

# Implementing a Scripting Engine\*

Jan Niestadt<sup>†</sup>

May 7, 1999

---

\*This article can be viewed online at <http://www.flipcode.com>

<sup>†</sup>This document was typeset on February 18, 2004 by Benjamin A. Collins

# Contents

<b>1 Overview</b>	<b>3</b>
1.1 Introduction . . . . .	3
1.2 Required Reading . . . . .	4
1.3 What, Why, and How . . . . .	5
1.4 The Lexer Rules! . . . . .	5
1.5 It Gets Better, Really . . . . .	8
1.6 Quote . . . . .	8
<b>2 The Parser</b>	<b>8</b>
2.1 Introduction . . . . .	8
2.2 A Little Language Theory . . . . .	9
2.3 Looks Familiar! . . . . .	10
<b>3 Whew!</b>	<b>12</b>
3.1 Quote . . . . .	12
<b>4 The Symbol Table &amp; Syntax Tree</b>	<b>13</b>
4.1 Introduction . . . . .	13
4.2 Passing Information Between Rules . . . . .	14
4.3 The Symbol Table . . . . .	14
4.4 The Syntax Tree . . . . .	15
4.5 That's Pretty Cool, But... . . . .	15
4.6 Quote . . . . .	16
<b>5 The Semantic Checker &amp; Intermediate Code Generator</b>	<b>16</b>
5.1 Introduction . . . . .	16
5.2 Checking the Semantics . . . . .	16
5.3 Generating Intermediate Code . . . . .	17
5.4 Whoa, what happened? . . . . .	20
5.5 Quote . . . . .	20
<b>6 Optimization</b>	<b>20</b>
6.1 Did you Spot The Bug? . . . . .	20
6.2 Introduction . . . . .	21
6.3 Extra Opcodes . . . . .	21
6.4 Code Transformations . . . . .	22
6.4.1 Algebraic identities . . . . .	23
6.4.2 Common subexpression elimination . . . . .	23
6.4.3 Loop optimizations . . . . .	23
6.4.4 Jump elimination . . . . .	24
6.5 Coming Up Next... . . . .	25
6.6 Quote . . . . .	25

<b>7</b>	<b>The Virtual Machine</b>	<b>25</b>
7.1	Introduction . . . . .	25
7.2	Machine Types . . . . .	26
7.3	A Virtual Piece of Cake . . . . .	26
7.4	Next Time . . . . .	27
7.5	Quote . . . . .	27
<b>8</b>	<b>Executable Code</b>	<b>28</b>
8.1	Introduction . . . . .	28
8.2	The Final Step . . . . .	28
8.3	It works! I can't believe it! . . . . .	29
8.4	So what happens now? . . . . .	29
8.5	Quote . . . . .	29
<b>9</b>	<b>Advanced Subjects</b>	<b>29</b>
9.1	Introduction . . . . .	29
9.2	A lockup-resistant VM . . . . .	30
9.3	Functions . . . . .	30
9.4	Quote . . . . .	30
9.5	Overloading . . . . .	31
9.6	Classes . . . . .	31
9.6.1	Inheritance . . . . .	31
9.6.2	Virtual functions . . . . .	32
9.6.3	Dynamic casts . . . . .	32
9.6.4	Type variables . . . . .	33
9.7	Game-specific language constructs . . . . .	33
9.7.1	States . . . . .	33
9.7.2	Latent functions . . . . .	33
9.8	That's it for now... . . . . .	34
9.9	Quote . . . . .	34

# 1 Overview

## 1.1 Introduction

Okay. You want a scripting language for your engine. Why? Because they're just really cool, and everybody has 'em these days.

First, decide what kind of scripting you want; Henry Robinson already wrote an introduction to the different kinds scripting (be sure to read it if you haven't already). In this tutorial series, I'll be talking about a compiler / virtual machine system like UnrealScript.

Next, you need to know two things: how to implement such a scripting engine, and some reasons why a scripting engine is not only cool but actually useful.

Here's what I made up:

- Useful new language features like states, latent code, etc.
- A sandbox environment that can't crash the game engine.
- Being able to create game content without knowledge of the engine internals and without having to recompile the engine.
- Fully platform-independent script code.

However, there are also drawbacks:

- Relatively slow - scripts run at least 15 times slower than executable code.
- Limiting - scripts can't be used to create entirely new visual effects (partly because of the lack of speed).
- People creating game content have to learn a new language.

Of course, we don't let those stop us; we had already made up our minds. Now, where to start?

## 1.2 Required Reading

I started when Unreal hadn't been released for very long. I was browsing their tech site and found the UnrealScript reference document. I had heard of UnrealScript of course, but didn't really know what it was. I read the document and thought the idea of a script language was really cool. I wanted to write my own, and link it to a game engine so that the whole world could easily create new content for my game.

Fortunately, I got a class in Compiler Construction that semester, and as a practical assignment we had to implement a VERY simple Pascal language.

```
print "Please enter your name > ";
input name;
if (name == "Jan") {                               // string comparison
    name = "my creator";                             // string assignment
    happy = "yes";
}
print "Thank you, " + name + "!\n" +                // string concatenation
      "You've just made a simple program very happy!";
```

As you can see, there are no grouping structures like functions, classes, etc. and not even numeric types. The final product, however, will be easily expandable.

But before we get to that, we've still got a long way to go - remember the list of components from last time? Today we'll implement the first one: the lexical analyzer, or lexer for short. It'll be a nice warm-up, since it's not really a difficult part of the compiler.

Okay, ready to get analyzing?

### 1.3 What, Why, and How

First, I guess you'd like to know what a lexical analyzer *is* and why we need it. The task of the lexical analyzer is to convert the character stream that is a source file into a token stream. Essentially it just looks at the characters sequentially and recognizes "words" in them.

We could of course write a function that compares the string at the current position in the source file to all our keywords, but that would be unbearably slow. Instead, we'll use a finite state machine to recognize words (if you don't know what that is, don't worry - you don't need to).

