HrwCC – Technical Report

Stefan Huber Christian Rathgeb Stefan Walkner

July 12, 2007

Abstract

HrwCC is a compiler for a proper subset of the programming language C with influences of C++. The project originated in the course *Compiler Construction* of Prof. Kirsch in the summer semester 2007.

Students in groups of two or three have been asked to build a self-compiling compiler. Our C-compiler generates GNU-assembler output, which can be linked by our linker and executed in our virtual machine. Or, respectively, can be assembled and linked by the GNU-assembler. The following technical report describes the architecture and basic concepts of all components. These include a pre-processor, scanner, parser, symbol-table, code-generator, linker and the virtual machine.

Contents

1 Introduction

1.1 Motivation

HrwCC is a project which has been built for the course *Compiler Construction* in summer semester 2007, held by Prof. Kirsch. Groups of two or three students were asked to develop a self-compiling compiler. This implies working on a subset of an already existing programming language[1]. The minimum requirements can be summarized as follows:

- A type-safe assignment-operator
- Composite statements (sequence, conditional, repetitive)
- Sub-routines with parameters and local variables
- Modules, separate compiling
- Basic datatypes like **int** and **char**
- Arrays and records
- Heap-allocation
- Error-Handling (strong/weak tokens)

1.2 Basic Decisions

There are several constitutional decisions which have to be clarified in the initial phase. Do we produce executables for a specific Operating System (GNU/Linux, hrwOS) or do we develop a virtual machine? Which output-language do we choose for the compiler (ELF object-files, assembler code or own language)? Strongly related: Which commands should be provided by the virtual machine? Should we keep the language simple and put more effort into the compiler[2] or vice versa?

We chose to develop a self-compiling compiler for a specific subset of the C programming language with influences of C++ like the nicer struct definition. The output-language is GNU-assembler code which can be assembled with GCC for many target systems. Instead of implementing our own assembler we decided to develop a virtual machine for (a useful subset of) GNU-assembler code. This leads to several advantages:

- The C programming language is close to hardware and very basic. So it might be easier to implement a compiler. Furthermore, C can be optimized in terms of language-design by dropping unfavorable language elements.
- GNU-assembler code can be used to assemble fast executing binaries and code is easy to debug instead of using a binary format. On the other hand, writing a simple linker and a virtual machine for assembler code is not substantially harder than for a binary format.

[1]Except the possibility of implementing the compiler twice or do a by-hand translation the very first time.

[2]which must be written in a lower-level language therefore.

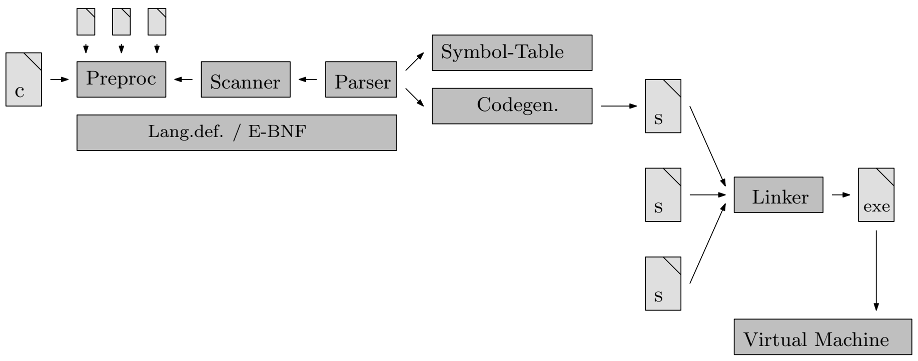

Figure 1: Basic architecture of the whole system.

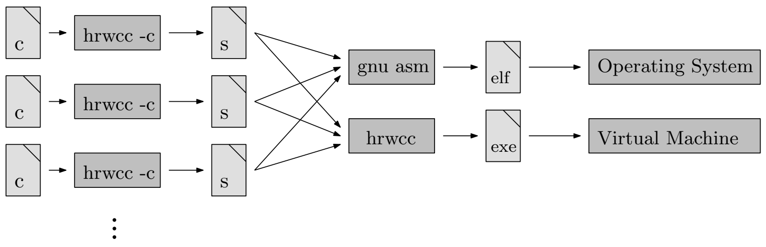

Figure 2: Two alternative ways of building and running an executable.

- Using a well-defined assembler language as ”intermediate language” allows us to develop the compiler and the virtual machine completely parallel. Compiled code can be executed using GCC without having a virtual machine or linker. This leads to a full parallel development process.[3]

Remarks on development Like in a previous project (Operating System hrwOS) too, we split the project up into several milestones (See Fig.-1). We again assigned deadlines to all milestones to get an estimated schedule.
Although it was more difficult this time because we just had a vague imagination of the full project but it became clearer every week. So it was necessary to adapt the schedule in the first weeks to correctly map the project in our milestone-plan.

1.3 Architecture

The basic architecture is illustrated in Fig.-1. The input language is defined in Sec.-B.1 and the object files contain GNU assembler code. The linker produces modified GNU assembler code by mixing all source files together and resolving all labels and simple expressions[4]. The Virtual Machine interprets this modified assembler code and does not have to deal to deal with labels, just with numbers and registers.

[3]It’s clear, that this can also be reached by a precise definition of the compiler-output, but this is hard to achieve in the very beginning.
[4]like $symtab+100

2 Preprocessor

The preprocessor is almost a project of its own because it is such a mighty tool since it supports the most important features a preprocessor could support.

Here is a list of directives the preprocessor implements:

- `#include`
- `#define`
- `#ifndef`
- `#ifdef`
- `#else`
- `#endif`

2.1 Implementation

The preprocessor is implemented using the "pipe-and-filter" programming paradigm. So several tasks of the preprocessor are split into so called "stages" and every stage is able to receive a single char from the previous stage. The C programming language (in particular HrwCC) does not support object oriented paradigms. Instead of implementing a class, a "preproc" struct has been defined, which contains the corresponding members like buffers, flags and so on.

2.1.1 File Stage

The file stage is the very first stage and its main task is to read from a file descriptor and pass the next character of the file to the successor stage. This stage also takes care of the `#include` statement. Because a file could include multiple files and the next character has to be read from the file that is currently dealt with the file stage handles a stack of files. So whenever an `#include` statement is found the file is opened and pushed onto the file stack. So the next character can always be read from the top of the file stack. If an EOF (end-of-file) symbol is reached, the top element of the file stack is closed and removed from the stack. No element remaining on the stack means that the complete source code has been processed and the file stage returns a character containing a "nullbyte". In order to store some meta information (for debugging etc.) with each character a Character type is introduced:

```
struct Character
{
    char value; //'real' value of this Character
    int fileId; //fileId where this character originates (file stack)
    int line;   //line where the character is located
    int column; //column where the character is located
};
```

Usually the scanner invokes the preprocessor and receives a stream of characters from it. By passing Character instances subsequent compiler components (scanner, parser, etc.) can use the embedded positional information for precise error output.

