUARE {

$$ = SyntaxTreeGenerator.generateIndexExpression($1, $3);

}

GenerateSyntaxTree.java

public static SyntaxTree generateIndexExpression(SyntaxTree leftTree,

SyntaxTree rightTree) {

SyntaxTree pointerTree = null, indexTree = null;

Type leftType = leftTree.getType(),

rightType = rightTree.getType();

if (leftType.isPointerOrArray() && rightType.isIntegral()) {

pointerTree = leftTree;

indexTree = rightTree;

}

else if (leftType.isIntegral() && rightType.isPointer()) {

pointerTree = rightTree;

indexTree = leftTree;

}

else {

Assert.error(null, "invalid index types");

}

if (!indexTree.getType().isInteger()) {

Type toType = indexTree.getType().isSigned() ? Type.SignedIntegerType

: Type.UnsignedIntegerType;

indexTree = TypeCast.explicitCast(indexTree, toType);

}

Type returnType = pointerTree.getType().getPointerOrArrayType();

return generateBinaryTree(MiddleOperator.Index, returnType,

pointerTree, indexTree);

}

### Function Call

When calling a function, first we need to check the number of actual and formal parameters, which shall be equal. Unless the function is elliptic, in which case the actual parameter list can be longer.

Each regular actual parameter is transformed to the type of the formal parameter, and each extra actual parameter is subjected to the type promotion described in Section 6.3.6. Unless the return type is void, we place the result value in a temporary symbol.

Parser.bison

| postfix\_expression

LEFT\_PAREN optional\_argument\_expression\_list RIGHT\_PAREN {

$$ = SyntaxTreeGenerator.generateFunctionCallExpression($1, $3);

};

GenerateSyntaxTree.java

public static SyntaxTree generateFunctionCallExpression(SyntaxTree tree,

List<SyntaxTree> argumentList){

Type type = tree.getType(), returnType = null;

List<Type> typeList = null;

boolean isEllipse = false;

if (type.isFunction()) {

typeList = type.getTypeList();

returnType = type.getReturnType();

isEllipse = type.isEllipse();

}

else if (type.isPointer() && type.getPointerType().isFunction()) {

typeList = type.getPointerType().getTypeList();

returnType = type.getPointerType().getReturnType();

isEllipse = type.getPointerType().isEllipse();

}

else {

Assert.error(tree, "not a function");

}

SyntaxTree callTree = generateUnaryTree(MiddleOperator.Call,

returnType, tree);

int extra = 0;

if (typeList != null) {

Assert.error((argumentList.size() == typeList.size()) ||

(isEllipse && (argumentList.size() > typeList.size())),

tree, "invalid number of parameters");

for (int index = 0; index < typeList.size(); ++index) {

SyntaxTree argTree =

checkFunctionArrayStringLogicalExpression(argumentList.get(index));

callTree.addChild(TypeCast.implicitCast(argTree,

typeList.get(index)));

}

for (int index = typeList.size(); index < argumentList.size(); ++index){

SyntaxTree paramTree = convertArgument(argumentList.get(index));

callTree.addChild(paramTree);

extra += paramTree.getType().getSize();

}

}

else {

for (SyntaxTree argumentTree : argumentList) {

SyntaxTree paramTree = convertArgument(argumentTree);

callTree.addChild(paramTree);

extra += paramTree.getType().getSize();

}

}

callTree.setInfo(extra);

return callTree;

}

Each actual parameter is checked. For a regular parameter (paramType is not null), a function is converted to a pointer to a function, otherwise implicit type conversion occurs. For an extra parameter, a function, array, or string is converted to a pointer, otherwise argument promotion occurs.

For an extra parameter: a char is converted to an integer and a float is converted to a double.

private static SyntaxTree convertArgument(SyntaxTree tree) {

tree = checkFunctionArrayStringLogicalExpression(tree);

switch (tree.getType().getSort()) {

case Signed\_Char:

case Signed\_Short\_Int:

return TypeCast.implicitCast(tree, Type.SignedIntegerType);

case Unsigned\_Char:

case Unsigned\_Short\_Int:

return TypeCast.implicitCast(tree, Type.UnsignedIntegerType);

case Float:

return TypeCast.implicitCast(tree, Type.DoubleType);

default:

return tree;

}

}

### Parameter Argument List

Each parameter is converted from a possible logical type to an integer by calling Generate.generateIntegerExpression.

Parser.bison

optional\_argument\_expression\_list:

/\* Empty. \*/ {

$$ = new LinkedList<>();

}

| argument\_expression\_list {

$$ = $1;

};

argument\_expression\_list:

assignment\_expression {

$$ = new LinkedList<>();

((List) $$).add($1);

}

| argument\_expression\_list COMMA assignment\_expression {

$1.add($3);

$$ = $1;

};

### Primary Expression

A primary expression is an identifier, which symbol we look up in the symbol table, a value, which we store in the global space, or an expression surrounded by parenthesis.

Parser.bison

primary\_expression:

IDENTIFIER {

$$ = SyntaxTreeGenerator.generateVariableExpression($1);

}

| VALUE {

$$ = SyntaxTreeGenerator.generateValueTree($1);

//Main.LinkerSet.add(new LinkerInfo($1));

}

| LEFT\_PAREN expression RIGHT\_PAREN {

$$ = $2;

};

GenerateSyntaxTree.java

public static SyntaxTree generateVariableExpression(String name) {

Symbol symbol = Main.CurrentTable.lookupSymbol(name);

Assert.error(symbol != null, name, "unknown variable");

Assert.error(symbol.getStorage() != Storage.Typedef,

name, "typedef name in expression");

symbol.setReferred(true);

return (new SyntaxTree(MiddleOperator.Variable, symbol));

}

}

# The Type System

C has a rather large set of types. The simple types are made up of the integral types signed and unsigned char, short, int, and long as well as the floating types float, double, and long double. The compound type are pointers, arrays, structs, unions, and functions. Values of enumeration types (enum) are stored as signed integer. Moreover, a type can also be constant or volatile.

## The Type Class

The Type class hold information about a type. The default constructor is private, which makes it impossible to explicitly create type objects. Instead, there is a series of static create-methods that create and return a Type object.

Type.java

package c\_compiler;

import java.math.\*;

import java.util.\*;

public class Type {

public static Type SignedIntegerType =

Type.createArithmeticType(Sort.Signed\_Int);

public static Type UnsignedIntegerType =

Type.createArithmeticType(Sort.Unsigned\_Int);

public static Type DoubleType = Type.createArithmeticType(Sort.Double);

public static Type LogicalType = Type.createArithmeticType(Sort.Logical);

public static Type SignedCharType =

Type.createArithmeticType(Sort.Signed\_Char);

public static Type UnsignedCharType =

Type.createArithmeticType(Sort.Unsigned\_Char);

public static Type StringType = Type.createArithmeticType(Sort.String);

public static Type VoidType = Type.createArithmeticType(Sort.Void);

public static Type PointerType = Type.createPointerType(SignedIntegerType);

### Arithmetic Types

An arithmetic type is either integral or floating, and is determined by the m\_sort field.

Sort.java

package c\_compiler;

public enum Sort {Void, Boolean, Signed\_Char, Unsigned\_Char,

Signed\_Short\_Int, Unsigned\_Short\_Int, Signed\_Int,

Unsigned\_Int, Signed\_Long\_Int, Unsigned\_Long\_Int,

Float, Double, Long\_Double, String,

Pointer, Array, Struct, Union, Enumeration,

Function, Logical};

Type.java

private Sort m\_sort;

public Sort getSort() {

return m\_sort;

}

public static Type createArithmeticType(Sort sort) {

Type type = new Type();

type.m\_sort = sort;

return type;

}

### Enumeration Types

The enumeration type (enum) is stored as an assigned integer. However, the Specifier class of Section 6.1.2 needs to know if the type is enum to initialize its value. Therefore, we add the m\_enum field and the isEnumeration method.

Type.java

private List<Symbol> m\_enumerationList;

public static Type createEnumerationType(List<Symbol> enumerationList) {

Type type = new Type();

type.m\_sort = Sort.Enumeration;

type.m\_enumerationList = enumerationList;

return type;

}

public List<Symbol> getEnumerationList() {

return m\_enumerationList;

}

### Bitfield Types

The bitfield type is basically an integral type, with the addition of the bitfield mask that is used when a bitfield variable is assigned a value to set the unused bits to zero.

Type.java

private BigInteger m\_bitfieldMask = null;

public boolean isBitfield() {

return (m\_bitfieldMask != null);

}

public BigInteger getBitfieldMask() {

return m\_bitfieldMask;

}

public void setBitfieldMask(int bits) {

m\_bitfieldMask = BigInteger.valueOf(2).pow(bits).subtract(BigInteger.ONE);

}

### Logical Types

Contrary to some other languages, there is no logical type in C. However, as C applies lazy evaluation, we need a logical type. Lazy evaluation means that an expression shall be not be evaluated more than necessary to determine its value. More specifically, if the left operand of the logical or-operator is true the right operand shall not be evaluated. In the same way, if the left operand of the logical and-operator is false the right operand shall not be evaluated. The same goes for the conditional operator, depending on whether the first expression is true or false, only the second or the third expression is evaluated. The logical type holds the two sets m\_trueSet and m\_falseSet, which hold middle code addresses to jumps instructions that will later be filled with the addresses to jump to if the expression is true or false.

Type.java

private Set<Integer> m\_trueSet, m\_falseSet;

public static Type createLogicalType(Set<Integer> trueSet,

Set<Integer> falseSet) {

Type logicalType = new Type();

logicalType.m\_sort = Sort.Logical;

logicalType.m\_trueSet = trueSet;

logicalType.m\_falseSet = falseSet;

return logicalType;

}

public Set<Integer> getTrueSet() {

return m\_trueSet;

}

public Set<Integer> getFalseSet() {

return m\_falseSet;

}

### Pointer Types

The type pointed at is null when the type is created, it will later be set by the declarator type.

Type.java

private Type m\_pointerType;

public static Type createPointerType(Type pointerType) {

Type type = new Type();

type.m\_sort = Sort.Pointer;

type.m\_pointerType = pointerType;

return type;

}

public Type getPointerType() {

return m\_pointerType;

}

public void setPointerType(Type pointerType) {

m\_pointerType = pointerType;

}

### Array Types

The array size can be zero, when the type is created. In that case it will later be set by the length of its initialization list or become uncomplete.

private int m\_arraySize;

private Type m\_arrayType;

public static Type createArrayType(int arraySize, Type arrayType) {

Type type = new Type();

type.m\_sort = Sort.Array;

type.m\_arraySize = arraySize;

type.m\_arrayType = arrayType;

return type;

}

public int getArraySize() {

return m\_arraySize;

}

public void setArraySize(int arraySize) {

m\_arraySize = arraySize;

}

public Type getArrayType() {

return m\_arrayType;

}

public void setArrayType(Type arrayType) {

m\_arrayType = arrayType;

}

public Type getPointerOrArrayType() {

return (m\_sort == Sort.Pointer) ? m\_pointerType : m\_arrayType;

}

### Function Types

C support both old-style and new-style function declarations. The old-style declaration takes a parameter list of identifiers, that is matched to a set of declarations, while the new-style declaration takes a parameter list of declarations.

Type.java

private Type m\_returnType;

private List<String> m\_nameList;

private List<Symbol> m\_parameterList;

private List<Type> m\_typeList;

private boolean m\_ellipse;

public static Type createOldStyleFunctionType(Type returnType,

List<String> nameList) {

Type type = new Type();

type.m\_sort = Sort.Function;

type.m\_returnType = returnType;

type.m\_nameList = nameList;

type.m\_parameterList = null;

type.m\_ellipse = false;

Set<String> nameSet = new ListSet<>();

for (String name : type.m\_nameList) {

Assert.error(nameSet.add(name), name,"duplicate name in parameter list");

}

return type;

}

public static Type createNewStyleFunctionType(Type returnType,

List<Symbol> parameterList, boolean ellipse) {

Type type = new Type();

type.m\_sort = Sort.Function;

type.m\_returnType = returnType;

type.m\_nameList = null;

type.m\_parameterList = parameterList;

type.m\_ellipse = ellipse;

for (Symbol symbol : parameterList) {

type.m\_typeList.add(symbol.getType());

}

return type;

}

public boolean isOldStyleFunction() {

return (m\_nameList != null);

}

public boolean isNewStyleFunction() {

return (m\_parameterList != null);

}

public List<String> getNameList() {

return m\_nameList;

}

public List<Symbol> getParameterList() {

return m\_parameterList;

}

public List<Type> getTypeList() {

return m\_typeList;

}

public Type getReturnType() {

return m\_returnType;

}

public void setReturnType(Type returnType) {

m\_returnType = returnType;

}

public boolean isEllipse() {

return m\_ellipse;

}

### Struct and Union Types

A struct and union takes a list of symbols and an indication whether the struct or union is tagged. A struct or union is tagged it is given a name:

|  |  |
| --- | --- |
| struct {  int a, b;  }; | struct s {  int a, b;  }; |
| (a) An untagged struct | (b) A struct tagged with the name “s” |

The only reason for including the m\_hasTag field is that an untagged struct or union with no defined variables shall raise a warning, since it is a meaningless declaration:

struct s {int i;}; // Meaningful, s can later be used to define variables.

struct {int i;} t; // Meaningful, t is a defined variable.

struct {int i;}; // Meaningless, shall raise a warning.

Type.java

private boolean m\_hasTag;

private Map<String,Symbol> m\_memberMap;

public static Type createStructOrUnionType(Sort sort,

Map<String,Symbol> symbolMap, boolean hasTag) {

Type type = new Type();

type.m\_sort = sort;

type.m\_hasTag = hasTag;

type.m\_memberMap = symbolMap;

return type;

}

public boolean hasTag() {

return m\_hasTag;

}

public Symbol lookupMember(String name) {

return m\_memberMap.get(name);

}

public Map<String,Symbol> getMemberMap() {

return m\_memberMap;

}

public void setMemberMap(Map<String,Symbol> memberMap) {

m\_memberMap = memberMap;

}

### Type Size

Each type has a size, even though void and function have size zero. The size of an array is its size times the size of its type, the size of a struct is the sum of the sizes of its members, and the size of a union is the size of its largest member. Note that a pointer always has the same size, regardless of what it points at.

Type.java

public static final int CharSize = 1;

public static final int ShortSize = 1;

public static final int IntegerSize = 2;

public static final int LongSize = 4;

public static final int FloatSize = 4;

public static final int DoubleSize = 8;

public static final int LongDoubleSize = 8;

public static final int PointerSize = 2;

private static Map<Sort,Integer> m\_sizeMap = new ListMap<>();

static {

m\_sizeMap.put(Sort.Logical, IntegerSize);

m\_sizeMap.put(Sort.Signed\_Char, CharSize);

m\_sizeMap.put(Sort.Unsigned\_Char, CharSize);

m\_sizeMap.put(Sort.Signed\_Short\_Int, ShortSize);

m\_sizeMap.put(Sort.Unsigned\_Short\_Int, ShortSize);

m\_sizeMap.put(Sort.Signed\_Int, IntegerSize);

m\_sizeMap.put(Sort.Unsigned\_Int, IntegerSize);

m\_sizeMap.put(Sort.Signed\_Long\_Int, LongSize);

m\_sizeMap.put(Sort.Unsigned\_Long\_Int, LongSize);

m\_sizeMap.put(Sort.Float, FloatSize);

m\_sizeMap.put(Sort.Double, DoubleSize);

m\_sizeMap.put(Sort.Long\_Double, LongDoubleSize);

}

public int getSize() {

switch (m\_sort) {

case Pointer:

case String:

return PointerSize;

case Array:

return m\_arraySize \* m\_arrayType.getSize();

case Struct: {

int size = 0;

for (Symbol symbol : m\_memberMap.values()) {

size += symbol.getType().getSize();

}

return size;

}

case Union: {

int size = 0;

for (Symbol symbol : m\_memberMap.values()) {

size = Math.max(size, symbol.getType().getSize());

}

return size;

}

default:

return m\_sizeMap.get(m\_sort);

}

}

### Completeness

It is possible to define an array without stating its size, in which case the array is given the size zero. In that case, the array size must be determined by the size of its initialization list. However, if the array definition lacks an initialization list, the array keeps the size zero and is considered incomplete. In the same way, it is possible to define only the name tag of a struct or union, with its member map to be defined later. In that case, the member map is given the value null, which is keep if the member map is not defined, and the struct or union is consider incomplete. Variables can only have complete types, and the pointer type, array type or function return type must be also complete.

Type.java

public boolean isComplete() {

switch (m\_sort) {

case Array:

return (m\_arraySize > 0);

case Struct:

case Union:

return (m\_memberMap != null);

default:

return true;

}

}

### Constant and Volatile Types

Simply put, a type is constant if its field m\_constant is true. However, a struct or union is regarded as constant if it is constant (m\_constant is true) or is any of its members if (recursively) constant.

The idea of the volatile qualifier is to prevent optimization, and since this book is focused on optimization techniques we have no real use for the volatile qualifier. However, for the sake of completeness we include the m\_volatile field in the Type class.

Type.java

private boolean m\_constant;

private boolean m\_volatile;

public boolean isConstant() {

if (m\_constant) {

return true;

}

else if (isStructOrUnion() && (m\_memberMap != null)) {

for (Symbol symbol: m\_memberMap.values()) {

if (symbol.getType().isConstant()) {

return true;

}

}

}

return false;

}

public void setConstant(boolean constant) {

m\_constant = constant;

}

public boolean isVolatile() {

return m\_volatile;

}

public void setVolatile(boolean isVolatile) {

m\_volatile = isVolatile;

}

### Equality

Two pointers are considered to be equal if the types they point at are equal, two arrays are equal if (1) their types are equal and (2) they have the same size or both are incomplete (their sizes are zero). Two structs or unions are equals if they both are incomplete (their member maps are null) or if their member maps are equal. Note that they must not only have the same members, the members must also appear in the same order. Two functions are equal if their return types are equal, they have the same style (old or new) and their name lists or type lists, respectively, are equal.

Type.java

public boolean equals(Object object) {

if (object instanceof Type) {

Type type = (Type) object;

if (m\_sort == type.m\_sort) {

switch (m\_sort) {

case Pointer:

return m\_pointerType.equals(type.m\_pointerType);

case Array:

return ((m\_arraySize == 0) || (type.m\_arraySize == 0) ||

(m\_arraySize == type.m\_arraySize)) &&

m\_arrayType.equals(type.m\_arrayType);

case Struct:

case Union:

return ((m\_memberMap == null) && (type.m\_memberMap == null)) ||

((m\_memberMap != null) && (type.m\_memberMap != null) &&

m\_memberMap.equals(type.m\_memberMap));

case Function:

return m\_returnType.equals(type.m\_returnType) &&

(((m\_typeList == null) && (type.m\_typeList == null)) ||

((m\_typeList != null) && (type.m\_typeList != null) &&

m\_typeList.equals(type.m\_typeList)));

default:

return true;

}

}

}

return false;

}

### Test Methods

Finally, the Type class holds a series of simple test methods.

Type.java

public boolean isVoid() {

return (m\_sort == Sort.Void);

}

public boolean isChar() {

return (m\_sort == Sort.Signed\_Char) || (m\_sort == Sort.Unsigned\_Char);

}

public boolean isShort() {

return (m\_sort == Sort.Signed\_Short\_Int) ||

(m\_sort == Sort.Unsigned\_Short\_Int);

}

public boolean isInteger() {

return (m\_sort == Sort.Signed\_Int) || (m\_sort == Sort.Unsigned\_Int);

}

public boolean isIntegral() {

return isSigned() || isUnsigned();

}

public boolean isSigned() {

switch (m\_sort) {

case Signed\_Char:

case Signed\_Short\_Int:

case Signed\_Int:

case Signed\_Long\_Int:

return true;

default:

return false;

}

}

public boolean isUnsigned() {

switch (m\_sort) {

case Unsigned\_Char:

case Unsigned\_Short\_Int:

case Unsigned\_Int:

case Unsigned\_Long\_Int:

return true;

default:

return false;

}

}

public boolean isFloating() {

switch (m\_sort) {

case Float:

case Double:

case Long\_Double:

return true;

default:

return false;

}

}

public boolean isLogical() {

return (m\_sort == Sort.Logical);

}

public boolean isPointer() {

return (m\_sort == Sort.Pointer);

}

public boolean isArray() {

return (m\_sort == Sort.Array);

}

public boolean isPointerOrArray() {

return isPointer() || isArray();

}

public boolean isString() {

return (m\_sort == Sort.String);

}

public boolean isFunction() {

return (m\_sort == Sort.Function);

}

public boolean isFunctionPointer() {

return (m\_sort == Sort.Pointer) && m\_pointerType.isFunction();

}

public boolean isFunctionOrArray() {

return isFunction() || isArray();

}

public boolean isFunctionArrayOrString() {

return isFunctionOrArray() || isString();

}

public boolean isArithmetic() {

return isIntegral() || isFloating();

}

public boolean isIntegralOrPointer() {

return isIntegral() || isPointer();

}

public boolean isArithmeticOrPointer() {

return isArithmetic() || isPointer();

}

public boolean isStruct() {

return (m\_sort == Sort.Struct);

}

public boolean isUnion() {

return (m\_sort == Sort.Union);

}

public boolean isStructOrUnion() {

return isStruct() || isUnion();

}

public boolean isEnumeration() {

return (m\_sort == Sort.Enumeration);

}

public String toString() {

return m\_sort.name().replace("\_", " ").toLowerCase();

}

}

## Type Casting

Type casting occurs on several occasions in C. There are two types of cast: explicit (manual) conversation where type cast is stated and implicit (automatic) where it the type cast is omitted:

double x = 3.1;

int y = x, // implicit cast

z = (int) x; // explicit cast

Even though implicit casts are allowed in C, a warning is often raised when the cast causes the value to be truncated in some form, like in the code above where the decimal part of the double value is lost when casted into an integer value.

Naturally, if the types of the source and target trees are the same, we just return the source tree.

TypeCast.Java

package c\_compiler;

public class TypeCast {

public static SyntaxTree explicitCast(SyntaxTree sourceTree,

Type targetType) {

sourceTree = SyntaxTreeOptimizer.optimizeExpression(sourceTree);

Type sourceType = sourceTree.getType();

if (sourceType.equals(targetType)) {

return sourceTree;

}

In case of two pointers, we create and return new pointer in the explicit case. The reason for this is the resulting type shall not be a left-value. We want to prevent that an explicit cast is assigned a value. An expression such as “((char\*) p) = &q;” is not allowed.

else if (sourceType.isIntegralOrPointer() &&

targetType.isIntegralOrPointer()) {

return SyntaxTreeGenerator.generateUnaryTree

(MiddleOperator.IntegralToIntegral, targetType, sourceTree);

}

In case of a cast from an integral type of one byte (pointers are always two bytes) to a floating type, it becomes a little bit more complicated since there is no assembly converting operator. Instead, we need to first convert the small integral value (one byte) to a larger integral value (two bytes), which we in turn convert to a floating value.

else if (sourceType.isIntegralOrPointer() && targetType.isFloating()) {

if (sourceType.getSize() == 1) {

Type tempType = sourceType.isSigned() ? Type.SignedIntegerType

: Type.UnsignedIntegerType;

SyntaxTree tempTree = SyntaxTreeGenerator.generateUnaryTree

(MiddleOperator.IntegralToIntegral, tempType, sourceTree);

return SyntaxTreeGenerator.generateUnaryTree

(MiddleOperator.IntegralToFloating, targetType, tempTree);

}

else {

return SyntaxTreeGenerator.generateUnaryTree

(MiddleOperator.IntegralToFloating, targetType, sourceTree);

}

}

The same goes for the other way around, when converting a floating value to a small (one byte) integral value, we first need to convert the floating value to an integral value of two bytes, which we in turn convert to an integral value of one byte.

else if (sourceType.isFloating() && targetType.isIntegralOrPointer()) {

if (targetType.getSize() == 1) {

Type tempType = sourceType.isSigned() ? Type.SignedIntegerType

: Type.UnsignedIntegerType;

SyntaxTree tempTree = SyntaxTreeGenerator.generateUnaryTree

(MiddleOperator.FloatingToIntegral, tempType, sourceTree);

return SyntaxTreeGenerator.generateUnaryTree

(MiddleOperator.IntegralToIntegral, targetType, tempTree);

}

else {

return SyntaxTreeGenerator.generateUnaryTree

(MiddleOperator.FloatingToIntegral, targetType, sourceTree);

}

}

The remaining cases are trivial, we just generate the appropriate converting operation.

else if (sourceType.isFloating() && targetType.isFloating()) {

return SyntaxTreeGenerator.generateUnaryTree

(MiddleOperator.FloatingToFloating, targetType, sourceTree);

}

else if (sourceType.isLogical() && targetType.isIntegralOrPointer()) {

return SyntaxTreeGenerator.generateUnaryTree

(MiddleOperator.LogicalToIntegral, targetType, sourceTree);

}

else if (sourceType.isLogical() && targetType.isFloating()) {

return SyntaxTreeGenerator.generateUnaryTree

(MiddleOperator.LogicalToFloating, targetType, sourceTree);

}

When converting a string to a character pointer we just return the source tree.

else if (sourceType.isString() && targetType.isPointer() &&

targetType.getPointerType().isChar()) {

return sourceTree;

}

When converting between pointers and arrays we check that they have the same type and (in the arrays case) the same size.

else if (sourceType.isPointerOrArray() && targetType.isPointerOrArray()) {

Assert.error(sourceType.getPointerOrArrayType().equals

(targetType.getPointerOrArrayType()),

sourceType, "cannot be converted to " + targetType);

if (sourceType.isArray() && targetType.isArray() &&

(targetType.getArraySize() != 0)) {

Assert.error(targetType.getArraySize() == targetType.getArraySize(),

sourceType, "different array sizes");

}

return sourceTree;

}

It is possible to explicitly convert a value to void. In that case, we just return null.

else if (targetType.isVoid()) {

return null;

}

If none of the cases above applies, we raise an error.

else {

Assert.error("Cannot convert from " + sourceType + " to " + targetType);

return null;

}

The implicitCast method does basically the same thing as explicitCast. The difference is that is raises warnings in case of converting from a larger type to a smaller type. Another difference is that we do not need to create an new type if it has the same size as the source type.

public static SyntaxTree implicitCast(SyntaxTree sourceTree,

Type targetType) {

sourceTree = SyntaxTreeOptimizer.optimizeExpression(sourceTree);

If the source tree has a value with the same size as the target type, we just return the source tree.

if ((sourceTree.getSymbol() != null) && sourceTree.getSymbol().hasValue()

&& sourceTree.getType().getSize() == targetType.getSize()) {

return sourceTree;

}

Type sourceType = sourceTree.getType();

if (sourceType.equals(targetType)) {

return sourceTree;

}

else if (sourceType.isPointer() && targetType.isPointer()) {

return sourceTree;

}

If case of convertion froom a larger integral type to a smaller type, we reaise a warning.

else if (sourceType.isIntegralOrPointer() &&

targetType.isIntegralOrPointer()) {

if (sourceType.getSize() != targetType.getSize()) {

Assert.warning((sourceType.getSize() <= targetType.getSize()), null,

"implicit conversion from " + sourceType + " to " + targetType);

return SyntaxTreeGenerator.generateUnaryTree

(MiddleOperator.IntegralToIntegral, targetType, sourceTree);

}

In case of convertion between tow different integral types of the same size we just return the source tree. Unlike the explicit case above, we do not need to generate an convertion operator.

else {

return sourceTree;

}

}

else if (sourceType.isIntegralOrPointer() && targetType.isFloating()) {

if (sourceType.getSize() == 1) {

Type tempType = sourceType.isSigned() ? Type.SignedIntegerType

: Type.UnsignedIntegerType;

SyntaxTree tempTree = SyntaxTreeGenerator.generateUnaryTree

(MiddleOperator.IntegralToIntegral, tempType, sourceTree);

return SyntaxTreeGenerator.generateUnaryTree

(MiddleOperator.IntegralToFloating, targetType, tempTree);

}

else {

return SyntaxTreeGenerator.generateUnaryTree

(MiddleOperator.IntegralToFloating, targetType, sourceTree);

}

}

An implicit convertion from a floating type to an integral or pointer type always raises a warning.

else if (sourceType.isFloating() && targetType.isIntegralOrPointer()) {

Assert.warning(null, "implicit conversion from " + sourceType+

" to " + targetType);

if (targetType.getSize() == 1) {

Type tempType = sourceType.isSigned() ? Type.SignedIntegerType

: Type.UnsignedIntegerType;

SyntaxTree tempTree = SyntaxTreeGenerator.generateUnaryTree

(MiddleOperator.FloatingToIntegral, tempType, sourceTree);

return SyntaxTreeGenerator.generateUnaryTree

(MiddleOperator.IntegralToIntegral, targetType, tempTree);

}

else {

return SyntaxTreeGenerator.generateUnaryTree

(MiddleOperator.FloatingToIntegral, targetType, sourceTree);

}

}

else if (sourceType.isFloating() && targetType.isFloating()) {

if (sourceType.getSize() != targetType.getSize()) {

Assert.warning((sourceType.getSize() <= targetType.getSize()), null,

"implicit conversion from " + sourceType + " to " + targetType);

return SyntaxTreeGenerator.generateUnaryTree

(MiddleOperator.FloatingToFloating, targetType, sourceTree);

}

else {

return sourceTree;

}

}

else if (sourceType.isLogical() && targetType.isIntegralOrPointer()) {

return SyntaxTreeGenerator.generateUnaryTree

(MiddleOperator.LogicalToIntegral, targetType, sourceTree);

}

else if (sourceType.isLogical() && targetType.isFloating()) {

return SyntaxTreeGenerator.generateUnaryTree

(MiddleOperator.LogicalToFloating, targetType, sourceTree);

}

else if (sourceType.isString() && targetType.isPointer() &&

targetType.getPointerType().isChar()) {

return sourceTree;

}

else if (sourceType.isPointerOrArray() && targetType.isPointerOrArray()) {

Assert.error(sourceType.getPointerOrArrayType().equals

(targetType.getPointerOrArrayType()),

sourceType, "cannot be converted to " + targetType);

if (sourceType.isArray() && targetType.isArray() &&

(targetType.getArraySize() != 0)) {

Assert.error(targetType.getArraySize() == targetType.getArraySize(),

sourceType, "different array sizes");

}

return sourceTree;

}

else {

Assert.error("Cannot convert from " + sourceType + " to " + targetType);

return null;

}

}

If the from type is string and the target type is integral or pointer, we just return the original symbol. The address of the first character in the string will be interpreted as an integer value in the object code generation (Section 16.3).

If the types are equal or if they both are integral of the same size, the original symbol is returned.

If both types are integral or pointer (of different sizes), we raise a warning in case of implicit cast if the target type is smaller than the source type. Then we create a register temporary symbol and generate a cast middle code instruction.

In case of cast to integral or pointer to floating type there a little bit more complicated due to the object code architecture. There is no operation for casting between an integer value of one byte and a floating value. In that case, we must generate two casting instructions.

When casting between floating values, we just raise a warning in case of implicit cast from a larger to a smaller value. Then we create a register temporary symbol and generate a case middle code instruction.

## Type Promotion

Type promotion is the process of converting the “smaller” type to the “larger” type in a binary expression. For instance, in the expression below, the integer i shall be converted to a double value:

int i = 3;

double x = 3.14, y;

y = x + i;

Below is a table holding the relations of the types. Note that float is considered to be larger than long, event thought they have the same size. The same goes for short and char.

|  |  |  |
| --- | --- | --- |
|  |  | Size |
| Largest type | long double | 8 |
|  | double | 8 |
|  | float | 4 |
|  | signed long int  unsigned long int | 4 |
|  | signed int  unsigned int | 2 |
|  | signed short int  unsigned short int | 1 |
| Smallest type | signed char  unsigned char | 1 |

In accordance with the Ansi C standard the signed and unsigned values have the same size, and a character is always one byte[[7]](#footnote-7). However, the standard does not state whether a signed value shall be converted to an unsigned value or the other way around in the expression below. In book we always convert the unsigned value to the matching signed type (if the hold the same size), with a warning in case of implicit conversion.

signed int i = -3;

unsigned int u = 6;

i + u;

The maxType method returns the largest type of the left and right type.

public static Type maxType(Type leftType, Type rightType) {

if (leftType.isString() && rightType.isString()) {

Type charType = Type.createArithmeticType(Sort.Signed\_Char);

charType.setConstant(true);

Type pointerType = Type.createPointerType(charType);

pointerType.setConstant(true);

return pointerType;

}

else if (leftType.isString() &&

rightType.isPointer() && rightType.getPointerType().isChar()) {

Type pointerType = Type.createPointerType(rightType.getPointerType());

pointerType.setConstant(true);

return pointerType;

}

else if (leftType.isPointer() && leftType.getPointerType().isChar() &&

rightType.isString()) {

Type pointerType = Type.createPointerType(leftType.getPointerType());

pointerType.setConstant(true);

return pointerType;

}

else if ((leftType.isIntegralOrPointer()&&rightType.isIntegralOrPointer())

|| (leftType.isFloating() && rightType.isFloating())) {

return (leftType.getSize() >= rightType.getSize()) ?leftType :rightType;

}

else if (leftType.isIntegral() && rightType.isIntegral()) {

int leftSize = leftType.getSize(), rightSize = rightType.getSize();

if (leftSize == rightSize) {

if (leftType.isUnsigned() && rightType.isUnsigned()) {

return leftType;

}

else {

Assert.warning((leftType.isSigned() && rightType.isSigned()) ||

(leftType.isUnsigned() && rightType.isUnsigned()),

null, "mixing signed and unsigned types");

return leftType.isSigned() ? leftType : rightType;

}

}

else {

return (leftSize >= rightSize) ? leftType : rightType;

}

}

else if (leftType.isIntegral() && rightType.isFloating()) {

return rightType;

}

else if (leftType.isFloating() && rightType.isIntegral()) {

return leftType;

}

else if (leftType.isPointer() && leftType.getPointerType().isVoid() &&

rightType.isPointer()) {

return leftType;

}

else if (leftType.isPointer() &&

rightType.isPointer() && rightType.getPointerType().isVoid()) {

return rightType;

}

else if (leftType.isPointer() && rightType.isArray()) {

return leftType;

}

else if (leftType.isArray() && rightType.isPointer()) {

return rightType;

}

return null;

}

}

## The Value Conversion Class

When optimizing the syntax trees, values may need to be computed in compile-time. The valueCast, unaryCast, and binaryCast methods casts the values to a BigDecimal, BigInteger, or Boolean object, depending of the type or operator.

ValueConversion.java

package c\_compiler;

import java.math.\*;

public class ValueConversion {

public static Object valueCast(Symbol fromSymbol, Type toType) {

if (fromSymbol.hasValue()) {

Object fromValue = fromSymbol.getValue();

if (toType.isFloating()) {

return BigDecimal.valueOf(((Number) fromValue).doubleValue());

}

else if (toType.isIntegral()) {

return BigInteger.valueOf(((Number) fromValue).longValue());

}

else {

return fromValue;

}

}

return null;

}

public static Object unaryCast(MiddleOperator operator, Object value) {

if (operator == MiddleOperator.StringToPointer) {

return value;

}

if (value == null) {

return null;

}

if (value instanceof BigInteger) {

BigInteger bigInteger = (BigInteger) value;

switch (operator) {

case UnaryAdd:

case IntegralToIntegral:

return value;

case UnarySubtract:

return bigInteger.negate();

case BitwiseNot:

return bigInteger.not();

case IntegralToFloating:

return (new BigDecimal(bigInteger));

case ArithmeticToLogical:

return !bigInteger.equals(BigInteger.ZERO);

}

}

else if (value instanceof BigDecimal) {

BigDecimal bigDecimal = (BigDecimal) value;

switch (operator) {

case UnaryAdd:

case FloatingToFloating:

return value;

case UnarySubtract:

return bigDecimal.negate();

case FloatingToIntegral:

return bigDecimal.toBigInteger();

case ArithmeticToLogical:

return !bigDecimal.equals(BigDecimal.ZERO);

}

}

else if (value instanceof Boolean) {

switch (operator) {

case LogicalNot:

return value.equals(Boolean.TRUE) ? Boolean.FALSE : Boolean.TRUE;

}

};

return null;

}

public static Object binaryCast(MiddleOperator operator, Object leftValue,

Object rightValue) {

if ((leftValue == null) || (rightValue == null)) {

return null;

}

if (leftValue instanceof BigInteger) {

BigInteger leftInteger = (BigInteger) leftValue,

rightInteger = (BigInteger) rightValue;

switch (operator) {

case BinaryAdd:

return leftInteger.add(rightInteger);

case BinarySubtract:

return leftInteger.subtract(rightInteger);

case Multiply:

return leftInteger.multiply(rightInteger);

case Divide:

Assert.error(!rightInteger.equals(BigInteger.ZERO),

"0", "division by zero in constant expression");

return leftInteger.divide(rightInteger);

case Modulo:

Assert.error(!rightInteger.equals(BigInteger.ZERO),

"0", "division by zero in constant expression");

return leftInteger.mod(rightInteger);

case BitwiseAnd:

return leftInteger.and(rightInteger);

case BitwiseIOr:

return leftInteger.or(rightInteger);

case BitwiseXOr:

return leftInteger.xor(rightInteger);

case LeftShift:

return leftInteger.shiftLeft(rightInteger.intValue());

case RightShift:

return leftInteger.shiftRight(rightInteger.intValue());

case Equal:

return (leftInteger.compareTo(rightInteger) == 0);

case NotEqual:

return (leftInteger.compareTo(rightInteger) != 0);

case LessThan:

return (leftInteger.compareTo(rightInteger) < 0);

case LessThanEqual:

return (leftInteger.compareTo(rightInteger) <= 0);

case GreaterThan:

return (leftInteger.compareTo(rightInteger) > 0);

case GreaterThanEqual:

return (leftInteger.compareTo(rightInteger) >= 0);

}

}

else if (leftValue instanceof BigDecimal) {

BigDecimal leftDecimal = (BigDecimal) leftValue,

rightDecimal = (BigDecimal) rightValue;

switch (operator) {

case BinaryAdd:

return leftDecimal.add(rightDecimal);

case BinarySubtract:

return leftDecimal.subtract(rightDecimal);

case Multiply:

return leftDecimal.multiply(rightDecimal);

case Divide:

Assert.error(!rightDecimal.equals(BigDecimal.ZERO),

"0", "division by zero in constant expression");

return leftDecimal.divide(rightDecimal);

case Equal:

return (leftDecimal.compareTo(rightDecimal) == 0);

case NotEqual:

return (leftDecimal.compareTo(rightDecimal) != 0);

case LessThan:

return (leftDecimal.compareTo(rightDecimal) < 0);

case LessThanEqual:

return (leftDecimal.compareTo(rightDecimal) <= 0);

case GreaterThan:

return (leftDecimal.compareTo(rightDecimal) > 0);

case GreaterThanEqual:

return (leftDecimal.compareTo(rightDecimal) >= 0);

}

}

else if (leftValue instanceof Boolean) {

Boolean leftBoolean = (Boolean) leftValue,

rightBoolean = (Boolean) rightValue;

switch (operator) {

case LogicalAnd:

return (leftBoolean.booleanValue() && rightBoolean.booleanValue());

case LogicalOr:

return (leftBoolean.booleanValue() || rightBoolean.booleanValue());

}

}

return null;

}

}

# The Symbol Table

As the name implies, the symbol table is a table holding symbols. It also holds struct and union tags (but not enumeration tags). Moreover, there is one symbol table for the global space, one table for each function, and one table for each block in a function. In this way, the symbol tables form a hierarchy where each table has a reference to its parent table (except the table for the global space, which table parent reference is null), and a set of references to its child tables.

## The Symbol Class

The Symbol class describes a variable, a value, a function, or a type specified with typedef. It holds the following features:

* Name. A symbol actually has two names: the regular name it has been defined with and a unique name, which is the same for symbols with external linkage and prefixed with the number of its block is has been defined in for symbols without external linkage.
* Status. A symbol may be defined, defined with value, parameter, temporary, solid temporary, or value. A solid temporary symbol is a temporary symbol which value shall be stored between base blocks (see Section 14.2.1).
* Storage. A symbol may be auto, register, static, extern or typedef. If the storage specifier is omitted a global variable is regarded as static and a local variable is regarded as auto. In this book, the only difference between an auto or register symbol is that the address operator cannot be applied to a register symbol.
* Type. Each symbol has a type as described in Section 7.1.
* Value. A symbol may have a value of integral,
* External. A symbol may have external linkage; that is, it is visible from other source code files.
* Parent Symbol. If the symbol is a member of a struct or union, the parent symbol is that struct or union.
* Child Set.
* Address Symbol. In case of the address, arrow, or index operator the address symbol is the dereferred symbol. In case of the arrow operator or constant index we also store the offset of the address symbol.
* Address Offset.
* Offset. Each auto or register symbol has an offset on the activation record. Each struct member has also an offset relative the beginning of the struct (the first member has offset zero), all union members have offset zero.
* When the final object code is generated, we mark each auto or register symbol as used. In the end, all non-used symbols are being removed.

The name is somewhat misleading since we also store functions and types, defined by typedef, in the table. In some compiler technique text books the table is called the symbol table. However, I think that term is too wide for our purposes.

### Symbol Storage

A symbol can hold the storage Typedef, Static, Extern, Auto, and Register. If the storage specifier is omitted, a global variable is regarded as static, and a local variable is regarded as auto. The Register storage has been included for the sake of completeness, but in this book is has no effect. On every occasion, a symbol with the Auto or Register storage are treated equally. Moreover, a symbol has a name, type, and potential value.

Storage.java

package c\_compiler;

public enum Storage {NoStorage, Typedef, Extern,

Static, Auto, Register};

Symbol.java

package c\_compiler;

import java.io.\*;

import java.math.\*;

import java.util.\*;

public class Symbol {

private static enum Status {Variable, Parameter, Temporary, Solid};

private Status m\_status;

protected String m\_name, m\_uniqueName;

private Storage m\_storage;

protected Type m\_type;

protected Object m\_value;

private int m\_offset, m\_entryPoint;

private Symbol m\_addressSymbol;

private int m\_addressOffset;

private Symbol m\_parentSymbol;

private Set<Symbol> m\_childSymbolSet = new ListSet<>();

private boolean m\_referred;

private List<Byte> m\_byteList;

private List<String> m\_textList;

private Map<Integer,String> m\_accessMap;

private Map<Integer,String> m\_callMap;

private Set<Integer> m\_returnSet;

private Map<Integer,Integer> m\_byteToTextMap;

public Symbol() {

// Empty.

}

private Symbol(Status status, String name, boolean external, Storage storage,

Type type, Object value, List<Byte> byteList,

Map<Integer,String> accessMap, Map<Integer,String> callMap,

Set<Integer> returnSet, Map<Integer,Integer> byteToTextMap,

List<String> textList) {

m\_status = status;

m\_name = name;

m\_storage = storage;

m\_type = type;

m\_value = value;

m\_byteList = byteList;

m\_accessMap = accessMap;

m\_callMap = callMap;

m\_returnSet = returnSet;

m\_byteToTextMap = byteToTextMap;

m\_textList = textList;

if (m\_name != null) {

if (external || m\_name.endsWith(Main.NumberId)) {

m\_uniqueName = m\_name;

}

else {

m\_uniqueName = m\_name + Main.FileMarker +

String.valueOf(Main.CurrentTable.getTableId());

}

}

}

public static Symbol createOffsetSymbol(Symbol symbol) {

Symbol offsetSymbol =

new Symbol(Status.Variable, symbol.m\_name, false, Storage.Auto,

symbol.m\_type, null, null, null, null, null, null, null);

offsetSymbol.m\_offset = symbol.m\_offset;

return offsetSymbol;

}

public static Symbol createVariableSymbol(String name, boolean external,

Storage storage, Type type,Object value){

if (!type.isFunction() && (storage == Storage.Static)) {

List<Byte> byteList = new LinkedList<>();

Map<Integer,String> accessMap = new ListMap<>();

List<String> textList = new LinkedList<>();

if (value != null) {

GenerateInitializerStatic.generateByteTextList

(type, value, byteList, accessMap, textList);

}

else {

GenerateInitializerStatic.generateZeroByteTextList

(type.getSize(), byteList, textList);

}

textList.add(0, name + ":");

Symbol symbol =

new Symbol(Status.Variable, name, external, storage, type, value,

byteList, accessMap, null, null, null, textList);

Main.GlobalSet.add(symbol);

return symbol;

}

else {

return (new Symbol(Status.Variable, name, external, storage, type, value,

null, null, null, null, null, null));

}

}

public static Symbol createValueSymbol(Type type, Object value) {

Assert.error(value != null);

List<Byte> byteList = new LinkedList<>();

Map<Integer,String> accessMap = new ListMap<>();

List<String> textList = new LinkedList<>();

GenerateInitializerStatic.generateByteTextList

(type, value, byteList, accessMap, textList);

String name = GenerateInitializerStatic.generateValueName(type, value);

Symbol symbol =

new Symbol(Status.Variable, name, true, Storage.Static, type, value,

byteList, accessMap, null, null, null, textList);

textList.add(0, "\n" + symbol.getUniqueName() + ":");

Main.GlobalSet.add(symbol);

return symbol;

}

public static Symbol createInitSymbol(String name, boolean external,

Storage storage, Type type, List<Byte> byteList,

Map<Integer,String> accessMap, List<String> textList){

Symbol symbol =

new Symbol(Status.Variable, name, external, storage, type, null,

byteList, accessMap, null, null, null, textList);

textList.add(0, "\n" + symbol.getUniqueName() + ":");

Main.GlobalSet.add(symbol);

return symbol;

}

public static Symbol createStaticSymbol(String name, List<Byte> byteList,

Map<Integer,String> accessMap, List<String> textList){

Symbol symbol =

new Symbol(Status.Variable, name, true, Storage.Static, null, null,

byteList, accessMap, null, null, null, textList);

textList.add(0, "\n" + symbol.getUniqueName() + ":");

Main.GlobalSet.add(symbol);

return symbol;

}

public static Symbol createStaticSymbol(String name, List<Byte> byteList,

Map<Integer,String> accessMap,Map<Integer,String> callMap,

Set<Integer> returnSet, Map<Integer,Integer> byteToTextMap,

List<String> textList) {

Symbol symbol =

new Symbol(Status.Variable, name, true, Storage.Static, null, null,

byteList, accessMap,callMap,returnSet,byteToTextMap,textList);

textList.add(0, "\n" + symbol.getUniqueName() + ":");

Main.GlobalSet.add(symbol);

return symbol;

}

public static Symbol createLogicalSymbol(Set<Integer> trueSet,

Set<Integer> falseSet) {

Type logicalType = Type.createLogicalType(trueSet, falseSet);

return (new Symbol(Status.Temporary, null, false, Storage.Auto,

logicalType, null, null, null, null, null, null, null));

}

public static Symbol createSolidSymbol(Type type) {

String name = "$solid" + (Main.SolidCount++);

return (new Symbol(Status.Solid, name, false, Storage.Auto, type,

null, null, null, null, null, null, null));

}

public static Symbol createTemporarySymbol(Type type) {

String name = "$temp" + (Main.TemporaryCount++);

return (new Symbol(Status.Temporary, name, false, Storage.Auto,

type, null, null, null, null, null, null, null));

}

public static Symbol createMemorySymbol(Symbol parentSymbol, Type type,

Symbol memberSymbol) {

String name = parentSymbol.getName() + "." + memberSymbol.getName() +

Main.SeparatorId + memberSymbol.getOffset();

Symbol resultSymbol =

new Symbol(parentSymbol.m\_status, name, false, parentSymbol.getStorage(),

type, null, null, null, null, null, null, null);

resultSymbol.m\_uniqueName = parentSymbol.m\_uniqueName;

resultSymbol.setParentSymbol(parentSymbol);

resultSymbol.setOffset(parentSymbol.getOffset() +memberSymbol.getOffset());

return resultSymbol;

}

public static Symbol createParameterSymbol(String name, Storage storage,

Type type) {

return (new Symbol(Status.Parameter, name, false, storage, type,

null, null, null, null, null, null, null));

}

public Register baseRegister() {

switch (m\_storage) {

case Auto:

case Register:

return !Main.FuncSymbol.getType().isEllipse() ||

(m\_status == Status.Parameter) ? ObjectCode.EllipseRegister

: ObjectCode.FrameRegister;

default:

return null;

}

}

public String getName() {

return m\_name;

}

public void setName(String name) {

m\_name = name;

}

public String getUniqueName() {

return m\_uniqueName;

}

public Storage getStorage() {

return m\_storage;

}

public void setStorage(Storage storage) {

if ((m\_storage != Storage.Static) && (storage == Storage.Static)) {

m\_byteList = new LinkedList<>();

m\_accessMap = new ListMap<>();

m\_textList = new LinkedList<>();

GenerateInitializerStatic.generateByteTextList

(m\_type, m\_value, m\_byteList, m\_accessMap,m\_textList);

m\_textList.add(m\_uniqueName + ":");

Main.GlobalSet.add(this);

}

m\_storage = storage;

}

public Type getType() {

return m\_type;

}

public void setType(Type type) {

m\_type = type;

}

public int getOffset() {

return m\_offset;

}

public void setOffset(int offset) {

m\_offset = offset;

}

public int getEntryPoint() {

return m\_entryPoint;

}

public void setEntryPoint(int entryPoint) {

m\_entryPoint = entryPoint;

}

public boolean isAutoOrRegister() {

return (m\_storage == Storage.Auto) ||

(m\_storage == Storage.Register);

}

public boolean isStatic() {

return (m\_storage == Storage.Static);

}

public boolean isExtern() {

return (m\_storage == Storage.Extern);

}

public boolean isStaticOrExtern() {

return isStatic() || isExtern();

}

public boolean isTypedef() {

return (m\_storage == Storage.Typedef);

}

public boolean isParameter() {

return (m\_status == Status.Parameter);

}

public boolean isTemporary() {

Symbol symbol = this;

while (symbol != null) {

if (symbol.m\_status != Status.Temporary) {

return false;

}

symbol = symbol.getParentSymbol();

}

return true;

}

public boolean isSolid() {

return (m\_status == Status.Solid);

}

public boolean hasValue() {

return (m\_value != null);

}

public Object getValue() {

return m\_value;

}

public void setValue(Object value) {

m\_value = value;

}

public BigInteger getSignedValue() {

if (m\_type.isUnsigned()) {

BigInteger intValue = (BigInteger) m\_value;

if (intValue.compareTo(BigInteger.ZERO) < 0) {

switch (m\_type.getSize()) {

case Type.CharSize:

return intValue.subtract(new BigInteger("128"));

case Type.IntegerSize:

return intValue.subtract(new BigInteger("32768"));

case Type.LongSize:

return (new BigInteger("2147483648")).add(intValue);

}

}

}

return ((BigInteger) m\_value);

}

public boolean hasAddressSymbol() {

return (m\_addressSymbol != null);

}

public Symbol getAddressSymbol() {

return m\_addressSymbol;

}

public void setAddressSymbol(Symbol addressSymbol) {

m\_addressSymbol = addressSymbol;

}

public int getAddressOffset() {

return m\_addressOffset;

}

public void setAddressOffset(int addressOffset) {

m\_addressOffset = addressOffset;

}

public Symbol getParentSymbol() {

return m\_parentSymbol;

}

public void setParentSymbol(Symbol parentSymbol) {

m\_parentSymbol = parentSymbol;

}

public Set<Symbol> getChildSymbolSet() {

return m\_childSymbolSet;

}

public void addChildSymbol(Symbol childSymbol) {

m\_childSymbolSet.add(childSymbol);

}

public boolean isReferred() {

return m\_referred;

}

public void setReferred(boolean referred) {

m\_referred = referred;

}

public List<Byte> getByteList() {

return m\_byteList;

}

public void setByteList(List<Byte> byteList) {

m\_byteList = byteList;

}

public Map<Integer,String> getAccessMap() {

return m\_accessMap;

}

public void setAccessMap(Map<Integer,String> accessMap) {

m\_accessMap = accessMap;

}

public Map<Integer,String> getCallMap() {

return m\_callMap;

}

public void setCallMap(Map<Integer,String> callMap) {

m\_callMap = callMap;

}

public Set<Integer> getReturnSet() {

return m\_returnSet;

}

public void setReturnSet(Set<Integer> returnSet) {

m\_returnSet = returnSet;

}

public Map<Integer,Integer> getByteToTextMap() {

return m\_byteToTextMap;

}

public void setByteToTextMap(Map<Integer,Integer> byteToTextMap) {

m\_byteToTextMap = byteToTextMap;

}

public List<String> getTextList() {

return m\_textList;

}

public void setTextList(List<String> textList) {

m\_textList = textList;

}

public boolean isLeftValue() {

switch (m\_type.getSort()) {

case Array:

case Function:

return false;

default:

return !m\_type.isConstant();

}

}

public String toString() {

return m\_storage.name() + " " + m\_type.toString() + " "+m\_name.toString();

}

public void save(DataOutputStream outStream) throws Exception {

writeString(outStream, m\_uniqueName);

outStream.writeInt(m\_entryPoint);

if (m\_byteList != null) {

outStream.writeInt(m\_byteList.size());

for (byte b : m\_byteList) {

outStream.writeByte(b);

}

}

else {

outStream.writeInt(0);

}

if (m\_accessMap != null) {

outStream.writeInt(m\_accessMap.size());

for (Map.Entry<Integer,String> entry : m\_accessMap.entrySet()) {

outStream.writeInt(entry.getKey());

writeString(outStream, entry.getValue());

}

}

else {

outStream.writeInt(0);

}

if (m\_callMap != null) {

outStream.writeInt(m\_callMap.size());

for (Map.Entry<Integer,String> entry : m\_callMap.entrySet()) {

outStream.writeInt(entry.getKey());

writeString(outStream, entry.getValue());

}

}

else {

outStream.writeInt(0);

}

if (m\_returnSet != null) {

outStream.writeInt(m\_returnSet.size());

for (Integer address : m\_returnSet) {

outStream.writeInt(address);

}

}

else {

outStream.writeInt(0);

}

if (m\_byteToTextMap != null) {

outStream.writeInt(m\_byteToTextMap.size());

for (Map.Entry<Integer,Integer> entry : m\_byteToTextMap.entrySet()) {

outStream.writeInt(entry.getKey());

outStream.writeInt(entry.getValue());

}

}

else {

outStream.writeInt(0);

}

if (m\_textList != null) {

outStream.writeInt(m\_textList.size());

for (String text : m\_textList) {

writeString(outStream, text);

}

}

else {

outStream.writeInt(0);

}

}

public static void writeString(DataOutputStream outStream, String text)

throws Exception {

outStream.writeInt(text.length());

for (char ch : text.toCharArray()) {

outStream.writeChar(ch);

}

}

public void load(DataInputStream inStream) throws Exception {

m\_uniqueName = readString(inStream);

m\_entryPoint = inStream.readInt();

{ m\_byteList = new LinkedList<>();

int byteListSize = inStream.readInt();

for (int index = 0; index < byteListSize; ++index) {

byte b = inStream.readByte();

m\_byteList.add(b);

}

}

{ m\_accessMap = new ListMap<>();

int accessMapSize = inStream.readInt();

for (int index = 0; index < accessMapSize; ++index) {

int address = inStream.readInt();

String name = readString(inStream);

m\_accessMap.put(address, name);

}

}

{ m\_callMap = new ListMap<>();

int callMapSize = inStream.readInt();

for (int index = 0; index < callMapSize; ++index) {

int address = inStream.readInt();

String name = readString(inStream);

m\_callMap.put(address, name);

}

}

{ m\_returnSet = new ListSet<>();

int returnSetSize = inStream.readInt();

for (int index = 0; index < returnSetSize; ++index) {

int address = inStream.readInt();

m\_returnSet.add(address);

}

}

{ m\_byteToTextMap = new ListMap<>();

int byteToTextMapSize = inStream.readInt();

for (int index = 0; index < byteToTextMapSize; ++index) {

int address = inStream.readInt();

int textLine = inStream.readInt();

m\_byteToTextMap.put(address, textLine);

}

}

{ m\_textList = new LinkedList<>();

int textListSize = inStream.readInt();

for (int index = 0; index < textListSize; ++index) {

String text = readString(inStream);

m\_textList.add(text);

}

}

}

public static String readString(DataInputStream inStream) throws Exception{

StringBuilder buffer = new StringBuilder();

int size = inStream.readInt();

for (int index = 0; index < size; ++index) {

buffer.append(inStream.readChar());

}

return buffer.toString().replace(Main.FileMarker, Main.SeparatorId +

Main.FileCount + Main.SeparatorId);

}

}

## The Symbol Table Class

The symbol table keeps track of the values, types, function, and variables of the code, variables defined by the programmer and well as temporary variables introduced by the compiler. The term symbol table is misleading since there actually is a stack of tables, where is table has a reference to the parent table, the parent reference table for the global space table is null. The current table is stored in Main.CurrentTable (see Chapter 20). The symbol table has set of features:

* Table Identity. Each table is given a unique consecutive integer value.
* Scope. A table can the scopes Global, Function, Block, Struct, Union, or Enumeration.
* Parent Table. Each table has apparent table which Main.CurrentTable is set to when the table reaches its end-of-scope.
* Entry List. The entries of the symbol table is stored in the entry list. An entry can be a symbol or a block, represented by another entry list.
* Tag List. Parallel to the entry list, there is also a tag list, holding tags of structs or unions, but not enums.
* Size. The size of the symbol table is the sum of the sizes of all its entries and is of interest when dimensioning the size of a function’s activation record.

SymbolTable.java

package c\_compiler;

import java.util.\*;

public class SymbolTable {

public static final int ReturnAddressOffset = 0;

public static final int RegularFrameOffset = Type.PointerSize;

public static final int EllipseFrameOffset = 2 \* Type.PointerSize;

public static final int FunctionRegularInitSize = 3 \* Type.PointerSize;

public static final int UseResultOffset = 3 \* Type.PointerSize;

public static final int FunctionFloatingInitSize =

3 \* Type.PointerSize + Type.CharSize;

public static int TableId = 0;

private Scope m\_scope;

private int m\_tableId;

private SymbolTable m\_parentTable;

private int m\_offset, m\_functionSize;

private Map<String,Object> m\_entryMap = new ListMap<>();

private Map<String,Type> m\_tagMap = new ListMap<>();

public SymbolTable(SymbolTable parentTable, Scope scope) {

m\_tableId = TableId++;

m\_parentTable = parentTable;

if (m\_parentTable != null) {

m\_parentTable.m\_entryMap.put(Main.SeparatorId +"table" + m\_tableId,this);

}

switch (m\_scope = scope) {

case Global:

case Struct:

case Union:

m\_offset = 0;

break;

case Function:

m\_offset = Main.FuncSymbol.getType().getReturnType().isFloating() ?

FunctionFloatingInitSize : FunctionRegularInitSize;

break;

case Block:

m\_offset = m\_parentTable.m\_offset;

break;

}

}

// ------------------------------------------------------------------------

public int getTableId() {

return m\_tableId;

}

public Scope getScope() {

return m\_scope;

}

public SymbolTable getParentTable() {

return m\_parentTable;

}

public Map<String,Object> getEntryMap() {

return m\_entryMap;

}

public int getOffset() {

return m\_offset;

}

public void setFunctionOffset() {

m\_functionSize = calculateSize();

m\_offset = m\_functionSize;

}

public int getFunctionSize() {

return m\_functionSize;

}

// ------------------------------------------------------------------------

public void addSymbol(Symbol newSymbol) {

String name = newSymbol.getName();

if (name != null) {

Symbol oldSymbol = (Symbol) m\_entryMap.get(name);

if ((oldSymbol != null) && !name.endsWith(Main.NumberId)) {

if (newSymbol.isTemporary()) {

return;

}

else if (newSymbol.isExtern()) {

Assert.error(oldSymbol.getType().equals(newSymbol.getType()),

name, "different types in redeclaration");

return;

}

else if (oldSymbol.isExtern()) {

Assert.error(oldSymbol.getType().equals(newSymbol.getType()),

name, "different types in redeclaration");

m\_entryMap.remove(name);

}

else {

Assert.error(name, "already defined");

}

}

}

else {

newSymbol.setName(Main.SeparatorId + "parameter" + m\_entryMap.size());

}

if (newSymbol.isAutoOrRegister() && (m\_scope != Scope.Union)) {

newSymbol.setOffset(m\_offset);

m\_offset += newSymbol.getType().getSize();

}

m\_entryMap.put(newSymbol.getName(), newSymbol);

}

public Symbol lookupSymbol(String name) {

Symbol symbol = (Symbol) m\_entryMap.get(name);

if (symbol != null) {

return symbol;

}

else if (m\_parentTable != null) {

return m\_parentTable.lookupSymbol(name);

}

return null;

}

public void addTag(String name, Type type) {

m\_tagMap.put(name, type);

}

public Type lookupTag(String name) {

Type type = m\_tagMap.get(name);

if (type != null) {

return type;

}

else if (m\_parentTable != null) {

return m\_parentTable.lookupTag(name);

}

return null;

}

public void checkUnreferredSymbol() {

for (int index = (m\_entryMap.size() - 1); index >= 0; --index) {

Object entry = m\_entryMap.get(index);

if (entry instanceof Symbol) {

Symbol symbol = (Symbol) entry;

if (symbol.isAutoOrRegister() && !symbol.isParameter() &&

!symbol.isReferred()) {

decrementOffset(index + 1, symbol.getType().getSize());

m\_entryMap.remove(index);

Assert.warning(symbol.isSolid() || symbol.isTemporary(),

symbol, "unreferred variable");

}

}

}

m\_functionSize = calculateSize();

}

private void decrementOffset(int startIndex, int size) {

for (int index = startIndex; index < m\_entryMap.size(); ++index) {

Object entry = m\_entryMap.get(index);

if (entry instanceof Symbol) {

Symbol symbol = (Symbol) entry;

symbol.setOffset(symbol.getOffset() - size);

}

else {

SymbolTable table = (SymbolTable) entry;

table.decrementOffset(0, size);

}

}

m\_offset -= size;

}

private int calculateSize() {

int size = m\_offset;

for (Object entry : m\_entryMap.values()) {

if (entry instanceof SymbolTable) {

SymbolTable table = (SymbolTable) entry;

size = Math.max(size, table.calculateSize());

}

}

return size;

}

public String toString() {

return "size " + m\_functionSize + ", (0," + Type.PointerSize +

",return address), (" + Type.PointerSize + "," + Type.PointerSize +

",stack pointer), (" + (2 \* Type.PointerSize) + "," +

Type.PointerSize + "," + (Main.FuncSymbol.getType().isEllipse() ?

"ellipse pointer)" : "not used)");

}

}

### Add Symbols

When adding a symbol with a name we have to check a few things:

* If the symbol is extern it cannot be initialized (its value is not-null).
* We look up the entry list, and if we find another symbol with the same name, we check whether at least one of the symbols is extern. If it is, the symbols shall have equals types. If the new symbols is extern, weare finished and we just return. If the old symbol is extern, we remove it and stored the new symbol in the entry list. If not at least one of the symbols is extern, an error message is reported.

It is easier to add or erase a tag. If it is already added an error message is reported. When removing the tag, the tag associated to the name is returned, or null is it not stored in the map.

### Looking Up Symbols

The lookupList method searches the current table for a symbol with the given name, while lookupSymbol searches the table hierarchy recursively upwards by calling lookupList until it finds a symbol with the given name or the parent table is null, which will happen when the table of the global space has been searched.

A tag is recursively looked up in a similar manner.

### Generate Offset

Before the final object code is generated, we need to set the offset each function and block entry. If the entry is an auto or register symbol, we set its offset if is used. If it is not used, we report a warning if it has name, is not volatile, parameter, or temporary, and Generate.Safe is not true. The reason for Generate.Safe check is that the offset may be generated twice, with Generate.Safe true and false, and we do not want the same warning to be reported twice.

SymbolTable.java

### Table Text

Strictly speaking, generateSymbolTableText is not necessary.

