Department of Mathematics and Computer Science

University of Southern Denmark

Bachelor Thesis in Computer Science

Design and implementation of an imperative programming language

Author
Casper Juul Majgaard Nielsen
caspn18@student.sdu.dk

Author
Christoffer Falk Bøgebjerg
chboe17@student.sdu.dk

Supervisor
Kim Skak Larsen
Professor

June 1, 2021

Resumé

Denne rapport gennemgår vores design og implementation af en imperativ compiler. Rapporten indledes med en kort introduktion, efterfulgt af hvordan man bruger vores compiler, og hernæst en sprogbeskrivelse. Herefter findes der følgende undersektioner:

- Scanning, som er ansvarlig for at oversætte input til et format der kan genkendes af parseren.
- Parsing, som er ansvarlig for at opbygge vores abstrakte syntaks træ (AST).
- Symbol collection, som opsamler variabel- og funktionsnavne.
- Type checking, som sikrer at alle operationer er lovlige i forhold til typerne. Denne fase er desuden udvidet med type coercion for at være mere fleksibel.
- Intermediate code generation, som bruger alt tidligere information til at danne et sæt pseudo-kode assembly instruktioner.
- Peephole optimization, som er en ekstra fase, der optimerer instruktionerne for et mindre og eller hurtigere program.
- Code emit, der oversætter pseudo-kode instruktionerne til linux x86 assembly.

Efter denne lange sektion findes der et afsnit om testing, der gennemgår vores korrekthedstests, og derudover vores optimeringstests vedrørende peephole optimization. Herefter findes konklusionen, der dækker vores afsluttende tanker om koden og projektet. Til sidst findes referencer og appendix.

Contents

1 Introduction

The following paper describes the design and implementation of an imperative programming language, with the following requirements:

The language will as a minimum contain the traditional control-flow structures in the form of looping, conditionals, and sub-routines. It should have more than one primitive type and operators for these.

The language will be implemented following the six traditional phases of scanning, parsing, symbol collection, type checking, intermediate code generation, and emission. Along with these six phases, an addition peephole optimization phase has been added between intermediate code generation and code emission. Furthermore, the type checking phase has been extended to include type coercion between doubles and integers. The compiler is be written in C and compiles to Assembly x86 using AT&T syntax. The scanner is crafted using flex[3] and the parser using bison[4].

This paper is structured in such a way that it will give a simple language description and then linearly go through each of the compiler phases, followed by a section regarding testing, a conclusion and finally references and appendix. Each of the subsections regarding the compiler phases will be structured in a way that will first describe the general design philosophies before looking closer at the implementation and code.

2 How to use

Our compiler is very easy to use. First you must navigate to the folder of our compiler: *chboecaspnBachelor*. Then the entire program can be compiled on linux by running the shell script *compileCompiler.sh*. From here you can either run the tests by running the shell script *runAllTests.sh*, or you can compile your own Charles program by running *compile.sh* followed by your desired *.txt* file like so:

```
./compile.sh example.txt
```

This will produce a file called *example.s* and a file called *example.out* in the same location as the input file. The assembly file contains the assembly code and the *.out* file can be run with the command:

```
./example.out
```

A valid installation of both flex and bison must be installed. On linux you can install flex by using the following commands:

```
sudo apt-get update
sudo apt-get install flex
```

And bison with:

```
sudo apt-get update
sudo apt-get install bison
```

On Windows we have included a folder with the flex and bison installation we used during development. This means our compiler will run on Windows but not the resulting program from compiling a Charles program. Our zip file contains a folder named *Compiler*. This folder contains all the files for the compiler written in C. In this folder you will also find auto-generated files.
Our *Tests* folder contains all our tests mentioned in section *5. Testing*.

3 Language description

The language we have developed is called Charles. This is an abbreviation for "Chars are useless" since we discovered midway through our development, that chars had no functionality besides being printable. A program in Charles consists of a series of statements terminated by a semi-colon. Charles was mainly developed for arithmetic computation which is reflected in the rather simple but sufficient type system.

The language contains 4 native types: ints, chars, doubles and booleans all of which are printable. You can declare a variable of a desired type by first writing the type followed by the name of the variable. From here you can either immediately assign the variable a value or you can choose to assign it later. Variables can be used in the current scope following its declaration and in any sub-scopes. If a variable is not declared in the immediate scope Charles will look for the variable in the parent-scopes and fetch the most recently declared variable of the same name. Thus Charles implements static scoping.

Functions are declared by firstly writing the return type, followed by the name of the function, opening parenthesis, a series of arguments declared as a type and a name, a closing parenthesis and finally a compound statement which can contain a series of statements, or a single statement. Example:

```
boolean bigger(int x, int y){ #code }
```

A function must be able to return a value of the given return type in all of its branches. A function can be used everywhere in the scope it is declared in and also in any sub-scopes. This means that Charles also supports recursive functions.

Charles supports long arithmetic expressions which follow standard mathematical precedence order from high to low:

- parentheses
- multiplication and division
- subtraction and addition
- boolean compare operators
- remaining boolean operators

All operators are left-associative.
Charles also supports implicit coercion from ints to doubles which is inferred from the context. This makes Charles a powerful yet simple-to-write computing language. Lastly, Charles has comments declared with a # and terminated by a new line character.

Lastly, Charles supports the standard control-flow structures like while-loops, which can be declared with the key-word *while* followed by parenthesises containing a boolean expression and then a compound or singular statement. Similarly, it supports if-then-else which is declared with the key-word *if* followed by parenthesis containing a boolean expression, then a compound statement and lastly an optional *else* and compound statement.

Here you can see an example of Charles code:

```
int i = 2;          #Standard declaration and assignment in a single line
char c;             #Declaration without an assignment
c = 't';            #The assignment without the declaration
double d = 2.5;
boolean b = true;

print(i*-d);        #Coercion from int to double
print(func(1));

int func(int i){
   while(i<10){
      i = i*2;
   }

   if(i < 10){
      return 1;     #Return in a branch of func
   } else {
      return 0;     #The other return which makes sure func always returns
   }
}
```

For a more detailed description, see section *4.2.1 The grammar*.

4 Design and implementation of compiler phases

4.1 Scanning

This phase is responsible for translating text into key-words that can later be parsed into our abstract syntax tree, from here denoted AST, that will create the basis for most of the remaining compiler phases.

We crafted a scanner using Flex as we had experience with Flex from a previous university course and it is an otherwise well functioning scanning tool. Our resulting Flex scanner was very traditional with a couple of reserved key-words:

- print
- return
- while
- if
- else
- true
- false
- the 4 types: int, double, boolean and char

Initially we also wanted to reserve the word main for easy differentiating the main function but we ultimately decided to remove this as we found no usage of a dedicated main function. Instead the global scope now serves as the main function. With the use of precedence rules we initially matched single characters as chars but we figured this would create problems with single character named variables which would no longer be matched as identifiers but instead chars. We had to decide whether we would sandwich chars between apostrophes or if we would rather handle this problem in the parser. We found the second solution to be bad and the first to be very natural as many other languages declare char values in between apostrophes.

Since the procedure for matching a regex by longest match followed by top to bottom precedence is nearly identical for all the regexes constructed we will look at a single example of how to handle such a match:

```
[a-zA-Z][a-zA-Z0-9]* {
                        yylval.stringconst = (char*)malloc(strlen(yytext)+1);
                        sprintf(yylval.stringconst,"%s",yytext);
                        return tIDENTIFIER;
                    }
}
```

This is the regex for matching an identifier. The name is saved in *yylval* and it returns a token called *tIDENTIFIER*. Similarly other matches return tokens, symbols that are just chars such as parenthesis and arithmetic symbols return these symbols except for compare symbols which return a textual representation of itself such as *LE*, *G* etc. The reserved keywords each return a capitalized version of this keyword. The top to bottom precedence was designed in such a way that our reserved keywords precede all others.
Besides these ideas not a lot of energy has been put into the design and implementation as this phase is very simple.