2.1.2 Comment Stage

The comment stage is responsible for removing all kinds of comments in source code. Currently C++ like comments are supported:

```
// single line comment
/* multiple
 * line
 * comment */
```

So the comment stage has a buffer containing a complete source code line. The buffer is iterated through and if the buffer contains either "\\" or "*", a comment starts and the following characters are disposed of until either a newline symbol or the ending "*\" sequence is found.

2.1.3 Directive Stage

The directive stage takes care of the following directives:

- `#define`: Whenever a define statement is discovered the successive identifier and its (possibly empty) value are collected and stored in a list containing all defines. The value of a define can be very long and including multiple lines. So parsing values is done by using a temporary buffer. If the buffer is full the content of the buffer is copied into a dynamically growing data structure using `malloc`. The HrwCC preprocessor supports the definition of "macro functions" as well. Please refer to `util/list.h` in order to see a good example of what the preprocessor can deal with.

- `#ifndef`, `#ifdef`, `#else`, `#endif`: The nesting of these branch directives is handled using a stack. The directives `#ifndef` and `#ifdef` push an element onto the stack and `#endif` pops an element from the stack. So whenever an `#ifndef` or `#ifdef` statement is detected and its condition is not true the subsequent code is disposed until the stack is on the same level as it was when starting the removal. The `#else` statement is handled analogously.

2.1.4 Substitution Stage

The macros and macro functions that were collected by the directive stage have to be replaced whenever a valid pattern occurs in the output stream. Therefore the substitution stage maintains a buffer which contains a whole source code line. Whenever the successive stage asks for a new character the position pointer in the buffer moves one character to the right.

At the new position the new substring in the buffer is compared to all so far detected macro names. If the substring matches a macro, a substitution is done. Of course some conditions about the context of the recognized patterns must be validated, too. In order to return the macro value from now on (until the complete value is replaced) a pointer is used which is always checked when a new character is returned. Whenever this pointer is set the index is incremented and the value at the index position is returned to the caller.

3 Scanner

One essential property of the preprocessor is that the output is a continuous sequence of characters consisting of the merged input files. Therefore the scanner is an instance of the compiler which can be completely reduced to a (deterministic) finite automaton. This is not just a straight and clean way in the sense of software engineering but also leads to a direct mapping of the automaton sketch into code. For this purpose HrwCC defines a scanner struct (see Lst.-1).

Listing 1: Representation of the scanner

```
struct scanner
{
        /**The next character to process*/
        Character nextinput;

        /**The state of the DFA*/
        int state;

        /**The preprocessor which is used*/
        preproc* pp;
};
```

The DFA starts at a well-defined state. For every character that is read from the preprocessor, the DFA changes its state. This is proceeded as long as the DFA halts at the only end-state. In this case the last read character is remembered and the state just before the end-state indicates the token which has been identified. Afterwards the DFA is initialized with the start-state and the last read character is fed to the new initialized automaton. A small part of the automaton is illustrated in Fig.-3.

Remark According to the theory of automata, one could think it is more natural if every state that indicates a token is an end-state. But by doing so, a look-ahead character is needed. For example, a "y" is read and because no keyword starts with this character the DFA is in the ident-state. If the ident-state is an end-state, a decision has to be taken, either to stop here or to read further on, so that the full ident becomes, for example, "yvalue".

We decided to inject this look-ahead character into the automaton. So there is a single end-state and the state just before the end-state indicates the token which has been scanned. By doing so the last read character has to be fed to the automaton after re-initializing.

Implementation Aspects The parser is offered a function called **int** scanner_getToken(scanner* scan, token* tok) to get the next token. This function calls the automaton-transition function **int** scanner_transFunc(**int** state, **char** input) as long as the end-state is not yet reached and saves the characters read so far in the token structure. When the transition-function leads to the end-state the previous state is used to extract the token-number. The scanner is completely implemented in scanner.cpp where all states of the DFA can be found.

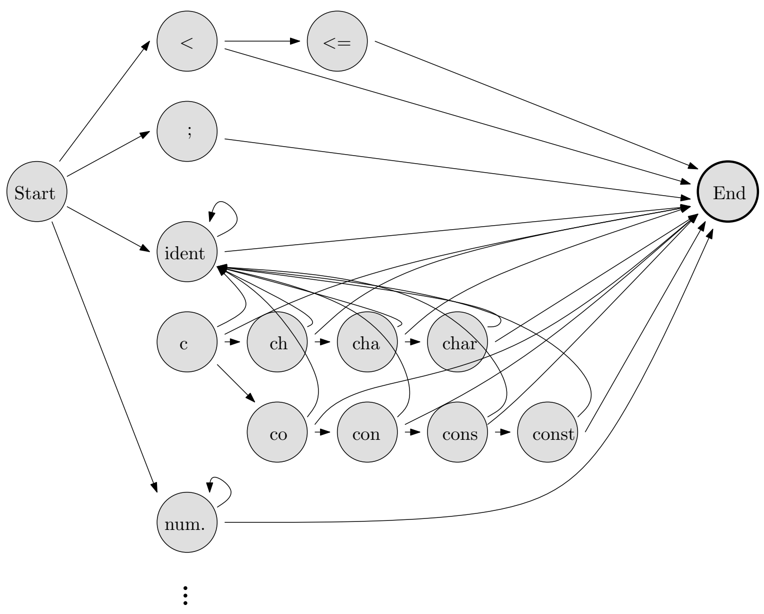

Figure 3: Part of the Deterministic Finite Automaton of the scanner

4 Parser

The parser is one of the core components of a compiler. One of the reasons might be that the syntax is the center of language design[5]. HrwCC implements a recursive descent parser for our language, which is an element of LL(k).[6]. The parser invokes the symbol-table and the code-generation at certain moments (see Fig.-1 and Sec.-4.4) and is therefore settled in the center. Some design aspects of our language are presented in the following and some details of implementation afterwards.

4.1 Language Design Aspects

The appendix (see Sec.-B.1) contains the full E-BNF of the programming language we implemented. There are several reasons why we made some details in this way and not in an arbitrary different one:

- The main focus was on a lightweight, short and clear E-BNF description of our language. The very first design prototypes summarized a full handwritten sheet of paper while our final release is written in 29 lines. Every production has its special meaning in the sense of semantics and the design takes care of "separation of concerns". This will be rewarded when coding the parser, symbol-table and code-generator.

