2013

Hoisting C Structures Into Clay In Device Drivers

Lianne Lairmore
Bucknell University, lel011@bucknell.edu

Follow this and additional works at: https://digitalcommons.bucknell.edu/honors_theses

HOISTING C STRUCTURES INTO CLAY IN DEVICE DRIVERS

by

Lianne E. Lairmore

A Thesis

Presented to the Faculty of

Bucknell University

in Partial Fulfillment of the Requirements for the Degree of

Bachelor of Science with Honors in Computer Science

April 26, 2013

Approved: _Lea Wittie_

Lea Wittie
Thesis Advisor

Steven Guattery
Chair, Department of Computer Science

HOISTING C STRUCTURES INTO CLAY IN DEVICE DRIVERS

by

Lianne E. Lairmore

A Thesis

Presented to the Faculty of
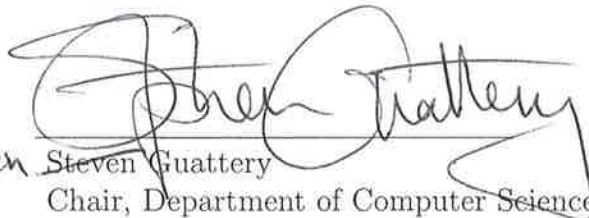
Bucknell University

in Partial Fulfillment of the Requirements for the Degree of

Bachelor of Science with Honors in Computer Science

April 26, 2013

Approved: _____

Lea Wittie
Thesis Advisor

_____

Steven Guattery
Chair, Department of Computer Science

Contents

Abstract

This project addresses the unreliability of operating system code, in particular in device drivers. Device driver software is the interface between the operating system and the device's hardware. Device drivers are written in low level code, making them difficult to understand. Almost all device drivers are written in the programming language C which allows for direct manipulation of memory. Due to the complexity of manual movement of data, most mistakes in operating systems occur in device driver code [1].

The programming language Clay can be used to check device driver code at compile-time. Clay does most of its error checking statically to minimize the overhead of run-time checks in order to stay competitive with C's performance time. The Clay compiler can detect a lot more types of errors than the C compiler like buffer overflows, kernel stack overflows, NULL pointer uses, freed memory uses, and aliasing errors [8]. Clay code that successfully compiles is guaranteed to run without failing on errors that Clay can detect. Even though C is unsafe, currently most device drivers are written in it.

Not only are device drivers the part of the operating system most likely to fail, they also are the largest part of the operating system[2]. As rewriting every existing device driver in Clay by hand would be impractical, this thesis is part of a project to automate translation of existing drivers from C to Clay. Although C and Clay both allow low level manipulation of data and fill the same niche for developing low level code, they have different syntax, type systems, and paradigms. This paper explores how C can be translated into Clay. It identifies what part of C device drivers cannot be translated into Clay and what information drivers in Clay will require that C cannot provide. It also explains how these translations will occur by explaining how each C structure is represented in the compiler and how these structures are changed to represent a Clay structure.

Chapter 1

Introduction and Background

1.1 The C Programming Language

The C programming language was developed in 1972 and has remained popular since it creation[5]. Although it was developed to be a general purpose language, it is most commonly used in systems like operating systems and compilers. It is considered a low level language since all built in operations can be implemented in hardware [6]. The primitive types in C are char, int, float and pointers. The only difference between pointers and integers are their type labels and in earlier versions of C they could be used interchangeably without a cast and still not produce warnings or errors[6]. A cast is the way a programmer signals to the computer to treat a variable as a different type than it is declared. In GCC, a commonly used C compiler, a cast is only required when converting from a type stored in a larger space into a type stored in a smaller space. If the types are the same size or the type being converted was smaller than the type it was converted to, the compiler would only produce a warning.

C requires manual memory management. There are two types of memory in a program, stack and heap. Variables on the stack have to be of the type of integer, float, character, pointer, struct, array, or union. These variables do not need manual memory allocated to store values or the use of memory addresses to look up their values. The other type of memory is heap memory. Heap memory can only be used after it has been allocated. To allocate memory a program requests a certain amount of memory from the operating system and the operating system returns the memory address where the block begins. The memory address is stored in a pointer. Using heap memory can produce errors that stack memory does not have. If heap memory is not allocated but used, either a bad value is read from a random memory address or the requested memory address is outside the program's memory and a segmentation fault occurs, halting the program. Another problem that can occur with heap memory is trying to access freed memory. This usually results in a segmentation fault too.

Although flexible, C allows problematic code to be written and run. The compiler does not restrict programmers from using pointer arithmetic. This allows flexibility of code but can easily become wrong code that will only cause errors during run-time. This is a common instance of C allowing implicit conversion of types. Errors occur when doing pointer arithmetic when a memory address being accessed is not allocated or has been freed.

1.2   Device Drivers

The operating system on a computer acts as an interface between software and hardware. Operating systems are made up of different modules which have different tasks, ones which manage memory usage, schedule software to run on the processor, communicate with input and output devices such as the keyboard, mouse, speakers and monitor, and communicate with hardware such as graphics cards, network interfaces, USB ports and memory. Part of an operating system are modules called device drivers. Device drivers implement the protocols that let the operating system communicate with peripherals and hardware.

Since operating systems control the functioning of all hardware and software it is important that they work well. Unfortunately, this usually is not the case. Operating systems are currently unsafe. This is due to the low level programming that goes into developing an operating system. There have been many studies looking into what areas of the operating system are producing the most bugs. These studies have shown that device drivers make up to 70% of the code in an operating system and contain up to seven times the amount of errors compared to other parts of the kernel[2][3]. One cause for these errors are confusing protocols between the driver and the devices and operating system[4].

Device drivers are written in C because C allows direct manipulation of memory and access to hardware. The language C is weakly typed meaning it puts few constraints on how values are manipulated. This has led to many errors in C code. C, due to its weak typing, does not have automated checks for NULL pointers and freed memory. These checks are purposely left out to minimize the run-time. The language does not initialize allocated memory as another way to save time. This leaves "garbage" in memory being used which can result in unpredictable code. Overall C is good for flexible memory access but compiles successfully on code with the potential to have many errors.

Device drivers are written in C but do not take full advantage of the language. Instead device drivers use a subset of the C language. Common device drivers take advantage of `int`, `char`, `struct`, `union`, `array`, and pointer types. They also use selection, loop, goto, and function control structures. They commonly use both stack and heap memory. One part of C that device drivers do not use is float and double types.

1.3   The Clay Programming Language

The Clay[9, 7, 8, 10, 11] is a low level programming language for writing operating system code. The Clay compiler does mostly static error checking and adds in necessary run-time checks to ensure safety in the resulting code. Unlike C, Clay is type-safe. A type-safe language has strict definitions of types and enforces their use. This means that the operations that can only be used for the types for which they were designated to be used on. For example, Clay would not allow a `char` typed variable to be divided or added. Clay is able to verify code statically by tracking the values of local and heap memory. It is also able to track logical values unavailable at compile-time. Run-time checks are necessary if user or hardware input cannot be verified to be safe. The Clay compiler uses proofs to verify that all memory and types are used safely, ensuring that Clay code that has compiled successfully is guaranteed to run without encountering many errors found in C code.

Clay uses singleton types to track variables. A singleton type is an immutable type which only represents one value. For example, `Int[5]` is a singleton type whose type is the set of all integers whose value is five. Integers are not the only type that have singletons in Clay. Pointers are treated like singletons too. A pointer's type tracks its location in memory. Since the pointers' addresses are unknown before run-time, Clay uses logical values instead of exact values to track pointers. When a variable's value changes, the variable is redeclared with its new singleton type.

Singleton types are then used with arithmetic constraints to track values when the exact value is unknown but is constrained such as `Int[N]` where `N < 5`. In Clay, operations, like addition and subtraction, can be applied to types, so as variables are being modified their types are too. The compiler is able to track a variable's value by tracking its type through the program. Its types are not restricted to a certain value or all integers, instead the compiler has a way to represent a variables value as a type that is the combination of types of other variables. A variable `x` can be defined as `let x = y + z` which would give it the type `Int[Y + Z]` instead of just the set of all integers whose value is five or all integers. Arithmetic constraints restrict the type of a variable but not to the extent that singleton types restrict them. Arithmetic constraints also allow the programmer to track and constrain function inputs and outputs. For example, a function sum takes `x` and `y` and returns `x+y`. The type of `x` and `y` are `Int[X]` and `Int[Y]`, respectively. The return type of the sum function is then `Int[X+Y]`. A more complicated function might return a range of values which can be specified in its return type as a number between one and ten.

Clay uses linear types on pointers to prevent aliasing errors. Linear types are tracked through the program like all other types. If a linear is dereferenced, the old value is invalidated and can no longer be accessed. The linear must then be written to become valid again. This ensures that only one variable can access head memory at any point in the program. Clay supports concurrency and uses locks to allow shared memory access.

The Clay language uses its advanced typing to track variable values during compilation. The types are then constrained on function calls to prevent bad input or output to the function. For example, a function that calculates a factorial should not have negative numbers. If a negative number was passed to a factorial function in Clay then there would be a compilation error. This functionality is especially important in device drivers where input and output functions are not always clear. Clay can prevent errors by limiting the parameters and return types, and therefore values.

1.4   Clay vs. C in Device Drivers

The operating system is the most important program running on a computer because it underlies and supports all of the other programs running on the system. It is almost always written in the programming languages C and C++ which allow low-level memory access. Operating systems without sufficient checks on memory access are very likely to have errors. Using a safer language is one solution to reduce errors. The programming language Clay fits this purpose. It allows the same low level memory access as C but checks types and memory access statically. The benefit of using Clay over C in device drivers is Clay's error checking capabilities.

Laddie[12, 13] is an extension to Clay. It is a language to write documentation for device driver input/output protocol that is both comprehensible to humans and computers. The documentation written in Laddie can be compiled into Clay functions with the appropriate restrictions on parameters and returns. With Laddie, a Clay program can identify if the input or output to a device is incorrect. Currently C has no such extension and if it did would not be able to identify at compile time if the parameter being passed was incorrect. C would only be able to check at run-time and these checks would be optional. Run-time checks that could prevent input and output errors are left out commonly both because they are optional and to make the code run faster.

The Clay programming language has more extensive static-time error checks than the C programming language which makes Clay code safer to run. With the use of singleton types, arithmetic constraints and linear types, Clay is able to track variables' values statically and verify the code will not fail on errors it is capable of tracking. Translating from C to Clay is difficult because of the information Clay requires to compile. Since C is weakly typed and Clay is strongly typed, the C code does not have all the information needed for Clay to compile. Clay's type system allows it to report more bugs than C but also makes it hard to write in or translate from C.

1.5   The Hoist Project

Hoist is the name of the project to semi-automatically translate existing device C code to Clay in order to detect errors statically. Clay is able to detect many errors not caught by C and then translate the drivers back to C where they can continue working with the operating system. Device drivers verified safe by Clay are significantly more reliable than device drivers written only in C. My thesis focuses on part of this translation. The Hoist project is broken into several parts, one of which is the translation of control structures. This thesis describes how C control structures can be translated into control structures found in Clay.