4.2 Parsing

While our parser is pretty generic in many ways there are still a few design decisions we want to highlight.
When designing this phase we quickly realised we had to do some serious considerations of what was a semantically correct definition of expressions, statements etc. For us this was a tough definition to get right so we kept revisiting certain areas of the parser as we explored later phases of the compiler to make sure we had correct abstractions. The two areas we kept revisiting were construction of main functions and construction of statements.

For a long time we wanted to have a main function, followed by other functions in the same scope as the main scope, that the main function could then call, without allowing any inner function declarations. For a time this was nice and very simple, but we also realised, that since you can have separate functions it would be nice to have global variables, so we allowed variable declarations and assignments into the global scope. Once we introduced global variables we realised we would have trouble resolving these variables. Let us look at an example:

```
int i = 2;

int main(){
   return myFunc(2);
}

int myFunc(int x){
   return i+x;
}
i = 4;
```

In this example it would be hard to figure which i is being used for *myFunc* since you do not know when main is being called. C solves this problem by simply not allowing redefinitions of global variables and we considered this solution but it also made us reconsider why we even had a main function to begin with. It was natural for us since languages like C and Java have main functions and those are languages we are used to. In C the main function acts as an entry point for the program and global variables are evaluated exactly as they are declared. In Java you can use the main function to pass additional information to the program. Our language is far from as complex as Java and we saw no reason to not simply enter the program at the very first statement so therefore we decided to remove our main function and simply execute from the very first line. This way we corrected our so far bad or partially completed abstraction.

This leads us to another decision we had to take during our parser design. What exactly is a statement? One approach would be to consider anything terminated by a semi-colon a statement. But this would introduce a lot of useless code like:

```
2+2;
```

This code is evaluated to 4 but it is not used for anything. However banning expressions terminated by semi-colons also seemed like a bad idea because of this example:

```
myFunc(2);
```

Initially this might seem useless, unless you consider the fact that functions can contain print statements so therefore a user of Charles might encounter a problem where it would be nice to print some information and dedicate an entire function to this. Furthermore, with the introduction of global variables, a simple function call can change a lot in the state of the program due to its side effects. We decided that we would allow lone expressions as statements if and only if they were single function calls. We believe that function calls are still expressions, so the kind of the statement is still called *expK*. Lone-standing expressions that are not function calls, are detected and removed during parsing within the function *makeSTMTexp*.

We also encountered a problem when declaring negative values, integers and doubles. For a long time we only tested positive values, but we realised that since our expressions are designed in the following format:

```
operand operator operand
```

we would encounter problems once either of these operands had a minus symbol in front. This problem was solved by letting an expression be formed from a minus followed by an int, double or identifier. A minus in front of an int or double tokens just resulted in an int/double expression holding the value -1 times the value of the token. Identifiers that had a minus in front of them, were reduced to a binop expression of type 0 - id.

Lastly we want to go over some syntactic sugar of the language. Let's look at an example:

```
int x = 4;
```

This single line says we have a variable of type int, named x with value 4. This is broken into 2 statements. The first statement declares a variable of type int with name x. The second statement says that a variable x has value 4, like so:

```
int x;
x=4;
```

Breaking this single statement allows the intermediate code generation phase to be a little more neat as we discovered later.

4.2.1 The grammar

Now, let us turn our attention to the grammar for a more detailed look at how this language works. While some of these grammar rules are uninteresting in design and or implementation they will still be listed if they are important for the understanding of the grammar construction. The rest can be found in the appendix and naturally in the code. As mentioned earlier a program in Charles is a series of statements as such:

Table 1: Rules of a program

`program` :	`statement-node`

From here you can construct statement-nodes as followed:

Table 2: Rules of statement nodes

`statement-node` :	`statement`
`statement-node` :	`statement statement-node`
`statement-node` :	`syntactic-sugar`
`statement-node` :	`syntactic-sugar statement-node`

The non-terminal symbol syntactic-sugar is created when what syntactically appears to be a single statement in the program is turned into multiple statements in the parser. This does however mean, that the syntactic-sugar symbol can be used in place of the statement symbol, wherever it appears, but since the type of the syntactic-sugar symbol is a *STMTNODE** and not a *STMT**, it must be handled differently. When a statement and a statement node are reduced to a new statement node, the statement is simply turned into a statement node, that points to the other statement node. When a syntactic-sugar symbol and a statement node are reduced to a new statement node, we must find the last statement node in the linked list of the syntatic-sugar node and connect it to the other node. An example of syntactic-sugar is, as mentioned above, a declaration and assignment on the same line. This case is handled like so:

```
STMT* stmt1 = makeSTMTvarDecl($1,$2);
STMT* stmt2 = makeSTMTassign($2,$4);
STMTNODE* stmtnode1 = makeSTMTNODE(stmt1,NULL);
STMTNODE* stmtnode2 = makeSTMTNODE(stmt2,NULL);
stmtnode1->next = stmtnode2;
$$ = stmtnode1;
```

Another interesting thing about our statement-nodes is that they are appended onto each other in reverse. Naturally we have a pointer to the head of our linked list of statement nodes and it is much faster to create a new head with a pointer to the previous head than it is to traverse through the entire linked list and append the new statement as a tail. Some readers with knowledge in amateur compilers might recognize this same idea from SCIL[2].
This problem is not entirely avoidable as with this design whenever we insert syntactic sugar we have to traverse through the statement nodes in the syntactic sugar and update the pointer for the last node's next pointer. The new head of the statement nodes will be the head of the syntactic sugar statement nodes. Here you can see how this is handled:

```
stmtnode : syntactic_sugar stmtnode
          | {STMTNODE* curr = $1;
             while(curr->next != NULL)
              curr = curr->next;
             curr->next = $2;
             $$ = $1;}
```

Statements can be created in a lot of ways like so:

Table 3: Rules of statements

`statement`	:	`WHILE ( expression ) cmp-statement`
`statement`	:	`IF ( expression ) cmp-statement ELSE cmp-statement`
`statement`	:	`IF ( expression ) cmp-statement`
`statement`	:	`function-declaration`
`statement`	:	`RETURN expression ;`
`statement`	:	`PRINT ( expression ) ;`
`statement`	:	`tIDENTIFIER ASSIGN expression ;`
`statement`	:	`type tIDENTIFIER ;`
`statement`	:	`expression ;`

From here you can create expressions in many ways:

Table 4: Rules of expressions

expression	:	tIDENTIFIER
expression	:	- tIDENTIFIER
expression	:	tINT
expression	:	- tINT
expression	:	tDOUBLE
expression	:	- tDOUBLE
expression	:	tBOOLEAN
expression	:	' tIDENTIFIER '
expression	:	(expression)
expression	:	expression - expression
expression	:	expression + expression
expression	:	expression * expression
expression	:	expression / expression
expression	:	expression L expression
expression	:	expression G expression
expression	:	expression LEQ expression
expression	:	expression GEQ expression
expression	:	expression EQ expression
expression	:	expression NEQ expression
expression	:	expression AND expression
expression	:	expression OR expression
expression	:	tIDENTIFIER (optional-actual-parameter-node)

Finally we have function declarations like this:

Table 5: Rules of function declarations

function-declaration	:	type tIDENTIFIER (optional-formal-parameter-node) cmp-statement

This optional-formal-parameter-node consists of type *tIDENTIFIER* separated by commas. With this the most important constructions of the grammar is concluded.

4.2.2 Structures

Let us now turn our attention to some of the many structures we had to design and implement to construct our abstract syntax tree.