- The language consists of three parts: The first part deals with declarations and definitions of variables, structs and functions. The second part describes all possible statements and structural components of the language and the last part contains all operations for calculating and data-handling.

  This fact is reflected in the code of the symbol-table and code-generator. The symbol-table deals with the first part. The code-generator with the second and third part, but the parser calls the code-generation just for the second one.

- It is useful to use reusable components in the syntax-definition. In our example the `<typed_ident>` production means all definitions/declarations which are tagged with a type. All definitions of variables, parameters or struct members have the same structure: A tree which has a `<typed_ident>` as first sub-tree. This simplifies the symbol-table and code-generation a whole lot.

- We tried to map as much as possible semantics in the productions. Especially in the third part of the E-BNF description. We could have summarized all binary and all unary operations as productions more or less in an arbitrary way, maybe separating logical and arithmetical operations. This would have described our language in a sufficient way.

  But the grammar in Sec.-B.1 maps the semantics of an expression to the structure of the syntax-tree. This means that $3 + 4 \cdot 5 + 6$ is implicitly parsed as $3 + ((4 \cdot 5) + 6)$. We needn't cope with operator-priority and extract the structure afterwards, which would have been very complicated.

[5]There is a good reason, why this is called "design".

[6]Set of all languages parseable with Left-to-right reading, left-most substitution, k characters read ahead

The pattern for this solution originates from [5] and has been extended to our requirements.

- LL(1) versus LL(2). In some cases the extra-effort to trim a language completely to an element of LL(1) is not paid off afterwards. It's sometimes better to let a few productions lead to a LL(2)-language, but with the advantage of a clear grammar.

  We chose the way of LL(2) design[7]. There are three situations where we use two look-ahead characters. But this doesn't increase the complexity of the parser in a measurable fashion because these situations can be solved easily.

4.2 Implementation Aspects

As well as the scanner the parser is represented by a struct (see Lst.-2) . This struct contains the scanner which creates the input tokens for the parser and the symbol-table and code-genderator which are called by the parser at specific positions. Further on, there is a counter of detected errors and a shift-register of look-ahead tokens.

Listing 2: Parser representation

```
struct parser
{
        /**The scanner to get the tokens from*/
        scanner* scan;

        /**The symbol table which is used*/
        symbolTable* symTable;

        /**The code-generator which is used*/
        codegen* cg;

        /**Error occured*/
        int cnterrors;

        /** The look ahead buffer of tokens
         *  (is used as look-ahead-queue) */
        token tokbuffer[PARSER_TOKBUFFER_SIZE];
};
```

An often used function in a parser is eatToken. This function shifts the shift-register to the left and therefore reads a new token for the right-most position. This implementation makes it easy to cope with LL(k) languages. Furthermore, there are addAndEatToken and addAndEatSpecificToken. The first one adds the current token to a syntax tree and calls eatToken. The second works like the first, but expects a token of a specific type. For every production rule parser.cpp implements a parse_xxx function. All functions together form a recursive-descent parser which accepts a LL(2) language.

[7]With a different language than C it would have been easier to reach LL(1)

4.3 Error Recovery

The goal is to detect as many errors as possible in a single run. The worst case would be if the parser stops parsing after an error has been detected. To reach this goal error recovery is necessary. Every language has elements which are more likely forgotten (i.e. semicolon) and others which are not (i.e. struct, while, break, ...). This fact is used to cope with syntax errors.

Weak symbols If HrwCC is missing a semicolon, an error is reported but parsing is continued. This is the only weak symbol which is handled by the parser.

Strong symbols In contrast to weak symbols HrwCC defines a whole set of strong symbols: `EOF`, `}`, `int`, `char`, `void`, `if`, `while`, `return`, `struct`. The parser provides a procedure sync_toStrongKeyword, which calls eatToken until a strong keyword is found. Although `}` is not a "strong keyword" we added it to the list to detect the end of a function- or struct-body.

The parser synchronizes the following productions after an error has been detected: `<stmt_block>`, `<func_body>`, `<struct_de>f` and `<program>`. If the token to which has been synchronized doesn't belong to this production the parser stops parsing in this production and "handles" the responsibility to a higher-level production. I.e. if in `<func_body>` an error occurs and the next synchronized token is `struct`.

4.4 Invocation of Symbol-Table and Code-Generator

One important question is, how the code-generator (and the symbol-table) is invoked. There are (at least) two possibilities:

- One could think of a compiler as a batch-shaped process. The preprocessor feeds the scanner, which feeds the parser, which feeds the semantical analyzer (symbol-table), which at last feeds the code generator. By doing so, the scanner increases the entropy of the character-salad by structuring it to tokens. The parser structures the token-sequences by building syntax-trees. And at last, the code-generation writes down this structure in form of an output-language with less entropy than the syntax-tree.

  So in a sense the front-end creates higher structured entities. The backend dismantles this structure and writes down the output in a specific output-language.

- The other possibility is to settle the parser in the center. This means that the parser invokes the symbol-table and the code-generation for particular code-fragments which have been parsed. This has at least two advantages:

  First of all, the parser already parsed the structure and right after parsing a variable declaration the symbol-table could be called. Otherwise the syntax-tree has to be traversed again in the symbol-table box. Secondly, the compiler could be implemented as a single-pass compiler. Which means, that at the very first time an instruction has been parsed, code for this instruction is generated.[8]

We decided to implement the second possibility – mostly because of the first advantage. But our compiler is in fact a single pass compiler. If you kill HrwCC during compilation, you get the code which has been generated up to the position where the parser stopped[9].

[8]It might be also possible with the batch paradigm. The parser has to stop parsing after each statement...

[9]At least up to the previous parsed statement.

5 Symbol-Table

The symbol-table is a special data-structure which takes and stores all defined and declared functions, variables and structs. By doing so, the symbol-table itself never comes in touch with generated code. It saves higher-level information and provides the answers to questions like:

- Which functions are declared and defined? Which parameters do they take and what is their return type?
- Which variables are defined, are they local or global and what is their size?
- Which structs are defined and what is their size?

This allows the symbol-table to report errors like multiple definitions, unknown types or defining functions which have been declared differently. In our case the symbol-table is presented by a C-struct (see Lst.-3) and contains more or less a list of symbols. A symbol again is described by a struct (see Lst.-4]. It is uniquely determined by its name and type. The type can be a function-declaration, function-definition, struct-declaration, local-variable, global-variable, parameter or a string.