The Hoist project builds on previous work which produced the type-safe language Clay. Clay is a C like language where additional information is carried in the type system. This information is used to statically check that code in Clay has a large set of safety properties. The static checking allows us to find errors before the driver is used on an operating system. C is unable to check these properties statically so they are checked while the driver is running or not at all. Clay's built in support for static error checking and the large range of errors it can detect make it a useful tool in device driver improvement. One reason device drivers tend to have errors is the programmer misunderstanding the protocols for the device. A previous portion of the Hoist project, Laddie, is a documentation tool for writing device driver input and output protocols. Normally documentation is written in book format for programmers to read. Laddie provides a format that is similar to the standard format found in documentation but is written in such a way that the computer understands it. Laddie compiler can use it to auto generate Clay functions with appropriate constraints. Laddie is another step to ensure that input and output between the driver and the device are correct.

The Hoist project takes advantage of the Clay and Laddie languages and already written device drivers. The project's goal is to use current device drivers in C and reuse the code in Clay. The input/output protocols for the device can then be written in Laddie. The Clay produced from the Hoist compiler and from the Laddie documentation then works together to ensure a reliable device driver. Manual translation from C to Clay is slow and can result in misunderstanding of situation leading to translation errors. By trying to automate the translation from C to Clay we can avoid translation errors without reverting to recreating device drivers from scratch.

1.6 Thesis Statement

My thesis explores which control structures found in device drivers written in C can automatically be translated to Clay. The scope of this project only includes primitive types on the stack used in device drivers which are `ints` and `chars`. Complex types like `structs`, `unions`, and `arrays` will be covered in a future project along with heap memory. The control structures I am working with can be split into the following categories; assignments, selections, loops, function calls, and `goto`. Assignments are the storage of values into variables. All of the variable values used in device drivers are variants of type `int`. This project focuses on stack memory whose types in C are just characters, floats, and integers. Device drivers do not use floats or characters in their code so there is no reason for translating them. Assignments can be directly and safely translated from C to Clay without the assistance of a programmer. Selections branch through different parts of the program. An if statement and switch statement are both examples of selections. Selections should be able to be translated automatically from C to Clay in most cases. Loops are used to run through parts of a program multiple times. The loops in C are for, while and do while. Basic loops which do not update variables in their body can be automatically translated. More complicated loops will be covered in a future project. Functions are another way programmers use control the flow of their programs. A function is usually code that will be used multiple times in a program but with different values. By moving code into a function, code can be reused without being rewritten. Functions can be automatically translated. The last control structure I looked at was goto. The goto keyword interrupts normal execution and moves the program counter to a label in code. The goto keyword cannot be translated to Clay automatically because the label can be any place in code and does not obey standard control flow rules. The usage of goto does not allow value tracking and thus there is no equivalent control structure in Clay. All code with `gotos` can be written in other ways which do not require `goto` but such translations

are beyond the scope of this project.

Compiler

1.7   Overview

A compiler is a program used to translate one language to another. Common compilers translate languages like C into byte code (or machine language). Compilers are complex systems which are broken down into parts. First a program is tokenized to label all tokens in code. The ordered tokens are then passed through the parser. The parser then orders the tokens into a hierarchy which allows a parse or syntax tree to be built easily. Translation then occurs by reading and modifying the tree. The new code is produced by walking the tree and printing the necessary parts in the new syntax.

1.8   Tokenizer

The tokenizer recognizes groups of characters and labels them, organizing them into labeled strings. The computer uses these labels to understand each string. I used a program called Flex [Flex] to create a tokenizer from a special Java file. The program Flex uses regular expressions to recognize keywords, variables, strings, numbers, and white spaces. My tokenizer identifies all tokens found in the C language. I used a Flex file for ANSI C available at http://www.quut.com/c/ANSI-C-grammar-l-1998.html#check-type [16]. This file was downloaded and modified since ANSI C and GNU GCC recognized C are different. The regular expressions and tokens used can be found in the appendix.

1.9 Parser

The parser uses the tokens generated by the tokenizer and structures the code using a grammar. A grammar defines the structures of the language by recognizing certain sequences in the tokens and defining these subgroupings of tokens as different structures. The parser uses the definitions of the structures from the grammar to create a tree. The program YACC [?] generates a parser from a grammar. I used YACC to create my parser using a C grammar[16]. Like the Flex file, the C grammar was originally for ANSI C. I modified the grammar to recognize GNU GCC C which can be found in the appendix.

1.10 Tree

In computer science, a tree is a common data structure. A tree has a root element which has branches and leaves. Each branch also can have branches and leaves. This data structure works well to store code in a way that is easy to modify. The trees in compilers can either be a syntax tree or a parse tree. A parse tree represents all of the structures found in the grammar. The root node for the tree is the program with branches for variable and function declarations and so on and so forth. This method can produce a very large tree due to the grammar's structure which uses the grammar's hierarchy to enforce order of operations. Another way to store a program to be modified is a syntax tree. A syntax tree is an abbreviated parse tree which does not store intermediate structures. A syntax tree is therefore easier to understand and work with and will be how I represent control structures in this paper. The compiler I have created uses a hybrid tree. It has not been reduced completely to a syntax tree but does not contain all the structures of the grammar.

1.11 Symbol Table

The symbol table is either built with the tree or while walking the tree. It stores information about variables and functions that can be looked up at any time. Symbol tables store the name, type and scope of all the variables and functions in a program. They can also store information like whether or not a variable has been declared or initiated at a point in the code. The compiler I have created has an extensive symbol

table. It stores not just the name and type but also whether or not a variable has been assigned a value yet or has been declared. This was necessary to track when the variable was declared in C and when it was assigned a variable. The variable should not be used before it was declared and assigned. The symbol table in this compiler uses a hash table to store the variables by using the variable name as the key.

1.12   Static Error Checking

As code goes through the tokenizer and the parser, some errors in the code can be recognized. The tokenizer recognizes lexical errors. These errors are recognized when the tokenizer finds a string of characters that do not match any of the regular expressions. This could occur, for example, if a character not recognized by the language is in an input file. The parser recognizes syntactic errors. A syntactic error occurs when a group of tokens is in an order not recognized by the grammar. Syntactic errors occur when the code's structure is incorrect. A good example of this is a missing semicolon at the end of a line. Most compilers also include semantic checking. Semantic checking verifies that the code makes sense. I did not include semantic error checking in this compiler. This is because translations occur after a programmer already has a working C device driver. Therefore any checking of the C code is unnecessary.

1.13   Tree Manipulation

When code is stored in a tree it is optimal to modify it. In most compilers tree manipulation is done to optimize the output code. In this compiler, I am using tree manipulation to translate C code into Clay code by transforming the C tree into a Clay tree. This paper shows the syntax trees from the C structure and their translations to syntax trees in Clay structure. I wrote code that recognizes and modifies different cases of translations. Since a lot of these control structures are nested into other control structures it makes more sense to do a little translation at a time. Some compilers, like C, are single pass compilers that will only go through the tree once before producing the output code. My compiler is like other compilers that go through the tree multiple times, modifying or gathering data a little at a time. Each pass on the tree translates more of it until it is finally all translated.

1.14   Code Output

The result of a compiler is code in a different language, often machine code or assembly. In the case of this compiler, the output language is Clay. The output is written to a file called out.clay. The code is produced by walking the tree and writing Clay code to represent the appropriate keywords and tokens using the control flow expressed by the tree. This translation is semi-automatic due to missing information in C and will not produce compile ready code most of the time. Future work will translate heap memory usages and complex data types allowing the compiler to translate more C to Clay on its own.

Control Structures

A computer goes through a program sequentially. Control structures interrupt the sequence and cause the code to skip code sections, run portions of the code repeatedly, decide between different sections of code to run and so it can be run from multiple points in the program without having to write the same code in multiple places. Most languages have similar control structures but the syntax differs. This section will introduce control structures found in C and how they can be translated to Clay.

1.15   Assignments

An assignment is when a variable is given a value to represent. It consists of the variable being assigned and an expression that will evaluate to its new value. In C there are two places that an assignment can happen. First, a variable can be assigned when it is declared. The other case is in an assignment expression. Clay combines them so all assignments in Clay are also declarations.

The difference between a C assignment and a Clay assignment is that Clay uses the keyword "`let`" which declares the variable and implicitly figures out its Singleton type when making the assignment. In this paper, Clay assignments will be referred to as let assignments. Only stack variable assignments and integer types are in the scope of this project. Heap variables and complex types will be included in the next stage of the Hoist project. Additionally, Clay only supports integer types so only the C types int, long, and short are translated.

There is one more case that must be dealt with when translating assignments. Clay uses Singleton types to track values of variables as they pass through control structures. To allow the Clay compiler to track values through function calls, the

C	Clay
`int x;`	
`int x = 0;`	`let x = 0;`
`int x = 0,y;`	`let x = 0;`
`int x = 0, y = x;`	`let x = 0;`
	`let y = x;`
`int x = y = z;`	`let y = z;`
	`let x = y;`
`int x = foo() + bar();`	`let temp1 = foo();`
	`let temp2 = bar();`
	`let x = temp1 + temp2;`
`int x = foo(bar());`	`let temp1 = bar();`
	`let x = foo(temp1);`

Figure 1.1: Declaration Translations from C to Clay

calls must be alone in an expression. Assignments in C which have function calls within function calls or arithmetic with the return of a function must be broken up into separate let assignments and temporary variables can be used to hold the intermediate results.

1.15.1   Declarations and Declaration Assignments

In the C programming language, variables are declared with their type before they can be used. A declaration can also include an initial value for that variable but it is not required. When a C declaration has no assignment it is not used in Clay and is removed from the syntax tree. All variables declared in C with an initial value need to be translated into let assignments where the type of declaration is implicit. A chart of declaration translations from C to Clay can be found in figure 1.1.