First let us look at our implementation of expressions.

```
typedef struct EXP {
    int lineno;
    enum {idK,intK,doubleK,boolK,charK,binopK,funK,coerceK} kind;
    union {
        struct {char* id; struct SYMBOL* symbol;} idE;
        char charE;
        int intE, boolE;
        double doubleE;
        struct {struct EXP* left; struct EXP* right; char* operator;}
            binopE;
        struct {char* id; struct SYMBOL* symbol, struct APARAMETERNODE*
            aparameternode;} funE;
        struct EXP* coerceE;
    } val;
    char* type;
} EXP;
```

Expressions contain an line number which sole purpose is for error throwing during compile errors. We also have an enumeration *kind* which is used in combination with the union *val* to access different values depending on the kind. Most of these union members are self explanatory like *binopE* which is used for binary operations. Here it contains a pointer to the left expression, a pointer to the right expression and the operator which should be applied to these two expressions.
On the other hand we also have quirkiness in *idE* which contains a char pointer, the name, and a symbol pointer. Ideally you should only need need the name of a variable to look through symbol tables and find the correct symbol. We encountered a bug during testing regarding variable resolution and as a fix we added this symbol pointer. More about this problem in Section *4.3 Symbol Collection*.

Next let us look at statements.

```
typedef struct STMT {
    int lineno;
    enum {whileK,assignK,ifElseK,returnK,printK,varDeclK,funDeclK,expK}
        kind;
    union {
        struct {struct EXP* guard; struct STMTCOMP* body;} whileS;
        struct {char* name; struct SYMBOL* symbol; struct EXP* val;}
            assignS;
        struct {struct EXP* cond; struct STMTCOMP* ifbody; struct STMTCOMP*
            elsebody;} ifElseS;
        struct {char* type; char* name;} varDeclS;
        struct FUNCTION* funDeclS;
        struct EXP* returnS;
        struct EXP* printS;
        struct EXP* expS;
    } val;
} STMT;
```

Very similar to expressions in its structure, our statements contains a line number, a *kind* enumeration and a union *val* containing different information depending on the kind of statement. Additionally we have a quirk regarding assign which contains both a name and a symbol for very similar reasons as for expressions. This error is not triggered by use in an expression but instead when assigning this variable.

Lastly we want to pay homage to linked lists which have been used extensively throughout our compiler as C has no native support for lists or list-like structures except arrays but arrays are too hard to work with if they change a lot. Our linked lists are very dull in the sense that they mostly just contain a pointer to some object and a pointer to the next linked list node as well. Nonetheless they were very useful so here are a few cases where they were used:

- Collecting statements into a list of statements.
- Collecting formal parameters into a list of formal parameters.
- Collecting actual parameters into a list of actual parameters.
- In a previous version we collected all function declarations into a list of function declarations.

4.3 Symbol collection

In this phase we collect variables, put them into symbol tables and check for symbols that aren't declared yet. First we considered what should be in a symbol table. Naturally variables and functions should go there but so should parameters for function calls, basically anything that can be accessed by an identifier. While designing the symbol collection phase we found ourselves with the problem of when to create the symbol tables. One approach was to create a new table after we entered a compound statement. This led to complications when dealing with function declarations, as their formal parameters should be in same symbol table as the compound statement but since we would traverse the compound statement after the formal parameters, we would have to pass down the formal parameters as arguments which seemed awkward since handling of parameters were dependent on a context we no longer had. Therefore we decided that every symbol table should be created before traversing a compound statement. This also led to the nice invariant of every traverse function containing a pointer to the current symbol table rather than sometimes the parents' symbol table. This had the drawback of some code duplication but it seemed purer in theory.
We also used this phase to check if all branches of a given function would return. We could either enforce all functions ending with a return statement or we could allow the return statements to be divided into if-statements if that was what the programmer desired. This was a and but natural feature as the language would feel weird to type if we had gone with the much simpler solution of functions always ending with a return statement. With our solution it is possible for the user to write useless code after a return statement, but we see that as a user error rather than a compile error.

Let us look at some of the more important structures of the symbol collection phase:

```
typedef struct SYMBOLTABLE {
    struct SYMBOLTABLE* par;
    SYMBOLNODE* symbols;
    int nextVariableLabel;
} SYMBOLTABLE;
```

A symbol table contains a linked list of symbols. This will be described in a moment. It also contains *nextVariableLabel* which is used to determine where, relative to the stack pointer of the current scope, a variable symbol will be.
Next we have symbols which are the entries in the symbol table:

```
typedef struct SYMBOL {
    enum {variable,formalParameter,function} kind;
    char* name;
    char* type;
    FPARAMETERNODE* fpn;
    int offset;
    char* label;
} SYMBOL;
```

Similarly to the structures found in section *4.2 Parsing* we have an enumeration of the different kinds of symbols. We then have a name and a type. For functions this type is their return type and for variables it is naturally their own type. We also have the formal parameters stored which is only used for functions and very useful in the intermediate code generation phase when accessing these parameters. Additionally each symbol has an offset relative to the base pointer which is important for variable symbols and formal parameter symbols. The label is used by function

symbols, each function must have a unique label.
To determine the offset of a formal parameter relative to the base pointer, we must traverse a formal parameter node twice, one time to find out how many more formal parameters that will also be on the stack, and then once more to set the offset of each formal parameter.

We wanted the nice feature of being able to use functions that have yet to be declared. E.g. this should be possible:

```
int x = g();

int g()
{
    return 3;
}
```

This phase is responsible for raising errors whenever symbols are accessed if they do not exist. This gives us an obvious problem since our compiler is not omniscient when encountering function calls and therefore does not know if the function will be declared later or if it never will. To solve this problem we split our symbol collection phase into two parts. The first part would look through declarations and usage of identifiers, but it would not look up the names of functions that are called, specifically because functions can be called before their declaration. After the first traversal, there might be calls to functions, that do not exist. To spot such erroneous code, a second traversal of the AST is made. The second traversal scans through all structures that might contain function-call-expressions, and when such function-call-expressions are found during the second traversal, the name of the called function MUST be in the symbol table, if the function exists within the same scope or parent scopes. This is a little inefficient, and initially this second check was simply made when function-call expressions were encountered in the type checking phase. However, this mvoed some functionality from what in theory should belong to the symbol collection phase into the type checking phase, and we wanted a clean separation of phases.

Now for the issue hinted at, at the end of section *4.2.2 Structures*. Let us look at an example.

```
int i = 5;
int g(){
    int j = i+2;
    char i = 'a';
    return j;
}
```

In a previous implementation this code would cause issues since during type checking, it would look at the current scope, saw i declared as a char but used as an int and therefore raised errors. The type-checking phase had no context of when a variable was declared and a potential solution to this was to look at line numbering. Since you can have multiple statements on a single line we disregarded this solution. Our final solution was to add symbol pointers during the first symbol collection round which should be used in the remaining phases instead of looking up the name. This leaves redundant information since name is contained both in the char pointer and also inside the symbol point and while it is not the most elegant solution it worked well. This issue stemmed from us allowing re-declaration of variables in sub-scopes which introduces ambiguity in symbol resolution.