The great thing about the lexer is that we don't actually have to do any hard work - we let the lexer be generated by a program called 'lex'. This is a standard Unix program; there are also several Win32 implementations (The one I use is called Flex and it's included in the zip file this part). For the complete Lex HTML manual, look here (<http://www.monmouth.com/~wstreett/lex-yacc/lex-yacc.html>).

Well, you know what the lexer does and how we're going to make it. Now would be a good time to download the tut2.zip file and have a look at the code. The source files for this part are string.l and main.cpp, plus a few header files. Note that the .zip file contains a directory structure that I'll use throughout this series; for example, flex.exe is located in the base dir and the files specific to this tutorial are in the tut2 directory.

### 1.4 The Lexer Rules!

Lex needs a few simple rules to be able to generate our lexer. Before I explain these rules, let's study the layout of the Lex input file.

```
<definitions>
%%
<rules>
%%
<user\_code>
```

The `<definitions>` section contains some regular expression macros (Regular expressions are explained in the Lex manual, and more thoroughly here). These tell Lex what we mean by a LETTER, a DIGIT, an IDENT (an identifier, which is defined as a letter followed by zero or more letters or digits) and a STR (a string constant, which is a string of characters enclosed in double quotes).

This section can also contain some initial code, like `#includes` for standard header files and forward references. This code should be put between the tags `%` and `%`. I included the header file `lexsymp.h` here, which we'll look at in a minute.

The `<user_code>` section can contain any code you need for analyzing; in our case it contains a function to skip all characters inside comment, functions to pass the identifier name and contents of string constants to the caller, and our main function.

The `lexsymb.h` file contains the declarations of the token symbols the lexer function will return. It also contains the declaration of a union `'yylval'` used to pass extra information (like the name of an identifier) to the caller; why we're using this specific union will become clear in the next part.

Now, let's look at the actual rules. Note that I'm using `/* */` comment; Lex is an old program and doesn't support `//` comment in its input files. We will produce a C lexer by the way - C++ versions of Lex are available but the standard Unix Lex produces C code, and we want to keep things portable, don't we?

```

‘‘if’’      {return IF;}
‘‘=’’      {return ASSIGN;}
‘‘;’’      {return END_STMT;}
{IDENT}    {Identifier ();          /* identifier: copy name */
           return ID;}
{STR}      {StringConstant ();     /* string constant: copy contents */
           return STRING;}
‘‘//’’     {EatComment();}         /* comment: skip */
\n         {lineno++;}            /* newline: count lines */
{WSPACE}   {}                     /* whitespace: (do nothing) */
.          {return ERROR_TOKEN;}  /* other char: error, illegal token */

```

I left out some rules that were very similar. As you can see, each rule starts with the pattern Lex should recognize, followed by some code telling Lex what to do (this code *can* contain C++ by the way, because Lex just copies it into the output file). Note that the topmost rules are evaluated first; this is sometimes important.

The first three rules are very simple; they just recognize a string of characters and return the appropriate token symbol to the caller. Note that you can just change the strings if you want “:=” to be the assignment operator, for example.

The fourth line is the first “interesting” one: it makes use of the `IDENT` macro, so it recognizes a string of characters/digits that doesn’t satisfy any of the previous rules. If found, it calls the `Identifier()` function, which copies the string from `yytext` (which contains the text for the current token) into a new char array. The lexer then returns the `ID` token; the caller can just look at the `yyval->str` pointer to find the name of the identifier. `STR` does the same for a string constant.

The next lines takes care of comment, newlines and whitespace. Note that the line number is counted; we’ll use this for error messages in the future. The final rule tells Lex that if the input doesn’t satisfy any of the above rules (the regular expression “.” means: any char except ‘\n’), we should return an error token and let the caller decide what to do.

This Lex input file can be compiled to `lex.cpp` to using the command:

```
flex -olex.cpp string.l
```

Also included in the .zip archive is a project file for MSVC 6.0 (`string.dsp`). I believe it also works with 5.0, but I’m not sure. This project file contains a custom build command for `string.l` so it compiles automatically.

Unfortunately, `lex` uses a non-standard include file, `unistd.h`, which is not available on Windows systems. In the base dir is an empty `unistd.h` file; include the base dir in your include files path (in MSVC: Tools→Options→Directories→Include).

The file `lex.cpp` contains a complete lexer that satisfies our rules. It’s that simple! The main program in this example just reads a token using the lexer function and shows the token name and value (if it’s `ID` or `STR`). You can try typing some stuff and see what the lexer makes of it; random characters will generally be seen as `ID`’s, but other unused chars like ‘\$’ will cause an `ERROR_TOKEN`. You can try `example.str` (in the base directory) too!

## 1.5 It Gets Better, Really

Well, now we have a program that can “read”. Unfortunately, it still has no idea what it’s reading and whether this is correct by our standards. It will just accept any token it knows about.

So it has to learn about grammar. By an amazing coincidence, grammar is exactly what the next part is about. The next component is called the parser and its task is to find out the structure of the program and check the syntax while it’s at it.

Then things really get interesting. We will actually be able to feed the program from the introduction to the compiler and it will accept it - not because it accepts just about anything, but because it knows the program is syntactically correct! I just know you’re as wildly excited about this as I am, but you’ll have to wait until the next part..

## 1.6 Quote

“And so it was only with the advent of pocket computers that the startling truth became finally apparent, and it was this:

Numbers written on restaurant bills within the confines of restaurants do not follow the same mathematical laws as numbers written on any other pieces of paper in any other parts of the Universe.

This single fact took the scientific world by storm. It completely revolutionized it. So many mathematical conferences got held in such good restaurants that many of the finest minds of a generation died of obesity and heart failure and the science of maths was put back by years.”

HHG 2:7

# 2 The Parser

## 2.1 Introduction

The executable from the previous part did a nice job converting programs to tokens. All keywords, operators, punctuators, identifiers and constants were immediately recognized and reported. However, you could type

```
{ this ) = ‘pointless’ + ;
```

and the program would just accept it and happily produce a list of tokens. Since this is clearly not something we want to allow (I have no idea what the above ‘statement’ should do), we have to be able to recognize syntax structures (or lack thereof) in the input program.

We do this by means of the parser, which finds the structure of the program and checks for any syntax errors.