Figure 1.2 shows the syntax tree for an example declaration in C. A declaration consists of a type and a list of declarators. Each declarator may or may not have an initializer, the expression the variable is to be assigned. During translation the declaration is removed and let assignments are made from any declarator with an initializer. The syntax tree for a let declaration can be seen in figure 1.3.
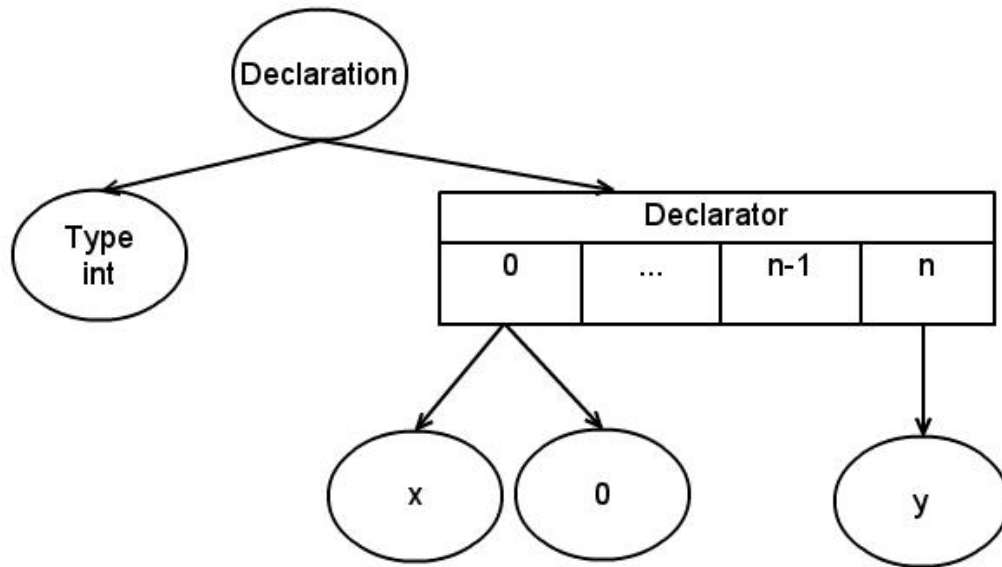
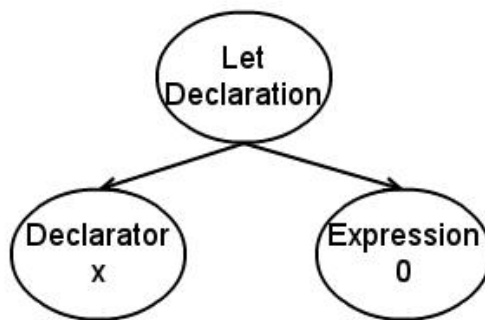**Figure 1.2:** The syntax tree for the C declaration `int x=0,y;`.

Figure 1.3: The syntax tree for the Clay let declaration `let x = 0;`.

C	Clay
x = 0;	let x = 0;
x = foo();	let x = foo();
x += 4;	let x = x + 4;
x = y = z;	let y = z; let x = y;

x = foo() + bar();	let temp1 = foo();
	let temp2 = bar();
	let x = temp1 + temp2;

x = foo(bar());	let temp1 = bar(); let x = foo(temp1);

Figure 1.4: Assignment Translations from C to Clay

1.15.2 Assignment Expressions

In C, a variable is assigned a value in an assignment expression. Translating a C assignment to a Clay let assignment simply involves changing the assignment to a "let" when the assignment operator is '=' and the right side of the assignment is one simple expression. The C language supports many other assignment operators like '+=', '-=', '*=', and '/='. The Clay language only supports the '=' assignment operator. In the case that these other assignment operators are used, the statement needs to be translated to equivalent C using the '=' operator before it can be translated to Clay. Assignments with complex expressions on the right side need to be split into separate parts using temporary variables before they are translated to Clay. Examples of assignment expression translations from C to Clay can be found in figure 1.4.

Figures 1.5 and 1.6 show the syntax tree for an example assignment expression and the syntax tree for the Clay let expression it was translated to, respectively. The syntax trees show the translation of an assignment using the '+=' operator. These translations show that stack memory integer-type variable assignments can safely be translated from C to Clay without supplemental information.
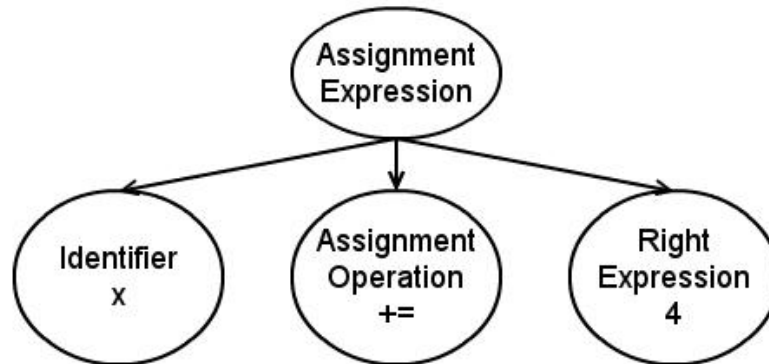
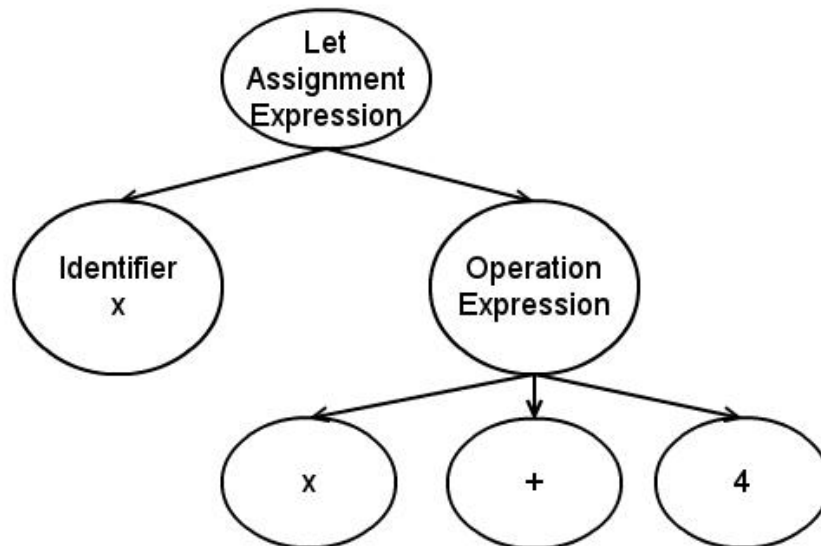**Figure 1.5:** The syntax tree for the C assignment `x += 4;`.

Figure 1.6: The syntax tree for the Clay let assignment expression `let x = x + 4;`.

1.16 Functions

A function in computer science is a type of control structure. A function in C is defined with a return type, a list of parameters each with a specific type, and a body. Parameters are the variables being passed into the function and used to produce a new value or to change values of global variables. The return type of a function is the type of the variable being returned. Code in a function's body can be called repeatedly and applied to different parameters which can help produce more readable and less repetitive code.

The Clay function can be very different from a C function. The return type and the type of the parameters can be constrained and functions can return tuples allowing multiple variables to be returned. That said, the basic form for a function in Clay is the same for C as long as the types of the function are Clay stack memory and are one of the integer types. The only translation needed for functions is to identify the return type and parameter types. If these types are not either void or integers (short, long, int) then the function is not translated and marked for a programmer or for the next part of the project to translate. For the function to be completely translate the code in each function needs to be translated using the other translation in this chapter.

In order to use the tracking capabilities in Clay, the types for a function should be updated by a programmer. In this project the compiler changes integer types to singleton types with unknown values. In this way a return type int will become `Int[K]`, representing an integer with value K and the parameters would each have their own unique type such as `Int[M]` and `Int[N]`. The necessary information to further constrain a function's parameters and return type cannot be found in C. The compiler leaves a special comment for the programmer telling them to update the function's types if possible.

1.17 Selection

Selections are control structures that choose between different sections of code to run. They decide on a branch depending on an expression known as the test. The selections found in C are if, if-else and switch statements. C also has a conditional expression which returns a value depending on its test and has the syntax `test ?  value1 :`

C	Clay
`if(x){}`	`if(x != 0){}`
`if(x > 5){}`	`if(x > 5){}`
`if((h = 5) > 4){}`	`let h = 5;`
	`if(h > 4){}`
`if(foo() > x){}`	`let temp1 = foo();`
	`if(temp1 > x){}`
`if(foo()){}`	`let temp1 = foo();`
	`if(temp1 != 0){}`

Figure 1.7: If-Test Translations from C to Clay

`value2;`. This conditional expression will be covered in future work. Clay also has an if-statement with an optional else statement but does not have a switch statement or conditional expression.

1.17.1 If and If-Else

There are two parts of an if-statement. The first part is a test to determine if the body of the if-statement will be executed. The second part is the body. The body of an if-statement is the code that is executed if the test evaluates to true, or in the case of C, not zero. An else-statement is used with conjunction of an if-statement. An if-statement must come before an else-statement. The else-statement has no test but does have a body. The code in the else-statement's body will execute if the if-statement's test evaluates to false or zero. All if-statements do not need an else but all else-statements must have an if-statement.

Test Translation

The first part of translating an if-statement is to translate the test. All tests need to have a variable or number being compared to another variable or number. There cannot be function calls or assignments in the test; these must be moved outside of the test and stored in a temporary variable to be compared. Clay also requires a relationship operator like `>` or `==`. Any test in C without a relationship operator has to be compared to 0. Examples of if-tests being translated can be found in **??**. The syntax tree for a C if-statement is generally the same as the syntax tree for a Clay
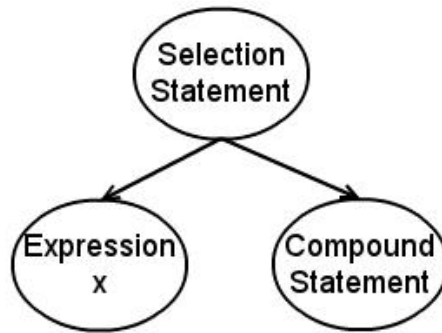
Figure 1.8: The syntax tree for the C If-Statement `if(x){}`.

if-statement. The only difference is what will be in the test. A C if-statement can have any expression in the test. A Clay if-statement must have a relational operator expression and all expressions below this point must not contain function calls. Figure 1.8 shows a C if-statement with a test expression "x" and an empty body. Since a variable is not a relational operator expression it must be modified before translation. The translation of this if-statement can be seen in figure 1.9. To make the expression into a relational operator expression the variable must be compared to 0. This is the case because C does not have a boolean type. Instead 0 means false in the test and all other numbers are true. Therefore if x is not equal to 0 then the test is true.

This example does not show the translation if a sub expression of the operational expression is a function call. In that case a let assignment, like the one shown in the section above, assigns the function call to a temporary variable. The function call is then replaced with the temporary variable in the test expression. The let statement is then put before the if-statement. Since the test expression could have any number of function calls the newly created let assignment might need to be translated to multiple let assignments.

Body Translation

Translating the body of an if-statement is more complicated than the test translation because of scope. Scope is the range of the program that a variable is relevant and is in most cases defined by blocks of code or new structures. The whole program has a global scope while each function then has its own scope. The selections, loops, and functions all begin new scopes. Every variable belongs to a certain scope. A variable
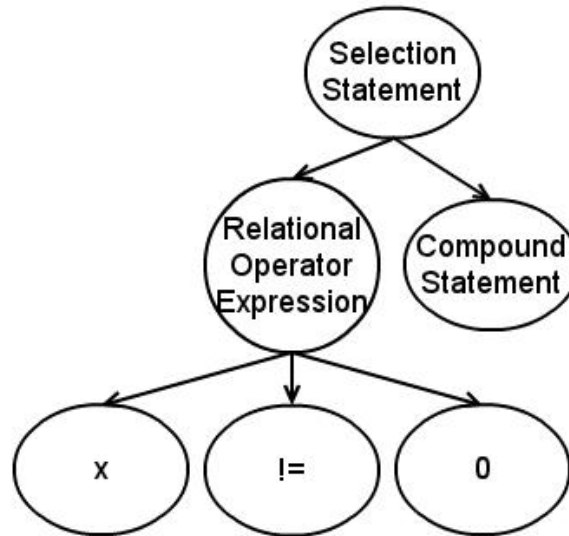
Figure 1.9: The syntax tree for the Clay If-Statement translated from C `if(x != 0){}`.

can be used in its own scope or a sub-scope. For example, if a variable is defined in a function, it can then be used in the body of an if-statement in that function. The if-statement would be defined in the function's scope and therefore have access to the variables defined in that function. Since a variable belongs to the scope it was declared in some problems arise because Clay does not declare variables until they are assigned values. Also every time a variable changes value in Clay, it is redeclared to reflect the change in Singleton type. Therefore if a variable changes value inside of an if-body its scope will only be in the body of the if-statement.