4.4 Type checking

This phase is responsible for making sure that all operators and operands are valid and compatible as well as making sure they are used in the right context, e.g.: the type of a while-guard expression must be boolean. In this phase there are mainly two important things to type check. The first thing is to make sure, that the resulting type of an expression is correct in the context it is used. In this case you always infer the required type of an expression from the context, having print statements as the only exception, because they can take any type of expression. When assigning an expression (a value) to a variable, the type of the expression must be the same as the type of the variable. For while-loop- and if-guards the type of the expression must be boolean. One thing we did not like about certain programming languages, is when you explicitly have to declare the type of expression, that you want to print. In C you explicitly declare the type by using %d, %f etc. for integers and doubles respectively. In Python it would infer the type and print accordingly. We did not want in our syntax to explicitly have to declare the type of an expression being printed, which means, that when print statements are encountered, the type of the expression to be printed must be inferred from the expression itself, and any type of expression can be printed. We developed print methods for each of our basic types: ints, doubles, chars and booleans. The second thing is type checking inside each expression to make sure operands and operators are compatible. We designed a list of operators and arguments which you can easily verify types of both arguments as well as the resulting type which this expression will take. In this way we also have overloading of our basic operators eg: +, -, *, /, L, G, and == are all defined for both doubles and ints.

As part of our type system we wanted to develop type coercion. For our defined types; ints, doubles, booleans and chars this would mainly be useful for easily combining doubles and integers in longer arithmetic expressions. While it could have been easy to simply leave type coercion after the int-double case, we were not sure if we wanted to add more types later and therefore we developed a more general solution for type coercion where it would be easy to add more types and coercions etc. As such we designed a type system that would consist of several linked lists. Every node in these linked lists contain a type and a reference to the next most general type. This is how our type/sub-type system looks:

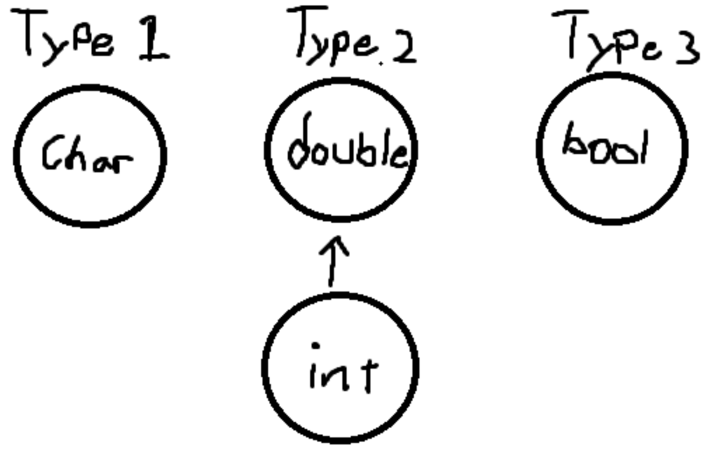

Figure 1: Type/Sub-type system

By our design there should be no native conversion between chars and numbers, numbers and booleans, booleans and chars. This also means that every single element in a given sub-type can be converted to the more general type. Naturally this occurs for int-double conversion. You can easily convert 2 into 2.0 but this should hold true for any arbitrary type you might want to add to the existing type system. This type system will help immensely during type checking and for our desired type coercion.

Now let us look at some of the important structures for this phase. First we have *searchOperations*. To explain this function properly we will first look at operations:

```
typedef struct OPERATION
{
    char* operator;
    int argc;
    TYPE* returnType;
    TYPE* argTypes[];
} OPERATION;
```

As you can see an operation contains an operator, argument count, arguments and return type. Initially we were not sure if we wanted operations with more or less than two arguments e.g modulus. After defining all our operations we found that we only had operators defined for two arguments. Let us look at an example of some operations:

```
OPERATION op0 = { "+", 2, &INT,{&INT,&INT}};
OPERATION op1 = { "+", 2, &DOUBLE,{&DOUBLE,&DOUBLE}};
OPERATION op2 = { "-", 2, &INT,{&INT,&INT}};
```

With this knowledge let us return to *searchOperations*. This function is too big and or complex so it will be explained below as well.

```
OPERATION_WRAPPER searchOperations(operator, typeLeft, typeRight){
    operationWrapper->opCount = 0;
    operationWrapper->op = NULL;
    int opCounter = 0;
    int leastCoercion = -1;
    for(int i = 0; i < operations; i++){
        if(perfectMatch){
            operationWrapper->opCount = 1;
            operationWrapper->op = ALL_OPS[i];
            return operationWrapper;
        }
        int coercionCount = coercionMatch(ALL_OPS[i], op, left, right);
        if(coercionCount != -1){
            if(coercionCount == leastCoercion)
                opCounter++;
            else if(coercionCount < leastCoercion || leastCoercion == -1){
                operation = ALL_OPS[i];
                leastCoercion = coercionCount;
                opCounter = 1;
            }
        }
    }
    operationWrapper->opCount = opCounter;
    operationWrapper->op = operation;
    return operationWrapper;
}
```

This function will basically search through our operations for matches on the operator and the argument types. If it finds a perfect match, no coercion required on any arguments, it will return this operation with some additional meta information. If not it will go through operations while considering coercion. It will return the operation with the least coercion count. The coercion count is the number of times the arguments must be up-casted. If there are multiple operations with the same number of coercions required it will throw an error about operator resolution.

With this information we can now look at how type checking is done for expressions in pseudo-code:

```
case binopK:
    {
        tcTraverseEXP(left);
        tcTraverseEXP(right);
        OPERATION_WRAPPER *opWrapper = searchOperations(operator, leftType,
            rightType);
        if (opWrapper->opCount == 0) {
            Error: Operator not defined
        }
        if (opWrapper->opCount > 1) {
            ERROR: Multiple resolutions
        }
        if (opWrapper.operation.leftType != leftType){
            EXP *ce = makeEXPcoerce(opWrapper->operation.leftType,
                leftExpression);
            e->val.binopE.left = ce;
        }
        if (opWrapper.operation.rightType != rightType) {
            EXP *ce = makeEXPcoerce(opWrapper->operation.rightType,
                rightExpression);
            e->val.binopE.right = ce;
        }
        e->type = opWrapper->op->returnType;
    }
```

As you can see type checking expressions are largely based on the result from *searchOpertions*. If no operator is found it throws an error. If there are multiple results it throws a resolution error. If a single operation is found it will coerce the arguments based on their types and the types from the operation.

For return statements we can look up the function and compare with the return type. Similarly for function calls we can look up the function and compare with the types declared in the formal parameters. Along with these we do type checking for other types expressions and statements but they can mostly be inferred by context. This way we implemented a type checking phase that we are content with as it also considers coercion and least coercion in the case of multiple resolutions for an operator and its operand types.

4.5 Intermediate code generation

The main focus of the intermediate code generation, ICG, is to make the output from this phase as close to assembly pseudo-code as possible, without exposing the quirky details of different architectures and hardware.

We keep the phase simple by generating very template-like sets of instructions, that can potentially be very inefficient, but in return make the phase easily readable, understandable and maintainable. Our hope is to catch and improve these inefficient template-like instructions during *peephole optimization*, which is a separate phase explained in *section 4.6 Peephole optimization*.

This section will go over some of the major design decisions, that were made, and some of the challenges we faced making this phase. Keeping the ICG phase completely uninfluenced by specific architectures, was a much harder ideal to obtain than anticipated, and why this is the case will also be explained later in this chapter.

4.5.1 Data structures

Before diving into this phase a clear idea, of the data structures needed, had to be in place. First of all we acknowledged, that the output from this phase is not a tree structure, but a linear list of instructions. We make use of a linked list of instruction nodes, of which each node have a pointer to "its own" instruction and the next instruction node.