Listing 3: Representation of the symbol-table.

```
struct symbolTable
{
        /** Parser which produces the symbols */
        parser* parse;

        /** Count the errors which have been detected. */
        int cnterrors;

        /** List of symbols saved */
        symbolTableNode_List list;
};
```

Listing 4: Representation of a symbol.

```
struct symbolTableNode
{
        //Ident of symbol (function name, struct identifier, ...)
        token name;

        //Type of symbol (its value is one of the SYMBOL_TYPE_XXX)
        int type;

        //The structure (struct content, variable type, ...)
        syntaxTreeNode* structure;

        DEFINE_LINK( symbolTableNode_List, symbolTableNode );
};
```

Size calculation Due to the fact that the parser generates the syntax-tree, every symbol takes its corresponding subtree of the syntax-tree. For example, a struct-symbol can access its whole definition. Calculating the size of a variable just requires looking at the corresponding subtree and multiplying it (if necessary) with the size of the array-specifier. The same holds for struct-definitions. Their sizes result in the sum of their members. Here the benefit of reusable language elements becomes clear (cf. Sec.-4.1).

There is only one interesting situation when a struct wants to refer to itself by a pointer as member. A prominent example is a linked list. In this case this scheme can lead to a recursive endless loop. This is the reason why we state the size of a pointer to 4 bytes without determining if the type exists. This problem is moved to the code-generation when the member is accessed. If the member is never accessed we do not see a conflict because the size is 4 bytes anyway.

6 Code-Generation

In HrwCC the code-generation is invoked by the parser as mentioned in Sec.-4.4. So every time a production of the E-BNF is recognized a code-generation procedure is called. The output language of the compiler is GNU-assembler code for IA-32 architecture from Intel, which has been discussed in Sec.-1.2. In the following we show some aspects of the code-generation and how we cope with standard problems.

6.1 Function Call

As we want to invoke libc functions like printf or open we have to use the same function call convention like standard C does. The corresponding convention is illustrated in Fig.-4. A similar semantically identical possibility is explained in [2]. So when a function call is parsed the following procedure is processed:

Listing 5: "Function call convention for caller."

```
    #save all used register. %eax will be overwritten
    #by called function for return values.
    SAVE_USED_REGISTERS

    #Make space for the parameters of function. PAR_BLOCKSIZE denotes
    #the size of all parameters.
    subl $PAR_BLOCKSIZE, %esp

    #push all arguments in reverse order. x stands for l or b
    movx argn
    ...
    movx arg1

    #Call the function
    call functionname

    #The return value is now in %eax
    #Get rid of pushed parameters
    addl $PAR_BLOCKSIZE, %esp

    #restore saved registers
    RESTORE_USED_REGISTERS
```

Similarly, when a function definition is parsed the following code is emitted. Please notice that the caller is responsible for saving registers if necessary. Clearly, local variables are held on the stack.

Listing 6: "Function call convention for callee."

```
.globl functionname
.type functionname, @function
functionname:
    #Save old base pointer and set new one to current stack pointer
    pushl %ebp
    movl %esp, %ebp

    #Make space for local variables. LOCVAR_BLOCKSIZE stands for
```

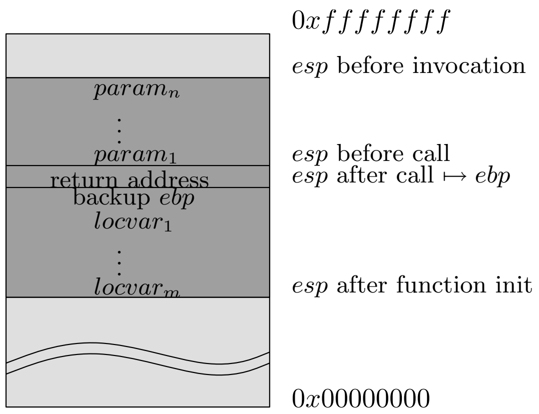

Figure 4: Function call convention of standard C and HrwCC.

```
    #an non-negative integer. So (%ebp) addresses the old %ebp
    #and -4(%ebp) for example the first local variable of long type.
    #The n-th parameter is accessed by 8(%ebp), because 4(%ebp) holds
    #the return address.
    subl $LOCVAR_BLOCKSIZE, %esp

    #...
    #Much code
    #...


    #Return value must already be in %eax
functionname_ret:
    #Restore stack pointer which now points to old %ebp
    movl %ebp, %esp
    #Get rid of old %ebp on stack and restore it
    popl %ebp
    #So here %esp points to the return address of the caller.
    ret
```

Parameter passing Like every other standard C-compiler HrwCC only knows call-per-value. Especially call-per-reference is not supported. But as HrwCC fully supports pointers (of variables, not of functions) a call per reference can be simulated. However, the pointer is passed by call-per-value as well but in this case the address, which is saved ”in“ the pointer-variable, is passed.

Parameter types HrwCC can pass variables of arbitrary types, particularly structs. In the case of passing integers or pointers, the parameters are passed via a `movl` statement to the stack. Respectively, chars are passed via `movb`. Passing a struct-instance is implemented via a set of `movl` and `movb` expressions.

However, arrays can not be passed natively. Which means that the array is not copied onto the stack. Moreover, the array is cast to a corresponding pointer type which is passed to the function in an ordinary way. This corresponds to the missing possibility of array-types for function parameters and meets the requirements of the C function call convention.

Calling undefined functions One advantage of C over Java is the distinction between function declaration and definition. A function can only be called if it has been (at least) declared. In this case the signature of the function is known and a proper type checking of parameters and return values is possible. This pre-condition can be forced – every function can be declared without being defined and so the signature can be known a priori. This fact drops the necessity of fix-up markers, etc. in the code generation.

Library functions take a special place here. They are never implemented (defined), just declared. We solved this general problem by specially signed functions, which are treated by the linker and the VM in a different way (cf. Sec.-7.1).

6.2 Lazy Evaluation in Logical Expressions

As we use GNU-assembler as our output language, labels are available, which enables implementing lazy evaluation in a trivial way. We implemented lazy evaluation for "logical ands" and for "logical ors". The following assembler pseudo-code should make it clear:

Listing 7: Code pattern for lazy evaluation of logical terms

```
    # Evaluate expr1 && expr2 && ... && exprn

    #Evaluate expr1 to %eax
    test    %eax, %eax
    jz      logtermfalse

    #Evaluate expr2 to %eax
    test    %eax, %eax
    jz      logtermfalse

    #...

    #Evaluate exprn to %eax
    test    %eax, %eax
    jz      logtermfalse

    movl    $1, %eax
    jmp     logtermtrue
logtermfalse:
    movl    $0, %eax
logtermtrue:
    #Here %eax contains 1, if expression is true
    #and 0 otherwise.
```

Analogous situation for logical expressions: In this case we stop evaluating when a true term has been found.