# The Syntax Tree

SyntaxTree.java

package c\_compiler;

import java.util.\*;

public class SyntaxTree {

private MiddleOperator m\_operator;

private Type m\_type;

private Symbol m\_symbol;

private SyntaxTree m\_parent;

private List<SyntaxTree> m\_childList = new LinkedList<>();

private boolean m\_rightValue;

private boolean m\_sideEffect;

private SyntaxTree m\_returnTree;

private Object m\_info;

public SyntaxTree(MiddleOperator operator) {

m\_operator = operator;

}

public SyntaxTree(MiddleOperator operator, Type type) {

m\_operator = operator;

m\_type = type;

}

public SyntaxTree(MiddleOperator operator, Symbol symbol) {

m\_operator = operator;

m\_type = symbol.getType();

m\_symbol = symbol;

}

public SyntaxTree(MiddleOperator operator, Type type, Symbol symbol) {

m\_operator = operator;

m\_type = type;

m\_symbol = symbol;

}

public SyntaxTree(MiddleOperator operator, Object info) {

m\_operator = operator;

m\_info = info;

}

public SyntaxTree(MiddleOperator operator, Type type,

Symbol symbol, Object info) {

m\_operator = operator;

m\_type = type;

m\_symbol = symbol;

m\_info = info;

}

public MiddleOperator getOperator() {

return m\_operator;

}

public void setOperator(MiddleOperator operator) {

m\_operator = operator;

}

public Type getType() {

return m\_type;

}

public boolean hasSideEffect() {

return m\_sideEffect;

}

public Object getInfo() {

return m\_info;

}

public void setInfo(int info) {

m\_info = info;

}

public SyntaxTree getReturnTree() {

return m\_returnTree;

}

public void setReturnTree(SyntaxTree returnTree) {

m\_returnTree = returnTree;

}

public Symbol getSymbol() {

return m\_symbol;

}

public boolean isRightValue() {

return m\_rightValue;

}

public void setRightValue() {

m\_rightValue = true;

}

public SyntaxTree getChild(int index) {

return m\_childList.get(index);

}

public void setChild(int index, SyntaxTree childTree) {

if (childTree != null) {

childTree.m\_parent = this;

}

if (index < m\_childList.size()) {

m\_childList.set(index, childTree);

}

else {

m\_childList.add(childTree);

}

}

public void addChild(SyntaxTree childTree) {

if (childTree != null) {

childTree.m\_parent = this;

}

m\_childList.add(childTree);

}

public SyntaxTree getParent() {

return m\_parent;

}

public List<SyntaxTree> getChildList() {

return m\_childList;

}

}

# Syntax Tree Optimization

SyntaxTreeOptimizer.java

package c\_compiler;

import java.math.\*;

import java.util.\*;

public class SyntaxTreeOptimizer {

public static SyntaxTree optimizeStatement(SyntaxTree stat) {

if (stat == null) {

return null;

}

switch (stat.getOperator()) {

case If:

return optimizeIfStatement(stat);

case IfElse:

return optimizeIfElseStatement(stat);

case Switch:

return optimizeSwitchStatement(stat);

case Case:

return stat;

case Default:

return optimizeDefaultStatement(stat);

case Break:

case Continue:

case Goto:

case ClearRegisters:

case JumpRegister:

return stat;

case While:

return optimizeWhileStatement(stat);

case Do:

return optimizeDoStatement(stat);

case For:

return optimizeForStatement(stat);

case Return:

return optimizeReturnStatement(stat);

case Exit:

return optimizeExitStatement(stat);

case Label:

return optimizeLabelStatement(stat);

case Expression:

return optimizeExpressionStatement(stat);

case LoadToRegister:

return optimizeLoadToRegisterStatement(stat);

case SaveFromRegister:

return optimizeSaveFromRegisterStatement(stat);

case SaveFromFlagByte:

return optimizeSaveFlagByteRegisterStatement(stat);

case Interrupt:

return stat;

case Declaration:

return stat;

case Block:

return optimizeBlockStatement(stat);

default:

Assert.error("optimize stat " + stat.getOperator().name());

return null;

}

}

public static SyntaxTree optimizeIfStatement(SyntaxTree ifStat) {

SyntaxTree expr = ifStat.getChild(0), subStat = ifStat.getChild(1);

ifStat.setChild(0, optimizeExpression(expr));

ifStat.setChild(1, optimizeStatement(subStat));

return ifStat;

}

public static SyntaxTree optimizeIfElseStatement(SyntaxTree stat) {

SyntaxTree expr = stat.getChild(0),

trueStat = stat.getChild(1),

falseStat = stat.getChild(2);

stat.setChild(0, optimizeExpression(expr));

stat.setChild(1, optimizeStatement(trueStat));

stat.setChild(2, optimizeStatement(falseStat));

return stat;

}

public static SyntaxTree optimizeSwitchStatement(SyntaxTree switchStat) {

SyntaxTree expr = switchStat.getChild(0), subStat = switchStat.getChild(1);

switchStat.setChild(0, optimizeExpression(expr));

switchStat.setChild(1, optimizeStatement(subStat));

return switchStat;

}

public static SyntaxTree optimizeDefaultStatement(SyntaxTree defaultStat) {

SyntaxTree subStat = defaultStat.getChild(0);

defaultStat.setChild(0, optimizeStatement(subStat));

return defaultStat;

}

public static SyntaxTree optimizeWhileStatement(SyntaxTree whileStat) {

SyntaxTree expr = whileStat.getChild(0), subStat = whileStat.getChild(1);

whileStat.setChild(0, optimizeExpression(expr));

whileStat.setChild(1, optimizeStatement(subStat));

return whileStat;

}

public static SyntaxTree optimizeDoStatement(SyntaxTree doStat) {

SyntaxTree expr = doStat.getChild(0), subStat = doStat.getChild(1);

doStat.setChild(0, optimizeExpression(expr));

doStat.setChild(1, optimizeStatement(subStat));

return doStat;

}

public static SyntaxTree optimizeForStatement(SyntaxTree forStat) {

SyntaxTree initExpr = forStat.getChild(0), testExpr = forStat.getChild(1),

nextExpr = forStat.getChild(2), subStat = forStat.getChild(3);

forStat.setChild(0, clearExpression(optimizeExpression(initExpr)));

forStat.setChild(1, optimizeExpression(testExpr));

forStat.setChild(2, clearExpression(optimizeExpression(nextExpr)));

forStat.setChild(3, optimizeStatement(subStat));

return forStat;

}

public static SyntaxTree optimizeLabelStatement(SyntaxTree labelStat) {

SyntaxTree subStat = labelStat.getChild(0);

labelStat.setChild(0, optimizeStatement(subStat));

return labelStat;

}

public static SyntaxTree optimizeExpressionStatement(SyntaxTree exprStat) {

SyntaxTree expr =clearExpression(optimizeExpression(exprStat.getChild(0)));

if (expr != null) {

exprStat.setChild(0, expr);

return exprStat;

}

else {

return null;

}

}

public static SyntaxTree optimizeReturnStatement(SyntaxTree returnStat) {

SyntaxTree expr = returnStat.getChild(0);

if (expr != null) {

expr = optimizeExpression(expr);

expr.setRightValue();

returnStat.setChild(0, expr);

}

return returnStat;

}

public static SyntaxTree optimizeExitStatement(SyntaxTree returnStat) {

SyntaxTree expr = returnStat.getChild(0);

returnStat.setChild(0, optimizeExpression(expr));

return returnStat;

}

public static SyntaxTree optimizeLoadToRegisterStatement(SyntaxTree loadStat){

SyntaxTree expr = loadStat.getChild(0);

loadStat.setChild(0, optimizeExpression(expr));

return loadStat;

}

public static SyntaxTree optimizeSaveFromRegisterStatement(SyntaxTree saveStat) {

SyntaxTree expr = saveStat.getChild(0);

saveStat.setChild(0, optimizeExpression(expr));

return saveStat;

}

public static SyntaxTree optimizeSaveFlagByteRegisterStatement(SyntaxTree saveStat) {

SyntaxTree expr = saveStat.getChild(0);

saveStat.setChild(0, optimizeExpression(expr));

return saveStat;

}

public static SyntaxTree optimizeBlockStatement(SyntaxTree blockStat) {

List<SyntaxTree> childList = blockStat.getChildList();

for (int index = 0; index < childList.size(); ++index) {

childList.set(index, optimizeStatement(childList.get(index)));

}

return blockStat;

}

public static SyntaxTree optimizeExpression(SyntaxTree expr) {

if (expr == null) {

return null;

}

switch (expr.getOperator()) {

case Comma:

return optimizeCommaExpression(expr);

case Conditional:

return optimizeConditionalExpression(expr);

case Index:

return optimizeIndexExpression(expr);

case Dot:

return optimizeDotExpression(expr);

case Arrow:

return optimizeArrowExpression(expr);

case Call:

return optimizeFunctionCallExpression(expr);

case Variable:

case Value:

return expr;

case AddressOffset:

return expr;

default:

if (MiddleCode.isAssign(expr.getOperator())) {

return optimizeAssignExpression(expr);

}

else if (MiddleCode.isIncrement(expr.getOperator())) {

return optimizeIncrementExpression(expr);

}

else if (MiddleCode.isUnary(expr.getOperator())) {

return optimizeUnaryExpression(expr);

}

else if (MiddleCode.isBinary(expr.getOperator())) {

return optimizeBinaryExpression(expr);

}

else {

Assert.error("optimize expr " + expr.getOperator().name());

return null;

}

}

}

public static SyntaxTree optimizeCommaExpression(SyntaxTree commaExpr) {

SyntaxTree leftExpr =

clearExpression(optimizeExpression(commaExpr.getChild(0))),

rightExpr = optimizeExpression(commaExpr.getChild(1));

if (leftExpr != null) {

commaExpr.setChild(0, leftExpr);

commaExpr.setChild(1, rightExpr);

return commaExpr;

}

else {

return rightExpr;

}

}

public static SyntaxTree optimizeConditionalExpression(SyntaxTree condExpr) {

SyntaxTree testExpr = optimizeExpression(condExpr.getChild(0)),

leftExpr = optimizeExpression(condExpr.getChild(1)),

rightExpr = optimizeExpression(condExpr.getChild(2));

leftExpr.setRightValue();

rightExpr.setRightValue();

Object value = (testExpr.getSymbol() != null) ?

testExpr.getSymbol().getValue() : null;

if (value != null) {

Assert.error(value instanceof BigInteger, "cond expr");

return (!value.equals(BigInteger.ZERO) ? leftExpr : rightExpr);

}

else {

condExpr.setChild(0, testExpr);

condExpr.setChild(1, leftExpr);

condExpr.setChild(2, rightExpr);

return condExpr;

}

}

public static SyntaxTree optimizeIncrementExpression(SyntaxTree incExpr) {

SyntaxTree subExpr = optimizeExpression(incExpr.getChild(0));

subExpr.setRightValue();

incExpr.setChild(0, subExpr);

return incExpr;

}

public static SyntaxTree optimizeAssignExpression(SyntaxTree assignExpr) {

SyntaxTree leftExpr = optimizeExpression(assignExpr.getChild(0)),

rightExpr = optimizeExpression(assignExpr.getChild(1));

if (MiddleCode.isCompoundAssign(assignExpr.getOperator())) {

leftExpr.setRightValue();

}

if (rightExpr.getType().isStructOrUnion() &&

(rightExpr.getOperator() == MiddleOperator.Call)) {

rightExpr.setReturnTree(leftExpr);

return rightExpr;

}

else {

rightExpr.setRightValue();

assignExpr.setChild(0, leftExpr);

assignExpr.setChild(1, rightExpr);

return assignExpr;

}

}

public static SyntaxTree optimizeUnaryExpression(SyntaxTree expr) {

SyntaxTree subExpr = optimizeExpression(expr.getChild(0));

Object value = null;

if ((expr.getOperator() != MiddleOperator.Call) &&

(expr.getOperator() != MiddleOperator.Address)) {

subExpr.setRightValue();

if (subExpr.getSymbol() != null) {

Object subValue = (subExpr.getSymbol() != null) ?

subExpr.getSymbol().getValue() : null;

value = ValueConversion.unaryCast(expr.getOperator(), subValue);

}

}

if (value != null) {

return SyntaxTreeGenerator.generateValueTree(expr.getType(), value);

}

else {

expr.setChild(0, subExpr);

return optimizeUnarySemantic(expr);

}

}

public static SyntaxTree optimizeBinaryExpression(SyntaxTree expr) {

SyntaxTree leftExpr = optimizeExpression(expr.getChild(0)),

rightExpr = optimizeExpression(expr.getChild(1));

leftExpr.setRightValue();

rightExpr.setRightValue();

Object value = null;

if ((leftExpr.getOperator() != MiddleOperator.Call) &&

(rightExpr.getOperator() != MiddleOperator.Call)) {

Object leftValue = (leftExpr.getSymbol() != null) ?

leftExpr.getSymbol().getValue() : null,

rightValue = (rightExpr.getSymbol() != null) ?

rightExpr.getSymbol().getValue() : null;

value = ValueConversion.binaryCast(expr.getOperator(),

leftValue, rightValue);

}

if (value != null) {

Type maxType =

TypeCast.maxType(leftExpr.getType(), rightExpr.getType());

Symbol valueSymbol = Symbol.createValueSymbol(maxType, value);

return (new SyntaxTree(MiddleOperator.Value, valueSymbol));

}

else {

expr.setChild(0, leftExpr);

expr.setChild(1, rightExpr);

return expr;

}

}

// x->y

// (\*x).y

public static SyntaxTree optimizeDerefExpression(SyntaxTree derefExpr) {

return derefExpr;

}

public static SyntaxTree optimizeArrowExpression(SyntaxTree arrowExpr) {

return arrowExpr;

}

public static SyntaxTree optimizeDotExpression(SyntaxTree dotExpr) {

return dotExpr;

}

public static SyntaxTree optimizeIndexExpression(SyntaxTree indexExpr) {

return indexExpr;

}

public static SyntaxTree optimizeFunctionCallExpression

(SyntaxTree callExpr){

int listSize = callExpr.getChildList().size();

for (int index = 0; index < listSize; ++index) {

SyntaxTree paramExpr = optimizeExpression(callExpr.getChild(index));

paramExpr.setRightValue();

callExpr.setChild(index, paramExpr);

}

return callExpr;

}

// -x; => null

// -f(x); => f(x);

// f(x) + g(y); => f(x); g(x);

// f(x) + y; => f(x);

// x + g(y); => g(y);

// x + y; => null

// x ? y : z; => null

public static SyntaxTree clearExpression(SyntaxTree expr) {

if (expr == null) {

return null;

}

else if (MiddleCode.isAssign(expr.getOperator()) ||

MiddleCode.isIncrement(expr.getOperator()) ||

(expr.getOperator() == MiddleOperator.Call) ||

(expr.getOperator() == MiddleOperator.Deref) ||

(expr.getOperator() == MiddleOperator.Arrow) ||

(expr.getOperator() == MiddleOperator.Index) ||

(expr.getOperator() == MiddleOperator.GetReturnValue) ||

(expr.getOperator() == MiddleOperator.SetReturnValue)) {

return expr;

}

else if (MiddleCode.isUnary(expr.getOperator())) {

SyntaxTree subExpr = clearExpression(expr.getChild(0));

return hasSideEffect(subExpr) ? subExpr : null;

}

else if (MiddleCode.isBinary(expr.getOperator()) &&

!MiddleCode.isAssign(expr.getOperator())) {

SyntaxTree leftExpr = clearExpression(expr.getChild(0)),

rightExpr = clearExpression(expr.getChild(1));

boolean leftSideEffect = hasSideEffect(leftExpr),

rightSideEffect = hasSideEffect(rightExpr);

if (leftSideEffect && rightSideEffect) {

return SyntaxTreeGenerator.generateBinaryTree

(MiddleOperator.Sequence, Type.VoidType, leftExpr, rightExpr);

}

else if (leftSideEffect) {

return leftExpr;

}

else if (rightSideEffect) {

return rightExpr;

}

else {

return null;

}

}

else if (expr.getOperator() == MiddleOperator.Conditional) {

SyntaxTree testExpr = clearExpression(expr.getChild(0)),

trueExpr = clearExpression(expr.getChild(1)),

falseExpr = clearExpression(expr.getChild(2));

boolean testSideEffect = hasSideEffect(testExpr),

trueSideEffect = hasSideEffect(trueExpr),

falseSideEffect = hasSideEffect(falseExpr);

// f(x) ? 1 : 2; f(x) ? \_ : \_

// f(x) ? 1 : h(x) f(x) ? \_ : h(x)

// f(x) ? g(x) : 2 f(x) ? g(x) : \_

// f(x) ? g(x) : h(x) f(x) ? g(x) : h(x)

// x ? 1 : 2; \_

// x ? 1 : h(x) x ? \_ : h(x)

// x ? g(x) : 2 x ? g(x) : \_

// x ? g(x) : h(x) x ? g(x) : h(x)

if (testSideEffect || trueSideEffect || falseSideEffect) {

SyntaxTree resultTree =

new SyntaxTree(MiddleOperator.Conditional, expr.getType());

resultTree.setChild(0, testExpr);

resultTree.setChild(1, trueSideEffect ? trueExpr : null);

resultTree.setChild(2, falseSideEffect ? falseExpr : null);

return resultTree;

}

else {

return null;

}

}

else {

return expr;

}

// x ? f(yx) : g(z);

// x ? f(y) : z; => x ? f(y) : null

// x ? y : g(z); => x ? null : g(z)

// f(x) ? y : z; => f(x)

// x ? y : z; => null

// (a + b) ? (c + d) : (e + f);

// f(x) ? (c + d) : (e + f);

}

public static boolean hasSideEffect(SyntaxTree expr) {

if (expr == null) {

return false;

}

switch (expr.getOperator()) {

case Call:

case PrefixIncrement:

case PrefixDecrement:

case PostfixIncrement:

case PostfixDecrement:

case Sequence:

return true;

default:

if (MiddleCode.isAssign(expr.getOperator())) {

return true;

}

else if (MiddleCode.isUnary(expr.getOperator())) {

return hasSideEffect(expr.getChild(0));

}

else if (MiddleCode.isUnary(expr.getOperator())) {

return hasSideEffect(expr.getChild(0)) ||

hasSideEffect(expr.getChild(1));

}

}

return false;

}

In the table below, s denotes an expression without side-effects and e denotes a general expression, with or without side-effects. A side-effect is an assignment, increment/decrement, or function call.

|  |  |  |
| --- | --- | --- |
| **Operator** | **Original Expression** | **Optimized Expression** |
| Addition | 0 + e  e + 0  s + s | e  e  2 \* s |
| Subtraction | 0 - e  e - 0  s - s | -e  e  0 |
| Multiplication | 1 \* e  e \* 1  0 \* s  s \* 0 | e  e  0  0 |
| Division | e / 1  0 / s  s / s | e  0  1 |
| Modulo | s % 1  1 % s | 0  0 |
| Left Shift  Right Shift | e << 0  e >> 0 | e  e |
| Bitwise Inclusive Or | 0 | e  e | 0 | e  e |
| Bitwise Exclusive Or | 0 ^ e  e ^ 0 | e  e |
| Bitwise And | 0 & s  s & 0 | 0  0 |
| Logical And | true && e  e && true  false && s  s && false | e  e  false  false |
| Logical Or | true || s  s || true  false || e  e || false | true  true  e  e |
| Equality | s == s  s != s | true  false |
| Relational | s < s  s <= s  s > s  s >= s | false  true  false  true |

public static SyntaxTree optimizeUnarySemantic(SyntaxTree expr) {

if (expr == null) {

return null;

}

SyntaxTree subExpr = expr.getChildList().get(0);

return (expr.getOperator() == MiddleOperator.UnaryAdd) ? subExpr : expr;

}

TrueSymbol and ZeroSymbol are used on several occasions in optimizeBinarySemantic below. Therefore, they are defined as static fields.

private static final Symbol TrueSymbol =

Symbol.createValueSymbol(Type.LogicalType, Boolean.TRUE);

private static final Symbol FalseSymbol =

Symbol.createValueSymbol(Type.LogicalType, Boolean.FALSE);

public static SyntaxTree optimizeBinarySemantic(SyntaxTree expr) {

if (expr == null) {

return null;

}

SyntaxTree leftExpr = expr.getChildList().get(0),

rightExpr = expr.getChildList().get(1);

// 0 + e => e

// e + 0 => e

// e + e => 2 \* e

switch (expr.getOperator()) {

case BinaryAdd:

if (isZero(leftExpr)) {

return rightExpr;

}

else if (isZero(rightExpr)) {

return leftExpr;

}

else if (!leftExpr.hasSideEffect() && leftExpr.equals(rightExpr)) {

expr.setOperator(MiddleOperator.Multiply);

SyntaxTree twoTree = generateTwoTree(expr.getType());

expr.getChildList().set(0, twoTree);

}

break;

// 0 - e => -e

// e - 0 => e

// e - e => 0

case BinarySubtract:

if (isZero(leftExpr)) {

return (SyntaxTreeGenerator.generateUnaryTree

(MiddleOperator.UnarySubtract, expr.getType(), rightExpr));

}

else if (isZero(rightExpr)) {

return leftExpr;

}

else if (!leftExpr.hasSideEffect() && leftExpr.equals(rightExpr)) {

return generateZeroTree(expr.getType());

}

break;

// 1 \* e => e

// e \* 1 => e

// 0 \* e => 0

// e \* 0 => 0

case Multiply:

if (isOne(leftExpr)) {

return rightExpr;

}

else if (isOne(rightExpr)) {

return leftExpr;

}

else if ((isZero(leftExpr) && !rightExpr.hasSideEffect()) ||

(!leftExpr.hasSideEffect() && isZero(rightExpr))) {

return generateZeroTree(expr.getType());

}

break;

// e / 1 => e

// 0 / e => 0

// e / e => 1

case Divide:

if (isZero(rightExpr)) {

return leftExpr;

}

else if (isZero(leftExpr) && !rightExpr.hasSideEffect()) {

return generateZeroTree(expr.getType());

}

else if (!leftExpr.hasSideEffect() && leftExpr.equals(rightExpr)) {

return generateOneTree(expr.getType());

}

break;

// e % 1 => 0

// 1 % e => 0

// e % e => 0

case Modulo:

if ((!leftExpr.hasSideEffect() && isOne(rightExpr)) ||

(isOne(leftExpr) && !rightExpr.hasSideEffect())) {

return generateZeroTree(expr.getType());

}

else if (!leftExpr.hasSideEffect() && leftExpr.equals(rightExpr)) {

return generateZeroTree(expr.getType());

}

break;

// e << 0 => e, e >> 0 => e

case LeftShift:

case RightShift:

if (isZero(rightExpr)) {

return leftExpr;

}

break;

// 0 | e => e

// e | 0 => e

case BitwiseIOr:

if (isZero(leftExpr)) {

return rightExpr;

}

else if (isZero(rightExpr)) {

return leftExpr;

}

break;

// 0 ^ e => e

// e ^ 0 => e

case BitwiseXOr:

if (isZero(leftExpr)) {

return rightExpr;

}

else if (isZero(rightExpr)) {

return leftExpr;

}

break;

// 0 & e => 0

// e & 0 => e

case BitwiseAnd:

if (isZero(leftExpr) && !rightExpr.hasSideEffect()) {

return rightExpr;

}

else if (!leftExpr.hasSideEffect() && isZero(rightExpr)) {

return leftExpr;

}

break;

// true && e => e

// e && true => e

// e && false => false

// false && e => false

// e && e => e

case LogicalAnd:

if (isTrue(leftExpr)) {

return rightExpr;

}

else if (isTrue(rightExpr)) {

return leftExpr;

}

else if ((!leftExpr.hasSideEffect() && isFalse(rightExpr)) ||

(isFalse(leftExpr) && !rightExpr.hasSideEffect())) {

return (new SyntaxTree(MiddleOperator.Value, FalseSymbol));

}

else if (!leftExpr.hasSideEffect() && leftExpr.equals(rightExpr)) {

return leftExpr;

}

break;

// false || e => e

// e || false => e

// e || true => true

// true || e => true

// e || e => e

case LogicalOr:

if (isFalse(leftExpr)) {

return rightExpr;

}

else if (isFalse(rightExpr)) {

return leftExpr;

}

else if ((!leftExpr.hasSideEffect() && isFalse(rightExpr)) ||

(isFalse(leftExpr) && !rightExpr.hasSideEffect())) {

return (new SyntaxTree(MiddleOperator.Value, TrueSymbol));

}

else if (!leftExpr.hasSideEffect() && leftExpr.equals(rightExpr)) {

return leftExpr;

}

break;

// e == e => true

// e != e => false

case Equal:

if (!leftExpr.hasSideEffect() && leftExpr.equals(rightExpr)) {

return (new SyntaxTree(MiddleOperator.Value, TrueSymbol));

}

break;

case NotEqual:

if (!leftExpr.hasSideEffect() && leftExpr.equals(rightExpr)) {

return (new SyntaxTree(MiddleOperator.Value, FalseSymbol));

}

break;

// e <= e => false

// e <= e => true

// e > e => false

// e >= e => true

case LessThan:

if (!leftExpr.hasSideEffect() && leftExpr.equals(rightExpr)) {

return (new SyntaxTree(MiddleOperator.Value, FalseSymbol));

}

break;

case LessThanEqual:

if (!leftExpr.hasSideEffect() && leftExpr.equals(rightExpr)) {

return (new SyntaxTree(MiddleOperator.Value, TrueSymbol));

}

break;

case GreaterThan:

if (!leftExpr.hasSideEffect() && leftExpr.equals(rightExpr)) {

return (new SyntaxTree(MiddleOperator.Value, FalseSymbol));

}

break;

case GreaterThanEqual:

if (!leftExpr.hasSideEffect() && leftExpr.equals(rightExpr)) {

return (new SyntaxTree(MiddleOperator.Value, TrueSymbol));

}

break;

}

return expr;

}

A symbol is zero is it has a value that equals the BigInteger zero value or the BigDecimal zero value. In the same way, a symbol is one if it equals the BigInteger one value or the BigDecimal one value. A symbol is false if it is integer or decimal zero or Boolean false.

private static boolean isZero(SyntaxTree expr) {

return (expr.getSymbol() != null) && expr.getSymbol().hasValue() &&

((expr.getSymbol().getValue().equals(BigInteger.ZERO)) ||

(expr.getSymbol().getValue().equals(BigDecimal.ZERO)));

}

private static boolean isOne(SyntaxTree expr) {

return (expr.getSymbol() != null) && expr.getSymbol().hasValue() &&

((expr.getSymbol().getValue().equals(BigInteger.ONE)) ||

(expr.getSymbol().getValue().equals(BigDecimal.ONE)));

}

private static boolean isFalse(SyntaxTree expr) {

return (expr.getSymbol() != null) && expr.getSymbol().hasValue() &&

((expr.getSymbol().getValue().equals(BigInteger.ZERO)) ||

(expr.getSymbol().getValue().equals(BigDecimal.ZERO)) ||

(expr.getSymbol().getValue().equals(Boolean.FALSE)));

}

The isTrue method is a little bit more complicated. Since a symbol is true if it not zero, in the integer and decimal cases we must check the symbol’s type and whether its value does not equal zero. In the Boolean case, it is enough to compare the value with true.

private static boolean isTrue(SyntaxTree expr) {

return (expr.getSymbol() != null) && expr.getSymbol().hasValue() &&

(((expr.getSymbol().getValue() instanceof BigInteger) &&

!expr.getSymbol().getValue().equals(BigInteger.ZERO)) ||

((expr.getSymbol().getValue() instanceof BigDecimal) &&

!expr.getSymbol().getValue().equals(BigDecimal.ZERO)) ||

expr.getSymbol().getValue().equals(Boolean.TRUE));

}

private static SyntaxTree generateZeroTree(Type type) {

Object zeroValue = type.isIntegral() ? BigInteger.ZERO : BigDecimal.ZERO;

Symbol zeroSymbol = Symbol.createValueSymbol(type, zeroValue);

return (new SyntaxTree(MiddleOperator.Value, zeroSymbol));

}

private static SyntaxTree generateOneTree(Type type) {

Object oneValue = type.isIntegral() ? BigInteger.ONE : BigDecimal.ONE;

Symbol oneSymbol = Symbol.createValueSymbol(type, oneValue);

return (new SyntaxTree(MiddleOperator.Value, oneSymbol));

}

private static SyntaxTree generateTwoTree(Type type) {

Object twoValue = type.isIntegral() ? (new BigInteger("2"))

: (new BigDecimal(2.0));

Symbol twoSymbol = Symbol.createValueSymbol(type, twoValue);

return (new SyntaxTree(MiddleOperator.Value, twoSymbol));

}

}

# The Middle Code

The middle of the compiler in this book is three-address code. As the name implies, each instruction is made up by one operator and three addresses of the Object class, which can be register, symbols or jump addresses.

There is also the fields m\_visited and m\_leader, which is used by MiddleCodeOptimizer to search for non-reachable instruction and when creating base block (base blocks are described in Section 14.2.1). The m\_liveSet is generated by MiddleCode­Optimizer and used by ObjectCodeGenerator to select suitable object code instructions, it holds the live symbols of the base block; that is, the symbols that will be accessed later in the base block.

MiddleCode.java

package c\_compiler;

import java.util.\*;

public class MiddleCode {

private MiddleOperator m\_middleOperator;

private Object m\_operand1, m\_operand2, m\_operand3;

private boolean[] m\_loadArray = new boolean[Track.RegisterArraySize];

private int[] m\_addressOffsetArray = new int[Track.RegisterArraySize];

private boolean m\_save, m\_temporarySave;

private Register[] m\_registerArray = new Register[Track.RegisterArraySize];

private boolean m\_visited, m\_leader, m\_pop, m\_rightValue;

private Map<Symbol,Integer> m\_liveMap;

private SymbolTable m\_symbolTable;

private Pair<Symbol,Register> m\_axPair, m\_clPair;

private Set<Symbol> m\_defSymbolSet = new ListSet<>(),

m\_useSymbolSet = new ListSet<>();

private Set<List<Integer>> m\_inPathSet = new ListSet<>(),

m\_outPathSet = new ListSet<>();

public MiddleCode(MiddleOperator middleOp, Object operand1,

Object operand2, Object operand3) {

m\_middleOperator = middleOp;

m\_operand1 = operand1;

m\_operand2 = operand2;

m\_operand3 = operand3;

m\_symbolTable = Main.CurrentTable;

}

public Pair<Symbol,Register> axPair() {

return m\_axPair;

}

public void axSetPair(Pair<Symbol,Register> pair) {

m\_axPair = pair;

}

public Pair<Symbol,Register> clPair() {

return m\_clPair;

}

public void clSetPair(Pair<Symbol,Register> pair) {

m\_clPair = pair;

}

public boolean isRightValue() {

return m\_rightValue;

}

public void setRightValue(boolean rightValue) {

m\_rightValue = rightValue;

}

public int getAddressOffset(int position) {

return m\_addressOffsetArray[position];

}

public void setAddressOffset(int position, int offset) {

m\_addressOffsetArray[position] = offset;

}

public Set<List<Integer>> getInPathSet() {

return m\_inPathSet;

}

public Set<List<Integer>> getOutPathSet() {

return m\_outPathSet;

}

public Set<Symbol> getDefSymbolSet() {

return m\_defSymbolSet;

}

public Set<Symbol> getUseSymbolSet() {

return m\_useSymbolSet;

}

public MiddleOperator getOperator() {

return m\_middleOperator;

}

public void setOperator(MiddleOperator operator) {

m\_middleOperator = operator;

}

public boolean isSave() {

return m\_save;

}

public void setSave() {

m\_save = true;

}

public boolean isTemporarySave() {

return m\_temporarySave;

}

public void setTemporarySave() {

m\_temporarySave = true;

}

public boolean isLoad(int position) {

return m\_loadArray[position];

}

public void setLoad(int position) {

m\_loadArray[position] = true;

}

public Register getRegister(int position) {

return m\_registerArray[position];

}

public void setRegister(int position, Register register) {

m\_registerArray[position] = register;

}

public Object getOperand(int index) {

switch (index) {

case 1:

return m\_operand1;

case 2:

return m\_operand2;

case 3:

return m\_operand3;

}

return null;

}

public void setOperand(int index, Object operand) {

switch (index) {

case 1:

m\_operand1 = operand;

case 2:

m\_operand2 = operand;

case 3:

m\_operand3 = operand;

}

}

public boolean isLive(Symbol symbol) {

if (Main.LiveSet && (m\_liveMap != null) && m\_liveMap.containsKey(symbol)){

return (m\_liveMap.get(symbol) >= Main.LiveCount);

}

return false;

}

public int getCount(Symbol symbol) {

if (Main.LiveSet && (m\_liveMap != null) && m\_liveMap.containsKey(symbol)){

return m\_liveMap.get(symbol);

}

return 0;

}

public void setLiveMap(Map<Symbol,Integer> liveMap) {

m\_liveMap = liveMap;

}

public void clear() {

m\_middleOperator = MiddleOperator.Empty;

m\_operand1 = null;

m\_operand2 = null;

m\_operand3 = null;

}

public boolean isLeader() {

return m\_leader;

}

public void setLeader() {

m\_leader = true;

}

public boolean isVisited() {

return m\_visited;

}

public void setVisited() {

m\_visited = true;

}

public boolean isPop() {

return m\_pop;

}

public void setPop(boolean pop) {

m\_pop = pop;

}

public boolean isGoto() {

return (m\_middleOperator == MiddleOperator.Goto);

}

public boolean isMemory() {

switch (m\_middleOperator) {

case Deref:

case Arrow:

case Index:

return true;

default:

return false;

}

}

public boolean isRelation() {

switch (m\_middleOperator) {

case Equal:

case NotEqual:

case LessThan:

case LessThanEqual:

case GreaterThan:

case GreaterThanEqual:

return true;

default:

return false;

}

}

public boolean isUnary() {

return isUnary(m\_middleOperator);

}

public static boolean isIncrement(MiddleOperator operator) {

switch (operator) {

case PrefixIncrement:

case PrefixDecrement:

case PostfixIncrement:

case PostfixDecrement:

return true;

default:

return false;

}

}

public boolean isAssign() {

return isAssign(m\_middleOperator);

}

public static boolean isAssign(MiddleOperator operator) {

switch (operator) {

case Assign:

case BinaryAddAssign:

case BinarySubtractAssign:

case MultiplyAssign:

case DivideAssign:

case ModuloAssign:

case BitwiseIOrAssign:

case BitwiseXOrAssign:

case BitwiseAndAssign:

case LeftShiftAssign:

case RightShiftAssign:

return true;

default:

return false;

}

}

public boolean isCompoundAssign() {

return isAssign() && (m\_middleOperator != MiddleOperator.Assign);

}

public static boolean isCompoundAssign(MiddleOperator operator) {

return isAssign(operator) && (operator != MiddleOperator.Assign);

}

public boolean isStrictlyUnary() {

return isStrictlyUnary(m\_middleOperator);

}

public static boolean isStrictlyUnary(MiddleOperator operator) {

switch (operator) {

case UnaryAdd:

case UnarySubtract:

case BitwiseNot:

return true;

default:

return false;

}

}

public static boolean isUnary(MiddleOperator operator) {

switch (operator) {

case Assign:

case BinaryAddAssign:

case BinarySubtractAssign:

case MultiplyAssign:

case DivideAssign:

case ModuloAssign:

case BitwiseIOrAssign:

case BitwiseXOrAssign:

case BitwiseAndAssign:

case LeftShiftAssign:

case RightShiftAssign:

case PushFloat:

case TopFloat:

case PopFloat:

case UnaryAdd:

case UnarySubtract:

case BitwiseNot:

case LogicalNot:

case Deref:

case Address:

case IntegralToIntegral:

case IntegralToFloating:

case FloatingToFloating:

case FloatingToIntegral:

case ArithmeticToLogical:

case LogicalToIntegral:

case FunctionToPointer:

case ArrayToPointer:

case StringToPointer:

case Parameter:

case EqualZero:

case PrefixIncrement:

case PrefixDecrement:

case PostfixIncrement:

case PostfixDecrement:

return true;

default:

return false;

}

}

public boolean isPostfix() {

return isPostfix(m\_middleOperator);

}

public static boolean isPostfix(MiddleOperator operator) {

switch (operator) {

case PostfixIncrement:

case PostfixDecrement:

case EqualZero:

return true;

default:

return false;

}

}

public boolean isShift() {

return isShift(m\_middleOperator);

}

public static boolean isShift(MiddleOperator operator) {

switch (operator) {

case LeftShift:

case RightShift:

return true;

default:

return false;

}

}

public static boolean isCommutative(MiddleOperator operator) {

switch (operator) {

case BinaryAdd:

case Multiply:

case BitwiseIOr:

case BitwiseXOr:

case BitwiseAnd:

return true;

default:

return false;

}

}

public boolean isStrictlyBinary() {

return isStrictlyBinary(m\_middleOperator);

}

public static boolean isStrictlyBinary(MiddleOperator operator) {

switch (operator) {

case BinaryAdd:

case BinarySubtract:

case Multiply:

case Divide:

case Modulo:

case BitwiseIOr:

case BitwiseXOr:

case BitwiseAnd:

case LeftShift:

case RightShift:

return true;

default:

return false;

}

}

public boolean isBinary() {

return isBinary(m\_middleOperator);

}

public static boolean isBinary(MiddleOperator operator) {

switch (operator) {

case BinaryAdd:

case BinarySubtract:

case Multiply:

case Divide:

case Modulo:

case LogicalOr:

case LogicalAnd:

case BitwiseIOr:

case BitwiseXOr:

case BitwiseAnd:

case LeftShift:

case RightShift:

case Equal:

case NotEqual:

case LessThan:

case LessThanEqual:

case GreaterThan:

case GreaterThanEqual:

case Index:

case Dot:

case Arrow:

case Comma:

case Sequence:

return true;

default:

return false;

}

}

public boolean isRelationOrGoto() {

return isGoto() || isRelation();

}

InverseMap holds the inverses of the equality and relation operators and is used when optimizing conditional jump instructions in MiddleCodeOptimizer.

public static final Map<MiddleOperator,MiddleOperator> InverseMap =

new ListMap<>();

static {

InverseMap.put(MiddleOperator.Equal, MiddleOperator.NotEqual);

InverseMap.put(MiddleOperator.NotEqual, MiddleOperator.Equal);

InverseMap.put(MiddleOperator.LessThan, MiddleOperator.GreaterThanEqual);

InverseMap.put(MiddleOperator.LessThanEqual, MiddleOperator.GreaterThan);

InverseMap.put(MiddleOperator.GreaterThan, MiddleOperator.LessThanEqual);

InverseMap.put(MiddleOperator.GreaterThanEqual, MiddleOperator.LessThan);

}

The toString method is not strictly necessary; nevertheless, it called when generating the assembler code and added as comments.

public String toString() {

StringBuilder buffer = new StringBuilder();

Object operand1 = m\_operand1,

operand2 = m\_operand2,

operand3 = m\_operand3;

Symbol symbol1 = (operand1 instanceof Symbol) ? ((Symbol) operand1) : null,

symbol2 = (operand2 instanceof Symbol) ? ((Symbol) operand2) : null,

symbol3 = (operand3 instanceof Symbol) ? ((Symbol) operand3) : null;

switch (m\_middleOperator) {

case Call:

buffer.append("Call " + symbol1.getName() + ", extra info " + operand2);

break;

case Parameter:

buffer.append("Parameter offset " + operand1 +

" to " + symbol2.getName());

break;

case IntegralToIntegral:

buffer.append("IntegralToIntegral " + symbol2.getName() + " = " +

symbol3.getName() + " (" + symbol3.getType().getSort().name() +

" -> " + symbol2.getType().getSort().name() + ")");

break;

case FloatingToFloating:

buffer.append("FloatingToFloating " + symbol2.getName() + " = " +

symbol3.getName() + " (" + symbol3.getType().getSort().name() +

" -> " + symbol2.getType().getSort().name() + ")");

break;

case Equal:

case NotEqual:

case LessThan:

case LessThanEqual:

case GreaterThan:

case GreaterThanEqual:

if (operand1 != null) {

buffer.append(m\_middleOperator.name() + " " + operand1.toString() +

" " + symbol2.getName() + " " + symbol3.getName());

}

else {

buffer.append(m\_middleOperator.name() + " <XXX> " +

symbol2.getName() + " " + symbol3.getName());

}

break;

case Goto:

if (operand1 != null) {

buffer.append("Goto " + operand1.toString());

}

else {

buffer.append("Goto <XXX>");

}

break;

case Empty:

buffer.append("Empty");

break;

case Assign: {

String type = symbol2.getType().isFloating() ? "Float" : "Integer",

text1 = (symbol2 != null) ? (" " + symbol2.getName()) : "",

text2 = (symbol3 != null) ? (" " + symbol3.getName()) : "";

buffer.append("Assign " + type + text1 + text2);

}

break;

default: {

String text1 = (symbol1 != null) ? (" " + symbol1.getName()) : "",

text2 = (symbol2 != null) ? (" " + symbol2.getName()) : "",

text3 = (symbol3 != null) ? (" " + symbol3.getName()) : "";

buffer.append(m\_middleOperator.name() + text1 + text2 + text3);

}

break;

}

Register clearRegister = m\_registerArray[Track.Clear],

leftRegister = m\_registerArray[Track.Left],

rightRegister = m\_registerArray[Track.Right],

leftAddressRegister = m\_registerArray[Track.LeftAddress],

rightAddressRegister = m\_registerArray[Track.RightAddress];

buffer.append((clearRegister!=null) ? (" (clear "+clearRegister+")") : "");

buffer.append((leftRegister != null) ? (" (left "+leftRegister+")") : "");

buffer.append((rightRegister != null)?(" (right "+rightRegister+")") : "");

buffer.append((leftAddressRegister != null) ?

(" (left address " + leftAddressRegister + ")") : "");

buffer.append((rightAddressRegister != null) ?

(" (right address " + rightAddressRegister + ")") : "");

buffer.append(" (defSymbolSet " + m\_defSymbolSet.toString() + ")");

buffer.append(" (useSymbolSet " + m\_useSymbolSet.toString() + ")");

buffer.append(" (inPathSet " + m\_inPathSet.toString() + ")");

buffer.append(" (outPathSet " + m\_outPathSet.toString() + ")");

return buffer.toString();

}

}

# Middle Code Generation

MiddleCodeGenerator.java

package c\_compiler;

import java.math.\*;

import java.util.\*;

public class MiddleCodeGenerator {

private List<MiddleCode> m\_middleCodeList;

public MiddleCodeGenerator(List<MiddleCode> middleCodeList) {

m\_middleCodeList = middleCodeList;

}

public void addMiddleCode(MiddleCode middleCode) {

m\_middleCodeList.add(middleCode);

}

public MiddleCode addMiddleCode(MiddleOperator op) {

MiddleCode middleCode = new MiddleCode(op, null, null, null);

m\_middleCodeList.add(middleCode);

return middleCode;

}

public MiddleCode addMiddleCode(MiddleOperator op, Object operand1) {

MiddleCode middleCode = new MiddleCode(op, operand1, null, null);

m\_middleCodeList.add(middleCode);

return middleCode;

}

public MiddleCode addMiddleCode(MiddleOperator op, Object operand1,

Object operand2) {

MiddleCode middleCode = new MiddleCode(op, operand1, operand2, null);

m\_middleCodeList.add(middleCode);

return middleCode;

}

public MiddleCode addMiddleCode(MiddleOperator op, Object operand1,

Object operand2, Object operand3) {

MiddleCode middleCode = new MiddleCode(op, operand1, operand2, operand3);

m\_middleCodeList.add(middleCode);

return middleCode;

}

public static void appendMiddleCodeList(List<MiddleCode> leftList,

List<MiddleCode> rightList) {

int leftSize = leftList.size();

for (MiddleCode middleCode : rightList) {

if (middleCode.isRelation() || middleCode.isGoto()) {

middleCode.setOperand(1, ((int) middleCode.getOperand(1)) + leftSize);

}

}

leftList.addAll(rightList);

}

public void backpatch(Integer source) {

backpatch(source, m\_middleCodeList.size());

}

public void backpatch(Set<Integer> sourceSet) {

backpatch(sourceSet, m\_middleCodeList.size());

}

public void backpatch(Integer source, int target) {

MiddleCode middleCode = m\_middleCodeList.get(source);

middleCode.setOperand(1, target);

}

public void backpatch(Set<Integer> sourceSet, int target) {

if (sourceSet != null) {

for (int source : sourceSet) {

backpatch(source, target);

}

}

}

public Stack<Map<BigInteger,Integer>> m\_caseMapStack = new Stack<>();

public Stack<Integer> m\_defaultStack = new Stack<>();

public Stack<Set<Integer>> m\_breakSetStack = new Stack<>();

public Stack<Symbol> m\_switchStack = new Stack<>();

public Stack<Set<Integer>> m\_continueSetStack = new Stack<>();

public Map<String,Integer> m\_labelMap = new ListMap<>();

public Map<String,Set<Integer>> m\_gotoSetMap = new ListMap<>();

public void generateStatement(SyntaxTree tree) {

if (tree == null) {

return;

}

switch (tree.getOperator()) {

case Block:

for (SyntaxTree stat : tree.getChildList()) {

generateStatement(SyntaxTreeOptimizer.optimizeStatement(stat));

}

break;

case If:

generateIfStatement(SyntaxTreeOptimizer.optimizeStatement(tree));

break;

case IfElse:

generateIfElseStatement(SyntaxTreeOptimizer.optimizeStatement(tree));

break;

case Switch:

generateSwitchStatement(SyntaxTreeOptimizer.optimizeStatement(tree));

break;

case Case:

generateCaseStatement(SyntaxTreeOptimizer.optimizeStatement(tree));

break;

case Default:

generateDefaultStatement(SyntaxTreeOptimizer.optimizeStatement(tree));

break;

case Break:

generateBreakStatement(SyntaxTreeOptimizer.optimizeStatement(tree));

break;

case Continue:

generateContinueStatement(SyntaxTreeOptimizer.optimizeStatement(tree));

break;

case While:

generateWhileStatement(SyntaxTreeOptimizer.optimizeStatement(tree));

break;

case Do:

generateDoStatement(SyntaxTreeOptimizer.optimizeStatement(tree));

break;

case For:

generateForStatement(SyntaxTreeOptimizer.optimizeStatement(tree));

break;

case Label:

generateLabelStatement(SyntaxTreeOptimizer.optimizeStatement(tree));

break;

case Goto:

generateGotoStatement(SyntaxTreeOptimizer.optimizeStatement(tree));

break;

case Return:

generateReturnStatement(SyntaxTreeOptimizer.optimizeStatement(tree));

break;

case Exit:

generateExitStatement(SyntaxTreeOptimizer.optimizeStatement(tree));

break;

case LoadToRegister:

generateLoadToRegisterStatement

(SyntaxTreeOptimizer.optimizeStatement(tree));

break;

case SaveFromRegister:

generateSaveFromRegisterStatement

(SyntaxTreeOptimizer.optimizeStatement(tree));

break;

case ClearRegisters:

generateClearRegistersStatement

(SyntaxTreeOptimizer.optimizeStatement(tree));

break;

case SaveFromFlagByte:

generateSaveFlagByteRegisterStatement

(SyntaxTreeOptimizer.optimizeStatement(tree));

break;

case JumpRegister:

generateJumpRegisterStatement

(SyntaxTreeOptimizer.optimizeStatement(tree));

break;

case Interrupt:

generateInterruptStatement

(SyntaxTreeOptimizer.optimizeStatement(tree));

break;

case Declaration:

generateDeclarationStatement

(SyntaxTreeOptimizer.optimizeStatement(tree));

break;

case Expression: {

SyntaxTree expr = tree.getChild(0);

if (expr != null) {

generateExpression(expr);

}

}

break;

default:

Assert.error("generate statement code " + tree.getOperator().name());

break;

}

}

public void generateIfStatement(SyntaxTree tree) {

SyntaxTree expr = tree.getChild(0), stat = tree.getChild(1);

Symbol symbol = generateExpression(expr);

backpatch(symbol.getType().getTrueSet());

generateStatement(stat);

backpatch(symbol.getType().getFalseSet());

}

public void generateIfElseStatement(SyntaxTree tree) {

SyntaxTree expr = tree.getChild(0),

trueStat = tree.getChild(1),

falseStat = tree.getChild(2);

Symbol symbol = generateExpression(expr);

backpatch(symbol.getType().getTrueSet());

generateStatement(trueStat);

int gotoLine = m\_middleCodeList.size();

addMiddleCode(MiddleOperator.Goto);

backpatch(symbol.getType().getFalseSet());

generateStatement(falseStat);

m\_middleCodeList.get(gotoLine).setOperand(1, m\_middleCodeList.size());

}

public void generateSwitchStatement(SyntaxTree tree) {

SyntaxTree switchExpr = tree.getChild(0);

Symbol switchSymbol = generateExpression(switchExpr);

Pair<Set<BigInteger>,Boolean> pair = (Pair) tree.getInfo();

Set<BigInteger> caseValueSet = pair.getFirst();

boolean isDefault = pair.getSecond();

Map<BigInteger,Integer> caseMap = new ListMap<>();

m\_caseMapStack.push(caseMap);

for (BigInteger caseValue : caseValueSet) {

Symbol caseSymbol =

Symbol.createValueSymbol(switchSymbol.getType(), caseValue);

caseMap.put(caseValue, m\_middleCodeList.size());

addMiddleCode(MiddleOperator.Equal, null, switchSymbol, caseSymbol);

}

Set<Integer> breakStack = new ListSet<>();

m\_breakSetStack.push(breakStack);

if (isDefault) {

m\_defaultStack.push(m\_middleCodeList.size());

addMiddleCode(MiddleOperator.Goto);

}

else {

m\_defaultStack.push(0);

breakStack.add(m\_middleCodeList.size());

addMiddleCode(MiddleOperator.Goto);

}

SyntaxTree stat = tree.getChild(1);

generateStatement(stat);

m\_caseMapStack.pop();

m\_defaultStack.pop();

backpatch(m\_breakSetStack.pop());

}

public void generateCaseStatement(SyntaxTree tree) {

BigInteger caseValue = (BigInteger) tree.getSymbol().getValue();

int address = m\_caseMapStack.peek().get(caseValue);

backpatch(address);

SyntaxTree stat = tree.getChild(0);

generateStatement(stat);

}

public void generateDefaultStatement(SyntaxTree tree) {

int address = m\_defaultStack.peek();

backpatch(address);

SyntaxTree stat = tree.getChild(0);

generateStatement(stat);

}

public void generateBreakStatement(SyntaxTree tree) {

m\_breakSetStack.peek().add(m\_middleCodeList.size());

addMiddleCode(MiddleOperator.Goto);

}

public void generateContinueStatement(SyntaxTree tree) {

m\_continueSetStack.peek().add(m\_middleCodeList.size());

addMiddleCode(MiddleOperator.Goto);

}

public void generateWhileStatement(SyntaxTree tree) {

m\_breakSetStack.push(new ListSet<Integer>());

m\_continueSetStack.push(new ListSet<Integer>());

SyntaxTree expr = tree.getChild(0), stat = tree.getChild(1);

int whileLine = m\_middleCodeList.size();

Symbol symbol = generateExpression(expr);

backpatch(symbol.getType().getTrueSet());

generateStatement(stat);

addMiddleCode(MiddleOperator.Goto, whileLine);

backpatch(symbol.getType().getFalseSet());

backpatch(m\_continueSetStack.pop(), whileLine);

backpatch(m\_breakSetStack.pop());

}

public void generateDoStatement(SyntaxTree tree) {

m\_breakSetStack.push(new ListSet<Integer>());

m\_continueSetStack.push(new ListSet<Integer>());

SyntaxTree expr = tree.getChild(0), stat = tree.getChild(1);

int doLine = m\_middleCodeList.size();

generateStatement(stat);

Symbol symbol = generateExpression(expr);

backpatch(symbol.getType().getTrueSet(), doLine);

backpatch(symbol.getType().getFalseSet());

backpatch(m\_continueSetStack.pop(), doLine);

backpatch(m\_breakSetStack.pop());

}

public void generateForStatement(SyntaxTree tree) {

m\_breakSetStack.push(new ListSet<Integer>());

m\_continueSetStack.push(new ListSet<Integer>());

SyntaxTree initExpr = tree.getChild(0),

testExpr = tree.getChild(1),

nextExpr = tree.getChild(2),

stat = tree.getChild(3);

if (initExpr != null) {

Symbol initSymbol = generateExpression(initExpr);

backpatch(initSymbol.getType().getTrueSet());

backpatch(initSymbol.getType().getFalseSet());

}

int testLine = m\_middleCodeList.size();

Symbol testSymbol = null;

if (testExpr != null) {

testSymbol = generateExpression(testExpr);

backpatch(testSymbol.getType().getTrueSet());

}

generateStatement(stat);

if (nextExpr != null) {

Symbol nextSymbol = generateExpression(nextExpr);

backpatch(nextSymbol.getType().getTrueSet(), testLine);

backpatch(nextSymbol.getType().getFalseSet(), testLine);

}

addMiddleCode(MiddleOperator.Goto, testLine);

if (testSymbol != null) {

backpatch(testSymbol.getType().getFalseSet());

}

backpatch(m\_continueSetStack.pop(), testLine);

backpatch(m\_breakSetStack.pop());

}

public void generateLabelStatement(SyntaxTree tree) {

m\_labelMap.put((String) tree.getSymbol().getValue(),

m\_middleCodeList.size());

}

public void generateGotoStatement(SyntaxTree tree) {

String label = (String) tree.getSymbol().getValue();

if (!m\_gotoSetMap.containsKey(label)) {

m\_gotoSetMap.put(label, new ListSet<Integer>());

}

m\_gotoSetMap.get(label).add(m\_middleCodeList.size());

addMiddleCode(MiddleOperator.Goto);

}

public void generateGotoSet() {

for (Map.Entry<Integer,String> entry : Main.GotoMap.entrySet()) {

int gotoLine = entry.getKey();

String label = entry.getValue();

int labelLine = Main.LabelMap.get(label);

backpatch(gotoLine, labelLine);

}

Main.LabelMap.clear();

Main.GotoMap.clear();

}

public void backpatchGoto() {

for (Map.Entry<String,Set<Integer>> entry : m\_gotoSetMap.entrySet()) {

String label = entry.getKey();

Set<Integer> gotoSet = entry.getValue();

int labelLine = m\_labelMap.get(label);

backpatch(gotoSet, labelLine);

}

}

public void generateReturnStatement(SyntaxTree tree) {

SyntaxTree expr = tree.getChild(0);

if (expr != null) {

if (expr.getType().isFloating()) {

Symbol checkSymbol = Symbol.createSolidSymbol(Type.SignedCharType);

checkSymbol.setOffset(SymbolTable.UseResultOffset);

Symbol zeroSymbol =

Symbol.createValueSymbol(Type.SignedCharType, BigInteger.ZERO);

int checkIndex = m\_middleCodeList.size();

addMiddleCode(MiddleOperator.Equal, null, checkSymbol, zeroSymbol);

Symbol symbol = generateExpression(expr);

addMiddleCode(MiddleOperator.SetReturnValue, null, symbol);

int gotoIndex = m\_middleCodeList.size();

addMiddleCode(MiddleOperator.Goto);

m\_middleCodeList.get(checkIndex).setOperand(1, gotoIndex + 1);

generateExpression(SyntaxTreeOptimizer.clearExpression(expr));

m\_middleCodeList.get(gotoIndex).setOperand(1, m\_middleCodeList.size());

}

else {

Symbol symbol = generateExpression(expr);

addMiddleCode(MiddleOperator.SetReturnValue, null, symbol);

}

}

addMiddleCode(MiddleOperator.Return);

}

public void generateExitStatement(SyntaxTree tree) {

SyntaxTree expr = tree.getChild(0);

Symbol symbol = (expr != null) ? generateExpression(expr) : null;

addMiddleCode(MiddleOperator.Exit, symbol);

}

public void generateLoadToRegisterStatement(SyntaxTree tree) {

Register register = (Register) tree.getInfo();

SyntaxTree expr = tree.getChild(0);

Symbol symbol = generateExpression(expr);

addMiddleCode(MiddleOperator.LoadToRegister, register, symbol);

}

public void generateSaveFromRegisterStatement(SyntaxTree tree) {

Register register = (Register) tree.getInfo();

SyntaxTree expr = tree.getChild(0);

Symbol symbol = generateExpression(expr);

addMiddleCode(MiddleOperator.SaveFromRegister, symbol, register);

}

public void generateClearRegistersStatement(SyntaxTree tree) {

addMiddleCode(MiddleOperator.ClearRegisters, null, null, null);

}

public void generateSaveFlagByteRegisterStatement(SyntaxTree tree) {

SyntaxTree expr = tree.getChild(0);

Symbol symbol = generateExpression(expr);

addMiddleCode(MiddleOperator.SaveFromFlagByte, symbol);

}

public void generateJumpRegisterStatement(SyntaxTree tree) {

addMiddleCode(MiddleOperator.JumpRegister, (Register) tree.getInfo());

}

public void generateInterruptStatement(SyntaxTree tree) {

addMiddleCode(MiddleOperator.Interrupt, tree.getSymbol().getValue());

}

public void generateDeclarationStatement(SyntaxTree tree) {

appendMiddleCodeList(m\_middleCodeList, (List<MiddleCode>) tree.getInfo());

}

public Symbol generateExpression(SyntaxTree tree) {

tree = SyntaxTreeOptimizer.optimizeExpression(tree);

if (tree == null) {

return null;

}

switch (tree.getOperator()) {

case Comma:

return generateCommaExpression(tree);

case Sequence:

return generateSequencialExpression(tree);

case Conditional:

return generateConditionalExpression(tree);

case LogicalAnd:

return generateLogicalAndExpression(tree);

case LogicalOr:

return generateLogicalOrExpression(tree);

case LogicalNot:

return generateLogicalNotExpression(tree);

case UnaryAdd:

case UnarySubtract:

case BitwiseNot:

case Address:

return generateUnaryExpression(tree);

case IntegralToIntegral:

return generateIntegralToIntegralExpression(tree);

case FloatingToFloating:

return generateFloatingToFloatingExpression(tree);

case IntegralToFloating:

return generateIntegralToFloatingExpression(tree);

case FloatingToIntegral:

return generateFloatingToIntegralExpression(tree);

case LogicalToIntegral:

return generateLogicalToIntegralExpression(tree);

case ArithmeticToLogical:

return generateArithmeticToLogicalExpression(tree);

case FunctionToPointer:

return generateFunctionToPointerExpression(tree);

case ArrayToPointer:

return generateArrayToPointerExpression(tree);

case StringToPointer:

return generateStringToPointerExpression(tree);

case BinaryAdd:

case BinarySubtract:

case Multiply:

case Divide:

case Modulo:

case BitwiseAnd:

case BitwiseIOr:

case BitwiseXOr:

case LeftShift:

case RightShift:

return generateArithmeticExpression(tree);

case Equal:

case NotEqual:

case LessThan:

case LessThanEqual:

case GreaterThan:

case GreaterThanEqual:

return generateRelationalExpression(tree);

case Deref:

return generateDerefExpression(tree);

case Index:

return generateIndexExpression(tree);

case Dot:

return generateDotExpression(tree);

case Arrow:

return generateArrowExpression(tree);

case Assign:

return generateSimpleAssignExpression(tree);

case BinaryAddAssign:

case BinarySubtractAssign:

case MultiplyAssign:

case DivideAssign:

case ModuloAssign:

case BitwiseAndAssign:

case BitwiseIOrAssign:

case BitwiseXOrAssign:

case LeftShiftAssign:

case RightShiftAssign:

return generateCompoundAssignExpression(tree);

case PrefixIncrement:

case PrefixDecrement:

return generatePrefixIncrementExpression(tree);

case PostfixIncrement:

case PostfixDecrement:

return generatePostfixIncrementExpression(tree);

case Call:

return generateFunctionCallExpression(tree);

case Variable:

return generateVariableExpression(tree);

case Value:

return generateValueExpression(tree);

default:

Assert.error("generate expression code " + tree.getOperator().name());

return null;

}

}

public Symbol generateCommaExpression(SyntaxTree tree) {

generateExpression(tree.getChild(0));

return generateExpression(tree.getChild(1));

}

public Symbol generateSequencialExpression(SyntaxTree tree) {

generateExpression(tree.getChild(0));

generateExpression(tree.getChild(1));

return null;

}

public Symbol generateIntegralToIntegralExpression(SyntaxTree tree) {

Symbol symbol = generateExpression(tree.getChild(0));

Symbol resultSymbol = Symbol.createTemporarySymbol(tree.getType());

addMiddleCode(MiddleOperator.IntegralToIntegral,

null, resultSymbol, symbol);

return resultSymbol;

}

public Symbol generateFloatingToFloatingExpression(SyntaxTree tree) {

return generateExpression(tree.getChild(0));

}

public Symbol generateIntegralToFloatingExpression(SyntaxTree tree) {

Symbol symbol = generateExpression(tree.getChild(0));

if (symbol.isTemporary() && (symbol.getOffset() == 0)) {

//symbol.addToSymbolTable();

Main.CurrentTable.addSymbol(symbol);

}

addMiddleCode(MiddleOperator.PushFloat, symbol);

return Symbol.createTemporarySymbol(tree.getType());

}

public Symbol generateFloatingToIntegralExpression(SyntaxTree tree) {

generateExpression(tree.getChild(0));

Symbol resultSymbol = Symbol.createTemporarySymbol(tree.getType());

//resultSymbol.addToSymbolTable();

Main.CurrentTable.addSymbol(resultSymbol);

boolean pop = !useResult(tree);

addMiddleCode(MiddleOperator.FloatingToIntegral, pop, resultSymbol, null);

return resultSymbol;

}

private static Symbol ZeroIntegerSymbol =

Symbol.createValueSymbol(Type.SignedIntegerType, BigInteger.ZERO);

private static Symbol ZeroDoubleSymbol =

Symbol.createValueSymbol(Type.DoubleType, BigDecimal.ZERO);

private static Symbol OneIntegerSymbol =

Symbol.createValueSymbol(Type.SignedIntegerType, BigInteger.ONE);

public Symbol generateLogicalToIntegralExpression(SyntaxTree tree) {

Symbol symbol = generateExpression(tree.getChild(0));

Symbol resultSymbol = Symbol.createSolidSymbol(tree.getType());

//resultSymbol.addToSymbolTable();

Main.CurrentTable.addSymbol(resultSymbol);

backpatch(symbol.getType().getTrueSet());

addMiddleCode(MiddleOperator.Assign, null, resultSymbol, OneIntegerSymbol);

addMiddleCode(MiddleOperator.Goto, m\_middleCodeList.size() + 2, null,null);

backpatch(symbol.getType().getFalseSet());

addMiddleCode(MiddleOperator.Assign, null, resultSymbol,ZeroIntegerSymbol);

return resultSymbol;

}

public Symbol generateArithmeticToLogicalExpression(SyntaxTree tree) {

Symbol symbol = generateExpression(tree.getChild(0));

Set<Integer> trueSet = new ListSet(m\_middleCodeList.size());

Symbol zeroSymbol = symbol.getType().isIntegralOrPointer() ?

