Principles of Compiler Design

*- The Brainf*ck Compiler -*

Clifford Wolf - www.clifford.at

http://www.clifford.at/papers/2004/compiler/

Introduction

Introduction

- My presentation at 20C3 about CPU design featuring a Brainf*ck CPU was a big success
- My original plan for 21C3 was to build a Brainf*ck CPU with tubes..
- But:
  The only thing more dangerous than a hardware guy with a code patch is a programmer with a soldering iron.
- So this is a presentation about compiler design featuring a Brainf*ck Compiler.

Overview (1/2)

In this presentation I will discuss:

- A little introduction to Brainf*ck
- Components of a compiler, overview
- Designing and implementing lexers
- Designing and implementing parsers
- Designing and implementing code generators
- Tools (flex, bison, iburg, etc.)

LINBIT Information Technologies GmbH

Overview (2/2)

- Overview of more complex code generators
  - Abstract syntax trees
  - Intermediate representations
  - Basic block analysis
  - Backpatching
  - Dynamic programming
  - Optimizations
- Design and implementation of the Brainf*ck Compiler
- Implementation of and code generation for stack machines
- Design and implementation of the SPL Project
- Design and implementation of LL(regex) parsers

Aim

- After this presentation, the auditors ..
- .. should have a rough idea of how compilers are working.
- .. should be able to implement parsers for complex configuration files.
- .. should be able to implement code-generators for stack machines.
- .. should have a rough idea of code-generation for register machines.

Brainf*ck

Overview

- Brainf*ck is a very simple turing-complete programming language.
- It has only 8 instructions and no instruction parameters.
- Each instruction is represented by one character:
  `< > + - . , [ ]`
- All other characters in the input are ignored.
- A Brainfuck program has an implicit byte pointer which is free to move around within an array of 30000 bytes, initially all set to zero. The pointer itself is initialized to point to the beginning of this array.

Some languages are designed to solve a problem.
Others are designed to prove a point.

Instructions

>	Increment the pointer. `++p;`
<	Decrement the pointer. `--p;`
+	Increment the byte at the pointer. `++*p;`
–	Decrement the byte at the pointer. `++*p;`
.	Output the byte at the pointer. `putchar(*p);`
,	Input a byte and store it in the byte at the pointer. `*p = getchar();`
[	Jump forward past the matching] if the byte at the pointer is zero. `while (*p) {`
]	Jump backward to the matching [unless the byte at the pointer is zero. }

Implementing "while"

- Implementing a while statement is easy, because the Brainf*ck `[ .. ]` statement is a while loop.
- So `while (x) { <foobar> }` becomes:

```
<move pointer to a>
[
<foobar>
<move pointer to a>
]
```

Implementing "x=y"

- Implementing assignment (copy) instructions is a bit more complex.
- The straight forward way of doing that resets y to zero:

```
<move pointer to y> [ -
<move pointer to x> +
<move pointer to y> ]
```

- So, a temporary variable t is needed:

```
<move pointer to y> [ -
<move pointer to t> +
<move pointer to y> ]

<move pointer to t> [ -
<move pointer to x> +
<move pointer to y> +
<move pointer to t> ]
```

Implementing "if"

- The if statement is like a while-loop, but it should run its block only once. Again, a temporary variable is needed to implement `if (x) { <foobar> }`:

```
<move pointer to x> [ -
<move pointer to t> +
<move pointer to x> ]

<move pointer to t> [

    [ -
    <move pointer to x> +
    <move pointer to t> ]

    <foobar>

<move pointer to t> ]
```

Functions

- Brainf*ck has no construct for functions.
- The compiler has support for macros which are always inlined.
- The generated code may become huge if macros are used intensively.
- So recursions must be implemented using explicit stacks.

Lexer and Parser

Lexer

- The lexer reads the compiler input and transforms it to lexical tokens.
- E.g. the lexer reads the input "`while`" and returns the numerical constant `TOKEN_WHILE`.
- Tokens may have additional attributes. E.g. the textual input "`123`" may be transformed to the token `TOKEN_NUMBER` with the integer value 123 attached to it.
- The lexer is usually implemented as function which is called by the parser.