An instruction itself is a data structure containing an operator and 0, 1 or 2 arguments (some asm instructions contain more arguments, but use none of these). The operator is a struct itself and an argument is also a struct. Furthermore, an argument contains a target and a mode, which are two separate structs.
For a quick visualization of the relation between structures see the figures below:

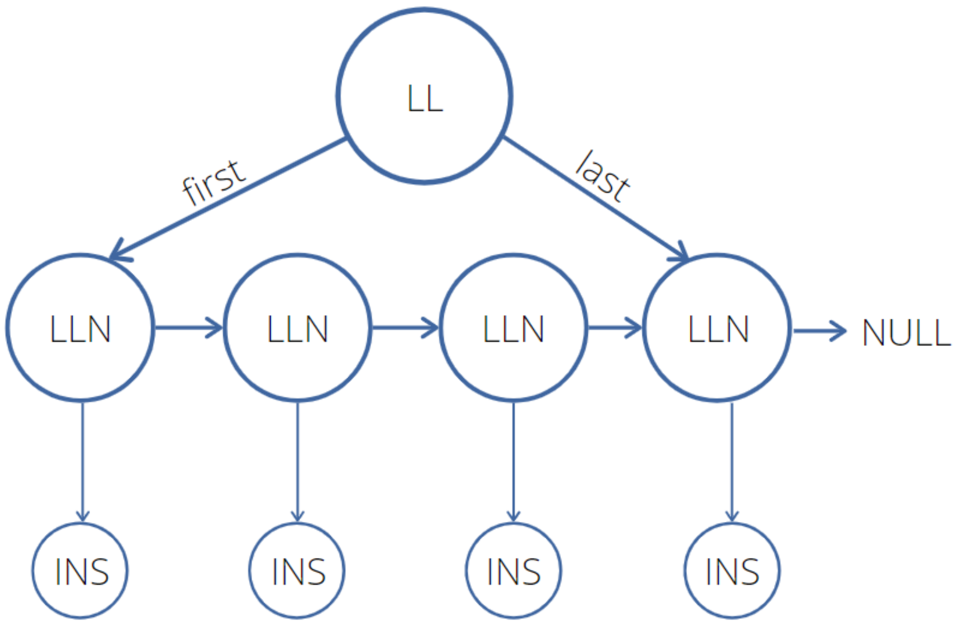

Figure 2: The linked list data structure containing all the instructions.

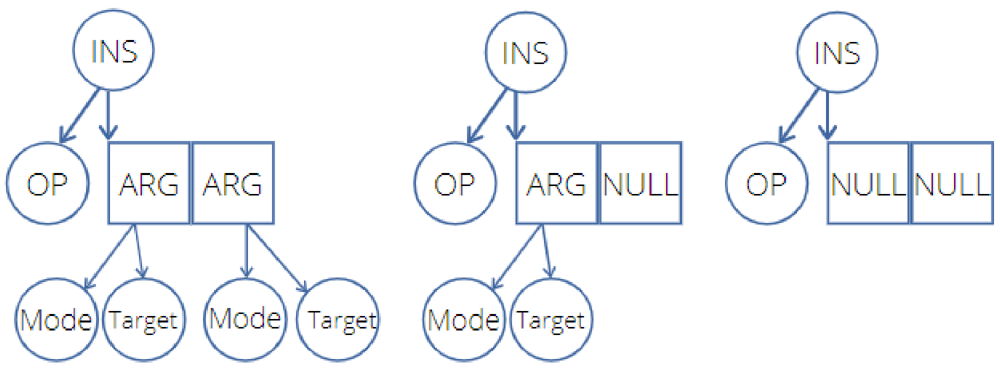

Figure 3: Leftmost instruction has an array of two arguments, middle instruction has one argument and the rightmost instruction has zero arguments.

Each of the data structures Mode, Target, OP also have other fields besides just pointers to other arguments. A lot of inspiration was taken from SCIL[2], when initially choosing what data structures were needed to represent the instructions.

4.5.2 ICG output order

The output of instructions generated in this phase is ordered in the following way:
1. Global Scope
2. Functions
3. Double Declarations
This makes it easy to read and understand the final assembly code once it has been emitted.

4.5.3 AT&T syntax

We wanted the ICG to be completely independent of any specific architecture. However, we quickly realised, that even before beginning to program this phase, we had to make a decision, that would heavily favor one architecture over another. The very first design decision we made, was whether to use AT&T instead of Intel syntax. Seeing as we planned to emit for linux assembly we chose AT&T syntax for our ICG.

```
AT&T syntax:   OP SRC, DEST
Intel syntax:  OP DEST, SRC
```

4.5.4 An important invariant

Arguably the most important invariant of the ICG phase is that after traversing an expression, the code will have been generated, that makes the result of the expression to be on top of the stack regardless of the size, type and complexity of the expression. This invariant simplifies the phase significantly, and traversing an expression is effectively equivalent to generating the code that pushes its result on the stack. Consider the following code for binop expressions:

```
icgTraverseEXP(e->val.binopE.left);        //PUSH OP0
icgTraverseEXP(e->val.binopE.right);       //PUSH OP1
quickPopReg(e->val.binopE.right->type, 1); //POP OP1
quickPopReg(e->val.binopE.left->type, 0); //POP OP0
/* More code to apply operation */
```

Because of this invariant, we can simply call *icgTraverseEXP* on the left expression and then the right expression, and the both of the results will be on the stack.

Another result of this invariant can be seen when assigning a value to a variable:

```
icgTraverseEXP(e);                     //PUSH result
quickPopRRT(getSuffixOfType(e->type)); //POP result into rrt
/* More code that moves RRT into the variable's memory location */
```

After traversing the expression its result will be on the stack so it can simply be popped into rrt and then moved from rrt into the memory location of the variable, that it is being assigned to.

4.5.5 Generating expressions

Having seen this invariant, lets take a look at how it is achieved. The expressions in our programming language are at the highest level categorized into eight types: idK, intK, doubleK, boolK, charK, binopK, funK and coerceK.
Since **intK**, **boolK** and **charK** expressions are all represented by integers, these expressions are simply generated by a push instruction, that pushes their respective integer value on the stack.

The **doubleK** expressions are slightly different, because to represent a double in assembly, we have to make a label for it. E.g. pushing the value 2.0 on the stack must become:

```
    push double_decl_label

double_decl_label:
    .long 0
    .long 1073741824
```

The instruction pushing the label is immediately generated, but as described in *section 4.5.2* the generation of the double label is postponed until the very end of the ICG phase.

A small improvement here would be to not generate a new double label, if a label with exactly that value already exists, so that even though the double constant 2.0 is present twice in the program, a double label declaration with the value 2.0 will only be once in the asm file.

The **idK** (identifier kind) expressions are retrieved by first finding the relative scope depth of the identifier, following the static link <depth> amount of times, and then pushing the variable at an offset from the newly computed base pointer. The offset of the variable relative to the base pointer is contained in the symbol of the expression.

The **binopK** expressions are generated by first traversing the left and then the right expressions and popping these into two registers. The operation is applied to them (add/divide/and/or etc.) and the result is pushed on the stack again.

There is however a different handling of the comparison operators. First the operands are compared according to the operator (>,<, etc.) and then a variant of the "set" instruction is generated and the 8bits-version of register0 is set if the comparison was true. E.g.:

```
cmp reg1, reg0
sete reg0b //The 8-bits version of the register
```

```
pushb reg0b //Push byte on stack (0 if false, 1 if true)
```

If the comparison is between two doubles, then the instructions generated are even more special, and additional code is executed prior to generating the instructions (more about this in *Section 4.5.7 Possible Improvements*).

The **coerceK** expressions are generated by checking the type being coerced from and the type of the expression being coerced from, and then it will accordingly generate an instruction, that does this coercion.[1]

The **funK** expressions are the most complex to generate. They are generated by first traversing the expressions from first to last of the actual parameter nodes (representing the arguments), and due to the invariant, this leaves the values of the arguments on the stack in correct order.
After traversing the last parameter node, the instruction calling the function must now be called. Before doing this, we must consider two things. First, the function being called might not be in the same scope, as the current scope. Second, the function being called must have the base pointer pointing to the same scope that its declaration is in.
Because of this, the template for generating function calls looks like this:

- 1. Push arguments (traverse actual parameternode from first to last)
- 2. Save caller base pointer
- 3. Follow the static link back to the scope expected by the function called
- 4. Call the function
- 5. Restore caller base pointer
- 6. Depush arguments

Step 6 is achieved by generating a single instruction, that adds the sum of the sizes of the arguments to the stack pointer, which has the same effect as popping all the arguments one by one.