Since variables that change value inside of the if-body or else-body will not change value outside of that scope the if-statement must be translated to a function. The values being modified in the if-statement's body can be returned and stored in the variable in the appropriate scope. For this translation, first the compiler must recognize that a variable which has already been declared, or would have been declared in C, is being assigned a value. Multiple variables can be assigned in the if-statement and the compiler must keep track of all of them and know their prior values, if they have one. All of this information is stored in the symbol table of this compiler.

If a variable that has been declared outside of the if-body is being assigned then a new function is generated with a unique name. This function will be placed before

```
void main(){

    int x = 5;
    int y;

    if(x!=0){
        x = 3;
        y = 2;
    }
}
```

Figure 1.10: Example in C of an If Statement with multiple externally declared variables being assigned

the current function. The parameter of this function will be the if-test and its return type will be the type of the variable being updated. If more than one variable changes value in the if-statement, more than one variable will need to be returned. In C this would not be possible without defining a structure or an array but Clay has a tuple type. A tuple can store any number of variables of different types and is perfect for returning multiple variables in a function. If statements that are moved into new functions need an else statement even if the original if-statement did not have one. The function must return a value regardless of whether the test evaluates to true or false. An else-statement that is added must return the original value of the variable. If the variable has not been assigned a value yet it will be given the value zero and a comment will be made in the code. This initializes a variable to a value that the original programmer did not assign it but is required because all variables have values in Clay. The Clay compiler will no longer be able to identify if this variable is being used before it has properly been assigned.

Figure 1.10 is an example if-statement which assigns values to two externally declared variables. The variable x has been previously initialized while y has not. The translation of this if-statement can be seen in figure 1.11. This example shows how a tuple can be used to return multiple variables from a function. Figure 1.12 shows the syntax tree for a slightly simpler if-statement where only one variable is being assigned. The syntax tree for the function that the if-statement is moved to can be seen in figure 1.13. Figure 1.14 shows the syntax tree for the let assignment expression that replaces the if-statement in the original function.

```
.[int, int]
/*~~ Return type inspecific. Should possibly be .[Int[4],Int[6]]
    if we know the returns 4 and 6 or
      exists [u32 J, u32 K; J > 4 && K < 6] .[Int[J],Int[K]]
    if we only know some constraints on the return type or
    .[int,int] if we have no idea. ~~*/
function1(int
/*~~ Return type inspecific. Should possibly be .[Int[4],Int[6]]
    if we know the returns 4 and 6 or
        exists [u32 J, u32 K; J > 4 && K < 6] .[Int[J],Int[K]]
    if we only know some constraints on the return type or
    .[int,int] if we have no idea. ~~*/
x){
    if (x != 0){
        return .(3,2);
    }else{
        return .(5,0);
    }
}

void clayMain(){
    let x = 5;
    let
    /*~~[] if this function returns an existential ~~*/
    (x,y) = function1(x);
}
```

Figure 1.11: The translation to Clay of the If-Statement in 1.10.
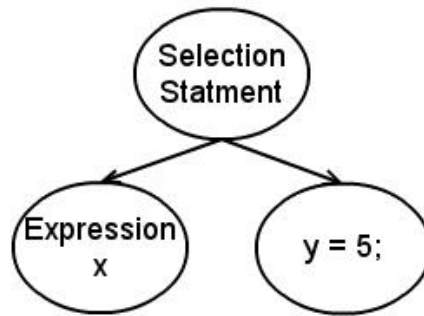
Figure 1.12: The syntax tree for a C If Statement assigning a new value to a variable that has already been declared.
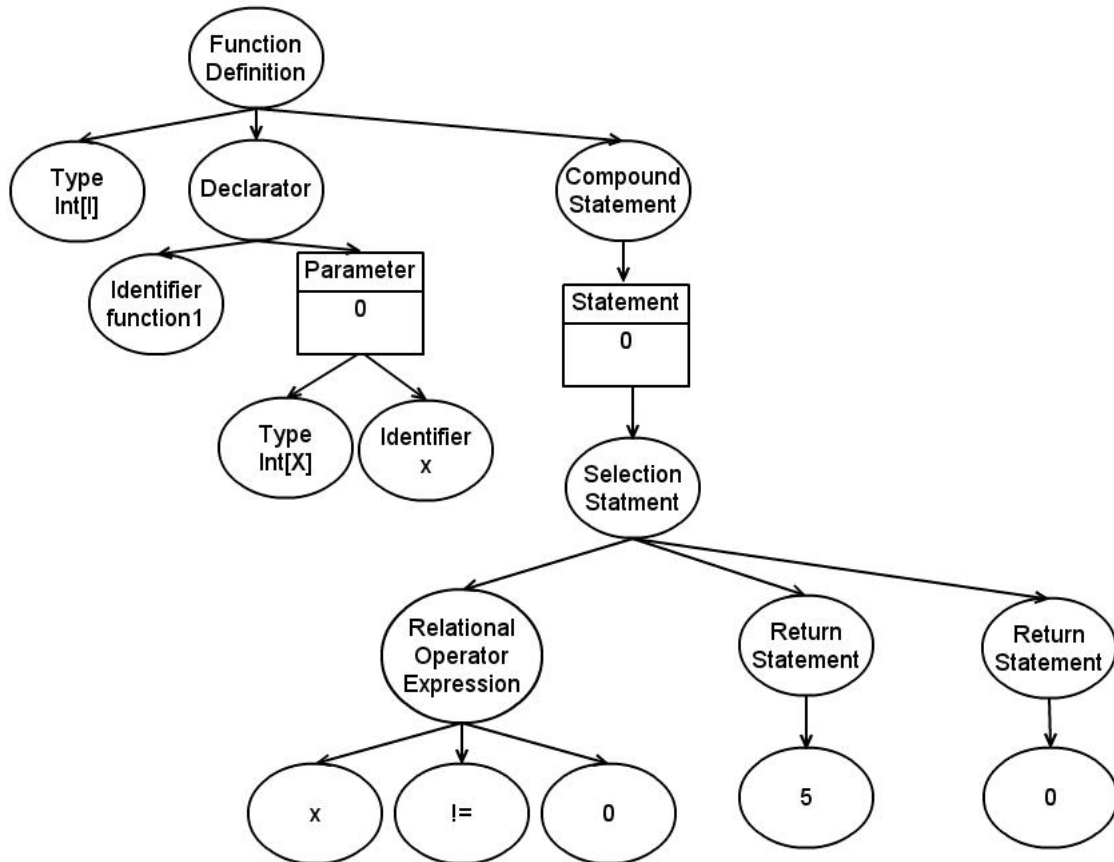

Figure 1.13: The syntax tree for the function translated from the If-Statement in figure 1.12.
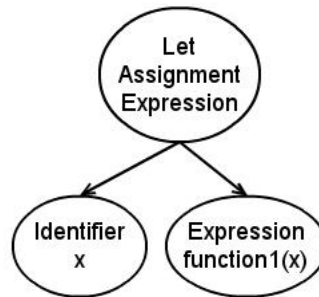
Figure 1.14: The syntax tree for the Clay function call replacing the If-Statement.

1.17.2   Switch

The switch statement is used to select between different values on a test variable. A switch works by comparing the variable given in the switch test to the constant values in each case. The case statements have blocks of code that will execute if the variable being switched is equivalent to that case. If none of the previous case statements are true then the default block is run. If a case block does not end in a break statement then the execution "falls through" to the next case's code block.

There is no switch statement in Clay but the switch statement in C can be translated into if-statements and else-statements. Translating switch statements into Clay is a two step process. First the switch statement is translated to if-else statements in C. The if-else statements then go through the translation to Clay. There are different ways to translate C switch statements to C if-else statements. If each case ended in a break then it could be translate into nested if-statements with the if-statement's test comparing the switch variable to the case constant. The first case would be the first if-statement with the code in the first case block going into the body of the if-statement. The next case would then be an if-statement in the first if-statement's else-statement with its case-body becoming the if-body. If there is a default case then its body would go in the last if-statements accompanying else. Switch statements where not all of the cases end in breaks are almost as simple to translate. These are easiest to translate backwards. If a case, excluding the default case, does not end in a break then the if-statement's body of that case is its own body and the body of the case below it. By doing this translation backwards then you do not have to check if the case body being copied is also missing a break since if the case below does not have a break then the appropriate case-body below that will already be copied. An example of a translation from a C switch statement into C if-else-statements can be seen in

```
switch(x){
    case 1: printf("one"); break;
    case 2: printf("two");
    case 4: printf("three"); break;
    default: printf("other"); break;
}
```

Figure 1.15: This is an example of a C switch statement.

```
if(x == 1){
  printf("one");
}else if(x == 2){
  printf("two");
  printf("three");
}else if(x == 3){
  printf("three");
}else{
  printf("other");
}
```

Figure 1.16: Translation of the C switch statement in figure 1.15 to a C If-Statements.

figures 1.15 and 1.16.