Listing 8: Code pattern for lazy evaluation of logical expressions

```
    # Evaluate expr1 || expr2 || ... || exprn

    #Evaluate expr1 to %eax
    test    %eax, %eax
    jnz     logexprtrue

    #Evaluate expr2 to %eax
    test    %eax, %eax
    jnz     logexprtrue

    #...

    #Evaluate exprn to %eax
    test    %eax, %eax
    jnz     logexprtrue

    movl    $0, %eax
    jmp     logtermfalse
logexprtrue:
    movl    $1, %eax
logexprfalse:
    #Here %eax contains 1, if expression is true
    #and 0 otherwise.
```

6.3 Arithmetical Expressions

The most interesting topic in code generation is arithmetical expressions. Especially the management of available registers has to be done carefully. The implementation in HrwCC strictly follows the E-BNF of arithmetical expressions. Regarding register allocation the following rule is applied: Every function which handles a specific E-BNF production can use the registers `eax` to `edx` no matter if they have already been used. In other words: Before calling a subroutine that handles a sub-expression the caller has to save all used registers onto the stack. The same concept is used for function calls.

The following example for the production `<arith_expr>` should illustrate this. All emitting functions have the same signature: They take of course an instance of the codegen structure and the tree where the corresponding syntax-subtree is stored. The parameter called result is set by the emitting function – it could be interpreted as an additional return value. The actual return value stores the type of the value returned by result.

The result variable can be a string like `$3`, `%eax` or `symtab+3(,%ebp,4)` and tells the calling function how the result is actually accessed.

Listing 9: Code generation for arith_expr

```


syntaxTreeNode* codegen_EmitArithExpr( codegen* cg,
                        syntaxTreeNode* tree, char* result)
{
        syntaxTreeNode* subtree;
        syntaxTreeNode* type;
        syntaxTreeNode* oldtype;
        int cnt;
        int idx;
        char result2[CODEGEN_MAXLINE_SIZE];
        char result3[CODEGEN_MAXLINE_SIZE];


        //Just one term, return it
        if( syntax_CountChilds(tree) == 1)
            return codegen_EmitArithTerm(cg, syntax_GetChild(tree, 0), result);


        cnt = syntax_CountChilds(tree);
        subtree = syntax_GetChild(tree,0);

        //First element is minus --> we begin with zero
        if( subtree->tok.type == TOK_MINUS )
        {
                codegen_emit(cg, "\tpushl\t$0\n");
                type = codegen_CreateIntType();
                //We begin with index for the expressions in the
                //loop below
                idx = 1;
        }
        //Otherwise we begin with the first term
        else
        {
                //Evalute first term
                type = codegen_EmitArithTerm(cg, subtree, result);

                //Cast to .long
                subtree = syntax_GetChild(tree, 1);

                //Pointer types are not converted. See later...
                if( !type_IsAPointerType(type) )
                    type = codegen_CastToInt(cg, subtree->tok, result, type);

                codegen_emit(cg, "\tpushl\t");
                codegen_emit(cg, result);
                codegen_emit(cg, "\n");

                //We begin with index 2
                idx = 2;
        }


        //Get further operands and operate on them
        while( idx < cnt)
        {
                //Operator
                subtree = syntax_GetChild(tree, idx-1);

                //Evaluate next operand
                oldtype = type;
                type = codegen_EmitArithTerm(cg, syntax_GetChild(tree, idx),
                                                    result2);
                type = codegen_CastToInt(cg, subtree->tok, result2, type);

                //Determine destination of last result
                if( strcmp(result2, "%ebx") == 0)
                    strcpy(result, "%eax");
                else
                    strcpy(result, "%ebx");

                //Get last operand
                codegen_emit(cg, "\tpopl\t");
                codegen_emit(cg, result);
                codegen_emit(cg, "\n");


                //Old type was a pointer!
                //This changes everything: let 'type* p' be a pointer
                //then p+1 means that to p is sizeof(type) added.
                //But this makes only sence if we have + or - operator
                if( type_IsAPointerType(oldtype) &&
                    ( subtree->tok.type == TOK_PLUS ||
                        subtree->tok.type == TOK_MINUS) )
                {
                        //First move the sizeof to %esi
                        subtree = type_RemoveStarFromType(oldtype);
                        sprintf(result3, "\tmovl\t$%d, %%esi\n",
                                symbol_Sizeof_DataType(cg->parse->symTable,
                                                        subtree));
                        codegen_emit(cg, result3);

                        //And multiply it with the summand
                        codegen_emit(cg, "\timull\t");
                        codegen_emit(cg, result2);
                        codegen_emit(cg, ", %esi\n");

                        //This is the new overall summand
                        strcpy(result2, "%esi");
                        syntax_FreeSyntaxTree(subtree);

                        //Swap oldtype and type
                        //Because now type should be freed and
                        //oldtype is propagated
                        subtree = oldtype;
                        oldtype = type;
                        type = subtree;
                }


                //Last operator
                subtree = syntax_GetChild(tree, idx-1);

                //Subtract result2 from result
                if( subtree->tok.type == TOK_PLUS )
                    codegen_emit(cg, "\taddl\t");
                if( subtree->tok.type == TOK_MINUS )
```

```
                        codegen_emit(cg, "\tsubl\t");
                    if( subtree->tok.type == TOK_PIPE )
                        codegen_emit(cg, "\torl\t");

                    // op 2nd -> 1st
                    codegen_emit(cg, result2);
                    codegen_emit(cg, ", ");
                    codegen_emit(cg, result);
                    codegen_emit(cg, "\n");

                    //Save new 1st
                    codegen_emit(cg, "\tpushl\t");
                    codegen_emit(cg, result);
                    codegen_emit(cg, "\n");


                    //Goto next operand
                    idx = idx+2;

                    //Remove the old type
                    syntax_FreeSyntaxTree(oldtype);
        }


        //Get result
        codegen_emit(cg, "\tpopl\t%eax\n");

        strcpy(result, "%eax");
        return type;
}
```

Please note that logical expressions can be handled by this scheme too. But since lazy evaluation is applied to logical expressions more complex calculations are not needed. Therefore the necessity of registers is dropped and the full intelligence lies in conditional jumps.

Efficiency of register allocation The register allocation principle used above is not very efficient in the sense of making use of as many registers as possible. First of all implementing an efficient register allocation algorithm can be quite complicated. The whole code-generation code has to keep track of which registers are used and which are free. Furthermore, intelligent decisions have to be made whether to swap out registers onto the stack.