4.5.6 Compound statements

A compound statement represents a new scope containing new local variables. Thus, whenever a compound statement is entered, a meta instruction with the meta kind, ALLOCATE_STACK_SPACE, is generated. This instruction is essentially just a collection of three seperate instructions:

- 1. push base pointer
- 2. move stack pointer into base pointer
- 3. subtract sum of sizes of local variables from rsp

These three instructions creates a new static link, that points to the old static link, and it creates room on the stack for all the local variables.

[1]We currently have only one coercion kind, namely from int to double, so only the cvtsi2sd (convert scalar integer to scalar double) opkind can be generated.

4.5.7 Function stack frame

Having explored expressions and compound statements, we can now present a visual representation of the function stack frame.

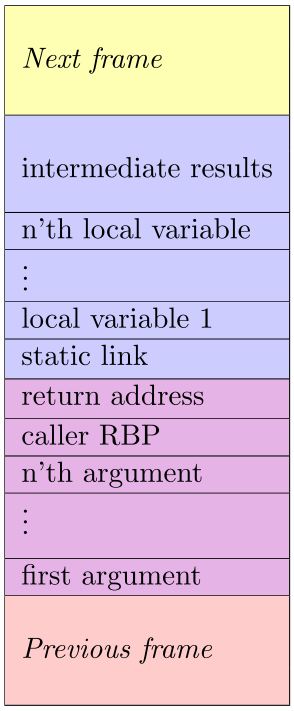

Figure 4: Stack Frame Layout

The light red part of the stack frame is what is generated before calling the function, except for the return address, which is thrown onto the stack by assembly automatically when a function call is made. The light blue part of the stack frame is everything, that the called function pushes onto the stack once it is entered, this includes the "static link", which is the base pointer that is pushed.

4.5.8 Generating statements

Initially the compiler had a lot of code for handling the expression of an **assign statement**, that would make the final output (after emit) contain a lot less instructions, but in return would make the code for handling assign statements would span about 30 lines. We reverted these optimizing changes to the assign function, in order for the ICG phase to be more "clean" and template-like, resulting in a very simple handling of assign statements, hoping that the optimizations could instead be done in the peephole phase. The steps in handling an assignment statement are:
1: Traverse expression
2. Calculate memory address of variable
3. Move stack top into memory address
4. Adjust stack pointer

Step 3 and 4 can actually be reduced to a single pop instruction (given that the register is not XMM).

Generating an **if statement** follows this template[2]:

- 1. Generate if-guard expression

[2]Before step 1 a unique if label is generated, which just makes the asm code easier to read.

- 2. Pop result of guard into reg0
- 3. Compare 1 (true) to reg0
- 4. If not equal jump to endif label
- 5. Generate if-part expression
- 6. Unconditional jmp endelse label
- 7. Generate endif label
- 8. Generate else-part expression
- 9. Generate endelse label

If there is not an else-part to the if-statement, then all steps concerning the else part, steps 6, 8 and 9, are skipped.

While loop statements are quite similar to if statements, they just contain an unconditional jump back to the start of the while loop guard expression. The template for generating while loops looks like this:

- 1. Generate while start label
- 2. Generate while-guard expression
- 3. Pop result of guard into reg0
- 4. Compare 1 (true) to reg0
- 5. Jmp if false to while end label
- 6. Generate while loop body
- 7. Unconditional jmp while start label
- 8. Generate while end label

Return statements are not generating any return instructions. Instead, when encountering return statements, the base pointer is simply reset to the scope of the function body. This can be done because the scope depth, relative to the function being traversed, is kept track of - when a compound statement is entered the scope depth is simply incremented and decremented upon leaving it.
Following this "static link reset" an unconditional jump instruction to a function end label is generated. This makes us able to generate the function epilogue meta instruction only once, namely at the end of a function declaration, regardless of the amount of return statements in a function.

Function declaration statements are handled by adding a pointer to the function to a list, and then after the statements of the global scope have been generated, all the function declarations are generated.

Print statements are simply handled by calling a function named print<type of expression>, meaning if the type of the expression to be printed is an int, then a function call to a function named printINT is generated, and a a flag is set in the struct returned by the ICG phase, so that the emit phase knows whether to include the predefined printINT function in the final asm code. The printINT function must be specified on the emit side, because depending on the architecture, such a print function will look different.

4.5.9 Possible improvements

This phase is so huge and defining for our compiler, that it gets its own section for possible improvements.

Perhaps the meta instructions, that are architecture independent, such as ALLOCATE_STACK_SPACE, should just be translated into the individual different instructions they represent. And then the meta instructions like PROGRAM_EPILOGUE, that are architecture specific, should remain as meta instructions.

We stated in the start of *Section 4.5*, that we wanted to make the ICG phase architecture independent and as close to pseudo-code as possible. The way the ICG phase currently is implemented does not fully comply to these ideals. E.g. when a comparison-binop expression is traversed, if the two operands are doubles, then if the operator is not one of the following: <=, <, ==, !=, then the two operands are swapped. To visualise this:

```
double x = ...;
double y = ...;

if(x < y) //LEGAL
if(x <= y) //LEGAL
if(x == y) //LEGAL
if(x != y) //LEGAL

if(x >= y) //Transform to -> (y <= x)
if(x > y) //Transform to -> (y < x)
```

This is because the compare instruction comisd for doubles in assembly only sets the ZF, CF and PF flags, and if one of the doubles is unordered, is QNaN or SNaN, then the jump above instruction, which jumps when ZF=0 and CF=0, will not do the jump because the CF will be set to 1.[1] So for two doubles x, y we cannot check if x > y or if x >= y, because some flags, that were expected to change, will in the case where one operand is unordered be unchanged. Instead we must transform the greater and greater than comparisons to the equivalent smaller and smaller than comparisons with the operands flipped.
Furthermore, all setcc (set on condition) operators are different for double, "set"-instruction must be converted from setl (set less) and setle (set less equal) to their respective double versions: setb (set below) and setbe (set below equal).

But why did we handle all of this in the ICG phase, and not just let the emit phase handle this? If we did not handle it here, then during the emit phase, if an instruction with the "cmp" opKind is encountered, then in order for the emit phase to know if it should flip the operands, one of two things must be changed:

1: The emit phase makes a one-instruction lookahead when it encounters a cmp instruction between doubles, in order to see if the following "set" instruction is of the kind setg or setge, in which case it must flip the operands and also transform the opKind of the following set instruction from a setg or setge to a setb or setbe.

2: The cmp opKind in the ICG phase is split up into many new kinds: cmpl, cmple, cmpe, cmpne, cmpg and cmpge, and then during the emit phase, when a "cmp" kind of operator is found, we would have enough information to flip the operands immediately.

The second solution probably would be the best, simply because having to make a look-a-head in the emit phase seems like a bad design, and also it would keep the intermediate phase simple, and more importantly oblivious to the processor architecture.

Another small improvement can be made in the way assign statements are generated. Instead of traversing the expression and popping the result into rrt and then moving rrt into the variable's memory location, we can directly pop into the variables memory location.

```
FROM:                       TO:
pop rrt                     pop offset(rbp)
move rrt, offset(rbp)
```

Changing this now will however cause the emit phase to be changed slightly. Specifically in the case where there is a pop instruction into an XMM register, which is illegal, then we would actually need to pop into rrt and make a move instruction from rrt to the XMM register.

Needless to say, the addition of the doubles as a primitive type greatly increased the coupling between intermediate code generation and emit, but this coupling is reducible by different implementation and design choices.