Figure 1.17 shows the syntax tree for a simple switch statement with two cases, where x is 1 and where x is 3, and a `default` case. The first case does not end in a break while the second case does end in a break. In this example if x's value is 1 then the program will print out ''1" and ''3". If x's value is 3 then the program will print out ''3". Last if x is neither 1 or 3 the program will print out the "other". Figure 1.18 shows the syntax tree of C if-else statements which are translated from the previous switch statement. Both syntax trees represent C code which will produce the same results.
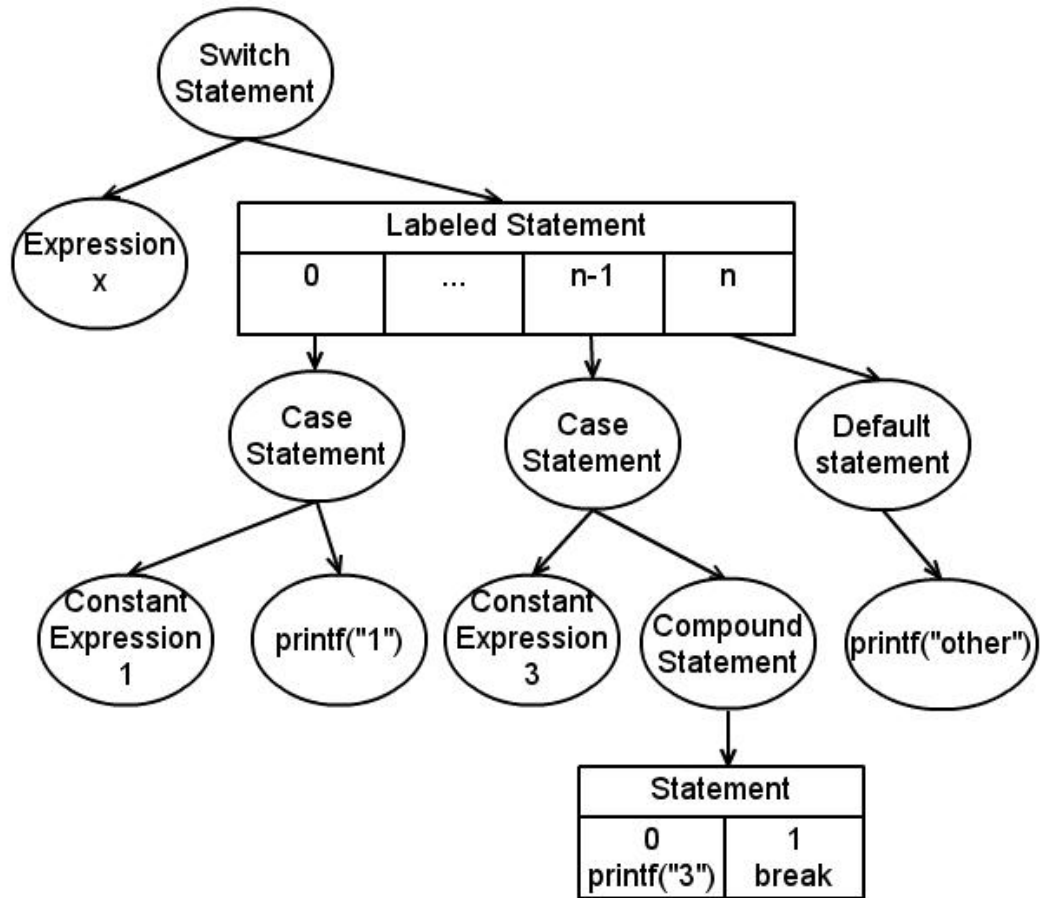
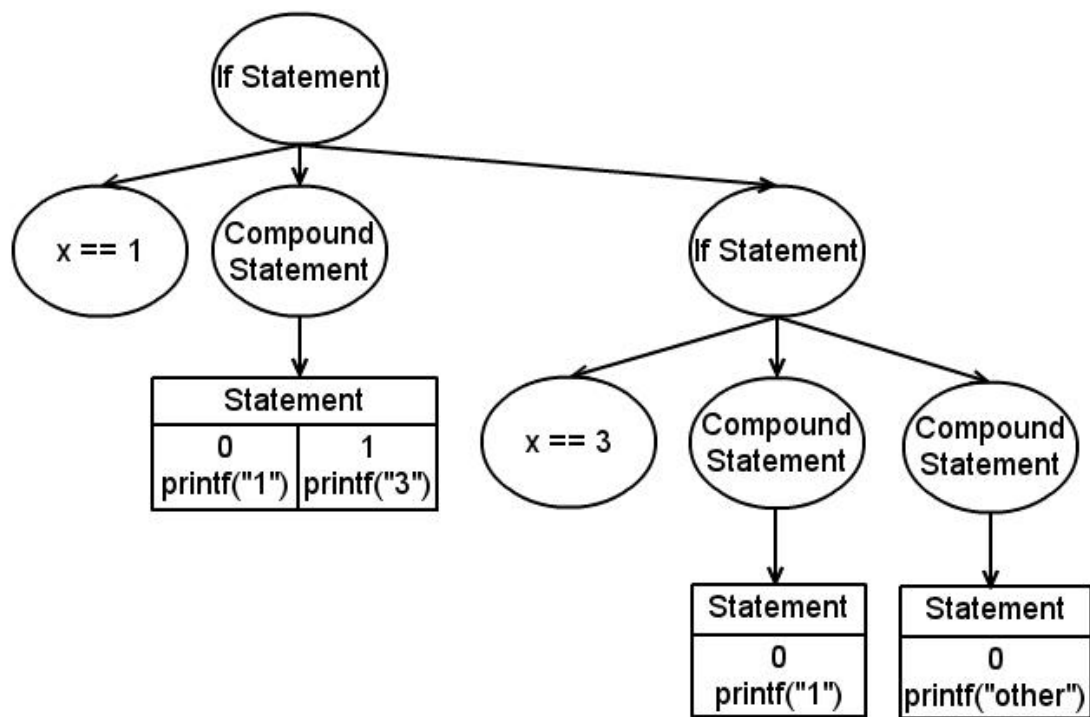**Figure 1.17:** The syntax tree for a C Switch Statement

Figure 1.18: The syntax tree for a C If Statement translated from a Switch Statement

1.18   Loops

Loops are used to repeat code until a certain condition is met. In C there are three types of loops; for, while, and do while. The basic structure for all three is similar. There is a body which is a block of code that is to be repeated. Then there is the test which if it evaluates to true will cause the body to be executed again.

In Clay there is only one type of loop, the for-loop. First while and do-while loops need to be translated into C for-loops which can then be translated into Clay for-loops. The test and bodies of the loop structures are restricted the same way as the selection bodies and tests. Tests need to be simple relation expressions with no function calls or assignments. These need to be moved out of the test and into let statements with a temporary variable and put both before the loop and inside the body. Since the loop's body is a different scope and, like the test's body, it needs to be checked for assignments of variables declared outside the scope of the loop. Loops which bodies assign new values to variables previously declared need to be put in functions with any assignments as the return value. The loop is then replaced with an assignment and a function call.

1.18.1   For-Loop

C also has a for-loop structure. The C for-loop has two parts, the parenthesis and the body. The parenthesis in the for-loop has three parts which, in a standard for-loop, will all be present. First is the initialization which assigns an initial value to the variable being looped through. The second part is the test of the loop which acts the same as the while test. Last is the update which updates the loop variable. The initialization will only happen once before the loop. The test will then occur and if evaluated true the body will be executed and if evaluated false then the loop body will be skipped. The test will then be evaluated again after each iteration of the loop. The update is executed at the end of each loop iteration before the test. Although initalizaion is usually an assignment, the test is usually a comparison and the update is usually an increment or decrement, these standard expressions are not required. Any expression can go in each of these three parts of the for-loop. No matter what expression is in each position the expression will still execute depending on their position in the for-parenthesis.

The Clay for-loop is similar to the C for-loop. It has a initialize expression, a

```
for(int i = 0; i < 5; i++){
    printf("%d\n", i);
}
```

Figure 1.19: Simple For Loop in C.

```
for [s32 I] (Int[I] i = 0; i < 5){
    print_int(i);
    continue(i+1);
}
```

Figure 1.20: Translation of the simple For Loop to Clay.

test expression and an update. The initialize expression and test expression, like in the C loop, are in the parenthesis of the for-loop while the update goes at the end of the for-loop body in a `continue`. The continue "calls" the for loop again with an updated variable. In this way the for-loop in Clay is really a recursive call. The `continue`'s syntax is the continue keyword, then open parenthesis, the value that the loop variable is to be updated to, and then close parenthesis. Instead of updating the for-loop variable, the new value for the loop variable is passed to the continue expression. The only other difference between C's for-loop and Clay's for-loop is the type declaration of the loop variable. Clay declares the type of the looped variable with a singleton type which requires two type declarations.

The translation of a simple for-loop is straight forward. The singleton type is added to the loop variable from the declaration. The type of the variable needs to be generated since no two variables can have the same type in Clay. Figure 1.20 shows how the variable is declared with type `Int[I]`. The brackets with "`s32 I`" are declaring that the variable type `I` is a signed 32 bit integer. It is this "`I`" type that needs to be generated since the next for-loop (if one occurs) can not have a loop variable with type `I`. The test can be directly copied from the C for-loop test into the Clay for-loop test. If the update is an assignment then the right side of the assignment can be used for the update in the continue. Notice that after `i++` is translated to Clay it is `let i = i +1;` and the update in the continue expression in figure 1.20 is `i + 1;` If the update is not an assignment than it can be copied to the last line of the for-loop body before the continue. The updated loop variable will then be passed to the continue expression. The continue expression is necessary since the loop variable's value is only updated for the next iteration through the continue expression. The rest of the body of the for-loop will be copied over to the body of the
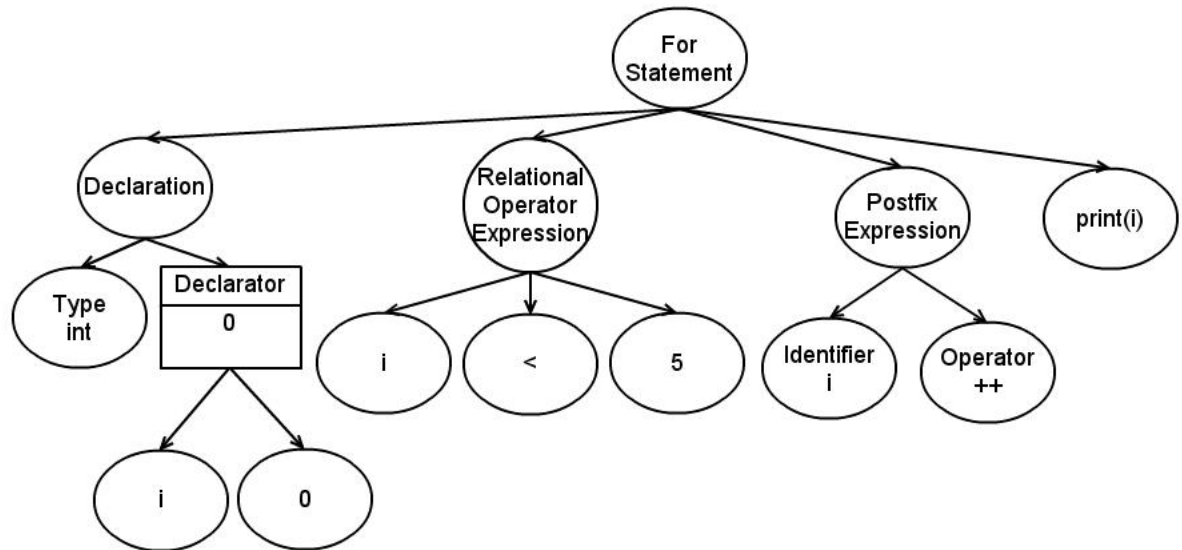
Figure 1.21: The syntax tree for a C For-Loop.

Clay for-loop. Figures 1.19 and 1.20 show the translation of a simple for-loop from C to Clay.

Figures 1.21 and 1.22 show the syntax trees for a simple for-loop in C and its translation to Clay. The differences in the two translation is the new singleton type for the loop variable and the update moved into the continue. The previous translation examples are basic for-loops but they do not take into account scope.