Secondly, the IA-32 architecture provides only 4 general purpose registers anyway. This means, the procedure above uses registers `eax` and `ebx` in most cases. For divisions register `ecx` and `edx` are needed too and for some operations `esi` is used additionally. So what could be optimized is a more balanced usage of `ecx` and `edx`. The situation would be different if the basic machine provided 30 registers and more, like the DLX.

7 Linker

The linkers main task is to resolve addresses. The code-generation uses so called "markers" to realize function calls, jumps and references to global variables and text strings. In order to have the VM executed, this code properly these markers have to be resolved. So the linker offers an API which allows multiple assembler files to be linked to one executable that can be executed by the "hrwvm" (please refer to chapter 9.4.1).

When adding a single file to the linker, the latter parses this file and collects all marker statements and assigns each instruction an unique address. Since the virtual machine stores the instructions in a special way, each instruction only increases the address counter by one. The linker manages two counters: A "text_addr" counter that counts the instructions (.text section) and a "data_addr" counter that counts the sizes of global variables (.data section). This is necessary since every input file contains these two sections and the first data address in the global output file can only be calculated if all instructions of all input files are counted. Since the linker cannot start creating an output file until all files are parsed every single source line will be stored in special lists to be written later on.

Default marker statements are only visible in the currently parsed file and are therefore stored to each file. To export markers into other files (i.e. make functions callable from other source files) there exist ".globl" statements which are stored in a special list that contains all globl markers of all files. After having collected all markers and resolved all instruction and data addresses, the linker loops through the list containing the globl markers and assigns each of them the now known address.

When all files are added and a special function, which takes the output file name as argument, is called the linker goes ahead and starts writing the executable. The first thing that has to be written is the so called "kickoff code". The kickoff code is the code that calls the "main" function and exits the program after its execution (return from main). To create the kickoff code the linker makes sure an "entry point" (a globl marker with the name "main") was defined. If the entry point is found the linker generates the following code:

```
.section .text

call ADDRESS_OF_MAIN
pushl %eax
call exit
```

where "ADDRESS_OF_MAIN" is the address of the marker that represents the main function. After the vm has executed the main function and returns from it, the return value of main (stored in the register %eax) is pushed onto the stack to be an argument of the libc function "exit" which is called afterwards to terminate the program and the virtual machine.

The next step is to loop through all instructions and replace all markers by their addresses and write them to the outputfile, too. The text section is followed by the data section and the linker produces the line

```
.section .data
```

followed by all code lines of all data sections. No replacements or any special treatment is done in the data section.

In order to make the produced executable code more readable, the linker offers a very nifty "debugging flag" which carries all commented lines (from the code generation) into the output file as well. Once this debugging flag is set, all lines containing a marker definition will be commented out (setting a "#" symbol at the first position) and so the executable code is really readable and understandable. If the debugging flag is not set, all comments will be thrown out.

7.1 Libc functions

Please note that the handling of libc functions is described in greater detail in the virtual machine chapter since the vm has to do the actual call. The only important thing about libc functions and the linker is that the linker may want to replace them since they look like markers.

In order to produce some output on stdout, a program may use "printf" which is generated to the following code segment:

```
pushl argn
...
pushl arg1
call printf
```

The idea is that "printf" is then wrapped by the VM and so no special worries about IO etc is necessary. As mentioned above, the linker may want to replace "printf" since it looks like a marker but mostly there won't be an address for it in the markers list. So the linker would produce an error informing the user that a marker could not be resolved. In order to avoid this, the file `include/hrwcccomp.h` is used which offers a list of all libc functions. So all the linker does is to execute the complete compiling stage (preprocessor, scanner, parser, symbol table) during initialization and stores a list of these special functions. Whenever a marker might be replaced, the special functions list is searched through to make sure no special functions are replaced or a linker error occurs (to inform the user that a marker could not be resolved).

8 Optimization

We implemented some simple optimizations in HrwCC. In general, optimizations take place at several stages in the compilation process. We implemented some optimizations at assembler level and at code generation level.

8.1 Assembler Pattern Optimization

The Assembler Pattern Optimizer (asmopt) is directly called from the hrwcc binary after the compilation process (if optimization "fasmopt" ist set) is successfully completed. The "asmopt" algorithm receives a filename as input and uses a so called code buffer which is basically a "window" over a certain number of lines in the input file. The code buffer is a fixed size array where each entry contains a list of tokens which represent a line. To manipulate the code buffer there exist several functions like to pop an element from the buffer (remove the element and move all other elements one position up) or to fill the buffer with new lines from the input file. The code buffer is then handled by (at the moment two) optimization functions which try to recognize patterns they can optimize.

8.1.1 Push-Pop Sequences

The push-pop optimizer loops through every entry in the code buffer and looks for line-sequences where two lines of code (represented through tokens in the code buffer) are analyzed. If the first line starts with the "pushl" and the following line with the "popl" command, the rule might be applied. Whenever such a sequence is detected, the optimizer has to make sure that at least one of the operands in one of these two lines starts with a "$" or a "%" token. If both lines are i.e. with indirect addressing the optimization can not be applied because this operation is not permitted in GNU assembler, which is our output language. If the lines pass this rule, the optimization can be applied:

The sequence

```
pushl %eax
popl %eax
```

is completely removed by the optmizer since it has no effect. If, however, the operands differ:

```
pushl %eax
popl %ebx
```

the optimizer will generate

```
movl %eax, %ebx
```

which does the same thing but faster.

8.1.2 Jump Sequences

The jump optimizer optimizes jump statements that are never reached. This can be explained best by looking at an example:

```
jmp some_marker
jnz some_other_marker
```

In this example the second code statement will never be executed since "jmp" is no conditional statement and will always be executed. So the optimizer looks for code sequences where a "jmp" statement is detected, followed by a line that starts with a "j*" command. If such a sequences is detected, the second statement will be removed.

8.2 Variable Expressions

In HrwCC, access to variables is implemented as general as needed without any shortcuts. So accessing a variable `i` is done in the same way as `& a->b.c->d[2]`. However, accessing a simple variable, like `i`, can be done very efficiently. This can be implemented in two ways:

- Efficient assignments
- Efficient access to the value of a variable

The assignment optimizations can be summarized to the following listening. Assignments to a simple variable of a simple array-element is done by more-or-less one assembler statement.