Parser

- The parser consumes the lexical tokens (terminal symbols) and reduces sequences of terminal and non-terminal symbols to non-terminal symbols.
- The parser creates the so-called parse tree.
- The parse tree never exists as such as memory-structure.
- Instead the parse-tree just defines the order in which so-called reduction functions are called.
- It is possible to create tree-like memory structures in this reduction functions which look like the parse tree. This structures are called "Abstract Syntax Tree".

BNF

BNF (Backus-Naur Form) is a way of writing down parser definitions. A BNF for parsing a simple assign statement (like "`x = y + z * 3`") could look like (yacc style syntax):

```
assign: NAME '=' expression;

primary: NAME | NUMBER
|        '(' expression ')';

product: primary
|        product '*' primary
|        product '/' primary;

sum: product
|    sum '+' product
|    sum '-' product;

expression: sum;
```

Reduce Functions

- Whenever a sequence of symbols is reduced to a non-terminal symbol, a reduce function is called. E.g.:

```
%union {
        int numval;
}
%type <numval> sum product

%%

sum: product
|    sum '+' product    { $$ = $1 + $3; }
|    sum '-' product    { $$ = $1 + $3; };
```

- The attributes of the symbols on the right side of the reduction can be accessed using $1 .. $*n*. The attributes of the resulting symbol can be accessed with $$.

Algorithms

- A huge number of different parser algorithms exists.
- The two most important algorithms are LL(N) and LALR(N).
- Other algorithms are LL(k), LL(regex), GLR and Ad-Hoc.
- Most hand written parsers are LL(1) parsers.
- Most parser generators create LALR(1) parsers.
- A detailed discussion of various parser algorithms can be found in “The Dragonbook” (see references on last slide).
- The design and implementation of LL(1) parsers is also discussed in the section about LL(regex) parsers.

Conflicts

- Sometimes a parser grammar is ambiguous.
- In this cases, the parser has to choose one possible interpretation of the input.
- LALR parsers distinguish between reduce-reduce and shift-reduce conflicts.
- Reduce-reduce conflicts should be avoided when writing the BNF.
- Shift-reduce conflicts are always solved by shifting.

Code Generators

Overview

- Writing the code generator is the most complex part of a compiler project.
- Usually the code-generation is split up in different stages, such as:
  - Creating an Abstract-Syntax tree
  - Creating an intermediate code
  - Creating the output code
- A code-generator which creates assembler code is usually much easier to write than a code-generator creating binaries.

Simple Code Generators

- Simple code generators may generate code directly in the parser.
- This is possible if no anonymous variables exist (BFC) or the target machine is a stack-machine (SPL).

Example:

```
if_stmt:
  TK_IF TK_ARGS_BEGIN TK_STRING TK_ARGS_END stmt
{
  $$ = xprintf(0, 0, "%s{", debug_info());
  $$ = xprintf($$, $5, "(#tmp_if)<#tmp_if>[-]"
                       "<%s>[-<#tmp_if>+]"
                       "<#tmp_if>[[-<%s>+]\n", $3, $
  $$ = xprintf($$, 0, "]}");
}
```

Tools

Overview

- There are tools for writing compilers.
- Most of these tools cover the lexer/parser step only.
- Most of these tools generate c-code from a declarative language.
- Use those tools but understand what they are doing!

Flex / Lex

- Flex (Fast Lex) is the GNU successor of Lex.
- The lex input file (`*.l`) is a list or regular expressions and actions.
- The “actions” are c code which should be executed when the lexer finds a match for the regular expression in the input.
- Most actions simply return the token to the parser.
- It is possible to skip patterns (e.g. white spaces) by not providing an action at all.

Yacc / Bison

- Bison is the GNU successor of Yacc (Yet Another Compiler Compiler).
- Bison is a parser generator.
- The bison input (`*.y`) is a BNF with reduce functions.
- The generated parser is a LALR(1) parser.
- Bison can also generate GLR parsers.

Burg / iBurg

- iBurg is the successor of Burg.
- iBurg is a “Code Generator Generator”.
- The code generator generated by iBurg implements the “dynamic programming” algorithm.
- It is a bit like a parser for an abstract syntax tree with an extremely ambiguous BNF.
- The reductions have cost values applied and an iBurg code generator chooses the cheapest fit.