## 2.2 A Little Language Theory

How can we tell the parser what our language looks like? Well, we can use a way of specifying syntax (or “grammar”) called Backus-Naur Form (BNF). This method of specification uses the basic concepts that a program is made up of. For example, expressions can be, among other things, identifiers or string constants. In BNF, this is written as follows:

```
expression: identifier | string;
```

A statement can be a print or an input statement:

```
statement
  : PRINT expression END_STMT
  | INPUT identifier END_STMT
  ;
```

(remember that PRINT, INPUT and END\_STMT were tokens returned by our lexer)

Now, a program can be expressed as being a list of statements in the following way:

```
program: | program statement;
```

which says that a program is either empty or a program followed by one statement, which is a nice recursive definition of a list of statements.

So, the language we’ve defined in BNF includes the statements:

```
print a;
print ‘Hello’;
input name;
```

Not legal is:

```
input ‘Hello’;
```

because we’ve defined the input statement so it can only take an identifier, not a string constant.

With the use of BNF, we can formally specify the whole syntax of our language. Note that this does not include semantics yet, so the statement

```
a = (b == c);
```

will be accepted by the parser even though it doesn’t make sense (we’re trying to assign a boolean value to a string variable). Semantics are checked at a later stage.

Great, we now know enough about language specifications to create our parser!

## 2.3 Looks Familiar!

The parser is also generated using an external program. This one is called Yacc (a standard Unix tool, just like Lex); we'll be using an improved version called Bison (get it?). The Bison manual can again be found here. The layout of a Yacc file (extension .y) is in fact very similar to that of a Lex file:

```
<definitions>
%%
<rules>
%%
<user_code>
```

The `definitions` section contains token definitions, type information, and the definition of the `yylval` union we saw in the previous part. That's why we used a union: Yacc uses this same union to pass information between the different 'language concepts' like expression, statement, and program. From these definitions, Yacc generates the `lexsyml.h` file for us (actually it creates `parse.cpp.h` but the `parser.bat` procedure renames it).

Again, just like a Lex file, this section may contain some initial code between the tags `%` and `%`. The section is not yet used in this part, but can also contain any additional code you need.

The rules are specified in Backus-Naur Form as explained in the previous section.

There is a nasty catch to using Yacc, though, and it's that your language specification has to be LR(1)... What exactly that means is explained extensively in the Dragon book (section 4.5 about bottom-up parsing), but basically the parser has to be able to tell what kind of syntax rule to use by looking at the current lexer token and not more than ONE token ahead. The following rule would generate a shift/reduce conflict:

```
A:
  | B C
  | B C D
  | D E F
  ;
```

The conflict would not arise when reading `B` from the input file and looking ahead to `C` as you may think, because these can be grouped (both of these alternatives will generate an `A` symbol eventually); the problem is that the second alternative ends with `D`, and the third begins with it: when the parser has read `C` and looks ahead to `D`, it can't decide whether to classify this as `A2` or as `A1` followed by a `A3`! So although the complete syntax definition may not be ambiguous at all, to the parser it *is* because it can only look ahead ONE token. Yacc will call this semi-ambiguity a shift/reduce or reduce/reduce conflict.

Well, don't let all of that scare you. Have a look at the rules. The most important one is perhaps the statement rule:

```

statement
: END_STMT                {puts (‘‘Empty statement’’);}
| expression END_STMT    {puts (‘‘Expression statement’’);}
| PRINT expression END_STMT {puts (‘‘Print statement’’);}
| INPUT identifier END_STMT {puts (‘‘Input statement’’);}
| if_statement           {puts (‘‘If statement’’);}
| compound_statement     {puts (‘‘Compound statement’’);}
| error END_STMT         {puts (‘‘Error statement’’);}

```

As you can see, this defines all the types of statements our language has, with code next to it telling the parser what to do when it finds each alternative. I think this rule is pretty straightforward. One thing though: the “Error statement” tells Yacc what to do if it encounters a parse error (such as an illegal token or a non-fitting token) while parsing a statement. In this case it will look for the next `END_STMT` token and continue parsing after that. Parse errors are always reported to the `yyerror()` function defined in `main.cpp` so our compiler can deal with it in an appropriate way. If you don’t supply an error rule anywhere in your `.y` file, your parser will stop when it finds a parsing error, which isn’t very graceful.

Perhaps you’re wondering why there are so many different expression rules: `expression`, `equal_expression`, `assign_expression`, `concat_expression` and `simple_expression`. This is to specify the precedence of the operators. If the parser sees this:

```
if (a == b + c)
```

it should know it shouldn’t evaluate `a==b` and then try to add the boolean result of this to the string variable `c`. The different expression rules make sure there’s only one way to parse this statement. Just look at it for a while; it works.

Another problem is parsing the following statement:

```
if (a == b)   if (c == d)   e = f;   else g = h;
```

The parser doesn’t know to which if-statement (the inner or outer if) the else belongs; it could think you meant

```
if (a == b) {if (c == d) e = f;} else g = h;
```

but the convention in all imperative languages is to group the else with the inner if.

Since there is no way to solve this by changing your rules, Yacc would report a shift/reduce conflict. This conflict is simply suppressed by adding the line

```
%expect 1
```

to the definitions section, meaning Yacc should expect 1 conflict. Yacc will “solve” this conflict by associating the else with the nearest if, just like we want. That’s just the default solution to any conflicts it finds.

The rest of the Yacc file is pretty self-explanatory once you understand BNF. If there's anything left that's unclear, you can always mail me about it or post a question on the messageboard.

This Yacc input file can be compiled using the command:

```
bison --defines --verbose -o parse.cpp
```

If you get any conflicts, look at the file `parse.cpp.output` which contains details about the conflict (even if you don't get errors it's an interesting file to look at). If you run into any conflicts you can't solve, just send me your `.y` file and I'll have a look at it.

If everything went OK (it should do so with the sample code) you get a working lexer in the file `parse.cpp`. All our new main program does is call the `yyparse()` function and the whole input file will be sliced & diced for us!