Listing 10: Some efficient patterns for variable expressions

```
;int g;
;int g2[1];
;
;void func( int p )
;{
;    int l;
;    int l2[1];
;
;    g = 1;
;    p = 2;
;    l = 3;
;
;    g2[0] =4;
;    l2[0] = 5;
;}


     movl    $1, symtab+0
     movl    $2, 8(%ebp)
     movl    $3, -4(%ebp)

     movl    $0,%esi
     movl    $4,symtab+4(,%esi,4)
     movl    $0,%esi
     movl    $5,-8(%ebp,%esi,4)
```

The following listening contains optimization patterns for accessing the value of simple variable expressions. The code below is generated with optimization.

Listing 11: Some efficient patterns for variable access

```
;int g;
;
```

```
;void func( int p )
;{
;        int d;
;        int l;
;
;        d = g;
;        d = p;
;        d = l;
;}

        movl    symtab+0,%eax
        movl    %eax,-4(%ebp)

        movl    8(%ebp),%eax
        movl    %eax,-4(%ebp)

        movl    -8(%ebp),%eax
        movl    %eax,-4(%ebp)
```

9 Virtual Machine

The Virtual Machine is written in `C++` which offers the opportunity to use object orientation. In the following we describe the basic functionality of the VM.

9.1 Mode of Operation

Basically, the VM parses a nearly pure Assembler file which we previously generated. Each line of the text and data section is handled seperately. Let's first focus on the the data section and on its elements which need to be stored in our virtual memory 9.2. Basically, the VM is looking for the following statements:

- `.string` which represents a simple string. For example `.string "result: %d"` could be stored and later on called with printf where the result is already lying on the stack. For every string its size has to be added to the already calculated size of the data section and a "
0" has to be appended.
- `.byte` is very simple to handle. The VM just has to store one byte in virtual memory.
- `.long` is nearly the same as `.byte` but stores a long which is similar to store four bytes so in other words, calling the setl function of the memory class just executes the setb function four times.
- `.rept` together with `.endr` which is kind of tricky to handle. The procedure of how to deal with `.rept` is described below.

At this point it is nessecary to handle `.rept` sections where the following lines may occur:

Listing 12: Occurrence of .rept in data section

```
.rept 17
    .byte 0
.endr
```

In this very simple case we just have to store 17 bytes but if this gets more complicated we have to be careful. After storing all lines of `.rept` the first time we need to prove: Is there still another storing cycle to run? Does the counted size of bytes, longs etc. conform to the original size? To handle this situation correctly we store all lines of `.rept` until the corresponding `.endr` is detected and afterwards we start storing all elements into virtual memory as often as nesseccary.

Every single instruction line of the text section is first parsed into tokens. Let's start with a simple example and take a look at the following instruction line:

Listing 13: Sample instruction line

```
movl    $2    4(%eax)
```

After parsing this line, the VM creates a token vector with the following tokens for each operand (in this case two):

Listing 14: Sample tokens

```
$
2

4
(
%
eax
)
```

Now when executing these instruction lines it makes it easier for the cpu to operate with the correct value on the right register. The VM has a socalled instruction vector with instruction objects as elements. An instruction object consists of an opcode and two operands at most. For example, while `addl` requires two parameters, `divl` just needs one. When finished parsing the execution of the instruction objects starts and control is turned over to the CPU 9.4.

9.2 Virtual Memory

In our VM we decided to simulate a 4GB address space as follows. We group the 4GB into 4096 slots (`byte *memory[4096]`) each carrying 1024kB. So functions like `getb` or `setb` which offer to access one byte in memory, first have to find the right slot and the block. Furthermore, handling integers we take care of socalled slot-boundries not to cause unused space or corrupt data where there shouldn't be. We also implemented the functions `malloc` and `free` which aquire to store a vector of markers to assign used and unused space in memory. Whenever `malloc` is called, we search the vector, which carries nodes of free space, and insert a new one where the request amount of space fits, afterwards this address is returned. The stack is integrated in this memory and takes up 2MB (=2 slots) at the start of execution. Command line arguments lie on the stack as well.

9.3 Registers

For the registers we use a simple array with register unions which we designed especially to operate on single bytes. The `%eax` register, for example, would look like this:

Listing 15: Representation of a register struct.

```
typedef union {
    /** the whole value */
    int eax;
    /** last two bytes */
    short ax;
    /** last two single bytes */
    char ah_al[2];
} Register;
```

As a special register we have the `%eflags` register which carries several bits like zf, df etc. that are set after instructions like, for example, `cmpl`. We implemented this register as a struct which just carries a bitflag. To show the purpose of this special register consider this pseudo code for the instruction `cmpl op1 op2`:

Listing 16: Use of %eflags

```
if ( (op2 - op1) == 0 )
{
    eflags.zf = 1; // set the zero flag
    eflags.sf = 0; // unset the sign flag
}
else if ( (op2 - op1) < 0 )
{
    eflags.zf = 0;
    eflags.sf = 1;
}
else
{
    eflags.zf = 0;
    eflags.sf = 0;
}
```

9.4 CPU

The CPU is responsible for executing single instructions of the instruction vector. It is the core piece of our VM which connects virtual memory, registers and instructions. For each instruction object the options are parsed according to addressing modes. The following modes are handled:

Addressing Mode	Example
Immediate mode	`$12`
Direct addressing	`%eax`
Indirect addressing	`(%eax)`
Basepointer addressing	`4(%eax)`
Indexed addressing	`10(,%eax,1)`

Depending on the instruction stack, register or memory opertations are performed. Another essential job is achieved by the CPU, namely, function wrapping.

9.4.1 Function Wrapping

Instructions like `call puts` and `call printf` for example, are handled by the CPU. The CPU fetches arguments form the stack or loads them from virtual memory and then executes the real libc functions. To do so, the CPU has functions to, for example, load stored strings from the virtual memory.

A Application

A.1 Usage of HrwCC

In order to simplify the usage of the compilation-chain a compiler-frontend was developed. It can be found in the subdirectory "hrwcc" where a binary called "hrwcc" can be created calling "make".

The usage of hrwcc is similar to the usage of the "gcc" binary (clearly, hrwcc does not support the whole set of flags which are offered by gcc). Usage information is shown when hrwcc is called with none or wrong parameters.

It is worth notice that the hwrcc binary observes the given file extensions in order to apply the appropriate compiling stages. This means that passing a ".cpp" file will execute the preprocessor, scanner, parser, symboltable and codegeneration while passing a ".s" file will only result in adding this file to the linker. Hrwcc also allows to "mix up" these file extensions in one call: i.e. `hrwcc file1.cpp file2.s file3.cpp` is a valid call.