ZeroIntegerSymbol : ZeroDoubleSymbol;

addMiddleCode(MiddleOperator.NotEqual, null, symbol, zeroSymbol);

Set<Integer> falseSet = new ListSet(m\_middleCodeList.size());

addMiddleCode(MiddleOperator.Goto);

return Symbol.createLogicalSymbol(trueSet, falseSet);

}

public Symbol generateFunctionToPointerExpression(SyntaxTree tree) {

Symbol symbol = generateExpression(tree.getChild(0));

return symbol;

}

public Symbol generateArrayToPointerExpression(SyntaxTree tree) {

Symbol symbol = generateExpression(tree.getChild(0));

return symbol;

}

public Symbol generateStringToPointerExpression(SyntaxTree tree) {

Symbol symbol = generateExpression(tree.getChild(0));

return symbol;

}

public Symbol generateConditionalExpression(SyntaxTree tree) {

Symbol testSymbol = generateExpression(tree.getChild(0));

backpatch(testSymbol.getType().getTrueSet());

boolean pop = !useResult(tree);

Symbol trueSymbol = generateExpression(tree.getChild(1));

Symbol resultSymbol = Symbol.createSolidSymbol(tree.getType());

//resultSymbol.addToSymbolTable();

Main.CurrentTable.addSymbol(resultSymbol);

if (!resultSymbol.getType().isFloating()) {

addMiddleCode(MiddleOperator.Assign, pop, resultSymbol, trueSymbol);

}

int gotoLineNo = m\_middleCodeList.size();

addMiddleCode(MiddleOperator.Goto);

backpatch(testSymbol.getType().getFalseSet());

Symbol falseSymbol = generateExpression(tree.getChild(2));

if (!resultSymbol.getType().isFloating()) {

addMiddleCode(MiddleOperator.Assign, pop, resultSymbol, falseSymbol);

}

backpatch(gotoLineNo);

return resultSymbol;

}

public Symbol generateLogicalAndExpression(SyntaxTree tree) {

Symbol leftSymbol = generateExpression(tree.getChild(0));

backpatch(leftSymbol.getType().getTrueSet());

Symbol rightSymbol = generateExpression(tree.getChild(1));

Set<Integer> falseSet = leftSymbol.getType().getFalseSet();

falseSet.addAll(rightSymbol.getType().getFalseSet());

return Symbol.createLogicalSymbol(rightSymbol.getType().getTrueSet(),

falseSet);

}

public Symbol generateLogicalOrExpression(SyntaxTree tree) {

Symbol leftSymbol = generateExpression(tree.getChild(0));

backpatch(leftSymbol.getType().getFalseSet());

Symbol rightSymbol = generateExpression(tree.getChild(1));

Set<Integer> trueSet = leftSymbol.getType().getTrueSet();

trueSet.addAll(rightSymbol.getType().getTrueSet());

return Symbol.createLogicalSymbol(trueSet,

rightSymbol.getType().getFalseSet());

}

public Symbol generateLogicalNotExpression(SyntaxTree tree) {

Symbol symbol = generateExpression(tree.getChild(0));

return Symbol.createLogicalSymbol(symbol.getType().getFalseSet(),

symbol.getType().getTrueSet());

}

public Symbol generateRelationalExpression(SyntaxTree tree) {

Symbol leftSymbol = generateExpression(tree.getChild(0));

Symbol rightSymbol = generateExpression(tree.getChild(1));

Set<Integer> trueSet = new ListSet(m\_middleCodeList.size());

addMiddleCode(tree.getOperator(), null, leftSymbol, rightSymbol);

Set<Integer> falseSet = new ListSet(m\_middleCodeList.size());

addMiddleCode(MiddleOperator.Goto);

return Symbol.createLogicalSymbol(trueSet, falseSet);

}

public Symbol generateArithmeticExpression(SyntaxTree tree) {

Symbol leftSymbol = generateExpression(tree.getChild(0)),

rightSymbol = generateExpression(tree.getChild(1));

Type leftType = leftSymbol.getType(), rightType = rightSymbol.getType();

if (leftType.isPointerOrArray() && rightType.isPointerOrArray() &&

(leftType.getPointerOrArrayType().getSize() > 1)) {

Type pointerType = Type.createPointerType(leftType.getPointerOrArrayType());

Symbol tempSymbol = Symbol.createTemporarySymbol(pointerType);

addMiddleCode(tree.getOperator(), tempSymbol, leftSymbol, rightSymbol);

int typeSize = leftType.getPointerOrArrayType().getSize();

Symbol sizeSymbol = Symbol.createValueSymbol(Type.SignedIntegerType,

BigInteger.valueOf(typeSize));

Symbol resultSymbol = Symbol.createTemporarySymbol(pointerType);

addMiddleCode(MiddleOperator.Divide, resultSymbol,

tempSymbol, sizeSymbol);

return resultSymbol;

}

else if (leftType.isPointerOrArray() && rightType.isIntegral() &&

(leftType.getPointerOrArrayType().getSize() > 1)) {

Type pointerType =

Type.createPointerType(leftType.getPointerOrArrayType());

Symbol tempSymbol = Symbol.createTemporarySymbol(pointerType);

int typeSize = leftType.getPointerOrArrayType().getSize();

Symbol sizeSymbol = Symbol.createValueSymbol(Type.SignedIntegerType,

BigInteger.valueOf(typeSize));

addMiddleCode(MiddleOperator.Multiply, tempSymbol,

rightSymbol, sizeSymbol);

Symbol resultSymbol = Symbol.createTemporarySymbol(pointerType);

addMiddleCode(tree.getOperator(), resultSymbol, leftSymbol, tempSymbol);

return resultSymbol;

}

else if (leftType.isIntegral() && rightType.isPointerOrArray() &&

(rightType.getPointerOrArrayType().getSize() > 1)) {

Type pointerType =

Type.createPointerType(leftType.getPointerOrArrayType());

Symbol tempSymbol = Symbol.createTemporarySymbol(pointerType);

int typeSize = rightType.getPointerType().getSize();

Symbol sizeSymbol = Symbol.createValueSymbol(Type.SignedIntegerType,

BigInteger.valueOf(typeSize));

addMiddleCode(MiddleOperator.Multiply, tempSymbol,

leftSymbol, sizeSymbol);

Symbol resultSymbol = Symbol.createTemporarySymbol(pointerType);

addMiddleCode(tree.getOperator(), resultSymbol, tempSymbol, rightSymbol);

return resultSymbol;

}

else {

Symbol resultSymbol = Symbol.createTemporarySymbol(tree.getType());

addMiddleCode(tree.getOperator(), resultSymbol, leftSymbol, rightSymbol);

return resultSymbol;

}

}

// ------------------------------------------------------------------------

public Symbol generateUnaryExpression(SyntaxTree tree){

Symbol unarySymbol = generateExpression(tree.getChild(0));

Assert.error(unarySymbol.getStorage() != Storage.Register,

unarySymbol, "address of register storage not allowed");

Symbol resultSymbol = Symbol.createTemporarySymbol(tree.getType());

addMiddleCode(tree.getOperator(), resultSymbol, unarySymbol);

return resultSymbol;

}

// x->y

// (\*x).y

public Symbol generateDerefExpression(SyntaxTree tree) {

Symbol derefSymbol = generateExpression(tree.getChild(0));

Type baseType = derefSymbol.getType().getPointerOrArrayType();

Symbol resultSymbol = Symbol.createTemporarySymbol(baseType);

resultSymbol.setAddressSymbol(derefSymbol);

resultSymbol.setAddressOffset(0);

MiddleCode derefCode =

addMiddleCode(MiddleOperator.Deref, 0, resultSymbol, derefSymbol);

derefCode.setRightValue(tree.isRightValue());

return resultSymbol;

}

public Symbol generateArrowExpression(SyntaxTree tree) {

Symbol pointerSymbol = generateExpression(tree.getChild(0));

Symbol memberSymbol = generateExpression(tree.getChild(1));

Symbol resultSymbol = Symbol.createTemporarySymbol(tree.getType());

resultSymbol.setAddressSymbol(pointerSymbol);

int memberOffset = memberSymbol.getOffset();

resultSymbol.setAddressOffset(memberOffset);

resultSymbol.setAddressSymbol(pointerSymbol);

MiddleCode arrowCode =

addMiddleCode(MiddleOperator.Arrow, memberOffset, resultSymbol, pointerSymbol);

arrowCode.setRightValue(tree.isRightValue());

return resultSymbol;

}

public Symbol generatePrefixIncrementExpression(SyntaxTree tree) {

Symbol incSymbol = generateExpression(tree.getChild(0));

boolean use = useResult(tree);

addMiddleCode(tree.getOperator(), !use, null, incSymbol);

BigInteger bitFieldMask = incSymbol.getType().getBitfieldMask();

if (bitFieldMask != null) {

Symbol maskSymbol =

Symbol.createValueSymbol(incSymbol.getType(), bitFieldMask);

addMiddleCode(MiddleOperator.BitwiseAnd, incSymbol,

incSymbol, maskSymbol);

}

return incSymbol;

}

public Symbol generatePostfixIncrementExpression(SyntaxTree tree) {

Symbol incSymbol = generateExpression(tree.getChild(0)),

resultSymbol;

if (useResult(tree)) {

if (incSymbol.getType().isFloating()) {

resultSymbol = Symbol.createSolidSymbol(incSymbol.getType());

//resultSymbol.addToSymbolTable();

Main.CurrentTable.addSymbol(resultSymbol);

addMiddleCode(tree.getOperator(), false, resultSymbol, incSymbol);

}

else {

resultSymbol = Symbol.createTemporarySymbol(incSymbol.getType());

addMiddleCode(tree.getOperator(), false, resultSymbol, incSymbol);

}

}

else {

resultSymbol = null;

MiddleOperator operator = MiddleOperator.valueOf

(tree.getOperator().name().replace("Postfix", "Prefix"));

addMiddleCode(operator, true, incSymbol, incSymbol);

}

BigInteger bitFieldMask = incSymbol.getType().getBitfieldMask();

if (bitFieldMask != null) {

Symbol maskSymbol =

Symbol.createValueSymbol(incSymbol.getType(), bitFieldMask);

addMiddleCode(MiddleOperator.BitwiseAnd, incSymbol,

incSymbol, maskSymbol);

}

return resultSymbol;

}

// push pop => nothing

// pop push => top

public Symbol generateSimpleAssignExpression(SyntaxTree tree) {

boolean pop = !useResult(tree);

SyntaxTree leftExpr = tree.getChild(0),

rightExpr = tree.getChild(1);

Symbol leftSymbol = generateExpression(leftExpr);

if (rightExpr.getOperator() == MiddleOperator.FloatingToIntegral) {

SyntaxTree floatingExpr = rightExpr.getChild(0);

generateExpression(floatingExpr);

addMiddleCode(MiddleOperator.FloatingToIntegral, pop, leftSymbol, null);

}

else {

Symbol rightSymbol = generateExpression(rightExpr);

addMiddleCode(MiddleOperator.Assign, pop, leftSymbol, rightSymbol);

BigInteger bitFieldMask = leftSymbol.getType().getBitfieldMask();

if (bitFieldMask != null) {

Symbol maskSymbol =

Symbol.createValueSymbol(leftSymbol.getType(), bitFieldMask);

addMiddleCode(MiddleOperator.BitwiseAnd, leftSymbol,

leftSymbol, maskSymbol);

}

}

return leftSymbol;

}

private final Map<MiddleOperator,MiddleOperator> m\_compoundMap =

new ListMap<>();

{ m\_compoundMap.put(MiddleOperator.BinaryAddAssign,

MiddleOperator.BinaryAdd);

m\_compoundMap.put(MiddleOperator.BinarySubtractAssign,

MiddleOperator.BinarySubtract);

m\_compoundMap.put(MiddleOperator.MultiplyAssign, MiddleOperator.Multiply);

m\_compoundMap.put(MiddleOperator.DivideAssign, MiddleOperator.Divide);

m\_compoundMap.put(MiddleOperator.ModuloAssign, MiddleOperator.Modulo);

m\_compoundMap.put(MiddleOperator.BitwiseAndAssign,

MiddleOperator.BitwiseAnd);

m\_compoundMap.put(MiddleOperator.BitwiseIOrAssign,

MiddleOperator.BitwiseIOr);

m\_compoundMap.put(MiddleOperator.BitwiseXOrAssign,

MiddleOperator.BitwiseXOr);

m\_compoundMap.put(MiddleOperator.LeftShiftAssign,

MiddleOperator.LeftShift);

m\_compoundMap.put(MiddleOperator.RightShiftAssign,

MiddleOperator.RightShift);

}

public Symbol generateCompoundAssignExpression(SyntaxTree tree) {

Symbol leftSymbol = generateExpression(tree.getChild(0));

Symbol rightSymbol = generateExpression(tree.getChild(1));

Symbol tempSymbol = Symbol.createTemporarySymbol(tree.getType());

String operatorName = tree.getOperator().name().replace("Assign", "");

MiddleOperator operator = MiddleOperator.valueOf(operatorName);

addMiddleCode(operator, tempSymbol, leftSymbol, rightSymbol);

boolean pop = !useResult(tree);

addMiddleCode(MiddleOperator.Assign, pop, leftSymbol, tempSymbol);

BigInteger bitFieldMask = leftSymbol.getType().getBitfieldMask();

if (bitFieldMask != null) {

Symbol maskSymbol =

Symbol.createValueSymbol(leftSymbol.getType(), bitFieldMask);

addMiddleCode(MiddleOperator.BitwiseAnd, leftSymbol,

leftSymbol, maskSymbol);

}

return leftSymbol;

}

public Symbol generateDotExpression(SyntaxTree tree) {

Symbol parentSymbol = generateExpression(tree.getChild(0));

Symbol memberSymbol = generateExpression(tree.getChild(1)),

resultSymbol;

if (parentSymbol.hasAddressSymbol()) {

resultSymbol = Symbol.createMemorySymbol(parentSymbol, tree.getType(),

memberSymbol);

}

else {

resultSymbol = Symbol.createTemporarySymbol(tree.getType());

resultSymbol.setOffset(parentSymbol.getOffset() +

memberSymbol.getOffset());

//resultSymbol.setAddressSymbol(parentSymbol.getAddressSymbol());

//resultSymbol.setAddressOffset(parentSymbol.getAddressOffset() + memberSymbol.getOffset());

}

resultSymbol.setParentSymbol(parentSymbol);

parentSymbol.addChildSymbol(resultSymbol);

if (tree.isRightValue() && resultSymbol.getType().isFloating()) {

if (resultSymbol.isTemporary() && (resultSymbol.getOffset() == 0)) {

//symbol.addToSymbolTable();

Main.CurrentTable.addSymbol(resultSymbol);

}

addMiddleCode(MiddleOperator.PushFloat, resultSymbol);

}

return resultSymbol;

}

public Symbol generateIndexExpression(SyntaxTree tree) {

Symbol pointerSymbol = generateExpression(tree.getChild(0));

SyntaxTree indexTree = tree.getChild(1);

Symbol indexSymbol = generateExpression(indexTree);

int indexSize = pointerSymbol.getType().getPointerOrArrayType().getSize();

Symbol resultSymbol;

if (indexTree.getOperator() == MiddleOperator.Value) {

resultSymbol = Symbol.createTemporarySymbol(tree.getType());

resultSymbol.setAddressSymbol(pointerSymbol);

resultSymbol.setParentSymbol(pointerSymbol);

pointerSymbol.addChildSymbol(resultSymbol);

int indexValue =

((BigInteger) indexTree.getSymbol().getValue()).intValue();

int memberOffset = indexValue \* indexSize;

resultSymbol.setAddressOffset(memberOffset);

MiddleCode indexCode =

addMiddleCode(MiddleOperator.Index, memberOffset, resultSymbol, pointerSymbol);

indexCode.setRightValue(tree.isRightValue());

}

else if (indexSize == 1) {

Symbol addSymbol = Symbol.createTemporarySymbol(indexSymbol.getType());

addMiddleCode(MiddleOperator.BinaryAdd, addSymbol,

pointerSymbol, indexSymbol);

resultSymbol = Symbol.createTemporarySymbol(pointerSymbol.getType().getPointerOrArrayType());

resultSymbol.setParentSymbol(pointerSymbol);

pointerSymbol.addChildSymbol(resultSymbol);

resultSymbol.setAddressSymbol(addSymbol);

resultSymbol.setAddressOffset(0);

MiddleCode indexCode =

addMiddleCode(MiddleOperator.Index, 0, resultSymbol, addSymbol);

indexCode.setRightValue(tree.isRightValue());

}

else {

Symbol sizeSymbol = Symbol.createValueSymbol(indexSymbol.getType(), BigInteger.valueOf(indexSize));

Symbol mulSymbol = Symbol.createTemporarySymbol(indexSymbol.getType());

addMiddleCode(MiddleOperator.Multiply, mulSymbol, indexSymbol, sizeSymbol);

Symbol addSymbol = Symbol.createTemporarySymbol(indexSymbol.getType());

addMiddleCode(MiddleOperator.BinaryAdd, addSymbol, pointerSymbol, mulSymbol);

resultSymbol = Symbol.createTemporarySymbol(pointerSymbol.getType().getPointerOrArrayType());

resultSymbol.setParentSymbol(pointerSymbol);

pointerSymbol.addChildSymbol(resultSymbol);

resultSymbol.setAddressSymbol(addSymbol);

resultSymbol.setAddressOffset(0);

MiddleCode indexCode =

addMiddleCode(MiddleOperator.Index, 0, resultSymbol, addSymbol);

indexCode.setRightValue(tree.isRightValue());

}

return resultSymbol;

}

boolean useResult(SyntaxTree tree) {

SyntaxTree parentTree = tree.getParent();

if (parentTree == null) {

return false;

}

else {

switch (parentTree.getOperator()) {

case For:

return (tree == parentTree.getChild(0)) ||

(tree == parentTree.getChild(2));

case Expression:

case Sequence:

return false;

default:

return true;

}

}

}

public Symbol generateFunctionCallExpression(SyntaxTree tree) {

Symbol calleeSymbol = generateExpression(tree.getChild(0));

Type returnType = calleeSymbol.getType().isFunction() ?

calleeSymbol.getType().getReturnType() :

calleeSymbol.getType().getPointerType().getReturnType();

int prevTotalOffset = Main.TotalParamOffset;

int initSize = returnType.isFloating()?SymbolTable.FunctionFloatingInitSize

:SymbolTable.FunctionRegularInitSize;

Main.TotalParamOffset += initSize;

for (int index = 1; index < tree.getChildList().size(); ++index) {

Symbol paramSymbol = generateExpression(tree.getChild(index));

addMiddleCode(MiddleOperator.Parameter, Main.TotalParamOffset,

paramSymbol, null);

int paramSize = paramSymbol.getType().isFunctionArrayOrString() ?

Type.PointerSize : paramSymbol.getType().getSize();

Main.TotalParamOffset += paramSize;

}

Main.TotalParamOffset = prevTotalOffset;

int extra = (Integer) tree.getInfo();

Pair<Integer,Boolean> pair = new Pair<>(extra, useResult(tree));

addMiddleCode(MiddleOperator.Call, calleeSymbol,

pair, Main.TotalParamOffset);

if (!returnType.isVoid()) {

if (useResult(tree)) {

SyntaxTree returnTree = tree.getReturnTree();

if (returnTree != null) {

Symbol returnSymbol = generateExpression(returnTree);

addMiddleCode(MiddleOperator.GetReturnValue,

returnSymbol, null, null);

return returnSymbol;

}

else {

Symbol returnSymbol = Symbol.createTemporarySymbol(tree.getType());

if (tree.getType().isStructOrUnion()) {

Main.CurrentTable.addSymbol(returnSymbol);

}

addMiddleCode(MiddleOperator.GetReturnValue,

returnSymbol, null, null);

return returnSymbol;

}

}

}

return null;

}

public Symbol generateVariableExpression(SyntaxTree tree) {

Symbol variableSymbol = tree.getSymbol();

if (tree.isRightValue() && variableSymbol.getType().isFloating()) {

if (variableSymbol.isTemporary() && (variableSymbol.getOffset() == 0)) {

//symbol.addToSymbolTable();

Main.CurrentTable.addSymbol(variableSymbol);

}

addMiddleCode(MiddleOperator.PushFloat, variableSymbol);

}

return variableSymbol;

}

public Symbol generateValueExpression(SyntaxTree tree) {

if (tree.getType().isFloating()) {

Assert.error(tree.getSymbol().isStaticOrExtern());

//tree.getSymbol().addToSymbolTable();

Main.CurrentTable.addSymbol(tree.getSymbol());

addMiddleCode(MiddleOperator.PushFloat, tree.getSymbol());

}

return tree.getSymbol();

}

}

# Initializer

In C, it is possible to initialize simple and compound variables. Therefore, we need to check that the initialized value has the correct type. Basically, there are two kinds of initialization: static and stack. Static initialization results in a block of bytes, and stack initialization results in a sequence of assignment middle code instructions.

### Static Initialization

Static initialization occurs when a static variable becomes initialized. The initialization value has to be constant and known at compile time. No code is generated, instead a memory block holding the value is created. There are several cases to consider:

|  |  |  |
| --- | --- | --- |
| **Variable** | **Initializer** | **Example** |
| Pointer | Address | static int array[3];  static int\* p = &a[2]; |
| Pointer to signed or unsigned char | String | static char \*p = “Hello”; |
| Pointer | List | int \*p = {1,2,3}; |
| Array | String | char s[] = “World”; |
| Array | List | Char st[] = {‘a’, ’b’, ‘c’}; |
| Struct | List |  |
| Union | List |  |
| Integral or Pointer |  |  |
| Floating |  |  |

GenerateInitializer.java

If the variable is a pointer and the initializer is an address we add its offset to the block and store the name of the address in the access map, the address value will later be looked up and added by the linker.

If the variable type is a pointer (signed or unsigned) char and the initializatior is a string, the address is stored in the access map and a zero address is stored in the block, it will later be properly looked up and set by the linker.

GenerateInitializer.java

package c\_compiler;

import java.nio.\*;

import java.math.\*;

import java.util.\*;

public class GenerateInitializerStatic {

public static Object generateValueExpression(Object init) {

if (init instanceof SyntaxTree) {

return SyntaxTreeOptimizer.optimizeExpression((SyntaxTree) init);

}

else {

List list = (List) init;

for (int index = 0; index < list.size(); ++index) {

list.set(index, generateValueExpression(list.get(index)));

}

return list;

}

}

public static Object generateAddressExpression(Object init) {

if (init instanceof SyntaxTree) {

SyntaxTree tree = (SyntaxTree) init;

switch (tree.getOperator()) {

case Address: {

SyntaxTree childTree = tree.getChildList().get(0);

switch (childTree.getOperator()) {

case Variable:

return (new SyntaxTree(MiddleOperator.AddressOffset,

tree.getType(), childTree.getSymbol(), 0));

case Index: {

SyntaxTree arrayTree = childTree.getChildList().get(0),

indexTree = childTree.getChildList().get(1);

if ((arrayTree.getOperator() == MiddleOperator.Variable) &&

(indexTree.getOperator() == MiddleOperator.Value)) {

int index = ((BigInteger)

indexTree.getSymbol().getValue()).intValue(),

indexSize =

arrayTree.getType().getPointerOrArrayType().getSize();

int offset = index \* indexSize;

return (new SyntaxTree(MiddleOperator.AddressOffset,

tree.getType(), arrayTree.getSymbol(), offset));

}

}

break;

}

}

break;

case BinaryAdd: {

SyntaxTree leftTree = tree.getChildList().get(0),

rightTree = tree.getChildList().get(1);

if (leftTree.getType().isPointerOrArray() &&

(leftTree.getOperator() == MiddleOperator.Variable) &&

(rightTree.getOperator() == MiddleOperator.Value)) {

int arraySize =

leftTree.getType().getPointerOrArrayType().getSize();

int offset =

((Integer) rightTree.getSymbol().getValue()) \* arraySize;

MiddleOperator operator = leftTree.getType().isPointer() ?

MiddleOperator.ValueOffset : MiddleOperator.AddressOffset;

return (new SyntaxTree(operator, tree.getType(),

leftTree.getSymbol(), offset));

}

else if ((leftTree.getOperator() == MiddleOperator.Value) &&

rightTree.getType().isPointerOrArray() &&

(rightTree.getOperator() == MiddleOperator.Variable)) {

int arraySize =

rightTree.getType().getPointerOrArrayType().getSize();

int offset = ((Integer)

leftTree.getSymbol().getValue()) \* arraySize;

MiddleOperator operator = rightTree.getType().isPointer() ?

MiddleOperator.ValueOffset : MiddleOperator.AddressOffset;

return (new SyntaxTree(operator, tree.getType(),

rightTree.getSymbol(), offset));

}

}

break;

case BinarySubtract: {

SyntaxTree leftTree = tree.getChildList().get(0),

rightTree = tree.getChildList().get(1);

if (leftTree.getType().isPointerOrArray() &&

(leftTree.getOperator() == MiddleOperator.Variable) &&

(rightTree.getOperator() == MiddleOperator.Value)) {

int arraySize = leftTree.getType().getPointerOrArrayType().getSize();

int offset = ((Integer) rightTree.getSymbol().getValue()) \* arraySize;

MiddleOperator operator = leftTree.getType().isPointer() ?

MiddleOperator.ValueOffset : MiddleOperator.AddressOffset;

return (new SyntaxTree(operator, tree.getType(),

leftTree.getSymbol(), -offset));

}

}

break;

}

return tree;

}

else {

List list = (List) init;

for (int index = 0; index < list.size(); ++index) {

list.set(index, generateAddressExpression(list.get(index)));

}

return list;

}

}

public static void generateByteTextList(Type type, Object value,

List<Byte> byteList,Map<Integer,String> accessMap,

List<String> textList) {

if (value instanceof String) {

String text = (String) value;

StringBuilder buffer = new StringBuilder();

boolean isText = false;

for (int index = 0; index < text.length(); ++index) {

char c = text.charAt(index);

int asciiValue = (int) c;

byteList.add((byte) asciiValue);

if ((c != '\"') && (c != '\'') &&

(asciiValue >= 32) && (asciiValue <= 126)) {

buffer.append((!isText ? ((((index == 0) ? "" : ", ")) + "\"") : "")+

((char) asciiValue));

isText = true;

}

else {

buffer.append((isText ? "\"" : "") + ((index == 0) ? "" : ", ") +

asciiValue);

isText = false;

}

}

byteList.add((byte) 0);

buffer.append((isText ? "\"" : "") + (!text.isEmpty() ? ", 0" : "0"));

textList.add("\tdb " + buffer.toString() + "\t; " + (text.length() + 1));

}

else {

switch (type.getSort()) {

case Signed\_Short\_Int:

case Unsigned\_Short\_Int: {

byte byteValue = ((BigInteger) value).byteValue();

byteList.add(byteValue);

textList.add("\tdb " + byteValue + "\t; " + type.getSize());

}

break;

case Signed\_Char:

case Unsigned\_Char: {

int asciiValue = (int) ((BigInteger) value).byteValue();

byteList.add((byte) asciiValue);

if ((asciiValue >= 32) && (asciiValue <= 126)) {

textList.add("\tdb '" + ((char) asciiValue) +

"'\t; " + type.getSize());

}

else {

textList.add("\tdb " + asciiValue + "\t; " + type.getSize());

}

}

break;

case Signed\_Int:

case Unsigned\_Int:

case Pointer: {

Assert.error(value instanceof BigInteger, "value");

int intValue = ((BigInteger) value).intValue();

byteList.add((byte) ((short) intValue));

byteList.add((byte) (((short) intValue) >> 8));

textList.add("\tdw " + intValue + "\t; " + type.getSize());

}

break;

case Signed\_Long\_Int:

case Unsigned\_Long\_Int: {

int intValue = ((BigInteger) value).intValue();

byteList.add((byte) intValue);

byteList.add((byte) (intValue >> 8));

byteList.add((byte) (intValue >> 16));

byteList.add((byte) (intValue >> 24));

textList.add("\tdd " + intValue + "\t; " + type.getSize());

}

break;

case Float: {

ByteBuffer byteBuffer = ByteBuffer.allocate(type.getSize());

float floatValue = ((BigDecimal) value).floatValue();

byteBuffer.putFloat(floatValue); // 4 bytes

byte[] byteArray = byteBuffer.array();

for (int index = (byteArray.length - 1); index >= 0; --index) {

byteList.add(byteArray[index]);

}

String text = ((BigDecimal) value).toPlainString();

String suffix = text.contains(".") ? "" : ".0";

textList.add("\tdd " + text + suffix + "\t; " + type.getSize());

}

break;

case Double:

case Long\_Double: {

ByteBuffer byteBuffer = ByteBuffer.allocate(type.getSize());

double doubleValue = ((BigDecimal) value).doubleValue();

byteBuffer.putDouble(doubleValue); // 8 bytes

byte[] byteArray = byteBuffer.array();

for (int index = (byteArray.length - 1); index >= 0; --index) {

byteList.add(byteArray[index]);

}

String text = ((BigDecimal) value).toPlainString();

String suffix = text.contains(".") ? "" : ".0";

textList.add("\tdq " + text + suffix+ "\t; " + type.getSize());

}

break;

}

}

}

public static void generateZeroByteTextList(int size, List<Byte> byteList,

List<String> textList) {

if (size > 0) {

StringBuilder buffer = new StringBuilder();

for (int index = 0; index < size; ++index) {

byteList.add((byte) 0);

buffer.append((index == 0) ? "0" : ", 0");

}

textList.add("\tdb " + buffer.toString() + "\t; " + size);

}

}

public static String generateValueName(Type type, Object value) {

if (value instanceof String) {

String text = (String) value;

StringBuilder buffer = new StringBuilder();

for (int index = 0; index < text.length(); ++index) {

if (Character.isLetterOrDigit(text.charAt(index)) ||

(text.charAt(index) == '\_')) {

buffer.append(text.charAt(index));

}

else if (text.charAt(index) != '\0') {

int asciiValue = (int) text.charAt(index);

char hex1 = "0123456789ABCDEF".charAt(asciiValue / 16),

hex2 = "0123456789ABCDEF".charAt(asciiValue % 16);

buffer.append(String.valueOf(hex1) + String.valueOf(hex2));

}

}

return Sort.String.name() + "\_" + buffer.toString() + Main.NumberId;

}

else {

String text = value.toString().replace("-", "minus");

switch (type.getSize()) {

case 1:

return "Byte\_" + text + Main.NumberId;

case 2:

return "Word\_" + text + Main.NumberId;

case 4:

return "DWord\_" + text + Main.NumberId;

case 8:

return "QWord\_" + text + Main.NumberId;

default:

Assert.error("");

return null;

}

}

}

private static String addOffset(int offset) {

if (offset > 0) {

return " + " + offset;

}

else if (offset < 0) {

return " - " + (-offset);

}

else {

return "";

}

}

private static String replaceOffset(String text, int oldOffset,

int newOffset) {

String oldText = (oldOffset == 0) ? "" : ((oldOffset > 0) ?

(" + " + oldOffset) : (" - " + (-oldOffset)));

String newText = (newOffset == 0) ? "" : ((newOffset > 0) ?

(" + " + newOffset) : (" - " + (-newOffset)));

return text.replace(oldText, newText);

}

public static Object staticConvert(Type toType, Object fromInit) {

if (toType.isStructOrUnion()) {

Assert.error(toType.isComplete(), toType,

"assigment of incomplete type");

}

if (fromInit instanceof SyntaxTree) {

SyntaxTree fromTree = (SyntaxTree) fromInit;

Type fromType = fromTree.getType();

if (fromType.isString()) {

Assert.error(toType.isPointerOrArray() &&

toType.getPointerOrArrayType().isChar(),

toType, "cannot be initalized by a string");

if (toType.isArray()) {

String text = (String) fromTree.getSymbol().getValue();

if (toType.getArraySize() == 0) {

toType.setArraySize(text.length() + 1);

}

Assert.error((text.length() + 1) <= toType.getArraySize(),

toType, "to many initializers");

Assert.warning((text.length() + 1) == toType.getArraySize(),

toType, "to few initializers");

}

}

else if (fromType.isPointer() && toType.isPointer()) {

Assert.error(fromType.getPointerType().isVoid() ||

toType.getPointerType().isVoid() ||

fromType.getPointerType().equals(toType.getPointerType()),

fromType, "cannot be converted to " + toType);

}

else if (fromType.isArray() && toType.isPointer()) {

Assert.error(fromType.getArrayType().isVoid() ||

toType.getPointerType().isVoid() ||

fromType.getArrayType().equals(toType.getPointerType()),

fromType, "cannot be converted to " + toType);

}

else if (fromType.isPointer() && toType.isArray()) {

Assert.error(toType, "an array cannot be initialized by a pointer");

}

else if (fromType.isArray() && toType.isArray()) {

Assert.error(fromType.getArrayType().equals(toType.getArrayType()),

fromType, "cannot be converted to " + toType);

if (toType.getArraySize() == 0) {

toType.setArraySize(fromType.getArraySize());

}

Assert.error(toType.getArraySize() == toType.getArraySize(),

fromType, "different array sizes");

}

else if (fromType.isFunction() && toType.isPointer() &&

toType.getPointerType().equals(fromType)) {

// Empty.

}

else if ((fromType.isLogical() || fromType.isArithmeticOrPointer()) &&

(toType.isArithmeticOrPointer())) {

Object fromValue = fromTree.getSymbol().getValue();

Object toValue = staticCast(fromValue, toType);

Symbol toSymbol = Symbol.createValueSymbol(toType, toValue);

fromInit = SyntaxTreeGenerator.generateValueTree(toSymbol);

}

else {

Assert.error(fromType.equals(toType), "invalid type");

}

}

else {

List<Object> fromList = (List) fromInit;

switch (toType.getSort()) {

case Pointer: {

Type subType = toType.getPointerType();

for (int index = 0; index < fromList.size(); ++index) {

fromList.set(index, staticConvert(subType, fromList.get(index)));

}

}

break;

case Array: {

if (toType.getArraySize() == 0) {

toType.setArraySize(fromList.size());

}

Assert.error(fromList.size() <= toType.getArraySize(),

toType, "to many initializers");

Assert.warning(fromList.size() == toType.getArraySize(),

toType, "to few initializers");

Type subType = toType.getArrayType();

for (int index = 0; index < fromList.size(); ++index) {

fromList.set(index, staticConvert(subType, fromList.get(index)));

}

}

break;

case Struct: {

Map<String,Symbol> memberMap = toType.getMemberMap();

Assert.error(fromList.size() <= memberMap.size(),

toType, "to many initializers");

Assert.warning(fromList.size() == memberMap.size(),

toType, "to few initializers");

int index = 0;

for (Symbol memberSymbol : memberMap.values()) {

if (index < fromList.size()) {

fromList.set(index, staticConvert(memberSymbol.getType(),

fromList.get(index++)));

}

}

}

break;

case Union: {

Assert.error(fromList.size() == 1, toType,

"a union can pnly be initalized by one value");

Map<String,Symbol> memberMap = toType.getMemberMap();

Type firstType = memberMap.values().iterator().next().getType();

fromList.set(0, staticConvert(firstType, fromList.get(0)));

}

break;

default:

Assert.error(toType, "cannot be initialized by a list");

break;

}

}

return fromInit;

}

public static Object staticCast(Object fromValue, Type toType) {

if ((fromValue instanceof Boolean) && toType.isIntegralOrPointer()) {

return (((Boolean) fromValue) ? BigInteger.ONE : BigInteger.ZERO);

}

else if ((fromValue instanceof Boolean) && toType.isFloating()) {

return (((Boolean) fromValue) ? BigDecimal.ONE : BigDecimal.ZERO);

}

if ((fromValue instanceof BigInteger) && toType.isFloating()) {

return (new BigDecimal((BigInteger) fromValue));

}

else if ((fromValue instanceof BigDecimal) && toType.isIntegralOrPointer()) {

Assert.warning(fromValue, "implicit conversion from " + fromValue +

" to " + toType);

return ((BigDecimal) fromValue).toBigInteger();

}

else {

return fromValue;

}

}

public static void generateStatic(Type type, Object init, List<Byte> byteList,

Map<Integer,String> accessMap, List<String> textList) {

if (type.isPointer()) {

if (init instanceof SyntaxTree) {

SyntaxTree tree =

SyntaxTreeOptimizer.optimizeExpression((SyntaxTree) init);

switch (tree.getOperator()) {

case Value:

if (tree.getType().isString()) {

String uniqueName =

generateValueName(tree.getType(), tree.getSymbol().getValue());

accessMap.put(byteList.size(), uniqueName);

byteList.add((byte) 0);

byteList.add((byte) 0);

textList.add("\tdw " + uniqueName + "\t; " + Type.PointerSize);

}

else {

Object value = tree.getSymbol().getValue();

generateByteTextList(type, value, byteList, accessMap, textList);

}

break;

case Variable:

if (tree.getType().isFunctionOrArray()) {

accessMap.put(byteList.size(), tree.getSymbol().getUniqueName());

byteList.add((byte) 0);

byteList.add((byte) 0);

textList.add("\tdw " + tree.getSymbol().getUniqueName() +

"\t; " + Type.PointerSize);

}

else {

Assert.error(tree.getSymbol().isStatic(), tree.getSymbol().getName(),

"a static variable can only be initalized by a static value");

for (Map.Entry<Integer,String> entry : tree.getSymbol().getAccessMap().entrySet()) {

accessMap.put(byteList.size() + entry.getKey(), entry.getValue());

}

byteList.addAll(tree.getSymbol().getByteList());

textList.addAll(remove\_labels(tree.getSymbol().getTextList()));

}

break;

case AddressOffset: { // array

accessMap.put(byteList.size(), tree.getSymbol().getUniqueName());

int offset = (Integer) tree.getInfo();

byteList.add((byte) offset);

byteList.add((byte) (offset << 8));

textList.add("\tdw " + tree.getSymbol().getUniqueName() +

addOffset(offset) + "\t; " + Type.PointerSize);

}

break;

case ValueOffset: { // pointer

for (Map.Entry<Integer,String> entry : tree.getSymbol().getAccessMap().entrySet()) {

accessMap.put(byteList.size() + entry.getKey(), entry.getValue());

}

List<Byte> oldByteList = tree.getSymbol().getByteList();

int oldOffset = oldByteList.get(0) + (oldByteList.get(1) >> 8);

int newOffset = (Integer) tree.getSymbol().getValue();

int newValue = oldOffset + newOffset;

byteList.add((byte) (newValue << 8));

byteList.add((byte) newValue);

textList.add(replaceOffset(tree.getSymbol().getTextList().get(0),

oldOffset, newValue));

}

break;

default:

Assert.error("X");

break;

}

}

else {

List<Byte> ptrByteList = new LinkedList<>();

Map<Integer,String> ptrAccessMap = new ListMap<>();

List<String> ptrTextList = new LinkedList<>();

Type arrayType = Type.createArrayType(0, type.getPointerType());

generateStatic(arrayType, init, ptrByteList, ptrAccessMap, ptrTextList);

String uniqueName = Main.SeparatorId + "ListInit" +

(Main.ListInitCount++) + Main.FileMarker;

accessMap.put(byteList.size(), uniqueName);

byteList.add((byte) 0);

byteList.add((byte) 0);

textList.add("\tdw " + uniqueName+ "\t; " + Type.PointerSize);

Symbol symbol = Symbol.createStaticSymbol(uniqueName, ptrByteList,

ptrAccessMap, ptrTextList);

Main.GlobalSet.add(symbol);

}

}

else if (init instanceof SyntaxTree) {

SyntaxTree tree =

SyntaxTreeOptimizer.optimizeExpression((SyntaxTree) init);

if ((tree.getOperator() == MiddleOperator.Value) &&

tree.getType().isString()) {

int arraySize = type.getArraySize();

String text = (String) tree.getSymbol().getValue();

int stringSize = text.length() + 1;

generateByteTextList(tree.getType(), text, byteList,

accessMap, textList);

generateZeroByteTextList((stringSize - arraySize) \* Type.CharSize,

byteList, textList);

}

else {

for (Map.Entry<Integer,String> entry : tree.getSymbol().getAccessMap().entrySet()) {

accessMap.put(byteList.size() + entry.getKey(), entry.getValue());

}

byteList.addAll(tree.getSymbol().getByteList());

textList.addAll(remove\_labels(tree.getSymbol().getTextList()));

}

}

else {

List list = (List) init;

switch (type.getSort()) {

case Array: {

Type subType = type.getArrayType();

for (int index = 0; index < list.size(); ++index) {

generateStatic(subType, list.get(index), byteList, accessMap, textList);

}

int arraySize = type.getArraySize();

generateZeroByteTextList((arraySize - list.size()) \* subType.getSize(),

byteList, textList);

}

break;

case Struct: {

int index = 0, size = 0;

for (Symbol memberSymbol : type.getMemberMap().values()) {

if (index < list.size()) {

Type memberType = memberSymbol.getType();

generateStatic(memberType, list.get(index++),

byteList, accessMap, textList);

size += memberType.getSize();

}

}

generateZeroByteTextList(type.getSize() - size, byteList,textList);

}

break;

case Union: {

Symbol firstSymbol =type.getMemberMap().values().iterator().next();

Assert.error(type.getSize() >= firstSymbol.getType().getSize(),

firstSymbol, "to large value");

generateStatic(firstSymbol.getType(), list.get(0), byteList,

accessMap, textList);

generateZeroByteTextList

(type.getSize() - firstSymbol.getType().getSize(),

byteList, textList);

}

break;

default:

Assert.error("Y");

break;

}

}

}

private static List<String> remove\_labels(List<String> textList) {

List<String> resultList = new LinkedList<>();

for (String text : textList) {

if (!text.endsWith(":")) {

resultList.add(text);

}

}

return resultList;

}

}

### Stack Initialization

Stack initialization occurs when an auto or register variable becomes initialized. Since the initialization value can be non-constant, no memory block is created. Instead, a sequence of assignment instructions is added to the middle code.

GenerateInitializerAuto.java

package c\_compiler;

import java.math.\*;

import java.util.\*;

public class GenerateInitializerAuto {

private static List<Object> fixList(int size, Type subType,

List<Object> list) {

Assert.error(subType.isArray(), "to many initializer");

int subSize = subType.getArraySize();

if ((size \* subSize) < list.size()) {

list = fixList(size \* subSize, subType.getArrayType(), list);

}

List<Object> newList = new LinkedList<>();

List<Object> subList = new LinkedList<>();

for (int index = 0; index < list.size();) {

subList.add(list.get(index++));

if ((index % subSize) == 0) {

newList.add(subList);

subList = new LinkedList<>();

}

}

if (!subList.isEmpty()) {

newList.add(subList);

}

return newList;

}

public static Object match(Type type, Object init) {

if (init instanceof SyntaxTree) {

SyntaxTree tree = (SyntaxTree) init;

if (type.isArray() && type.getArrayType().isChar() &&

(tree.getOperator() == MiddleOperator.Value) &&

(tree.getType().getSort() == Sort.String)) {

List<SyntaxTree> list = new LinkedList<>();

String text = (String) tree.getSymbol().getValue();

for (Character c : text.toCharArray()) {

Symbol charSymbol =

Symbol.createValueSymbol(Type.SignedCharType, BigInteger.valueOf(c));

list.add(new SyntaxTree(MiddleOperator.Value, charSymbol));

}

Symbol charSymbol =

Symbol.createValueSymbol(Type.SignedCharType, BigInteger.valueOf('\0'));

list.add(new SyntaxTree(MiddleOperator.Value, charSymbol));

return list;

}

else {

tree = SyntaxTreeGenerator.checkLogicalExpression(tree);

return TypeCast.implicitCast(tree, type);

}

}

else {

List list = (List) init;

switch (type.getSort()) {

case Pointer: {

Type subType = type.getPointerType();

for (int index = 0; index < list.size(); ++index) {

list.set(index, match(subType, list.get(index)));

}

return list;

}

case Array: {

if (type.getArraySize() == 0) {

type.setArraySize(list.size());

}

else {

Assert.warning(type.getArraySize() <= list.size(),

type, "too few initializers");

if (type.getArraySize() < list.size()) {

list = fixList(type.getArraySize(), type.getArrayType(), list);

}

}

Type subType = type.getArrayType();

for (int index = 0; index < list.size(); ++index) {

list.set(index, match(subType, list.get(index)));

}

return list;

}

case Struct: {

Assert.error(list.size() <= type.getMemberMap().size(),

type, "too many initializers");

Assert.warning(list.size() == type.getMemberMap().size(),

type, "too few initializers");

int index = 0;

for (Symbol memberSymbol : type.getMemberMap().values()) {

if (index < list.size()) {

list.set(index,

match(memberSymbol.getType(), list.get(index++)));

}

}

return list;

}

case Union:

Assert.error(list.size() == 1, type, "cannot be initialized with a" +

" list of more than one member");

Symbol firstMember = type.getMemberMap().values().iterator().next();

list.set(0, match(firstMember.getType(), list.get(0)));

return list;

default:

Assert.error(type, "cannot be initialized with a list");

return null;

}

}

}

public static Object generateAutoValueList(Object init) {

if (init instanceof SyntaxTree) {

return SyntaxTreeOptimizer.optimizeExpression((SyntaxTree) init);

}

else {

List list = (List) init;

for (int index = 0; index < list.size(); ++index) {

list.set(index, generateAutoValueList(list.get(index)));

}

return list;

}

}

public static void generateAuto(Symbol symbol, List<MiddleCode> middleCodeList,

Type type, Object init, String name, int offset) {

if (init instanceof SyntaxTree) {

MiddleCodeGenerator generateCode =

new MiddleCodeGenerator(middleCodeList);

SyntaxTree tree = (SyntaxTree) init;

Symbol initSymbol = generateCode.generateExpression(tree);

if (symbol == null) {

symbol = Symbol.createTemporarySymbol(type);

symbol.m\_name = name;

symbol.setOffset(offset);

}

generateCode.addMiddleCode(MiddleOperator.Assign,

true, symbol, initSymbol);

}

else {

List list = (List) init;

switch (type.getSort()) {

case Pointer: {

Type subType = type.getPointerType();

Type arrayType = Type.createArrayType(list.size(), subType);

List<Byte> subByteList = new LinkedList<>();

Map<Integer,String> subAccessMap = new ListMap<>();

List<String> subTextList = new LinkedList<>();

GenerateInitializerStatic.generateStatic

(arrayType, init, subByteList, subAccessMap, subTextList);

String uniqueName = Main.SeparatorId + "ListInit" +

(Main.ListInitCount++) + Main.FileMarker;

Symbol initSymbol =

Symbol.createInitSymbol(uniqueName, false, Storage.Static,

arrayType, subByteList, subAccessMap, subTextList);

Symbol tempSymbol = Symbol.createTemporarySymbol(type);

tempSymbol.m\_name = name;

tempSymbol.setOffset(offset);

(new MiddleCodeGenerator(middleCodeList)).addMiddleCode

(MiddleOperator.Assign, null, tempSymbol, initSymbol);

}

break;

case Array: {

Type subType = type.getArrayType();

if (type.getArraySize() == 0) {

type.setArraySize(list.size());

}

for (int index = 0; index < list.size(); ++index) {

generateAuto(null, middleCodeList, subType, list.get(index),

name + "[" + index + "]", offset);

offset += subType.getSize();

}

}

break;

case Struct: {

int index = 0;

for (Symbol memberSymbol : type.getMemberMap().values()) {

Type memberType = memberSymbol.getType();

String memberName = memberSymbol.getName();

generateAuto(null, middleCodeList, memberType, list.get(index++),

name + "." + memberName, offset);

offset += memberType.getSize();

}

}

break;

case Union: {

Symbol firstSymbol =type.getMemberMap().values().iterator().next();

Type firstType = firstSymbol.getType();

String firstName = firstSymbol.getName();

generateAuto(null, middleCodeList, firstType, list.get(0),

name + "." + firstName, offset);

}

break;

}

}

}

}

# Middle Code Optimization

When the middle code has been generated, we need to perform a number of optimizations since the code may be ineffective. Some ineffective parts may be introduced by the programmer, but most parts are likely to be introduced by the parser. The MiddleCodeOptimization class takes care of the optimization. Moreover, it does also generate base blocks and live sets, which are useful in the object code generation of Chapter 16.3.

## Middle Code Optimization

There is a set of optimizations available:

* We clear goto-next-statements: jumps to the next line.
* We also modify goto-next-double-statements: an conditional jump instruction that jumps two steps ahead and is followed by an unconditional jump instruction.
* We trace goto-chains: jump instructions that jump to other jump instructions.
* We clear unreachable code: code that is not reachable from the first function instruction.
* We remove empty code: code that has been cleared by the optimization above.

Since the optimization methods may reveal new optimization opportunities we need to repeat them until we do not detect any more opportunity. The field m\_update is set to true if we find an optimization opportunity.

MiddleCodeOptimizor.java

package c\_compiler;

import java.util.\*;

public class MiddleCodeOptimizer {

private boolean m\_update;

private DirectedGraph<BaseBlock> m\_controlFlowGraph;

private List<MiddleCode> m\_middleCodeList;

public MiddleCodeOptimizer(List<MiddleCode> middleCodeList) {

m\_middleCodeList = middleCodeList;

}

public void optimize() {

m\_update = true;

while (m\_update) {

m\_update = false;

clearGotoNextStatements();

clearDoubleRelationStatements();

traceGotoChains();

clearUnreachableCode();

removeClearedCode();

removePushPop();

mergePopPush();

}

}

### Next Goto Statements

It is completely meaningless to keep statements that jumps to the next line, why we remove them.

1. goto 2

2. ...

private void clearGotoNextStatements() {

for (int index = 1; index < (m\_middleCodeList.size() - 1); ++index) {

MiddleCode middleCode = m\_middleCodeList.get(index);

if (middleCode.isRelationOrGoto()) {

Integer target = (Integer) middleCode.getOperand(1);

if (target == (index + 1)) {

middleCode.clear();

m\_update = true;

}

}

}

}

### Next-Double Goto Statements

A conditional jump instruction that jumps two steps ahead and is followed by an unconditional jump instruction can be modified in that we reverse the condition and the target.

|  |  |  |
| --- | --- | --- |
| 1. if a < b goto 3  2. goto 10  3. … | 1. if a >= b goto 10  2. *removed*  3. ... |  |

private void clearDoubleRelationStatements() {

for (int index = 1; index < (m\_middleCodeList.size() - 1); ++index) {

MiddleCode thisCode = m\_middleCodeList.get(index),

nextCode = m\_middleCodeList.get(index + 1);

if (thisCode.isRelation() && nextCode.isGoto()) {

int target1 = (Integer) thisCode.getOperand(1),

target2 = (Integer) nextCode.getOperand(1);

if (target1 == (index + 2)) {

MiddleOperator operator1 = thisCode.getOperator();

thisCode.setOperator(MiddleCode.InverseMap.get(operator1));

thisCode.setOperand(1, target2);

nextCode.clear();

m\_update = true;

}

}

}

}

### Goto Chains

A goto chain is a sequence of unconditional jump instructions where each instruction in the chain except the last one jumps to another unconditional jump instruction. It would be more effective if they all jump to the same target as the last instruction.

|  |  |  |
| --- | --- | --- |
| 1. goto 3  2. ..  3. goto 5  4. ...  5. goto 7  6. ...  7. a = 1 | 1. goto 7  2. ..  3. goto 7  4. ...  5. goto 7  6. ...  7. a = 1 |  |

First, we trace all the unconditional jump instructions by calling traceGotoChains, then we change all the jump targets.

private void traceGotoChains() {

for (int index = 1; index < m\_middleCodeList.size(); ++index) {

MiddleCode middleCode = m\_middleCodeList.get(index);

if (middleCode.isRelationOrGoto()) {

Set<Integer> sourceSet = new ListSet<>();

sourceSet.add(index);

int firstTarget = (Integer) middleCode.getOperand(1);

int finalTarget = traceGoto(firstTarget, sourceSet);

if (firstTarget != finalTarget) {

for (int source : sourceSet) {

MiddleCode sourceCode = m\_middleCodeList.get(source);

sourceCode.setOperand(1, finalTarget);

}

m\_update = true;

}

}

}

}

private int traceGoto(int target, Set<Integer> sourceSet) {

MiddleCode objectCode = m\_middleCodeList.get(target);

if (!sourceSet.contains(target) && objectCode.isGoto()) {

sourceSet.add(target);

int nextTarget = (Integer) objectCode.getOperand(1);

return traceGoto(nextTarget, sourceSet);

}

else {

return target;

}

}

### Clear Unreachable Code

Code that is not reachable from the first instruction of the function shall be cleared. We call searchReachableCode that follows all conditional and unconditial jumps instructions and save their line numbers in visitedSet. We also warn if we reach the last instruction of a function that do not returns void.

private void clearUnreachableCode() {

Set<Integer> visitedSet = new ListSet<>();

searchReachableCode(0, visitedSet);

Assert.warning(!visitedSet.contains(m\_middleCodeList.size() - 1),

Main.FuncSymbol.getName(),"reached end of non-void function");

for (int index = 0; index < (m\_middleCodeList.size() - 1); ++index) {

if (!visitedSet.contains(index)) {

m\_middleCodeList.get(index).clear();

m\_update = true;

}

}

}

private void searchReachableCode(int index, Set visitedSet) {

for (; index < m\_middleCodeList.size(); ++index) {

if (visitedSet.contains(index)) {

return;

}

visitedSet.add(index);

MiddleCode middleCode = m\_middleCodeList.get(index);

if (middleCode.isRelation()) {

int target = (Integer) middleCode.getOperand(1);

searchReachableCode(target, visitedSet);

}

else if (middleCode.isGoto()) {

int target = (Integer) middleCode.getOperand(1);

searchReachableCode(target, visitedSet);

return;

}

else if ((middleCode.getOperator() == MiddleOperator.Exit) ||

(middleCode.getOperator() == MiddleOperator.Return)) {

return;

}

else if (middleCode.getOperator() == MiddleOperator.FunctionEnd) {

Symbol funcSymbol = (Symbol) middleCode.getOperand(1);

Assert.warning(!funcSymbol.getType().getReturnType().isVoid(),

funcSymbol.getName(), "reached the end of a non-void function");

return;

}

}

}

### Remove Cleared Code

Technically, it is not strictly necessary to remove cleared middle code instructions since they will not generate object code. However, the optimization methods of this section are easier to write if we do so.

private void removeClearedCode() {

for (int index1 = (m\_middleCodeList.size() - 2); index1 > 0;--index1){

if (m\_middleCodeList.get(index1).getOperator() ==

MiddleOperator.Empty){

for (int index2 = 0; index2 < m\_middleCodeList.size(); ++index2) {

MiddleCode middleCode = m\_middleCodeList.get(index2);

if (middleCode.isRelationOrGoto()) {

int target = (Integer) middleCode.getOperand(1);

if (target > index1) {

middleCode.setOperand(1, target - 1);

}

}

}

m\_middleCodeList.remove(index1);

}

}

}

## Generation

When the middle code has been optimized, we need to generate base blocks and live sets.

public void generate() {

{ generateDefAndUseLineSets();

Map<Integer,BaseBlock> leaderMap = new ListMap<>();

Set<BaseBlock> baseBlockSet = generateBaseBlockSet(leaderMap);

redundantExpressions(baseBlockSet);

removeClearedCode();

}

BaseBlock.clearCount();

Map<Integer,BaseBlock> leaderMap = new ListMap<>();

Set<BaseBlock> baseBlockSet = generateBaseBlockSet(leaderMap);

Set<Pair<BaseBlock,BaseBlock>> edgeSet =

generateEdgeSet(baseBlockSet, leaderMap);

m\_controlFlowGraph = new DirectedGraph<>(baseBlockSet, edgeSet);

generateLiveMaps();

generatePathSets();

if (Main.SmallWarning) {

generateDefAndUseLineSets();

generateDefAndUseBlockSets();

generatePathSets();

generateOutPathSets();

}

for (BaseBlock baseBlock : m\_controlFlowGraph.getVertexSet()) {

int firstIndex = baseBlock.getFirst(), lastIndex = baseBlock.getLast();

new RegisterPreparator(m\_middleCodeList, firstIndex, lastIndex);

for (int index = firstIndex; index <= lastIndex; ++index) {

Main.DebugStream.println(index + ": " + m\_middleCodeList.get(index));

}

}

}

### Base Blocks Generation

A base block is a maximal sequence of middle code where only the first instruction is a jump target. The idea is that we shall be able to trust that the registers hold the expected values thorugh the base block. In the first step, we look into all conditional and unconditional jump instructions and mark their targets as leaders, which will form the beginning of a base block. The first instruction of a function is always a leader and all instructions following a function all an interrupts system call is also marked as leaders, since a function or system call may affect the registers of the object code.

When all leaders have been established in the first step, we generate a list of pairs holding the first and last line number of each base block.

private Set<BaseBlock> generateBaseBlockSet

(Map<Integer,BaseBlock> leaderMap) {

m\_middleCodeList.get(0).setLeader(); // Function Start

for (int index = 0; index < (m\_middleCodeList.size() - 1); ++index) {

MiddleCode middleCode = m\_middleCodeList.get(index);

if (Main.CheckStackHeap &&

(middleCode.getOperator() == MiddleOperator.FunctionStart)) {

m\_middleCodeList.get(index + 1).setLeader();

}

if (middleCode.isRelationOrGoto()) {

int target = (Integer) middleCode.getOperand(1);

m\_middleCodeList.get(target).setLeader();

m\_middleCodeList.get(index + 1).setLeader();

}

else if ((middleCode.getOperator() == MiddleOperator.Call) ||

(middleCode.getOperator() == MiddleOperator.Interrupt)) {

m\_middleCodeList.get(index + 1).setLeader();

}

}

int startIndex = 0;

Set<BaseBlock> baseBlockSet = new ListSet<>();

for (int index = 1; index < m\_middleCodeList.size(); ++index) {

if (m\_middleCodeList.get(index).isLeader()) {

BaseBlock baseBlock = new BaseBlock(startIndex, index - 1);

leaderMap.put(startIndex, baseBlock);

baseBlockSet.add(baseBlock);

startIndex = index;

}

}

{ BaseBlock baseBlock =

new BaseBlock(startIndex, m\_middleCodeList.size() - 1);

leaderMap.put(startIndex, baseBlock);

baseBlockSet.add(baseBlock);

}

return baseBlockSet;

}

private Set<Pair<BaseBlock,BaseBlock>> generateEdgeSet(Set<BaseBlock> baseBlockSet,

Map<Integer,BaseBlock> leaderMap) {

Set<Pair<BaseBlock,BaseBlock>> edgeSet = new ListSet<>();

for (BaseBlock baseBlock : baseBlockSet) {

int firstIndex = baseBlock.getFirst(), lastIndex = baseBlock.getLast();

for (int index = firstIndex; index <= lastIndex; ++index) {

MiddleCode middleCode = m\_middleCodeList.get(index);

switch (middleCode.getOperator()) {

case Goto: {

int targetAddress = (int) middleCode.getOperand(1);

BaseBlock targetBlock = leaderMap.get(targetAddress);

edgeSet.add(new Pair<>(baseBlock, targetBlock));

}

break;

case Equal:

case NotEqual:

case LessThan:

case LessThanEqual:

case GreaterThan:

case GreaterThanEqual: {

BaseBlock nextBlock = leaderMap.get(index + 1);

edgeSet.add(new Pair<>(baseBlock, nextBlock));

int targetAddress = (int) middleCode.getOperand(1);

BaseBlock targetBlock = leaderMap.get(targetAddress);

edgeSet.add(new Pair<>(baseBlock, targetBlock));

}

break;

case FunctionEnd:

case Return:

case Exit:

break;

default:

if (index == lastIndex) {

BaseBlock nextBlock = leaderMap.get(index + 1);

edgeSet.add(new Pair<>(baseBlock, nextBlock));

}

break;

}

}

}

return edgeSet;

}

### Live Sets Generation

We equip each middle code instruction with a live set; that is, the set of all symbols which values are in demand later in the base block. A symbol is live if it is an operand in a unary or binary expression, or as a parameter or return value. However, it is not live if it is the result of an expression.

private void generateLiveMaps() {

for (BaseBlock baseBlock : m\_controlFlowGraph.getVertexSet()) {

int firstIndex = baseBlock.getFirst(), lastIndex = baseBlock.getLast();

Map<Symbol,Integer> liveMap = new ListMap<>();

for (int index = lastIndex; index >= firstIndex; --index) {

MiddleCode middleCode = m\_middleCodeList.get(index);

middleCode.setLiveMap(((ListMap) liveMap).clone());

MiddleOperator middleOp = middleCode.getOperator();

switch (middleOp) {

case Address:

break;

default: {

Object resultObject = middleCode.getOperand(1),

leftObject = middleCode.getOperand(2),

rightObject = middleCode.getOperand(3);

Symbol resultSymbol, leftSymbol, rightSymbol;

if (middleCode.getOperator() == MiddleOperator.Assign) {

resultSymbol = (leftObject instanceof Symbol) ? ((Symbol) leftObject) : null;

leftSymbol = (rightObject instanceof Symbol) ? ((Symbol) rightObject) : null;

rightSymbol = null;

}

else {

resultSymbol = (resultObject instanceof Symbol) ? ((Symbol) resultObject) : null;

leftSymbol = (leftObject instanceof Symbol) ? ((Symbol) leftObject) : null;

rightSymbol = (rightObject instanceof Symbol) ? ((Symbol) rightObject) : null;

}

Symbol resultAddressSymbol = (resultSymbol != null) ? resultSymbol.getAddressSymbol() : null,

leftAddressSymbol = (leftSymbol != null) ? leftSymbol.getAddressSymbol() : null,

rightAddressSymbol = (rightSymbol != null) ? rightSymbol.getAddressSymbol() : null;

removeIfNotNull(resultSymbol, liveMap);

addIfNotNull(resultAddressSymbol, liveMap);

addIfNotNull(leftSymbol, liveMap);

addIfNotNull(leftAddressSymbol, liveMap);

addIfNotNull(rightSymbol, liveMap);

addIfNotNull(rightAddressSymbol, liveMap);

/\* if (middleCode.isMultiply()) {

Set<Symbol> removeSet = new ListSet<>();

for (Map.Entry<Symbol,Pair<Integer,Boolean>> entry : liveMap.entrySet()) {

Symbol symbol = entry.getKey();

Pair<> pair = entry.getValue();

boolean isMul = pair.getSecond();

if (isMul) {

removeSet.add(symbol);

}

}

for (Symbol symbol : removeSet) {

liveMap.removeParentsAndChildSymbols(symbol);

}

}\*/

}

}

}

}

}

private void removeIfNotNull(Symbol symbol, Set<Symbol> liveSet) {

if (symbol != null) {

liveSet.remove(symbol);

}

}

private void addIfNotNull(Symbol symbol, Set<Symbol> liveSet) {

if (symbol != null) {

liveSet.add(symbol);

}

}

### Path Sets

private void generatePathSets() {

BaseBlock firstBlock = m\_controlFlowGraph.getVertexSet().iterator().next();

Assert.error(m\_controlFlowGraph.getInNeighbourSet(firstBlock).isEmpty());

int size = m\_controlFlowGraph.getVertexSet().size();

for (BaseBlock startBlock : m\_controlFlowGraph.getVertexSet()) {

for (BaseBlock inBlock:m\_controlFlowGraph.getInNeighbourSet(startBlock)) {

generateInPathSet(firstBlock, startBlock, inBlock,

new LinkedList<Object>(), startBlock.getInPathSet());

}

for (BaseBlock outBlock:m\_controlFlowGraph.getOutNeighbourSet(startBlock)){

generateOutPathSet(startBlock, outBlock, new LinkedList<Object>(),

startBlock.getOutPathSet());

}

}

}

private void generateInPathSet(BaseBlock firstBlock, BaseBlock startBlock,

BaseBlock currBlock, List<Object> inPath,

Set<List<Object>> inPathSet){

if (currBlock == firstBlock) {

inPath.add(0, currBlock);

inPathSet.add(new LinkedList<>(inPath));

inPath.remove(0);

}

else if ((currBlock != startBlock) && !inPath.contains(currBlock)) {

inPath.add(0, currBlock);

for (BaseBlock inBlock:m\_controlFlowGraph.getInNeighbourSet(currBlock)) {

generateInPathSet(firstBlock, startBlock, inBlock, inPath, inPathSet);

}

inPath.remove(0);

}

}

private void generateOutPathSet(BaseBlock startBlock, BaseBlock currBlock,

List<Object> outPath, Set<List<Object>> outPathSet) {

Set<BaseBlock> outNeighbourSet =

m\_controlFlowGraph.getOutNeighbourSet(currBlock);

if ((currBlock == startBlock) || outNeighbourSet.isEmpty()) {

outPath.add(currBlock);

outPathSet.add(new LinkedList<>(outPath));

outPath.remove(outPath.size() - 1);

}

else if (outPath.contains(currBlock)) {

outPathSet.add(new LinkedList<>(outPath));

}

else {

outPath.add(currBlock);

Set<List<Object>> longPathSet = new ListSet<>();

for (BaseBlock outBlock : outNeighbourSet) {

generateOutPathSet(startBlock, outBlock, outPath, longPathSet);

}

if (!longPathSet.isEmpty()) {

Set<List<Object>> shortPathSet = new ListSet<>();

for (List<Object> longPath : longPathSet) {

shortPathSet.add(longPath.subList(outPath.size(), longPath.size()));

}

List<Object> newPath = new LinkedList<>(outPath);

if (shortPathSet.size() == 1) {

newPath.addAll(shortPathSet.iterator().next());

outPathSet.add(newPath);

}

else {

newPath.add(shortPathSet);

outPathSet.add(newPath);

}

}

outPath.remove(outPath.size() - 1);

}

}

private void generateDefAndUseBlockSets() {

for (BaseBlock baseBlock : m\_controlFlowGraph.getVertexSet()) {

int firstIndex = baseBlock.getFirst(), lastIndex = baseBlock.getLast();

for (int index = lastIndex; index >= firstIndex; --index) {

MiddleCode middleCode = m\_middleCodeList.get(index);

addParentsAndChildSymbols(baseBlock.getHardDefSymbolSet(),

middleCode.getDefSymbolSet());

addParentsAndChildSymbols(baseBlock.getHardUseSymbolSet(),

middleCode.getUseSymbolSet());

addParentsAndChildSymbols(baseBlock.getSoftDefSymbolSet(),

middleCode.getDefSymbolSet());

removeParentsAndChildSymbols(baseBlock.getSoftDefSymbolSet(),

baseBlock.getHardUseSymbolSet());

removeParentsAndChildSymbols(baseBlock.getSoftUseSymbolSet(),

middleCode.getDefSymbolSet());

addParentsAndChildSymbols(baseBlock.getSoftUseSymbolSet(),

middleCode.getUseSymbolSet());

}

}

}

private void addParentsAndChildSymbols(Set<Symbol> mainSet,

Set<Symbol> addSet) {

for (Symbol symbol : addSet) {

Symbol parentSymbol = symbol;

while (parentSymbol != null) {

mainSet.add(parentSymbol);

parentSymbol = parentSymbol.getParentSymbol();

}

addChildSet(mainSet, symbol.getChildSymbolSet());

}

}

private void addChildSet(Set<Symbol> mainSet, Set<Symbol> childSet) {

for (Symbol childSymbol : childSet) {

mainSet.add(childSymbol);

addChildSet(mainSet, childSymbol.getChildSymbolSet());

}

}

private void removeParentsAndChildSymbols(Set<Symbol> mainSet,

Set<Symbol> removeSet) {

for (Symbol removeSymbol : removeSet) {

Symbol parentSymbol = removeSymbol;

while (parentSymbol != null) {

mainSet.remove(parentSymbol);

parentSymbol = parentSymbol.getParentSymbol();

}

removeChildSet(mainSet, removeSymbol.getChildSymbolSet());

}

}

private void removeChildSet(Set<Symbol> mainSet, Set<Symbol> childSet) {

for (Symbol childSymbol : childSet) {

mainSet.remove(childSymbol);

removeChildSet(mainSet, childSymbol.getChildSymbolSet());

}

}

private void removeIfNotNull(Symbol symbol, Map<Symbol,Integer> liveMap) {

if (symbol != null) {

liveMap.remove(symbol);

}

}

private void addIfNotNull(Symbol symbol, Map<Symbol,Integer> liveMap) {

if ((symbol != null) && !symbol.hasValue() && !symbol.isSolid()) {

int count = liveMap.containsKey(symbol) ? liveMap.get(symbol) : 0;

liveMap.put(symbol, count + 1);

}

}

}

### Removal of Unused Code

Unused code is code that does not perform any meaningful task, such as:

|  |  |  |
| --- | --- | --- |
| x \* y; | 1. $0 = x \* y |  |

We use the live sets and remove all instructions that temporary symbols that are not a member of the live set.