Try `example.str` again and watch the error it produces. Error? Yes, that's right, I forgot one `';` at the end of line 13. But hey, it works! Great huh?

### 3 Whew!

That was quite a lot we did today. We learned some formal language theory, how to use it in Yacc, why Yacc is so picky about which grammars it supports, and how to specify operator precedence. And finally, we made a working parser!

Well, I think the hardest part is actually behind us. If you understand this, the rest will be a piece of cake. However, if I've lost you somewhere while ranting about LR(1) grammars, post on the messageboard or mail me so I can try to clarify things! Any other questions or comments are also welcome, if only so I know people are actually reading this stuff (okay, I believe Kurt at least skims over it while converting to HTML ;-)

Now, what lies in the future? Next time we'll probably write TWO new components: the symbol table and the syntax tree. Until then, you have a week to experiment with the code. Tip: try to get the compiler to accept C-style while statements!

#### 3.1 Quote

“The major problem is quite simply one of grammar, and the main work to consult in this matter is Dr Dan Streetment's *Time Traveller's Handbook of 1001 Tense Formations*. It will tell you for instance how to describe something that was about to happen to you in the past before you avoided it by time-jumping forward two days in order to avoid it. The event will be described differently according to whether you are talking about it from the standpoint of your own natural time, from a time in the further future, or a time in the further past and is further complicated by the possibility of conducting conversations whilst you are actually travelling from one time to another with the intention of becoming your own father or mother.”

HHG 2:18

## 4 The Symbol Table & Syntax Tree

### 4.1 Introduction

If we are to do something useful with the lexer and parser we created in the last two parts, we need to store the information we can gather from the program in data structures. This is what we're going to do next. Two important components are involved in this: the symbol table and the syntax tree.

The symbol table is, like the name suggests, a table that contains all symbols used in our program; in our case, all the string variables, and the constant strings too. If your language includes functions and classes, their symbols would be stored in the symbol table too.

The syntax tree is a tree representation of the program structure; see the picture below for the idea. We use this representation in the next part to generate intermediate code. Although it is not strictly necessary to actually build the syntax tree (we already have all the information about program structure from the parser), I think it makes the compiler more transparent, and that's what I'm aiming for in this series. This is the first part that includes 'real' code,

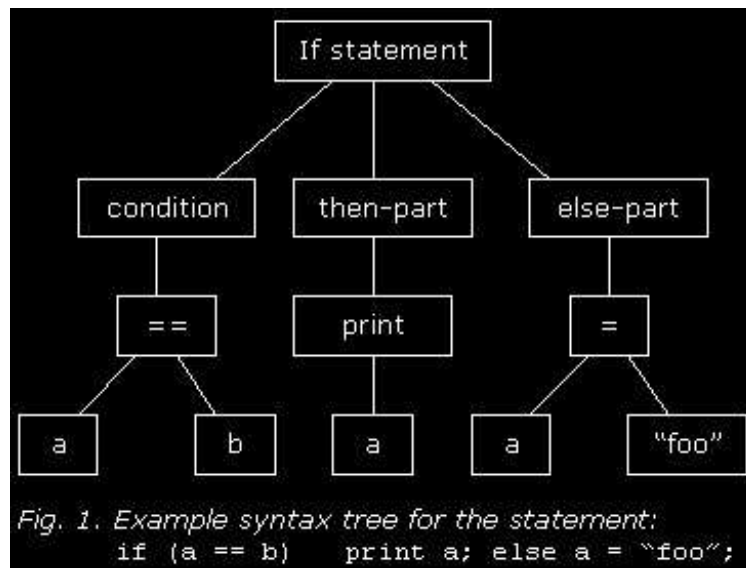


Figure 1: Example syntax tree for the statement: `if (a==b) print a; else a = 'foo';`

and before you look at it, I'd like to make it clear that this code was written to be easily understandable rather than well-structured. It will suffice for the compiler we're making here, but in a real-world script compiler you'll want to do a lot of things differently. I'll try to mention these things as we come across them.

## 4.2 Passing Information Between Rules

Obviously, we have to add functionality to our parser; particularly, when we find a symbol we enter it into the symbol table – but we also want the “parent” rule (the rule that actually uses the identifier) to have access to the symbol description.

Something similar is required when we’re building a syntax tree: we want the parent rule to have a pointer to the nodes of his ‘child rules’ (the rules the parent rule is constructed of).

Well, remember the yylval union? Yacc uses this to pass information between the rules, too. Every rule can have an associated field in the yylval union; this is the rule’s type. At the top of string.y you can see type declarations like the following:

```
%type <symbol> identifier string
%type <tnode> statement expression
```

symbol and tnode are new members of the union; they represent a pointer to a symbol description and a pointer to a syntax tree node, respectively.

Now the statement rule uses these types as follows:

```
| expression END_STMT      { $$ = new TreeNode (EXPR_STMT, $1); }
```

This means: if you find an expression statement, construct a new tree node (and ‘return’ the node pointer) of type `EXPR_STMT` with one child: the expression that the statement is composed of. So `$$` represents the ‘return value’ of a rule, and `$1` is the value returned by the first symbol in the rule definition (expression). `$2` has no meaning here, because the lexer doesn’t set an yylval-member for the token `END_STMT`.

I hope this explanation is clear enough, because it is important. Essentially, the rules form a hierarchy and every rule can return a value to a ‘higher’ rule.

Now let’s see what data structures we use for the symbol table and the syntax tree.

## 4.3 The Symbol Table

The symbol table in our example contains very little information; basically it’s only the name of the variable and the line in which it was first declared. We will use it to store more data later on, though.

The implementation is very simplistic: it just builds a singly-linked list of symbol descriptions and searches this list linearly when we retrieve a symbol (have a look at the `symtab.cpp`, it’s really straightforward). For a real compiler, the symbol table is usually implemented as a binary search tree or a hash table, so symbols can be looked up much faster.

All we need to do to enter our symbols into the table when the parser finds them is this:

```

identifier
  : ID
  {
    $$ = st.Find ($1);
    if ($$ == NULL) { // doesn't exist yet; create it
      $$ = new SymDesc ($1, STR_VAR, NULL, lineno);
      st.Add ($$);
    }
  }
;

```

We treat string constants as constant variables, so we generate a name for them and also enter them into the table.