Bootstrap In the compilers root directory one can find a little shell script called `bootstrap.sh`. This file executes a bootstrap process in the following way: The first step is to compile the hrwcc source code with `g++`. The resulting hrwcc binary compiles its own sourcecode in the second step and a new (self-compiled) hrwcc binary is created. The second step is repeated to generate a third generation of the hrwcc binary. In order to verify the correctness the MD5 sums of the last two binaries have to match.

A.2 Testing

We applied fuzz-testing to HrwCC. For this purpose we build a fuzz-testing tool called "fuzzer" which works as follows: It reads from stdin, character by character, and modifies a specific amount before writing to stdout. The modification rate can be adjusted by a parameter.

The tool can be used in combination with Unix-pipes to manipulate C-source code which is further compiled by HrwCC. With lower modification rates we can test the error handling capabilities of HrwCC. With higher modification rates we test HrwCC in the original sense of fuzz-testing.

The big advantage, in contrast to pure randomization, is that parts of the structure of C-input files is kept. Therefore, special cases and conditions can be tested more efficiently. For pure randomized input the parser will just jump in a few steps through the character salad. At [1] you can find further information on fuzz testing and fuzz testing software.

Test source At [4] you can find a `features.c` which works like a Unit-testfile for our compiler to test basic funktionality. You may want to inspect this file to get an impress what our compiler supports. This file can be also found in `<HrwCC-root>/hrwcc/testdata/features.c`.

For a user-friendly presentation you may want to execute `hrwvm-demo.sh` in the root of the project directory. This demo builds hrwcc, a self-compiled hrwcc (to get the hrwvm input) and the VM itself. After that the features file is build with the ELF-binary of hrwcc and with the hrwvm executed hrwcc. After that a MD5 sum checks that the output of both are equal. Finally the hrwvm-executed hrwcc-output is executed – with hrwvm.

A.3 Contributions

One of the key steps when writing a compiler is the E-BNF. Since a single person can forget important elements we decided that every team member creates its own E-BNF. These three E-BNFs were analyzed together and the best concepts and elements have been merged to the currently implemented one.

Since we chose the well defined GNU assembler code as output language we were able to parallelize the development process. Stefan Walkner started to implement the preprocessor while Christian Rathgeb and Stefan Huber implemented the Scanner. After that Christian Rathgeb started to implemented the Virtual Machine afterwards and Stefan Huber started implementing the parser and the symbol table. The code generation as a central part have been implemented more or less in a joint manner while Stefan Walkner implemented the linker.

B Language

B.1 Parser

```
<program>           ::= { <struct_def> | <varfunc_defdec> } <EOF>

<struct_def>        ::= "struct" <ident> "{" <var_decl> {<var_decl>} "}" ";"
<varfunc_defdec>    ::= <typed_ident>  ( [ "[" <number> "]" ] ";" |
                                        <arg_list_def> ( ";" | <func_body> ))
<arg_list_def>      ::= "(" [ <arg_def>  { "," <arg_def> } ] ")"
<arg_def>           ::= <typed_ident>
<func_body>         ::= "{" {<var_decl} {<statement>} "}"
<var_decl>          ::= <typed_ident> [ "[" <number> "]" ] ";"
<typed_ident>       ::= <data_type> <ident>

<statement>         ::= <assign_stmt> | <func_call_stmt> | <if_stmt> |
                        <while_stmt> | <return_stmt> | <break_stmt> |
                        <continue_stmt>
<func_call_stmt>    ::= <func_call> ";"
<assign_stmt>       ::= <variable_expr> "=" <log_expr> ";"
<if_stmt>           ::= "if" "(" <log_expr> ")" <stmt_block>
                        [ "else" <stmt_block> ]
<while_stmt>        ::= "while" "(" <log_expr> ")" <stmt_block>
<stmt_block>        ::= "{" { <statement> } "}" | <statement>
<return_stmt>       ::= "return" [<log_expr>] ";"
<break_stmt>        ::= "break" ";"
<continue_stmt>     ::= "continue" ";"

<func_call>         ::= <ident> "(" [ <log_expr> { "," <log_expr> } ] ")"
<variable_expr>     ::= ["*" | "&"] <ident> { ("->" | ".") <ident> }
                        [ "[" <log_expr> "]" ]
<sizeof_expr>       ::= "sizeof" "(" <data_type> ")"

<atomic_val_expr>   ::= <variable_expr> | <string> | <number> |
                        <singlechar> | <func_call> | <sizeof_expr>
<arith_factor>      ::= ["~"] <atomic_val_expr> | "(" <log-expr> ")"
<arith_term>        ::= <arith_factor>
                        { ("&" | "^" | "*" | "/" | "%" ) <arith_factor> }
<arith_expr>        ::= ["-"] <arith_term> { ("+", "-", "|") <arith_term> }
<rel_expr>          ::= <arith_expr>
                        [ ("==", "!=", "<", "<=", ">", ">=") <arith_expr> ]
<log_factor>        ::= ["!"] <rel_expr>
<log_term>          ::= <log_factor> { "&&" <log_factor> }
<log_expr>          ::= <log_term> { "||" <log_term> }

<data_type>         ::= ( "int" | "char" | "void" | <ident>) {"*"}
```

B.2 Scanner

```
<ident>             ::= (<alpha> | "_") { <alphanum> | "_" }
```

```
<number>            ::= <digit> { <digit> }
<string>            ::= """ { <stringchar> } """
<singlechar>        ::= "'" { <stringchar> } "'"
<stringchar>        ::= everything but " and ' (escaping)
```

Tokens

```
{ } ( ) [ ] , ; . -> =  + - * / % & | ^ && || == != < > <= >= ~ !
if else while return struct const int char void sizeof break continue
<ident> <number> <string> <singlechar> <EOF>
```

B.3 Pre-Processor

```
#define <ident> [some-text]
#define <ident> "(" arg_1 "," ... "," arg_n ")" some-text
#ifdef <ident> [some-text] #endif
#ifndef <ident> [some-text] #endif
#include "<" some-text ">"
#include """ some-text """

//comment for rest of line
/* blocked comment */
```

References

[1] Fuzz testing website. `http://www.cs.wisc.edu/~bart/fuzz/fuzz.html`, 2007.

[2] J. Bartlett. *Programming from the Ground Up.* 2003.

[3] Intel ®. *Intel ®64 and IA-32 Architectures Software Developer's Manual: Volume 1, Basic Architecture*, 2006. Order Number: 253665-021.

[4] Stefan Huber, Christian Rathgeb, Stefan Walkner. Hrwcc website. `http://www.cs.uni-salzburg.at/~ck/wiki/index.php?n=CC-Summer-2007.HrwCC`, 2007.

[5] N. Wirth. *Compiler Construction.* Addison Wesley, 1996. ISBN-10 0-201-40353-6.

Index