4.6 Peephole optimization

When designing our ICG phase we realised that in many cases the very template like format could be optimized by converting certain instructions to other instructions or by completely removing them. Therefore we extended our base compiler with a phase dedicated to peephole optimization. Here are some of the important design decisions we faced when designing this phase.

When designing this phase we had to choose whether this phase should be placed before or after code emission. We ended up placing the PO phase right after the ICG phase. Although we are only implementing code emission for a single architecture, linux assembly x86 using AT&T syntax, we wanted to develop the PO phase as if we were developing for multiple architectures, like in our ICG phase, and therefore we placed it here to impact as many architectures as possible. This was not an easy decision, as placing PO after code emission would allow us to potentially optimize further, because some singular intermediate instructions are converted to a multitude of instructions in the emit phase due to architecture weirdness.

The patterns we wanted to optimize could be divided into 2 categories:

- Null sequences.
- Combine operations.

The null sequences-reductions represent instructions that do not have any effect on the program such as static link computation when a desired variable is in the current scope. Since there is no static link computation required, the overhead instructions added can simply be removed. Combine-operations reductions represent several instructions that in some cases could reduced to fewer such as pushing a value to the stack and immediately popping it into another register. These two instructions can be reduced to a simple move instruction.

We realised while designing these patterns that sometimes reducing certain patterns might create new patterns in certain contexts which can also be reduced. For this reason it is important to run the peephole optimization multiple times until no changes have been made. This raises the question of whether or not the peephole optimization will ever terminate. If two patterns were to create a cycle eg:

```
pattern a -> pattern b
pattern b -> pattern c
pattern c -> pattern a
```

the program would never terminate since every cycle would result in change which would imply a new cycle and so on. Therefore we came up with a series of principles that would guide the design of all our patterns.

- 1. Every pattern should be strictly smaller in terms of instruction count.
- 2. Every pattern must be better in either memory usage or in execution speed.
- 3. Every pattern must never alter the output of the program.

From principle one it is easy to show that the peephole optimization phase must terminate at some point. If peephole optimization starts we know that there are a finite number of instructions. If not the ICG phase would never terminate and probably crash the program due to finite memory. We also know that if the algorithm recognizes a pattern of instructions it will be reduced to another set of instructions at least one less in size. If it always detects a change it will run until there are 0 instructions left in which case it can not match any patterns and therefore terminate. If it at any point does not detect any changes it will also terminate since we know from the previous cycle there are no patterns to match. This way we can ensure our peephole optimization phase will terminate on any input.
The second principle is important since memory usage and execution speed as our main measurable factors of optimization. More about these measurements in the testing section.
The third principle is important as this phase is purely about optimization and changing the output of the program is not an optimization.

Now let us look at some of the important structures of the implementation of this phase. First we have the function *peepholeOptimization* which runs through our ICG instructions and replaces them based on patterns.

```
void peepholeOptimize(LL* code){
    LLN* previous;
    LLN* current;
    int change = 1;
    while(change)
    {
        change = 0;
        previous = NULL;
        current = code->first;
        while(current != NULL)
        {
            for(int i = 4; i<pattern_count; ++i)
            {
                change |= (*patterns[i])(previous,&current);
                if(current == NULL)
                    break; //Break out of for loop
            }
            previous = current;
            if(current)
                current = current->next;
        }
    }
}
```

The first while loop is responsible for detecting change. After this we loop through our ICG instructions and here we loop through all our peephole patterns. Now let us look at an example of a pattern:

```c
//ADDQ $0, <OP0> -> nothing
int pattern3(LLN* previous, LLN** current){
    INS* ins = (*current)->ins;
    if(ins->op->opK == add && ins->args[0] != NULL &&
        ins->args[0]->target->targetK == imi
    && ins->args[0]->target->additionalInfo == 0){
        previous->next = (*current)->next;
        *current = NULL;
        return 1;
    }
    return 0;
}
```

In peephole optimization we do not look at a "window" of instructions, instead if a pattern needs to look at the current instruction and two more instructions, it just checks if the next two instructions are NULL before starting. An invariant of this phase is also, that the "current" instruction pointer passed to each pattern is never NULL.
If a pattern matches it will construct the replacement instructions and correct the current and previous instruction pointers.

4.6.1 Other Patterns

Consider a binop expression in the tree adding together two immediate integers x and y. This instructions generated by this expression are in pseudo code:

```
    push $x
    push $y
    pop reg1
    pop reg0
    add reg1, reg0
    push reg0
```

This whole sequence of six instructions, that are generated whenever two immediate integers are added together, can be reduced to just this one single instruction, and note that z = x + y:

```
    push $z
```

This is just one example of constant folding, another example of a pattern we could easily detect also is the if(true) statement, which generates the following pattern:

```
    pushb $1
    popb reg0b
    cmp $1, reg0b
    jne endif-label
```

This entire sequence of instructions can be replaced by ϵ. If time allowed it, we would implement many more of these patterns.

4.7 Code emit

This phase is responsible for translating the ICG instructions into a specific architecture. We chose to implement code emission only for one architecture, namely linux x86 assembly AT&T syntax.

The output from the ICG phase is a linked list of instructions, and most of these have a simple conversion to asm source code.

4.7.1 Emitting Instructions

Some instructions are hard to emit, because they must be translated from a single instruction to a sequence of instructions.

One of these special cases are division between two integers, who are of size 32 bits. In this case you have to sign extend the argument to be divided, and put the divisor in register in the %rdx register and the dividend in %rax.

Emitting arguments is simple. If the mode is direct, the target can simply be emitted, if the mode is ind or irl, the target must be surrounded by parentheses and prefixed by an eventual offset and the 64 bits version of it must be used.

Emitting targets is also fairly simple. To convert a register-target we use our *opSuffix* enumeration to index which size of a specific register we want use. Initially *opSuffix* was called *opSize* aand contained the following:

- bits_8, //0
- bits_32, //1
- bits_64, //2

These values would be used index into the different variants of the registers, e.g.:

```
char* raxVariants = {%al, %eax, %rax};
char* raxIntVariant = raxVariants[bits_32]; //%eax
```

However, with the introduction of doubles *opSize* was extended with *bits_64_d* and we would no longer be able to differentiate sizes of just one since doubles are also stored in 64 bits. If the *opSuffix* was *bits_64_d* we would for many operations such as add, divide, compare, have to use the SSE register variants. Perhaps a better naming of *opSuffix* would be *registerVariant*.

4.7.2 Pushing and popping values

There is no "push byte" or "push long" instructions for linux x86 architecture, so in order to handle one of these, let us look at an example with ints:

```
    sub $4, %rsp   //Make space for int on stack
    movl location, %eax //Move the int to eax
    movl %eax, (%rsp) //Move the int from eax to mem
```

The reason why the int is first moved to %eax, is because its location might been in memory and double memory references are illegal. First moving it to %eax and then on the stack is the safest way to deal with this. In many cases this additional move is unnecessary, and since we did ICG before the emit phase, we cannot pattern on match on this unfortunately.
Because of double memory references, we also decided, when seeing a move instruction involving doubles, to first move the double toi %rax and then to the corresponding XMM register, which is many cases also is an unnecessary additional instruction.

4.7.3 Emitting meta instructions

Meta instructions are instructions that perform tasks with very little or no context. Let us look at some examples.
For a properly structured assembly program there must be some program prologue. This program prologue is the same for all programs generated by our compiler.

```
.section .data
.section .text
.global _start
_start:
```

Another good example would be the program epilogue which exits the program. No matter which of our compiled programs you see, these things will always be the same.
Now let us look at a meta instruction that has some context. The meta instruction allocating stack space requires knowing how much stack space you are allocating. This information is stored in the meta integer value of the instruction.