Note that a more advanced compiler would probably let the lexer store and retrieve identifiers. This is because in a complex language there are many different meanings identifiers can have: variables, functions, types, etc. The lexer could retrieve the identifier description and directly return the appropriate token to the parser. Since our identifiers are always variables, I just let the parser handle them.

#### 4.4 The Syntax Tree

For the syntax tree I have created a very simple `TreeNode` class. It just stores pointers to children and some additional information (the node type and a link to a symbol if applicable). Have a look, there's nothing complicated going on in there.

As you saw before, we can easily build our syntax tree from the recognized parser rules:

```

equal_expression
  : expression EQUAL assign_expression  {$$ = newTreeNode(EQUAL_EXPR, $1, $3);}
  | assign_expression                    {$$ = $1;}
;

```

You can see that we sometimes just pass the information from a child rule on to our parent rule unchanged; if your `equal_expression` is actually just an `assign_expression`, there's no point in making an extra node for it; you just use the one created in `assign_expression`. Remember that the only reason we created so many expression rules was to unambiguously handle operator precedence.

Compilation of this part (and the following parts) is the same as for the previous parts. The program still accepts syntactically correct programs, but now shows a dump of the symbol table and syntax tree it has built.

#### 4.5 That's Pretty Cool, But...

Okay, so it reads the program and it analyzes it. But it doesn't really do anything smart or useful with it, does it?

Well, no. Not yet. We still have some more components to implement. The next part will cover semantic checking and intermediate code generation. Those will take us a long way towards a compiled program.

I hope you don't think it's going too slow, but I want to focus on each component separately, not just rush through things. If you understand all this stuff immediately, be happy with that and experiment!

See you next time.

## 4.6 Quote

(Part of the Guide entry on the Babel Fish)

“Now it is such a bizarrely improbable coincidence that anything so mind-bogglingly useful could have evolved purely by chance that some thinkers have chosen to see it as the final and clinching proof of the non-existence of God.

The argument goes something like this: ‘I refuse to prove that I exist,’ says God, ‘for proof denies faith, and without faith I am nothing.’

‘But,’ says Man, ‘The Babel fish is a dead giveaway, isn't it? It could not have evolved by chance. It proves you exist, and so therefore, by your own arguments, you don't. QED.’ “

HHG 1:6

# 5 The Semantic Checker & Intermediate Code Generator

## 5.1 Introduction

A little late this time.. Exams are horrible things, they really interfere with useful stuff.

Right, last time I promised results and you're gonna get 'em. More than you wished for, probably ;-).

But first a general remark about these tutorials. I have a tendency to write very compact explanations. The information is all there, but often there are two important facts per sentence.. The disadvantage of this is that if something is not clear, you probably can't follow the rest of the tutorial. Please tell me when I'm going too fast so I can clear things up.

Back to this part. It's about semantics and intermediate code. Checking the semantics will make sure that our program is really correct, and the intermediate code will be a giant leap towards a virtual executable.

Let's get checking!

## 5.2 Checking the Semantics

Semantic checking is done to make sure that not only the syntax of the program is correct, but the statements also make sense. The number of parameters supplied to a function, for example, should be the number this function expects.

The major part of semantic checking is type checking: determining the types of expressions and reporting any inconsistencies, like trying to compare a boolean to a string or passing the wrong type of argument to a function.

Of course you may want to allow some of these 'inconsistencies'; for example when someone uses the following statement

```
print 'a and b equal: ' + (a == b);
```

he probably means that the expression `(a == b)` should be converted to a string automatically, resulting in the string "true" or "false". This is called a coercion. In our sample compiler I've only allowed bool to string coercion, but you could easily add string to bool coercion if you think that's useful.

The code for our semantic checker is not complicated. I've added a member function to `TreeNode` called `Check()` (in `synttree.cpp`) that checks a node's semantics, assuming that all its children have already been checked. `Check()` is automatically called from the `TreeNode` constructors, so that's a safe assumption.

`Check` sets a new member variable called `rettype`, the 'return type' of the expression. A condition, for example, has boolean as its return type, while a string concatenation returns another string. `Rettype` is used to check the semantics of the parent node. The function `CoerceToString` is used to coerce any expression to string type (if it isn't already) by inserting a new node, `COERCE_TO_STR`, as the new parent of the node to be coerced.

For a simple compiler this is easy, but generally it's not. If your language includes more base types, references, arrays, classes and (operator) overloading things get ugly real fast; you'd better have a solid type checking system if you ever want your program to run.

In a real compiler a lot of the work goes into this: there are more coercions, you have to figure out which of the overloaded functions must be used, type equivalence isn't so trivial anymore, etc.

So yes, in this case it's simple, and it's a useful learning experience to try to expand this system with some more types, but at some point you'll want to take a more general approach.

The rest of the code pretty much explains itself. It just enforces simple things like: if-conditions should be boolean, assignment expressions should be strings, etc.

### 5.3 Generating Intermediate Code

Intermediate code is a sort of graph representation of your program: every instruction has a pointer to the next instruction, and jumps have a pointer to their target instructions.

I can think of two advantages of doing it this way (with pointers) instead of immediately generating code into a big array. First, because of the pointer-representation, it's easy to concatenate pieces of code together, and even cutting some instructions can be done without having to update all the jumps, etc. So optimization is relatively easy to do. Second, if you want to change some

instructions on your virtual machine it's easier to adapt your compiler to the new VM because you only have to change the translation step from intermediate to final code, which is relatively easy.