Like the if statement, the body of a for-loop is a new scope and runs into the same problems. If a variable that has been defined outside of the for-loop is assigned a new value then the loop is moved into a new function and returns the values of the updated variable. The return of the function has to be placed in the loop because the values are lost after the execution has left the for-loop. To do this correctly first the update has to be moved out of the continue and back into an assignment at the end of the loop body. After the update assignment an if statement is added. This if body will just contain the return statement returning the proper variables that had been updated. The if's test will need to be the opposite of the loop test since the return should happen after the loop would have stopped and the loop test is true when the loop should continue. This can be done with the boolean not-operator (!). That way when the loop test is not true the if body will be executed and the function
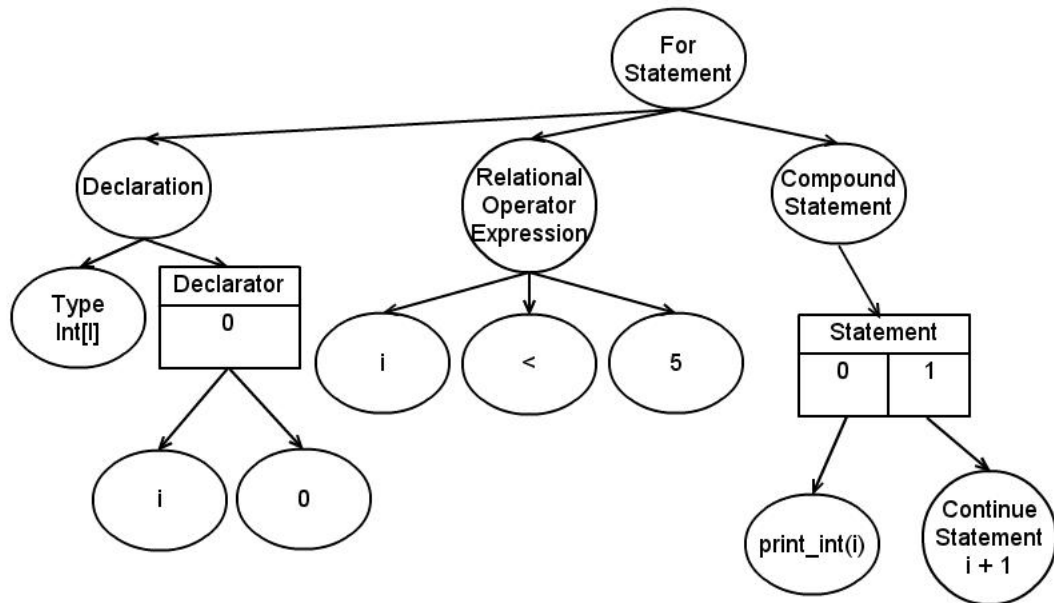
Figure 1.22: The translation of the C For-Loop into Clay.

```
void main(){
    int x = 0;
    for(int i = 0; i < 10; i++){
        x = 3;
    }
    printf("%d\n", x);
}
```

Figure 1.23: C for-loop with assignment of externally declared variable.

will return the variables as they would be after the loop. The for-loop will then be replaced with an assignement of the updated variables to the function just generated. Figures 1.23 and 1.24 show an example of a for-loop being moved into a function so that the variable can be updated.

Another complication with for-loops is that only the loop variable will be stored through each iteration. This means that any variables assigned outside of the for-loop will have their previous value at the beginning of every iteration. More complex for-loops are left for the future work.

1.18.2   While and Do While Loops

Two common types of loops are the while and do while. The only difference between a while and a do while is that a do while goes through the code in the body once before executing the while test. If the while test evaluates to true then the code in the while or do while's body is executed again. This continues until the while test evaluates false.

Clay does not have a while or do while loop. Like the switch statement, while and do while statements are translated in two steps. First it is translated into a C for-loop and then the C for-loop is translated into a Clay for-loop. The only difference between a while and a do while is when the test is preformed. Since a do while is easily translated into a while by copying the body of the loop before the loop and changing it into a while I will only explain the translation of a while into a for-loop.

A while loop can be translated very easily into a for-loop. Since the initialization and update expressions can be empty the easiest way to translate a while loop into

```
int
/*~~ Return type inspecific. Should possibly be Int[5]
    if we know the returns 5 or exists [u32 J; J > 4] Int[J]
      if we only know some constraints
    on the return type or int if we have no idea. ~~*/
function1 [u32 I] (Int[I] x){
    for [s32 I] (Int[I] i  = 0; i < 10){
        let x = 3;
        let i = i +1;
        if(!(i < 10)){
            return x;
        }
        continue(i);
    }
    return 0;
}


void clayMain(){
    let x = 0;
    let
    /*~~[] if this function returns an existential ~~*/
    x = function1(x);
    print_int(x);
}
```

Figure 1.24: Clay for-loop with assignment of externally declared variable.
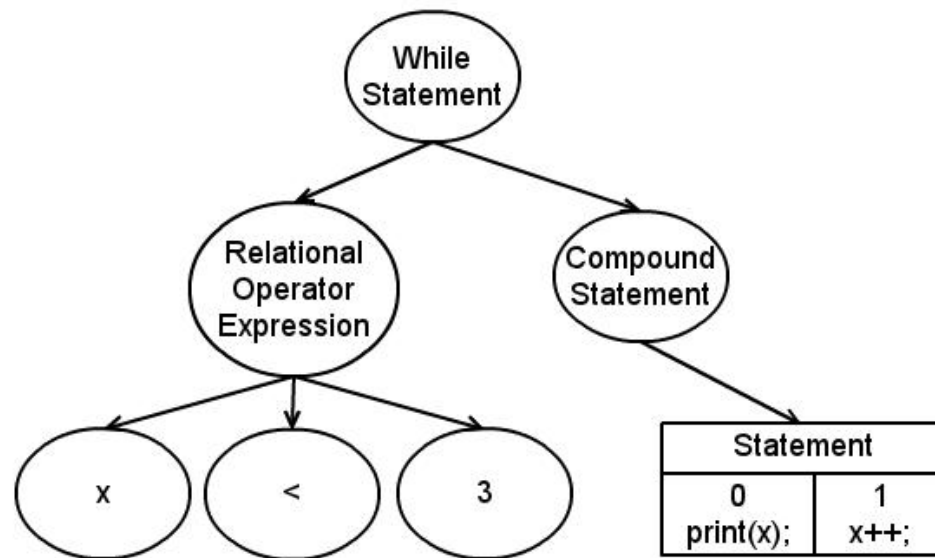
Figure 1.25: The syntax tree for a C While Statement

a C for-loop is to copy the body into the for-loop and copy the test into the second expression in the for-loop's parenthesis. The variable in the test should already be initialized and the update will already be in the loop body since these are necessary for the while to work. While loops can be translated other ways where the initialization and update are placed in the for-loop's parenthesis but these are more complicated especially in non-standard while loops.

Figure 1.25 shows the syntax tree for a basic while-loop and figure 1.26 shows this while loop translated into a for-loop. In this example the update of the loop variable could have easily been moved to the update expression in the for-loop structure but if the update of the variable had come before the print statement then moving the update to the for-loop's update would have changed the code. If the update of the loop variable happens anywhere but the end of the loop body, translating to a for-loop while moving the update becomes a lot more complicated. It is easier to keep the update inside of the loop variable and not to use the for's update.
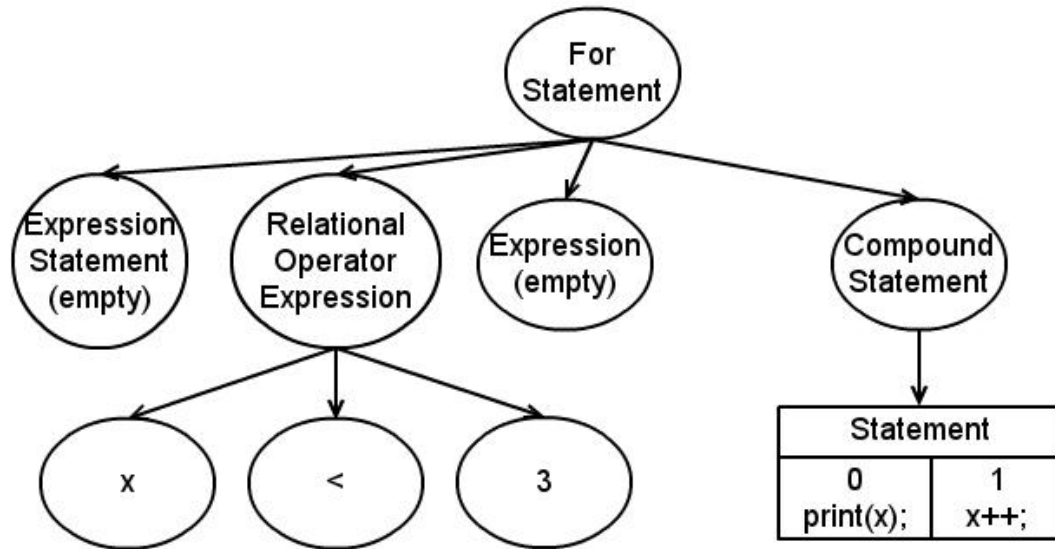
Figure 1.26: The syntax tree for a For-Loop translated from the While Statement in figure 1.25.

1.19   Goto

The goto keyword is used to move the point of execution from one place in code to another. There is no standard flow and no restrictions about where a goto can lead. It is able to do this with labels in a program. It is generally accepted to be unsafe because it is hard to track and leads to spaghetti-code. Spaghetti-code is code that acts like a noodle in a pile of spaghetti, it is hard to follow and weaved in with a lot of irrelevant code. There is no translation to Clay because it does not have a structure in Clay that is equivalent. Clay would not support any structure like a goto because it would lose its ability to track variable's values through a goto call.

Without the programmers understanding of where the goto leads to and why it was necessary the goto can not be translated to a safe control structure like the ones described above. All goto statements need to be translated by a programmer. It cannot be automated.

Problems

Throughout this project there have been problems that have slowed down my progress. The first problem that I ran into was trying to run my compiler on actual device drivers. The GNU GCC compiler allows user to define MACROs which will be substituted with the appropriate code during preprocessing. My compiler does not contain a preprocessor so the code needs to be first compiled using the GNU GCC compiler and then use the C code produced in my own compiler. This would only work in the correct libraries are linked with the GNU GCC compiler so that the compiler could look up the MACROs in the appropriate header files. Since operating systems use kernel headers along with normal C header files, I was unable to ever successfully get rid of all the MACROs defined in the device drivers which caused syntax errors in my own compiler. After working on this for a month my advisor and I decided that compiling device drivers in C would be part of another project.

Another problem that arose during my research is transforming the ANSI C grammar I found into a grammar that recognizes GNU GCC. The GNU GCC extends the ANSI C grammar. I had to add tokens to the original Flex file for such keywords like `inline`, `asm`, `typeof`, and more. These keywords then had to be added to the grammar in the appropriate places. Most keywords were simple to add but the `asm` and `attribute` keywords have none standard parameters. It is important to have these keywords since they are common in device drivers. I eventually had to write a regular expression for their parameters and was able to use this new regular expression to correctly identify the `asm` and `attribute` uses in the C code.