PCCTS

- PCCTS is the “Purdue Compiler-Compiler Tool Set”.
- PCCTS is a parser generator for LL(k) parsers in C++.
- The PCCTS toolkit was written by Terence J. Parr of the MageLang Institute.
- His current project is antlr 2 - a complete redesign of pccts, written in Java, that generates Java or C++.
- PCCTS is now maintained by Tom Moog, Polhode, Inc.

Complex Code Generators

Overview

- Unfortunately it's not possible to cover code generation in depth in this presentation.
- However, I will try to give a rough overview of the topic and explain the most important terms.

Abstract syntax trees

- With some languages it is hard to create intermediate code directly from the parser.
- In compilers for such languages, an abstract syntax tree is created from the parser.
- The intermediate code generation can then be done in different phases which may process the abstract syntax tree bottom-up and top-down.

Intermediate representations

- Most compilers create intermediate code from the input and generate output code from this intermediate code.
- Usually the intermediate code is some kind of three-address code assembler language.
- The GCC intermediate language is called RTL and is a wild mix of imperative and functional programming.
- Intermediate representations which are easily converted to trees (such as functional approaches) are better for dynamic programming, but are usually not optimal for ad-hoc code generators.

Basic block analysis

- A code block from one jump target to the next is called “Basic Block”.
- Optimizations in basic blocks are an entirely different class of optimization than those which can be applied to a larger code block.
- Many compilers create intermediate language trees for each basic block and then create the code for it using dynamic programming.

Backpatching

- It is often necessary to create jump instructions without knowing the jump target address yet.
- This problem is solved by outputting a dummy target address and fixing it later.
- This procedure is called backpatching.
- The Brainf*ck compiler doesn't need backpatching because Brainf*ck doesn't have jump instructions and addresses.
- However, the Brainf*ck runtime bundled with the compiler is using backpatching to optimize the runtime speed.

Dynamic programming

- Dynamic programming is an algorithm for generating assembler code from intermediate language trees.
- Code generators such as Burg and iBurg are implementing the dynamic programming algorithm.
- Dynamic programming uses two different phases.
- In the first phase, the tree is labeled to find the cheapest matches in the rule set (bottom-up).
- In the 2nd phase, the code for the cheapest solution is generated (top-down).

Optimizations

- Most optimizing compilers perform different optimizations in different compilation phases.
- So most compilers don't have a separate "the optimizer" code path.
- Some important optimizations are:
  - Global register allocation
  - Loop detection and unrolling
  - Common subexpression elimination
  - Peephole optimizations
- The Brainf*ck compiler does not optimize.
- The SPL compiler has a simple peephole optimizer.

The BF Compiler

Overview

- The project is split up in an assembler and a compiler.
- The assembler handles variable names and manages the pointer position.
- The compiler reads BFC input files and creates assembler code.
- The assembler has an ad-hoc lexer and parser.
- The compiler has a flex generated lexer and a bison generated parser.
- The compiler generates the assembler code directly from the parser reduce functions.

Assembler

- The operators `[` `+` and `-` are unmodified.
- The `]` operator sets the pointer back to the position where it was at `[`.
- A named variable can be defined with `(x)`.
- The pointer can be set to a named variable with `<x>`.
- A name space is defined with `{ ... }`.
- A block in single quotes is passed through unmodified.
- Larger spaces can be defined with `(x.42)`.
- An alias for another variable can be defined with `(x:y)`.

Compiler

- Variables are declared with `var x;`.
- C-like expressions for `=`, `+=`, `-=`, `if` and `while` are available.
- Macros can be defined with `macro x() { ... }`.
- All variables are passed using call-by-reference.
- The compiler can't evaluate complex expressions.
- Higher functions (such as comparisons and multiply) are implemented using built-in functions.

Running

- The compiler and the assembler are both filter programs.
- So compilation is done by:

```
$ ./bfc < hanoi.bfc | ./bfa > hanoi.bf
Code: 53884 bytes, Data: 275 bytes.
```

- The `bfrun` executable is a simple Brainf*ck interpreter:

```
$ ./bfrun hanoi.bf
```

Implementation

Code review of the assembler.