So, with all that in mind, we design our intermediate code language. The opcodes in this language will be very similar if not identical to the ones our virtual machine will execute. Have a look at them:

```
enum Opcode {
    OP_NOP,           // no operation
    OP_PUSH,          // push string [var]
    OP_GETTOP,        // get string from top of stack (=assign) [var]
    OP_DISCARD,       // discard top value from the stack
    OP_PRINT,         // print a string
    OP_INPUT,         // input a string [var]
    OP_JMP,           // unconditional jump [dest]
    OP_JMPF,          // jump if false [dest]
    OP_STR_EQUAL,     // test whether two strings are equal
    OP_BOOL_EQUAL,    // test whether two bools are equal
    OP_CONCAT,        // concatenate two strings
    OP_BOOL2STR,      // convert bool to string
    JUMPTARGET        // not an opcode but a jump target;
                    // the target field points to the jump instruction
};
```

You can see our VM is going to be a stack machine: opcodes will operate on values from the stack and will put values back on the stack. I think this is the simplest type of machine, both to generate code for and to execute code on.

One note about the JUMPTARGET “opcode”: whenever we have a (conditional) jump in our code, it points not to the actual target instruction, but to a prefixed “JUMPTARGET” instruction. The reason for this is that when we optimize we have to know every jump target point in the code, or we might optimize a target instruction away and mess up our program. The JUMPTARGETs won't be in our final bytecode.

In general, all opcodes operate on items on top of the stack. So OP\_STR\_EQUAL pops the top two items off the stack (these must be strings), checks if they're equal, and pushes the resulting boolean on the stack. Your program can then use the OP\_JMPF instruction to react to that result: it jumps to the target instruction (which IS supplied with the instruction, not on the stack) if the boolean on top of the stack is false and continues execution if it's true.

The instructions are stored in a very simple intermediate instruction class, which just stores the opcode, a 'symbol'-operand (e.g. for OP\_INPUT) and a jump target instruction if applicable, a next instruction pointer and a line number. The line number is actually just so we can make the code human-readable with the Show() function.

Now, let's look at how we generate the intermediate code (intcode.cpp). Generally we just recursively generate code for all subtrees in the syntax tree. So main calls the GenIntCode() function with the root of the tree; GenIntCode

then takes care of the rest and returns a pointer to the beginning of the intermediate code.

First a simple case, the INPUT\_STMT node:

```
case INPUT_STMT:
    return new IntInstr (OP_INPUT, root->symbol);
```

This just generates a new OP\_INPUT instruction and returns it. Note that this instruction is also a “block of instructions” of length 1 (the next pointer is set to NULL by default).

The PRINT\_STMT is a little harder:

```
case PRINT_STMT:
    blk1 = GenIntCode (root->child[0]);
    blk2 = new IntInstr (OP_PRINT);
    return Concatenate (blk1, blk2);
```

First we generate the code that evaluates the expression supplied to the print statement (`root->child[0]`). Then we create a new instruction OP\_PRINT that prints the string on top of the stack. Note that we assume the expression leaves its value on top of the stack. We’ll have to make sure of this ourselves, of course. Finally we concatenate the two block of instructions and return the result.

Now a really hard one: the IFTHEN\_STMT. I just generate the blocks needed for this, and then concatenate them all together. The approach is to check the condition, jump to the end if it’s false, and execute the then-part if it’s true.

```
case IFTHEN_STMT:
    // First, create the necessary code parts
    cond      = GenIntCode (root->child[0]);
    jump2end  = new IntInstr (OP_JMPF);      // set target below
    thenpart  = GenIntCode (root->child[1]);
    endif     = new IntInstr (JUMPTARGET, jump2end);
    jump2end->target = endif;

    // Now, concatenate them all
    Concatenate (cond, jump2end);
    Concatenate (jump2end, thenpart);
    Concatenate (thenpart, endif);
    return cond;
```

Remember that `root->child[0]` is the condition-subtree and `root->child[1]` the then-subtree.

Well, if that was understandable, you won’t have a problem with the rest of the code. All tree nodes are translated in this way. The `Show()` function just shows the code we’ve generated. Have a look at what all this does:

```
Program:
    if (a==b) a; else b;
```

Intermediate code:  
1: OP\_NOP  
2: OP\_PUSH a  
3: OP\_PUSH b  
4: OP\_STR\_EQUAL  
5: OP\_JMPF 9  
6: OP\_PUSH a  
7: OP\_DISCARD  
8: OP\_JMP 12  
9: JUMPTARGET 5  
10: OP\_PUSH b  
11: OP\_DISCARD  
12: JUMPTARGET 8

That looks an awful lot like assembly code, doesn't it? Well, that's because it is. It's Virtual Assembly, and essentially we only need to write an assembler program to generate a Virtual Executable.

## 5.4 Whoa, what happened?

That went fast, didn't it? One moment we're wondering if we're ever going to do something interesting, and suddenly we have generated virtual assembly code! We're almost done!

Well, not quite. Next time we're gonna look at some optimizations (I'm sure you can think of some if you look at the some of the output from this part). And soon, we'll produce actual virtual machine code - but I guess we'd better have a virtual machine first! We'll see where we go from there. You're welcome to send me any ideas or suggestions about what direction to explore.

Bottom line: some interesting stuff is coming up. Stay tuned!  
See you next time.

## 5.5 Quote

The story so far:

In the beginning the Universe was created.

This has made a lot of people very angry and been widely regarded as a bad move.

HHG 2:1

# 6 Optimization

## 6.1 Did you Spot The Bug?

Noticed anything funny about the code of the last two parts? A memory leak maybe? Emmanuel Astier did; he found a bug in the symbol table: when

deleting the symbol table, I only delete the first entry in the linked list, not the others... Okay, so the program doesn't crash, but it ain't pretty. This will be fixed in the next tutorial. Thanks Emmanuel!

## 6.2 Introduction

Well, my exams are done so I can finally continue the tutorial!

In this part I'll cover ways to optimize our intermediate code. Remember that we're using a very simple code generation algorithm so the code can probably be optimized quite a bit.

Because we're executing on a virtual machine, optimization becomes extra important: each of our instructions will take at least 20 CPU instructions to execute (it's very hard to get below that) so the fewer instructions, the better.

Note that I'll only be talking about machine-independent optimizations; machine-dependent optimization is a very different topic altogether, where you have to consider things like pipelining efficiency, register usage, etc. And of course, machine-dependent optimizations for your code are only needed if you actually run on hardware, which we won't. Of course there may be lots of ways to speed up execution of the virtual machine itself, but we'll see to that later.