Conclusion

The purpose of this project is to create reliable device drivers to produce reliable operating systems. Due to its extensive error checking, Clay is a better language for device drivers than C, the language they are written in now. Clay can guarantee that common C errors are not in the code produced from a successful Clay compilation. There are a multitude of device drivers that run on an operating system. Each of these drivers is already written in the C programming language. It is much more practical to translate these existing device drivers than to write write new drivers from scratch. Translation can be time consuming and therefore expensive especially with the number of device drivers already in use. Automatic translation of C code to Clay takes less time and will not have the problem of a wrong translation because of wrong interpretations in either language. The system used to translate a programming language into a new language is a compiler. Compilers are complex systems with many steps. Do to this complexity the automatic translation of C to Clay has been broken into two parts, translating stack memory and control structures and translating heap memory.

This paper describes how stack memory and control structures can be automatically translated to Clay. Since the Clay programming language guarantees type safety and C is so loosely typed, direction translations are not always possible. Any structure that cannot be safely translate either in whole or partially will be labeled for a programmer to fix after compilation. Most likely, due to the fact that most device drivers do have errors, even code that has been completely translated automatically will not compile successfully in Clay and a programmer will need to fix the code.

While translating control structures from C to Clay I have found that most structures can be translated automatically if they only use stack memory. The only exception to this is the goto structure which is incapable of being translated into Clay. Assignments and functions can be translated automatically and safely if they only use

stack memory variables. While loops and selections can in some cases be translated safely but in cases where variables values are being updated not all structures can be updated.

Future Work

This thesis is just the beginning of the Hoist project. The end result for the Hoist project is a compiler which can translate C code to Clay code almost completely. The next step is to finish translating for-loops which are more complex than the ones described above.

Another part of the project that needs to be completed is the translations of complex types. These would include `struct`, `union`, and `array`. Pointers will also need to be translated. Pointers are the way of manipulating heap memory in C. Heap access will need to be translated from C to Clay.

Related Work

The reliability of device drivers has been seen as a problem recently. There are many projects working to improve new device drivers and to debug older drivers. One project being led by Microsoft is the SLAM[SLAM] project. The SLAM project helps developers ensure that the interfaces are being properly used between the software of the device driver and the driver hardware. It is used for developing new drivers which work well. Similarly the Devil [Devil]project is developing a language to write device interfaces. Devil fulfills the same role that Laddie does, to describe in detail the interface between the hardware and software of a device. Another language like Clay called NDL[20] was developed to write device drivers in also. This language would be used to develop new drivers which preform better. Research is also being done to debug current drivers like the SymDrive [SymDrive] project being worked on at University of Wisconsin-Madison. This project simulates the hardware so drivers can be tested without an actual device. The Hoist project combines the benefits of a new language which allows specific interfaces and static error checking with the ability to fix older drivers.

References

[1] Andy Chou, Junfeng Yang, Benjamin Chelf, Seth Hallem, and Dawson Engler, *An Empirical Study of Operating Systems Errors*, ACM, (2001).

[2] Leonid Ryzhyk, Peter Chubb, Ihor Kuz, Etienne Le Sueur, Gernot Heiser, *Automatic Device Driver Synthesis with Termite*, ACM, (October, 2009).

[3] Michael M. Swift, Mathukaruppan Annamalai, Brian N. Bershad, and Henry M. Levy, *Recovering Device Drivers*, ACM, (November, 2006).

[4] Leonid Ryzhyk, Peter Chubb, Ihor Kuz, and Gernot Heiser, *Dingo: Taming Device Drivers*, ACM, (April, 2009).

[5] Yashavant P. Kanetkar, Let Us C, (2008).

[6] Brian W. Kernighan, Deinnis M. Ritchie, The C Programming Language, (1988).

[7] Lea Wittie, Type-Safe Operating System Abstractions Dartmouth Technical Report TR2004-526, (June 2004).

[8] Lea Wittie, Clay: A Type-Safe Systems Programming Language Bucknell Computer Science Technical Report #08-1, (2008).

[9] Chris Hawblitzel, Edward Wei, Heng Huang, Eric Krupski, and Lea Wittie, *Low-Level Linear Memory Management*, (2004).

[10] Chris Hawblitzel, Heng Huang, and Lea Wittie, *Composing a Well-Typed Region*, (2004).

[11] Heng Huang, Lea Wittie, and Chris Hawblitzel, *Formal Properties of Linear Memory Types Dartmouth Technical Report TR2003-468*, (August 2003).

[12] Lea Wittie, *Laddie: The Language for Automated Device Drivers Bucknell Computer Science Technical Report #08-2*, (2008).

[13] Lea Wittie, Chris Hawblitzel, and Derrin Pierret, *Generating a Statically-Checkable Device Driver I/O Interface*, (2007).

[Flex] jflex.de.

[BYACCJ] byaccj.sourceforge.net.

[16] Jeff Lee, http://www.quut.com/c/ANSI-C-grammar-l-1998.html#check-type, (1985).

[17] Chris Bassett, Bucknell Student Project, (2009).

[SLAM] SLAM, Microsoft Research, http://research.microsoft.com/en-us/projects/slam/.

[Devil] Devil, INRIA Research Group, http://phoenix.inria.fr/software/past-projects/devil.

[20] Christopher L. Conway and Stephen A. Edwards, *NDL: A Domain-Specific Language for Device Drivers*, (2004).

[SymDrive] Matthew J. Renzelmann, Asim Kadav, and Michael M. Swift, *SymDrive: Testing Drivers Without Devices*, (2012).

Appendix

1.20 Flex

```
D             = [0-9]
L             = [a-zA-Z_]
H             = [a-fA-F0-9]
E             = [Ee][+-]?{D}+
P             = [Pp][+-]?{D}+
FS            = (f|F|l|L)
IS            = ((u|U)|(u|U)?(l|L|ll|LL)|(l|L|ll|LL)(u|U))

StringLiteral  = L?\"(\\.|[^\\\"])*\"

LineTerminator = \r|\n|\r\n
InputCharacter = [^\r\n]

Preprocess      = "#" {InputCharacter}*{LineTerminator}


Asm           = "asm" | "__asm" | "__asm__"
Volatile      = "volatile" | "__volatile__"

Comment = {TraditionalComment} | {EndOfLineComment} | {DocumentationComment}
TraditionalComment   = "/*" [^*] ~"*/" | "/*" "*"+ "/"
EndOfLineComment     = "//" {InputCharacter}* {LineTerminator}
DocumentationComment = "/**" {CommentContent} "*"+ "/"
CommentContent       = ( [^*] | \*+ [^/*] )*
```

```
{Comment}              /* do nothing */
{Preprocess}           /*do nothing*/
"printk"               PRINTK

{Asm} ({CommentContent});   ASM

"__extension__"        EXTENSION
"__attribute__"        ATTRIBUTE

"__alignof__"          ALIGNOF
"typeof"               TYPEOF

"auto"                 AUTO
"_Bool"                BOOL
"break"                BREAK
"case"                 CASE
"char"                 CHAR
"_Complex"             COMPLEX
"const"                CONST
"continue"             CONTINUE
"default"              DEFAULT
"do"                   DO
"double"               DOUBLE
"else"                 ELSE
"enum"                 ENUM
"extern"               EXTERN
"float"                FLOAT
"for"                  FOR
"goto"                 GOTO
"if"                   IF
"_Imaginary"           IMAGINARY
"inline" | "__inline" | "__inline__"        INLINE
"int"                  INT
"long"                 LONG
"register"             REGISTER
"restrict"             RESTRICT
"return"               RETURN
"short"                SHORT
```

```
"signed" | "__signed__"        SIGNED
"sizeof"             SIZEOF
"static"             STATIC
"struct"             STRUCT
"switch"             SWITCH
"typedef"            TYPEDEF
"union"              UNION
"unsigned"           UNSIGNED
"void"               VOID
"__volatile__" | "volatile"    VOLATILE
"while"              WHILE



{L}({L}|{D})*        IDENTIFIER

0[xX]{H}+{IS}?       CONSTANT
0[xX]{H}+{IS}?       CONSTANT
0{D}+{IS}?           CONSTANT
{D}+{IS}?            CONSTANT
L?'(\\.|[^\\'\n])+'  CONSTANT
{D}+{E}{FS}?         CONSTANT
{D}*"."{D}+({E})?{FS}?        CONSTANT
{D}+"."{D}*({E})?{FS}?        CONSTANT
0[xX]{H}+{P}{FS}?            CONSTANT
0[xX]{H}*"."{H}+({P})?{FS}?   CONSTANT
0[xX]{H}+"."{H}*({P})?{FS}?   CONSTANT

{StringLiteral}      STRING_LITERAL

 "..."         ELLIPSIS
 ">>="         RIGHT_ASSIGN
 "<<="         LEFT_ASSIGN
 "+="          ADD_ASSIGN
 "-="          SUB_ASSIGN
 "*="          MUL_ASSIGN
 "/="          DIV_ASSIGN
 "%="          MOD_ASSIGN
 "&="          AND_ASSIGN
```

```
"^="              XOR_ASSIGN
"|="              OR_ASSIGN
">>"              RIGHT_OP
"<<"              LEFT_OP
"++"              INC_OP
"--"              DEC_OP
"->"              PTR_OP
"&&"              AND_OP
"||"              OR_OP
"<="              LE_OP
">="              GE_OP
"=="              EQ_OP
"!="              NE_OP


("{"|"<%")        '{'
("}"|"%>")        '}'
("["|"<:")        '['
("]"|":>")        ']'
","               ','
":"               ':'
"="               '='
"("               '('
")"               ')'
";"               ';'
"."               '.'
"&"               '&'
"!"               '!'
"~"               '~'
"-"               '-'
"+"               '+'
"*"               '*'
"/"               '/'
"%"               '%'
"<"               '<'
">"               '>'
"^"               '^'
"|"               '|'
"?"               '?'
```