# Object Code Preparation

## Register Preparation

RegisterPreparator.java

package c\_compiler;

import java.util.\*;

public class RegisterPreparator {

private List<MiddleCode> m\_middleCodeList;

private int m\_axLog = -1;

public RegisterPreparator(List<MiddleCode> middleCodeList,

int firstIndex, int lastIndex,Set<Track> trackSet){

m\_middleCodeList = middleCodeList;

prepareRegister(firstIndex, lastIndex, trackSet);

}

public static final Map<Integer,Register> LeftMultiplyMap = new ListMap<>(),

ClearMultiplyMap = new ListMap<>();

public static final Map<Pair<MiddleOperator,Integer>,Register>

ResultMultiplyMap = new ListMap<>();

static {

LeftMultiplyMap.put(Type.ShortSize, Register.al);

LeftMultiplyMap.put(Type.IntegerSize, Register.ax);

LeftMultiplyMap.put(Type.LongSize, Register.eax);

ClearMultiplyMap.put(Type.ShortSize,Register.ah);

ClearMultiplyMap.put(Type.IntegerSize, Register.dx);

ClearMultiplyMap.put(Type.LongSize, Register.edx);

ResultMultiplyMap.put(new Pair<>(MiddleOperator.Multiply, Type.ShortSize),

Register.al);

ResultMultiplyMap.put(new Pair<>(MiddleOperator.Multiply,Type.IntegerSize),

Register.ax);

ResultMultiplyMap.put(new Pair<>(MiddleOperator.Multiply, Type.LongSize),

Register.eax);

ResultMultiplyMap.put(new Pair<>(MiddleOperator.Divide, Type.ShortSize),

Register.al);

ResultMultiplyMap.put(new Pair<>(MiddleOperator.Divide, Type.IntegerSize),

Register.ax);

ResultMultiplyMap.put(new Pair<>(MiddleOperator.Divide, Type.LongSize),

Register.eax);

ResultMultiplyMap.put(new Pair<>(MiddleOperator.Modulo, Type.ShortSize),

Register.ah);

ResultMultiplyMap.put(new Pair<>(MiddleOperator.Modulo, Type.IntegerSize),

Register.dx);

ResultMultiplyMap.put(new Pair<>(MiddleOperator.Modulo, Type.LongSize),

Register.edx);

}

private Track createTrack(Symbol symbol, Map<Symbol,Track> trackMap,

Set<Track> trackSet) {

Track track = new Track(symbol);

if (!symbol.isSolid()) {

trackMap.put(symbol, track);

}

trackSet.add(track);

return track;

}

private Track createTrack(Type type, Set<Track> trackSet) {

Track track = new Track(type);

trackSet.add(track);

return track;

}

/\*private boolean mustLoadAddressRegister(int position, int index, Symbol symbol,

Map<Symbol,Track>trackMap, Set<Track> trackSet) {

Symbol getAddressSymbol = symbol.getAddressSymbol();

boolean load = false;

if (getAddressSymbol != null) {

Track addressTrack = trackMap.get(getAddressSymbol);

if (addressTrack == null) {

addressTrack = createTrack(getAddressSymbol, trackMap, trackSet);

load = true;

}

addressTrack.addCode(position, index);

addressTrack.setPointer();

}

return load;

}\*/

/\*private Track mayLoadAddressRegister(int position, int index, Symbol symbol,

Map<Symbol,Track>trackMap, Set<Track> trackSet) {

Symbol getAddressSymbol = symbol.getAddressSymbol();

if (getAddressSymbol != null) {

Track addressTrack = trackMap.get(getAddressSymbol);

if (addressTrack != null) {

addressTrack.addCode(position, index);

addressTrack.setPointer();

return addressTrack;

}

}

return null;

}

private Symbol lookupRegister(Register register, Map<Symbol,Track> trackMap) {

for (Map.Entry<Symbol,Track> entry : trackMap.entrySet()) {

Symbol symbol = entry.getKey();

Track track = entry.getValue();

if (ObjectCode.registerOverlap(register, track.getRegister())) {

return symbol;

}

}

return null;

}\*/

public void prepareRegister(int firstLine, int lastLine,Set<Track> trackSet){

Map<Symbol,Track> trackMap = new ListMap<>();

for (int line = firstLine; line <= lastLine; ++line) {

MiddleCode middleCode = m\_middleCodeList.get(line);

MiddleOperator operator = middleCode.getOperator();

/\*Set<Symbol> removeSet = new ListSet<>();

for (Symbol symbol : trackMap.keySet()) {

if (!middleCode.isLive(symbol)) {

removeSet.add(symbol);

}

}

for (Symbol symbol : removeSet) {

trackMap.remove(symbol);

}\*/

Object operand1 = middleCode.getOperand(1),

operand2 = middleCode.getOperand(2),

operand3 = middleCode.getOperand(3);

switch (operator) {

case UnaryAdd:

case UnarySubtract:

case BitwiseNot: {

Symbol resultSymbol = (Symbol) operand1,

unarySymbol = (Symbol) operand2;

if (resultSymbol.getType().isIntegralOrPointer()) {

Track unaryTrack = trackMap.get(unarySymbol);

if (unaryTrack == null) {

middleCode.setLoad(Track.Left);

unaryTrack = createTrack(unarySymbol, trackMap, trackSet);

tryLoadAddressSymbol(Track.LeftAddress, line,

unarySymbol, trackMap);

/\*Symbol getAddressSymbol = unarySymbol.getAddressSymbol();

if (getAddressSymbol != null) {

Track addressTrack = trackMap.get(getAddressSymbol);

if (addressTrack != null) {

addressTrack.addCode(Track.LeftAddress, line, unarySymbol.getType());

}

}\*/

}

trackMap.remove(unarySymbol);

trackMap.put(resultSymbol, unaryTrack);

unaryTrack.addCode(Track.Left, line, unarySymbol.getType());

}

}

break;

/\*

case UnaryAdd:

case UnarySubtract:

case BitwiseNot: {

if (middleCode.toString().contains("UnarySubtract $temp141 count")) {

int i = 1;

}

Symbol resultSymbol = (Symbol) getOperand1,

unarySymbol = (Symbol) getOperand2;

if (resultSymbol.getType().isIntegralOrPointer()) {

Track unaryTrack = trackMap.get(unarySymbol);

if (unaryTrack != null) {

unaryTrack.addCode(Track.Left, line, unarySymbol.getType());

trackMap.remove(unarySymbol);

trackMap.put(resultSymbol, unaryTrack);

}

else if (middleCode.isLive(resultSymbol)) {

middleCode.setLoad(Track.Left);

unaryTrack = createTrack(unarySymbol, trackMap, trackSet);

unaryTrack.addCode(Track.Left, line, unarySymbol.getType());

trackMap.put(resultSymbol, unaryTrack);

}

if (unaryTrack == null) {

mayLoadAddressRegister(Track.LeftAddress, line, unarySymbol, trackMap, trackSet);

}

}

}

break;

\*/

case EqualZero: {

Symbol equalSymbol = (Symbol) operand2;

if (equalSymbol.getType().isIntegralOrPointer()) {

Track equalTrack = trackMap.get(equalSymbol);

if (equalTrack != null) {

equalTrack.addCode(Track.Left, line, equalSymbol.getType());

}

else if (middleCode.isLive(equalSymbol)) {

middleCode.setLoad(Track.Left);

equalTrack = createTrack(equalSymbol, trackMap, trackSet);

equalTrack.addCode(Track.Left, line, equalSymbol.getType());

trackMap.put(equalSymbol, equalTrack);

}

if (equalTrack == null) {

tryLoadAddressSymbol(Track.LeftAddress, line,

equalSymbol, trackMap);

}

}

}

break;

case Address: {

Symbol resultSymbol = (Symbol) operand1,

symbol = (Symbol) operand2;

Symbol addressSymbol = symbol.getAddressSymbol();

// &a[i]

// &p->m

// &s.m

if (addressSymbol != null) {

Track addressTrack = trackMap.get(addressSymbol);

if (addressTrack == null) {

middleCode.setLoad(Track.Right);

addressTrack = createTrack(addressSymbol, trackMap, trackSet);

}

addressTrack.addCode(Track.Right, line);

addressTrack.addCode(Track.Left, line);

trackMap.put(resultSymbol, addressTrack);

}

else {

Track resultTrack = createTrack(resultSymbol, trackMap,trackSet);

resultTrack.addCode(Track.Left, line);

}

}

break;

case PrefixIncrement:

case PrefixDecrement: {

Symbol incSymbol = (Symbol) operand3;

if (incSymbol.getType().isIntegralOrPointer()) {

Track incTrack = trackMap.get(incSymbol);

if (incTrack != null) {

incTrack.addCode(Track.Right, line, incSymbol.getType());

trackMap.put(incSymbol, incTrack);

}

else {

tryLoadAddressSymbol(Track.RightAddress, line,

incSymbol, trackMap);

if (middleCode.isLive(incSymbol)) {

middleCode.setLoad(Track.Right);

incTrack = createTrack(incSymbol, trackMap, trackSet);

incTrack.addCode(Track.Right, line, incSymbol.getType());

trackMap.put(incSymbol, incTrack);

}

}

}

}

break;

case PostfixIncrement:

case PostfixDecrement: {

Symbol resultSymbol = (Symbol) operand2,

incSymbol = (Symbol) operand3;

if (incSymbol.getType().isIntegralOrPointer()) {

Track incTrack = trackMap.get(incSymbol);

if (incTrack != null) {

incTrack.addCode(Track.Right, line, incSymbol.getType());

trackMap.put(incSymbol, incTrack);

}

else {

tryLoadAddressSymbol(Track.RightAddress, line,

incSymbol, trackMap);

if (middleCode.isLive(incSymbol)) {

middleCode.setLoad(Track.Right);

incTrack = createTrack(incSymbol, trackMap, trackSet);

incTrack.addCode(Track.Right, line, incSymbol.getType());

trackMap.put(incSymbol, incTrack);

}

}

Track resultTrack = createTrack(resultSymbol, trackMap,trackSet);

resultTrack.addCode(Track.Left, line, resultSymbol.getType());

trackMap.put(resultSymbol, resultTrack);

}

}

break;

case Return: {

middleCode.setLoad(Track.Left);

Track returnTrack = createTrack(Type.PointerType, trackSet);

returnTrack.addCode(Track.Left, line);

}

break;

case Parameter: { // mov [bp + 10], 123; mov [bp + 10], ax

Symbol paramSymbol = (Symbol) operand2;

if ((paramSymbol.getType().isIntegralOrPointer() ||

paramSymbol.getType().isFunctionOrArray()) &&

!paramSymbol.hasValue()) {

Track paramTrack = trackMap.get(paramSymbol);

if (paramTrack == null) {

middleCode.setLoad(Track.Left);

paramTrack = createTrack(paramSymbol, trackMap, trackSet);

tryLoadAddressSymbol(Track.LeftAddress, line,

paramSymbol, trackMap);

}

paramTrack.addCode(Track.Left, line, paramSymbol.getType());

}

else if (paramSymbol.getType().isStructOrUnion()) {

Symbol rightAddressSymbol = paramSymbol.getAddressSymbol();

Track rightAddressTrack;

if (rightAddressSymbol != null) {

rightAddressTrack = trackMap.get(rightAddressSymbol);

if (rightAddressTrack == null) {

middleCode.setLoad(Track.Right);

rightAddressTrack = createTrack(rightAddressSymbol,

trackMap, trackSet);

}

}

else {

middleCode.setLoad(Track.Right);

rightAddressTrack = createTrack(Type.PointerType, trackSet);

}

rightAddressTrack.addCode(Track.RightAddress, line);

rightAddressTrack.setPointer();

trackMap.remove(rightAddressSymbol);

middleCode.setLoad(Track.Left);

Track targetAddressTrack =

createTrack(Type.PointerType, trackSet);

targetAddressTrack.addCode(Track.LeftAddress, line);

targetAddressTrack.setPointer();

Symbol valueSymbol =

Symbol.createTemporarySymbol(Type.SignedCharType);

Track valueTrack = createTrack(valueSymbol, trackMap, trackSet);

valueTrack.addCode(Track.Left, line);

int typeSize = paramSymbol.getType().getSize();

Type countType = (typeSize < 256) ? Type.SignedCharType

: Type.SignedIntegerType;

Symbol countSymbol = Symbol.createTemporarySymbol(countType);

Track countTrack = createTrack(countSymbol, trackMap, trackSet);

countTrack.addCode(Track.Right, line);

}

}

break;

case SetReturnValue: {

Symbol returnSymbol = (Symbol) operand2;

if (returnSymbol.getType().isIntegralOrPointer() ||

returnSymbol.getType().isFunctionArrayOrString()) {

int typeSize = returnSymbol.getType().isFunctionArrayOrString() ?

Type.PointerSize : returnSymbol.getType().getSize();

Register returnRegister =

ObjectCode.registerToSize(ObjectCode.ReturnValueRegister, typeSize);

Track returnTrack = trackMap.get(returnSymbol);

if (returnTrack != null) {

if (returnTrack.getRegister() == null) {

returnTrack.setRegister(returnRegister);

}

}

else {

middleCode.setLoad(Track.Left);

returnTrack = createTrack(returnSymbol, trackMap, trackSet);

returnTrack.setRegister(returnRegister);

tryLoadAddressSymbol(Track.LeftAddress, line,

returnSymbol, trackMap);

}

returnTrack.addCode(Track.Left, line, returnSymbol.getType());

}

else if (returnSymbol.getType().isStructOrUnion()) {

Symbol returnAddressSymbol = returnSymbol.getAddressSymbol();

Track returnAddressTrack;

if (returnAddressSymbol != null) {

returnAddressTrack = trackMap.get(returnAddressSymbol);

if (returnAddressTrack == null) {

middleCode.setLoad(Track.Right);

returnAddressTrack =

createTrack(returnAddressSymbol, trackMap, trackSet);

}

}

else {

middleCode.setLoad(Track.Right);

returnAddressTrack = createTrack(Type.PointerType, trackSet);

}

returnAddressTrack.addCode(Track.LeftAddress, line);

}

}

break;

case GetReturnValue: {

Symbol returnSymbol = (Symbol) operand1;

if (returnSymbol.getType().isIntegralOrPointer()) {

Track returnTrack = createTrack(returnSymbol, trackMap, trackSet);

returnTrack.addCode(Track.Left, line,

ObjectCode.ReturnValueRegister);

if (!middleCode.isLive(returnSymbol)) {

middleCode.setSave();

Main.CurrentTable.addSymbol(returnSymbol);

}

}

else if (returnSymbol.getType().isFloating()) {

// Empty.

}

else if (returnSymbol.getType().isStructOrUnion()) {

Symbol returnAddressSymbol = returnSymbol.getAddressSymbol();

Track returnAddressTrack;

if (returnAddressSymbol != null) {

returnAddressTrack = trackMap.get(returnAddressSymbol);

if (returnAddressTrack == null) {

middleCode.setLoad(Track.Left);

returnAddressTrack =

createTrack(returnAddressSymbol, trackMap, trackSet);

}

}

else {

middleCode.setLoad(Track.Left);

returnAddressTrack = createTrack(Type.PointerType, trackSet);

}

returnAddressTrack.addCode(Track.LeftAddress, line);

returnAddressTrack.setPointer();

Symbol valueSymbol =

Symbol.createTemporarySymbol(Type.SignedCharType);

Track valueTrack = createTrack(valueSymbol, trackMap, trackSet);

valueTrack.addCode(Track.Left, line);

int typeSize = returnSymbol.getType().getSize();

Type countType = (typeSize < 256) ? Type.SignedCharType

: Type.SignedIntegerType;

Symbol countSymbol = Symbol.createTemporarySymbol(countType);

Track countTrack = createTrack(countSymbol, trackMap, trackSet);

countTrack.addCode(Track.Right, line);

}

}

break;

case Exit:

if (operand2 != null) {

Symbol exitSymbol = (Symbol) operand2;

if (exitSymbol.getType().isIntegralOrPointer()) {

Track exitTrack = trackMap.get(exitSymbol);

if (exitTrack != null) {

if (exitTrack.getRegister() == null) {

exitTrack.setRegister(Register.dl);

}

}

else {

middleCode.setLoad(Track.Left);

exitTrack = createTrack(exitSymbol, trackMap, trackSet);

tryLoadAddressSymbol(Track.LeftAddress, line,

exitSymbol, trackMap);

}

exitTrack.addCode(Track.Left, line, Register.dl);

}

}

break;

case Call: {

Symbol calleeSymbol = (Symbol) operand1;

if (calleeSymbol.getType().isPointer()) {

Track calleeTrack = trackMap.get(calleeSymbol);

if (calleeTrack != null) {

calleeTrack.addCode(Track.Left, line, calleeSymbol.getType());

}

else {

tryLoadAddressSymbol(Track.LeftAddress, line, calleeSymbol, trackMap);

}

}

}

break;

case Deref:

case Arrow:

case Index: {

Symbol leftSymbol = (Symbol) operand2,

rightSymbol = (Symbol) operand3;

Track rightTrack = trackMap.get(rightSymbol);

if (rightTrack == null) {

middleCode.setLoad(Track.Right);

rightTrack = createTrack(rightSymbol, trackMap, trackSet);

tryLoadAddressSymbol(Track.RightAddress, line,

rightSymbol, trackMap);

}

rightTrack.addCode(Track.Right, line);

rightTrack.addCode(Track.LeftAddress, line);

rightTrack.setPointer();

if (leftSymbol.getType().isIntegralOrPointer()) {

if (!leftSymbol.isTemporary()) { // XXX

Track leftTrack = createTrack(leftSymbol, trackMap, trackSet);

leftTrack.addCode(Track.Left, line, leftSymbol.getType());

middleCode.setSave();

}

}

}

break;

case PushFloat: {

Symbol pushSymbol = (Symbol) middleCode.getOperand(1);

if (pushSymbol.getType().isIntegralOrPointer()) {

Track pushTrack = trackMap.get(pushSymbol);

if (pushTrack != null) {

pushTrack.addCode(Track.Left, line);

}

else {

tryLoadAddressSymbol(Track.LeftAddress, line,

pushSymbol, trackMap);

}

}

}

break;

case Assign: {

Symbol leftSymbol = (Symbol) operand2,

rightSymbol = (Symbol) operand3;

tryLoadAddressSymbol(Track.LeftAddress, line, leftSymbol,trackMap);

tryLoadAddressSymbol(Track.RightAddress,line,rightSymbol,trackMap);

if (leftSymbol.getType().isIntegralOrPointer()) {

if (rightSymbol.hasValue()) {

if (middleCode.isLive(leftSymbol)) {

Track leftTrack = createTrack(leftSymbol, trackMap,trackSet);

leftTrack.addCode(Track.Left, line, leftSymbol.getType());

trackMap.put(leftSymbol, leftTrack);

}

}

else {

Track rightTrack = trackMap.get(rightSymbol);

if (rightTrack == null) {

middleCode.setLoad(Track.Right);

rightTrack = createTrack(rightSymbol, trackMap, trackSet);

tryLoadAddressSymbol(Track.RightAddress, line,

rightSymbol, trackMap);

trackMap.put(rightSymbol, rightTrack);

}

rightTrack.addCode(Track.Left, line, leftSymbol.getType());

rightTrack.addCode(Track.Right, line, rightSymbol.getType());

trackMap.put(leftSymbol, rightTrack);

}

mustLoadAddressSymbol(Track.LeftAddress, line, leftSymbol,

middleCode, trackMap, trackSet);

middleCode.setSave();

}

else if (leftSymbol.getType().isFloating()) {

mustLoadAddressSymbol(Track.LeftAddress, line, leftSymbol,

middleCode, trackMap, trackSet);

}

else if (leftSymbol.getType().isStructOrUnion()) {

Symbol leftAddressSymbol = leftSymbol.getAddressSymbol(),

rightAddressSymbol = rightSymbol.getAddressSymbol();

Track leftAddressTrack, rightAddressTrack;

if (leftAddressSymbol != null) {

leftAddressTrack = trackMap.get(leftAddressSymbol);

if (leftAddressTrack == null) {

middleCode.setLoad(Track.Left);

leftAddressTrack =

createTrack(leftAddressSymbol, trackMap, trackSet);

}

}

else {

middleCode.setLoad(Track.Left);

leftAddressTrack = createTrack(Type.PointerType, trackSet);

}

leftAddressTrack.addCode(Track.LeftAddress, line);

leftAddressTrack.setPointer();

trackMap.remove(leftAddressSymbol);

if (rightAddressSymbol != null) {

rightAddressTrack = trackMap.get(rightAddressSymbol);

if (rightAddressTrack == null) {

middleCode.setLoad(Track.Right);

rightAddressTrack =

createTrack(rightAddressSymbol, trackMap, trackSet);

}

}

else {

middleCode.setLoad(Track.Right);

rightAddressTrack = createTrack(Type.PointerType, trackSet);

}

rightAddressTrack.addCode(Track.RightAddress, line);

rightAddressTrack.setPointer();

trackMap.remove(rightAddressSymbol);

Symbol valueSymbol =

Symbol.createTemporarySymbol(Type.SignedCharType);

Track valueTrack = createTrack(valueSymbol, trackMap, trackSet);

valueTrack.addCode(Track.Left, line);

int typeSize = leftSymbol.getType().getSize();

Type countType = (typeSize < 256) ? Type.SignedCharType

: Type.SignedIntegerType;

Symbol countSymbol = Symbol.createTemporarySymbol(countType);

Track countTrack = createTrack(countSymbol, trackMap, trackSet);

countTrack.addCode(Track.Right, line);

}

}

break;

case Equal:

case NotEqual:

case LessThan:

case LessThanEqual:

case GreaterThan:

case GreaterThanEqual: {

Symbol leftSymbol = (Symbol) operand2,

rightSymbol = (Symbol) operand3;

tryLoadAddressSymbol(Track.LeftAddress, line,leftSymbol, trackMap);

tryLoadAddressSymbol(Track.RightAddress,line,rightSymbol,trackMap);

if (leftSymbol.getType().isIntegralOrPointer()) {

if (leftSymbol.hasValue()) {

Track rightTrack = trackMap.get(rightSymbol);

if ((rightTrack != null) && rightSymbol.hasValue()) {

rightTrack.addCode(Track.Right, line, rightSymbol.getType());

}

else if (middleCode.isLive(rightSymbol) && rightSymbol.hasValue()) {

middleCode.setLoad(Track.Right);

rightTrack = createTrack(rightSymbol, trackMap, trackSet);

rightTrack.addCode(Track.Right, line, rightSymbol.getType());

}

}

else if (rightSymbol.hasValue()) {

Track leftTrack = trackMap.get(leftSymbol);

if ((leftTrack != null) && rightSymbol.hasValue()) {

leftTrack.addCode(Track.Left, line, leftSymbol.getType());

}

else if (middleCode.isLive(leftSymbol) && rightSymbol.hasValue()) {

middleCode.setLoad(Track.Left);

leftTrack = createTrack(leftSymbol, trackMap, trackSet);

leftTrack.addCode(Track.Left, line, leftSymbol.getType());

}

}

else {

Track leftTrack = trackMap.get(leftSymbol),

rightTrack = trackMap.get(rightSymbol);

if ((leftTrack == null) && middleCode.isLive(leftSymbol)) {

middleCode.setLoad(Track.Left);

leftTrack = createTrack(leftSymbol, trackMap, trackSet);

}

if ((rightTrack == null) && middleCode.isLive(rightSymbol)) {

middleCode.setLoad(Track.Right);

rightTrack = createTrack(rightSymbol, trackMap, trackSet);

}

if (leftTrack != null) {

leftTrack.addCode(Track.Left, line, leftSymbol.getType());

}

if (rightTrack != null) {

rightTrack.addCode(Track.Right, line, rightSymbol.getType());

}

if ((leftTrack == null) && (rightTrack == null)) {

int leftCount = middleCode.getCount(leftSymbol),

rightCount = middleCode.getCount(leftSymbol);

if (leftCount >= rightCount) {

middleCode.setLoad(Track.Left);

leftTrack = createTrack(leftSymbol, trackMap, trackSet);

leftTrack.addCode(Track.Left, line, leftSymbol.getType());

}

else {

middleCode.setLoad(Track.Right);

rightTrack = createTrack(rightSymbol, trackMap, trackSet);

rightTrack.addCode(Track.Right,line,rightSymbol.getType());

}

}

}

}

}

break;

case BinaryAdd:

case BinarySubtract:

case BitwiseIOr:

case BitwiseXOr:

case BitwiseAnd: {

Symbol resultSymbol = (Symbol) operand1,

leftSymbol = (Symbol) operand2,

rightSymbol = (Symbol) operand3;

if (resultSymbol.getType().isIntegralOrPointer()) {

Track leftTrack = trackMap.get(leftSymbol);

if (leftTrack == null) {

middleCode.setLoad(Track.Left);

leftTrack = createTrack(leftSymbol, trackMap, trackSet);

tryLoadAddressSymbol(Track.LeftAddress, line,

leftSymbol, trackMap);

}

leftTrack.addCode(Track.Left, line, leftSymbol.getType());

if (!rightSymbol.hasValue()) {

Track rightTrack = trackMap.get(rightSymbol);

if (rightTrack != null) {

rightTrack.addCode(Track.Right, line, rightSymbol.getType());

}

else {

if (middleCode.isLive(rightSymbol)) {

middleCode.setLoad(Track.Right);

rightTrack = createTrack(rightSymbol, trackMap, trackSet);

rightTrack.addCode(Track.Right,line,rightSymbol.getType());

}

tryLoadAddressSymbol(Track.RightAddress, line,

rightSymbol, trackMap);

}

}

trackMap.remove(leftSymbol);

Symbol rightAddressSymbol = rightSymbol.getAddressSymbol();

if ((rightAddressSymbol != null) &&

leftTrack == trackMap.get(rightAddressSymbol)) {

trackMap.remove(rightAddressSymbol);

}

trackMap.put(resultSymbol, leftTrack);

}

}

break;

case LeftShift:

case RightShift: {

Symbol resultSymbol = (Symbol) operand1,

leftSymbol = (Symbol) operand2,

rightSymbol = (Symbol) operand3;

Track leftTrack = trackMap.get(leftSymbol),

rightTrack = trackMap.get(rightSymbol);

if (leftTrack == null) {

middleCode.setLoad(Track.Left);

leftTrack = createTrack(leftSymbol, trackMap, trackSet);

tryLoadAddressSymbol(Track.LeftAddress,line,leftSymbol,trackMap);

}

leftTrack.addCode(Track.Left, line, leftSymbol.getType());

trackMap.remove(leftSymbol);

trackMap.put(resultSymbol, leftTrack);

if (!rightSymbol.hasValue()) {

leftTrack.setNotCL(true);

saveTemporarySymbol(middleCode,rightSymbol,Register.dl,trackMap); // XXX

if (rightTrack != null) {

if (rightTrack.getRegister() == null) {

rightTrack.setRegister(Register.cl);

}

rightTrack.addCode(Track.Right, line, rightSymbol.getType());

}

else {

if (middleCode.isLive(rightSymbol)) {

middleCode.setLoad(Track.Right);

rightTrack = createTrack(rightSymbol, trackMap, trackSet);

rightTrack.addCode(Track.Right, line, rightSymbol.getType());

}

tryLoadAddressSymbol(Track.RightAddress, line,

rightSymbol, trackMap);

}

}

}

break;

case Multiply:

case Divide:

case Modulo: {

Symbol resultSymbol = (Symbol) operand1,

leftSymbol = (Symbol) operand2,

rightSymbol = (Symbol) operand3;

if (resultSymbol.getType().isIntegralOrPointer()) {

int typeSize = leftSymbol.getType().getSize();

Register axRegister = LeftMultiplyMap.get(typeSize);

saveTemporarySymbol(middleCode, leftSymbol, axRegister,trackMap);

Track leftTrack = trackMap.get(leftSymbol);

if (leftTrack != null) {

if (!leftTrack.isEmpty()) {

int lastTrackLine = leftTrack.getLastLine();

if (lastTrackLine <= m\_axLog) {

middleCode.setLoad(Track.Left);

leftTrack = createTrack(leftSymbol, trackMap, trackSet);

m\_middleCodeList.get(lastTrackLine).setSave();

}

}

Register trackRegister = leftTrack.getRegister();

if (trackRegister == null) {

leftTrack.setRegister(axRegister);

leftTrack.addCode(Track.Left, line, axRegister);

}

else {

leftTrack.addCode(Track.Left, line, trackRegister);

}

}

else {

middleCode.setLoad(Track.Left);

leftTrack = createTrack(leftSymbol, trackMap, trackSet);

tryLoadAddressSymbol(Track.LeftAddress, line,

leftSymbol, trackMap);

leftTrack.addCode(Track.Left, line, axRegister);

}

trackMap.remove(leftSymbol);

Track rightTrack = trackMap.get(rightSymbol);

if (rightTrack != null) {

rightTrack.addCode(Track.Right, line, rightSymbol.getType());

}

else {

tryLoadAddressSymbol(Track.RightAddress, line,

rightSymbol, trackMap);

}

Register dxRegister = ClearMultiplyMap.get(typeSize);

if (!((leftTrack != null) &&

ObjectCode.registerOverlap(dxRegister, leftTrack.getRegister())) &&

!((rightTrack != null) &&

ObjectCode.registerOverlap(dxRegister, rightTrack.getRegister()))) {

saveTemporarySymbol(middleCode, null, dxRegister, trackMap);

Track clearTrack = new Track(leftSymbol.getType());

trackSet.add(clearTrack);

clearTrack.addCode(Track.Clear, line, dxRegister);

}

Track resultTrack = createTrack(resultSymbol, trackMap, trackSet);

Register resultRegister =

ResultMultiplyMap.get(new Pair<>(operator, typeSize));

resultTrack.setRegister(resultRegister);

trackMap.put(resultSymbol, resultTrack);

}

m\_axLog = line;

// left ax, clear dx, right bx

// mul bx

}

break;

case LoadToRegister: { // load\_register(bx, a \* b);

Symbol symbol = (Symbol) operand2;

if (!symbol.hasValue()) {

Register register = (Register) operand1;

middleCode.setRegister(Track.Left, register);

Track track = trackMap.get(symbol);

if (track != null) {

if (track.getRegister() == null) {

track.addCode(Track.Left, line, register);

}

else {

track.addCode(Track.Left, line, symbol.getType());

}

}

else {

middleCode.setLoad(Track.Left);

track = createTrack(symbol, trackMap, trackSet);

track.addCode(Track.Left, line, register);

tryLoadAddressSymbol(Track.LeftAddress, line, symbol, trackMap);

}

}

}

break;

case SaveFromRegister: {

Symbol symbol = (Symbol) operand1;

trackMap.remove(symbol);

Track track = createTrack(symbol, trackMap, trackSet);

Register register = (Register) operand2;

track.addCode(Track.Left, line, register);

}

break;

case ClearRegisters: {

trackMap.clear();

}

break;

case LoadStatusByte:

break;

case SaveFromFlagByte: {

Symbol symbol = (Symbol) operand1;

saveTemporarySymbol(middleCode, null, Register.ax, trackMap);

Track track = createTrack(symbol, trackMap, trackSet);

track.addCode(Track.Left, line, Register.al);

middleCode.setSave();

}

break;

case IntegralToIntegral: {

Symbol leftSymbol = (Symbol) operand2,

rightSymbol = (Symbol) operand3;

Track fromTrack = trackMap.get(rightSymbol);

boolean set = false;

tryLoadAddressSymbol(Track.RightAddress, line, rightSymbol, trackMap);

if (fromTrack == null) {

middleCode.setLoad(Track.Right);

fromTrack = createTrack(rightSymbol, trackMap, trackSet);

fromTrack.addCode(Track.Right, line, rightSymbol.getType());

set = true;

}

if (leftSymbol.getType().getSize() != rightSymbol.getType().getSize()) {

fromTrack.addCode(Track.Left, line, leftSymbol.getType());

if (!set) {

fromTrack.addCode(Track.Right, line, rightSymbol.getType());

}

}

trackMap.put(leftSymbol, fromTrack);

}

break;

case IntegralToFloating: {

Symbol fromSymbol = (Symbol) operand3;

Track fromTrack = trackMap.get(fromSymbol);

if (fromTrack != null) {

fromTrack.addCode(Track.Right, line);

}

}

break;

case FloatingToIntegral:

break;

}

}

}

private void tryLoadAddressSymbol(int position, int line, Symbol symbol,

Map<Symbol,Track> trackMap) {

Symbol addressSymbol = symbol.getAddressSymbol();

if (addressSymbol != null) {

Track addressTrack = trackMap.get(addressSymbol);

if (addressTrack != null) {

addressTrack.addCode(position, line);

m\_middleCodeList.get(line).setAddressOffset(position, symbol.getAddressOffset());

}

}

}

private void mustLoadAddressSymbol(int position, int line, Symbol symbol, MiddleCode middleCode,

Map<Symbol,Track> trackMap, Set<Track> trackSet) {

Symbol addressSymbol = symbol.getAddressSymbol();

if (addressSymbol != null) {

Track addressTrack = trackMap.get(addressSymbol);

if (addressTrack == null) {

middleCode.setLoad(position);

addressTrack = createTrack(addressSymbol, trackMap, trackSet);

}

middleCode.setAddressOffset(position, symbol.getAddressOffset());

addressTrack.addCode(position, line);

addressTrack.setPointer();

}

}

private void saveTemporarySymbol(MiddleCode middleCode, Symbol saveSymbol,

Register register, Map<Symbol,Track> trackMap) {

Set<Pair<Symbol,Register>> axPairSet =

extractSymbolSet(register, trackMap);

for (Pair<Symbol,Register> pair : axPairSet) {

Symbol symbol = pair.getFirst();

if (symbol.isTemporary() && middleCode.isLive(symbol)) {

Assert.error(middleCode.axPair() == null);

middleCode.axSetPair(pair);

if (symbol.getOffset() == 0) {

//symbol.addToSymbolTable();

Main.CurrentTable.addSymbol(symbol);

}

}

if (!symbol.equals(saveSymbol)) {

trackMap.remove(symbol);

}

}

}

private Set<Pair<Symbol,Register>> extractSymbolSet(Register register,

Map<Symbol,Track> trackMap){

Set<Pair<Symbol,Register>> resultSet = new ListSet<>();

for (Map.Entry<Symbol,Track> entry : trackMap.entrySet()) {

Symbol symbol = entry.getKey();

Track track = entry.getValue();

Register trackRegister = track.getRegister();

if (ObjectCode.registerOverlap(register, trackRegister)) {

resultSet.add(new Pair<>(symbol, trackRegister));

}

}

return resultSet;

}

}

## Register Allocation

Graph coloring is a NP-complete problem, which means that there is no known algorithm that works on polynomial time. To be sure that we have found the best solution, we have to exam every possible solution, which would require an unrealistic amount of time. In this book we perform a deep search, where we sort the vertices into a list and for each vertices try to found a register not already allocated by any of its neighbors. If we cannot find such register we backtrack and try another combination. If we finally find a solution where each vertex is mapped to a register not mapped by any of its neighbors, we have found an optimal solution.

If the deep search does not found such a solution, we have to do something about the graph. In this book we remove one edge and try the deep search again. In this way we continue to remove edges until we have found a solution. The benefit of removing edges is that the likeliness of founding a solution increases for each removed edge. The drawback is that each removed edge means that two tracks have to share we one register and that we have to add store and load instruction for the code to work. When we need to remove an edge we sort the edges by counting the number of load and store instructions necessary to add as compensation for the removal of the edge. There is no guaranty that this method generates the optimal solution but since shortage of registers only occur on rare occasions it shall be good enough.

RegisterAllocator.java

package c\_compiler;

import java.util.\*;

public class RegisterAllocator {

public RegisterAllocator(List<MiddleCode> middleCodeList,

Set<Track> trackSet) {

Graph<Track> totalTrackGraph = new Graph<Track>(trackSet);

generateEdgeSet(totalTrackGraph);

Set<Graph<Track>> trackGraphSet = totalTrackGraph.partitionate();

for (Graph<Track> trackGraph : trackGraphSet) {

checkColorEdge(trackGraph);

generateMatch(middleCodeList, trackGraph);

}

}

private void checkColorEdge(Graph<Track> trackGraph) {

for (UnorderedPair<Track,Track> pair : trackGraph.getEdgeSet()) {

Track track1 = pair.getFirst(), track2 = pair.getSecond();

Assert.error(!ObjectCode.registerOverlap(track1.getRegister(),

track2.getRegister()),

"register overlap " + track1.getRegister() +

" " + track2.getRegister());

}

}

private void generateEdgeSet(Graph<Track> trackGraph) {

Set<Track> trackSet = trackGraph.getVertexSet();

for (Track track1 : trackSet) {

for (Track track2 : trackSet) {

if (!track1.equals(track2) && track1.doesOverlap(track2)) {

trackGraph.addEdge(track1, track2);

}

}

}

}

//Track x;

private void generateMatch(List<MiddleCode> middleCodeList,

Graph<Track> trackGraph) {

List<Track> trackList = new LinkedList<>(trackGraph.getVertexSet());

for (Pair<Track,Track> edge : trackGraph.getEdgeSet()) {

Assert.error(((edge.getFirst().getRegister() == null) &&

(edge.getFirst().getRegister() == null)) ||

(edge.getFirst().getRegister() !=

edge.getSecond().getRegister()));

}

try {

listSearch(trackList, trackGraph);

Assert.error("Out of registers");

}

catch (FoundColorMatchException exception) {

for (Track track : trackGraph.getVertexSet()) {

Assert.error(track.getRegister() != null);

}

for (Pair<Track,Track> edge : trackGraph.getEdgeSet()) {

Assert.error(edge.getFirst().getRegister() != edge.getSecond().getRegister());

}

}

for (Track track : trackGraph.getVertexSet()) {

track.generate(middleCodeList, track.getRegister());

}

}

private void listSearch(List<Track> trackList, Graph<Track> trackGraph)

throws FoundColorMatchException {

if (trackList.isEmpty()) {

throw (new FoundColorMatchException());

}

else {

Track track = trackList.remove(0);

if (track.getRegister() != null) {

listSearch(trackList, trackGraph);

}

else {

Set<Register> possibleSet = getPossibleSet(track, track.getNotCL());

Set<Track> neighbourSet = trackGraph.getNeighbourSet(track);

for (Register possibleRegister : possibleSet) {

boolean match = true;

for (Track neighbourTrack : neighbourSet) {

Register neighbourRegister = neighbourTrack.getRegister();

if (ObjectCode.registerOverlap(possibleRegister, neighbourRegister)) {

match = false;

break;

}

}

if (match) {

//System.out.println(indent + "match\n\n");

track.setRegister(possibleRegister);

listSearch(trackList, trackGraph);

}

}

//System.out.println();

track.setRegister(null);

}

trackList.add(0, track);

}

}

/\*private boolean matchNeighbourSet(Register register,

Set<Track> neighbourSet) {

for (Track neighbourTrack : neighbourSet) {

if (ObjectCode.registerOverlap(register, neighbourTrack.getRegister())){

return false;

}

}

return true;

}\*/

public static Set<Register> m\_pointerRegisterSet =

new ListSet<>(Arrays.asList(Register.si, Register.di, Register.bx));

private static Set<Register> m\_shortRegisterSet =

new ListSet<>(Arrays.asList(Register.al, Register.ah, Register.bl,

Register.bh, Register.cl, Register.ch,

Register.dl, Register.dh)),

m\_normalRegisterSet =

new ListSet<>(Arrays.asList(Register.ax, Register.bx,

Register.cx, Register.dx));

private static Set<Register> getPossibleSet(Track track, boolean notDL) {

Set<Register> possibleSet = new ListSet<>();

if (track.isPointer()) {

possibleSet.addAll(m\_pointerRegisterSet);

if (Main.FuncSymbol.getType().isEllipse()) {

possibleSet.remove(ObjectCode.EllipseRegister);

}

}

else if (track.getMaxSize() == Type.ShortSize) {

possibleSet.addAll(m\_shortRegisterSet);

if (notDL) {

possibleSet.remove(Register.dl);

}

}

else {

possibleSet.addAll(m\_normalRegisterSet);

if (notDL) {

possibleSet.remove(Register.dx);

}

}

return possibleSet;

}

}

# Object Code Generation

The object code generation[[8]](#footnote-8) is the …

## The Object Operator

ObjectOperator.java

package c\_compiler;

public enum ObjectOperator {add, add\_byte, add\_dword, add\_word,

and, and\_byte, and\_dword, and\_word,

cmp, cmp\_byte, cmp\_dword, cmp\_word,

dec, dec\_byte, dec\_dword, dec\_word,

div, div\_byte, div\_dword, div\_word,

fabs, faddp, fadd, fchs, fcompp,

fdivp, fdivrp, fdivr, fdiv,

fild\_dword, fild\_word, finit,

fistp\_dword, fistp\_word, fist\_dword,

fist\_word, fld1, fldcw, fldz,

fld\_dword, fld\_qword, fmulp, fmul,

fstcw, fstp\_dword, fstp\_qword,

fstsw, fst\_dword, fst\_qword, fsubp,

fsubrp, fsubr, fsub, ftst, fwait,

idiv, idiv\_byte, idiv\_dword,

idiv\_word, imul, imul\_byte,

imul\_dword, imul\_word, inc,

inc\_byte, inc\_dword, inc\_word,

interrupt, jae, ja, jbe, jb, jc, je,

jge, jg, jle, jl, jmp, short\_jmp,

long\_jmp, jnc, jne, jnz, jz, lahf,

mov, mov\_byte, mov\_dword, mov\_word,

mul, mul\_byte, mul\_dword, mul\_word,

neg, neg\_byte, neg\_dword, neg\_word,

nop, not, not\_byte, not\_dword,

not\_word, or, or\_byte, or\_dword,

or\_word, sahf, shl, shr, sub,

sub\_byte, sub\_dword, sub\_word, xor,

xor\_byte, xor\_dword, xor\_word, call,

empty, label, label2, comment};

## The Object Code

ObjectCode.java

package c\_compiler;

import java.util.\*;

import java.lang.reflect.\*;

public class ObjectCode {

public static final int CallSize = 3;

public static final int RelationSize = 2;

public static final int LongJumpSize = 3;

public static final int ShortJumpSize = 2;

public static final int JumpDifference = LongJumpSize - ShortJumpSize;

public static final byte NopOperator = -112;

public static final byte ShortJumpOperator = -21;

public static final byte LongJumpOperator = -23;

public static final int MainInitSize = 31;

public static final int MainInitArgsSize = 74;

public static final Register CallRegister = Register.ax;

public static final Register FrameRegister = Register.bp;

public static final Register EllipseRegister = Register.di;

public static final Register ReturnValueRegister = Register.bx;

public static final Register ReturnPointerRegister = Register.bx;

public static final Register ExitValueRegister = Register.al;

public static final Register ShiftRegister = Register.cl;

public static final Register MultiplyRegister = Register.ax;

public static final Register MultiplyResult = Register.ax;

public static final Register DivideResult = Register.ah;

public static final Register ModuloResult = Register.al;

public static final Register SourceRegister = Register.dx;

public static final Register TargetRegister = Register.dx;

public static final Register MemoryCount = Register.ax;

public static final Register MemoryValue = Register.bl;

private ObjectOperator m\_operator;

private Object m\_operandArray[] = new Object[3];

//private Object m\_operandArray[0], m\_operandArray[1], m\_operandArray[2];

private String m\_returnLabel;

private boolean m\_longToShort = false;

public boolean isLongToShort() {

return m\_longToShort;

}

public void setLongToShort() {

m\_longToShort = true;

}

public ObjectCode(ObjectOperator operator, Object operand1,

Object operand2, Object operand3) {

this(operator, operand1, operand2, operand3, null);

}

public ObjectCode(ObjectOperator operator, Object operand1,

Object operand2, Object operand3, String returnLabel) {

m\_operator = operator;

m\_operandArray[0] = operand1;

m\_operandArray[1] = operand2;

m\_operandArray[2] = operand3;

m\_returnLabel = returnLabel;

optimizeIncrement();

}

public static final Map<Pair<ObjectOperator,Integer>,ObjectOperator>

m\_incrementMap = new ListMap<>();

static {

m\_incrementMap.put(new Pair<>(ObjectOperator.add, 1),

ObjectOperator.inc);

m\_incrementMap.put(new Pair<>(ObjectOperator.add, -1),

ObjectOperator.dec);

m\_incrementMap.put(new Pair<>(ObjectOperator.sub, 1), ObjectOperator.dec);

m\_incrementMap.put(new Pair<>(ObjectOperator.sub, -1),

ObjectOperator.inc);

}

private void optimizeIncrement() {

if ((m\_operator == ObjectOperator.add) ||

(m\_operator == ObjectOperator.sub)) {

if ((m\_operandArray[0] instanceof Register) &&

((m\_operandArray[1] != null) &&

(m\_operandArray[1].equals(1) || m\_operandArray[1].equals(-1))) &&

(m\_operandArray[2] == null)) {

m\_operator =

m\_incrementMap.get(new Pair<>(m\_operator, m\_operandArray[1]));

m\_operandArray[1] = null;

}

else if ((m\_operandArray[0] instanceof Register) &&

(m\_operandArray[1] instanceof Integer) &&

((m\_operandArray[2] != null) &&

(m\_operandArray[2].equals(1) ||

m\_operandArray[2].equals(-1)))) {

m\_operator =

m\_incrementMap.get(new Pair<>(m\_operator, m\_operandArray[2]));

m\_operandArray[2] = null;

}

}

}

public String getReturnLabel() {

return m\_returnLabel;

}

public ObjectOperator getOperator() {

return m\_operator;

}

public void setOperator(ObjectOperator operator) {

m\_operator = operator;

}

public Object getOperand(int index) {

return m\_operandArray[index - 1];

}

public void setOperand(int index, Object operand) {

m\_operandArray[index - 1] = operand;

}

public boolean isRelation() {

switch (m\_operator) {

case je:

case jne:

case jl:

case jle:

case jg:

case jge:

case jb:

case jbe:

case ja:

case jae:

case jc:

case jnc:

return true;

default:

return false;

}

}

public boolean isJump() {

return isJump(m\_operator);

}

public boolean isCallRegular() {

return isCallRegular(m\_operator);

}

public boolean isCallRegister() {

return isCallRegister(m\_operator);

}

public boolean isCallMemory() {

return isCallMemory(m\_operator);

}

public boolean isCallAuto() {

return isCallAuto(m\_operator);

}

public boolean isCallStatic() {

return isCallStatic(m\_operator);

}

public boolean isJump(ObjectOperator operator) {

return ((m\_operator == ObjectOperator.short\_jmp) ||

(m\_operator == ObjectOperator.long\_jmp)) &&

!(m\_operandArray[0] instanceof Register);

}

// call test Integer null null

public boolean isCallRegular(ObjectOperator operator) {

return (m\_operator == ObjectOperator.call) &&

(m\_operandArray[0] instanceof String) &&

(m\_operandArray[1] == null) &&

(m\_operandArray[2] == null);

}

// call ax Register null null

public boolean isCallRegister(ObjectOperator operator) {

return (m\_operator == ObjectOperator.call) &&

(m\_operandArray[0] instanceof Register) &&

(m\_operandArray[1] == null) &&

(m\_operandArray[2] == null);

}

public boolean isCallMemory(ObjectOperator operator) {

return isCallAuto(operator) || isCallStatic(operator);

}

// call [bp + 10] Register Integer null

public boolean isCallAuto(ObjectOperator operator) {

return (m\_operator == ObjectOperator.call) &&

(m\_operandArray[0] instanceof Register) &&

(m\_operandArray[1] instanceof Integer) &&

(m\_operandArray[2] == null);

}

// call [1000] null Integer null

public boolean isCallStatic(ObjectOperator operator) {

return (m\_operator == ObjectOperator.call) &&

(m\_operandArray[0] == null) &&

(m\_operandArray[1] instanceof Pair) &&

(m\_operandArray[2] == null);

}

public boolean isRegisterLoad(int position) {

return ((position == 1) && (m\_operandArray[0] instanceof Register) &&

((m\_operandArray[1] instanceof Register) ||

(m\_operandArray[1] instanceof Integer)));

}

public boolean isRegisterSave(int position) {

return ((position == 3) && (m\_operandArray[2] instanceof Register));

}

public static boolean registerOverlap(Register register1,

Register register2) {

if ((register1 == null) || (register2 == null)) {

return false;

}

if (register1.equals(register2)) {

return true;

}

String name1 = register1.name(), name2 = register2.name();

name1 = (name1.length() == 3) ? name1.substring(1) : name1;

name2 = (name2.length() == 3) ? name2.substring(1) : name2;

if ((name1.contains("h") && name2.contains("l")) ||

(name1.contains("l") && name2.contains("h"))) {

return false;

}

name1 = (name1.contains("h") || name1.contains("l") ||

name1.contains("x")) ? name1.substring(0, 1) : name1;

name2 = (name2.contains("h") || name2.contains("l") ||

name2.contains("x")) ? name2.substring(0, 1) : name2;

return name1.equals(name2);

}

public static Register registerToSize(Register register, int size) {

switch (size) {

case Type.ShortSize:

return Register.valueOf(register.name().replace("x", "l").

replace("e", ""));

case Type.IntegerSize:

return Register.valueOf(register.name().replace("h", "x").

replace("l", "x").replace("e", ""));

case Type.LongSize:

return Register.valueOf("e" + register.name().replace("l", "x").

replace("e", ""));

}

return null;

}

public static int registerSize(Register register) {

String name = register.name();

if (name.contains("e")) {

return c\_compiler.Type.LongSize;

}

else if (name.contains("l") || name.contains("h")) {

return c\_compiler.Type.ShortSize;

}

else {

return c\_compiler.Type.IntegerSize;

}

}

public static int operatorSize(ObjectOperator operator) {

if (operator.name().contains("\_byte")) {

return Type.CharSize;

}

else if (operator.name().contains("\_word")) {

return Type.IntegerSize;

}

else if (operator.name().contains("\_dword")) {

return Type.LongSize;

}

Assert.error("operator size");

return 0;

}

public static ObjectOperator operatorSize(ObjectOperator operator,

int size) {

if (operator == ObjectOperator.interrupt) {

return ObjectOperator.interrupt;

}

switch (size) {

case Type.CharSize:

return ObjectOperator.valueOf(operator.name() + "\_byte");

case Type.IntegerSize:

return ObjectOperator.valueOf(operator.name() + "\_word");

case Type.LongSize:

return ObjectOperator.valueOf(operator.name() + "\_dword");

default:

return null;

}

}

public static int operandSize(Object operand) {

if (operand instanceof Pair) {

return Type.PointerSize;

}

else if (operand instanceof Integer) {

int value = (int) operand;

return (value == 0) ? 0 : ((value < 128) ? 1 : 2);

}

return 0;

}

public static int valueSize(int value) {

if (value == 0) {

return 0;

}

else if (value < 256) {

return 1;

}

else if (value < 65536) {

return 2;

}

else {

return 4;

}

}

public int byteSizeCompare(int registerSize, int value, boolean signed) {

switch (registerSize) {

case 1:

if ((value >= 128) && (value <= 255)) {

value -= 256;

}

break;

case 2:

if ((value >= 32768) && (value <= 65535)) {

value -= 65536;

}

break;

}

int size;

if (value == 0) {

size = 0;

}

else if ((value >= -128) && (value <= 127)) {

size = 1;

}

else if ((value >= -32768) && (value <= 32767)) {

size = 2;

}

else {

size = 4;

}

return Math.min(size, valueSize());

}

public int valueSize() {

if (m\_operator == ObjectOperator.long\_jmp) {

return c\_compiler.Type.PointerSize;

}

else if (isRelation()) {

int address = (Integer) m\_operandArray[0];

return ((address >= -128) && (address <= 127)) ? 1 : 2;

}

else if (m\_operator == ObjectOperator.interrupt) {

return c\_compiler.Type.ShortSize;

}

else if (m\_operator.name().contains("byte")) {

return c\_compiler.Type.ShortSize;

}

else if (m\_operator.name().contains("dword")) {

return c\_compiler.Type.LongSize;

}

else if (m\_operator.name().contains("word")) {

return c\_compiler.Type.IntegerSize;

}

else {

return registerSize((Register) m\_operandArray[0]);

}

}

// ------------------------------------------------------------------------

// mov [global1}, global2

public List<Byte> byteList() {

ObjectOperator operator = m\_operator;

if ((operator == ObjectOperator.empty) ||

(operator == ObjectOperator.label) ||

(operator == ObjectOperator.comment)) {

return (new LinkedList<>());

}

else if (isCallRegular()) {

return lookupByteArray(ObjectOperator.jmp, c\_compiler.Type.PointerSize,

null, null);

}

else if (isCallRegister()) {

Register register = (Register) m\_operandArray[0];

return lookupByteArray(ObjectOperator.jmp, register, null, null);

}

else {

if (isCallMemory()) {

operator = ObjectOperator.jmp;

}

Object fillParam1 = (m\_operandArray[0] instanceof Pair) ?

((Pair) m\_operandArray[0]).getSecond() : m\_operandArray[0],

fillParam2 = (m\_operandArray[1] instanceof Pair) ?

((Pair) m\_operandArray[1]).getSecond() : m\_operandArray[1],

fillParam3 = (m\_operandArray[2] instanceof Pair) ?

((Pair) m\_operandArray[2]).getSecond() : m\_operandArray[2];

int size1 = (m\_operandArray[0] instanceof Pair) ?

Type.PointerSize : operandSize(m\_operandArray[0]),

size2 = (m\_operandArray[1] instanceof Pair) ?

Type.PointerSize : operandSize(m\_operandArray[1]),

size3 = (m\_operandArray[2] instanceof Pair) ?

Type.PointerSize : operandSize(m\_operandArray[2]);

if ((operator == ObjectOperator.inc\_word) && (m\_operandArray[1] != null)

&& m\_operandArray[1].toString().contains("g\_outChars") &&

(m\_operandArray[0] == null) && (m\_operandArray[2] == null)) {

size3 = 0;

}

if (operator.name().startsWith("cmp\_") &&

(m\_operandArray[1] instanceof Integer)) {

size3 = Math.max(1, size3);

}

if (isJump()) {

size1 = (operator == ObjectOperator.short\_jmp) ?

Type.CharSize : Type.PointerSize;

operator = ObjectOperator.jmp;

}

if (isRelation()) {

size1 = Type.CharSize;

}

Object byteParam1 = (fillParam1 instanceof Integer) ? size1 : fillParam1,

byteParam2 = (fillParam2 instanceof Integer) ? size2 : fillParam2,

byteParam3 = (fillParam3 instanceof Integer) ? size3 : fillParam3;

if ((operator == ObjectOperator.mov) &&

(m\_operandArray[0] instanceof Register) &&

(m\_operandArray[1] instanceof Integer) &&

(m\_operandArray[2] == null)) {

byteParam2 = size2 = registerSize((Register) m\_operandArray[0]);

}

if ((operator == ObjectOperator.add) &&

(m\_operandArray[0] == Register.eax) &&

(m\_operandArray[1] instanceof Integer) &&

(m\_operandArray[2] == null)) {

byteParam2 = size2 = registerSize((Register) m\_operandArray[0]);

}

if ((operator == ObjectOperator.and) &&

(m\_operandArray[0] instanceof Register) &&

(m\_operandArray[1] instanceof Integer) &&

!m\_operandArray[1].toString().equals("1") &&

(m\_operandArray[2] == null)) {

byteParam2 = size2 = registerSize((Register) m\_operandArray[0]);

}

if ((m\_operandArray[0] == null) && // mov word [100], 1

(m\_operandArray[1] instanceof Integer) &&

(m\_operandArray[2] == null)) {

byteParam2 = Type.PointerSize;

}

if ((operator == ObjectOperator.add\_word) &&

(m\_operandArray[0] == Register.bp) &&

(m\_operandArray[1] != null) &&

m\_operandArray[1].toString().equals("20") &&

(m\_operandArray[2] != null) &&

m\_operandArray[2].toString().equals("0")) {

size3 = Type.CharSize;

}

if (operator.name().startsWith("mov\_")) {

byteParam3 = size3 = ObjectCode.operatorSize(operator);

}

if (isRelation() || isJump()) {

byteParam2 = byteParam3 = null;

size2 = size3 = 0;

}

if ((m\_operandArray[0] instanceof Integer) &&

(m\_operandArray[1] instanceof Register) &&

(m\_operandArray[2] == null)) {

byteParam1 = ObjectCode.registerSize((Register) m\_operandArray[1]);

}

if (operator.name().contains("mov\_") &&

(m\_operandArray[1] instanceof Integer) &&

(m\_operandArray[2] instanceof Integer) &&

m\_operandArray[2].toString().equals("0")) {

size3 = operatorSize(operator);

}

if ((operator == ObjectOperator.jae) &&

(m\_operandArray[0] instanceof Integer) &&

m\_operandArray[0].toString().equals("127")) {

List<Byte> byteList = new LinkedList<>();

byteList.add((byte) 15);

byteList.add((byte) -125);

byteList.add((byte) 127);

byteList.add((byte) 0);

return byteList;

}

else if ((operator == ObjectOperator.je) &&

(m\_operandArray[0] instanceof Integer) &&

m\_operandArray[0].toString().equals("126")) {

List<Byte> byteList = new LinkedList<>();

byteList.add((byte) 15);

byteList.add((byte) -124);

byteList.add((byte) 126);

byteList.add((byte) 0);

return byteList;

}

else {

// shl ebx, [bp]

List<Byte> byteList =

lookupByteArray(operator, byteParam1, byteParam2, byteParam3);

loadByteList(byteList, byteList.size() - (size1 + size2 + size3),

size1, fillParam1);

loadByteList(byteList, byteList.size() - (size2 + size3), size2,

fillParam2);

loadByteList(byteList, byteList.size() - size3, size3, fillParam3);

return byteList;

}

}

}

private void loadByteList(List<Byte> byteList, int index,

int size, Object param) {

if (size > 0) {

int value = (Integer) param;

fillByteList(byteList, index, size, value);

}

}

public static void fillByteList(List<Byte> byteList, int index,

int size, int value) {

switch (size) {

case 0:

break;

case Type.CharSize:

byteList.set(index, (byte) value);

break;

case Type.IntegerSize:

byteList.set(index, (byte) value);

byteList.set(index + 1, (byte) (value >> 8));

break;

case Type.LongSize:

byteList.set(index, (byte) value);

byteList.set(index + 1, (byte) (value >> 8));

byteList.set(index + 2, (byte) (value >> 16));

byteList.set(index + 3, (byte) (value >> 24));

break;

}

}

public String toString() {

if (m\_operator == ObjectOperator.empty) {

return "";

}

else if (m\_operator == ObjectOperator.label) {

return ("\n" + m\_operandArray[0] + ":" +

((m\_operandArray[1] != null) ? ("\t; " + m\_operandArray[1]) : ""));

}

else if (m\_operator == ObjectOperator.comment) {

return ("\n;\t" + m\_operandArray[0]);

}

String operatorName = (m\_operator == ObjectOperator.call) ? "jmp" :

(m\_operator.name().endsWith("jmp") ? "jmp" :

(m\_operator == ObjectOperator.interrupt) ? "int" :

m\_operator.name().replace("\_", " "));

StringBuilder buffer = new StringBuilder("\t" + operatorName);

if (isCallRegular()) {

buffer.append(" " + toString(m\_operandArray[0]));

}

else if (isCallRegister()) {

buffer.append(" " + m\_operandArray[0]);

}

else if (isCallAuto()) {

buffer.append(" [" + toString(m\_operandArray[0]) +

withSign(m\_operandArray[1]) + "]");

}

else if (isCallStatic()) {

buffer.append(" [" + toString(m\_operandArray[1]) + "]");

}

else if (isRelation() || isJump()) {

buffer.append(" " + m\_operandArray[2]);

}

else if (m\_returnLabel != null) {

if (m\_operandArray[2] != null) {

buffer.append(" [" + toString(m\_operandArray[0]) +

withSign(m\_operandArray[1]) + "], " + m\_returnLabel);

}

else {

buffer.append(" " + toString(m\_operandArray[0]) + ", " +m\_returnLabel);

}

}

else if ((m\_operandArray[0] != null) && (m\_operandArray[1] != null) &&

(m\_operandArray[2] != null)) {

if (m\_operandArray[1] instanceof Integer) {

buffer.append(" [" + toString(m\_operandArray[0]) +

withSign(m\_operandArray[1]) + "], " +

toString(m\_operandArray[2]));

}

else {

buffer.append(" " + toString(m\_operandArray[0]) +

", [" + toString(m\_operandArray[1]) +

withSign(m\_operandArray[2]) + "]");

}

}

else if ((m\_operandArray[0] == null) && (m\_operandArray[1] != null) &&

(m\_operandArray[2] != null)) {

buffer.append(" [" + toString(m\_operandArray[1]) + "], " +

toString(m\_operandArray[2]));

}

else if ((m\_operandArray[0] != null) && (m\_operandArray[1] == null) &&

(m\_operandArray[2] != null)) {

buffer.append(" " + toString(m\_operandArray[0]) + ", [" +

toString(m\_operandArray[2]) + "]");

}

else if ((m\_operandArray[0] == null) && (m\_operandArray[1] != null) &&

(m\_operandArray[2] == null)) {

buffer.append(" [" + toString(m\_operandArray[1]) + "]");

}

else if ((m\_operator.name().equals("fstcw") ||

m\_operator.name().equals("fldcw")) &&

(m\_operandArray[0] != null) && (m\_operandArray[1] != null) &&

(m\_operandArray[2] == null)) {

buffer.append(" [" + toString(m\_operandArray[0]) +

withSign(m\_operandArray[1]) + "]");

}

else if (m\_operator.name().contains("\_") && (m\_operandArray[0] != null) &&

(m\_operandArray[1] != null) && (m\_operandArray[2] == null)) {

buffer.append(" [" + toString(m\_operandArray[0]) +

withSign(m\_operandArray[1]) + "]");

}

else if (!m\_operator.name().contains("\_") && (m\_operandArray[0] != null) &&

(m\_operandArray[1] != null) && (m\_operandArray[2] == null)) {

buffer.append(" " + toString(m\_operandArray[0]) + ", " +

toString(m\_operandArray[1]));

}

else if ((m\_operandArray[0] != null) && (m\_operandArray[1] == null) &&

(m\_operandArray[2] == null)) {

buffer.append(" " + toString(m\_operandArray[0]));

}

else if ((m\_operandArray[0] == null) && (m\_operandArray[1] == null) &&

(m\_operandArray[2] == null)) {

// Empty.

}

else {

Assert.error("object code toString");

}

buffer.append("\t; " + byteList().size());

return buffer.toString();

}

private String withSign(Object object) {

int value = (int) object;

if (value > 0) {

return " + " + value;

}

else if (value < 0) {

return " - " + (-value);

}

else {

return "";

}

}

private String toString(Object object) {

if (object instanceof Pair) {

Pair pair = (Pair) object;

String name = (String) pair.getFirst();

int offset = (int) pair.getSecond();

return name + withSign(offset);

}

else if (object instanceof Register) {

return ((Register) object).name();

}

else {

return object.toString();

}

}

public static List<Byte> lookupByteArray(ObjectOperator operator,

Object operand1, Object operand2, Object operand3){

try {

if ((operator == ObjectOperator.shl) ||

(operator == ObjectOperator.shr)) {

operand1 = (operand1 instanceof Integer) ? 0 : operand1;

operand2 = (operand2 instanceof Integer) ? 0 : operand2;

operand3 = (operand3 instanceof Integer) ? 0 : operand3;

}

Tuple<ObjectOperator,Object,Object,Object> tuple =

new Tuple<>(operator, operand1, operand2, operand3);

for (int index = 0; index < Main.ObjectCodeTableMapSize; ++index) {

String classPath =

"c\_compiler.object\_code\_table.ObjectCodeTable" + index;

Class classType = Class.forName(classPath);

Object classInst = classType.newInstance();

Field fieldType = classType.getField("ByteArrayMap");

Map<Tuple<ObjectOperator,Object,Object,Object>,byte[]> byteArrayMap =

(Map<Tuple<ObjectOperator,Object,Object,Object>,byte[]>)

fieldType.get(classInst);

byte[] byteArray = byteArrayMap.get(tuple);

if (byteArray != null) {

List<Byte> byteList = new LinkedList<>();

for (byte b : byteArray) {

byteList.add(b);

}

return byteList;

}

}

}

catch (Exception exception) {

exception.printStackTrace();

Assert.error(exception.getMessage());

}

return null;

}

}

### Object Code Generation

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Case | Operator | Operand 1 | Operand 2 | Operand 3 | Full Name | Example |
| 1 | mov | null | null | Integer | null |  |
| 2 | mov\_wordX | Register | Integer | null |  |  |
| 3 |  | Integer | Null | Null | Null | int 21 |
| 4 |  |  |  |  |  |  |
| 5 |  |  |  |  |  |  |
|  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |

## Object Code Generation

ObjectCodeGenerator.java

package c\_compiler;

import java.math.\*;

import java.util.\*;

public class ObjectCodeGenerator {

private List<MiddleCode> m\_middleCodeList;

private List<ObjectCode> m\_objectCodeList;

private int m\_stackSize = 0;

private static int m\_loopCount = 0;

private final Map<Symbol,Track> m\_trackMap = new ListMap<>();

private Map<Integer,Integer> m\_middleToObjectMap = new ListMap<>();

public ObjectCodeGenerator(List<MiddleCode> middleCodeList,

List<ObjectCode> objectCodeList) {

m\_middleCodeList = middleCodeList;

m\_objectCodeList = objectCodeList;

}

private void addObjectCode(ObjectOperator operator, Object operand1,

Object operand2, Object operand3) {

m\_objectCodeList.add(new ObjectCode(operator, operand1,

operand2, operand3, null));

}

private int addCountObjectCode(ObjectOperator operator, Object operand1,

Object operand2, Object operand3) {

ObjectCode objectCode = new ObjectCode(operator, operand1,

operand2, operand3, null);

m\_objectCodeList.add(objectCode);

return byteListSize(objectCode);

}

private void addObjectCode(ObjectOperator operator, Object operand1,

Object operand2, Object operand3, String returnLabel) {

m\_objectCodeList.add(new ObjectCode(operator, operand1, operand2,

operand3, returnLabel));

}

Track setPointerTrack(Symbol addressSymbol, Track track) {

if (!track.setPointer()) {

m\_trackMap.remove(addressSymbol);

track.setPointer();

}

return track;

}

public static Register baseRegister(Symbol symbol) {

Storage storage = symbol.getStorage();

if ((storage == Storage.Auto) || (storage == Storage.Register)) {

if (Main.FuncSymbol.getType().isEllipse() && !symbol.isParameter()) {

return ObjectCode.EllipseRegister;

}

else {

return ObjectCode.FrameRegister;

}

}

else {

return null;

}

}

public void generateObjectCodeList() {

for (int index = 0; index < m\_middleCodeList.size(); ++index) {

MiddleCode middleCode = m\_middleCodeList.get(index);

MiddleOperator middleOp = middleCode.getOperator();

m\_middleToObjectMap.put(index, m\_objectCodeList.size());

String middleText = middleCode.toString();

if (middleCode.getOperator() == MiddleOperator.FunctionStart) {

middleText += ", " + Main.CurrentTable.toString();

}

if (middleCode.isLeader()) {

middleText += " (leader)";

}

String label = Main.FuncSymbol.getUniqueName() +

((index > 0) ? (Main.SeparatorId + index) : "");

addObjectCode(ObjectOperator.label, label, middleText, null);

switch (middleOp) {

case Address:

case Dot:

generateMemory(middleCode);

break;

case Deref:

case Arrow:

case Index: {

Symbol resultSymbol = (Symbol) middleCode.getOperand(2);

if (resultSymbol.getType().isIntegralOrPointer()) {

generateIntegral(middleCode, index);

}

else {

generateMemory(middleCode);

}

}

break;

case FunctionStart:

case FunctionEnd:

case Call:

case Return:

case Goto:

case Empty:

generateFunction(middleCode, index);

break;

case FloatingToFloating:

case IntegralToIntegral:

case FloatingToIntegral:

generateConvert(middleCode, index);

break;

case Exit:

case LoadToRegister:

case SaveFromRegister:

case SaveFromFlagByte:

case JumpRegister:

case Interrupt:

generateIntegral(middleCode, index);

break;

case ClearRegisters:

break;

case PushFloat:

case PopFloat:

case TopFloat:

case CheckReturnValue1:

case CheckReturnValue2:

case CheckReturnValue3:

generateFloating(middleCode, index);

break;

case PrefixIncrement:

case PrefixDecrement:

case PostfixIncrement:

case PostfixDecrement: {

Symbol incSymbol = (Symbol) middleCode.getOperand(3);

if (incSymbol.getType().isIntegralOrPointer()) {

generateIntegral(middleCode, index);

}

else {

generateFloating(middleCode, index);

}

}

break;

default: {

Object operand2 = middleCode.getOperand(2);

if (operand2 == null) {

operand2 = middleCode.getOperand(1);

}

Symbol symbol = (operand2 instanceof Symbol) ?