Sorry, there's no example code for this part. Some of the optimization ideas are quite easy to implement, so you'll have no trouble with those. Others are more complex and require quite a lot of work to implement. I don't have the time to do this so I'll just give the general idea.

There are two important ways we can speed up our code. One way is to translate the code so it uses less instructions. Another is to make more powerful instructions.

## 6.3 Extra Opcodes

Higher-level instructions tend to be relatively fast to execute on a VM since the overhead from stack manipulation and instruction pointer updating is still (roughly) the same while we do more work. So we're going to forget RISC and go crazy with exotic instructions! ;-)

Let's look at some code. This is part of `example.vasm`, the compiled version of `example.str`:

```
1: OP_NOP
2: OP_PUSH strconst1
3: OP_GETTOP a
4: OP_DISCARD
5: OP_PUSH strconst2
6: OP_GETTOP b
7: OP_DISCARD
8: OP_PUSH a
9: OP_PUSH b
10: OP_CONCAT
```

```

11: OP_GETTOP s
12: OP_DISCARD
13: OP_PUSH s
14: OP_PRINT

```

I noticed a few things about this. First, at three places in the code there's an `OP_DISCARD` following an `OP_GETTOP`. We could speed this up by converting it to one `OP_POP` opcode that gets the top value and removes it from the stack. I could've done this in the first place, but I thought this was easier.

Second, I see `OP_PUSH; OP_GETTOP; OP_DISCARD` twice. That's the code for simple assignments like "a = b;". We could provide a special opcode `OP_COPY` for this that copies the value of one variable to another.

Third, in the complete code for this program there are quite a few "double pushes": two pushes following each other. We could make a separate opcode `OP_PUSH2` to speed this up.

You can probably think of other high-level instructions to add. For example, a `OP_CONCATTO` opcode to concatenate to an existing string (`s += "foo";`). If you pick these carefully they can really speed up execution, so take the time to study your assembly code and find optimization candidates.

## 6.4 Code Transformations

Another way of optimizing output code is to transform some part of the code to something that does the same thing faster. An example of this:

<i>Source</i>	<i>Assembly</i>	<i>Optimized</i>
<pre> s = a; if(s==d)... </pre>	<code>OP_PUSH a</code>	<code>OP_PUSH a</code>
	<code>OP_GETTOP s</code>	<code>OP_GETTOP s</code>
	<code>OP_DISCARD</code>	(cut away)
	<code>OP_PUSH s</code>	(cut away)
	<code>OP_PUSH d</code>	<code>OP_PUSH d</code>
	<code>OP_STR_EQUAL</code>	<code>OP_STR_EQUAL</code>
	<code>...</code>	<code>...</code>

Below are some established algorithms to transform your code, saving instructions and thus time.

Most optimizations concentrate on optimizing little pieces of code known as "basic blocks". A basic block is has the following properties: you can only jump into it at the beginning and you can only jump out of it at the end. So there are no jumps or jump targets in the middle of the block. This means that within the block we can be sure of certain things about the value of our variables and can use this information to optimize the code. For example, if you could just jump to somewhere inside the block we could never be sure that, at that point, `t` still has the value `(a * b - c)`.

Pointers make basic block optimizations a lot harder, since you have to be sure that a variable isn't modified through a pointer instead of through its

name somewhere inside the basic block. Sometimes you just can't be sure about this (with pointer to pointers it's almost impossible to know what variable is changed) and you just have to skip an optimization step that may have been valid.

#### 6.4.1 Algebraic identities

A simple way of optimizing code is to replace 'naive' calculations with faster versions that produce the same result. Note that these naive calculations are usually introduced by a simple code generation scheme rather than explicitly by the programmer. Have a look at the table, these are pretty obvious.

<i>Before</i>	<i>After</i>
<code>x + 0</code>	<code>x</code>
<code>x * 1</code>	<code>x</code>
<code>x ** 2</code>	<code>x * x</code>
<code>2.0 * x</code>	<code>x + x</code>
<code>x/2</code>	<code>x * 0.5</code>

#### 6.4.2 Common subexpression elimination

This optimization takes advantage of the fact that a certain expression may be used several times in a small piece of code:

```
a = a + (b - 1);
c = c + (b - 1);
```

Here, `(b - 1)` is a common subexpression and can be reused (the second `(b - 1)` expression can be 'eliminated')

```
t = b - 1; // store the subexpression in a temporary variable
a = a + t;
c = c + t;
```

#### 6.4.3 Loop optimizations

A well-known bit of Programmer Wisdom is "a program spends 90% of its execution time in 10% of the code", and although these percentages differ per program, everyone will agree most of the running time is spent in inner loops.

So if we can somehow optimize those loops, we can save a LOT of time.. Well, there are several ways of saving time in loops; I will briefly discuss two of them, code motion and induction variables.

Code motion is kind of like subexpression elimination, but instead of doing so inside a basic block, it calculates an expression before the loop begins (thus before the basic block) and uses this value throughout the loop:

```
while ( i <= limit-2 )
```

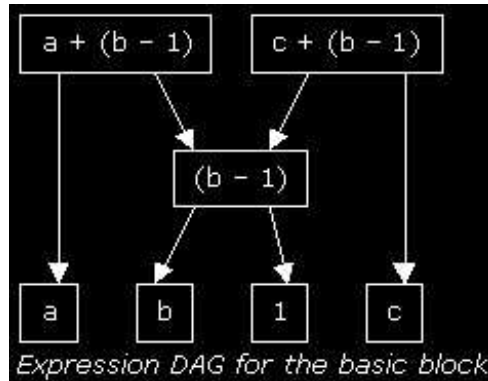


Figure 2: Expression DAG for the basic block.

becomes

```
t = limit - 2;
while ( i <= t )
```

Loops may not have many constant expressions, though. What they often DO have is a loop counter, and this loop counter is used frequently for calculations like array indices, etc. That's where induction variables can help us.