1.21   C Grammar

```
token PRINTK ASM ATTRIBUTE ALIGNOF TYPEOF PREPROCESS EXTENSION
token IDENTIFIER CONSTANT STRING_LITERAL SIZEOF
token PTR_OP INC_OP DEC_OP LEFT_OP RIGHT_OP LE_OP GE_OP EQ_OP NE_OP
token AND_OP OR_OP MUL_ASSIGN DIV_ASSIGN MOD_ASSIGN ADD_ASSIGN
token SUB_ASSIGN LEFT_ASSIGN RIGHT_ASSIGN AND_ASSIGN
token XOR_ASSIGN OR_ASSIGN TYPE_NAME
token TYPEDEF EXTERN STATIC AUTO REGISTER INLINE RESTRICT
token CHAR SHORT INT LONG SIGNED UNSIGNED FLOAT DOUBLE CONST VOLATILE VOID
token BOOL COMPLEX IMAGINARY
token STRUCT UNION ENUM ELLIPSIS
token CASE DEFAULT IF ELSE SWITCH WHILE DO FOR GOTO CONTINUE BREAK RETURN


primary_expression
    : IDENTIFIER
    | CONSTANT {
    | STRING_LITERAL
    | '(' expression ')'
    ;


postfix_expression
    : primary_expression
    | postfix_expression '[' assignment_expression ']'
    | postfix_expression '(' ')'
    | postfix_expression '(' argument_expression_list ')'
    | postfix_expression '.' IDENTIFIER
    | postfix_expression PTR_OP IDENTIFIER
    | postfix_expression INC_OP
    | postfix_expression DEC_OP
    | '(' type_name ')' '{' initializer_list '}'
    | '(' type_name ')' '{' initializer_list ',' '}'
    ;


printk_arguement_list
    : assignment_expression
    |  assignment_expression',' printk_arguement_list
    | assignment_expression printk_arguement_list
    ;
```

```
argument_expression_list
    : assignment_expression
    | argument_expression_list ',' assignment_expression
    ;

unary_expression
    : postfix_expression
    | INC_OP unary_expression
    | DEC_OP unary_expression
    | unary_operator cast_expression
    | SIZEOF unary_expression
    | SIZEOF '(' type_name ')'
    | ALIGNOF '(' type_name ')'
    ;

unary_operator
    : '&'
    | '*'
    | '+'
    | '-'
    | '~'
    | '!'
    ;

cast_expression
    : '(' type_name ')' cast_expression
    | '(' IDENTIFIER '*' ')' cast_expression
    | '(' IDENTIFIER ')' cast_expression
    | unary_expression
    ;

multiplicative_expression
    : cast_expression
    | multiplicative_expression '*' cast_expression
    | multiplicative_expression '*' init_declarator
    | multiplicative_expression '/' cast_expression
    | multiplicative_expression '%' cast_expression
    ;
```

```
additive_expression
    : multiplicative_expression
    | additive_expression '+' multiplicative_expression
    | additive_expression '-' multiplicative_expression
    ;

shift_expression
    : additive_expression
    | shift_expression LEFT_OP additive_expression
    | shift_expression RIGHT_OP additive_expression
    ;

relational_expression
    : shift_expression
    | relational_expression '<' shift_expression
    | relational_expression '>' shift_expression
    | relational_expression LE_OP shift_expression
    | relational_expression GE_OP shift_expression
    ;

equality_expression
    : relational_expression
    | equality_expression EQ_OP relational_expression
    | equality_expression NE_OP relational_expression
    ;
and_expression
    : equality_expression
    | and_expression '&' equality_expression
    ;

exclusive_or_expression
    : and_expression
    | exclusive_or_expression '^' and_expression
    ;

inclusive_or_expression
    : exclusive_or_expression
    | inclusive_or_expression '|' exclusive_or_expression
```

```
    ;

logical_and_expression
    : inclusive_or_expression
    | logical_and_expression AND_OP inclusive_or_expression
    ;
logical_or_expression
    : logical_and_expression
    | logical_or_expression OR_OP logical_and_expression
    ;

conditional_expression
    : logical_or_expression
    | logical_or_expression '?' expression ':' conditional_expression
    ;

assignment_expression
    : conditional_expression
    | unary_expression assignment_operator assignment_expression
    ;

assignment_operator
: '='
| MUL_ASSIGN
| DIV_ASSIGN
| MOD_ASSIGN
| ADD_ASSIGN
| SUB_ASSIGN
| LEFT_ASSIGN
| RIGHT_ASSIGN
| AND_ASSIGN
| XOR_ASSIGN
| OR_ASSIGN
;

expression
: assignment_expression
| expression ',' assignment_expression
;
```

```
constant_expression
: conditional_expression
;

declaration
: EXTENSION declaration
|declaration_specifiers ';'
| declaration_specifiers init_declarator_list ';'
| ASM
;

declaration_specifiers
: storage_class_specifier
| storage_class_specifier declaration_specifiers
| function_specifier
| type_qualifier
| type_qualifier declaration_specifiers
| type_specifier
| type_modifiers_list
| type_modifiers_list defined_type_specifier
| typeof_function
;

typeof_function
: TYPEOF '(' parameter_declaration ')'
| TYPEOF '(' expression ')'
;

specifier_qualifier_list
: type_specifier
| type_modifiers_list
| type_modifiers_list defined_type_specifier
| type_qualifier specifier_qualifier_list
| type_qualifier
;


type_modifiers_list
```

```
: type_modifier
| type_modifier type_modifiers_list
| type_modifier type_qualifier
| type_modifier function_specifier
| type_modifier storage_class_specifier
;

type_modifier
: SHORT
| LONG
| SIGNED
| UNSIGNED
;

type_specifier
: VOID
| CHAR
| SHORT
| INT
| LONG
| FLOAT
| DOUBLE
| SIGNED
| UNSIGNED
| BOOL
| COMPLEX
| IMAGINARY
| struct_or_union_specifier
| enum_specifier
| IDENTIFIER
| type_specifier function_specifier
;

defined_type_specifier
: VOID
| CHAR
| SHORT
| INT
| LONG
```

```
| FLOAT
| DOUBLE
| SIGNED
| UNSIGNED
| BOOL
| COMPLEX
;

init_declarator_list
: init_declarator
| init_declarator_list ',' init_declarator
;

init_declarator
: declarator
| declarator '=' initializer
;

storage_class_specifier
: TYPEDEF
| EXTERN
| STATIC
| AUTO
| REGISTER
;

struct_or_union_specifier
: struct_or_union_declaration
| struct_or_union_specifier attribute_function
;

struct_or_union_declaration
: struct_or_union IDENTIFIER '{' struct_declaration_list '}'
| struct_or_union '{' struct_declaration_list '}'
| struct_or_union '{' '}'
| struct_or_union IDENTIFIER
| struct_or_union IDENTIFIER '{' '}'
;
```

```
struct_or_union
: STRUCT
| UNION
;


struct_declaration_list
: struct_declaration
| struct_declaration_list struct_declaration
;


struct_declaration
: specifier_qualifier_list struct_declarator_list ';'
| specifier_qualifier_list ';'
;


struct_declarator_list
: struct_declarator
| struct_declarator_list ',' struct_declarator
;


struct_declarator
: declarator
| ':' constant_expression
| declarator ':' constant_expression
;


enum_specifier
: ENUM '{' enumerator_list '}'
| ENUM IDENTIFIER '{' enumerator_list '}'
| ENUM '{' enumerator_list ',' '}' {
| ENUM IDENTIFIER '{' enumerator_list ',' '}'
| ENUM IDENTIFIER
;


enumerator_list
: enumerator
| enumerator_list ',' enumerator
;
```

```
enumerator
: IDENTIFIER
| IDENTIFIER '=' constant_expression
;

type_qualifier
: CONST
| RESTRICT
| VOLATILE
;

function_specifier
: INLINE
| attribute_function
;


declarator
: pointer direct_declarator
| direct_declarator
;

attribute_function
: ATTRIBUTE '(' expression ')'
| ATTRIBUTE '(' ')'
;

direct_declarator
: IDENTIFIER
| '(' declarator ')'
| direct_declarator '[' type_qualifier_list assignment_expression ']'
| direct_declarator '[' type_qualifier_list ']'
| direct_declarator '[' assignment_expression ']'
| direct_declarator '[' STATIC type_qualifier_list assignment_expression ']'
| direct_declarator '[' type_qualifier_list STATIC assignment_expression ']'
| direct_declarator '[' type_qualifier_list '*' ']'
| direct_declarator '[' '*' ']'
| direct_declarator '[' ']'
| direct_declarator '(' parameter_type_list ')'
```

```
| direct_declarator '(' identifier_list ')'
| direct_declarator '(' ')'
;

pointer
: '*'
| '*' type_qualifier_list
| '*' pointer
| '*' type_qualifier_list pointer
;

type_qualifier_list
: type_qualifier
| type_qualifier_list type_qualifier
;


parameter_type_list
: parameter_list
| parameter_list ',' ELLIPSIS
;

parameter_list
: parameter_declaration
| '(' parameter_list ')'
| parameter_list ',' parameter_declaration
;

parameter_declaration
: declaration_specifiers declarator
| declaration_specifiers abstract_declarator
| declaration_specifiers
;

identifier_list
: IDENTIFIER
| identifier_list ',' IDENTIFIER
;
```

```
type_name
: specifier_qualifier_list
| specifier_qualifier_list abstract_declarator
;

abstract_declarator
: pointer
| direct_abstract_declarator
| pointer direct_abstract_declarator
;

direct_abstract_declarator
: '(' abstract_declarator ')'
| '[' ']'
| '[' assignment_expression ']'
| direct_abstract_declarator '[' ']'
| direct_abstract_declarator '[' assignment_expression ']'
| '[' '*' ']'
| direct_abstract_declarator '[' '*' ']'
| '(' ')'
| '(' parameter_type_list ')'
| direct_abstract_declarator '(' ')'
| direct_abstract_declarator '(' parameter_type_list ')'
;

initializer
: assignment_expression
| IDENTIFIER ':' assignment_expression
| '{' initializer_list '}'
| '{' initializer_list ',' '}'
| '{' '}'
| '(' initializer ')'
;

initializer_list
: initializer
| designation initializer
| initializer_list ',' initializer
| initializer_list ',' designation initializer
```

```
;

designation
: designator_list '='
;

designator_list
: designator
| designator_list designator
;

designator
: '[' constant_expression ']'
| '.' IDENTIFIER
;

statement
: labeled_statement
| compound_statement
| expression_statement
| selection_statement
| iteration_statement
| jump_statement
;

labeled_statement
: IDENTIFIER ':' statement
| CASE constant_expression ':' statement
| DEFAULT ':' statement
;

compound_statement
: '{' '}'
| '{' block_item_list '}'
;

block_item_list
: block_item
| block_item_list block_item
```

```
;

block_item
: declaration
| statement
;

expression_statement
: ';'
| expression ';'
;

selection_statement
: IF '(' expression ')' statement
| IF '(' expression ')' statement ELSE statement
| SWITCH '(' expression ')' statement
;

iteration_statement
: WHILE '(' expression ')' statement
| DO statement WHILE '(' expression ')' ';'
| FOR '(' expression_statement expression_statement ')' statement
| FOR '(' expression_statement expression_statement expression ')' statement
| FOR '(' declaration expression_statement ')' statement
| FOR '(' declaration expression_statement expression ')' statement
;

jump_statement
: GOTO IDENTIFIER ';'
| CONTINUE ';'
| BREAK ';'
| RETURN ';'
| RETURN expression ';'
;

program
: translation_unit

translation_unit
```

```
: external_declaration
| external_declaration  translation_unit
;

external_declaration
: function_definition
| declaration
;

function_definition
: declaration_specifiers declarator declaration_list compound_statement
| declaration_specifiers declarator compound_statement
;

declaration_list
: declaration
| declaration_list declaration
;
```