((Symbol) operand2) : null;

if (symbol.getType().isIntegralOrPointer() ||

symbol.getType().isFunctionArrayOrString()) {

generateIntegral(middleCode, index);

}

else if (symbol.getType().isFloating()) {

generateFloating(middleCode, index);

}

else if (symbol.getType().isStructOrUnion()) {

generateStructure(middleCode, index);

}

}

break;

}

}

}

private void generateMemory(MiddleCode middleCode) {

MiddleOperator middleOp = middleCode.getOperator();

Object operand1 = middleCode.getOperand(1),

operand2 = middleCode.getOperand(2),

operand3 = middleCode.getOperand(3);

Register leftRegister = middleCode.getRegister(Track.Left),

leftAddressRegister = middleCode.getRegister(Track.LeftAddress),

rightRegister = middleCode.getRegister(Track.Right),

rightAddressRegister = middleCode.getRegister(Track.RightAddress);

int leftAddressOffset = middleCode.getAddressOffset(Track.LeftAddress),

rightAddressOffset = middleCode.getAddressOffset(Track.RightAddress);

switch (middleOp) {

case Deref:

case Arrow:

case Index: {

int memberOffset = (int) operand1;

Symbol leftSymbol = (Symbol) operand2,

rightSymbol = (Symbol) operand3;

if (middleCode.isLoad(Track.Right)) {

loadRegister(rightRegister, rightSymbol,

rightAddressRegister, rightAddressOffset);

}

if (leftSymbol.getType().isIntegralOrPointer()) {

if (middleCode.isSave()) {

addObjectCode(ObjectOperator.mov, leftRegister,

rightRegister, memberOffset);

saveRegister(leftRegister, leftSymbol,

leftAddressRegister, leftAddressOffset);

}

}

else if (leftSymbol.getType().isFloating()) {

if (middleCode.isRightValue()) {

pushFloating(leftSymbol,leftAddressRegister,leftAddressOffset);

}

}

}

break;

case Address: {

Symbol symbol = (Symbol) operand2;

if (rightRegister == null) {

loadAddressRegister(leftRegister, symbol, null);

}

}

break;

default:

Assert.error("default memory");

break;

}

}

private void generateFunction(MiddleCode middleCode, int index) {

MiddleOperator middleOp = middleCode.getOperator();

Object operand1 = middleCode.getOperand(1),

operand2 = middleCode.getOperand(2),

operand3 = middleCode.getOperand(3);

switch (middleOp) {

case FunctionStart:

if (Main.CheckStackHeap) {

String returnLabel =

Main.FuncSymbol.getUniqueName() + Main.SeparatorId + (index + 1);

addObjectCode(ObjectOperator.comment,

"Check Stack and Heap", null, null);

addObjectCode(ObjectOperator.mov, Register.ax,

Main.CurrentTable.getFunctionSize(), null);

addObjectCode(ObjectOperator.mov, Register.bx, ObjectCode.CallSize +

Type.PointerSize, null, returnLabel);

if (Main.FuncSymbol.getType().isEllipse()) {

addObjectCode(ObjectOperator.call,

Main.CheckStackHeapFunctionEllipse, null, null);

}

else {

addObjectCode(ObjectOperator.call,

Main.CheckStackHeapFunctionRegular, null, null);

}

}

break;

case Call: {

Symbol calleeSymbol = (Symbol) operand1;

int stackSize = (int) operand3;

int recordSize = Main.CurrentTable.getFunctionSize() + stackSize;

boolean callerEllipse = Main.FuncSymbol.getType().isEllipse(),

calleeEllipse = calleeSymbol.getType().isFunction() ?

calleeSymbol.getType().isEllipse() :

calleeSymbol.getType().getPointerType().isEllipse();

Type returnType = calleeSymbol.getType().isFunction() ?

calleeSymbol.getType().getReturnType() :

calleeSymbol.getType().getPointerType().getReturnType();

Register callRegister = middleCode.getRegister(Track.Left),

addressRegister = middleCode.getRegister(Track.LeftAddress);

int addressOffset = middleCode.getAddressOffset(Track.LeftAddress);

if (middleCode.isLoad(Track.Left)) {

loadRegister(callRegister, calleeSymbol,

addressRegister, addressOffset);

}

int byteSize = 0;

int returnLine = m\_objectCodeList.size();

String returnLabel =

Main.FuncSymbol.getUniqueName() + Main.SeparatorId + (index + 1);

Pair<Integer,Boolean> pair = (Pair) operand2;

int extraSize = pair.getFirst();

boolean useResult = pair.getSecond();

if (callerEllipse) {

// mov word [di + 10], 20

addObjectCode(ObjectOperator.mov\_word, ObjectCode.EllipseRegister,

recordSize + SymbolTable.ReturnAddressOffset, null,returnLabel);

byteSize +=

addCountObjectCode(ObjectOperator.mov,ObjectCode.EllipseRegister,

recordSize + SymbolTable.RegularFrameOffset,

ObjectCode.FrameRegister);

byteSize +=

addCountObjectCode(ObjectOperator.mov,ObjectCode.EllipseRegister,

recordSize + SymbolTable.EllipseFrameOffset,

ObjectCode.EllipseRegister);

if (returnType.isFloating()) {

byteSize +=

addCountObjectCode(ObjectOperator.mov\_byte, ObjectCode.EllipseRegister,

recordSize + SymbolTable.UseResultOffset,

useResult ? 1 : 0);

}

byteSize += addCountObjectCode(ObjectOperator.add, ObjectCode.EllipseRegister,

recordSize, null);

byteSize += addCountObjectCode(ObjectOperator.mov, ObjectCode.FrameRegister,

ObjectCode.EllipseRegister, null);

}

else {

addObjectCode(ObjectOperator.mov\_word, ObjectCode.FrameRegister,

recordSize + SymbolTable.ReturnAddressOffset, null, returnLabel);

byteSize += addCountObjectCode(ObjectOperator.mov, ObjectCode.FrameRegister,

recordSize + SymbolTable.RegularFrameOffset,

ObjectCode.FrameRegister);

byteSize += addCountObjectCode(ObjectOperator.mov, ObjectCode.FrameRegister,

recordSize + SymbolTable.EllipseFrameOffset,

ObjectCode.EllipseRegister);

if (returnType.isFloating()) {

byteSize += addCountObjectCode(ObjectOperator.mov\_byte, ObjectCode.FrameRegister,

recordSize + SymbolTable.UseResultOffset,

useResult ? 1 : 0);

}

byteSize += addCountObjectCode(ObjectOperator.add, ObjectCode.FrameRegister,

recordSize, null);

if (calleeEllipse) {

byteSize += addCountObjectCode(ObjectOperator.mov, ObjectCode.EllipseRegister,

ObjectCode.FrameRegister, null);

}

}

if (extraSize > 0) {

byteSize += addCountObjectCode(ObjectOperator.add, ObjectCode.EllipseRegister,

extraSize, null);

}

if (calleeSymbol.getType().isFunction()) {

byteSize += addCountObjectCode(ObjectOperator.call,

calleeSymbol.getUniqueName(), null, null);

}

else {

if (calleeSymbol.isAutoOrRegister()) {

calleeSymbol.setReferred(true);

calleeSymbol = Symbol.createOffsetSymbol(calleeSymbol);

if (callerEllipse && !calleeSymbol.isParameter()) {

calleeSymbol.setOffset(calleeSymbol.getOffset() -

(recordSize + extraSize));

}

else {

calleeSymbol.setOffset(calleeSymbol.getOffset() - recordSize);

}

}

byteSize +=

generateUnaryExpression(ObjectOperator.call, callRegister,

calleeSymbol, addressRegister, addressOffset);

}

ObjectCode returnCode = m\_objectCodeList.get(returnLine);

returnCode.setOperand(3, byteSize + Type.PointerSize);

}

break;

case Return: {

Register returnAddressRegister = middleCode.getRegister(Track.Left);

addObjectCode(ObjectOperator.mov, returnAddressRegister,

ObjectCode.FrameRegister, SymbolTable.ReturnAddressOffset);

addObjectCode(ObjectOperator.mov, ObjectCode.EllipseRegister,

ObjectCode.FrameRegister, SymbolTable.EllipseFrameOffset);

addObjectCode(ObjectOperator.mov, ObjectCode.FrameRegister,

ObjectCode.FrameRegister, SymbolTable.RegularFrameOffset);

addObjectCode(ObjectOperator.jmp, returnAddressRegister, null, null);

}

break;

case Goto:

addObjectCode(ObjectOperator.long\_jmp, null, null, operand1);

break;

}

}

// ------------------------------------------------------------------------

private static final Map<MiddleOperator,ObjectOperator>

m\_middleToIntegralTargetMap = new ListMap<>(),

m\_middleToFloatingTargetMap = new ListMap<>();

private static final Map<Pair<MiddleOperator,Boolean>,ObjectOperator>

m\_middleToSignedIntegralTargetMap = new ListMap<>();

private static final Map<ObjectOperator,ObjectOperator>

m\_inverseMap = new ListMap<>();

private static final Map<Pair<Integer,Integer>,Integer>

m\_maskMap = new ListMap<>();

private static final Map<Sort,ObjectOperator> m\_floatPushMap=new ListMap<>();

private static final Map<Sort,ObjectOperator> m\_floatTopMap =new ListMap<>(),

m\_floatPopMap =new ListMap<>();

static {

m\_middleToIntegralTargetMap.

put(MiddleOperator.Assign, ObjectOperator.mov);

m\_middleToIntegralTargetMap.

put(MiddleOperator.BitwiseNot,ObjectOperator.not);

m\_middleToIntegralTargetMap.

put(MiddleOperator.UnarySubtract, ObjectOperator.neg);

m\_middleToIntegralTargetMap.

put(MiddleOperator.PrefixIncrement, ObjectOperator.inc);

m\_middleToIntegralTargetMap.

put(MiddleOperator.PrefixDecrement, ObjectOperator.dec);

m\_middleToIntegralTargetMap.

put(MiddleOperator.PostfixIncrement, ObjectOperator.inc);

m\_middleToIntegralTargetMap.

put(MiddleOperator.PostfixDecrement, ObjectOperator.dec);

m\_middleToIntegralTargetMap.

put(MiddleOperator.BitwiseAnd, ObjectOperator.and);

m\_middleToIntegralTargetMap.

put(MiddleOperator.BitwiseIOr, ObjectOperator.or);

m\_middleToIntegralTargetMap.

put(MiddleOperator.BitwiseXOr, ObjectOperator.xor);

m\_middleToIntegralTargetMap.

put(MiddleOperator.LeftShift, ObjectOperator.shl);

m\_middleToIntegralTargetMap.

put(MiddleOperator.RightShift, ObjectOperator.shr);

m\_middleToIntegralTargetMap.

put(MiddleOperator.BinaryAdd, ObjectOperator.add);

m\_middleToIntegralTargetMap.

put(MiddleOperator.BinarySubtract, ObjectOperator.sub);

m\_middleToIntegralTargetMap.

put(MiddleOperator.Interrupt, ObjectOperator.interrupt);

m\_middleToSignedIntegralTargetMap.

put(new Pair<>(MiddleOperator.Multiply, true), ObjectOperator.imul);

m\_middleToSignedIntegralTargetMap.

put(new Pair<>(MiddleOperator.Multiply, false), ObjectOperator.mul);

m\_middleToSignedIntegralTargetMap.

put(new Pair<>(MiddleOperator.Divide, true), ObjectOperator.idiv);

m\_middleToSignedIntegralTargetMap.

put(new Pair<>(MiddleOperator.Divide, false), ObjectOperator.div);

m\_middleToSignedIntegralTargetMap.

put(new Pair<>(MiddleOperator.Modulo, true), ObjectOperator.idiv);

m\_middleToSignedIntegralTargetMap.

put(new Pair<>(MiddleOperator.Modulo, false), ObjectOperator.div);

m\_middleToSignedIntegralTargetMap.

put(new Pair<>(MiddleOperator.Equal, true), ObjectOperator.je);

m\_middleToSignedIntegralTargetMap.

put(new Pair<>(MiddleOperator.NotEqual, true), ObjectOperator.jne);

m\_middleToSignedIntegralTargetMap.

put(new Pair<>(MiddleOperator.LessThan, true), ObjectOperator.jl);

m\_middleToSignedIntegralTargetMap.

put(new Pair<>(MiddleOperator.LessThanEqual, true),ObjectOperator.jle);

m\_middleToSignedIntegralTargetMap.

put(new Pair<>(MiddleOperator.GreaterThan, true), ObjectOperator.jg);

m\_middleToSignedIntegralTargetMap.

put(new Pair<>(MiddleOperator.GreaterThanEqual,true),ObjectOperator.jge);

m\_middleToSignedIntegralTargetMap.

put(new Pair<>(MiddleOperator.Equal, false), ObjectOperator.je);

m\_middleToSignedIntegralTargetMap.

put(new Pair<>(MiddleOperator.NotEqual, false), ObjectOperator.jne);

m\_middleToSignedIntegralTargetMap.

put(new Pair<>(MiddleOperator.LessThan, false), ObjectOperator.jb);

m\_middleToSignedIntegralTargetMap.

put(new Pair<>(MiddleOperator.LessThanEqual,false),ObjectOperator.jbe);

m\_middleToSignedIntegralTargetMap.

put(new Pair<>(MiddleOperator.GreaterThan, false), ObjectOperator.ja);

m\_middleToSignedIntegralTargetMap.

put(new Pair<>(MiddleOperator.GreaterThanEqual,false),ObjectOperator.jae);

m\_middleToFloatingTargetMap.

put(MiddleOperator.UnarySubtract, ObjectOperator.fchs);

m\_middleToFloatingTargetMap.

put(MiddleOperator.BinaryAdd, ObjectOperator.fadd);

m\_middleToFloatingTargetMap.

put(MiddleOperator.BinarySubtract, ObjectOperator.fsub);

m\_middleToFloatingTargetMap.

put(MiddleOperator.Multiply, ObjectOperator.fmul);

m\_middleToFloatingTargetMap.

put(MiddleOperator.Divide, ObjectOperator.fdiv);

m\_middleToFloatingTargetMap.

put(MiddleOperator.PrefixIncrement, ObjectOperator.fadd);

m\_middleToFloatingTargetMap.

put(MiddleOperator.PrefixDecrement, ObjectOperator.fsub);

m\_middleToFloatingTargetMap.

put(MiddleOperator.PostfixIncrement, ObjectOperator.fadd);

m\_middleToFloatingTargetMap.

put(MiddleOperator.PostfixDecrement, ObjectOperator.fsub);

m\_middleToFloatingTargetMap.

put(MiddleOperator.Equal, ObjectOperator.je);

m\_middleToFloatingTargetMap.

put(MiddleOperator.NotEqual, ObjectOperator.jne);

m\_middleToFloatingTargetMap.

put(MiddleOperator.LessThan, ObjectOperator.ja);

m\_middleToFloatingTargetMap.

put(MiddleOperator.LessThanEqual, ObjectOperator.jae);

m\_middleToFloatingTargetMap.

put(MiddleOperator.GreaterThan, ObjectOperator.jb);

m\_middleToFloatingTargetMap.

put(MiddleOperator.GreaterThanEqual, ObjectOperator.jbe);

m\_inverseMap.put(ObjectOperator.je, ObjectOperator.jne);

m\_inverseMap.put(ObjectOperator.jne, ObjectOperator.je);

m\_inverseMap.put(ObjectOperator.jl, ObjectOperator.jge);

m\_inverseMap.put(ObjectOperator.jle, ObjectOperator.jg);

m\_inverseMap.put(ObjectOperator.jg, ObjectOperator.jle);

m\_inverseMap.put(ObjectOperator.jge, ObjectOperator.jl);

m\_inverseMap.put(ObjectOperator.ja, ObjectOperator.jbe);

m\_inverseMap.put(ObjectOperator.jae, ObjectOperator.jb);

m\_inverseMap.put(ObjectOperator.jb, ObjectOperator.jae);

m\_inverseMap.put(ObjectOperator.jbe, ObjectOperator.ja);

m\_maskMap.put(new Pair<>(1,2), 0x00FF);

m\_maskMap.put(new Pair<>(1,4), 0x000000FF);

m\_maskMap.put(new Pair<>(2,4), 0x0000FFFF);

m\_floatPushMap.put(Sort.Signed\_Int, ObjectOperator.fild\_word);

m\_floatPushMap.put(Sort.Unsigned\_Int, ObjectOperator.fild\_word);

m\_floatPushMap.put(Sort.Signed\_Long\_Int, ObjectOperator.fild\_dword);

m\_floatPushMap.put(Sort.Unsigned\_Long\_Int, ObjectOperator.fild\_dword);

m\_floatPushMap.put(Sort.Float, ObjectOperator.fld\_dword);

m\_floatPushMap.put(Sort.Double, ObjectOperator.fld\_qword);

m\_floatPushMap.put(Sort.Long\_Double, ObjectOperator.fld\_qword);

m\_floatTopMap.put(Sort.Signed\_Int, ObjectOperator.fist\_word);

m\_floatTopMap.put(Sort.Unsigned\_Int, ObjectOperator.fist\_word);

m\_floatTopMap.put(Sort.Pointer, ObjectOperator.fist\_word);

m\_floatTopMap.put(Sort.Signed\_Long\_Int, ObjectOperator.fist\_dword);

m\_floatTopMap.put(Sort.Unsigned\_Long\_Int, ObjectOperator.fist\_dword);

m\_floatTopMap.put(Sort.Float, ObjectOperator.fst\_dword);

m\_floatTopMap.put(Sort.Double, ObjectOperator.fst\_qword);

m\_floatTopMap.put(Sort.Long\_Double, ObjectOperator.fst\_qword);

m\_floatPopMap.put(Sort.Signed\_Int, ObjectOperator.fistp\_word);

m\_floatPopMap.put(Sort.Unsigned\_Int, ObjectOperator.fistp\_word);

m\_floatPopMap.put(Sort.Pointer, ObjectOperator.fistp\_word);

m\_floatPopMap.put(Sort.Signed\_Long\_Int, ObjectOperator.fistp\_dword);

m\_floatPopMap.put(Sort.Unsigned\_Long\_Int, ObjectOperator.fistp\_dword);

m\_floatPopMap.put(Sort.Float, ObjectOperator.fstp\_dword);

m\_floatPopMap.put(Sort.Double, ObjectOperator.fstp\_qword);

m\_floatPopMap.put(Sort.Long\_Double, ObjectOperator.fstp\_qword);

}

private void loadRegister(Register register, Symbol symbol,

Register addressRegister, int addressOffset) {

if (symbol.hasValue()) {

int value = ((BigInteger) symbol.getSignedValue()).intValue();

addObjectCode(ObjectOperator.mov, register, value, null);

}

else if (addressRegister != null) {

if (symbol.getType().isFunctionOrArray()) {

addObjectCode(ObjectOperator.mov, register, addressRegister, null);

if (addressOffset != 0) {

addObjectCode(ObjectOperator.add, register, addressOffset, null);

}

}

else {

addObjectCode(ObjectOperator.mov, register,

addressRegister, addressOffset);

}

}

else if (symbol.isAutoOrRegister()) {

Register stackRegister = getStackRegister(symbol);

if (symbol.getType().isFunctionOrArray()) {

addObjectCode(ObjectOperator.mov, register, stackRegister, null);

addOffset(register, symbol.getOffset());

}

else {

addObjectCode(ObjectOperator.mov, register,

stackRegister, symbol.getOffset());

}

}

else {

if (symbol.getType().isFunctionOrArray()) {

Pair<String,Integer> pair =

new Pair<>(symbol.getUniqueName(), symbol.getOffset());

addObjectCode(ObjectOperator.mov, register, pair, null); // mov ax, f

}

else {

Pair<String,Integer> pair =

new Pair<>(symbol.getUniqueName(), symbol.getOffset());

addObjectCode(ObjectOperator.mov, register, null, pair); // mov ax, [g]

if (!symbol.getName().contains(".")) {

addOffset(register, symbol.getOffset());

}

}

}

}

private void addOffset(Register register, int offset) {

if (offset > 0) {

addObjectCode(ObjectOperator.add, register, offset, null);

}

}

private Register getStackRegister(Symbol symbol) {

if (!symbol.isParameter() && Main.FuncSymbol.getType().isEllipse()) {

return ObjectCode.EllipseRegister;

}

else {

return ObjectCode.FrameRegister;

}

}

private void loadAddressRegister(Register register, Symbol symbol,

Register baseRegister) {

if (symbol.isAutoOrRegister()) {

if (baseRegister == null) {

baseRegister = getStackRegister(symbol);

}

addObjectCode(ObjectOperator.mov, register, baseRegister, null);

addOffset(register, symbol.getOffset());

}

else {

Pair<String,Integer> pair =

new Pair<>(symbol.getUniqueName(), symbol.getOffset());

addObjectCode(ObjectOperator.mov, register, pair, null);

addOffset(register, symbol.getOffset());

}

}

private void saveRegister(Register register, Symbol symbol,

Register addressRegister, int addressOffset) {

if (addressRegister != null) {

addObjectCode(ObjectOperator.mov, addressRegister,

addressOffset, register);

}

else if (symbol.isAutoOrRegister()) {

Register stackRegister = getStackRegister(symbol);

addObjectCode(ObjectOperator.mov, stackRegister,

symbol.getOffset(), register);

}

else {

Pair<String,Integer> pair =

new Pair<>(symbol.getUniqueName(), symbol.getOffset());

addObjectCode(ObjectOperator.mov, null, pair, register);

}

}

private int generateUnaryExpression(ObjectOperator operator, Register register,

Symbol symbol, Register addressRegister, int addressOffset) {

if (register != null) {

return addCountObjectCode(operator, register, null, null);

}

else {

if (operator != ObjectOperator.call) {

operator=ObjectCode.operatorSize(operator, symbol.getType().getSize());

}

if (addressRegister != null) {

return addCountObjectCode(operator,addressRegister,addressOffset,null);

}

else if (symbol.isAutoOrRegister()) {

Register stackRegister = getStackRegister(symbol);

return addCountObjectCode(operator, stackRegister,

symbol.getOffset(), null);

}

else {

Pair<String,Integer> pair =

new Pair<>(symbol.getUniqueName(), symbol.getOffset());

return addCountObjectCode(operator, null, pair, null);

}

}

}

private void generateBinaryExpression(ObjectOperator operator, Register leftRegister,

Symbol leftSymbol, Register leftAddressRegister,

int leftAddressOffset, Register rightRegister,

Symbol rightSymbol, Register rightAddressRegister, int rightAddressOffset) {

if ((leftRegister != null) && (rightRegister != null)) {

addObjectCode(operator, leftRegister, rightRegister, null);

}

else if ((leftRegister != null) && rightSymbol.hasValue()) {

Object rightValue = rightSymbol.getSignedValue().intValue();

addObjectCode(operator, leftRegister, rightValue, null);

}

else if (leftSymbol.hasValue() && (rightRegister != null)) {

Object leftValue = leftSymbol.getSignedValue().intValue();

addObjectCode(operator, leftValue, rightRegister, null);

}

else if (leftRegister != null) {

generateBinaryMemory(operator, leftRegister, null, 0, null, null,

rightAddressRegister, rightAddressOffset, rightSymbol);

}

else if (rightRegister != null) {

generateBinaryMemory(operator, null, leftAddressRegister,

leftAddressOffset, leftSymbol, rightRegister, null, 0, null);

}

else if (leftSymbol.hasValue()) {

Object leftValue = leftSymbol.getSignedValue().intValue();

generateBinaryMemory(operator, leftValue, null, 0, null, null,

rightAddressRegister, rightAddressOffset, rightSymbol);

}

else if (rightSymbol.hasValue()) {

Object rightValue = rightSymbol.getSignedValue().intValue();

generateBinaryMemory(operator, null, leftAddressRegister,

leftAddressOffset, leftSymbol, rightValue, null, 0, null);

}

else {

Assert.error("generateBinaryExpression)");

}

}

private void generateBinaryMemory(ObjectOperator operator, Object leftRegisterValue, Register leftAddressRegister, int leftAddressOffset, Symbol leftSymbol,

Object rightRegisterValue, Register rightAddressRegister, int rightAddressOffset, Symbol rightSymbol) {

if (leftRegisterValue != null) {

if (leftRegisterValue instanceof Integer) {

operator =

ObjectCode.operatorSize(operator, leftSymbol.getType().getSize());

}

if (rightAddressRegister != null) {

addObjectCode(operator, leftRegisterValue,

rightAddressRegister, rightAddressOffset);

}

else if (rightSymbol.isAutoOrRegister()) {

Register stackRegister = getStackRegister(rightSymbol);

if (rightSymbol.getType().isFunctionOrArray()) {

addObjectCode(operator, leftRegisterValue, stackRegister, null);

addObjectCode(operator, leftRegisterValue,

rightSymbol.getOffset(), null);

}

else {

addObjectCode(operator, leftRegisterValue,

stackRegister, rightSymbol.getOffset());

}

}

else {

Pair<String,Integer> rightPair =

new Pair<>(rightSymbol.getUniqueName(), rightSymbol.getOffset());

if (rightSymbol.getType().isFunctionOrArray()) {

addObjectCode(operator, leftRegisterValue, rightPair, null);

}

else {

addObjectCode(operator, leftRegisterValue, null, rightPair);

}

}

}

else if (rightRegisterValue != null) {

if (rightRegisterValue instanceof Integer) {

operator =

ObjectCode.operatorSize(operator, leftSymbol.getType().getSize());

}

if (leftAddressRegister != null) {

addObjectCode(operator, leftAddressRegister,

leftAddressOffset, rightRegisterValue);

}

else if (leftSymbol.isAutoOrRegister()) {

Register stackRegister = getStackRegister(leftSymbol);

addObjectCode(operator, stackRegister,

leftSymbol.getOffset(), rightRegisterValue);

}

else {

Pair<String,Integer> leftPair =

new Pair<>(leftSymbol.getUniqueName(), leftSymbol.getOffset());

addObjectCode(operator, null, leftPair, rightRegisterValue);

}

}

else {

Assert.error("generateBinaryMemory)");

}

}

private void generateIntegral(MiddleCode middleCode, int index) {

MiddleOperator middleOp = middleCode.getOperator();

Object operand1 = middleCode.getOperand(1),

operand2 = middleCode.getOperand(2),

operand3 = middleCode.getOperand(3);

Symbol resultSymbol = (operand1 instanceof Symbol) ? ((Symbol) operand1) : null,

leftSymbol = (operand2 instanceof Symbol) ? ((Symbol) operand2) : null,

rightSymbol = (operand3 instanceof Symbol) ? ((Symbol) operand3) : null;

Pair<Symbol,Register> axPair = middleCode.axPair();

if (axPair != null) {

Symbol saveSymbol = axPair.getFirst();

Register saveRegister = axPair.getSecond();

saveRegister(saveRegister, saveSymbol, null, 0);

}

Register leftRegister = middleCode.getRegister(Track.Left),

leftAddressRegister = middleCode.getRegister(Track.LeftAddress),

rightRegister = middleCode.getRegister(Track.Right),

rightAddressRegister = middleCode.getRegister(Track.RightAddress);

int leftAddressOffset = middleCode.getAddressOffset(Track.LeftAddress),

rightAddressOffset = middleCode.getAddressOffset(Track.RightAddress);

if (middleCode.toString().contains("Parameter offset 15 to $temp1675")) {

int i = 1;

}

if ((leftRegister != null) && middleCode.isLoad(Track.Left)) {

loadRegister(leftRegister, leftSymbol,

leftAddressRegister, leftAddressOffset);

}

if (!middleCode.isMemory() && (rightRegister != null) &&

(rightSymbol != null) && middleCode.isLoad(Track.Right)) {

loadRegister(rightRegister, rightSymbol,

rightAddressRegister, rightAddressOffset);

}

if (!middleCode.isMemory() && (leftAddressRegister != null) &&

middleCode.isLoad(Track.LeftAddress)) { // && (middleOp != MiddleOperator.Parameter)) {

loadRegister(leftAddressRegister, leftSymbol.getAddressSymbol(), null,0);

}

if ((rightRegister != null) && middleCode.isLoad(Track.RightAddress)) {

loadRegister(rightAddressRegister,rightSymbol.getAddressSymbol(),null,0);

}

switch (middleOp) { // mov [bp + 10], 123

case Parameter: { // mov [bp + 10], ax

Symbol paramSymbol = (Symbol) operand2;

Register stackRegister = getStackRegister(paramSymbol);

int recordSize = Main.CurrentTable.getFunctionSize();

int paramOffset = recordSize + ((int) operand1);

Register baseRegister = Main.FuncSymbol.getType().isEllipse() ?

ObjectCode.EllipseRegister :

ObjectCode.FrameRegister;

if (paramSymbol.hasValue()) {

if (paramSymbol.getValue() instanceof String) {

GenerateInitializerStatic.generateByteTextList(Type.StringType, paramSymbol.getValue(),

paramSymbol.getByteList(), paramSymbol.getAccessMap(), paramSymbol.getTextList());

Pair<String,Integer> pair = new Pair<>(paramSymbol.getUniqueName(), paramSymbol.getOffset());

addObjectCode(ObjectOperator.mov\_word, baseRegister, paramOffset, pair);

}

else {

int value = ((BigInteger) paramSymbol.getSignedValue()).intValue();

ObjectOperator objectOp =

ObjectCode.operatorSize(ObjectOperator.mov, paramSymbol.getType().getSize());

addObjectCode(objectOp, baseRegister, paramOffset, value);

}

}

else { // mov [bp + 2], bp + 10

if (leftRegister != null) {

addObjectCode(ObjectOperator.mov, baseRegister, paramOffset, leftRegister);

}

else {

Assert.error(paramSymbol.getType().isFunctionOrArray());

if (paramSymbol.isAutoOrRegister()) {

addObjectCode(ObjectOperator.mov, baseRegister, paramOffset, stackRegister);

addObjectCode(ObjectOperator.add\_word, baseRegister, paramOffset, paramSymbol.getOffset());

}

else {

Pair<String,Integer> paramPair = new Pair<>(paramSymbol.getUniqueName(), paramSymbol.getOffset());

addObjectCode(ObjectOperator.mov\_word, baseRegister, paramOffset, paramPair);

}

}

}

}

break;

case Exit:

if (leftSymbol == null) {

addObjectCode(ObjectOperator.mov, Register.al, 0x00, null);

}

else if (leftRegister != Register.al) {

addObjectCode(ObjectOperator.mov, Register.al, leftRegister, null);

}

addObjectCode(ObjectOperator.mov, Register.ah, 0x4C, null);

addObjectCode(ObjectOperator.interrupt, 33, null, null);

break;

case LoadToRegister: {

Register register = (Register) operand1;

if (leftSymbol.hasValue()) {

loadRegister(register, leftSymbol,

rightAddressRegister, rightAddressOffset);

}

else if (register != leftRegister) {

addObjectCode(ObjectOperator.mov, register, leftRegister, null);

}

}

break;

case SaveFromRegister:

saveRegister(leftRegister, resultSymbol,

leftAddressRegister, leftAddressOffset);

break;

case ClearRegisters:

break;

case SaveFromFlagByte:

addObjectCode(ObjectOperator.sahf, null, null, null);

addObjectCode(ObjectOperator.mov, Register.al, Register.ah, null);

break;

case LoadStatusByte:

addObjectCode(ObjectOperator.lahf, null, null, null);

break;

// x = x + 1

// x = 1 + x

// x += 1

// x++

// $1 = y + z

// x = $1

// x = y + z

// $1 = -y

// x = $1

// x = -y

// x = -x -> neg [bp + 10]

case BitwiseNot:

case UnarySubtract: {

ObjectOperator objectOp = m\_middleToIntegralTargetMap.get(middleOp);

generateUnaryExpression(objectOp, leftRegister, leftSymbol,

leftAddressRegister, leftAddressOffset);

}

break;

case PrefixIncrement:

case PrefixDecrement: {

ObjectOperator objectOp = m\_middleToIntegralTargetMap.get(middleOp);

generateUnaryExpression(objectOp, rightRegister, rightSymbol,

rightAddressRegister, rightAddressOffset);

if (rightRegister != null) {

saveRegister(rightRegister, rightSymbol, null, 0);

}

}

break;

case PostfixIncrement:

case PostfixDecrement: {

if (rightRegister != null) {

addObjectCode(ObjectOperator.mov, leftRegister,rightRegister,null);

}

else {

loadRegister(leftRegister, rightSymbol, null, 0);

}

ObjectOperator objectOp = m\_middleToIntegralTargetMap.get(middleOp);

generateUnaryExpression(objectOp, rightRegister, rightSymbol,

rightAddressRegister, rightAddressOffset);

if (rightRegister != null) {

saveRegister(rightRegister, rightSymbol, null, 0);

}

}

break;

case Interrupt: {

int intValue = ((BigInteger) operand1).intValue();

addObjectCode(ObjectOperator.interrupt, intValue, null, null);

}

break;

case JumpRegister: {

Register jumpRegister = (Register) operand1;

addObjectCode(ObjectOperator.jmp, jumpRegister, null, null);

}

break;

case Multiply:

case Divide:

case Modulo: {

if (middleCode.toString().contains("Modulo $temp1170 num denum")) {

int i = 1;

}

int typeSize = leftSymbol.getType().getSize();

Register axRegister = RegisterPreparator.LeftMultiplyMap.get(typeSize),

dxRegister = RegisterPreparator.ClearMultiplyMap.get(typeSize);

if (leftRegister != axRegister) {

Assert.error(ObjectCode.registerSize(leftRegister) ==

ObjectCode.registerSize(dxRegister));

addObjectCode(ObjectOperator.mov, axRegister, leftRegister, null);

}

Assert.error(dxRegister != null);

addObjectCode(ObjectOperator.mov, dxRegister, 0, null);

boolean signed = leftSymbol.getType().isSigned();

ObjectOperator objectOp =

m\_middleToSignedIntegralTargetMap.get(new Pair<>(middleOp, signed));

generateUnaryExpression(objectOp, rightRegister, rightSymbol,

rightAddressRegister, rightAddressOffset);

}

break;

case Deref:

case Arrow:

case Index: {

if (middleCode.isLoad(Track.Right)) {

loadRegister(rightRegister, rightSymbol, null, 0);

}

if (leftRegister != null) {

int addressOffset = (int) middleCode.getOperand(1);

addObjectCode(ObjectOperator.mov, leftRegister,

rightRegister, addressOffset);

}

}

break;

case Assign:

if (rightSymbol.hasValue()) {

Object value = rightSymbol.getValue();

if (value instanceof String) {

String stringName = GenerateInitializerStatic.generateValueName(Type.StringType, value);

value = new Pair<>(stringName, 0);

}

else {

value = ((BigInteger) rightSymbol.getValue()).intValue();

}

if (leftRegister != null) {

addObjectCode(ObjectOperator.mov, leftRegister, value, null);

}

else if (leftAddressRegister != null) {

ObjectOperator operator = ObjectCode.operatorSize(ObjectOperator.mov, leftSymbol.getType().getSize());

addObjectCode(operator, leftAddressRegister, leftAddressOffset, value);

}

else {

ObjectOperator operator = ObjectCode.operatorSize(ObjectOperator.mov, leftSymbol.getType().getSize());

if (leftSymbol.isAutoOrRegister()) {

Register stackRegister = getStackRegister(leftSymbol);

addObjectCode(operator, stackRegister, leftSymbol.getOffset(), value);

}

else {

Pair<String,Integer> staticPair = new Pair<>(leftSymbol.getUniqueName(), leftSymbol.getOffset());

addObjectCode(operator, null, staticPair, value);

}

}

}

//else if (leftRegister == null) {

else if ((!leftSymbol.isTemporary() || (leftRegister == null)) &&

(leftRegister != rightRegister)) {

generateBinaryExpression(m\_middleToIntegralTargetMap.get(middleOp),

leftRegister, leftSymbol, leftAddressRegister, leftAddressOffset,

rightRegister, rightSymbol, rightAddressRegister, rightAddressOffset);

}

if (middleCode.isSave() && (leftRegister != null) &&

!(leftSymbol.isTemporary() && (leftAddressRegister == null) && (leftSymbol.getOffset() == 0))) {

saveRegister(leftRegister, leftSymbol, leftAddressRegister, leftAddressOffset);

}

//}

break;

case BinaryAdd:

case BitwiseIOr:

case BitwiseXOr:

case BitwiseAnd:

case BinarySubtract:

generateBinaryExpression(m\_middleToIntegralTargetMap.get(middleOp),

leftRegister, leftSymbol, leftAddressRegister,

leftAddressOffset, rightRegister, rightSymbol,

rightAddressRegister, rightAddressOffset);

break;

case LeftShift:

case RightShift: {

if (rightRegister != null) {

if (middleCode.isLoad(Track.Right)) {

loadRegister(rightRegister, rightSymbol,

rightAddressRegister, rightAddressOffset);

}

else if (rightRegister != Register.cl) {

Assert.error(ObjectCode.registerSize(rightRegister) ==

ObjectCode.registerSize(Register.cl));

addObjectCode(ObjectOperator.mov, Register.cl,

rightRegister, null);

rightRegister = Register.cl;

}

}

generateBinaryExpression(m\_middleToIntegralTargetMap.get(middleOp),

leftRegister, leftSymbol, leftAddressRegister,

leftAddressOffset, rightRegister, rightSymbol,

rightAddressRegister, rightAddressOffset);

}

break;

case Equal:

case NotEqual:

case LessThan:

case LessThanEqual:

case GreaterThan:

case GreaterThanEqual: {

generateBinaryExpression(ObjectOperator.cmp, leftRegister, leftSymbol,

leftAddressRegister, leftAddressOffset,

rightRegister, rightSymbol,

rightAddressRegister, rightAddressOffset);

int target = (Integer) operand1;

boolean signed = leftSymbol.getType().isSigned();

ObjectOperator objectOp =

m\_middleToSignedIntegralTargetMap.get(new Pair<>(middleOp,signed));

ObjectOperator inverseOp = m\_inverseMap.get(objectOp);

addObjectCode(inverseOp, null, null, index + 1);

addObjectCode(ObjectOperator.long\_jmp, null, null, target);

}

break;

case SetReturnValue: {

Register returnRegister = ObjectCode.registerToSize(ObjectCode.ReturnValueRegister, ObjectCode.registerSize(leftRegister));

if (leftRegister != returnRegister) {

addObjectCode(ObjectOperator.mov, returnRegister, leftRegister, null);

}

}

break;

case GetReturnValue:

break;

default:

//Assert.error("default integral");

break;

}

if (middleCode.isSave() && (resultSymbol != null)) {

saveRegister(leftRegister, resultSymbol, leftAddressRegister, leftAddressOffset);

}

}

// ------------------------------------------------------------------------

private void pushFloating(Symbol symbol, Register addressRegister, int addressOffset) {

ObjectOperator objectOp = m\_floatPushMap.get(symbol.getType().getSort());

if (symbol.hasValue()) {

Object value = symbol.getValue();

if (value instanceof BigInteger) {

value = new BigDecimal(((BigInteger) value).intValue());

}

if (((Number) value).doubleValue() == 0.0) {

addObjectCode(ObjectOperator.fldz, null, null, null);

}

else if (((Number) value).doubleValue() == 1.0) {

addObjectCode(ObjectOperator.fld1, null, null, null);

}

else {

Pair<String,Integer> pair = new Pair<>(symbol.getUniqueName(), 0);

addObjectCode(objectOp, null, pair, null);

}

}

else if (addressRegister != null) {

addObjectCode(objectOp, addressRegister, addressOffset, null);

}

else if (symbol.isAutoOrRegister()) {

addObjectCode(objectOp, getStackRegister(symbol), symbol.getOffset(), null);

}

else {

Pair<String,Integer> pair = new Pair<>(symbol.getUniqueName(), symbol.getOffset());

addObjectCode(objectOp, null, pair, null);

}

++m\_stackSize;

Main.DebugStream.println("Push " + symbol.getName() + " " + m\_stackSize);

}

private void topPopFloating(Symbol symbol, boolean pop, Register addressRegister,

int addressOffset, Register baseRegister) {

if (symbol == null) {

Assert.error(pop, "pop");

if (pop) {

--m\_stackSize;

addObjectCode(ObjectOperator.fstsw, null, null, null); // XXX

Main.DebugStream.println("Pop " + m\_stackSize);

}

else {

Main.DebugStream.println("Top " + m\_stackSize);

}

}

else { // if (!symbol.isTemporary()) {

ObjectOperator objectOp = pop ? m\_floatPopMap.get(symbol.getType().getSort())

: m\_floatTopMap.get(symbol.getType().getSort());

if (addressRegister != null) {

addObjectCode(objectOp, addressRegister, 0, null);

}

else if (symbol.isAutoOrRegister()) {

if (baseRegister == null) {

baseRegister = getStackRegister(symbol);

}

addObjectCode(objectOp, baseRegister, symbol.getOffset(), null);

}

else {

Pair<String,Integer> pair = new Pair<>(symbol.getUniqueName(), symbol.getOffset());

addObjectCode(objectOp, null, pair, null);

}

if (pop) {

--m\_stackSize;

Main.DebugStream.println("Pop " + symbol.getName() + " " + m\_stackSize);

}

else {

Main.DebugStream.println("Top " + m\_stackSize);

}

}

}

private void generateFloating(MiddleCode middleCode, int index) {

MiddleOperator middleOp = middleCode.getOperator();

Object operand1 = middleCode.getOperand(1),

operand2 = middleCode.getOperand(2),

operand3 = middleCode.getOperand(3);

Register leftAddressRegister = middleCode.getRegister(Track.LeftAddress);

int leftAddressOffset = middleCode.getAddressOffset(Track.LeftAddress);

switch (middleOp) {

case PushFloat: {

Symbol pushSymbol = (Symbol) operand1;

Register leftRegister = middleCode.getRegister(Track.Left);

if ((leftRegister != null) && pushSymbol.isTemporary()) {

Assert.error(pushSymbol.getOffset() != 0);

leftRegister =

ObjectCode.registerToSize(leftRegister, pushSymbol.getType().getSize());

saveRegister(leftRegister, pushSymbol, null, 0);

}

pushFloating(pushSymbol, leftAddressRegister, leftAddressOffset);

}

break;

case TopFloat: {

Symbol symbol = (Symbol) operand1;

topPopFloating(symbol, false, leftAddressRegister,

leftAddressOffset, null);

}

break;

case PopFloat: {

Symbol symbol = (Symbol) operand1;

Assert.error(symbol != null);

topPopFloating(symbol, true, leftAddressRegister,

leftAddressOffset, null);

}

break;

case UnaryAdd:

break;

case UnarySubtract:

addObjectCode(ObjectOperator.fchs, null, null, null);

break;

case Assign: {

boolean pop = (Boolean) operand1;

Symbol resultSymbol = (Symbol) operand2;

if (middleCode.isLoad(Track.LeftAddress)) {

loadRegister(leftAddressRegister,

resultSymbol.getAddressSymbol(), null, 0);

}

topPopFloating(resultSymbol, pop, leftAddressRegister,

leftAddressOffset, null);

}

break;

case PrefixIncrement:

case PrefixDecrement: {

Symbol incSymbol = (Symbol) operand3,

oneSymbol =

Symbol.createValueSymbol(incSymbol.getType(), BigDecimal.ONE);

pushFloating(oneSymbol, null, 0);

addObjectCode(m\_middleToFloatingTargetMap.get(middleOp),

null, null, null);

boolean pop = (boolean) operand1;

topPopFloating(incSymbol, pop, leftAddressRegister,

leftAddressOffset, null);

}

break;

case PostfixIncrement:

case PostfixDecrement: {

Symbol resultSymbol = (Symbol) operand2,

incSymbol = (Symbol) operand3,

oneSymbol =

Symbol.createValueSymbol(incSymbol.getType(),BigDecimal.ONE);

topPopFloating(resultSymbol, false, leftAddressRegister,

leftAddressOffset, null);

pushFloating(oneSymbol, null, 0);

addObjectCode(m\_middleToFloatingTargetMap.get(middleOp),

null, null, null);

topPopFloating(incSymbol, true, leftAddressRegister,

leftAddressOffset, null);

pushFloating(resultSymbol, null, 0);

}

break;

case BinaryAdd:

case BinarySubtract:

case Multiply:

case Divide: {

addObjectCode(m\_middleToFloatingTargetMap.get(middleOp),

null, null, null);

--m\_stackSize;

Main.DebugStream.println("Binary " + m\_stackSize);

}

break;

case Equal:

case NotEqual:

case LessThan:

case LessThanEqual:

case GreaterThan:

case GreaterThanEqual: {

int target = (Integer) operand1;

addObjectCode(ObjectOperator.fcompp, null, null, null);

addObjectCode(ObjectOperator.fstsw, Register.ax, null, null);

addObjectCode(ObjectOperator.sahf, null, null, null);

ObjectOperator objectOp = m\_middleToFloatingTargetMap.get(middleOp);

ObjectOperator inverseOp = m\_inverseMap.get(objectOp);

addObjectCode(inverseOp, null, null, index + 1);

addObjectCode(ObjectOperator.long\_jmp, null, null, target);

m\_stackSize -= 2;

Main.DebugStream.println("Relation " + m\_stackSize);

}

break;

case Parameter: {

Symbol paramSymbol = (Symbol) operand2;

Symbol saveSymbol =

Symbol.createTemporarySymbol(paramSymbol.getType());

int recordSize = Main.CurrentTable.getFunctionSize();

int paramOffset = recordSize + ((int) operand1);

saveSymbol.setOffset(paramOffset);

Register baseRegister = Main.FuncSymbol.getType().isEllipse() ?

ObjectCode.EllipseRegister :

ObjectCode.FrameRegister;

topPopFloating(saveSymbol, true, null, 0, baseRegister);

}

break;

case GetReturnValue:

break;

case CheckReturnValue1:

addObjectCode(ObjectOperator.cmp\_byte, ObjectCode.FrameRegister,

SymbolTable.UseResultOffset, 0);

addObjectCode(ObjectOperator.je, null, null, index + 1);

addObjectCode(ObjectOperator.long\_jmp, null, null, operand1);

break;

case CheckReturnValue2:

addObjectCode(ObjectOperator.long\_jmp, null, null, operand1);

break;

case CheckReturnValue3:

break;

case SetReturnValue:

break;

case Deref:

break;

case Index:

break;

default:

Assert.error("default floating", middleOp.toString());

break;

}

}

// ------------------------------------------------------------------------

private void generateStructure(MiddleCode middleCode, int index) {

MiddleOperator middleOp = middleCode.getOperator();

Object operand1 = middleCode.getOperand(1),

operand2 = middleCode.getOperand(2),

operand3 = middleCode.getOperand(3);

switch (middleOp) {

case Assign: {

Symbol targetSymbol = (Symbol) operand2,

sourceSymbol = (Symbol) operand3;

int copySize = targetSymbol.getType().getSize();

Register targetAddressRegister =

middleCode.getRegister(Track.LeftAddress),

sourceAddressRegister =

middleCode.getRegister(Track.RightAddress),

countRegister = middleCode.getRegister(Track.Left),

valueRegister = middleCode.getRegister(Track.Right);

if (middleCode.isLoad(Track.Left)) {

loadAddressRegister(targetAddressRegister, targetSymbol, null);

}

if (middleCode.isLoad(Track.Right)) {

loadAddressRegister(sourceAddressRegister, sourceSymbol, null);

}

generateMemoryCopy(targetAddressRegister, sourceAddressRegister,

countRegister, valueRegister, copySize, index);

}

break;

case Parameter: {

int recordSize = Main.CurrentTable.getFunctionSize();

int paramOffset = recordSize + ((int) operand1);

Symbol paramSymbol = (Symbol) operand2;

Register targetAddressRegister =

middleCode.getRegister(Track.LeftAddress),

sourceAddressRegister =

middleCode.getRegister(Track.RightAddress),

countRegister = middleCode.getRegister(Track.Left),

valueRegister = middleCode.getRegister(Track.Right);

if (middleCode.isLoad(Track.Right)) {

loadAddressRegister(sourceAddressRegister, paramSymbol, null);

}

Symbol targetSymbol = Symbol.createVariableSymbol("parameter", false,

Storage.Auto, paramSymbol.getType(), null);

targetSymbol.setOffset(paramOffset);

Register baseRegister = Main.FuncSymbol.getType().isEllipse() ?

ObjectCode.EllipseRegister :

ObjectCode.FrameRegister;

loadAddressRegister(targetAddressRegister,

targetSymbol, baseRegister);

generateMemoryCopy(targetAddressRegister, sourceAddressRegister,

countRegister, valueRegister,

paramSymbol.getType().getSize(), index);

}

break;

case GetReturnValue: {

Symbol returnSymbol = (Symbol) operand1;

Register returnAddressRegister =

middleCode.getRegister(Track.LeftAddress),

countRegister = middleCode.getRegister(Track.Left),

valueRegister = middleCode.getRegister(Track.Right);

if (middleCode.isLoad(Track.Left)) {

loadAddressRegister(returnAddressRegister, returnSymbol, null);

}

generateMemoryCopy(returnAddressRegister, ObjectCode.ReturnPointerRegister,

countRegister, valueRegister,

returnSymbol.getType().getSize(), index);

}

break;

case SetReturnValue: {

Symbol returnSymbol = (Symbol) operand2;

loadAddressRegister(ObjectCode.ReturnPointerRegister,

returnSymbol, null);

}

break;

case Deref:

break;

case Index:

break;

default:

Assert.error("default struct", middleOp.toString());

break;

}

}

private void generateMemoryCopy(Register targetAddressRegister,

Register sourceAddressRegister,

Register countRegister, Register valueRegister,

int size, int index) {

addObjectCode(ObjectOperator.mov, countRegister, size, null);

String label = Main.FuncSymbol.getUniqueName() + Main.SeparatorId +

index + Main.SeparatorId + "Loop";

addObjectCode(ObjectOperator.label, label, null, null);

addObjectCode(ObjectOperator.mov, valueRegister, sourceAddressRegister, 0);

addObjectCode(ObjectOperator.mov, targetAddressRegister, 0, valueRegister);

addObjectCode(ObjectOperator.inc, sourceAddressRegister, null, null);

addObjectCode(ObjectOperator.inc, targetAddressRegister, null, null);

addObjectCode(ObjectOperator.dec, countRegister, null, null);

addObjectCode(ObjectOperator.cmp, countRegister, 0, null);

addObjectCode(ObjectOperator.jne, null, m\_objectCodeList.size() - 7, label, null);

}

// ------------------------------------------------------------------------

private void generateConvert(MiddleCode middleCode, int index) {

MiddleOperator middleOp = middleCode.getOperator();

Register leftRegister = middleCode.getRegister(Track.Left),

rightRegister = middleCode.getRegister(Track.Right),

leftAddressRegister = middleCode.getRegister(Track.LeftAddress),

rightAddressRegister = middleCode.getRegister(Track.RightAddress);

int leftAddressOffset = middleCode.getAddressOffset(Track.LeftAddress),

rightAddressOffset = middleCode.getAddressOffset(Track.RightAddress);

Symbol leftSymbol = (Symbol) middleCode.getOperand(2),

rightSymbol = (Symbol) middleCode.getOperand(3);

if (middleCode.isLoad(Track.Left) && (leftRegister != null)) {

loadRegister(leftRegister, leftSymbol, leftAddressRegister, leftAddressOffset);

}

if (middleCode.isLoad(Track.Right) && (rightRegister != null)) {

loadRegister(rightRegister, rightSymbol, rightAddressRegister, rightAddressOffset);

}

switch (middleOp) {

case IntegralToIntegral: {

if (leftSymbol.getType().getSize() != rightSymbol.getType().getSize()) {

if (leftSymbol.getType().getSize() != rightSymbol.getType().getSize()) {

int leftSize = leftSymbol.getType().getSize(),

rightSize = rightSymbol.getType().getSize();

Assert.error(!leftRegister.name().contains("h"), "left register");

//Assert.error(!getRightRegister.getName().contains("h"), "right getRegister");

if (rightRegister.name().contains("h")) {

Register highRegister = rightRegister;

rightRegister = Register.valueOf(rightRegister.name().replace("h", "l"));

addObjectCode(ObjectOperator.mov, rightRegister, highRegister, null);

}

if (rightSize < leftSize) {

int mask = m\_maskMap.get(new Pair<>(rightSize, leftSize));

addObjectCode(ObjectOperator.and, leftRegister, mask, null);

}

if (rightSymbol.getType().isSigned()) {

ObjectCode negSource = new ObjectCode(ObjectOperator.neg, rightRegister, null, null);

ObjectCode negTarget = new ObjectCode(ObjectOperator.neg, leftRegister, null, null);

addObjectCode(ObjectOperator.cmp, rightRegister, 0, null);

addObjectCode(ObjectOperator.jge, null, null, index + 1);

m\_objectCodeList.add(negSource);

m\_objectCodeList.add(negTarget);

}

}

}

}

break;

case IntegralToFloating:

if (rightSymbol.isTemporary() && (rightAddressRegister == null)) {

saveRegister(rightRegister, rightSymbol,

rightAddressRegister, rightAddressOffset);

}

pushFloating(rightSymbol, rightAddressRegister, rightAddressOffset);

break;

case FloatingToIntegral:

topPopFloating(leftSymbol, true, leftAddressRegister,

leftAddressOffset, null);

break;

}

}

// ------------------------------------------------------------------------

public void addInitCode() {

List<Byte> byteList = new LinkedList<>();

Map<Integer,String> accessMap = new ListMap<>();

List<String> textList = new LinkedList<>();

//textList.add("\n" + Main.StackTopName + ":");

Symbol symbol = Symbol.createStaticSymbol(Main.StackTopName, byteList,

accessMap, textList);

Main.GlobalSet.add(symbol);

addObjectCode(ObjectOperator.comment, "Initialize Stack Pointer", null, null);

Pair<String,Integer> pair = new Pair<>(Main.StackTopName, 0);

addObjectCode(ObjectOperator.mov, ObjectCode.FrameRegister, pair, null);

addObjectCode(ObjectOperator.comment, "Initialize Heap Pointer", null, null);

addObjectCode(ObjectOperator.mov\_word, null, 65534, -2); // 65534);

addObjectCode(ObjectOperator.comment, "Initialize FPU Control Word, truncate mode => set bit 10 and 11.", null, null);

addObjectCode(ObjectOperator.fstcw, ObjectCode.FrameRegister, 0, null);

addObjectCode(ObjectOperator.or\_word, ObjectCode.FrameRegister, 0, 0x0C00);

addObjectCode(ObjectOperator.fldcw, ObjectCode.FrameRegister, 0, null);

}

public void addArgsCode() {

{ Symbol symbol = Symbol.createStaticSymbol(Main.PathName, null, null,

new LinkedList<String>());

Main.GlobalSet.add(symbol);

}

String label1 = Main.SeparatorId + "Initialize" + Main.SeparatorId + "1",

label2 = Main.SeparatorId + "Initialize" + Main.SeparatorId + "2",

label3 = Main.SeparatorId + "Initialize" + Main.SeparatorId + "3",

label4 = Main.SeparatorId + "Initialize" + Main.SeparatorId + "4",

label5 = Main.SeparatorId + "Initialize" + Main.SeparatorId + "5";

addObjectCode(ObjectOperator.comment,"Initialize Argument List",null,null);

addObjectCode(ObjectOperator.mov, Register.si, Register.bp, null);

Pair<String,Integer> pair = new Pair<>(Main.PathName, 0);

addObjectCode(ObjectOperator.mov\_word, Register.bp, 0, pair);

addObjectCode(ObjectOperator.add, Register.bp, 2, null);

addObjectCode(ObjectOperator.mov, Register.ax, 1, null);

addObjectCode(ObjectOperator.mov, Register.bx, 129, null);

addObjectCode(ObjectOperator.label, label1, null, null);

addObjectCode(ObjectOperator.cmp\_byte, Register.bx, 0, 32);

addObjectCode(ObjectOperator.je, null, m\_objectCodeList.size()+5, label2);

addObjectCode(ObjectOperator.cmp\_byte, Register.bx, 0, 13);

addObjectCode(ObjectOperator.je, null, m\_objectCodeList.size()+17, label5);

addObjectCode(ObjectOperator.inc, Register.bx, null, null);

addObjectCode(ObjectOperator.long\_jmp, null,

m\_objectCodeList.size() - 6, label1);

addObjectCode(ObjectOperator.label, label2, null, null);

addObjectCode(ObjectOperator.cmp, Register.ax, 1, null);

addObjectCode(ObjectOperator.je, null, m\_objectCodeList.size() + 2,label3);

addObjectCode(ObjectOperator.mov\_byte, Register.bx, 0, 0);

addObjectCode(ObjectOperator.label, label3, null, null);

addObjectCode(ObjectOperator.inc, Register.bx, null, null);

addObjectCode(ObjectOperator.cmp\_byte, Register.bx, 0, 32);

addObjectCode(ObjectOperator.je, null, m\_objectCodeList.size() - 3,label3);

addObjectCode(ObjectOperator.label, label4, null, null);

addObjectCode(ObjectOperator.mov, Register.bp, 0, Register.bx);

addObjectCode(ObjectOperator.add, Register.bp, 2, null);

addObjectCode(ObjectOperator.inc, Register.ax, null, null);

addObjectCode(ObjectOperator.long\_jmp, null, m\_objectCodeList.size() - 19, label1);

addObjectCode(ObjectOperator.label, label5, null, null);

addObjectCode(ObjectOperator.mov\_byte, Register.bx, 0, 0);

addObjectCode(ObjectOperator.mov\_word, Register.bp, 0, 0);

addObjectCode(ObjectOperator.add, Register.bp, 2, null);

addObjectCode(ObjectOperator.mov, Register.bp, 6, Register.ax);

addObjectCode(ObjectOperator.mov, Register.bp, 8, Register.si);

}

public void generateStackHeapCheckFunction() {

{ List<Byte> byteList = new LinkedList<>();

Map<Integer,String> accessMap = new ListMap<>();

List<String> textList = new LinkedList<>();

GenerateInitializerStatic.generateByteTextList(Type.StringType,

Main.StackOverflowMessage, byteList, accessMap, textList);

Map<Integer,String> callMap = new ListMap<>();

Set<Integer> returnSet = new ListSet<>();

Map<Integer,Integer> byteToTextMap = new ListMap<>();

Symbol symbol = Symbol.createStaticSymbol(Main.StackOverflowName, byteList, accessMap, callMap, returnSet, byteToTextMap, textList);

Main.GlobalSet.add(symbol);

}

{ m\_objectCodeList.clear();

addObjectCode(ObjectOperator.add, Register.ax, ObjectCode.FrameRegister, null);

String label = Main.CheckStackHeapFunctionRegular + Main.SeparatorId + "1";

addObjectCode(ObjectOperator.jc, null, m\_objectCodeList.size() + 4, label);

addObjectCode(ObjectOperator.cmp, Register.ax, -2, null); // 65534

addObjectCode(ObjectOperator.ja, null, m\_objectCodeList.size() + 2, label);

addObjectCode(ObjectOperator.jmp, Register.bx, null, null);

addObjectCode(ObjectOperator.label, label, null, null);

addObjectCode(ObjectOperator.mov, Register.ah, 0x40, null);

addObjectCode(ObjectOperator.mov, Register.bx, 1, null);

addObjectCode(ObjectOperator.mov, Register.cx, Main.StackOverflowMessage.length(), null);

Pair<String,Integer> pair = new Pair<>(Main.StackOverflowName, 0);

addObjectCode(ObjectOperator.mov, Register.dx, pair, null);

addObjectCode(ObjectOperator.interrupt, 0x21, null, null);

addObjectCode(ObjectOperator.mov, Register.ah, 0x4C, null);

addObjectCode(ObjectOperator.mov, Register.al, -1, null);

addObjectCode(ObjectOperator.interrupt,0x21,null,null);

generateJumpInfo();

List<Byte> byteList = new LinkedList<>();

Map<Integer,String> accessMap = new ListMap<>();

Map<Integer,String> callMap = new ListMap<>();

Set<Integer> returnSet = new ListSet<>();

Map<Integer,Integer> byteToTextMap = new ListMap<>();

List<String> textList = new LinkedList<>();

generateTargetByteList(byteList, accessMap, callMap, returnSet, byteToTextMap, textList);

Symbol symbol =

Symbol.createStaticSymbol(Main.CheckStackHeapFunctionRegular, byteList, accessMap,

callMap, returnSet, byteToTextMap, textList);

Main.GlobalSet.add(symbol);

}

{ m\_objectCodeList.clear();

addObjectCode(ObjectOperator.add, Register.ax, ObjectCode.EllipseRegister, null);

String label = Main.CheckStackHeapFunctionEllipse + Main.SeparatorId + "1";

addObjectCode(ObjectOperator.jc, null, m\_objectCodeList.size() + 4, label);

addObjectCode(ObjectOperator.cmp, Register.ax, -2, null); // 65534

addObjectCode(ObjectOperator.ja, null, m\_objectCodeList.size() + 2, label);

addObjectCode(ObjectOperator.jmp, Register.bx, null, null);

addObjectCode(ObjectOperator.label, label, null, null);

addObjectCode(ObjectOperator.mov, Register.ah, 0x40, null);

addObjectCode(ObjectOperator.mov, Register.bx, 1, null);

addObjectCode(ObjectOperator.mov, Register.cx, Main.StackOverflowMessage.length(),null);

Pair<String,Integer> pair = new Pair<>(Main.StackOverflowName, 0);

addObjectCode(ObjectOperator.mov, Register.dx, pair, null);

addObjectCode(ObjectOperator.interrupt, 0x21, null, null);

addObjectCode(ObjectOperator.mov, Register.ah, 0x4C, null);

addObjectCode(ObjectOperator.mov, Register.al, -1, null);

addObjectCode(ObjectOperator.interrupt, 0x21, null, null);

generateJumpInfo();

List<Byte> byteList = new LinkedList<>();

Map<Integer,String> accessMap = new ListMap<>();

Map<Integer,String> callMap = new ListMap<>();

Set<Integer> returnSet = new ListSet<>();

Map<Integer,Integer> byteToTextMap = new ListMap<>();

List<String> textList = new LinkedList<>();

generateTargetByteList(byteList, accessMap, callMap, returnSet, byteToTextMap, textList);

Symbol symbol = Symbol.createStaticSymbol(Main.CheckStackHeapFunctionEllipse, byteList, accessMap, callMap, returnSet, byteToTextMap, textList);

Main.GlobalSet.add(symbol);

}

}

// ------------------------------------------------------------------------

private static int byteListSize(ObjectCode objectCode) {

if (objectCode.isCallRegular() || (objectCode.getOperator() == ObjectOperator.long\_jmp)) {

return ObjectCode.LongJumpSize;

}

else if (objectCode.isRelation() || (objectCode.getOperator() == ObjectOperator.short\_jmp)) {

return ObjectCode.RelationSize;

}

else {

return objectCode.byteList().size();

}

}

public void generateJumpInfo() {

generatMiddleToObjectJumps();

Map<Integer,Integer> objectToByteMap = new ListMap<>();

iterateOverJumps(objectToByteMap);

generateObjectToAbsoluteByteJumps(objectToByteMap);

}

private void generatMiddleToObjectJumps() {

for (int line = 0; line < m\_objectCodeList.size(); ++line) {

ObjectCode objectCode = m\_objectCodeList.get(line);

if ((objectCode.isRelation() || objectCode.isJump()) &&

(objectCode.getOperand(3) instanceof Integer)) {

Assert.error((objectCode.getOperand(1) == null) &&

(objectCode.getOperand(2) == null), "middle to object");

int middleTarget = (int) objectCode.getOperand(3);

objectCode.setOperand(2, m\_middleToObjectMap.get(middleTarget));

objectCode.setOperand(3, Main.FuncSymbol.getUniqueName() + Main.SeparatorId + middleTarget);

}

}

}

public void iterateOverJumps(Map<Integer,Integer> objectToByteMap) {

boolean update = true;

while (update) {

int byteSize = 0;

objectToByteMap.clear();

for (int line = 0; line < m\_objectCodeList.size(); ++line) {

objectToByteMap.put(line, byteSize);

byteSize += byteListSize(m\_objectCodeList.get(line));

}

objectToByteMap.put(m\_objectCodeList.size(), byteSize);

update = false;

for (int line = 0; line < (m\_objectCodeList.size() - 1); ++line) {

ObjectCode thisCode = m\_objectCodeList.get(line),

nextCode = m\_objectCodeList.get(line + 1);

if (thisCode.isRelation() && nextCode.isJump()) {

Assert.error((thisCode.getOperand(1) == null) &&

(thisCode.getOperand(2) instanceof Integer) &&

(thisCode.getOperand(3) instanceof String), "this code");

Assert.error((nextCode.getOperand(1) == null) &&

(nextCode.getOperand(2) instanceof Integer) &&

(nextCode.getOperand(3) instanceof String), "next code");

int thisTarget = (int) thisCode.getOperand(2);

Assert.error(thisTarget == (line + 2), "this target");

int nextTarget = (int) nextCode.getOperand(2);

int fromByteAddress = objectToByteMap.get(line + 2),

toByteAddress = objectToByteMap.get(nextTarget);

int byteDistance = toByteAddress - fromByteAddress;

int nextSize = byteListSize(nextCode);

if (((byteDistance + nextSize) >= -128) && (byteDistance <= 127)) {

thisCode.setOperator(m\_inverseMap.get(thisCode.getOperator()));

thisCode.setOperand(2, nextTarget);

thisCode.setOperand(3, nextCode.getOperand(3));

nextCode.setOperator(ObjectOperator.empty);

update = true;

}

}

else if ((thisCode.isRelation() || thisCode.isJump())) {

Assert.error((thisCode.getOperand(1) == null) &&

(thisCode.getOperand(2) instanceof Integer) &&

(thisCode.getOperand(3) instanceof String), "this code");

int thisTarget = (int) thisCode.getOperand(2);

int fromByteAddress = objectToByteMap.get(line + 1),

toByteAddress = objectToByteMap.get(thisTarget);

int byteDistance = toByteAddress - fromByteAddress;

if (byteDistance == 0) {

thisCode.setOperator(ObjectOperator.empty);

update = true;

}

else if ((thisCode.getOperator() == ObjectOperator.long\_jmp) &&

(byteDistance >= -129) && (byteDistance <= 127)) {

thisCode.setOperator(ObjectOperator.short\_jmp);

update = true;

}

}

}

}

}

private void generateObjectToAbsoluteByteJumps(Map<Integer,Integer> objectToByteMap) {

for (int line = 0; line < m\_objectCodeList.size(); ++line) {

ObjectCode objectCode = m\_objectCodeList.get(line);

if (objectCode.isRelation() || objectCode.isJump()) {

Assert.error((objectCode.getOperand(1) == null) &&

(objectCode.getOperand(2) instanceof Integer) &&

(objectCode.getOperand(3) instanceof String),

"object to absolute byte");

int objectTarget = (int) objectCode.getOperand(2);

int fromByteAddress = objectToByteMap.get(line + 1),

toByteAddress = objectToByteMap.get(objectTarget);

objectCode.setOperand(1, toByteAddress - fromByteAddress);

}

}

}

// ------------------------------------------------------------------------

private int operandSize(ObjectOperator operator, Object operand) {

if (operator.name().contains("mov\_")) {

return ObjectCode.operatorSize(operator);

}

else if (operator.name().contains("cmp\_")) {

return Math.max(1, valueSize(operand));

}

else {

return valueSize(operand);

}

}

private int valueSize(Object operand) {

if (operand instanceof Integer) {

return ObjectCode.valueSize((int) operand);

}

return 0;

}

public void generateTargetByteList(List<Byte> byteList, Map<Integer,String> accessMap,

Map<Integer,String> callMap, Set<Integer> returnSet,

Map<Integer,Integer> byteToTextMap, List<String> textList) {

for (int line = 0; line < m\_objectCodeList.size(); ++line) {

ObjectCode objectCode = m\_objectCodeList.get(line);

byteList.addAll(objectCode.byteList());

textList.add(objectCode.toString());

String returnLabel = objectCode.getReturnLabel();

if (objectCode.isRelation() || objectCode.isJump()) {

// Empty.

}

else if ((objectCode.getOperator() == ObjectOperator.empty) ||

(objectCode.getOperator() == ObjectOperator.comment) ||

(objectCode.getOperator() == ObjectOperator.label)) {

// Empty.

}

else if ((objectCode.getOperator() == ObjectOperator.call) &&

(objectCode.getOperand(1) instanceof String)) {

String calleeName = (String) objectCode.getOperand(1);

int address = byteList.size() - Type.PointerSize;

callMap.put(address, calleeName);

byteToTextMap.put(address, textList.size() - 1);

}

else if (returnLabel != null) {

int address = byteList.size() - Type.PointerSize;

returnSet.add(address);

}

else if (objectCode.getOperand(1) instanceof Pair) {

int address = byteList.size() - valueSize(objectCode.getOperand(2)) - valueSize(objectCode.getOperand(3)) - Type.PointerSize;

String name = (String) ((Pair) objectCode.getOperand(1)).getFirst();

accessMap.put(address, name);

}

else if (objectCode.getOperand(2) instanceof Pair) {

int address = byteList.size() - operandSize(objectCode.getOperator(), objectCode.getOperand(3)) - Type.PointerSize;

String name = (String) ((Pair) objectCode.getOperand(2)).getFirst();

accessMap.put(address, name);

}

else if (objectCode.getOperand(3) instanceof Pair) {

int address = byteList.size() - Type.PointerSize;

String name = (String) ((Pair) objectCode.getOperand(3)).getFirst();

accessMap.put(address, name);

}

else if (objectCode.getOperand(1) instanceof String) {

Main.DebugStream2.println("1: " + objectCode.getOperand(1));

}

else if (objectCode.getOperand(2) instanceof String) {

Main.DebugStream2.println("2: " + objectCode.getOperand(2));

}

else if (objectCode.getOperand(3) instanceof String) {

Main.DebugStream2.println("3: " + objectCode.getOperand(3));

}

}

}

}

# The Linker

## The Linker Class

The LinkerInfo class holds the information of a function definition or a static variable:

* m\_fullName: the full name of the function or static variable. The full name of a function or global variable with external linkage is just its regular name, while the full name of
* m\_entryPoint: the function call entry point, which may differ from the start address of the function[[9]](#footnote-9).
* m\_accessMap: the map of all accesses of function or static variables.
* m\_callMap: the map of all function calls. The reason we distinguish between m\_accessMap and m\_callMap is that accesses are absolute while function calls are relative.
* m\_returnSet: the set of all return address assignments.
* m\_byteList: the actual code of the function or initialization data of the static variable. If the static variable is uninitialized, all data is zero.

The Linker class merge code of the object files and modifies the global accesses, function calls, and return assignments and saves the final code in a com-file, which is executable in 16-bits Windows. More specific, the linker has the following tasks:

* First we load all function and static variables from all the object files and to make sure that two elements do not share the same name.
* The we identify the main function and trace all functions and global variables reachable from main. All elements not reachable from main are discarded.
* For each function or global variable we modifies all accesses, and for each function we modify all calls and return assignments.
* Finally, we save the code and data for all elements reachable from main in the com-file.