.. and the compiler.

.. and the built-ins library.

.. and the hanoi example.

Stack Machines

Overview

- Stack machine are a computer architecture, like register machines or accumulator machines.
- Every instruction pops it's arguments from the stack and pushes the result back on the stack.
- Special instructions push the content of a variable on the stack or pop a value from the stack and write it back to a variable.
- Stack machines are great for virtual machines in scripting languages because code generation is very easy.
- However, stack machines are less efficient than register machines and are harder to implement in hardware.

Example


```
x = 5 * ( 3 + y );
```

```
PUSHC "5"
PUSHC "3"
PUSH  "y"
IADD
IMUL
POP   "x"
```

The SPL Project

Overview

- SPL is an embeddable scripting language with C-like syntax.
- It has support for arrays, hashes, objects, perl regular expressions, etc. pp.
- The entire state of the virtual machine can be dumped at any time and execution of the program resumed later.
- In SPL there is a clear separation of compiler, assembler, optimizer and virtual machine.
- It's possible to run pre-compiled binaries, program directly in the VM assembly, use multi threading, step-debug programs, etc. pp.
- SPL is a very small project, so it is a good example for implementing high-level language compilers for stack machines.

WebSPL

- WebSPL is a framework for web application development.
- It creates a state over the stateless HTTP protocol using the dump/restore features of SPL.
- I.e. it is possible to print out an updated HTML page and then call a function which “waits” for the user to do anything and returns then.
- WebSPL is still missing some bindings for various SQL implementations, XML and XSLT bindings, the WSF (WebSPL Forms) library and some other stuff..
- Right now I’m looking for people who want to participate in the project.

LINBIT Information Technologies GmbH

Example


```
object Friend {
        var id;
<...>
        method winmain(sid) {
                title = name;
                .sid = sid;
                while (1) {
                        template = "show";
                        bother_user();
                        if ( defined cgi.param.edit ) {
                                template = "edit";
                                bother_user();
                                name = cgi.param.new_name;
                                phone = cgi.param.new_phone;
                                email = cgi.param.new_email;
                                addr = cgi.param.new_addr;
                                title = name;
                        }
                        if ( defined cgi.param.delfriend ) {
                                delete friends.[id].links.[cgi.param.delfriend];
                                delete friends.[cgi.param.delfriend].links.[id];
                        }
                        if ( defined cgi.param.delete ) {
                                delete friends.[id];
                                foreach f (friends)
                                        delete friends.[f].links.[id];
                                &windows.[winid].finish();
                        }
                }
        }
}
```

LL(regex) parsers

Overview

- LL parsers (recursive decent parsers) are straight-forward implementations of a BNF.
- Usually parsers read lexemes (tokens) from a lexer.
- A LL(N) parser has access to N lookahead symbols to decide which reduction should be applied.
- Usually LL(N) parsers are LL(1) parsers.
- LL(regex) parsers are LL parsers with no lexer but a regex engine.
- LL(regex) parsers are very easy to implement in perl.

LINBIT Information Technologies GmbH

Left recursions

- Often a BNF contains left recursion:

```
<...>
product: primary
|        product '*' primary
|        product '/' primary;
<...>
```

- Left recursions cause LL parsers to run into an endless recursion.
- There are algorithms for converting left recursions to right recursions without effecting the organization of the parse tree.
- But the resulting BNF is much more complex than the original one.
- Most parser generators do that automatically (e.g. bison).

Example

Code review of llregex.pl.

http://www.clifford.at/papers/2004/compiler/llregex.pl

URLs and References

URLs and References

- My Brainf*ck Projects:
  http://www.clifford.at/bfcpu/
- The SPL Project:
  http://www.clifford.at/spl/
- Clifford Wolf:
  http://www.clifford.at/
- “The Dragonbook”
  Compilers: Principles, Techniques and Tools
  by Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman
  Addison-Wesley 1986; ISBN 0-201-10088-6
- LINBIT Information Technologies
  http://www.linbit.com/

http://www.clifford.at/papers/2004/compiler/