4.7.4 Print Functions

The ICG phase returns a linked list of instructions along with some additional flags. If during the ICG phase a print to an expression of type boolean was found, then a ”printBOOL” flag will have been set to 1, and the same goes for chars, ints and doubles. At the end of the emit phase, depending on what print flags are set, some specific pre-defined print functions will be additionally emitted.

4.7.5 Callee and caller save registers

Currently our compiler does not push and pop the standard callee and caller-save registers, but this could quickly be changed. These instructions are already included from the ICG phase, but they are simply translated to the empty string on the emit side. Since we do not store variables or other important information in the registers, there is currently no benefits to implementing this.

5 Testing

In this section we will go over our testing of our compiler. This will be divided into two subsections, one covering testing of correctness and one covering testing of our peephole optimization.

5.1 Testing of correctness

To test the correctness of our program we have created a series of tests designed to trigger different errors or to test if the correct values are printed. On the following page you will see a table of the tests, a short description, their expected value and their actual value. All tests can be found in the *Tests* folder of the compiler and they can be run by the shell script, *runAllTests.sh* on linux.

As can be seen we are passing all the tests. A valid critique for this approach to testing is that we are not testing potentially missing functionality but if there was any missing functionality we would have tried to cover them in this testing.

Table 6: Tests

Name	**Description**	**Expected**	**Found**
allBranchesReturn1	Checks if compiler allows return in separate branches	1	1
allBranchesReturn2	Checks if compiler throws error in if branch missing return	Error	Error
allBranchesReturn3	Checks if compiler throws error in else branch missing return	Error	Error
allBranchesReturn4	Checks if compiler recursively checks different branches	2	2
booleans	Check all operators for booleans are working	t,f,f,t,t,f	t,f,f,t,t,f
integers	Checks all operators for integers are working	6,0,9,1,f,f,t,f,t,t	6,0,9,1,f,f,t,f,t,t
doubles	Checks all operators for doubles are working	6.0,0,9.0,1.0,f,f,t,f,t,t	6.0,0,9.0,1.0,f,f,t,f,t,t
coercion	Checks coercion from integer to double	$(2^2) * 1.5$	$(2^2) * 1.5$
functionCall1	Checks function calls are working	3	3
functionCall2	Checks parameters in function call	2,'c'	2,'c'
functionCall3	Checks deeply nested function call	6	6
variableScope1	Checks scoping rules for variables	3	3
variableScope2	Checks if compiler throws error on wrong scoping	Error	Error
functionScope	Checks scoping rules for functions	2	2
parsingError	Checks if errors are thrown during parsing	Error	Error
expressionStatement1	Checks if lone statement function call is allowed	3	3
expressionStatement2	Checks if errors are thrown a lone statement is not a function call	Error	Error
symbolCollectionError1	Checks if errors are thrown if variable is not declared	Error	Error
symbolCollectionError2	Checks if errors are thrown if function is not declared	Error	Error
typeCheckingError	Checks if errors are thrown during type checking	Error	Error
functionRedeclaration	Checks if functions can be redeclared in an inner scope	3	3
operatorNotDefined	Checks error if operator operands combination is not defined	Error	Error
bigTest	Checks if a bigger program can be used	200	200
factorial	Checks if factorial function works	120	120

5.2 Testing of peephole optimization

We were curious to explore how well our peephole optimization worked so we designed some tests, for which we could compare the assembly code before and after our optimization. Our approach was to write a test that would match our patterns as many times as we thought would be realistic in a standard program.
Obviously this test data is not at all accurate for a standard program, the program we wrote was specially tailored to match the patterns. Instead we could have found some generic programs online and converted them into Charles and then tested the peephole optimization on those. Nonetheless we wanted to present our results. Here is the test file we came up with:

```
int i = 0;
while(i < 10000000)
{
    int x = 100;
    x = x + 0;
    int y = x + 2;
    i = i + 1;
}
```

And these are the results for the compiled Charles file, assembly code, we got:

Table 7: Optimization Tests

Line count before	**Line count after**	**Reduction**	**Reduction%**
122	71	51	41.8%
Run time before	**Run time after**	**Reduction**	**Reduction%**
0.112s	0.058s	0.054s	48.2%

While these may be impressive results initially it is important to remember that this file was our preconceived idea of a generic program and might not at all be a real representation. This is how many times each pattern was matched:

Table 8: Pattern matches

Pat. 0	**Pat. 1**	**Pat. 2**	**Pat. 3**	**Pat. 4**	**Pat. 5**	**Pat. 6**
11	0	4	0	0	0	3

Some of our patterns were not used at all but we left them in anyways since they might be useful for future reductions. Also remember that the peephole optimization was done before the emit phase, so some patterns we found (by manually looking at our output code), could not be matched during peephole optimization, simply because they were made during emit. This yields some of our patterns currently unusable, at least until we design more patterns.

6 Conclusion

In conclusion we have created a compiler, which compiles from an imperative language dubbed Charles into x86 assembly using AT&T syntax. Our compiler follows the 6 traditional phases of scanning, parsing, symbol collection, type checking, intermediate code generation and code emit. We have extended our type checking phase to include support of implicit type coercion but due to a small number of types it only supports int to double coercion. In addition to this we have implemented a seventh phase between ICG and code emit called peephole optimization, that optimizes speed and size of our resulting assembly program.
In our testing section we passed all the correctness tests and our optimization test yielded a 41.8% decrease in line count and a 48.2% decrease in run time, although the optimization test was designed to show the effect of our peephole patterns.
In general we are happy with our result but if we were to remake this project we would like to redesign and reimplement our ICG and code emit phases as unforeseen complexities emerged during their implementation. We would have also liked to work more with the peephole optimization phase to gain a greater yield, and we can easily implement more peephole patterns. Despite these two annoyances we are very happy with our product.

7 References

References

[1] Intel. Intel® 64 and ia-32 architectures software developer's manual.
http://web.archive.org/web/20190606075330/https://software.intel.com/sites/default/files/managed/a4/60/325383-sdm-vol-2abcd.pdf.
Last visited: 06.01.2021.

[2] Kim Skak Larsen. Scil.
https://imada.sdu.dk/~kslarsen/dm565/scil.php.
Last visited: 31.05.2021.

[3] Vern Paxson. Flex github repo.
https://github.com/westes/flex.
Last visited: 06.01.2021.

[4] The GNU Project. Gnu bison.
https://www.gnu.org/software/bison/.
Last visited: 06.01.2021.

8 Appendix

8.1 Additional grammar rules

Table 9: Rules of compound statements

`cmp-statement`	: {`statement-node`}

Table 10: Rules of syntactic sugar

`syntactic-sugar`	: `type tIDENTIFIER ASSIGN exp ;`

Table 11: Rules of actual parameters

`actual-parameter`	: `expression`

Table 12: Rules of actual parameter nodes

`actual-parameter-node`	: `actual-parameter`
`actual-parameter-node`	: `actual-parameter , actual-parameter-node`

Table 13: Rules of optional actual parameter nodes

`optional-actual-parameter-node`	: ϵ
`optional-actual-parameter-node`	: `actual-parameter-node`

Table 14: Rules of formal parameters

`formal-parameter`	: `type tIDENTIFIER`

Table 15: Rules of formal parameter nodes

`formal-parameter-node`	: `formal-parameter`
`formal-parameter-node`	: `formal-parameter, formal-parameter-node`

Table 16: Rules of optional formal parameter nodes

`optional-formal-parameter-node`	: ϵ
`optional-formal-parameter-node`	: `formal-parameter-node`

Table 17: Rules of types

`type`	: `BOOLEAN`
`type`	: `CHAR`
`type`	: `DOUBLE`
`type`	: `INT`