There are several maps:

* m\_globalMap: holds all functions and static variables of all object files.
* m\_globalList: holds all functions and static variables reachable from main, where main is always placed first.
* m\_addressMap holds the beginning of each function or static variables in the final code.
* m\_entryMap holds the entry point of each function.[[10]](#footnote-10)

Linker.java

package c\_compiler;

import java.io.\*;

import java.util.\*;

public class Linker {

private int m\_totalSize = 256;

private Map<String,Symbol> m\_globalMap = new ListMap<>();

private List<Symbol> m\_globalList = new LinkedList<>();

private Map<String,Integer> m\_addressMap = new ListMap<>(),

m\_entryMap = new ListMap<>();

private Symbol m\_stackTop = null;

private File m\_comFile, m\_asmFile;

public Linker(File comFile, File asmFile) {

m\_comFile = comFile;

m\_asmFile = asmFile;

}

To begin with, addAll is called by Main class of Section 20. If the same symbol is loaded from two object files, unless the name starts with “x#” (in that case it is a numerical constant, which is allowed in more than one object file), an error message is reported. All loaded elements are stored in m\_globalMap.

public void add(Symbol symbol) {

String name = symbol.getUniqueName();

if (name.equals(Main.StackTopName)) {

m\_stackTop = symbol;

}

else {

Symbol result = m\_globalMap.put(name, symbol);

Assert.error((result == null) || name.endsWith(Main.NumberId),

simpleName(name), "duplicate global identifier");

}

}

### Generation

The generate method traces the elements reachable from main by calling generateTrace, and sets the total size of the code at the beginning of the main code to initialize the stack.

public void generate() throws Exception {

m\_globalMap.put(Main.PathName, generatePathSymbol());

{ Symbol mainInfo = m\_globalMap.get("main");

Assert.error(mainInfo != null, "main", "function missing");

generateTrace(mainInfo);

m\_addressMap.put(m\_stackTop.getUniqueName(), m\_totalSize);

m\_globalList.add(m\_stackTop);

m\_totalSize += m\_stackTop.getByteList().size();

}

for (Symbol symbol : m\_globalList) {

List<Byte> byteList = symbol.getByteList();

int startAddress = m\_addressMap.get(symbol.getUniqueName());

generateAccess(symbol.getAccessMap(), byteList);

generateCall(startAddress, symbol.getCallMap(), byteList,

symbol.getTextList(), symbol.getByteToTextMap());

generateReturn(startAddress, symbol.getReturnSet(), byteList);

}

for (Symbol symbol : m\_globalList) {

int startAddress = m\_addressMap.get(symbol.getUniqueName());

splitTextList(symbol.getTextList());

generateTextAddress(startAddress, symbol.getByteList(),

symbol.getTextList());

}

{ System.out.println("Generating \"" + m\_comFile.getPath() + "\".");

OutputStream comStream = new FileOutputStream(m\_comFile);

for (Symbol symbol : m\_globalList) {

for (Byte b : symbol.getByteList()) {

comStream.write(b);

}

}

comStream.close();

}

{ System.out.println("Generating \"" + m\_asmFile.getPath() + "\".");

PrintStream asmStream = new PrintStream(m\_asmFile);

asmStream.println("\torg 100h");

for (Symbol symbol : m\_globalList) {

for (String text : symbol.getTextList()) {

if (text != null) {

asmStream.println(text);

}

}

}

asmStream.println("x" + m\_totalSize + ":");

asmStream.close();

}

}

private Symbol generatePathSymbol() {

List<Byte> byteList = new LinkedList<>();

Map<Integer,String> accessMap = new ListMap<>(),

callMap = new ListMap<>();

Set<Integer> returnSet = new ListSet<>();

Map<Integer,Integer> byteToTextMap = new ListMap<>();

List<String> textList = new LinkedList<>();

GenerateInitializerStatic.generateByteTextList

(Type.StringType, m\_comFile.getPath(),byteList, accessMap, textList);

Symbol symbol =

Symbol.createStaticSymbol(Main.PathName, byteList, accessMap,

callMap, returnSet, byteToTextMap, textList);

symbol.setByteList(byteList);

symbol.setTextList(textList);

return symbol;

}

### Main Trace Generation

The generateTrace method looks into the globalInfo parameter, and adds it to m\_globalList. However, we must check that the element is not already a member since recursive function calls may occur. The start address and entry point of the element is added m\_addressMap and m\_entryMap, respectively. The m\_totalSize field keeps track of the current size of the code.

Then we continue to call generateTrace recursively for each function call in the call map and global variables access in the access map. If any function or variable is missing, an error message is reported.

private void generateTrace(Symbol symbol) {

if (!m\_globalList.contains(symbol)) {

m\_globalList.add(symbol);

m\_addressMap.put(symbol.getUniqueName(), m\_totalSize);

m\_entryMap.put(symbol.getUniqueName(), m\_totalSize +

symbol.getEntryPoint());

m\_totalSize += symbol.getByteList().size();

Set<String> accessNameSet =

new ListSet<>(symbol.getAccessMap().values());

for (String accessName : accessNameSet) {

Symbol accessSymbol = m\_globalMap.get(accessName);

if (!accessName.equals(Main.StackTopName)) {

Assert.error(accessSymbol != null, "Linkage",

"missing external variable \"" + simpleName(accessName)+

"\" in \"" + simpleName(symbol.getUniqueName()) +"\"");

generateTrace(accessSymbol);

}

}

Set<String> callNameSet = new ListSet<>(symbol.getCallMap().values());

for (String callName : callNameSet) {

Symbol funcSymbol = m\_globalMap.get(callName);

Assert.error(funcSymbol != null, callName,

"missing extern function \"" +

simpleName(callName) + "\" in \"" +

simpleName(symbol.getUniqueName()) + "\"");

generateTrace(funcSymbol);

}

}

}

For each element, reachable from main, we need to modify its accesses, function calls and return assignment (for a global variable, the call map and return set are empty) by calling generateAccess, generateCall, and generateReturn, respectivly.

The generateAccess method looks up the name and address of each global access. The address of the name is looked up in accessMap, and original address is calculated by adding the low anad high byte of the address. Note that the access may be not zero in some cases, such as pointer arithmentic with a static pointer and constant integer. In the code below, the initial value of p will be two, since the address is two bytes higher than the address of a; generateAccess will then look up and add the address of a in order to properly initialize p.

static char arr[10];

static int p = &a[2];

private void generateAccess(Map<Integer,String> accessMap,

List<Byte> byteList) {

for (Map.Entry<Integer,String> entry : accessMap.entrySet()) {

int address = entry.getKey();

String name = entry.getValue();

int oldLowByte = checkValue(byteList.get(address)),

oldHighByte = checkValue(byteList.get(address + 1));

int oldTarget = (oldHighByte << 8) + oldLowByte;

int newTarget = oldTarget + m\_addressMap.get(name);

byte newLowByte = (byte) newTarget,

newHighByte = (byte) (newTarget >> 8);

byteList.set(address, newLowByte);

byteList.set(address + 1, newHighByte);

}

}

private static int checkValue(int i) {

return (i < 0) ? (i + 256) : i;

}

The generateCall method sets the address of each function call by looking up the function. Unlike the access case above, we do not have to look up the original address, it is always zero. However, since jumps are relative we need to calculate the relative address between the source and target. Note that the caller address is the address of the instruction following the caller instruction, which is the address stored in callMap plus the start address of the calling function plus another two bytes.

Moreover, we have two special cases and two general cases. ObjectCodeGenerator of Section 16.3 generated long jumps for each function call; that is, a long jump instruction followed two bytes holding the relative address. However, if the relative address only one byte, we change to a short jump instruction followed by one byte holding the relative address and insert a nop instruction in order to keep the size of the code[[11]](#footnote-11).

The first special case occurs when the relative address is 127, in which case we set 127 as the low byte jump address and 0 as the high byte. The second special case is when the relative address is -129, in which case we change the jump instruction from long jump to short jump with the relative address -128.

The first general case occurs when the relative address is between -128 and 126, inclusive. In that case, we change the jump instruction from long the short jump and insert a nop instruction. Finally, in all other cases we set the low and high byte of the relative jump address.

private void generateCall(int startAddress, Map<Integer,String> callMap,

List<Byte> byteList, List<String> textList,

Map<Integer,Integer> byteToTextMap) {

for (Map.Entry<Integer,String> entry : callMap.entrySet()) {

int address = entry.getKey();

int callerAddress = startAddress + address + 2;

int calleeAddress = m\_entryMap.get(entry.getValue());

int relativeAddress = calleeAddress - callerAddress;

if (relativeAddress == -129) {

byteList.set(address - 1,(byte) ObjectCode.ShortJumpOperator);

byteList.set(address, (byte) -128);

byteList.set(address + 1, (byte) ObjectCode.NopOperator);

}

else if ((relativeAddress >= -128) && (relativeAddress <= 127)) {

byteList.set(address - 1, (byte) ObjectCode.NopOperator);

byteList.set(address, (byte) ObjectCode.ShortJumpOperator);

byteList.set(address + 1, (byte) relativeAddress);

int textLine = byteToTextMap.get(address);

Assert.error(textLine < textList.size(), "text line");

String text = textList.get(textLine);

textList.set(textLine,"\tnop\t; 1\n" + text.replace("\t; 3", "\t; 2"));

}

else {

byteList.set(address, (byte) relativeAddress);

byteList.set(address + 1, (byte) (relativeAddress >> 8));

}

}

}

The generateReturn method modifies all return address assignments. When a function call is generated in Section **Fel! Hittar inte referenskälla.**, the return assignment address is relative to the assignment itself, we need to add the function start address and the address of the return assignment to the relative address.

private void generateReturn(int startAddress, Set<Integer> returnSet,

List<Byte> byteList) {

for (int address : returnSet) {

int relativeLowByte = checkValue(byteList.get(address)),

relativeHighByte = checkValue(byteList.get(address + 1));

int relativeAddress = (relativeHighByte << 8) + relativeLowByte;

int globalAddress = startAddress + address + relativeAddress;

byte globalLowByte = (byte) globalAddress;

byte globaHighByte = (byte) (globalAddress >> 8);

byteList.set(address, globalLowByte);

byteList.set(address + 1, globaHighByte);

}

}

private void generateTextAddress(int address, List<Byte> byteList,

List<String> textList) {

int byteIndex = 0;

for (int line = 0; line < textList.size(); ++line) {

String text = textList.get(line);

if ((text != null) && !text.isEmpty() && !text.contains(":\t;") &&

!text.startsWith(";") && !text.endsWith(":")) {

StringBuilder buffer = new StringBuilder(text);

buffer.insert(0, "x" + address + ":");

int semiIndex = buffer.lastIndexOf("; "), size = 0;

try {

size = Integer.valueOf(buffer.substring(semiIndex + 2));

}

catch (NumberFormatException exception) {

Assert.error(exception.getMessage());

}

buffer.append(":");

for (int index = 0; index < size; ++index) {

buffer.append(" " + byteList.get(byteIndex++));

}

textList.set(line, buffer.toString());

address += size;

}

}

Assert.error(byteIndex == byteList.size(), "linker byte list");

}

private void splitTextList(List<String> textList) {

for (int index = (textList.size() - 1); index >= 0; --index) {

String text = textList.get(index);

int newLineIndex = text.indexOf("\n");

if (newLineIndex != -1) {

textList.set(index, text.substring(0, newLineIndex));

textList.add(index + 1, text.substring(newLineIndex + 1));

}

}

}

String simpleName(String name) {

int index = name.lastIndexOf(Main.SeparatorId);

return ((index != -1)) ? name.substring(0, index) : name;

}

}

# Auxiliary Classes

Java comes with a large class library, with functionality for almost all requirements. However, there are some classes that are missing, which we need to write ourselves: a class for error handling, container classes, and a graph class.

## Error Handling

To begin with, we need error handling, a way to notify the programmer of errors and warnings. In the C Standard Library there is a macro called assert, and in this application we imitate that macro by defining a class with the corresponding behavior. It has the two method sets error and warning, that writes a message. The difference between them is that error stops the execution.

Assert.java

package c\_compiler;

import java.io.\*;

public class Assert {

public static void error(boolean test, String message) {

error(test, null, message);

}

public static void error(boolean test) {

error(test, null, null);

}

public static void error(boolean test, Object object, String message) {

if (!test) {

error((object != null) ? object.toString() : null, message);

}

}

public static void error(String message) {

error(null, message);

}

public static void warning(String message) {

warning(null, message);

}

public static void error(Object object, String message) {

message("Error", (object != null) ? object.toString() : null, message);

Main.DebugStream.close();

Main.DebugStream2.close();

System.exit(-1);

}

public static void warning(boolean test, Object object, String message) {

if (!test) {

warning((object != null) ? object.toString() : null, message);

}

}

public static void warning(Object object, String message) {

if (Main.SmallWarning) {

message("Warning", (object != null) ? object.toString() : null, message);

if (Main.LargeWarning) {

System.exit(-1);

}

}

}

private static void message(String type, String name, String message) {

String funcText = (Main.FuncSymbol != null) ? (" in function \"" +

Main.FuncSymbol.getName() + "\"") : "";

PrintStream errorStream = (Main.ErrorStream != null) ?

Main.ErrorStream : System.err;

if ((name != null) && (Main.Path != null)) {

errorStream.println(type + " in file \"" + Main.Path + "\"" + funcText +

" at line " + Main.Line + ". \"" + name + "\": " + message + ".");

}

else if ((name == null) && (Main.Path != null)) {

errorStream.println(type + " in file \"" + Main.Path + "\"" + funcText +

" at line " + Main.Line + ": " + message + ".");

}

else if ((name != null) && (Main.Path == null)) {

errorStream.println(type + ": \"" + name + "\", " + message + ".");

}

else if ((name != null) && (Main.Path == null)) {

errorStream.println(type + ": \"" + name + "\", " + message + ".");

}

}

}

## Container Classes

Java has a large class library holding many container classes. However, there are no classes for ordered or unordered pairs, there is also no classes for triples or tuples.

### Ordered Pair

In the preprocessor (and in several other classes in the following chapters) we also need the auxiliary class Pair.

Pair.java

package c\_compiler;

public class Pair<FirstType,SecondType> {

protected FirstType m\_first;

protected SecondType m\_second;

public Pair(FirstType first, SecondType second) {

m\_first = first;

m\_second = second;

}

public FirstType getFirst() {

return m\_first;

}

public void setFirst(FirstType first) {

m\_first = first;

}

public SecondType getSecond() {

return m\_second;

}

public void setSecond(SecondType second) {

m\_second = second;

}

Two pairs are considered equal if (for both the first and second value) the values ar both null, or they are both not null and the values are equal (the equals method returns true).

public boolean equals(Object object) {

if (object instanceof Pair) {

Pair pair = (Pair) object;

return equalsFirst(pair) && equalsSecond(pair);

}

return false;

}

private boolean equalsFirst(Pair pair) {

return (((m\_first == null) && (pair.m\_first == null)) ||

((m\_first != null) && (pair.m\_first != null) &&

m\_first.equals(pair.m\_first)));

}

private boolean equalsSecond(Pair pair) {

return (((m\_second == null) && (pair.m\_second == null)) ||

((m\_second != null) && (pair.m\_second != null) &&

m\_second.equals(pair.m\_second)));

}

public String toString() {

String firstText = (m\_first != null) ? m\_first.toString() : "<null>",

secondText = (m\_second != null) ? m\_second.toString() : "<null>";

return "(" + firstText + "," + secondText + ")";

}

}

### Unordered Pair

UnPair is a subclass of Pair, the difference is that the pair is unordered and that two pair are considered equals even if their values are switched.

UnorderdPair.java

package c\_compiler;

public class UnorderedPair<FirstType,SecondType>

extends Pair<FirstType,SecondType>{

public UnorderedPair(FirstType first, SecondType second) {

super(first, second);

}

public boolean equals(Object object) {

if (object instanceof UnorderedPair) {

UnorderedPair pair = (UnorderedPair) object;

return super.equals(pair) ||

(equalsFirstSecond(pair) && equalsSecondFirst(pair));

}

return false;

}

private boolean equalsFirstSecond(UnorderedPair pair) {

return (((m\_first == null) && (pair.m\_second == null)) ||

((m\_first != null) && (pair.m\_second != null) &&

m\_first.equals(pair.m\_second)));

}

private boolean equalsSecondFirst(UnorderedPair pair) {

return (((m\_second == null) && (pair.m\_first == null)) ||

((m\_second != null) && (pair.m\_first != null) &&

m\_second.equals(pair.m\_first)));

}

public String toString() {

return "(" + m\_first.toString() + "," + m\_second.toString() + ")";

}

}

### Triple

Moreover, we also need the class Triple, to hold three values. The Triple class is a direct sub class of Pair, it only adds the third value.

Triple.java

package c\_compiler;

public class Triple<FirstType,SecondType,ThirdType>

extends Pair<FirstType,SecondType> {

protected ThirdType m\_third;

public Triple(FirstType first, SecondType second, ThirdType third) {

super(first, second);

m\_third = third;

}

public ThirdType getThird() {

return m\_third;

}

public void setThird(ThirdType third) {

m\_third = third;

}

public boolean equals(Object object) {

if (object instanceof Triple) {

Triple triple = (Triple) object;

return super.equals((Pair) triple) &&

(((m\_third == null) && (triple.m\_third == null)) ||

((m\_third != null) && (triple.m\_third != null) &&

m\_third.equals(triple.m\_third)));

}

return false;

}

}

### Tuple

Tuple is a direct subclass of Triple.

Tuple.java

package c\_compiler;

public class Tuple<FirstType,SecondType,ThirdType,FourthType>

extends Triple<FirstType,SecondType,ThirdType> {

private FourthType m\_fourth;

public Tuple(FirstType first, SecondType second,

ThirdType third, FourthType fourth) {

super(first, second, third);

m\_fourth = fourth;

}

public FourthType getFourth() {

return m\_fourth;

}

public boolean equals(Object object) {

if (object instanceof Tuple) {

Tuple tuple = (Tuple) object;

return super.equals((Triple) tuple) &&

(((m\_fourth == null) && (tuple.m\_fourth == null)) ||

((m\_fourth != null) && (tuple.m\_fourth != null) &&

m\_fourth.equals(tuple.m\_fourth)));

}

return false;

}

}

## Undirected Graph

Graph implements a general unordered graph. It holds of set of vertices and a set of edges.

Graph.java

package c\_compiler;

import java.util.\*;

public class Graph<VertexType> {

private Set<VertexType> m\_vertexSet = new ListSet<>();

private Set<UnorderedPair<VertexType,VertexType>> m\_edgeSet = new ListSet<>();

public Graph() {

m\_vertexSet = new ListSet<>();

m\_edgeSet = new ListSet<>();

}

public Graph(Set<VertexType> vertexSet) {

m\_vertexSet = vertexSet;

m\_edgeSet = new ListSet<>();

}

public Graph(Set<VertexType> vertexSet,

Set<UnorderedPair<VertexType,VertexType>> edgeSet) {

m\_vertexSet = vertexSet;

m\_edgeSet = edgeSet;

}

public Set<VertexType> getVertexSet() {

return m\_vertexSet;

}

public Set<UnorderedPair<VertexType,VertexType>> getEdgeSet() {

return m\_edgeSet;

}

The neighbourSet method goes through all edges and add each found neighbor to the vertex.

public Set<VertexType> getNeighbourSet(VertexType vertex) {

Set<VertexType> neighbourSet = new ListSet<>();

for (UnorderedPair<VertexType,VertexType> edge : m\_edgeSet) {

if (edge.getFirst() == vertex) {

neighbourSet.add(edge.getSecond());

}

if (edge.getSecond() == vertex) {

neighbourSet.add(edge.getFirst());

}

}

return neighbourSet;

}

### Addition and Removal of Vertices and Edges

public void addVertex(VertexType vertex) {

m\_vertexSet.add(vertex);

}

public void eraseVertex(VertexType vertex) {

m\_vertexSet.remove(vertex);

}

public void addEdge(VertexType vertex1, VertexType vertex2) {

m\_edgeSet.add(new UnorderedPair<>(vertex1, vertex2));

}

public void addEdge(UnorderedPair<VertexType,VertexType> edge) {

m\_edgeSet.add(edge);

}

public void eraseEdge(VertexType vertex1, VertexType vertex2) {

m\_edgeSet.remove(new UnorderedPair<>(vertex1, vertex2));

}

public void eraseEdge(UnorderedPair<VertexType,VertexType> edge) {

m\_edgeSet.remove(edge);

}

### Graph Partition

The method partitionate divides the graph into free subgraphs; that is, subgraphs which vertices have no neighbors in any of the other free subgraphs. First we go through the vertices and perform a deep search to find all vertices reachable from the vertex. Then we generate a subgraph for each such vertex set.

public Set<Graph<VertexType>> partitionate() {

Set<Set<VertexType>> qliqueSet = new ListSet<>();

for (VertexType vertex : m\_vertexSet) {

Set<VertexType> vertexSet = new ListSet<>();

deepSearch(vertex, vertexSet);

qliqueSet.add(vertexSet);

}

Set<Graph<VertexType>> graphSet = new ListSet<>();

for (Set<VertexType> vertexSet : qliqueSet) {

graphSet.add(generateSubGraph(vertexSet));

}

return graphSet;

}

The deepSearch method search recursively through the graph in order to find all vertices reachable from the given vertex. To avoid cyclic search, the search is terminated if the vertex is already a member of the result set.

private void deepSearch(VertexType vertex, Set<VertexType> resultSet) {

if (!resultSet.contains(vertex)) {

resultSet.add(vertex);

Set<VertexType> neighbourSet = getNeighbourSet(vertex);

for (VertexType neighbour : neighbourSet) {

deepSearch(neighbour, resultSet);

}

}

}

The generateSubGraph method goes through the edge set and add all edges which both end vertices are members of the vertex set.

public Graph<VertexType> generateSubGraph(Set<VertexType> vertexSet) {

Set<UnorderedPair<VertexType,VertexType>> resultEdgeSet = new ListSet<>();

for (UnorderedPair<VertexType,VertexType> edge : m\_edgeSet) {

if (vertexSet.contains(edge.getFirst()) &&

vertexSet.contains(edge.getSecond())) {

resultEdgeSet.add(edge);

}

}

return (new Graph(vertexSet, resultEdgeSet));

}

public String toString() {

return "<" + m\_vertexSet + "," + m\_edgeSet + ">";

}

}

## Directed Graph

Besides the undirected graph, we also need the directed graph.

DirectedGraph.java

package c\_compiler;

import java.util.\*;

public class DirectedGraph<VertexType> {

private Set<VertexType> m\_vertexSet = new ListSet<>();

private Set<Pair<VertexType,VertexType>> m\_edgeSet = new ListSet<>();

public DirectedGraph() {

m\_vertexSet = new ListSet<>();

m\_edgeSet = new ListSet<>();

}

public DirectedGraph(Set<VertexType> vertexSet) {

m\_vertexSet = vertexSet;

m\_edgeSet = new ListSet<>();

}

public DirectedGraph(Set<VertexType> vertexSet,

Set<Pair<VertexType,VertexType>> edgeSet) {

m\_vertexSet = vertexSet;

m\_edgeSet = edgeSet;

}

public Set<VertexType> getVertexSet() {

return m\_vertexSet;

}

public Set<VertexType> getInNeighbourSet(VertexType vertex) {

Set<VertexType> inNeighbourSet = new ListSet<>();

for (Pair<VertexType,VertexType> edge : m\_edgeSet) {

if (edge.getSecond() == vertex) {

inNeighbourSet.add(edge.getFirst());

}

}

return inNeighbourSet;

}

public Set<VertexType> getOutNeighbourSet(VertexType vertex) {

Set<VertexType> outNeighbourSet = new ListSet<>();

for (Pair<VertexType,VertexType> edge : m\_edgeSet) {

if (edge.getFirst() == vertex) {

outNeighbourSet.add(edge.getSecond());

}

}

return outNeighbourSet;

}

public Set<Pair<VertexType,VertexType>> getEdgeSet() {

return m\_edgeSet;

}

// ------------------------------------------------------------------------

public void addVertex(VertexType vertex) {

m\_vertexSet.add(vertex);

}

public void eraseVertex(VertexType vertex) {

m\_vertexSet.remove(vertex);

}

public void addEdge(VertexType vertex1, VertexType vertex2) {

m\_edgeSet.add(new Pair<>(vertex1, vertex2));

}

public void addEdge(Pair<VertexType,VertexType> edge) {

m\_edgeSet.add(edge);

}

public void eraseEdge(VertexType vertex1, VertexType vertex2) {

m\_edgeSet.remove(new Pair<>(vertex1, vertex2));

}

public void eraseEdge(Pair<VertexType,VertexType> edge) {

m\_edgeSet.remove(edge);

}

public Set<DirectedGraph<VertexType>> partitionate() {

Set<Set<VertexType>> qliqueSet = new ListSet<>();

for (VertexType vertex : m\_vertexSet) {

Set<VertexType> vertexSet = new ListSet<>();

deepSearch(vertex, vertexSet);

qliqueSet.add(vertexSet);

}

Set<DirectedGraph<VertexType>> graphSet = new ListSet<>();

for (Set<VertexType> vertexSet : qliqueSet) {

graphSet.add(generateSubGraph(vertexSet));

}

return graphSet;

}

private void deepSearch(VertexType vertex, Set<VertexType> resultSet) {

if (!resultSet.contains(vertex)) {

resultSet.add(vertex);

Set<VertexType> outNeighbourSet = getOutNeighbourSet(vertex);

for (VertexType outNeighbour : outNeighbourSet) {

deepSearch(outNeighbour, resultSet);

}

}

}

public DirectedGraph<VertexType> generateSubGraph(Set<VertexType>vertexSet){

Set<Pair<VertexType,VertexType>> resultEdgeSet = new ListSet<>();

for (Pair<VertexType,VertexType> edge : m\_edgeSet) {

if (vertexSet.contains(edge.getFirst()) &&

vertexSet.contains(edge.getSecond())) {

resultEdgeSet.add(edge);

}

}

return (new DirectedGraph(vertexSet, resultEdgeSet));

}

public String toString() {

return "<" + m\_vertexSet + "," + m\_edgeSet + ">";

}

}

# The Standard Library

## Integral and Floating Limits

Limits.h

#ifndef \_\_LIMITS\_H\_\_

#define \_\_LIMITS\_H\_\_

#define CHAR\_BIT 8

#define CHAR\_MAX 127s

#define CHAR\_MIN -128s

#define SCHAR\_MAX 127s

#define SCHAR\_MIN -128s

#define UCHAR\_MAX 255us

#define SHRT\_MAX 127s

#define SHRT\_MIN -128s

#define USHRT\_MAX 255us

#define INT\_MAX 32767

#define INT\_MIN -32768

#define UINT\_MAX 65535u

#define LONG\_MAX 2147483647l

#define LONG\_MIN -2147483647l

#define ULONG\_MAX 4294967295lu

#endif

Float.h

#ifndef \_\_FLOAT\_H\_\_

#define \_\_FLOAT\_H\_\_

#define FLT\_RADIX 2

#define FLT\_ROUNDS 1

#define FLT\_DIG 6

#define FLT\_EPSILON 1e-6

#define FLT\_MANT\_DIG 2

#define FLT\_MAX 1e6

#define FLT\_MAX\_EXP 6

#define FLT\_MIN 1e-6

#define FLT\_MIN\_EXP -6

#define DBL\_DIG 6

#define DBL\_EPSILON 1e-6

#define DBL\_MANT\_DIG 2

#define DBL\_MAX 1e+6

#define DBL\_MAX\_EXP 6

#define DBL\_MIN 1e-6

#define DBL\_MIN\_EXP -6

#endif

## The Assert Macro

Assert.h

#ifndef \_\_ASSERT\_H\_\_

#define \_\_ASEERT\_H\_\_

#ifndef NDEBUG

#include <StdIO.h>

#include <StdLib.h>

#define assert(expression) if (!(expression)) { \\

fprintf(stderr, "Assertion failed: \"%s\", file \"%s\", line %i.\n", \\

#expression, \_\_FILE\_\_, \_\_LINE\_\_); abort(); }

#else

#define assert(expression)

#endif

#endif

## Locale Data

Locale.h

#ifndef \_\_LOCALE\_H\_\_

#define \_\_LOCALE\_H\_\_

#define LC\_COLLATE 0x01

#define LC\_CTYPE 0x02

#define LC\_MONETARY 0x04

#define LC\_NUMERIC 0x08

#define LC\_TIME 0x10

#define LC\_ALL 0x1F

struct lconv {

int summerTimeZone, // time

winterTimeZone;

char \*\*shortDayList;

char \*\*longDayList;

char \*\*shortMonthList;

char \*\*longMonthList;

char \*lowerCase; // ctype

char \*upperCase;

char \*\*messageList; // errno

};

extern char\* enMessageList[];

extern char\* setlocale(int flag, char\* name);

extern struct lconv \*localeconv(void);

#endif

Locale.c

#include <StdIO.h>

#include <StdLib.h>

#include <String.h>

#include <Locale.h>

static char\* enShortDayList[] = { "Sun", "Mon", "Tue", "Wed", "Thu", "Fri", "Sat" };

static char\* enLongDayList[] = { "Sunday", "Monday", "Tuesday", "Wednesday", "Thursday", "Friday", "Saturday" };

static char\* enShortMonthList[] = {"Jan", "Feb", "Mar", "Apr", "May", "Jun", "Jul", "Aug", "Sep", "Oct", "Nov", "Dec"};

static char\* enLongMonthList[] = {"January", "February", "March", "April", "May", "June",

"July", "August", "September", "October", "November", "December"};

char\* enMessageList[] = {"no error", "function number invalid",

"file not found", "path not found", "no handle available", "access denied",

"out of domain", "out of range", "invalid multibyte sequence",

"error while opening", "error while flushing", "error while closing", "open mode invalid",

"error while writing", "error while reading", "error while seeking",

"error while telling", "error while removing file", "error while renaming file"};

static struct lconv en\_US\_utf8 = {-5, -4, enShortDayList, enLongDayList, enShortMonthList, enLongMonthList,

"abcdefghijklmnopqrstuvwxyz", "ABCDEFGHIJKLMNOPQRSTUVWXYZ", enMessageList};

static char\* swShortDayList[] = {"Son", "Man", "Tis", "Ons", "Tor", "Fre", "Lor"};

static char\* swLongDayList[] = {"Sondag", "Mandag", "Tisdag", "Onsdag", "Torsdag", "Fredag", "Lordag"};

static char\* swShortMonthList[] = {"Jan", "Feb", "Mar", "Apr", "Maj", "Jun", "Jul", "Aug", "Sep", "Okt", "Nov", "Dec"};

static char\* swLongMonthList[] = {"Januari", "Februari", "Mars", "April", "Maj", "Juni", "Juli", "Augusit", "September", "Oktober", "November", "December"};

static char\* swMessageList[] = {"inga fel", "felaktigt functionsnummer",

"hittar ej filen", "hittar ej sokvagen", "inget handtag tillgangligt", "atkomst nekad",

"utanfor doman", "utanfor range", "felaktig multibyte-sekvens",

"fel vid oppning", "fel vid flushing", "fel vid stangning", "fel oppningslage",

"fel vid skrivning", "fel vid lasning", "fel vid sokning",

"fel vid telling", "fel vid borttagning av fil", "fel vid namnbyte av fil"};

static struct lconv sw\_EN\_utf8 = {1, 2, swShortDayList, swLongDayList, enShortMonthList, swLongMonthList,

"abcdefghijklmnopqrstuvwxyz", "ABCDEFGHIJKLMNOPQRSTUVWXYZ", swMessageList};

static struct \_s {

char\* name;

struct lconv\* localePtr;

} sArray[] = {{"", &en\_US\_utf8}, {"C", &en\_US\_utf8}, {"US", &en\_US\_utf8}, {"SE", &sw\_EN\_utf8}};

static int sSize = (sizeof sArray) / (sizeof sArray[0]);

static struct \_s\* g\_currStructPtr = &sArray[0];

#define PRINT(x,y) { printf(#x " = <%" #y ">\n", (x)); }

char\* setlocale(int /\*flag\*/, char\* newName) {

int index;

char \*oldName = (g\_currStructPtr != NULL) ? g\_currStructPtr->name : NULL;

g\_currStructPtr = NULL;

if (newName != NULL) {

for (index = 0; index < sSize; ++index) {

if (strcmp(newName, sArray[index].name) == 0) {

g\_currStructPtr = &sArray[index];

break;

}

}

}

return oldName;

}

struct lconv\* localeconv(void) {

return (g\_currStructPtr != NULL) ? g\_currStructPtr->localePtr : NULL;

}

## Character Types

CType.h

#ifndef \_\_CTYPE\_H\_\_

#define \_\_CTYPE\_H\_\_

extern int islower(int c);

extern int isupper(int c);

extern int isalpha(int c);

extern int isdigit(int c);

extern int isalnum(int c);

extern int isxdigit(int c);

extern int isgraph(int c);

extern int isprint(int c);

extern int ispunct(int c);

extern int iscntrl(int c);

extern int isspace(int c);

extern int tolower(int c);

extern int toupper(int c);

#endif

CType.c

#include <CType.h>

#include <StdIO.h>

#include <Locale.h>

#include <String.h>

#include <StdDef.h>

int islower(int c) {

struct lconv\* localeConvPtr = localeconv();

if (localeConvPtr != NULL) {

return (strchr(localeConvPtr->lowerCase, c) != NULL);

}

else {

return ((c >= 'a') && (c <= 'z'));

}

}

int isupper(int c) {

struct lconv\* localeConvPtr = localeconv();

if (localeConvPtr != NULL) {

return (strchr(localeConvPtr->upperCase, c) != NULL);

}

else {

return ((c >= 'A') && (c <= 'Z'));

}

}

int isalpha(int c) {

return islower(c) || isupper(c);

}

int isdigit(int c) {

return (c >= '0') && (c <= '9');

}

int isalnum(int c) {

return isalpha(c) || isdigit(c);

}

int isxdigit(int c) {

return isdigit(c) || ((c >= 'a') && (c <= 'f'))

|| ((c >= 'A') && (c <= 'F'));

}

int isgraph(int c) {

return (c >= 32) && (c <= 126);

}

int isprint(int c) {

return isgraph(c) && (c != ' ');

}

int ispunct(int c) {

return isgraph(c) && !isalnum(c);

}

int iscntrl(int c) {

return !isprint(c);

}

int isspace(int c) {

return (c == ' ') || (c == '\f') || (c == '\n') ||

(c == '\r') || (c == '\t') || (c == '\v');

}

int tolowerX(int c) {

return (isupper(c)) ? (c + 32) : c;

}

int tolower(int c) {

if (isupper(c)) {

struct lconv\* localeConvPtr = localeconv();

if (localeConvPtr != NULL) {

char \*lowerCase = localeConvPtr->lowerCase, \*upperCase = localeConvPtr->upperCase;

int index = (strchr(upperCase, c) - upperCase);

return ((int) lowerCase[index]);

}

else {

return (c + 32);

}

}

else {

return c;

}

}

int toupperX(int c) {

return (islower(c)) ? (c - 32) : c;

}

int toupper(int c) {

if (islower(c)) {

struct lconv\* localeConvPtr = localeconv();

if (localeConvPtr != NULL) {

char \*lowerCase = localeConvPtr->lowerCase,

\*upperCase = localeConvPtr->upperCase;

int index = (strchr(lowerCase, c) - lowerCase);

return ((int) upperCase[index]);

}

else {

return (c - 32);

}

}

else {

return c;

}

}

## Strings

String.h

#ifndef \_\_STRING\_H\_\_

#define \_\_STRING\_H\_\_

#define size\_t int

extern char\* strcpy(char\* target, const char\* source);

extern char\* strncpy(char\* target, const char\* source, size\_t size);

extern char\* strcat(char\* target, const char\* source);

extern char\* strncat(char\* target, const char\* source, size\_t size);

extern int strcmp(const char\* left, const char\* right);

extern int strncmp(const char\* left, const char\* right, size\_t size);

extern char\* strchr(const char\* text, int i);

extern char\* strrchr(const char\* text, int i);

extern size\_t strspn(const char\* mainString, const char\* charSet);

extern size\_t strcspn(const char\* mainString, const char\* charSet);

extern char\* strpbrk(const char\* mainString, const char\* charSet);

extern char\* strstr(const char\* mainString, const char\* subString);

extern size\_t strlen(const char\* string);

extern char\* strerror(int error);

extern char\* strtok(char\* string, const char\* charSet);

extern void\* memcpy(void\* target, const void\* source, size\_t size);

extern void\* memmove(void\* target, const void\* source, size\_t size);

extern int memcmp(const void\* left, const void\* right, size\_t size);

extern void\* memchr(const void\* block, int i, size\_t size);

extern void\* memset(void\* block, int i, size\_t size);

#endif

String.c

#include <ErrNo.h>

#include <StdIO.h>

#include <StdDef.h>

#include <String.h>

#include <Locale.h>

char\* strcpy(char\* target, const char\* source) {

size\_t index;

for (index = 0; source[index] != '\0'; ++index) {

target[index] = source[index];

}

target[index] = '\0';

return target;

}

char\* strncpy(char\* target, const char\* source, size\_t size) {

size\_t index;

for (index = 0; (index < size) && (source[index] != '\0'); ++index) {

target[index] = source[index];

}

for (; index < size; ++index) {

target[index] = '\0';

}

return target;

}

char\* strcat(char\* target, const char\* source) {

size\_t index;

size\_t targetLength = strlen(target);

for (index = 0; source[index] != '\0'; ++index) {

target[targetLength + index] = source[index];

}

target[targetLength + index] = '\0';

return target;

}

char\* strncat(char\* target, const char\* source, size\_t size) {

size\_t index;

size\_t targetLength = strlen(target);

for (index = 0; (index < (size - 1)) && (source[index] != '\0'); ++index) {

target[targetLength + index] = source[index];

}

target[targetLength + size - 1] = '\0';

return target;

}

int strcmp(const char\* left, const char\* right) {

size\_t index;

for (index = 0; TRUE; ++index) {

if ((left[index] == '\0') && (right[index] == '\0')) {

return 0;

}

else if (left[index] == '\0') {

return -1;

}

else if (right[index] == '\0') {

return 1;

}

else if (left[index] < right[index]) {

return -1;

}

else if (left[index] > right[index]) {

return 1;

}

}

}

int strncmp(const char\* left, const char\* right, size\_t size) {

size\_t index;

for (index = 0; index < size; ++index) {

if ((left[index] == '\0') && (right[index] == '\0')) {

return 0;

}

else if (left[index] == '\0') {

return -1;

}

else if (right[index] == '\0') {

return 1;

}

else if (left[index] < right[index]) {

return -1;

}

else if (left[index] > right[index]) {

return 1;

}

}

return 0;

}

char\* strchr(const char\* text, int i) {

size\_t index;

char c = (char) i;

for (index = 0; text[index] != '\0'; ++index) {

if (text[index] == c) {

return &text[index];

}

}

return NULL;

}

char\* strrchr(const char\* text, int i) {

size\_t index;

char\* result = NULL;

char c = (char) i;

for (index = 0; text[index] != '\0'; ++index) {

if (text[index] == c) {

result = &text[index];

}

}

return result;

}

size\_t strspn(const char\* mainString, const char\* charSet) {

size\_t index;

for (index = 0; mainString[index] != '\0'; ++index) {

if (strchr(charSet, mainString[index]) == NULL) {

return index;

}

}

return -1;

}

size\_t strcspn(const char\* mainString, const char\* charSet) {

size\_t index;

for (index = 0; mainString[index] != '\0'; ++index) {

if (strchr(charSet, mainString[index]) != NULL) {

return index;

}

}

return -1;

}

char\* strpbrk(const char\* mainString, const char\* charSet) {

size\_t index;

for (index = 0; mainString[index] != '\0'; ++index) {

if (strchr(charSet, mainString[index]) != NULL) {

return &mainString[index];

}

}

return NULL;

}

char\* strstr(const char\* mainString, const char\* subString) {

size\_t index;

for (index = 0; mainString[index] != '\0'; ++index) {

if (strcmp(mainString + index, subString) == 0) {

return &mainString[index];

}

}

return NULL;

}

size\_t strlen(const char\* string) {

int index;

for (index = 0; string[index] != '\0'; ++index) {

// Empty.

}

return index;

}

extern char\* enMessageList[];

char\* strerror(int errno) {

struct lconv\* localeConvPtr = localeconv();

char\*\* messageList = (localeConvPtr != NULL) ?

localeConvPtr->messageList : NULL;

messageList = (messageList != NULL) ? messageList : enMessageList;

return messageList[errno];

}

char\* token = NULL;

char\* strtok(char\* string, const char\* charSet) {

int index;

char\* tokenStart;

if (string != NULL) {

if (string[0] == '\0') {

return NULL;

}

for (index = 0; string[index] != '\0'; ++index) {

if (strchr(charSet, string[index]) != NULL) {

string[index] = '\0';

token = &string[index + 1];

return string;

}

}

token = &string[index];

return string;

}

else if (token == NULL) {

return NULL;

}

else {

if (token[0] == '\0') {

return NULL;

}

for (index = 0; token[index] != '\0'; ++index) {

if (strchr(charSet, token[index]) != NULL) {

char\* tokenStart2 = token;

token[index] = '\0';

token = &token[index + 1];

return tokenStart2;

}

}

tokenStart = token;

token = &token[index];

return tokenStart;

}

}

void\* memcpy(void\* target, const void\* source, size\_t size) {

char\* charTarget = (char\*) target;

const char\* charSource = (const char\*) source;

size\_t index;

for (index = 0; index < size; ++index) {

charTarget[index] = charSource[index];

}

return (void\*) target;

}

void\* memmove(void\* target, const void\* source, size\_t size) {

char\* charTarget = (char\*) target;

const char\* charSource = (const char\*) source;

size\_t index;

if (source < target) {

for (index = (size - 1); index >= 0; --index) {

charTarget[index] = charSource[index];

}

}

else {

for (index = 0; index < size; ++index) {

charTarget[index] = charSource[index];

}

}

return (void\*) target;

}

int memcmp(const void\* left, const void\* right, size\_t size) {

const char\* charLeft = (const char\*) left;

const char\* charRight = (const char\*) right;

size\_t index;

for (index = 0; index < size; ++index) {

if (charLeft[index] < charRight[index]) {

return -1;

}

else if (charLeft[index] > charRight[index]) {

return 1;

}

}

return 0;

}

void\* memchr(const void\* block, int i, size\_t size) {

size\_t index;

char\* charBlock = (char\*) block;

char c = (char) i;

for (index = 0; index < size; ++index) {

if (charBlock[index] == c) {

return (void\*) &charBlock[index];

}

}

return NULL;

}

void\* memset(void\* block, int i, size\_t size) {

char\* charBlock = (char\*) block;

char c = (char) i;

size\_t index;

for (index = 0; index < size; ++index) {

charBlock[index] = c;

}

return block;

}

## Long Jumps

SetJmp.h

typedef int jmp\_buf[3];

int setjmp(jmp\_buf env);

void longjmp(jmp\_buf env, int value);

SetJmp.c

#include <SetJmp.h>

#include <StdIO.h>

int setjmp(jmp\_buf buf) {

int\* bp\_pointer;

store\_register(bp, bp\_pointer);

buf[0] = bp\_pointer[0]; // return address

buf[1] = bp\_pointer[1]; // normal stack

buf[2] = bp\_pointer[2]; // ellipse stack

return 0;

}

void longjmp(jmp\_buf buf, int return\_value) {

load\_register(bx, return\_value);

load\_register(cx, buf[0]); // return address

load\_register(bp, buf[1]); // normal stack

load\_register(di, buf[2]); // ellipse stack

jump\_register(cx);

}

## Mathematical Functions

Math.h

#ifndef \_\_MATH\_H\_\_

#define \_\_MATH\_H\_\_

#define PI 3.1415926535897932384626433

#define E 2.7182818284590452353602874

#define EPSILON 1e-9

extern double exp(double value);

extern double log(double value);

extern double log10(double value);

extern int log10\_int(double value);

extern double pow(double base, double exponent);

extern double pow\_int(double base, int exponent);

extern double ldexp(double value, int exponent);

extern double frexp(double value, int\* exponent);

extern double sin(double value);

extern double cos(double value);

extern double tan(double value);

extern double sinh(double value);

extern double cosh(double value);

extern double tanh(double value);

extern double sqrt(double value);

extern double asin(double value);

extern double acos(double value);

extern double atan(double value);

extern double atan2(double num, double denum);

extern double floor(double value);

extern double ceil(double value);

extern double round(double value);

extern double fabs(double value);

extern double modf(double value, double\* integralPart);

extern double fmod(double num, double denum);

#endif

Math.c

#include <Math.h>

#include <ErrNo.h>

#include <StdIO.h>

#include <StdDef.h>

#include <StdlIB.h>

double exp(double x) {

double n = 0, faculty = 1, power = 1, term, sum = 0;

do {

term = power / faculty;

sum += term;

power \*= x;

faculty \*= ++n;

} while (fabs(term) >= EPSILON);

return sum;

}

static double logX(double x) { // [1/E,1]

double n = 1, plusMinusOne = 1, x\_minus\_1 = x - 1,

term, sum = 0, power = x\_minus\_1;

do {

term = plusMinusOne \* (power / n++);

sum += term;

power \*= x\_minus\_1;

plusMinusOne \*= -1.0;

} while (fabs(term) > EPSILON);

return sum;

}

double log(double x) {

if (x <= 0) {

errno = EDOM;

return 0;

}

else if (x == 1) {

return 0;

}

else if (x > 1) {

int expo = 0;

while (x > 1) {

x /= E;

++expo;

}

return (logX(x) + expo);

}

else {

int expo = 0;

while (x < 1) {

x \*= E;

++expo;

}

return (logX(x / E) - (expo - 1));

}

}

double log10(double x) {

return 0.4342944820 \* log(x);

}

int log10\_int(double x) {

if (x > 0) {

if (x == 1) {

return 0;

}

else if (x > 1) {

int count = 0;

while (x > 1) {

x /= 10;

++count;

}

return (count - 1);

}

else {

int count = 0;

while (x < 1) {

x \*= 10;

++count;

}

return -count;

}

}

else {

errno = EDOM;

return 0;

}

}

double pow\_int(double x, int y) {

BOOL minus = FALSE;

if (y < 0) {

y = -y;

minus = TRUE;

}

double product = 1;

int index;

for (index = 0; index < y; ++index) {

product \*= x;

}

return minus ? (1 / product) : product;

}

double pow(double x, double y) {

if (x > 0) {

return exp(y \* log(x));

}

else {

errno = EDOM;

exit(-1);

return 0;

}

}

double ldexp(double x, int n) {

return x \* pow(2, (double)n);

}

double frexp(double x, int\* p) {

if (p != NULL) {

if (x == 0) {

\*p = 0;

return 0;

}

else {

\*p = (int) (log(fabs(x)) / log(2)) + 1;

double quotient = fabs(x) / pow(2, \*p);

return (x < 0) ? -quotient : quotient;

}

}

else {

if (x == 0) {

return 0;

}

else {

int n = (int)(log(fabs(x)) / log(2)) + 1;

double a = fabs(x) / pow(2, n);

return (x < 0) ? -a : a;

}

}

}

double sin(double x) {

double n = 0, plusMinusOne = 1, faculty = 1, power = x, term, sum = 0;

do {

term = plusMinusOne \* (power / faculty);

sum += term;

plusMinusOne \*= -1;

power \*= x \* x;

faculty \*= (n + 2) \* (n + 3);

n += 2;

} while (fabs(term) >= EPSILON);

return sum;

}

double cos(double x) {

double n = 0, plusMinusOne = 1, faculty = 1, power = 1, term, sum = 0;

do {

term = plusMinusOne \* (power / faculty);

sum += term;

plusMinusOne \*= -1;

power \*= x \* x;

faculty \*= (n + 1) \* (n + 2);

n += 2;

} while (fabs(term) >= EPSILON);

return sum;

}

double tan(double x) {

double cos\_value = cos(x);

if (cos\_value != 0) {

return sin(x) / cos\_value;

}

else {

errno = EDOM;

return 0;

}

}

double sinh(double x) {

return (exp(x) - exp(-x)) / 2;

}

double cosh(double x) {

return (exp(x) + exp(-x)) / 2;

}

double tanh(double x) {

return sinh(x) / cosh(x);

}

double sqrt(double v) {

if (v >= 0) {

double x\_nplus1 = 1, x;

do {

x = x\_nplus1;

x\_nplus1 = (x + (v / x)) / 2;

} while (fabs(x\_nplus1 - x) >= EPSILON);

return x\_nplus1;

}

else {

errno = EDOM;

return 0;

}

}

double asin(double v) {

if (v == 1) {

return PI / 2;

}

else if (v == -1) {

return -PI / 2;

}

else if (fabs(v) <= 1) {

double x\_nplus1 = 1, x;

do {

x = x\_nplus1;

x\_nplus1 = x - tan(x) + (v / cos(x));

} while (fabs(x\_nplus1 - x) >= EPSILON);

return x\_nplus1;

}

else {

errno = EDOM;

return 0;

}

}

double acos(double v) {

if (v == 1) {

return 0;

}

else if (fabs(v) <= 1) {

double x\_nplus1 = 1, x;

do {

x = x\_nplus1;

x\_nplus1 = x + (1 / tan(x)) - (v / sin(x));

} while (fabs(x\_nplus1 - x) >= EPSILON);

return x\_nplus1;

}

else {

errno = EDOM;

return 0;

}

}

double square(double x) {

return x \* x;

}

double atan(double v) {

if (v == 0) {

return 0;

}

else if (v == 1) {

return PI / 4;

}

else if (v == -1) {

return -PI / 4;

}

else if (fabs(v) < 0.5) {

int sign = 1, denominator = 1; // count = 0;

double product = v, term, sum = 0;

do {

term = sign \* product / denominator;

sum += term;

sign = -sign;

product \*= v \* v;

denominator += 2;

} while (fabs(term) >= EPSILON);

return sum;

}

else if (fabs(v) < 1) {

double x\_nplus1 = 1, x;

do {

x = x\_nplus1;

x\_nplus1 = x - ((tan(x) - v) \* square(cos(2 \* x) + 1)) / 2;

} while (fabs(x\_nplus1 - x) >= EPSILON);

return x\_nplus1;

}

else {

errno = EDOM;

return 0;

}

}

double atanX(double v) {

if (v == 0) {

return 0;

}

else if (fabs(v) <= 1) {

double x\_nplus1 = 1, x;

do {

x = x\_nplus1;

x\_nplus1 = x - ((tan(x) - v) \* square(cos(2 \* x) + 1)) / 2;

} while (fabs(x\_nplus1 - x) >= EPSILON);

return x\_nplus1;

}

else {

errno = EDOM;

return 0;

}

}

double atan2(double num, double denum) {

if (denum > 0) {

return atan(num / denum);

}

else if ((num >= 0) && (denum < 0)) {

return PI + atan(num / denum);

}

else if ((num < 0) && (denum < 0)) {

return (-PI) + atan(num / denum);

}

else if ((num > 0) && (denum == 0)) {

return PI / 2;

}

else if ((num < 0) && (denum == 0)) {

return (-PI) / 2;

}

else {

errno = EDOM;

return 0;

}

}

double floor(double x) {

if (x < 0) {

return -ceil(-x);

}

return (double) (long) x;

}

double ceil(double x) {

if (x < 0) {

return -floor(-x);

}

return (double) (long) (x + 0.999999999999);

}

double round(double x) {

return (double) (long) ((x < 0) ? (x - 0.5) : (x + 0.5));

}

double fabs(double x) {

return (x < 0) ? -x : x;

}

double modf(double x, double\* p) {

double integral = (double) (long) fabs(x);

if (p != NULL) {

\*p = (x > 0) ? (fabs(x) - integral) : -(fabs(x) - integral);

}

return (x > 0) ? integral : -integral;

}

double fmod(double x, double y) {

if (y != 0) {

double quotient = x / y;

double remainder = fabs(quotient - (double) (long) quotient);

return (x > 0) ? remainder : -remainder;

}

else {

errno = EDOM;

return 0;

}

}

## Standard Input/Output

StdIO.h

#ifndef \_\_STDIO\_H\_\_

#define \_\_STDIO\_H\_\_

#include <Math.h>

#include <CType.h>

#include <StdArg.h>

#include <StdDef.h>

#include <File.h>

#include <Temp.h>

#include <Scanf.h>

#include <Printf.h>

#endif

### Printing

Printf.h

#ifndef \_\_PRINTF\_H\_\_

#define \_\_PRINTF\_H\_\_

#define DEVICE 0

#define STRING 1

extern int g\_outStatus, g\_charCount;

extern void\* g\_outDevice;

int putc(int c, FILE\* stream);

int fputc(int c, FILE\* stream);

int putchar(int c);

void printChar(char c);

void printString(char\* s, int precision);

void printIntRec(int intValue);

void printInt(int intValue, BOOL plus, BOOL space);

void printDoublePlain(double doubleValue, BOOL plus, BOOL space,

BOOL grid, int precision);

void printLongDoublePlain(long double doubleValue, BOOL plus,

BOOL space, BOOL grid, int precision);

void printInt(int intValue, BOOL plus, BOOL space);

void printLongInt(long longIntValue, BOOL plus, BOOL space);

void printLongDoubleFraction(long double longDoubleValue,

BOOL grid, int precision);

void printLongDoublePlain(long double longDoubleValue, BOOL plus,

BOOL space, BOOL grid, int precision);

int printFormat(char\* format, va\_list arg\_list);

int printf(char\* format, ...);

int vprintf(char\* format, va\_list arg\_list);

int fprintf(FILE\* outStream, char\* format, ...);

int vfprintf(FILE\* outStream, char\* format, va\_list arg\_list);

int sprintf(char\* outString, char\* format, ...);

int vsprintf(char\* outString, char\* format, va\_list arg\_list);

#endif

Printf.c

#include <Math.h>

#include <CType.h>

#include <StdIO.h>

#include <StdDef.h>

#include <StdArg.h>

#include <StdLib.h>

#include <Scanf.h>

#include <Printf.h>

#define DEVICE 0

#define STRING 1

#define BLANK 2

#define PRINT(x,y) { printf(#x " = <%" #y ">\n", (x)); }

int g\_outStatus, g\_outChars;

void\* g\_outDevice;

#define DEFAULT\_PRECISION 10

int putc(int i, FILE\* stream) {

g\_outStatus = DEVICE;

g\_outDevice = (void\*) stream;

printChar((char) i);

return 1;

}

int fputc(int i, FILE\* stream) {

g\_outStatus = DEVICE;

g\_outDevice = (void\*) stream;

printChar((char) i);

return 1;

}

int putchar(int i) {

g\_outStatus = DEVICE;

g\_outDevice = (void\*) stdout;

printChar((char) i);

return 1;

}

void printChar(char c) {

int handle;

char\* outString;

switch (g\_outStatus) {

case DEVICE: {

FILE\* stream = (FILE\*) g\_outDevice;

handle = stream->handle;

load\_register(ah, 0x40s);

load\_register(bx, handle);

load\_register(cx, 1);

load\_register(dx, &c);

interrupt(0x21s);

++g\_outChars;

break;

}

case STRING: {

outString = (char\*) g\_outDevice;

outString[g\_outChars++] = c;

}

break;

case BLANK:

g\_outChars++;

break;

}

}

void printString(char\* s, int precision) {

if (s != NULL) {

int index;

if (precision == 0) {

for (index = 0; s[index] != '\0'; ++index) {

printChar(s[index]);

}

}

else {

for (index = 0; (precision-- > 0) && (s[index] != '\0'); ++index) {

printChar(s[index]);

}

}

}

else {

printChar('<');

printChar('N');

printChar('U');

printChar('L');

printChar('L');

printChar('>');

}

}

void printLongIntRec(long longValue) {

if (longValue != 0) {

int digit = (int) (longValue % 10L);

printLongIntRec(longValue / 10L);

printChar((char) (digit + '0'));

}

}

void printLongInt(long longValue, BOOL plus, BOOL space) {

if (longValue < 0L) {

longValue = -longValue;

printChar('-');

}

else if (space) {

printChar(' ');

}

else if (plus) {

printChar('+');

}

if (longValue == 0L) {

printChar('0');

}

else {

printLongIntRec(longValue);

}

}

char digitToChar(int digit, BOOL capital) {

if (digit < 10) {

return ((char) ('0' + digit));

}

else if (capital) {

return ((char) ('A' + (digit - 10)));

}

else {

return ((char) ('a' + (digit - 10)));

}

}

void printUnsignedLongRec(unsigned long unsignedValue, unsigned long base, BOOL capital) {

if (unsignedValue > 0ul) {

int digit = (int) (unsignedValue % base);

printUnsignedLongRec(unsignedValue / base, base, capital);

char c = digitToChar(digit, capital);

printChar(c);

}

}

void printUnsignedLong(unsigned long unsignedValue, BOOL plus, BOOL space,

BOOL grid, unsigned long base, BOOL capital) {

if (plus) {

printChar('+');

}

if (space) {

printChar(' ');

}

if (grid) {

if (base == 8ul) {

printChar('0');

}

if (base == 16ul) {

printChar('0');

printChar(capital ? 'X' : 'x');

}

}

if (unsignedValue == 0ul) {

printChar('0');

}

else {

printUnsignedLongRec(unsignedValue, base, capital);

}

}

void printLongDoubleFraction(long double longDoubleValue, BOOL grid, int precision) {

longDoubleValue -= (long) longDoubleValue;

if (precision == 0) {

precision = DEFAULT\_PRECISION;

}

if (grid || (precision > 0)) {

printChar('.');

}

while (precision-- > 0) {

long double longDoubleValue10 = 10.0L \* longDoubleValue;

int digitValue = (int) longDoubleValue10;

char c = (char) (digitValue + '0');

printChar(c);

longDoubleValue = longDoubleValue10 - ((long double) digitValue);

}

}

void printLongDoublePlain(long double longDoubleValue, BOOL plus, BOOL space, BOOL grid, int precision) {

if (longDoubleValue < 0.0L) {

printChar('-');

longDoubleValue = -longDoubleValue;

plus = FALSE;

space = FALSE;

}

long longValue = (long) longDoubleValue;

printLongInt(longValue, plus, space);

longDoubleValue -= (long double) longValue;

printLongDoubleFraction(longDoubleValue, grid, precision);

}

void printLongDoubleExpo(long double value, BOOL plus, BOOL space,

BOOL grid, int precision, BOOL capital) {

if (value == 0.0L) {

printChar('0');

printLongDoubleFraction(0.0L, precision, grid);

printChar(capital ? 'E' : 'e');

printChar('0');

}

else {

if (value < 0.0L) {

printChar('-');

value = -value;

}

int expo = (int) log10(value);

value /= pow(10, expo);

printLongDoublePlain(value, plus, space, grid, precision);

printChar(capital ? 'E' : 'e');

printLongInt(expo, TRUE, FALSE);

}

}

va\_list checkWidthAndPrecision(va\_list arg\_list, int\* widthPtr, int\* precisionPtr) {

if ((widthPtr != NULL) && (\*widthPtr == -1)) {

\*widthPtr = va\_arg(arg\_list, int);

}

if ((precisionPtr != NULL) && (\*precisionPtr == -1)) {

\*precisionPtr = va\_arg(arg\_list, int);

}

return arg\_list;

}

va\_list printArgument(char\* format, va\_list arg\_list, BOOL plus, BOOL space,

BOOL grid, int\* widthPtr, int precision, BOOL shortInt,

BOOL longInt, BOOL longDouble, BOOL sign, BOOL\* negativePtr) {

char c = format[0], charValue;

int /\*logValue, \*/ \*intPtr;

long double longDoubleValue;

void\* ptrValue;

switch (c) {

case 'i':

case 'd': {

long longValue;

if (shortInt) {

longValue = (long) (short) va\_arg(arg\_list, int);

}

else if (longInt) {

longValue = va\_arg(arg\_list, long);

}

else {

longValue = (long) va\_arg(arg\_list, int);

}

if (negativePtr != NULL) {

\*negativePtr = (longValue < 0);

}

if (!sign) {

longValue = labs(longValue);

}

arg\_list = checkWidthAndPrecision(arg\_list, widthPtr, &precision);

printLongInt(longValue, plus, space);

}

break;

case 'c':

charValue = (char) va\_arg(arg\_list, int);

arg\_list = checkWidthAndPrecision(arg\_list, widthPtr, &precision);

printChar(charValue);

break;

case 's': {

char\* stringValue = va\_arg(arg\_list, char\*);

arg\_list = checkWidthAndPrecision(arg\_list, widthPtr, &precision);

printString(stringValue, precision);

}

break;

case 'u':

case 'o':

case 'b':

case 'x':

case 'X': {

unsigned long base = ((c == 'u') ? 10ul : ((c == 'o') ? 8ul : ((c == 'b') ? 2ul : 16ul)));

unsigned long value;

if (shortInt) {

value = (unsigned long) (unsigned short) va\_arg(arg\_list, unsigned int);

}

else if (longInt) {

value = va\_arg(arg\_list, unsigned long);

}

else {

value = (unsigned long) va\_arg(arg\_list, unsigned int);

}

arg\_list = checkWidthAndPrecision(arg\_list, widthPtr, &precision);

printUnsignedLong(value, plus, space, grid, base, isupper(c));

}

break;

case 'f':

case 'e':

case 'E':

case 'g':

case 'G':

if (longDouble) {

longDoubleValue = va\_arg(arg\_list, long double);

}

else {

longDoubleValue = (long double) va\_arg(arg\_list, double);

}

if (negativePtr != NULL) {

\*negativePtr = (longDoubleValue < 0);

}

if (!sign) {

longDoubleValue = fabs(longDoubleValue);

}

arg\_list = checkWidthAndPrecision(arg\_list, widthPtr, &precision);

if (c == 'f') {

printLongDoublePlain(longDoubleValue, plus, space, grid, precision);

}

else if (tolower(c) == 'e') {

printLongDoubleExpo(longDoubleValue, plus, space, grid, precision, isupper(c));

}

else {

int expo = (int) log10(fabs(longDoubleValue));

if ((expo >= -3) && (expo <= 2)) {

printLongDoublePlain(longDoubleValue, plus, space, grid, precision);

}

else {

printLongDoubleExpo(longDoubleValue, plus, space, grid, precision, isupper(c));

}

}

break;

case 'p':

ptrValue = va\_arg(arg\_list, void\*);

arg\_list = checkWidthAndPrecision(arg\_list, widthPtr, &precision);

printUnsignedLong((unsigned long) ptrValue, FALSE, FALSE, FALSE, 10u, FALSE);

break;

case 'n':

ptrValue = va\_arg(arg\_list, void\*);

intPtr = va\_arg(arg\_list, int\*);

arg\_list = checkWidthAndPrecision(arg\_list, widthPtr, &precision);

\*intPtr = g\_outChars;

break;

case '%':

//printf("<X>");

arg\_list = checkWidthAndPrecision(arg\_list, widthPtr, &precision);

printChar('%');

break;

}

return arg\_list;

}

int printFormat(char\* format, va\_list arg\_list) {

int index, count = 0, width = 0, precision = 0;

BOOL percent = FALSE, plus = FALSE, minus = FALSE, space = FALSE,

zero = FALSE, grid = FALSE, widthStar = FALSE,

period = FALSE, precisionStar = FALSE,

shortInt = FALSE, longInt = FALSE, longDouble = FALSE;

g\_outChars = 0;

for (index = 0; format[index] != '\0'; ++index) {

char c = format[index];

if (percent) {

switch (c) {

case '+':

plus = TRUE;

break;

case '-':

minus = TRUE;

break;

case ' ':

space = TRUE;

break;

case '0':

zero = TRUE;

break;

case '#':

grid = TRUE;

break;

case '.':

period = TRUE;

break;

case '\*':

if (!period) {

width = -1;

}

else {

precision = -1;

}

break;

case 'h':

shortInt = TRUE;

break;

case 'l':

longInt = TRUE;

break;

case 'L':

longDouble = TRUE;

break;

case 'i':

case 'd':

case 'u':

case 'b':

case 'o':

case 'x':

case 'X':

case 'c':

case 's':

case 'f':

case 'e':

case 'E':

case 'g':

case 'G':

case 'p':

case 'n':

case '%': {

if (minus) {

int startChars = g\_outChars;

arg\_list = printArgument(&format[index], arg\_list, plus, space, grid, &width,

precision, shortInt, longInt, longDouble, TRUE, NULL);

int field = g\_outChars - startChars; // fill after

while (field++ < width) {

printChar(' ');

}

}

else if (zero) {

int startChars = g\_outChars, oldOutStatus = g\_outStatus;

BOOL negative = FALSE;

g\_outStatus = BLANK;

printArgument(&format[index], arg\_list, FALSE, FALSE, grid, &width,

precision, shortInt, longInt, longDouble, FALSE, &negative);

g\_outStatus = oldOutStatus;

int field = g\_outChars - startChars;

g\_outChars = startChars;

if (negative) {

printChar('-');

++field;

}

else if (plus) {

printChar('+');

++field;

}

else if (space) {

printChar(' ');

++field;

}

while (field++ < width) {

printChar('0');

}

arg\_list = printArgument(&format[index], arg\_list, FALSE, FALSE, grid, NULL,

precision, shortInt, longInt, longDouble, FALSE, NULL);

}

else {

int startChars = g\_outChars, oldOutStatus = g\_outStatus;

g\_outStatus = BLANK;

printArgument(&format[index], arg\_list, plus, space, grid, &width,

precision, shortInt, longInt, longDouble, TRUE, NULL);

g\_outStatus = oldOutStatus;

int field = g\_outChars - startChars;

g\_outChars = startChars;

while (field++ < width) {

printChar(' ');

}

arg\_list = printArgument(&format[index], arg\_list, plus, space, grid, NULL,

precision, shortInt, longInt, longDouble, TRUE, NULL);

}

percent = FALSE;

}

break;

default: {

int value = 0;

while (isdigit(c)) {

value = (10 \* value) + (c - '0');

c = format[++index];

}

--index;

if (!period) {

width = value;

}

else {

precision = value;

}

}

break;

}

}

else {

if (c == '%') {

percent = TRUE;

plus = FALSE;

minus = FALSE;

space = FALSE;

zero = FALSE;

grid = FALSE;

widthStar = FALSE;

period = FALSE;

precisionStar = FALSE;

shortInt = FALSE;

longInt = FALSE;

longDouble = FALSE;

width = 0;

precision = 0;

}

else {

printChar(c);

}

}

}

if (g\_outStatus == STRING) {

char\* outString = (char\*) g\_outDevice;

outString[g\_outChars] = '\0';

}

return g\_outChars;

}

int printf(char\* format, ...) {

va\_list arg\_list;

va\_start(arg\_list, format);

return vprintf(format, arg\_list);

}

int vprintf(char\* format, va\_list arg\_list) {

return vfprintf(stdout, format, arg\_list);

}

int fprintf(FILE\* outStream, char\* format, ...) {

va\_list arg\_list;

va\_start(arg\_list, format);

return vfprintf(outStream, format, arg\_list);

}

int vfprintf(FILE\* outStream, char\* format, va\_list arg\_list) {

g\_outStatus = DEVICE;

g\_outDevice = (void\*) outStream;

return printFormat(format, arg\_list);

}

int sprintf(char\* outString, char\* format, ...) {

va\_list arg\_list;

va\_start(arg\_list, format);

return vsprintf(outString, format, arg\_list);

}

int vsprintf(char\* outString, char\* format, va\_list arg\_list) {

g\_outStatus = STRING;

g\_outDevice = (void\*) outString;

return printFormat(format, arg\_list);

}

### Scanning

Scanf.h

#ifndef \_\_SCANF\_H\_\_

#define \_\_SCANF\_H\_\_

#define DEVICE 0

#define STRING 1

#define EOF -1

extern int g\_inStatus, g\_inChars;

extern void\* g\_inDevice;

char scanChar(void);

void unscanChar(char c);

void scanString(char\* string, int precision);

long scanLongInt(void);

unsigned long scanUnsignedLongInt(unsigned long base);

long double scanLongDouble(void);

int scanf(char\* format, ...);

int vscanf(char\* format, va\_list arg\_list);

int fscanf(FILE\* inStream, char\* format, ...);

int vfscanf(FILE\* inStream, char\* format, va\_list arg\_list);

int sscanf(char\* inString, char\* format, ...);

int vsscanf(char\* inString, char\* format, va\_list arg\_list);

#endif

Scanf.c

#include <Math.h>

#include <CType.h>

#include <StdIO.h>

#include <StdDef.h>

#include <StdArg.h>

#include <String.h>

#include <Scanf.h>

#include <Printf.h>

#define PRINT(x,y) { printf(#x " = <%" #y ">\n", (x)); }

int g\_inStatus, g\_inChars;

void\* g\_inDevice;

int g\_inCount;

char scanChar(void) {

char c = '\0';

FILE\* stream;

int handle;

char\* inString;

switch (g\_inStatus) {

case DEVICE:

stream = (FILE\*) g\_inDevice;

if ((g\_inDevice != stdin) && feof(stream)) {

return ((char) EOF);

}

else {

handle = stream->handle;

load\_register(ah, 0x3Fs);

load\_register(bx, handle);

load\_register(cx, 1);

load\_register(dx, &c);

interrupt(0x21s);

++g\_inChars;

return c;

}

case STRING:

inString = (char\*) g\_inDevice;

return inString[g\_inChars++];

default:

return '\0';

}

}

void unscanChar(char /\* c \*/) {

switch (g\_inStatus) {

case DEVICE:

/\* stream = (FILE\*) g\_inDevice;

handle = stream->handle;

load\_register(ah, 0x3F);

load\_register(bx, handle);

load\_register(cx, 1);

load\_register(dx, &c);

interrupt(0x21s);

\*/

--g\_inChars;

break;

case STRING:

--g\_inChars;

break;

}

}

void scanPattern(char\* string, char\* pattern, BOOL not) {

int index = 0;

char input = scanChar();

while (isspace(input)) {

input = scanChar();

}

if (string != NULL) {

while ((!not && strchr(pattern, input)) ||

(not && !strchr(pattern, input))) {

string[index++] = input;

input = scanChar();

}

string[index] = '\0';

}

else {

while ((!not && strchr(pattern, input)) ||

(not && !strchr(pattern, input))) {

input = scanChar();

}

}

}

void scanString(char\* string, int precision) {

int index = 0;

char input = scanChar();

BOOL found = FALSE;

while (isspace(input)) {

input = scanChar();

}

if (string != NULL) {

if (precision == 0) {

while (!isspace(input) && (input != EOF)) {

string[index++] = input;

input = scanChar();

found = TRUE;

++g\_inChars;

}

string[index] = '\0';

++g\_inChars;

}

else {

while ((precision-- > 0) && (!isspace(input) && (input != EOF))) {

string[index++] = input;

input = scanChar();

found = TRUE;

++g\_inChars;

}

if (precision > 0) {

string[index] = '\0';

++g\_inChars;

}

}

}

else {

if (precision == 0) {

while (!isspace(input) && (input != EOF)) {

input = scanChar();

found = TRUE;

++g\_inChars;

}

++g\_inChars;

}

else {

while ((precision-- > 0) && (!isspace(input) && (input != EOF))) {

input = scanChar();

found = TRUE;

++g\_inChars;

}

if (precision > 0) {

++g\_inChars;

}

}

}

if (found) {

++g\_inCount;

}

}

unsigned long digitToValue(char input) {

if (isdigit(input)) {

return (input - '0');

}

else if (islower(input)) {

return ((input - 'a') + 10ul);

}

else {

return ((input - 'A') + 10ul);

}

}

long scanLongInt(void) {

long longValue = 0l;

BOOL minus = FALSE, found = FALSE;

char input = scanChar();

while (isspace(input)) {

input = scanChar();

}

if (input == '+') {

input = scanChar();

}

else if (input == '-') {

minus = TRUE;

input = scanChar();

}

while (isdigit(input)) {

longValue = (10l \* longValue) + (input - '0');

input = scanChar();

found = TRUE;

}

if (minus) {

longValue = -longValue;

}

if (found) {

++g\_inCount;

}

unscanChar(input);

return longValue;

}

unsigned long scanUnsignedLongInt(unsigned long base) {

unsigned long unsignedLongValue = 0ul, digit;

char input = scanChar();

BOOL found = TRUE;

while (isspace(input)) {

input = scanChar();

}

if (input == '0') {

input = scanChar();

if (tolower(input) == 'x') {

base = (base == 0ul) ? 16ul : base;

input = scanChar();

}

else {

base = (base == 0ul) ? 8ul : base;

}

}

if (base == 0ul) {

base = 10ul;

}

while (isxdigit(input)) {

digit = digitToValue(input);

if (digit >= base) {

break;

}

unsignedLongValue = (unsignedLongValue \* base) + digit;

found = TRUE;

input = scanChar();

}

if (found) {

++g\_inCount;

}

unscanChar(input);

return unsignedLongValue;

}

long double scanLongDouble(void) {

BOOL minus = FALSE, found = FALSE;

long double value = 0.0L, factor = 1.0L;

char input = scanChar();

while (isspace(input)) {

input = scanChar();

}

if (input == '+') {

input = scanChar();

}

else if (input == '-') {

minus = TRUE;

input = scanChar();

}

while (isdigit(input)) {

value = (10.0L \* value) + ((long double) (input - '0'));

input = scanChar();

found = TRUE;

}

if (input == '.') {

input = scanChar();

while (isdigit(input)) {

factor /= 10.0L;

value += factor \* ((long double) (input - '0'));

input = scanChar();

found = TRUE;

}

}

if (tolower(input) == 'e') {

double exponent = (double) scanLongInt();

value \*= pow(10.0, exponent);

}

else {

unscanChar(input);

}

if (minus) {

value = -value;

}

if (found) {

++g\_inCount;

}

return value;

}

int scanFormat(char\* format, va\_list arg\_list) {

char c, \*charPtr;

BOOL percent = FALSE, shortInt = FALSE, longIntOrDouble = FALSE, longDouble = FALSE, star = FALSE;

long longValue, \*longPtr;

short\* shortPtr;

int index, \*intPtr, \*charsPtr;

unsigned long unsignedLongValue, \*unsignedLongPtr;

unsigned short\* unsignedShortPtr;

unsigned int\* unsignedIntPtr;

long double longDoubleValue;

g\_inCount = 0;

g\_inChars = 0;

for (index = 0; format[index] != '\0'; ++index) {

c = format[index];

if (percent) {

switch (c) {

case 'h':

shortInt = TRUE;

break;

case 'l':

longIntOrDouble = TRUE;

break;

case 'L':

longDouble = TRUE;

break;

case '\*':

star = TRUE;

break;

case 'c': {

char charValue = scanChar();

if (!star) {

charPtr = va\_arg(arg\_list, char\*);

\*charPtr = charValue;

}

percent = FALSE;

if (charValue != EOF) {

++g\_inCount;

}

}

break;

case 's':

if (!star) {

charPtr = va\_arg(arg\_list, char\*);

scanString(charPtr, 0);

}

else {

scanString(NULL, 0);

}

percent = FALSE;

break;

case 'i':

case 'd':

longValue = scanLongInt();

if (!star) {

if (shortInt) {

shortPtr = va\_arg(arg\_list, short\*);

\*shortPtr = (short) longValue;

}

else if (!longIntOrDouble) {

intPtr = va\_arg(arg\_list, int\*);

\*intPtr = (int) longValue;

}

else {

longPtr = va\_arg(arg\_list, long\*);

\*longPtr = longValue;

}

}

percent = FALSE;

break;

case 'o':

unsignedLongValue = scanUnsignedLongInt(8ul);

if (!star) {

if (shortInt) {

unsignedShortPtr = va\_arg(arg\_list, unsigned short\*);

\*unsignedShortPtr = (short) unsignedLongValue;

}

else if (!longIntOrDouble) {

unsignedIntPtr = va\_arg(arg\_list, unsigned int\*);

\*unsignedIntPtr = (int) unsignedLongValue;

}

else {

unsignedLongPtr = va\_arg(arg\_list, unsigned long\*);

\*unsignedLongPtr = unsignedLongValue;

}

}

percent = FALSE;

break;

case 'x':

unsignedLongValue = scanUnsignedLongInt(16ul);

if (!star) {

if (shortInt) {

unsignedShortPtr = va\_arg(arg\_list, unsigned short\*);

\*unsignedShortPtr = (short) unsignedLongValue;

}

else if (!longIntOrDouble) {

unsignedIntPtr = va\_arg(arg\_list, unsigned int\*);

\*unsignedIntPtr = (int) unsignedLongValue;

}

else {

unsignedLongPtr = va\_arg(arg\_list, unsigned long\*);

\*unsignedLongPtr = unsignedLongValue;

}

}

percent = FALSE;

break;

case 'u':

unsignedLongValue = scanUnsignedLongInt(0ul);

if (!star) {

if (shortInt) {

unsignedShortPtr = va\_arg(arg\_list, unsigned short\*);

\*unsignedShortPtr = (short) unsignedLongValue;

}

else if (!longIntOrDouble) {

unsignedIntPtr = va\_arg(arg\_list, unsigned int\*);

\*unsignedIntPtr = (int) unsignedLongValue;

}

else {

unsignedLongPtr = va\_arg(arg\_list, unsigned long\*);

\*unsignedLongPtr = unsignedLongValue;

}

}

percent = FALSE;

break;

case 'e':

case 'f':

case 'g':

longDoubleValue = scanLongDouble();

if (!star) {

if (longIntOrDouble) {

double\* doublePtr = va\_arg(arg\_list, double\*);

\*doublePtr = (double) longDoubleValue;

}

else if (longDouble) {

long double\* longDoublePtr = va\_arg(arg\_list, long double\*);

\*longDoublePtr = longDoubleValue;

}

else {

float\* floatPtr = va\_arg(arg\_list, float\*);

\*floatPtr = (float) longDoubleValue;

}

}

percent = FALSE;

break;

case '[': {

BOOL not = FALSE;

++index;

if (format[index] == '^') {

not = TRUE;

++index;

}

int startIndex = index;

while (format[index] != ']') {

++index;

}

format[index] = '\0';

if (!star) {

char\* string = va\_arg(arg\_list, char\*);

scanPattern(string, &format[startIndex], not);

}

else {

scanPattern(NULL, &format[startIndex], not);

}

}

break;

case 'n':

charsPtr = va\_arg(arg\_list, int\*);

\*charsPtr = g\_inChars;

percent = FALSE;

break;

}

}

else {

if (c == '%') {

percent = TRUE;

shortInt = FALSE;

longIntOrDouble = FALSE;

longDouble = FALSE;

star = FALSE;

}

}

}

return g\_inCount;

}

int scanf(char\* format, ...) {

va\_list arg\_list;

va\_start(arg\_list, format);

return vscanf(format, arg\_list);

}

int vscanf(char\* format, va\_list arg\_list) {

return vfscanf(stdin, format, arg\_list);

}

int fscanf(FILE\* inStream, char\* format, ...) {

va\_list arg\_list;

va\_start(arg\_list, format);

return vfscanf(inStream, format, arg\_list);

}

int vfscanf(FILE\* inStream, char\* format, va\_list arg\_list) {

g\_inStatus = DEVICE;

g\_inDevice = (void\*) inStream;

return scanFormat(format, arg\_list);

}

int sscanf(char\* inString, char\* format, ...) {

va\_list arg\_list;

va\_start(arg\_list, format);

return vsscanf(inString, format, arg\_list);

}

int vsscanf(char\* inString, char\* format, va\_list arg\_list) {

g\_inStatus = STRING;

g\_inDevice = (void\*) inString;

return scanFormat(format, arg\_list);

}

### File Handling

File.h

#ifndef \_\_FILE\_H\_\_

#define \_\_FILE\_H\_\_

#define FOPEN\_MAX 16

#define FILENAME\_MAX 16

#define fpos\_t int

#define CARRY\_FLAG 0x01u

#define EOF -1

typedef unsigned int UINTX;

typedef struct {

BOOL open;

unsigned int handle;

char name[FILENAME\_MAX], ungetc;

int errno;

unsigned long position, size;

BOOL temporary;

} FILE;

extern FILE\* stdin;

extern FILE\* stdout;

extern FILE\* stderr;

extern enum {EEXIST, ENOENT, EACCES};

extern enum {SEEK\_SET, SEEK\_CUR, SEEK\_END};

#define getc(stream) fgetc(stream)

BOOL fileexists(const char\* name);

FILE\* fopen(const char\* filename, const char\* mode);

FILE\* freopen(const char\* filename, const char\* mode, FILE\* stream);

int fflush(FILE\* stream);

int fclose(FILE\* stream);

int remove(const char\* name);

int rename(const char\* oldName, const char\* newName);

int setvbuf(FILE\* stream, char\* buffer, int mode, size\_t size);

void setbuf(FILE\* stream, char\* buffer);

int fgetc(FILE\* stream);

char\* fgets(char\* s, int n, FILE\* stream);

int fputc(int i, FILE\* stream);

int fputs(const char\* s, FILE\* stream);

int getchar(void);

char\* gets(char\* s);

int putchar(int c);

int puts(const char\* s);

int ungetc(int c, FILE\* stream);

size\_t fread(void\* ptr, size\_t size, size\_t nobj, FILE\* stream);

size\_t fwrite(const void\* ptr, size\_t size, size\_t nobj, FILE\* stream);

long fseek(FILE\* stream, long offset, int origin);

long ftell(FILE\* stream);

void rewind(FILE\* stream);

int fgetpos(FILE\* stream, fpos\_t\* ptr);

int fsetpos(FILE\* stream, const fpos\_t\* ptr);

void clearerr(FILE\* stream);

BOOL feof(FILE\* stream);

int ferror(FILE\* stream);

void perror(const char\* s);

#endif

File.c

#include <StdIO.h>

#include <ErrNo.h>

#include <Locale.h>

#include <String.h>

FILE g\_fileArray[FOPEN\_MAX] = {{TRUE, 0}, {TRUE, 1}, {TRUE, 2}};

FILE\* stdin = &g\_fileArray[0];

FILE\* stdout = &g\_fileArray[1];

FILE\* stderr = &g\_fileArray[2];

static enum { EEXIST, ENOENT, EACCES };

static enum { SEEK\_SET = 0, SEEK\_CUR = 1, SEEK\_END = 2 };

static enum {READ = 0x40, WRITE = 0x41, READ\_WRITE = 0x42};

#define MAX(a,b) (((a) > (b)) ? (a) : (b))

#define FILE\_NOT\_FOUND 0x02

#define PRINT(x,y) { printf(#x " = <%" #y ">\n", (x)); }

long filesize(int handle) {

load\_register(ah, 0x42s);

load\_register(bx, handle);

load\_register(cx, 0);

load\_register(dx, 0);

interrupt(0x21s);

int high, low;

store\_register(dx, high);

store\_register(ax, low);

clear\_registers();

return (((long) high) << 4) + ((long) low);

}

static int filecreate(const char\* name) {

load\_register(ah, 0x3Cs);

load\_register(cx, 0x00);

load\_register(dx, name);

interrupt(0x21s);

int handle;

store\_register(ax, handle);

return handle;

}

BOOL fileexists(const char\* name) {

load\_register(ah, 0x43s);

load\_register(al, 0x00s);

load\_register(dx, name);

interrupt(0x21s);

{ int code;

store\_register(ax, code);

clear\_registers();

return (code != FILE\_NOT\_FOUND);

}

}

static int fileopen(const char\* name, unsigned short mode) {

load\_register(ah, 0x3Ds);

load\_register(al, mode);

load\_register(dx, name);

interrupt(0x21s);

int handle;

store\_register(ax, handle);

if (handle > 2) {

return handle;

}

else {

errno = handle;

return -1;

}

}

FILE\* fopen(const char\* filename, const char\* mode) {

int index;

for (index = 0; index < FOPEN\_MAX; ++index) {

if (!g\_fileArray[index].open) {

return freopen(filename, mode, &g\_fileArray[index]);

}

}

return NULL;

}

FILE\* freopen(const char\* filename, const char\* mode, FILE\* stream) {

int handle = -1;

if (strcmp(mode, "r") == 0) {

handle = fileopen(filename, (unsigned short) READ);

}

else if (strcmp(mode, "w") == 0) {

handle = filecreate(filename);

}

else if (strcmp(mode, "a") == 0) {

handle = fileopen(filename, (unsigned short) WRITE);

if (handle != -1) {

fseek(stream, 0L, (int) SEEK\_END);

}

else {

handle = filecreate(filename);

}

}

else if (strcmp(mode, "r+") == 0) {

handle = fileopen(filename, (unsigned short) READ\_WRITE);

}

else if (strcmp(mode, "w+") == 0) {

handle = filecreate(filename);

}

else if (strcmp(mode, "a+") == 0) {

handle = fileopen(filename, (unsigned short) READ\_WRITE);

if (handle != -1) {

fseek(stream, 0L, (int) SEEK\_END);

}

else {

handle = filecreate(filename);

}

}

if (handle != -1) {

stream->open = TRUE;

stream->handle = handle;

stream->size = 0l; // filesize(handle);

strcpy(stream->name, filename);

stream->temporary = FALSE;

return stream;

}

else {

stream->open = FALSE;

return NULL;

}

}

int fflush(FILE\* stream) {

if (stream == NULL) {

int index;

for (index = 0; index < FOPEN\_MAX; ++index) {

if (g\_fileArray[index].open) {

if (fflush(&g\_fileArray[index]) == EOF) {

return EOF;

}

}

}

}

return 0;

}

int fclose(FILE\* stream) {

if (stream != NULL) {

int handle = stream->handle;

load\_register(ah, 0x3Es);

load\_register(bx, handle);

interrupt(0x21s);

{ int handle;

short flagbyte;

store\_register(ax, handle);

store\_flagbyte(flagbyte);

if (!(flagbyte & CARRY\_FLAG)) {

return 0;

}

else {

errno = handle;

return -1;

}

}

}

else {

int index;

for (index = 0; index < FOPEN\_MAX; ++index) {

if (g\_fileArray[index].open) {

if (fclose(&g\_fileArray[index]) == EOF) {

return EOF;

}

}

}

return 0;

}

}

int remove(const char\* name) {

load\_register(ah, 0x41s);

load\_register(dx, name);

load\_register(cl, 0s);

interrupt(0x21s);

{/\* short flagbyte;

store\_flagbyte(flagbyte);

if (flagbyte & CARRY\_FLAG) {

errno = FREMOVE;

return EOF;

}

\*/

return 0;

}

}

int rename(const char\* oldName, const char\* newName) {

load\_register(ah, 0x56s);

load\_register(cl, 0s);

load\_register(dx, oldName);

load\_register(di, newName);

interrupt(0x21s);

{ /\*short flagbyte;

store\_flagbyte(flagbyte);

if (flagbyte & CARRY\_FLAG) {

errno = errno = FRENAME;

return EOF;

}\*/

return 0;

}

}

int setvbuf(FILE\* /\* stream \*/, char\* /\* buffer \*/, int /\* mode \*/, size\_t /\* size \*/) {

return 0;

/\*

if (buffer != NULL) {

buffer = malloc(size);

if (buffer == NULL) {

return -1;

}

}

stream->buffer = buffer;

stream->bufferMode = mode;

stream->bufferMaxSize = size;

return 0;\*/

}

void setbuf(FILE\* /\* stream \*/, char\* /\* buffer \*/) {

/\*

if (buffer == NULL) {

free(stream->buffer);

stream->buffer = NULL;

stream->bufferMode = \_IONBF;

}

else {

setvbuf(stream, buffer, \_IOFBF, BUFSIZ);

}\*/

}

int fgetc(FILE\* stream) {

char c = '\0';

if (fread(&c, sizeof (char), 1, stream) > 0) {

return (int) c;

}

return EOF;

}

char\* fgets(char\* text, int size, FILE\* stream) {

int count = 0;

char prevChar = '\0';

while ((count < (size - 1))) {

char currChar = '\0';

fscanf(stream, "%c", &currChar);

if ((prevChar == '\r') && (currChar == '\n')) {

text[count] = '\0';

break;

}

if ((prevChar != '\r') && (currChar == '\n')) {

text[count] = '\0';

break;

}

if (currChar == EOF) {

text[count] = '\0';

break;

}

if ((currChar != '\r') && (currChar != '\n')) {

text[count++] = currChar;

}

prevChar = currChar;

}

return text;

}

int fputs(const char\* s, FILE\* stream) {

int size = strlen(s) \* sizeof (char);

return (fwrite(s, size, 1, stream) == size) ? 0 : EOF;

}

int getchar(void) {

return fgetc(stdin);

}

char\* gets(char\* s) {

if (fgets(s, -1, stdin) != NULL) {

int size = strlen(s);

if (size > 0) {

s[size - 1] = '\0';

}

return s;

}

else {

return NULL;

}

}

int puts(const char\* s) {

if (fputs(s, stdout) != 0) {

return fputc('\n', stdout);

}

return EOF;

}

int ungetc(int c, FILE\* stream) {

if (stream->ungetc != EOF) {

stream->ungetc = (char) c;

}

return c;

}

size\_t fread(void\* ptr, size\_t size, size\_t nobj, FILE\* stream) {

/\* if (((size \* nobj) > 0) && (stream->ungetc != EOF)) {

char c = stream->ungetc;

memcpy(ptr, &c, 1);

stream->ungetc = EOF;

return fread(((char\*) ptr) + 1, (size \* nobj) - 1, 1, stream);

}\*/

int handle = stream->handle, total = size \* nobj;

load\_register(ah, 0x3Fs);

load\_register(bx, handle);

load\_register(cx, total);

load\_register(dx, ptr);

interrupt(0x21s);

{ short flagbyte;

store\_flagbyte(flagbyte);

if (flagbyte & CARRY\_FLAG) {

stream->errno = errno = FREAD;

return 0;

}

else {

int count;

store\_register(ax, count);

// stream->position += count;

return count;

}

}

}

size\_t fwrite(const void\* ptr, size\_t size, size\_t nobj, FILE\* stream) {

int handle = stream->handle, total = size \* nobj;

load\_register(ah, 0x40s);

load\_register(bx, handle);

load\_register(cx, total);

load\_register(dx, ptr);

interrupt(0x21s);

{ short flagbyte;

store\_flagbyte(flagbyte);

if (flagbyte & CARRY\_FLAG) {

stream->errno = errno = FWRITE;

return 0;

}

else {

int count;

store\_register(ax, count);

// stream->position += count;

// stream->size = MAX(stream->size, stream->position);

return count;

}

}

}

long fseek(FILE\* stream, long offset, int origin) {

load\_register(al, (short) origin);

load\_register(ah, 0x42s);

load\_register(bx, stream->handle);

load\_register(cx, (int) (offset >> 16));

load\_register(dx, (int) offset);

interrupt(0x21s);

{ int highPosition, lowPositionHandle;

store\_register(dx, highPosition);

store\_register(ax, lowPositionHandle);

short flagbyte;

clear\_registers();

store\_flagbyte(flagbyte);

if (!(flagbyte & CARRY\_FLAG)) {

stream->position = (((unsigned long) highPosition) << 16) +

((unsigned long) lowPositionHandle);

return stream->position;

}

else {

stream->errno = (errno = lowPositionHandle);

return -1l;

}

}

}

long ftell(FILE\* stream) {

return fseek(stream, 0L, (int) SEEK\_CUR);

}

void rewind(FILE\* stream) {

(void) fseek(stream, 0L, (int) SEEK\_SET);

}

int fgetpos(FILE\* stream, fpos\_t\* ptr) {

\*ptr = (fpos\_t) ftell(stream);

return 0;

}

int fsetpos(FILE\* stream, const fpos\_t\* ptr) {

return ((int) fseek(stream, \*ptr, (int) SEEK\_SET));

}

void clearerr(FILE\* stream) {

stream->errno = errno = 0;

}

BOOL feof(FILE\* stream) {

long unsigned currPosition = fseek(stream, 0L, (int) SEEK\_CUR);

long unsigned lastPosition = fseek(stream, 0L, (int)SEEK\_END);

fseek(stream, currPosition, (int) SEEK\_SET);

{ BOOL endOfFile = (currPosition == lastPosition);

return endOfFile;

}

}

int ferror(FILE\* stream) {

return stream->errno;

}

void perror(const char\* s) {

printf("%s: %s.\n", s, strerror(errno));

}

## The Standard Library

StdLib.h

#ifndef \_\_STDLIB\_H\_\_

#define \_\_STDLIB\_H\_\_

#define NULL ((void\*) 0)

double atof(char\* s);

int atoi(char\* s);

long atol(char\* s);

double strtod(char\* s, char\*\* endp);

long strtol(char\* s, char\*\* endp, int base);

unsigned long strtoul(char\* s, char\*\* endp, int base);

int rand(void);

void srand(unsigned int seed);

char\* getenv(const char\* name);

int system(const char\* command);

void abort(void);

void exit(int status);

#define FUNC\_MAX 256

#define OPEN\_MAX 16

typedef void(\*FCN)(void);

int atexit(FCN fcn);

int abs(int value);

long labs(long value);

void\* malloc(size\_t size);

void\* realloc(void\* ptr, size\_t newSize);

void\* calloc(size\_t num, size\_t size);

void free(void\* ptr);

void qsort(const void\* valueList, size\_t listSize, size\_t valueSize,

int (\*compare)(const void\*, const void\*), ...);

void\* bsearch(const void\* key, const void\* valueList, size\_t listSize,

size\_t valueSize, int (\*compare)(const void\*, const void\*));

int abs(int value);

long labs(long value);

typedef struct {

int quot, rem;

} div\_t;

div\_t div(int num, int denum);

typedef struct {

long quot, rem;

} ldiv\_t;

ldiv\_t ldiv(long num, long denum);

#endif

StdLib.c

#include <Math.h>

#include <CType.h>

#include <ErrNo.h>

#include <StdArg.h>

#include <StdDef.h>

#include <String.h>

#include <StdLib.h>

#include <StdIO.h>

extern FILE g\_fileArray[];

int atoi(char\* s) {

return (int) strtol(s, (char\*\*) NULL, 10);

}

long atol(char\* s) {

return strtol(s, (char\*\*) NULL, 10);

}

long strtol(char\* s, char\*\* endp, int /\* base \*/) {

int chars = 0;

long value = 0;

sscanf(s, "%li%n", &value, &chars);

if (endp != NULL) {

\*endp = s + chars;

}

return value;

}

unsigned long strtoul(char\* s, char\*\* endp, int /\* base \*/) {

int chars = 0;

unsigned long value = 0;

sscanf(s, "%lu%n", &value, &chars);

if (endp != NULL) {

\*endp = s + chars;

}

return value;

}

double atof(char\* s) {

return strtod(s, (char\*\*) NULL);

}

double strtod(char\* s, char\*\* endp) {

int chars = '\0';

double value = 0;

sscanf(s, "%lf%n", &value, &chars);

if (endp != NULL) {

\*endp = s + chars;

}

return value;

}

void abort(void) {

load\_register(ah, 0x4Cs);

load\_register(al, -1s);

interrupt(0x21s);

}

char\* getenv(char\* /\* name \*/) {

return NULL;

}

int system(const char\* /\* command \*/) {

return -1;

}

void memswp(void\* value1, void\* value2, int valueSize) {

char\* charValue1 = (char\*) value1;

char\* charValue2 = (char\*) value2;

int index;

for (index = 0; index < valueSize; ++index) {

char tempValue = charValue1[index];

charValue1[index] = charValue2[index];

charValue2[index] = tempValue;

}

}

void\* bsearch(const void\* keyPtr, const void\* valueList, size\_t listSize,

size\_t valueSize, int (\*compare)(const void\*, const void\*)) {

int firstIndex = 0, lastIndex = listSize - 1;

if (listSize == 0) {

return NULL;

}

while (TRUE) {

{ char\* firstValuePtr = ((char\*) valueList) + (firstIndex \* valueSize);

int firstCompare = compare(keyPtr, firstValuePtr);

if (firstCompare < 0) {

return NULL;

}

else if (firstCompare == 0) {

return firstValuePtr;

}

}

{ char\* lastValuePtr = ((char\*) valueList) + (lastIndex \* valueSize);

int lastCompare = compare(keyPtr, lastValuePtr);

if (lastCompare > 0) {

return NULL;

}

else if (lastCompare == 0) {

return lastValuePtr;

}

}

{ int middleIndex = (firstIndex + lastIndex) / 2;

char\* middleValuePtr = ((char\*) valueList) + (middleIndex \* valueSize);

int middleCompare = compare(keyPtr, middleValuePtr);

if (middleCompare < 0) {

lastIndex = middleIndex;

}

else if (middleCompare > 0) {

firstIndex = middleIndex;

}

else {

return middleValuePtr;

}

}

}

}

static long g\_randValue;

#define A 1664525l

#define C 1013904223l

#define RAND\_MAX 127 // 32767

int rand(void) {

g\_randValue = ((A \* g\_randValue) + C) % RAND\_MAX;

return (int) g\_randValue;

}

void srand(unsigned int seed) {

g\_randValue = (long) seed;

}

FCN g\_funcArray[FUNC\_MAX] = {NULL};

int atexit(FCN fcn) {

int index;

for (index = 0; index < FUNC\_MAX; ++index) {

if (g\_funcArray[index] == NULL) {

g\_funcArray[index] = fcn;

return 0;

}

}

return -1;

}

void exit(int status) {

int index;

for (index = 0; index < OPEN\_MAX; ++index) {

if (g\_fileArray[index].open) {

fclose(&g\_fileArray[index]);

}

}

for (index = (FUNC\_MAX - 1); index >= 0; --index) {

if (g\_funcArray[index] != NULL) {

g\_funcArray[index]();

}

}

load\_register(ah, 0x4Cs);

load\_register(al, (short) status);

interrupt(0x21s);

}

void swap(char\* leftValuePtr, char\* rightValuePtr, int valueSize) {

int index;

for (index = 0; index < valueSize; ++index) {

char tempValue = leftValuePtr[index];

leftValuePtr[index] = rightValuePtr[index];

rightValuePtr[index] = tempValue;

}

}

void qsort(const void\* valueList, size\_t listSize, size\_t valueSize,

int (\*compare)(const void\*, const void\*)) {

BOOL update;

char\* charList = (char\*) valueList;

int index1;

for (index1 = (listSize - 1); index1 > 0 ; --index1) {

int index2;

update = FALSE;

for (index2 = 0; index2 < index1; ++index2) {

char\* valuePtr1 = charList + (index2 \* valueSize);

char\* valuePtr2 = charList + ((index2 + 1) \* valueSize);

if (compare(valuePtr1, valuePtr2) > 0) {

swap(valuePtr1, valuePtr2, valueSize);

update = TRUE;

}

}

if (!update) {

break;

}

}

}

int abs(int value) {

return (value < 0) ? -value : value;

}

long labs(long value) {

return (value < 0l) ? -value : value;

}

div\_t div(int num, int denum) {

div\_t result = {0, 0};

if (denum == 0) {

errno = EDOM;

return result;

}

result.quot = num / denum;

result.rem = num % denum;

return result;

}

ldiv\_t ldiv(long num, long denum) {

ldiv\_t result = {0, 0};

if (denum == 0) {

errno = EDOM;

return result;

}

result.quot = num / denum;

result.rem = num % denum;

return result;

}

## Time

Time.h

#ifndef \_\_TIME\_H\_\_

#define \_\_TIME\_H\_\_

#define size\_t int

#define time\_t unsigned long

#define clock\_t long

struct tm {

int tm\_sec;

int tm\_min;

int tm\_hour;

int tm\_mday;

int tm\_mon;

int tm\_year;

int tm\_wday;

int tm\_yday;

int tm\_isdst;

};

extern clock\_t clock(void);

extern time\_t time(time\_t\* time);

extern double difftime(time\_t time2, time\_t time1);

extern time\_t mktime(struct tm\* timeStruct);

extern char\* asctime(const struct tm\* timeStruct);

extern char\* ctime(const time\_t\* time);

extern struct tm\* gmtime(const time\_t\* time);

extern struct tm\* localtime(const time\_t\* time);

extern struct tm\* localtimeX(const time\_t\* time);

extern size\_t strftime(char\* buffer, size\_t size,

const char\* format, const struct tm\* timeStruct);

#endif

Time.c

#include <Time.h>

#include <StdIO.h>

#include <StdLib.h>

#include <String.h>

#include <Locale.h>

#include <Assert.h>

clock\_t clock(void) {

return -1;

}

time\_t time(time\_t\* timePtr) {

int year;

short month, monthDay;

load\_register(ah, 0x2As);

interrupt(0x21s);

store\_register(cx, year);

store\_register(dh, month);

store\_register(dl, monthDay);

clear\_registers();

year -= 1900;

--month;

clear\_registers();

short hour, min, sec;

load\_register(ah, 0x2Cs);

interrupt(0x21s);

store\_register(ch, hour);

store\_register(cl, min);

store\_register(dh, sec);

clear\_registers();

struct tm s = {sec, min, hour, monthDay, month, year, 0, 0, 1};

time\_t time = mktime(&s);

if (timePtr != NULL) {

\*timePtr = time;

}

return time;

}

time\_t mktime(struct tm\* tp) {

int leapDays = ((tp->tm\_year - 69) / 4);

int totalDays = 365 \* (tp->tm\_year - 69) + leapDays + (tp->tm\_mday - 1);

BOOL leapYear = ((tp->tm\_year % 4) == 0);

if (leapYear) {

int leapMonths[] = {0, 31, 60, 91, 121, 152, 182, 213, 243, 274, 304, 335};

totalDays += leapMonths[tp->tm\_mon];

}

else {

int regularMonths[] = {0, 31, 59, 90, 120, 151, 181, 212, 242, 273, 303,334};

totalDays += regularMonths[tp->tm\_mon];

}

unsigned long totalSeconds = (86400lu \* totalDays) + (3600lu \* tp->tm\_hour)+

(60lu \* tp->tm\_min) + tp->tm\_sec;

unsigned long result = (((unsigned long)tp->tm\_isdst) << 31) |

(0x7FFFFFFF & totalSeconds);

return result;

}

static struct tm g\_timeStruct;

#define JAN\_1\_1969 3 // Wednesday January 1, 1969

#define DAYS\_OF\_FOUR\_YEARS 1461

#define DAYS\_OF\_ONE\_YEAR 365

struct tm\* localtime(const time\_t\* timePtr) {

if (timePtr != NULL) {

time\_t time = \*timePtr;

g\_timeStruct.tm\_isdst = (time >> 31);

unsigned long totalSeconds = (time & 0x7FFFFFFF);

{ unsigned long totalDays = totalSeconds / 86400lu;

g\_timeStruct.tm\_wday = (int) ((totalDays + JAN\_1\_1969) % 7);

int fourYears = totalDays / DAYS\_OF\_FOUR\_YEARS;

totalDays %= DAYS\_OF\_FOUR\_YEARS;

int years = totalDays / DAYS\_OF\_ONE\_YEAR;

totalDays %= DAYS\_OF\_ONE\_YEAR;

g\_timeStruct.tm\_year = 69 + (4 \* fourYears) + years;

g\_timeStruct.tm\_yday = totalDays;

BOOL leapYear = ((g\_timeStruct.tm\_year % 4) == 0);

if (leapYear) {

int leapMonths[] = {0, 31, 60, 91, 121, 152,

182, 213, 243, 274, 304, 335, 366},

arraySize = (sizeof leapMonths) / (sizeof leapMonths[0]), index;

for (index = 1; index < arraySize; ++index) {

if (totalDays < leapMonths[index]) {

g\_timeStruct.tm\_mon = index - 1;

g\_timeStruct.tm\_mday = (totalDays % leapMonths[index - 1]) + 1;

break;

}

}

}

else {

int regularMonths[] = {0, 31, 59, 90, 120, 151,

181, 212, 242, 273, 303, 334, 365},

arraySize = (sizeof regularMonths) / (sizeof regularMonths[0]),

index;

for (index = 1; index < arraySize; ++index) {

if (totalDays < regularMonths[index]) {

g\_timeStruct.tm\_mon = index - 1;

g\_timeStruct.tm\_mday = (totalDays % regularMonths[index - 1]) + 1;

break;

}

}

}

}

{ unsigned long daySeconds = totalSeconds % 86400lu;

g\_timeStruct.tm\_hour = (int) (daySeconds / 3600);

g\_timeStruct.tm\_min = (int) ((daySeconds % 3600) / 60);

g\_timeStruct.tm\_sec = (int) (daySeconds % 60);

}

return &g\_timeStruct;

}

return NULL;

}

struct tm\* localtimeX(const time\_t\* timePtr) {

if (timePtr != NULL) {

time\_t time = \*timePtr;

g\_timeStruct.tm\_isdst = (time >> 31);

unsigned long totalSeconds = (time & 0x7FFFFFFF);

{ unsigned long totalDays = totalSeconds / 86400lu;

g\_timeStruct.tm\_wday = (int) ((totalDays + JAN\_1\_1969) % 7);

int year;

BOOL leapYear;

for (year = 69;; ++year) {

leapYear = ((year % 4) == 0);

int yearDays = leapYear ? 366 : 365;

if (totalDays >= yearDays) {

totalDays -= yearDays;

}

else {

break;

}

}

g\_timeStruct.tm\_year = year;

g\_timeStruct.tm\_yday = totalDays;

int month, months[12] = {31, leapYear ? 29 : 28, 31, 30, 31, 30,

31, 31, 30, 31, 30, 31};

for (month = 0; month < 12; ++month) {

if (totalDays >= months[month]) {

totalDays -= months[month];

}

else {

break;

}

}

g\_timeStruct.tm\_mon = month;

g\_timeStruct.tm\_mday = totalDays + 1;

}

{ unsigned long daySeconds = totalSeconds % 86400lu;

g\_timeStruct.tm\_hour = (int) (daySeconds / 3600lu);

g\_timeStruct.tm\_min = (int) ((daySeconds % 3600lu) / 60lu);

g\_timeStruct.tm\_sec = (int) (daySeconds % 60);

}

return &g\_timeStruct;

}

return NULL;

}

double difftime(time\_t time1, time\_t time2) {

long totalSeconds1 = (time1 &0x7FFFFFFF),

totalSeconds2 = (time2 &0x7FFFFFFF);

return (double) (totalSeconds2 - totalSeconds1);

}

static char g\_timeString[256];

static char\* g\_defaultShortDayList[] =

{"Sun", "Mon", "Tue", "Wed", "Thu", "Fri", "Sat"};

static char\* g\_defaultLongDayList[] = {"Sunday", "Monday", "Tuesday",

"Wednesday", "Thursday", "Friday", "Saturday"};

static char\* g\_defaultShortMonthList[] = {"Jan", "Feb", "Mar", "Apr", "May",

"Jun", "Jul", "Aug", "Sep", "Oct", "Nov", "Dec"};

static char\* g\_defaultLongMonthList[] = {"January", "February", "March",

"April", "May", "June", "July", "August",

"September", "October", "November", "December"};

void default\_test() {

printf("<%p>", g\_defaultShortDayList);

printf("<%s>", g\_defaultShortDayList[1]);

}

char\* asctime(const struct tm\* tp) {

struct lconv\* localeConvPtr = NULL;//localeconv();

char\*\* shortDayList = (localeConvPtr != NULL) ?

localeConvPtr->shortDayList : NULL;

char\*\* shortMonthList = (localeConvPtr != NULL) ?

localeConvPtr->shortMonthList : NULL;

shortDayList = (shortDayList != NULL) ? shortDayList

: g\_defaultShortDayList;

shortMonthList = (shortMonthList != NULL) ? shortMonthList

: g\_defaultShortMonthList;

sprintf(g\_timeString, "%s %s %2i %02i:%02i:%02i %04i\n",

shortDayList[tp->tm\_wday], shortMonthList[tp->tm\_mon],

tp->tm\_mday, tp->tm\_hour, tp->tm\_min,

tp->tm\_sec, tp->tm\_year + 1900);

return g\_timeString;

}

char\* ctime(const time\_t\* time) {

return asctime(localtime(time));

}

struct tm\* gmtime(const time\_t\* timePtr) {

time\_t t = \*timePtr;

int timeZone = 0;

struct tm\* tmPtr = localtime(timePtr);

struct lconv\* localeConvPtr = localeconv();

if (localeConvPtr != NULL) {

timeZone = tmPtr->tm\_isdst ? localeConvPtr->summerTimeZone

: localeConvPtr->winterTimeZone;

}

t -= 3600l \* timeZone;

return localtime(&t);

}

size\_t strftime(char\* s, size\_t smax, const char\* fmt, const struct tm\* tp) {

struct lconv\* localeConvPtr = localeconv();

char\*\* shortDayList = (localeConvPtr != NULL) ?

(localeConvPtr->shortDayList) : NULL;

char\*\* shortMonthList = (localeConvPtr != NULL) ?

(localeConvPtr->shortMonthList) : NULL;

char\*\* longDayList = (localeConvPtr != NULL) ?

(localeConvPtr->longDayList) : NULL;

char\*\* longMonthList = (localeConvPtr != NULL) ?

(localeConvPtr->longMonthList) : NULL;

strcpy(s, "");

shortDayList = (shortDayList != NULL) ? shortDayList : g\_defaultShortDayList;

longDayList = (longDayList != NULL) ? longDayList : g\_defaultLongDayList;

shortMonthList = (shortMonthList != NULL) ? shortMonthList

: g\_defaultShortMonthList;

longMonthList = (longMonthList != NULL) ? longMonthList

: g\_defaultLongMonthList;

int index;

for (index = 0; fmt[index] != '\0'; ++index) {

char add[20];

if (fmt[index] == '%') {

switch (fmt[++index]) {

case 'a':

strcpy(add, shortDayList[tp->tm\_wday]);

break;

case 'A':

strcpy(add, longDayList[tp->tm\_wday]);

break;

case 'b':

strcpy(add, shortMonthList[tp->tm\_mon]);

break;

case 'B':

strcpy(add, longMonthList[tp->tm\_mon]);

break;

case 'c':

sprintf(add, "%04d-%02d-%02d %02d:%02d:%02d", 1900 + tp->tm\_year,

tp->tm\_mon + 1, tp->tm\_mday, tp->tm\_hour, tp->tm\_min,

tp->tm\_sec);

break;

case 'd':

sprintf(add, "%02d", tp->tm\_mday);

break;

case 'H':

sprintf(add, "%02d", tp->tm\_hour);

break;

case 'I':

sprintf(add, "%02d", (tp->tm\_hour % 12) + 1);

break;

case 'j':

sprintf(add, "%03d", tp->tm\_yday);

break;

case 'm':

sprintf(add, "%02d", tp->tm\_mon);

break;

case 'M':

sprintf(add, "%02d", tp->tm\_min);

break;

case 'p':

sprintf(add, "%s", (tp->tm\_hour < 12) ? "AM" : "PM");

break;

case 'S':

sprintf(add, "%02d", tp->tm\_sec);

break;

case 'U':

sprintf(add, "%02d", 0 /\*WeekNumberSunday(tp) \*/);

break;

case 'w':

sprintf(add, "%02d", tp->tm\_wday);

break;

case 'W':

sprintf(add, "%02d", 0 /\* WeekNumberMonday(tp) \*/);

break;

case 'x':

sprintf(add, "%04d-%02d-%02d", 1900 + tp->tm\_year, tp->tm\_mon,

tp->tm\_mday);

break;

case 'X':

sprintf(add, "%02d:%02d:%02d", tp->tm\_hour, tp->tm\_min, tp->tm\_sec);

break;

case 'y':

sprintf(add, "%02d", tp->tm\_year % 100);

break;

case 'Y':

sprintf(add, "%04d", 1900 + tp->tm\_year);

break;

case 'Z':

strcpy(add, "");

break;

case '%':

strcpy(add, "%");

}

}

else {

add[0] = fmt[index];

add[1] = '\0';

}

if ((strlen(s) + strlen(add)) < smax) {

strcat(s, add);

}

else {

break;

}

}

return strlen(s);

}

# Main.java

# The C Grammar

This grammar of the C compiler of this book is based on (and to a large extent identical with) the grammar defined in the *American National Standard for Information systems – Programming Language C, X3.159-1989* standard, which is described in the second edition of *The C Programming Language* by Brian W. Kernighan and Dennis M. Ritchie.

## The Preprocessor Grammar

The following grammar is implemented in CUP in Chapter 3.

control\_line ::=

# define identifier token-sequence

| # define identifier( identifier, … , identifier ) token-sequence

| # undef identifier

| # include <filename>

| # include “filename“

| # line constant

| # line constant “filename”

| # error optional\_token\_sequence

| # pragma optional\_token\_sequence

| #

| preprocessor\_conditional

optional\_token\_sequence ::=

/\* empty \*/

| token\_sequence

preprocessor\_conditional ::=

if-line text optional\_elif-parts optional\_else\_line # endif

if-line ::=

# if constant\_expression

| # ifdef identifier

| # ifndef identifier

optional\_elif-parts ::=

/\* empty \*/

| elif-part optional\_elif-parts

elif-part ::=

# elif constant\_expression text

optional\_else\_part ::=

/\* empty \*/

| #else text

## The Language Grammar

The following grammar is implemented in Bison in Chapter 6.

*translation*\_*unit* ::=

*external*\_*declaration*

| *translation*\_*unit* *external*\_*declaration*

*external*\_*declaration* ::=

*function*\_*definition*

| *declaration*

*function*\_*definition* ::=

*declarator* *optional*\_*declaration*\_*list* **(** *optional*\_*statement*\_*list* **)**

| *declaration*\_*specifier*\_*list* *declarator* *optional*\_*declaration*\_*list*

**(** *optional*\_*statement*\_*list* **)**

*optional*\_*declaration*\_*list* ::=

/\* *empty* \*/

| *optional*\_*declaration*\_*list* *declaration*

*declaration* ::=

*declaration*\_*specifier*\_*list* **;**

| *declaration*\_*specifier*\_*list* *declarator*\_*list* **;**

*declaration*\_*specifier*\_*list* ::=

*declaration*\_*specifier*

| *declaration*\_*specifier*\_*list* *declaration*\_*specifier*

*declaration*\_*specifier* ::=

**const** | **volatile** | **auto** | **register** | **static** | **extern** | **typedef** | **void**

| **char** | **wchar\_t** | **short** | **int** | **long** | **float** | **double** | **signed** | **unsigned**

| *struct\_union*\_*specifier |* *enum*\_*specifier* | *typedef*\_*name*

*struct\_union*\_*specifier* ::=

struct\_union *optional*\_*identifier* **(** *declaration*\_*list* **)**

| *struct\_union* **identifier**

*struct\_union ::=*

**struct** | **union**

*optional*\_*identifier* ::=

/\* *empty* \*/

| **identifier**

*declaration*\_*list* ::=

*declaration*

| *declaration*\_*list* *declaration*

*enum*\_*specifier* ::=

**enum** *optional*\_*identifier* **(** *enum*\_*list* **)**

| **enum** **identifier**

*enum*\_*list* ::=

*enum*

| *enum*\_*list* **,** *enum*

*enum* ::=

**identifier**

| **identifier** **=** *const*\_*expression*

*declarator*\_*list* ::=

*init*\_*bitfield*\_*declarator*

| *declarator*\_*list* **,** *init*\_*bitfield*\_*declarator*

*init*\_*bitfield*\_*declarator* ::=

*declarator*

| *declarator* **=** *initializer*

| *declarator* **:** *const*\_*expression*

| **:** *const*\_*expression*

*declarator* ::=

*optional*\_*pointer*\_*list* *direct*\_*declarator*

*direct*\_*declarator* ::=

**identifier**

| **(** *declarator* **)**

| *direct*\_*declarator* **[** *optional*\_*const*\_*expression* **]**

| *direct*\_*declarator* **(** *parameter*\_*ellipse*\_*list* **)**

| *direct*\_*declarator* **(** *optional*\_*identifier*\_*list* **)**

*optional*\_*pointer*\_*list* ::=

/\* *empty* \*/

| *pointer*\_*list*

*pointer*\_*list* ::=

*pointer*

| *pointer*\_*list* *pointer*

*pointer* ::=

**\*** | **\*** *declaration*\_*specifier*\_*list*

*parameter*\_*ellipse*\_*list* ::=

*parameter*\_*list*

| *parameter*\_*list* **,** **...**

*parameter*\_*list* ::=

*parameter*\_*declaration*

| *parameter*\_*list* **,** *parameter*\_*declaration*

*parameter*\_*declaration* ::=

*declaration*\_*specifier*\_*list*

| *declaration*\_*specifier*\_*list* *declarator*

| *declaration*\_*specifier*\_*list* *abstract*\_*declarator*

*optional*\_*identifier*\_*list* ::=

/\* *empty* \*/

| *identifier*\_*list*

*identifier*\_*list* ::=

**identifier**

| *identifier*\_*list* **,** **identifier**

*initializer*\_*list* ::=

*initializer*

| *initializer*\_*list* **,** *initializer*

*initializer* ::=

*assignment*\_*expression*

| **(** *initializer*\_*list* **)**

| **(** *initializer*\_*list* **,** **)**

*type*\_*name* ::=

*declaration*\_*specifier*\_*list*

| *declaration*\_*specifier*\_*list* *abstract*\_*declarator*

*abstract*\_*declarator* ::=

*pointer*\_*list*

| *optional*\_*pointer*\_*list* *direct*\_*abstract*\_*declarator*

*direct*\_*abstract*\_*declarator* ::=

**(** *abstract*\_*declarator* **)**

| **[** *optional*\_*const*\_*expression* **]**

| *direct*\_*abstract*\_*declarator* **[** *optional*\_*const*\_*expression* **]**

| **(** *optional*\_*parameter*\_*ellipse*\_*list* **)**

| *direct*\_*abstract*\_*declarator* **(** *optional*\_*parameter*\_*ellipse*\_*list* **)**

*optional*\_*statement*\_*list* ::=

/\* *empty* \*/

| *statement*\_*list*

*statement*\_*list* ::=

*statement*

| *statement*\_*list* *statement*

*statement* ::=

*open*\_*statement*

| *closed*\_*statement*

*open*\_*statement* ::=

**if** **(** *expression* **)** *statement*

| **if** **(** *expression* **)** *closed*\_*statement* **else** *open*\_*statement*

| **switch** **(** *expression* **)** *open*\_*statement*

| **case** *const*\_*expression* **:** *open*\_*statement*

| **while** **(** *expression* **)** *open*\_*statement*

| **for** **(** *optional*\_*expression* **;** *optional*\_*expression* **;**

*optional*\_*expression* **)** *open*\_*statement*

| **identifier** **:** *open*\_*statement*

*closed*\_*statement* ::=

**if** **(** *expression* **)** *closed*\_*statement* **else** *closed*\_*statement*

| **switch** **(** *expression* **)** *closed*\_*statement*

| **while** **(** *expression* **)** *closed*\_*statement*

| **for** **(** *optional*\_*expression* **;** *optional*\_*expression* **;**

*optional*\_*expression* **)** *closed*\_*statement*

| *declaration*

| **case** *const*\_*expression* **:** *closed*\_*statement*

| **default** **:** *closed*\_*statement*

| *optional*\_*expression* **;**

| **(** *optional*\_*statement*\_*list* **)**

| **do** *statement* **while** **(** *expression* **)** **;**

| **goto** **identifier ;**

| **continue ;**

| **break ;**

| **return** *optional*\_*expression* **;**

| **load\_register** **(** **identifier ,** *expression* **) ;**

| **store\_register** **(** **identifier** **,** *expression* **) ;**

| **store\_flagbyte** **(** *expression* **) ;**

| **jump\_register** **(** **identifier** **) ;**

| **interupt** **(** *const*\_*expression* **) ;**

*optional*\_*expression* ::=

/\* *empty* \*/

| *expression*

*expression* ::=

*assignment*\_*expression*

| *expression* **,** *assignment*\_*expression*

*assignment*\_*expression* ::=

*conditional*\_*expression*

| *prefix*\_*expression* *assignment*\_*operator* *assignment*\_*expression*

*assignment*\_*operator* ::=

**=** | \***=** | /**=** | %**=** | +**=** | -**=** | <<**=** | <<**=** | &**=** | |**=** | ^**=**

*conditional*\_*expression* ::=

*logical*\_*or*\_*expression*

| *logical*\_*or*\_*expression*

**?** *expression* **:** *conditional*\_*expression*

*optional*\_*const*\_*expression* ::=

/\* *empty* \*/

| *const*\_*expression*

*const*\_*expression* ::=

*conditional*\_*expression*

*logical*\_*or*\_*expression* ::=

*logical*\_*and*\_*expression*

| *logical*\_*or*\_*expression* *||* *logical*\_*and*\_*expression*

*logical*\_*and*\_*expression* ::=

*bitwise*\_*ior*\_*expression*

| *bitwise*\_*and*\_*expression* **&&** *bitwise*\_*ior*\_*expression*

*bitwise*\_*ior*\_*expression* ::=

*bitwise*\_*xor*\_*expression*

| *bitwise*\_*ior*\_*expression* **|** *bitwise*\_*xor*\_*expression*

*bitwise*\_*xor*\_*expression* ::=

*bitwise*\_*and*\_*expression*

| *bitwise*\_*xor*\_*expression* **^** *bitwise*\_*and*\_*expression*

*bitwise*\_*and*\_*expression* ::=

*equality*\_*expression*

| *bitwise*\_*and*\_*expression* **&** *equality*\_*expression*

*equality*\_*expression* ::=

*relation*\_*expression*

| *equality*\_*expression* *equality*\_*operator* *relation*\_*expression*

*equality*\_*operator* ::=

**==** | **!=**

*relation*\_*expression* ::=

*shift*\_*expression*

| *relation*\_*expression* *relation*\_*operator* *shift*\_*expression*

*relation*\_*operator* ::=

**<** | **<=** | **>** | **>=**

*shift*\_*expression* ::=

*add*\_*expression*

| *shift*\_*expression* *shift*\_*operator* *add*\_*expression*

*shift*\_*operator* ::=

**<<** | **>>**

*add*\_*expression* ::=

*multiply*\_*expression*

| *add*\_*expression* *binary*\_*add*\_*operator* *multiply*\_*expression*

*binary*\_*add*\_*operator* ::=

**+** | **-**

*multiply*\_*expression* ::=

*cast*\_*expression*

| *multiply*\_*expression* *multiply*\_*operator* *cast*\_*expression*

*multiply*\_*operator* ::=

**\*** | **/** | **%**

*cast*\_*expression* ::=

*prefix*\_*expression*

| **(** *type*\_*name* **)** *cast*\_*expression*

*prefix*\_*expression* ::=

*postfix*\_*expression*

| *increment*\_*operator* *prefix*\_*expression*

| *prefix*\_*add*\_*operator* *cast*\_*expression*

| **!** *cast*\_*expression*

| **~** *cast*\_*expression*

| **&** *cast*\_*expression*

| **\*** *cast*\_*expression*

| **sizeof** *prefix*\_*expression*

| **sizeof** **(** *type*\_*name* **)**

*prefix*\_*add*\_*operator* ::=

**+** | **-**

*increment*\_*operator* ::=

**++** | **--**

*postfix*\_*expression* ::=

*primary*\_*expression*

| *postfix*\_*expression* **[** *expression* **]**

| *postfix*\_*expression* **(** *optional*\_*argument*\_*expression*\_*list* **)**

| *postfix*\_*expression* **.** **identifier**

| *postfix*\_*expression* **->** **identifier**

| *postfix*\_*expression* *increment*\_*operator*

*primary*\_*expression* ::=

**identifier**

| **value**

| **(** *expression* **)**

*optional*\_*argument*\_*expression*\_*list* ::=

/\* *empty* \*/

| *argument*\_*expression*\_*list*

*argument*\_*expression*\_*list* ::=

*assignment*\_*expression*

| *argument*\_*expression*\_*list* , *assignment*\_*expression*

# A Crash Course in C

This appendix gives a crash course in the C programming language. The story of C began in the late sixties with the Multics project, which was a predecessor to the the UNIX operating system. The first two versions of Multics were written in the assembler language. However, since programming in assembly languages was rather cumbersome, an appropriate high-level language was called for.

The researchers Dennis Ritchie and Ken Thompon created the language B by simplifying the research language BCPL[[12]](#footnote-12) in 1969. The new versions, called New B, were introduced in 1972. The name did soon evolve into C. In 1978, Brian Kernighan and Dennis Ritchie wrote the book The C Programming Language, which introduced the informal K&R C standard, named after the authors. In 1983, a working group was formed by the American National Standard Institute (ANSI) in order to define a C standard, which was finally adopted in 1989. The ANSI C standard is described in an appendix of the second edition of The C Programming Languages.

## The Compiler and the Linker

The source code is the the actual text the programmer writes. The compiler is the program that translates the source code into object code, which the computer can read. The linker puts several compiled files into an executable file

Let us say we have a C program in the source code file Program.c and a routine used by the program in Routine.c. Furthermore, the program calls a function in the standard library. In this case, the compiler translates the source code into object code and the linker joins the code into the executable file Program.exe.

Compiler

Program.c

Linker

Program.obj

Routine.c

Routine.obj

Compiler

Standard Library

Program.exe

If the compiler reports an error we refer to it as compile-time error, and if an error occurs during the execution of the program we call it a run-time error.

## The Hello-World Program

In the first edition of The C Programming Language the hello-world example was introduced; that is, a small program that prints the text “Hello, World!”[[13]](#footnote-13) on the screen. To write a hello-world programoften means to write a small program to test that the compiler and linker work properly. The execution of the program does always start with the function main.

#include <stdio.h>

void main(void) {

puts("Hello, World!");

}

When executing, the program above generates the following printout.



## Comments

In C, it is possible to insert comments to describe and clarify the meaning of the program. The comments are ignored by the compiler[[14]](#footnote-14). There are two types of comments: line comments and block comments. Line comments start with two slashes and ends at the end of the line.

printf("Hello, World!"); // Prints "Hello, World!".

Block comments begin with a slash and an asterisk and ends with an asterisk and a slash. A block comment may range over several lines.

/\* This is an example of a C program. It writes the text

"Hello, World!" at the screen. \*/

#include <stdio.h>

void main(void) {

printf("Hello, World!");

}

Block comments cannot be nested. The following example will result in a compile-time error.

/\* A block comment cannot be /\* nested inside \*/ another

block comment. \*/

A piece of advice is that you use the line comments for regular comments, and save the block comments for situations when you need to comment a whole block of code for debugging purposes. Actually, there is a third level of comments if you use conditional programming, see macros (Section 22.9) for details.

#if 0

This is an example of a C program. It writes the text

"Hello, World!" at the screen.

#endif

#include <stdio.h>

void main(void) {

printf("Hello, World!");

}

### Types and Variables

There are several types in C. They can be divided into two groups: simple and compunded. The simple types can be further classified into integral, floating, and logical types. The compunded types are arrays, structures, unions, and pointers. They all (directly or indirectly) composed by simple types. We can also define a type with our own integer values, called the enumeration type.

### Simple Types

There are four simple types intended for storing integers: char, short int, int, and long int. They are called the integral types. The types short int and long int may be abbreviated to short and long, respectively. As the names implies, they are designed for storing characters, small integers, normal integers, and large integers, respectively. The exact limits of the values possible to store varies between different compilers. Such property that is not defined in the standard, but is rather defined by the compiler, linker, or operating system, is said to implementation dependent.

Furthermore, the integral types may be signed or unsigned. An unsigned type must not have negative values. If the word signed or unsigned is left out, a short int, int, and long int will be signed. Weather a char will be signed or unsigned is implementation dependent. However, a char is always one byte long, which means that it always holds a single character, regardless whether it is unsigned or not.

The next category of simple types is the floating types, which are used to store decimal numbers. The types are float, double, and long double; where float stores the smallest value and long double the largest one. How large values each type can store is implementation dependent. A floating type cannot be unsigned.

In many languages, there is a logical type (named bool, boolean, or logical) that stores logical value true or false. In C, however, there is no such type[[15]](#footnote-15). Instead, int is used as a logical type, where zero represents false and all other values represent true.

### Variables

A variable can be regarded as a box in the memory. In almost every case, we do not need to know its exact memory address. A variable always has a name, a type, and a value. We define a variable by simple writing its type and name. If we want to, we can initialize the variable; that is, assign it a value. If we do not, the variable’s value is undefined[[16]](#footnote-16). In this chapter, that value is denoted by a question mark (?).

int i = 123, j;

double d = 3.14;

char c = 'a';

123

i

?

j

3.14

d

'a'

c

We can transform values between the types by stating the new type within parentheses. The use of stating the type is not strictly necessary. However, if omitted the compiler may give warnings[[17]](#footnote-17).

int i = 123;

double x = 1.23;

int j = (int) x;

double y = (double) i;

### Constants

As the name implies, a constant is a variable whose value cannot be altered since it has been initialized. Unlike variables, constants must always be initialized and are often written in capital letters.

const double PI = 3.14;

### Output

As noted in the first section of this chapter, the standard function for output is printf (formatted print) writes to the standard output device (normally a text window).

The first argument is the format strings. With it, we can write every kind of value. However, there are many format codes to keep track of. A format code begins with the per cent character (%) followed with optional digits and signs ending with a character. We can write all the types we have went through so far. The character '\n' represents a new line. Below follows a list of the most common format codes.

|  |  |
| --- | --- |
| Code | Type |
| d | decimal integer |
| o | octal integer |
| x | hexadecimal integer |
| f | double |
| lf | long double |
| c | char |
| s | char \* |
| p | Prints the address of the variable in hexadecimal form. |

In the program below, *i* represents an integer, *f[[18]](#footnote-18)* a double, and *c* a character. Note that it is the responsible of the programmer to match the values following the format string with values of the correct types. There is no built-in error handling and mistakes would likely result in a run-time error.

#include <stdio.h>

void main() {

char c = 'a';

int i = 123;

double d = 1.23;

printf("char %c, int %d, double %f\n", c, i, d);

}

There is also the puts function that prints a string followed by a new-line. The following two lines are equivalent.

printf("Hello, World!\n");

puts("Hello, world!");

### Input

The standard functions for input is and scanf (formatted scan), it reads from the standard input device (normally the keyboard).

Similar to printf, its first argument is the format string. The format codes mostly agree with those of printf, but not completely. See the table at the end of the chapter for a list of format codes. For instance, scanf expects lf to read a double value, in contrast to printf that expects f*[[19]](#footnote-19)*.

Another difference is that while printf wants the values corresponding to the format codes, scanf wants the addresses of the variables that are going to store the values. This is accomplished by using the ampersand (&), see the Section 11.1.1 on pointers. Below follows a list of the most common format codes.

|  |  |
| --- | --- |
| Code | Type |
| d | decimal integer |
| o | octal integer |
| x | hexadecimal integer |
| f | float |
| lf | double |
| ld | long double |
| c | Char |
| S | char \* |

In the program below, note that scanf expects lf to represent a double float, in contrast to printf that expects f.

#include <stdio.h>

void main() {

char c;

int i;

double d;

printf("char: ");

scanf("%c", &c);

printf("int: ");

scanf("%d", &i);

printf("double: ");

scanf("%lf", &d);

printf("char %c, int %d, double %f\n", c, i, d);

}

There is also the gets function that reads a string until it reaches a new-line, which differs from scanf that reads a string until it reaches a white-space character[[20]](#footnote-20). It the user prints “Hello World” followed by a new-line twice when executing the code below, gets will place “Hello world” is s, while scanf will place “Hello” in t. Note that we do not give the addresses of s or t, since they are pointer to characters in themselves.

char s[20], t[20];

gets(s);

scanf("%s", t);

### Enumerations

An enumeration is a way to create our own integral type. We can define which values a variable of the type can store. In practice, however, enumerations are essentially an easy way to define integer constants[[21]](#footnote-21).

enum Cars {FORD, VOLVO, TOYOTA, VOLKSWAGEN};

Unless we state otherwise, the constants are assigned to zero, one, two, three, and so on. In the example above, FORD is an integer constant with the value zero; VOLVO has the value one, TOYOTA three, and VOLKSWAGEN four.

We do not have to name the enumeration type. In the example above, Cars can be omitted. We can also assign an integer value to some (or all) of the constants. In the example below, TOYOTA is assigned the value 10. The constants without assigned values will be given the value of the preceding constant before, adding one. This implies that VOLKSWAGEN will be assigned the value 11.

enum {FORD, VOLVO, TOYOTA = 10, VOLKSWAGEN};

### Arrays

An array is a variable compiled by several values of the same type. The values are stored on consecutive locations in memory. The array size may be stated explicitly of implicitly by a list of values. In the following example, a is given the size three, b is given size two and c is given size four, even though only its first two values are defined, which may cause the compiler to emit a warning. Naturally, d is given the size three.

int a[3] = {11, 12, 13};

double b[] = {1.2, 3.4};

char c[4] = {'a', 'b'}, d[3];

d

11

a

b

c

12

13

1.2

3.4

'a'

'b'

?

?

?

?

?

0

1

2

0

1

0

1

2

3

0

1

2

A value of an array can be accessed by index (brackets) notation.

int i = a[2];

double x = b[0];

char t = c[1];

13

i

1.2

x

'b'

t

### Characters and Strings

As stated in Section 2.1.1, a char is a small integer type intended to store exactly one character. A string stores a (possible empty) sequence of characters. There is no built in type for describing a string. Instead, a string is defined by an array of characters. Note that characters are enclosed by single quotations while strings are enclosed by double quotations.

char c = 'a';

char s[] = "Hello, World!";

'a'

c

"Hello, World!"

s

When defining a string, the compiler adds a special zero character '\0' at the end. Its task is to mark the end of the string. If the code above, the character array s has a size of fourteen (thirteen regular characters, including space, comma, and exlamation mark as well as the finishing zero character). The two following definitions are equivalent, they both define character arrays of size four (three regular characters and the finishing zero character).

char s1[] = "abc";

char s2[] = {'a', 'b', 'c', '\0'};

### The ASCII Table

So far we have assigned characters to char variables. Technically, the compiler translates a character literal into an integer in accordance to the ASCII table (see Appendix **Fel! Hittar inte referenskälla.**), where character has a corresponding integer value. For instance, ‘a’ corresponds to 97, ‘b’ to 98, and so on. The line above could have been written as the following code instead. The possibility to write character and string literal is for convenience only.

char s2[] = {97, 98, 99, 0};

### Pointers

A pointer is a variable containing the address of value. For example, let us say that the integer i has the value 123 that is stored on the memory address 10,000, and that the size of each value is four bytes. As p is a pointer to i, it holds the value 10,000.

10000

p

123

10000

9992

9996

10004

In almost all cases, we do not really need to know the address of the value. Therefore, a simpler (and clearer) way to illustrate the same thing is to draw an arrow from the pointer to the value.

p

123

i

The following code gives rise to the diagram above[[22]](#footnote-22), where the ampersand (&) denotes the address of the variable.

int i = 123;

int \*p = &i;

If we want to access the value pointed at, we use the asterisk (\*), which derefers the pointer (“follows the arrow”). As the address operator (&) gives the address of a variable (“against the arrow”), they address and dereferring operators can be regarded as each other reverses. Note that the asterisk is used on two occasions, when we define a pointer variable and when we derefer a pointer. The asterisk is in fact used on a third occasion, when multiplying two values. The ampersand is used on two occasions: address and bitwise and[[23]](#footnote-23).

int i = 999;

int \*p = &i; // Pointer definition.

int j = \*p; // Derefereeing of a pointer.

int a = 1, b = 2;

int c = a \* b; // Multiplication.

### Pointers and Dynamic Memory

Pointers can also be used to allocate dynamic memory. There is a section of the memory called the heap that is used for dynamically allocated memory blocks. The standard functions malloc, calloc, realloc and free used to allocate, reallocate, and deallocate the memory, respectively. Memory not dynamically allocated is referred to static memory.

p

123

int \*p = malloc(sizeof(int));

\*p = 123;

free(p);

We can also allocate memory for a whole array. Even though p is a pointer in the example below, we can use the array index notation to access a value of the array in the allocated memory block.

p

123

124

125

0

1

2

int \*p = calloc(3, sizeof(int));

p[0] = 123;

p[1] = 124;

p[2] = 125;

free(p);

Once we have allocated a memory, we can allocate a block of different size by calling realloc, which copies to contents of the memory block to new address, is necessary.

int \*p = malloc(sizeof(int));

p[0] = 123;

p = realloc(2 \* sizeof(int));

p[1] = 456;

free(p);

p

123

456

0

1

The difference between malloc and calloc is that calloc resets the bytes of the memory blocks to zero, which malloc do not. Otherwise, the following two calls are equivalent.

int \*p = calloc(3, sizeof(int));

int \*q = malloc(3 \* sizeof(int));

The predefined constant NULL holds the pointer equivalence of the zero value. We say that the pointer is set to null. In a diagram, we can simple write null.

p

NULL

int \*p = NULL;

Sometimes, the electric ground symbol is used to symbolize a null pointer. For this reason, a null pointer is sometimes said to be a grounded pointer.

p

There is a special type void, which actually marks the absence of a type. We can define a pointer to void; we cannot, however, derefer the pointer. It is useful in low-level application where we want to exam a specific location in memory.

void\* voidPtr = (void\*) 10000;

The void type is useful on one further occasion: to mark that a function does not return a value or misses parameters, see the function section later in this chapter.

In the example below, the memory block has been deallocated, but p has not been set to null. It has become a dangling pointer; it is not null and does not really points at anything. In spite of that, we try to access the value p points at. This is a dangerous operation and would most likely result in a run-time error.

p

dangling pointer

int \*p = malloc(sizeof(int));

\*p = 1;

free(p);

\*p = 2; // Undefined behavior

In the example below, we allocate memory for two pointers, p and q. Then we assign p to q, by doing so we have created a memory leak. There is no way we can access or deallocate the memory block that was pointed at by p. In fact, we deallocate the same memory block twice as both pointers by now points at the same memory block. This dangerous operation will most likely also result in a run-time error.

p

(a)

q

1

2

p

(b)

q

1

2

int \*p = malloc(sizoef(int)); // (a)

int \*q = malloc(sizeof(int));

\*p = 1;

\*q = 2;

p = q; // (b)

free(p); // Undefined behavior. Deallocates the same memory block twice,

free(q); // as p and q points at the same memory block. Also memory leak,

// since the memory block holding 1 cannot be accessed any more.

### Defining our own types

It is possible to define our own type with typedef, which is a great tool for increasing the readability of the code. However, too many defined types tend to make the code less readable. Therefore, a piece of advice would be to use typedef with care.

int i = 1;

typedef unsigned int unsigned\_int;

unsigned\_int u = 2;

typedef int\* int\_ptr;

int\_ptr ip = &i;

typedef unsigned\_int\* uint\_ptr;

uint\_ptr up = &u;

### The Type Size

The operator sizeof gives us the size of a type[[24]](#footnote-24), either by taking the type surrounded by parentheses or a value of the type. The size of a character is always one byte and the signed and unsigned forms of each integral type always have the same size. Otherwise, the sizes are decided by the compiler. In these cases, when a property is not specified by the standard by rather the compiler, we say the the property is implementation dependant. The operator returns a value of the type size\_t, which exact definition is also implementation dependant. However, it is often an unsigned integer.

#include <stdio.h>

void main() {

int intSize1 = sizeof (int);

int intSize2 = sizeof intSize1;

int\* intPtr = &intSize1;

int ptrSize = sizeof intPtr;

int array[3] = {1, 2, 3};

int arraySize = sizeof array;

printf("Integer sizes: %d and %d.\n", intSize1, intSize2);

printf("Pointer size: %d\n", ptrSize);

printf("Array size: %d\n\n", arraySize);

}

## Expressions and Operators

The operations of C are divided into the arithmetic, relational, logical, and bitwise operators as well as simple and compound assignment. Moreover, there is the conditional operator.

In the figure below, + is an operator, 1 and 2 are operands and the whole term is an expression.



Operator

Operand

Operand

### Arithmetic Operators

The arithmetic operators are addition (+), subtraction (-), multiplication (\*), division (/), and modulo (%). The first four operators are equivalent to the four fundamental rules of arithmetic. The operators can take operands of integral and floating types. The last operator, modulo, gives the remainder of integer division. If we mix integral and floating types in the expression, the result will have floating type. The modulo operator, however, only works with integral operands. The last assignment in the following code may give rise to a compiler warning as the result of the division is a double and is converted into an integer.

int a = 10, b = 3, c;

c = a + b; // 13

c = a - b; // 7

c = a \* b; // 30

c = a / b; // 3, integer division

c = a % 3; // 1, remainder

double d = 3.0, e;

e = a / d; // 3.333, floating type

### Pointer Arithmetic

The addition and subtraction operators are also applicable on pointers. It is called pointer arithmetic. An integral value can be added to or subtracted from a pointer. The value of the pointer is then changed by the integral value times the size of the type the pointer points at. As the void type is not really a type, but rather the absence of a type, it has no size. Therefore, we cannot perform pointer arithmetic on void pointers.

In the code below, let us assume that number is stored at memory location 10,000 and that the integer type has the size of four bytes. Then the pointer numberPtr will assume the values 10,000, 10,004, 10,008, and 10,012, not the values 10,000, 10,001, 10,002, and 10,003, as pointer arithmetic always takes the size of the type into consideration.

numberPtr

102

100

101

103

10008

10000

10004

10012

int number = 100;

int\* numberPtr = &number;

numberPtr = numberPtr + 1;

\*numberPtr = number + 1;

numberPtr = numberPtr + 1;

\*numberPtr = number + 2;

numberPtr = numberPtr + 1;

\*numberPtr = number + 3;

There is also possible to subtract two pointers pointing at the same type. The result will be the difference in bytes between their two memory locations divided with the size of the type. Again, pointer arithmetic always considers the type size.

int array[] = {1, 2, 3};

int\* p1 = &array[0];

int\* p2 = &array[2];

int diff = p2 - p1; // 2

The index notation for arrays is equivalent to the derefereeing of pointers together with pointer arithmetic. The second and third lines of the following code are by definition interchangeable.

int array[] = {1, 2, 3};

array[1] = array[2] + 1;

\*(array + 1) = \*(array + 2) + 1;

### Increment and decrement

There are two rather special operators: increment (++) and decrement (--)[[25]](#footnote-25). They add or subtract one from its operand. The operator can be placed before (prefix) or after (postfix) its operand.

int a = 1, b = 1;

++a; // 2, prefix increment

b++; // 2, postfix increment

However, there is a difference between prefix and postfix increment or decrement. In the prefix case below, the subtraction occurs first and the new value is returned; in the postfix case, after the subtraction the original value is returned.

int a = 1, b = 1, c, d;

c = --a; // c = 0, prefix decrement

d = b--; // d = 1, postfix decrement

### Relational Operators

There are six relational operators: equal to (==), not equal to (!=), less than (<), less than or equal to (<=), greater than (>), greater than or equal to (>=)[[26]](#footnote-26). Observe that the equal to operator is constituted by two equal signs rather than one. The operators give a logical value, true (an integer value not equal zero) or false (zero). The operands shall be of integral of floating type.

int i = 3;

double x = 1.2;

int b = i > 0; // true (an integer value not equal to zero, usually one)

int c = (x == 2); // false (always zero)

### Logical Operators

There are three logical operators: not (!), or (||), and and (&&). They take and return numerical values interpreted as logical values.

int i = 3;

int b, c, d, e;

b = (i == 3); // true (not zero)

c = !b; // false (zero)

d = b || c; // true (not zero)

e = b && c; // false (zero)

C applies lazy evaluation (also called short-circuit evaluation), which means that an expression will not be evaluated to a further extent than necessary to find its value. In the following example, the evaluation of the expression is completed when the left expression (i != 0) is evaluated to false. If the left expression is false, the whole expression must also be false because it needs both the left and right sub expressions to be true for the whole expression to be true. This gives that the right expression (1 / i == 1) will never be evaluated and the division with zero will never occur.

int i = 0;

int b = (i != 0) && (1 / i == 1); // false (zero);

### Bitwise Operators

An integer value can be viewed as a bit pattern. Our familiar decimal system has base 10; it can be marked with an index 10.



An integer value can also be viewed with the binary system, which has base 2. A single digit is called a bit, and the integer value is called a bit pattern. A bit may have the values one and zero.



There are four bitwise operations in C: inverse (~), and (&), or (|), and exclusive or (^). Exclusive or means that the result is one if one of its operand bits, but not both, is one. They all operate on integral values on bit level; that is, they examine each individual bit of an integer value.

101010102 101010102 101010102

& 100101102 | 100101102 ^ 100101102 ~ 100101102

----------- ------------ ------------ ------------

= 100000102 = 101111102 = 001111002 = 011010012

int a = 170; // 101010102

int b = 150; // 100101102

int c = a & b; // 100000102 = 13010

int d = a | b; // 101111102 = 19010

int e = a ^ b; // 001111002 = 6010

int f = ~b; // 011010012 = 10510

An integer value can also be shifted to the left (<<) or to the right (>>). Each left shift is equivalent with doubling the value, and each right shift is equivalent with (integer) dividing the value with 2. Overflowing bits are dropped for unsigned values; the behaviour of signed values is implementation dependent. Therefore, I recommend you only perform bitwise operations on unsigned values.

unsigned char a = 172; // 101011002

unsigned char b = a << 2; // 101100002 = 17610

unsigned char c = 166; // 101001102

unsigned char d = c >> 2; // 001010012 = 4110

### Assignment

There are two kinds of assignment operators: simple and compound. The simple variant is quite trivial, one or several variables are assigned the value of an expression. In the example below, a, b, and c are all assigned the value 123.

int a, b, c, d = 123;

a = d;

b = c = d;

The compound variant is more complicated. Let us start with the additional assignment operator. In the example below, the value of a is increased by the value of b; that is, a is given the value 4.

int a = 2, b = 4, c = 2;

a += c; // 4, equivalent to a = a + c.

b -= c; // 2, equivalent to a = a - c.

In a similar manner, there are operations -=, \*=, /=, %=, |=, &=, ^=, <<=, and >>=. Note the difference between a -= 2 and a =- 2. In the former case, a is decreased by 2. In the latter case, a is assigned -2.

### The Condition Operator

The condition operator resembles the if-else statement of the next section. It is the only C operator taking three operands. The first expression is evaluated. If it is true, the second expression is evaluated and its value is returned. If the first expression instead is false, the third expression is evaluated and its value is returned.

int a = 1, b = 2, max;

max = (a > b) ? a : b; // The maximal value of a and b.

Too frequent use of this operator tends to make the code compact and hard to read. A piece of advice is that you restrict your use of the operator to the trivial cases.

### Precedence and Associatively

Given the expression 1 + 2 \* 5, what is its value? It is 11 because we first multiply two with five and then add one. We say that multiplication has a higher precedence than addition.

What if we limit ourselves to one operator? Let us pick subtraction. What value has the expression 8 – 4 – 2? As we first subtract 4 from 8 and then subtract 2, the result is 2. As we evaluate the value from left to right, we say that subtraction is left associative.

Below follows a table over the priorities and associative of the C operators. The first operator in the table has the highest precedence.

|  |  |  |
| --- | --- | --- |
| Group | Operators | Associatively |
| Brackets and fields | () [] -> . | Left to Right |
| Unary operator | ! ~ ++ -- + - (type) sizeof | Right to Left |
| Arithmetic operators | \* / %  + - | Left to Right  Left to Right |
| Shift operators | << >> | Left to Right |
| Relation operators | < <= > >=  == != | Left to Right |
| Bitwise operators | &  ^  | | Left to Right |
| Logical operators | &&  || | Left to Right |
| Conditional operator | ?: | Right to Left |
| Assignment operators | = += -= \*/ /= %= &= ^= |= <<= >>= | Right to Left |
| Comma operator | , | Left to Right |

Note that unary +, -, &, and \* have higher precedence than their binary forms. Note that we always can change the evaluation order of an expression by inserting parantheses at appropriate posititions. The expression (1 + 2) \* 5 has the value 15.

## Statements

There are four kinds of statements in C: the selection, iteration, jump, and expression statements.

|  |  |
| --- | --- |
| Kind | Statements |
| Selection | if, if-else, switch |
| Iteration | for, while, do-while |
| Jump | break, continue, goto, return |
| Expression | expression ; |

### Selection statements

The if statement needs in its simplest form a logical expression to decide whether to execute the following statement. The example below means that the text will be output if i not equal zero.

if (i != 0) {

puts("i does not equal zero");

}

As all values not equal to zero are interpreted as true, the following statement is equivalent with the previous one.fexacltly

if (i) {

puts("i does not equal zero");

}

We can also attach an else part, which is executed if the expression is false.

if (i > 0) {

puts("i is greater than zero");

}

else { puts("i is not greater than zero");

}

Between the if and else part we can insert one or more else if parts.

if (i > 0) {

puts("i is greater than zero");

}

else if (i < 0) {

puts("i is less than zero");

}

else {

puts("i is equal to zero");

}

In the examples above, there is not strictly necessary to surround the output statements with brackets. However, in this book brackets are always used for sake of clarity. Moreover, it is always necessary in case of several statements. The brackets and the code in between are called a block.

if (i > 0) { int j = i + 1;

printf("j is %d\n", j);

}

A warning may be in order. In an if statement, it is perfectly legal to use one equal sign instead of two signs when comparing two values. As one equals sign is used for assignment, not comparison, the variable i in the following code will be assign the value one, and the expression will always be true.

if (i = 1) { // Always true.

// ...

}

One way to avoid the mistake above is to swap the variable and the value. As a value can be compared but not assigned, the compiler will issue an error message and the error will be recognized.

if (1 = i) { // Compile-time error.

// ...

}

The switch statement is simpler than the if statement, but not as powerful. It evaluates the switch expression and jumps to a case statement with the matching (constant) value. If no value matches, it jumps to the default statement, if present. The break statement is used to jump out of a switch or iteration statements. The switch expression must have an integral or pointer type and two case statements cannot hold the same value. We can only have one default statement, and it can be omitted. However, it needs not to be placed at the end of the switch statement, even though I recommend you to do so.

switch (i) { case 1:

puts("i is equal to 1");

break;

case 2:

puts("i is equal to 2");

break;

case 3:

puts("i is equal to 3");

break;

default:

puts("i is not equal to 1, 2, or 3.");

break;

}

It is important to remember the break statement. Otherwise, the execution would simple continue with the code attached to the next case statement[[27]](#footnote-27). However, in C we can use the fact that an omitted break statement makes the execution continue with the next statement to group several case statements together.

switch (i) { case 1:

case 2:

case 3:

printf("i is equal to 1, 2, or 3");

break;

// ...

}

### Iteration statements

Iteration statements, also called loops, iterate one or several statements as long as a certain condition is true; while is the simplest loop. It repeats one statement or a block of statements as long as the given expression is true. The example below uses the while statement to write the numbers one to ten.

int i = 1;

while (i <= 10) { printf("%d\n", i);

++i;

}

The same thing can be done with a do-while statement.

int i = 1;

do { printf("%d\n", i);

++i;

}

while (i <= 10);

However, the do-while statement is less powerful. If the expression is false to begin with, the while statement just skips the repetitions altogether, but the do-while statement must always execute the iteration statement at least once in order to reach the continuation expression at the end.

We can also use the for statement, which is a more compact variant of while. It takes three expressions, separated by semicolons. The first expression initializes the variable, the repetition continues as long as the second expression is true and the third expression is executed at the end of each repetition.

int i;

for (i = 1; i <= 10; ++i) { printf("%d\n", i);

}

Similar to the switch statement, an iteration statement can be interrupted by the break statement. Below is an infinite loop (remember that every non-zero value is regarded as true) that is stopped by the break statement.

int i = 1;

while (1) { printf("%d\n", i);

++i;

if (i > 10) { break;

}

}

The same effect can be archived by omitting the second expression of the for statement.

int i;

for (i = 1; ; ++i) { printf("%d\n", i);

if (i == 10) { break;

}

}

An iteration statement can also include a continue statement. It skips the rest of the current repetition. The following example prints the numbers from 1 to 10 with the exception of 5.

int i;

for (i = 1; i <= 10; ++i) { if (i == 5) { continue;

}

printf("%d\n", i);

}

The following example, however, will not work as intended. Because the continue statement will skip the rest of the while block, i will never be updated and we will be stuck in an infinite loop. Therefore, I recommend that you use continue with care, or avoid it altogether.

int i = 1;

while (i <= 10) { if (i == 5) { continue;

}

printf("%d\n", i);

++i;

}

### Jump statements

We can jump from one location to another inside the same function by marking the latter location with a label inside the block with the goto statement. The following code is yet another example of how to print the numbers from one to ten, inclusive. As you can, it is not as clear as the previous examples.

int i = 1;

label: printf("%d\n", i);

++i;

if (i <= 10) { goto label;

}

The goto statement is considered to give rise to unstructured code, so called “spagetti code.” I strongly recommended that you omit goto altogether. To put it bluntly: now when you seen it, forget it.

### Expression statements

An expression can form a statement.

a = b + 1; // Assignment operator.

puts("Hello, World!"); // Function call.

In the above examples, we are only interested in the side effects; that a is assigned a new value and that a text is written. We are allowed to write expression statements without side effects; even thought it has no meaning and is likely to be erased by the compiler.

a + b \* c;

### Volatile

A variable can be volatile, which has no functional effect but is rather a notification to the compiler that the variable shall not be subjected to optimization. Let us say we that the memory location 10000 is connected to an input port in a low-level application, and that we want to wait until we receive a signal through the port. If p is not marked as volatile, there is a risk that the compiler would replace the for-loop with another statement since the value of \*p does not seem to change inside the loop.

volatile int\* p = (void\*) 10000;

while (\*p != 0) {

// Wait.

}

## Functions

A function can be compared to a black box. We send in information (input) and we receive information (output). In C, the input values are called parameters and the output value is called the return value.

Input (Parameters)

Function

Output (Return Value)

To start with, let us try the function Square; it takes an integer and returns its square.

int Square(int n) {

return n \* n;

}

void main() {

int i = Square(2); // Square returns 4.

}

2

Square

4

In the example above, the parameter n in the Square definition is called a formal parameter, and the value 3 in the Square call in main is called an actual parameter or an argument.

Now, let us try a more complicated function: SquareRoot takes a value of double type and returns its square root. The idea is that the function iterates and calculates increasingly better root values by taking the mean value of the original value divided with the current root value and the previous root value. The process continues until the difference between two consecutive root values has reached an acceptable tolerance. Just like main[[28]](#footnote-28), a function can have local variables; root and oldRoot hold the current and previous value of the root, respectively.

#include <stdio.h>

double SquareRoot(double value) {

const double EPSILON = 1e-12;

double root = value, oldRoot = value;

while (1) { root = ((value / root) + root) / 2;

if ((oldRoot - root) <= EPSILON) { return root;

}

oldRoot = root;

}

}

void main() {

double input = 16;

printf("SquareRoot of %f: %f\n", input,

SquareRoot(input));

}

### Void Functions

A function does not have to return a value; in that case, we set void as the return type. As mentioned above, void is used to state the absence of a type rather than a type in itself. We can return from a void function by just stating return without a value.

void PrintSign(int value) {

if (value < 0) { puts("Negative.");

return;

}

if (value > 0) { puts("Positive.");

return;

}

puts("Zero");

}

There is not a problem if the execution of a void function reaches the end of the code, it just jump back to the calling function. However, a function not returning void shall always return a value before reaching the end of the function. Otherwise, the compiler often gives a warning.

### Local and global variables

There are two kinds of variables: local and global. A global variable is defined outside a function and a local variable is defined inside a function.

#include <stdio.h>

int global = 1;

void main() {

int local = 2;

printf("Global variable: %d, Local variable: %d.\n",

global, local);

}

A global and a local variable can have the same name. In that case, the name inside the function refers to the local variable. We cannot access the global variable in the inner scope[[29]](#footnote-29) when a local variable with same name is present.

#include <stdio.h>

int number = 1;

void main() {

int number = 2;

printf("Variable: %d.\n", number); // 2

}

One way to distinguish them is to precede the global variable name with ’g\_’.

#include <stdio.h>

int g\_number = 1;

void main() {

int number = 2;

printf("Global variable: %d, Local variable: %d.\n",

g\_number, number); // 1, 2

}

Global variables can only be initialized with constant values.

int g\_number1 = 1; // Right.

int g\_number2 = g\_number1 + 1; // Wrong.

### Inner Blocks with Local Variables

In C, you are allowed to use the bracket notation when initializing structures, unions, and arrays, but not when assigning[[30]](#footnote-30). Therefore, the following code will not work.

INT\_PAIR Block(int\* p) {

INT\_PAIR result;

if (p != NULL) {

result = {\*p - 1, \*p + 1}; // Wrong.

return result;

}

else {

result = {0, 0}; // Wrong.

return result;

}

}

One solution would be to a more clumsy code.

INT\_PAIR Block(int\* p) {

INT\_PAIR result;

if (p != NULL) {

result.a = \*p - 1;

result.b = \*p + 1;

return result;

}

else {

result.a = 0;

result.b = 0;

return result;

}

}

However, a more elegant solution would be to use inner blocks with local variables. Each block can define its own variable, which is especially valuable when initializing structures.

struct IntPair {

int a, b;

};

struct IntPair Block(int\* p) {

if (p != NULL) {

struct IntPair result = {\*p - 1, \*p + 1};

return result;

}

else {

struct IntPair empty = {0, 0};

return empty;

}

}

An alternative way is to use typedef.

typedef struct {

int a, b;

} INT\_PAIR;

INT\_PAIR Block(int\* p) {

if (p != NULL) {

INT\_PAIR result = {\*p - 1, \*p + 1};

return result;

}

else {

INT\_PAIR empty = {0, 0};

return empty;

}

}

### Call-by-Value and Call-by-Reference

Say we want to write a function for switching the values of two variables.

#include <stdio.h>

void Swap(int number1, int number2) {

int temp = number1; // (a)

number1 = number2; // (b)

number2 = temp; // (c)

}

void main() {

int num1 = 1, num2 = 2;

printf("Before: %d, %d\n", num1, num2); // 1, 2

Swap(num1, num2);

printf("After: %d, %d\n", num1, num2); // 1, 2

}

Unfortunately, it will not work; the variables keep their values. The explanation is that the values of firstNum and secondNum in main are copied into num1 and num2 in Swap. Then num1 and num2 exchange values with the help of temp. However, their values are not copied back into firstNum and secondNum in main.

1

2

main:

1

firstNum

2

Swap:

num2

1

temp

secondNum

num1

2

2

num2

1

temp

num1

2

1

num2

1

temp

num1

(a)

(b)

(c)

The problem can be solved with reference calls. Instead of sending the values of the actual parameters, we send theirs addresses by define the formal parameters as pointers to the type.

#include <stdio.h>

void Swap(int\* numberPtr1, int\* numberPtr2) {

int temp = \*numberPtr1; // (a)

\*numberPtr1 = \*numberPtr2; // (b)

\*numberPtr2 = temp; // (c)

}

void main() {

int num1 = 1, num2 = 2;

printf("Before: %d, %d\n", num1, num2); // 1, 2

Swap(&num1, &num2);

printf("After: %d, %d\n", num1, num2); // 2, 1

}

In this case, we do not send the values of firstNum and secondNum, but rather their addresses. Therefore, num1 and num2 in Swap does in fact contain the addresses of firstNum and secondNum on main. As in the reference section above, we illustrate this with dashed arrows. Therefore, when num1 and num2 exchanges values, in fact the values of firstNum and secondNum are exchanged.

1

2

firstNum

num2

1

temp

secondNum

num1

(a)

2

2

firstNum

num2

1

temp

num1

(b)

2

1

firstNum

num2

1

temp

num1

(c)

### Static, Extern, and Register Variables

In the function below, s\_count (the prefix ‘\_s’ is used for static variables) is a static local variable, which means that it is initialized when the execution of the program starts rather when the function is called. If s\_count was a regular local variable (without the keyword static), the function would, at every call, write that the function has been called once, as s\_count would be initialized to zero at every call.

void KeepCount() {

static int s\_count = 0;

s\_count++;

printf("This function has been called %d times.", s\_count);

}

If we define a global variable in one file, it will be accessable in another file if we declare[[31]](#footnote-31) it as extern. If we omit the extern keyword in the second file, the linker would complain about us defining two global variables with the same name.

File1.c

int g;

File2.c

extern int g;

Finally, a variable can also be marked with the keyword register, which is a notification to the compiler that the variable is suitable to place in a process register rather them the memory. The only function difference is that we cannot get the address of a register variable, since it might be located in a register.

register int i;

int\* p = &i; // Wrong

### Recursion

A function may call itself; it is called recursion. In the following example, the mathematical function factorial (n!) is implemented. It can be defined in two ways. The first definition is rather straightforward. The result of the function applied to a positive integer n is the product of all positive integers up to and including n.



int Factorial(int n) {

int result = 1, i;

for (i = 1; i <= n; ++ i) { result \*= i;

}

return result;

}

An equivalent definition involves a recursive call that is easier to implement.



int Factorial(int n) {

if (n == 1) {

return 1;

}

else { return n \* Factorial(n - 1);

}

}

### Definition and Declaration

It important to distinguish between definition and declaration. For a function, its definition generates code while the declaration is merely an item of information to the compiler. A function declaration is also called a prototype.

When it comes to mutual recursion (two functions calling each other), at least the second of them must have a prototype to avoid compiler warnings. I recommend that you put prototypes for all functions at the beginning of the file (or in a header file, see Section 22.7). In the following example, we use two functions to decide whether a given non-negative integer is even or odd according to the formulas below.





Note that it is not necessary to name the parameters in a function prototype. If names after all are given, they will be ignored by the compiler, which means that the parameters do not need to have the same names in the function declaration and definition. The only restriction is that two parameters cannot have the same name.

int IsEven(int);

int IsOdd(int n);

As C does not include a logical type, we use int to represent true (1) or false (0).

int IsEven(int num) {

if (num == 0) {

return 1;

}

else { return IsOdd(num - 1);

}

}

int IsOdd(int num) {

if (num == 0) {

return 0;

}

else {

return IsEven(num - 1);

}

}

One peculiar thing about prototypes in C is that they can have an unspecified parameter list. If the parameter list is completely omitted in a function prototype, every parameter list is allowed in the call. The following code is will not result in any compile-time errors. However, there is likely to be run-time errors.

void Print();

void main(void) {

Print();

Print(1);

Print(1, 2);

Print(1, 2, 3);

}

The obvious way to avoid the run-time errors is to always state the parameter list in function prototypes. If the function does not have any parameters, it can be stated by the void type. In that case, the function can only be called without parameters.

void Print(void);

### Higher-Order Functions

A function that takes another function as a parameter is called a higher-order function. Technically, it does not take the function itself as a parameter, but rather a pointer to the function. However, the pointer marker (\*) may be omitted. The parameter function is also called a callback function. The following example takes an array of the given size and applies the given function to each integer in the array. ApplyArray a higher order function and Apply is a callback function.

#include <stdio.h>

void ApplyArray(int intArray[], int size, int Apply(int)) {

int index;

for (index = 0; index < size; ++index) { intArray[index] = Apply(intArray[index]);

}

}

int Double(int number) {

return 2 \* number;

}

int Square(int number) {

return number \* number;

}

void PrintArray(int intArray[], int size) {

int index;

for (index = 0; index < size; ++index) { printf("%d ", intArray[index]);

}

printf("\n");

}

void main() {

int numberArray[] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};

int arraySize = sizeof numberArray / sizeof numberArray[0];

PrintArray(numberArray, arraySize);

// Doubles every value in the array.

ApplyArray(numberArray, arraySize, Double);

PrintArray(numberArray, arraySize);

// Squares every value in the array.

ApplyArray(numberArray, arraySize, Square);

PrintArray(numberArray, arraySize);

}

An alternative way is to define the type of the callback function. In the following code segment, the APPLY\_FUNC type is a pointer to a function that takes an integer parameter and returns an integer value.

typedef int (\*APPLY\_FUNC)(int);

void ApplyArray(int intArray[], int size,

APPLY\_FUNC pApplyFunc) {

int index;

for (index = 0; index < size; ++index) { intArray[index] = pApplyFunc(intArray[index]);

}

}

One additional point of the example above is how to of find the size of an array: we divide the size of the array with the size of its first value. This function does only work on static arrays, not on dynamically allocated arrays or array given as parameters to functions. A parameter array is in fact converted to a pointer to the array type. The following two function definitions are (by definition) equivalent. This gives that the function must have an extra integer parameter stating the array size.

void PrintArray(int intArray[], int arraySize) {

// ...

}

void PrintArray(int\* intArray, int arraySize) {

// ...

}

## Structures and Linked Lists

A struct is a compound type. Like arrays, it holds several values, called fields. However, in contrast to arrays, the values can hold different types.

A pointer may point at a struct value as well as a simple value, and a struct may have a pointer to another struct value as a field, which in turn points at another struct value, and so one. In this way, a linked list can be constructed. The list must end eventually, so the last pointer points at null. A pointer to the next cell in the list is called a link.

struct Cell {

int value;

struct Cell\* nextPtr;

};

struct Cell cell3 = {3, NULL};

struct Cell cell2 = {2, &cell3};

struct Cell cell1 = {1, &cell2};

1

value

nextPtr

2

value

nextPtr

3

value

nextPtr

NULL

### Stacks and Linked Lists

A stack is very valuable in a number of applications and it can be implemented with a linked list. We can add a value on top of the stack, we can inspect or remove the topmost value, and we can check whether the stack is empty. However, we cannot do anything to the values that are not on top. The function adding a new value on top of the stack is called push and the function removing it is called pop. Let us say that we push our stack three times with the values 1, 2, and 3. Then we can only access the topmost value, 3, and not the two below, 1 or 2.

1

2

3

push 1

push 2

push 3

Cell.h

struct Cell { int m\_value;

struct Cell\* m\_nextPtr;

};

typedef struct Cell CELL;

void CellInit(CELL\* cellPtr, int value, CELL\* nextPtr);

void CellSetValue(CELL\* cellPtr, int value);

int CellGetValue(CELL\* cellPtr);

void CellSetNext(CELL\* cellPtr, CELL\* nextPtr);

CELL\* CellGetNext(CELL\* cellPtr);

Cell.c

#include "Cell.h"

void CellInit(CELL\* cellPtr, int value, CELL\* nextPtr) {

cellPtr->m\_value = value;

cellPtr->m\_nextPtr = nextPtr;

}

void CellSetValue(CELL\* cellPtr, int value) {

cellPtr->m\_value = value;

}

int CellGetValue(CELL\* cellPtr) {

return cellPtr->m\_value;

}

void CellSetNext(CELL\* cellPtr, CELL\* nextPtr) {

cellPtr->m\_nextPtr = nextPtr;

}

CELL\* CellGetNext(CELL\* cellPtr) {

return cellPtr->m\_nextPtr;

}

Main.c

#include <stdlib.h>

#include "Cell.h"

void main() {

CELL\* cellPtr1 = malloc(sizeof(CELL));

CELL\* cellPtr2 = malloc(sizeof(CELL));

CELL\* cellPtr3 = malloc(sizeof(CELL));

CellInit(cellPtr3, 3, NULL);

CellInit(cellPtr2, 2, cellPtr3);

CellInit(cellPtr1, 1, cellPtr2);

}

The following struct is created by main above.

1

m\_value

m\_nextPtr

2

m\_value

m\_nextPtr

3

m\_value

m\_nextPtr

NULL

cellPtr1

cellPtr2

cellPtr3

However, there is one more thing to think about. What happens if we run out of dynamic memory or try to access the topmost value of an empty stack? We can deal with the problem in some different ways, everything from ignoring it to abort the execution. In this book, I have limited the error handling of memory shortage to the use of assert, which is a macro (see Section 22.9) that takes a logical value and aborts the execution if the value is false, adding an error message with information about the file name and the code line number. To keep things simple, let us use that function in the stack structure too.

Stack.h

struct Stack { CELL\* m\_firstCellPtr;

};

typedef struct Stack STACK;

void StackInit(STACK\* stackPtr);

void StackClear(STACK\* stackPtr);

void StackPush(STACK\* stackPtr, int value);

void StackPop(STACK\* stackPtr);

int StackTop(STACK\* stackPtr);

int StackIsEmpty(STACK\* stackPtr);

Stack.c

#include <stdlib.h>

#include <assert.h>

#include "Cell.h"

#include "Stack.h"

void StackInit(STACK\* stackPtr) {

stackPtr->m\_firstCellPtr = NULL;

}

void StackClear(STACK\* stackPtr) {

CELL\* cellPtr = stackPtr->m\_firstCellPtr;

while (cellPtr != NULL) { CELL\* tempCellPtr = cellPtr;

cellPtr = CellGetNext(cellPtr);

free(tempCellPtr);

}

}

void StackPush(STACK\* stackPtr, int value) {

CELL\* newCellPtr= malloc(sizeof(CELL));

assert(newCellPtr!= NULL);

CellInit(newCellPtr, value, stackPtr->m\_firstCellPtr);

stackPtr->m\_firstCellPtr = newCellPtr;

}

void StackPop(STACK\* stackPtr) {

CELL\* tempCellPtr;

assert(stackPtr->m\_firstCellPtr != NULL);

tempCellPtr = stackPtr->m\_firstCellPtr;

stackPtr->m\_firstCellPtr = CellGetNext(stackPtr->m\_firstCellPtr);

free(tempCellPtr);

}

int StackTop(STACK\* stackPtr) {

assert(stackPtr->m\_firstCellPtr != NULL);

return CellGetValue(stackPtr->m\_firstCellPtr);

}

int StackIsEmpty(STACK\* stackPtr) {

return (stackPtr->m\_firstCellPtr == NULL);

}

Main.c

#include <stdlib.h>

#include "Cell.h"

#include "Stack.h"

void main() {

STACK stack;

StackInit(&stack);

StackPush(&stack, 1);

StackPush(&stack, 2);

StackPush(&stack, 3);

StackClear(&stack);

}

The stack in main above gives rise to the following struct. However, when StackClear is called, it deallocates the allocated memory. All memory allocated with malloc or calloc must (or at least should) be deallocated with free.

m\_nextPtr

3

2

m\_value

m\_nextPtr

1

m\_value

m\_nextPtr

NULL

m\_value

m\_firstCellPtr

### Unions

A union is a compound type like struct. The difference is that while a struct places each of its members on different memory addresses in order for them to not intervene with each other, a union locates each of its members on the same address. This implies that when one member is assigned a value, it overwrites the other values. Unions are mostly used in low-level applications. In the code below, the two structures overlap each other.

struct \_Bits8Registers { // A character always has the size of 1 byte (8 bits).

unsigned char ah, al, bh, bl, ch, cl, dh, dl;

};

struct \_Bits16Registers { // Assuming short has a size of 2 bytes (16 bits).

unsigned short ax, bx, cx, dx;

};

union \_Registers {

struct \_Bits8Registers bits8Registers;

struct \_Bits16Registers bits16Registers;

};

### Bitfields

A bitfield is a struct with the possibility to define the size of the members in bits, which is useful in low-level applications. Only integral types can have bits, and the total number of bits must not exceed the number of bits an integer (eight times sizeof (int)). In the code below, the whole struct shares one byte (eight bits). The second field does not have a value; it only marks that the second bit shall be ignored.

struct Communicate {

int alert : 1;

int :1;

int read : 3;

int write : 3;

};

## Structures with function pointers

A struct can hold pointers to functions as well as values. However, in order to obtain the specific struct value, we have to add its address a pointer parameter[[32]](#footnote-32) in the function call.

#include <stdio.h>

struct Person { char name[100];

int age;

void (\*Print)(struct Person\*);

};

void PrintNameAndAge(struct Person\* personPtr) {

printf("Name: %s\n", personPtr->name);

printf("Age: %d\n\n", personPtr->age);

}

void PrintOnlyName(struct Person\* personPtr) {

printf("Name: %s\n\n", personPtr->name);

}

void PrintOnlyAge(struct Person\* personPtr) {

printf("Age: %d\n\n", personPtr->age);

}

void main() {

struct Person adam = {"Adam", 10, &PrintNameAndAge};

struct Person bertil = {"Bertil", 20, &PrintOnlyName};

struct Person ceasar = {"Ceasar", 30, &PrintOnlyAge};

adam.Print(&adam); // Prints the name and age.

bertil.Print(&bertil); // Prints only the name.

ceasar.Print(&ceasar); // Prints only the age.

}

## The Preprocessor

The preprocessor is a tool that precedes the compiler in interpreting the code. The #include directive is one of its parts. It opens the file and includes its text. So far, we have only included system header files, whose names are surrounded by arrow brackets (for instance, <StdIO.h>). Later on, we will include our own header files. Then we will use parentheses instead of arrow brackets. The difference is that the pre-processor looks for the system header files in a special system directory while it looks for our header files in the local directory (usually the source code directory).

Another part of the pre-processor is the macros. They acts like functions with the difference that they do not perform any type checking, they just replace text. We have already used the assert macro. A macro is introduced with the #define directive and is often written with capitals.

#define ADD(a, b) ((a) + (b))

printf("%d\n", ADD(1 + 2, 3 \* 4)); // 15

Unlike regular C code, if we add a page-break we have to mark it by a backslash.

#define ADD(a, b) ((a) + \

(b))

It is also possible to perform conditional programming by checking the value of macros. In the following example, we define a system integer according to the underlying operational system.

#ifdef WINDOWS

#define SYSINT int

#endif

#ifdef LINUX

#define SYSINT unsigned int

#endif

#ifdef MAC

#define SYSINT long int

#endif

SYSINT iOpData = 0;

Condition programming can also be used as a third level of comments (above line and block comments, see Section 22.9). In the following code the whole segment between the #if and #endif directives will be omitted, regardless of the comments, as zero always is interpreted as false.

#if 0

int Square(int value)

{ return value \* value; // Square.

}

#endif

## The Standard Library

The C standard library hold around 200 functions and some macros divided into several sections.

### File Management

We can open, write to, read from, and close files with the help of file pointers. A file pointer is a pointer to a value of the FILE type; it can be considered a connection to the file. The program below reads a series of integers from the text file Input.txt and writes their squares to the file Output.txt. The function feof returns zero when there is no more value to be read from the file. Finally, we must not forget to close the file.

#include <stdlib.h>

#include <assert.h>

#include <stdio.h>

void main(void) {

FILE\* inFile = fopen("Input.txt", "r");

FILE\* outFile = fopen("Output.txt", "w");

assert(inFile != NULL);

assert(outFile != NULL);

while (!feof(inFile)) { double value;

fscanf(inFile, "%lf", &value);

fprintf(outFile, "%f\n", value \* value);

}

fclose(inFile);

fclose(outFile);

}

The text files are written in plain texts and can be viewed by the editor.

Input.txt

1

2

3

4

5

Output.txt

1

4

9

16

25

We can also read and write binary data. The program below writes the number one to ten to the file Numbers.bin and then reads the same series of values from the file. The functions write and read takes the address of the value to be read or written and the size of the value in bytes. They return the number of bytes actually read or written. When reading, we can check whether we have reached the end of the file by counting the number of read bytes; if it is zero, we have reached the end.

BinaryFile.c

#include <stdlib.h>

#include <assert.h>

#include <stdio.h>

void main(void) {

int index, value;

FILE \*outFile = fopen("Number.bin", "w"), \*inFile;

assert(outFile != NULL);

for (index = 1; index <= 10; ++index) { fwrite(&index, 1, sizeof index, outFile);

}

fclose(outFile);

inFile = fopen("Number.bin", "r");

assert(inFile != NULL);

while (fread(&value, 1, sizeof value, inFile) > 0) { printf("%d\n", value);

}

fclose(inFile);

}

The values are stored in compressed form in the binary file Numbers.bin, why they are not readable in the editor. Here is a screen dump of the file.



### Program Parameters

It possible to start the program execution with parameters. In main, the argc parameter holds the number of input strings, and argv holds the string themselves.

#include <StdIO.h>

void main(int argc, char\* argv[]) {

int index;

printf("argc: %d\n", argc);

for (index = 0; index < argc; ++index) {

printf("argv[%d]: %s\n", index, argv[index]);

}

}

The parameters are given when the program executes. Note that the value of argv[0] always is the program path.



### Environment Functions

There is a set of functions that communicates with the surrounding environment: exit quits the execution and sends an integer value to the environment; atexit states a function that is called when exit is called; getenv reads the value of an environment variable, and system executes a system command. There are two macros [EXIT\_SUCCESS](http://www.cplusplus.com/reference/clibrary/cstdlib/EXIT_SUCCESS/) and [EXIT\_FAILURE](http://www.cplusplus.com/reference/clibrary/cstdlib/EXIT_FAILURE/) that can used with the exit function.

#include <StdIO.h>

#include <stdlib.h>

void ExitFunction(void) {

printf("The Program is exiting.\n");

}

void main() {

char\* path = getenv("PATH");

system("dir \*.c");

atexit(&ExitFunction);

if (path != NULL) {

printf("Path: %s.\n", path);

exit(EXIT\_SUCCESS);

}

else {

printf("Path not found.\n");

exit(EXIT\_FAILURE);

}

}

Another way to send an exit message to the surrounding environment is to define int as main’s return type. It has the same effect as calling exit in main (except that the exit function will not be called).

#include <StdIO.h>

#include <StdLib.h>

int main() {

char\* path = getenv("PATH");

if (path != NULL) {

printf("Path: %s.\n", path);

return EXIT\_SUCCESS;

}

else {

printf("Path not found.\n");

return EXIT\_FAILURE;

}

}

### Searching and Sorting

The bsearch function performs a binary search through a sorted list of values. We need to add a pointer function parameter specifying a function comparing two values.

#include <StdLib.h>

int CompareIntegers(const int\* intPtr1, const int\* intPtr2) {

return (\*intPtr1 < \*intPtr2) ? -1 : (((\*intPtr1 > \*intPtr2) ? 1 : 0));

}

void main() {

int intArray[] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10},

arraySize = sizeof intArray / sizeof intArray[0],

value = 6;

int\* resultPtr = bsearch(&value, intArray, arraySize,

sizeof (int), &CompareIntegers);

if (resultPtr != NULL) {

int index = ((int\*) resultPtr) - intArray;

printf("Index %d.\n", index);

}

else {

printf("Not found.\n");

}

}

There is also the qsort function that performs a fast sorting.

#include <StdLib.h>

void main() {

int index, intArray[] = {7, 2, 5, 9, 3, 1, 10, 8, 4, 6},

arraySize = sizeof intArray / sizeof intArray[0],

key = 5;

qsort(intArray, arraySize, sizeof (int), &CompareIntegers);

for (index = 0; index < arraySize; ++index) {

printf("%d\n", intArray[index]);

}

}

### Functions with Variable Number of Parameters

In C, there is possible to define functions with a variable number of parameters (such as printf and scanf). One condition is that the function has at least one regular parameter in order for the va\_start macro to hook up to. The va\_arg macro reads the arguments and the va\_end macro ends the reading. Note that the macros do not give any information about the number or types of the arguments. In the following code, we use the first argument count to count the number of arguments thereafter.

#include <stdio.h>

#include <stdarg.h>

void Print(int count, ...) {

va\_list arg\_list;

va\_start(arg\_list, count);

while (count > 0) { int value = va\_arg(arg\_list, int);

printf("%d ", value);

--count;

}

printf("\n");

va\_end(arg\_list);

}

void main(void) {

Print(1, 1);

Print(2, 1, 2);

Print(3, 1, 2, 3);

}

### String Management

There are several functions dealing with strings: strcpy copies a string; strcat adds a string; strlen gives the length of a string; strcmp compares two strings; strchr and strrchr gives the first and last occurrence of a character in a string. All functions look for the finishing null character (‘\0’), strcmp returns a value less than zero if the first string is smaller than second string, a value greater than zero if the second string is larger, and zero if they are equals. Note that the comparison continues until the finishing zero character has been found, even though the character arrays may be longer. As strchr and strrchr return a pointer to the found character (or null if it was not found), we can calculate the array index by subtracting the pointers.

int FirstIndexOf(char s[], char c) {

char\* p = strchr(s, c);

return (p != NULL) ? (p - s) : -1;

}

int LastIndexOf(char s[], char c) {

char\* p = strrchr(s, c);

return (p != NULL) ? (p - s) : -1;

}

void main(void) {

char s1[] = "Hello", s2[] = "World";

char t1[20], t2[20];

strcpy(t1, s1);

strcat(t1, s2);

sprintf(t2, "%s, %s!", s1, s2);

printf("t1: \"%s\", length of t1: %d, t2: \"%s\"\n",

t1, strlen(t1), t2);

printf("Comparing t1 and t2: %d\n", strcmp(t1, t2));

printf("First Index of 'l' in t2: %d, last index of 'l' in t2: %d.\n",

FirstIndexOf(t2, 'l'), LastIndexOf(t2, 'l'));

}

It is also possible to write to a string with sprintf and read from a string with sscanf. These function come in handy when to cast between strings and numerical values.

void main() {

int i;

double x;

char s1[20], s2[20];

sprintf(s1, "%d", 123);

sprintf(s2, "%f", 123.456e-3);

printf("s1 = \"%s\", s2 = \"%s\"\n", s1, s2);

sscanf(s1, "%d", &i);

sscanf(s2, "%lf", &x);

printf("i = %d, x = %f\n", i, x);

}

The functions return the number of values written or read, which gives that we can check that the conversion worked.

if (sscanf(s2, "%lf", &x) == 1) {

printf("x = %f\n", x);

}

else {

printf("Could not cast the string \"%s\".\n", s2);

}

Moreover, we can also count the number of read character and decide whether to whole string has been read.

sscanf(s2, "%lf%n", &x, &charCount);

if (charCount == strlen(s2)) {

printf("x = %f\n", x);

}

else {

printf("Could not cast the string \"%s\".\n", s2);

}

### Memory Management

There is also a set of function for general memory management: memset sets each byte of a memory block to a given value; memcmp compares the memory blocks; memcpy and memmove copies a memory block. The difference between them is that memcpy copies the bytes of the block directly when memmove uses a buffer, which gives that memcpy is faster and memmove is safer when copying overlapping blocks.

These functions work similar to their string counterparts. However, note that they do not look for a finishing character. Instead, the sizes of the function blocks are given as a parameter.

#include <StdLib.h>

void main() {

int a1[3], a2[3];

memset(a1, 1, sizeof a1);

memcpy(a2, a1, sizeof a2);

printf("Comparing a1 and a2: %d\n", memcmp(a1, a2, sizeof a2));

}

### Mathematical Functions

C standard library includes the trigonometric cos, sin, tan, acos, asin, and atan, and atan2; the hyperbolic functions cosh, sinh, and tanh; ceil and floor that rounds a floating value to the closest higher and lower value, respectively; exp returning the natural exponent; log returning the natural logarithm; log10 returning the logarithm of ten; sqrt returning the square root, and pow returning the power of two values. All of these functions takes and returns a double value (except atan2 and pow that takes two values). The trigonometric functions deal with angles in radians.

When converting a floating value to an integral value, it will always be truncated; that is, rounded to the integer value closest to zero. One way to find the closest integer value is to add 0.5 to the double value before converting it.

printf("double %f, rounded to the closest integer value: %d\n",

x, ((int) (x + 0.5)));

### Long Jumps

Regular jumps with goto can only occur in the same function body; long jumps with the standard function setjmp and longjump can, on the other hand, occur throw call sequences; setjmp saves the state of the program execution and longjmp uses that information to restore that state, which gives the illusion of a backwards jump.

In the program below, the user inputs a number, if it does not equal zero its inverse is return. If the number does equal zero, a long jump occurs. The call to setjump returns zero when the jump is set and a non-zero value when a long jump occurs[[33]](#footnote-33).

#include <stdio.h>

#include <setjmp.h>

jmp\_buf env;

double Divide(double numerator, double denominator) {

if (denominator == 0) { longjmp(env, -1); // -1 to represents an error state.

}

return numerator / denominator;

}

double Inverse(double number) {

return Divide(1.0, number);

}

void main() {

if (setjmp(env) == 0) { double value;

printf("Value: ");

scanf("%lf", &value);

printf("Inverse: %f\n", Inverse(value));

}

else {

puts("Division by zero.");

}

}

### Time

The time function gives the number of seconds since January 1, 1970. That value can then be used to fill a tm struct with values regarding local date and time by calling the localtime function. If we call instead call gmtime, the [Greenwich Mean Time](http://sv.wikipedia.org/wiki/Greenwich_Mean_Time) (Coordinated Universal Time) will be given. Note that the tm\_year field gives the year starting from 1900.

#include <Time.h>

void main() {

time\_t t = time(NULL);

struct tm time;

char \*months[] = {"Jan", "Feb", "Mar", "Apr",

"May", "Jun", "Jul", "Aug",

"Sep", "Oct", "Nov","Dec"};

char \*days[] = {"Sun", "Mon", "Tue", "Wed",

"Thu", "Fri", "Sat"};

localtime\_s(&time, &t);

printf("%02d:%02d:%02d\n", time.tm\_hour, time.tm\_min, time.tm\_sec);

printf("%s %s %d, %d\n", days[time.tm\_wday], months[time.tm\_mon],

time.tm\_mday, 1900 + time.tm\_year);

printf("Day of year: %d.\n", time.tm\_yday);

printf("Daylight saving time: %s.\n", time.tm\_isdst ? "Yes" : "No");

}

It is also possible to measure the period between two events.

#include <StdIO.h>

#include <Time.h>

void main() {

time\_t t1 = time(NULL);

int index;

for (index = 0; index < 100000; ++index) {

printf("%d\n", index);

}

{ time\_t t2 = time(NULL);

printf("The loop took %f seconds.\n", difftime(t2, t1));

}

}

### Random Numbers

It is possible to generate a sequence of pseudo-random numbers with the rand function. In order to initialize the sequence we need to sow a random seed with the srand function. One way to generate different seeds is to use the time function. The [RAND\_MAX](http://www.cplusplus.com/reference/clibrary/cstdlib/RAND_MAX/) macro gives the largest possible random numbers; however, usually we want random numbers in a certain interval. The following code generates five sequences of ten numbers between 1 and 100.

#include <StdLib.h>

void main() {

int iIndex1, iIndex2;

for (iIndex1 = 0; iIndex1 < 5; ++iIndex1) {

srand((unsigned int) time(NULL));

for (iIndex2 = 0; iIndex2 < 10; ++iIndex2) {

int randomNumber = rand() % 100 + 1;

printf("%d\n", randomNumber);

}

printf("\n");

}

}

### Limits of Integral and Floating Types

Since the type sizes are implementation dependent, there are a set of macros defining the minimum and maximum values of the integrated and floating types. The minimum values of unsigned integral types are always zero.

#include <stdio.h>

#include <limits.h> // The integral type limit constants.

#include <float.h> // The floating type limit constants.

void main() {

printf("Minimum signed char: %d\n", SCHAR\_MIN);

printf("Maximum signed char: %d\n", SCHAR\_MAX);

printf("Minimum signed short int: %d\n", SHRT\_MIN);

printf("Maximum signed short int: %d\n", SHRT\_MAX);

printf("Minimum signed int: %d\n", INT\_MIN);

printf("Maximum signed int: %d\n", INT\_MAX);

printf("Minimum signed long int: %ld\n", LONG\_MIN);

printf("Maximum signed long int: %ld\n\n", LONG\_MAX);

printf("Maximum unsigned char: %u\n", UCHAR\_MAX);

printf("Maximum unsigned short int: %u\n", USHRT\_MAX);

printf("Maximum unsigned int: %u\n", UINT\_MAX);

printf("Maximum unsigned long int: %lu\n\n", ULONG\_MAX);

printf("Minimum float: %e\n", FLT\_MIN);

printf("Maximum float: %e\n", FLT\_MAX);

printf("Minimum double: %e\n", DBL\_MIN);

printf("Maximum double: %e\n", DBL\_MAX);

printf("Minimum long double: %le\n", LDBL\_MIN);

printf("Maximum long double: %le\n", LDBL\_MAX);

}

### Character Functions

There are a set of functions for classifying a character.

if (isdigit(c)) {

printf("%c is a digit.", c);

}

Below follows a table of the character functions. The ASCII codes are stated within brackets.

|  |  |
| --- | --- |
| Function | Checks |
| **isalpha** | Letter: a-z and A-Z. |
| **islower** | Lower case letter: a-z. |
| **isupper** | Upper case letter: A-Z. |
| **isdigit** | Digit: 0-9. |
| **Isxdigit** | Hexadecimal digit: a-f, a-f, and 0-9. |
| **Isalnum** | Alphanumeric character: letter or digit. |
| **Isspace** | White-space character: space (0x20), horizontal tab (0x09), vertical tab (0x0B), new-line (0x0A), form-feed (0x0D), or carriage return (0x0C). |
| **Iscntrl** | Control character: characters with ASCII codes 0x00 to 0x0F, inclusive, and delete (0x1F). |
| **Isprint** | Printable character: any character that is not a control character. |
| **Isgraph** | Graphical character: any printable character that is not a white-space character. |
| **ispunct** | Punctuation character: any graphical character that is not an alphanumeric character. |

There is also the tolower and toupper functions that returns the character in lower and upper case, respectively.

printf("%c in lower case: %c, and in upper case: %c.",

c, tolower(c), toupper(c));

## Further Reading

I recommend *The C Programming Language* by Brian Kernighan and Dennis Ritchie (2nd Edition, Prentice Hall, 1988). It describes ANSI C and the standard library in a short and consist way, and holds an appendix about the ANSI C standard.

If you want to learn more about C programming, I recommend *C How to Program* by P. J. [Deitel](http://www.amazon.com/s/ref=ntt_athr_dp_sr_1?_encoding=UTF8&sort=relevancerank&search-alias=books&ie=UTF8&field-author=Paul%20J.%20Deitel) (6th Edition, Prentice Hall, 2009), which which discusses in depth the theory as well as practice of programming in C, with many clear and comprehensive examples. C in a Nutshell by [Peter Prinz, Tony Crawford](http://shop.oreilly.com/product/9780596006976.do#tab_04) (O'Reilly Media, 2005) and Practical C Programming (3th edition, O'Reilly Media, 1997) are also good choices.

When you have mastered the basic of C programming, I recommend Expert C Programming by P. Van Der Linden (Prentice Hall, 1994). Do not worry; you do not have to be an expert to read it.

# A Crash Course in Java

# Foreword

The compiler is made up of several components:

#### The Type System

Every programming language has a type system, which determines the properties of its types.

#### The Symbol Table

In many languages, the source code can hold a definition of a variables that is later referred to. In that case the, the variable (its name, type, and potential value) needs to be stored in a symbol table.

#### Scanning

The scanner takes a stream of characters and put them together into a sequence of *tokens*; that is, the least meaningful parts of the source code. For instance, the characters ‘i' and ‘f’ is put together into the keyword **if**, and the characters ‘!’ and ‘=’ is put together into the operator **not\_equal**.

#### Parsing and Middle Code Generation

The syntax of a programming language is often defined by a grammar, and the parser checks whether the token sequence generated by the scanner is accepted by the grammar. Moreover, the parser does also generate middle code.

#### Middle Code Optimization

The middle code generated by the parser often holds unnecessary instructions that needs to be removed.

#### Target Code Generation

When the middle code has been generated and optimized, it is time to generate the target code. In this book, we actually generate target code for two formats: Intel x86 assembler source code and executable code in the 16-bits COM-format.

#### Register Allocation

When the target code has been generated, the registers need to be allocated. We use a graph coloring algorithm to do that.

#### Linking

When each source file has been completely compiled, they need to be linked together into a executable file format.

**The Preprocessor**

The C programming language comes with a preprocessor that needs to be.

**The Standard Library**

The C programming language also comes with a standard library.

3. I have another book proposal for you, this time about compiler technology. The book will describe the design and implementation of a C++ compiler (including the preprocessor, linker, and standard library), with all code given.

So far, I have written a C compiler and I plan to extend it to a C++ compiler in the near future. Even though C++ is a much larger language than C, it not too much extra job. Most of the job has already been done with the C compiler.

The idea of the book is basically the same as the Windows book: I explain the details and provide all source code of the compiler. The source code written in Java with CUP and JLex and generates both assembler code and executable code.

The books below are the latest editions of three classic books. They describe in detail the basic features of compiler construction and to some extent advanced features. The structure of these books is similar to mine, one can say that I have followed these books. However, while these books give many small examples, my book describes in each chapter the features necessary to be included in a C++ compiler.

Alfred V. Aho, Lam, Monica S. Lam, Ravi Sethi, Jeffrey D. Ullman. Compilers – Principles, Techniques, and Tools, 2nd edition. Prentice Hall, 2006.

Keith Cooper, Linda Torczon. Engineering a Compiler, 2nd edition. Morgan Kaufman, 2011.

Charles N. Fischer, Ron K. Cytron, Richard J. LeBlanc. Crafting a Compiler. Pearson Education, 2009.

The following books describe compiler construction in Java, C, and ML. They describe a compiler for a smaller language with excerpt from the code given in the book (the complete code is downloadable). However, these books are briefer than the books above, and describe the compiler features in less detail.

Andrew W Appel. Modern Compiler Implementation in Java, 2nd edition. Cambridge University Press, 2002.

Andrew W Appel. Modern Compiler Implementation in C, 2nd edition. Cambridge University Press, 2004.

Andrew W Appel. Modern Compiler Implementation in ML, 2nd edition. Cambridge University Press, 2004.

The following book could be said to be closely related to mine, as it also described the code for a C compiler. However, this book is more or less unreadable. It presents the code, which (in my option) is unstructured, and it does not include much text describing the code. Hopefully, my book will describe the topic in a clearer way. I also plan to describe the theory of a compiler, not just the code to implement it. Moreover, this book does not include the preprocessor, linker, or standard library, and the code is written in C, not Java.

Christopher W. Fraser, David R. Hanson. A Retargetable C Compiler : Design and Implementation. Benjamin Cummings, 1995.

This books deals with the more advanced parts of compiler construction; that is, compiler optimization. Even though I will include some optimization at the end of my book, it does not compete with this book.

Steven S. Muchnick. Advanced Compiler Design & Implementation. Morgan Kaufmann, 2003.

Here is a suggestion for the chapters:

1. Introduction. Introduces the compiler phases described in the following chapters by demonstrating a compiler for a small toy language generating MIPS-code executable in the SPIM simulator.

2. The Scanner. The scanner is a relatively small part of the compiler. Its task is to put together characters into tokens (the smallest significant parts of the source code). Examples are as key words, operators, and numerical values. The scanner is written in JLex, a lexical generator for Java, based on Lex for C.

3. The Parser. The parser, on the other hand, is a large part of the compiler. Its task is to confirm that the given tokens (generated by the scanner) agree with the syntax of the programming language, which is represented by a set of grammatical rules. The parser is defined in CUP, a syntax generator for Java, based on Yacc for C. Each rule can also be equipped with code dealing with type checking and target code generation. However, I will have try to omit as much as possible of the code in this chapter. Most of the code is made up of calls to methods defined in later chapters.

4. Declarations and the Symbol Table. C++ has a rather complicated declaration system with aggregated types such as classes, structures and arrays with a corresponding complicated syntax. All defined variables and functions are stored in the symbol table, which is a hierarchical structure matching the program structure.

5. Type Checking. C++ has a rather large set of operators with complicated rules that need to be checked.

6. Intermediate Code Generation. When the types of expressions and statements are checked, three-address-code are generated, which is a simple intermediate language used to represent the code internally. Type conversation is also included in this chapter.

7. Static and Dynamic Initialization. In C++, variables can be initialized. If the variable is static, the data shall be generated and placed in the static area of the final target code. If it is dynamic, the initialization will result in a series of assignments. One thing that complicates the issue is that it is possible to initialize hierarchical structures made up by structures and arrays with one flat list.

8. Intermediate Code Optimization. The intermediate code can be optimized. For instance, code that is never reached and assignment of variables that are never used shall be removed.

9. Target Code Generation. The target code of the C++ compiler of this book is Intel x86, which is harder to deal with than the MIPS code the first chapter. It holds a few registers and registers of different sizes overlap. Therefore, the register allocation process needs to closely keep track on which variable values that are currently stored in the registers.

10. The Preprocessor. Before the actually compilation starts, the source code has been traversed by the preprocessor that replaces macros with text, includes header files, and provides conditional programming.

11. The Linker. When the target code has finally been generated, it becomes stored into an object file. As the source code can be distributed over several files, the target code need to be merged into one executable file. This is the task of the linker, it merges the code and data area, resolve the static and extern references and generate the executable file.

12. The C Standard Library, made up by functions and macros, which will be included by the linker in the final executable file.

13. The C++ Standard Library, made up by classes, to a certain extent is based on C standard library.

The compiler is made up of several phases:

Preprocessor

Source Code

Parser

Processed Code

Syntax Tree Optimister

Syntax Tree

Middle Code Generator

Optimized Syntax Tree

Middle Code Optimizer

Middle Code List

Optimized Middle Code List

Scanner

Symbol Table

Preprocessor

Source Code

Parser

Processed Code

Syntax Tree Optimister

Syntax Tree

Middle Code Generator

Optimized Syntax Tree

Middle Code Optimizer

Middle Code List

Optimized Middle Code List

Scanner

Symbol Table

1. Then name Yacc is an abbreviation for Yet Another Compiler-Compiler. However, it is pronounced as the animal yak. Therefore, the name Bison referrers to the rather similar animal. [↑](#footnote-ref-1)
2. In other languages, which syntax includes the keyword then, it is called as the if-then-else problem. [↑](#footnote-ref-2)
3. Technically, it is possible to store null values in the map, in which case put would return null even when the key is present in the map. However, in this book, we never store null values in any map. [↑](#footnote-ref-3)
4. As we all know, goto has no place in well-structurated programs. Labels and goto are included in the language of historical reasons only. More recent languages, such as Java, have omitted goto. [↑](#footnote-ref-4)
5. An extra parameter in an elliptic function call is a parameter that is not defined in the regular parameter list, but rather added as a result of the elliptic status. For instance, in the printf call below only the format string "int: %i, double %f" is a regular parameter, 123 and 23.45 are extra parameters.

   printf("int: %i, double %f", 123, 23.45); [↑](#footnote-ref-5)
6. This means that is both operands are integral of the same size, with one signed and the other unsigned, the signed value is converted to the unsigned type, and the resulting type is the unsigned type. [↑](#footnote-ref-6)
7. This due to historical reasons. In the early versions of C, there was no void type. Instead, char was used a generic type in pointer expressions. Therefore, it was (and still is) important that a char always holds one byte. [↑](#footnote-ref-7)
8. In some books, the process of generating the low-level executable code is called the target code generation. However, in this book I have chosen the term object code generation in order not confuse it with source and targets in jump statements. That would have resulted in rather awkward variable names, like targetTargetCode. [↑](#footnote-ref-8)
9. Actually, a function’s start address and its the entry point differs in one case only: when main is defined with the argc and argv parameters and is called recursively, which is allowed in the C standard. In that case, the first part of the main function holds code for initializing argc and argv, which shall not be executed in a recursive call. [↑](#footnote-ref-9)
10. As mention in Section 17.1, a function’s start address and its the entry point differs only when main is defined with the argc and argv parameters and is called recursively. [↑](#footnote-ref-10)
11. We could add code that modifies the code in a way similar to generateJumpInfo in Section **Fel! Hittar inte referenskälla.**. However, I feel that it is too much trouble for saving a few bytes. After all, most function calls are performed by long jumps. [↑](#footnote-ref-11)
12. BCPL stands for Basic Command Programming Language and was developed at join effort at London University and Cambridge University. Later on, BCPL has jokely been said to be an acronym for the Before C Programming Language. [↑](#footnote-ref-12)
13. Actually, their program wrote "hello, world\n" without a capital letter or an exclamation mark. Their example was inherited from the internal 1974 memorandum Programming in C: A Tutorial by [Brian Kernighan](http://en.wikipedia.org/wiki/Brian_Kernighan). The oldest known instance of the usage of the words ”hello” and ”world” together occurred in Kernighan’s 1972 tutorial *Introduction to the Language* [B](http://en.wikipedia.org/wiki/B_(programming_language)), even though the syntax was different from C. [↑](#footnote-ref-13)
14. Technically, the compiler replaces every block comment with a single space character, which makes it unwise to place a comment inside a name. [↑](#footnote-ref-14)
15. Usually, the problem is solved by the following macros (see Section 22.9):

    #define BOOL int

    #define TRUE 1

    #define FALSE 0 [↑](#footnote-ref-15)
16. Technically, it is given the value that happened to be stored on the memory address. [↑](#footnote-ref-16)
17. Depending of the compiler and its warning level settings. [↑](#footnote-ref-17)
18. Actually, f represent both float or double. The reason is that a float value is always converted to double when used as input to printf (or any other function). [↑](#footnote-ref-18)
19. As we inputs pointers to the variables rather than the variables themselves when calling scanf, no conversation from float to double occur. That is why f represents float and lf represents double. That lf (long float) represents double in scanf is an emergency solution since d represents an integer in decimal notation in both printf and scanf. [↑](#footnote-ref-19)
20. A white-space character is a space (‘ ‘), horizontal tabulator (‘\t’), vertical tabulator (‘\v’), new-line (‘\n’), form-feed (‘\f’), or carriage return (‘\r’). [↑](#footnote-ref-20)
21. Technically, each enumeration value holds the type unsigned int. [↑](#footnote-ref-21)
22. However, we have no knowledge about the actual address; it may be 10,000 or anything else. [↑](#footnote-ref-22)
23. In C++, the ampersand is also used to define references variables, which are not included in C. [↑](#footnote-ref-23)
24. More accurately, the number of bytes needed to hold a value of the type, [↑](#footnote-ref-24)
25. They were added to C in order to match two specific assembler instructions. Since then, they have stuck. More recent languages, such as C++, Java, and C#, do also support increment and decrement. [↑](#footnote-ref-25)
26. Some programmers prefer to denote equal to and not equal to as comparison operators. [↑](#footnote-ref-26)
27. In C#, the break statement is mandatory. [↑](#footnote-ref-27)
28. Technically, main is a regular function. The only difference, compared to other functions, is that the linker generates code that starts the program execution by calling main. Therefore, the program must include exactly one **main** function. [↑](#footnote-ref-28)
29. In C++, it is possible to access the global variable by prefixing colons, like ::number. [↑](#footnote-ref-29)
30. There is no rational explanation, just accept it. [↑](#footnote-ref-30)
31. Note the distinction of variable definition and declaration: a definition reserves memory for the variable while a declaration is simple a notification to the compiler. [↑](#footnote-ref-31)
32. In object-oriented languages the pointer is hidden and referred to as the this pointer. [↑](#footnote-ref-32)
33. This construction performs the same task as exception handling does in object-oriented languages. [↑](#footnote-ref-33)