If  $j$  is our loop counter, and  $j * 4$  is calculated for every loop iteration, we can introduce an induction variable and replace this multiply by an add:

```
for (j = 0; j < n; j++) {
    .... (j * 4) ....
}
```

becomes

```
t = 0;
for (j = 0; j < n; j++) {
    .... t ....
    t += 4;
}
```

#### 6.4.4 Jump elimination

Sometimes you can eliminate a jump by looking at its target block. For example, when you have:

```
1: jmp 7
   ...
7: str_equal
8: jmpf 10
9: ...
```

you can copy code from the target block and save a jump (if the condition is false):

```
1a: str_equal // | copy of the target block
1b: jmpf 10   // |
1c: jmp 9     // if condition was true, goto 9
    ...
7: str_equal
8: jmpf 10
9: ...
```

It's up to you to decide how large parts of code you're willing to duplicate in order to eliminate one jump, but in inner loops it can save a lot of time.

## 6.5 Coming Up Next...

Hopefully with this information, your programs can become a little more efficient. Compiler code optimization is a very complex area though, and we've covered only little bits of it. The Dragon book (see tut 1) covers a lot more of it, so check it out if you're interested.

Next time we'll create our virtual machine, and maybe generate our virtual machine code too. Then we can finally run a program.. So all our work hasn't been for nothing!

## 6.6 Quote

Somewhere on the wall a small white light flashed.

"Come," said Slartibartfast, "you are to meet the mice. Your arrival on the planet has caused considerable excitement. It has already been hailed, so I gather, as the third most improbable event in the history of the Universe."

"What were the first two?"

"Oh, probably just coincidences," said Slartibartfast carelessly.

HHG 1:30

# 7 The Virtual Machine

## 7.1 Introduction

We've generated intermediate code in part 5 and we would like to convert it to executable code so we can execute a program. But I've decided to create the virtual machine first, so we know what we're dealing with when generating the executable.

The virtual machine is of course a very important component of a scripting engine. It's where the code is executed at runtime, so it had better be fast. I won't focus on speed this time though.

Oh yeah: you get my Amazing Stack Template absolutely FREE with this part, No Extra Charge! AND you get a cool string class that was written

especially for this part, with at least 5 minutes work in it! Now that's real value for your money!

But first, an explanation of different machine types. In tut 5 I just said what kind our VM would be, without explaining what other possibilities there were. Andy Campbell asked me about these other possibilities, and I figured other people might be interested, so here goes:

## 7.2 Machine Types

As said, our machine will be a stack machine. In real machines, stack CPUs were used for early computers (and may still be used in simple devices today). The disadvantage is the many stack manipulations needed: a PUSH for every operand and a POP for every result. Often, though, you use the results directly for the next calculation so that's not always necessary.

Most CPUs nowadays have registers (a very limited number of very fast memory locations) it operates on instead of just a stack; a stack is still used for passing parameters to functions. Machines that can only operate on registers are called load/store machines since you have to load each value before you use it and store each result after you calculate it.

Some processors only operate on memory data; no stack and no registers are used. Machines with this type of processor are called three-address machines because most instructions have three address operands (i.e. ADD dest, src1, src2). I don't think they're used much in hardware because of the limited memory bandwidth, but they ARE a viable option for virtual machines.

For virtual machines, the stack machine is very easy to implement because you don't need temporary variables to store intermediate results when calculating an expression; you put everything on the stack (it's very similar to the way you process a postfix expression). We WILL use temporaries here though, because I didn't want to have to stack pointers. But more about that later.

It is not clear to me whether three-address machines might have an advantage; speed would be the most important one, and although I've tried both I can't say for certain which does fewer calculations in the most optimized case.. I do think it's easier to optimize three-address code, so that may give this type of machine an advantage.

Java apparently uses a stack machine (I'm not familiar with the Java VM but I've heard this).

## 7.3 A Virtual Piece of Cake

Our virtual machine object isn't complicated at all. Its most important members are: an instruction array, a string table and a stack. It has three main interface functions: **Reset**, **Read** and **Execute**.

The instruction array contains the instructions that comprise our program. The instruction class is extremely simple and looks like the one we used for intermediate code in tut 5.

The string table is just an array of pointers that can either be `NULL` or point to a string that's currently in use. This might be a program variable, or a temporary variable on the stack.

Our stack is made up of integers. They point to the string table so we know what string is actually on the stack. Why did I use integers and not just pointers to the string class? Because I wanted to keep things simple (for the readers, but also for myself :-): remember that we also want to stack booleans sometimes, so we would have to create a 'stack item'-class that could contain a string pointer or a boolean.. Now we just have an integer: if it's non-negative, we know it points to a string, and if it's negative it's a boolean. It's dirty coding, but it does the job and everyone can understand it. Don't try this at home. Or rather, don't do this on a real project.

Now, the interface functions. `Reset` reinitializes the virtual machine. It's a pathetically simple function.

`Read` should read in the program. Next time we'll change this function so it reads from `stdin`, but for now it's got a test program hardcoded into it. Change it if you like - just be very careful that the program stays correct, 'cause our VM does not crash gracefully.

`Execute` runs the program currently in memory. This is also a simple function: it has an instruction pointer, it looks at an instruction and executes the right code using a switch statement. A little note about the temporary variables: whenever we put a variable on the stack, we need to have a copy of it: we can't just push the variable's index in the string table, because its value might change and then the value on the stack would also change. This is why, for almost every stack operation, `NewTempCopy` and `DelTempCopy` are used.

Some little notes on optimizing the VM: We should make sure our stack manipulation is as fast as possible; my stack template is not especially fast. The same goes for string manipulations. In general, you should make the common case fast. All normal optimization techniques apply on VMs as well.

There's more to say on virtual machines: allocation schemes, garbage collection, keeping them both stable and fast, but I think I'll delay that till one of the next parts.

## 7.4 Next Time

Next time we'll FINALLY generate executable code. Then we'll be "done" with our simple scripting engine. After that I'll probably give an overview of the complexities of a real-life scripting engine, and talk about requested subjects. I'm no expert but I like to think that if you've just learned something yourself, you can teach it more easily. So if there's something about scripting you want to know, try me!

## 7.5 Quote

"Come," called the old man, "come now or you will be late."  
"Late?" said Arthur. "What for?"

“What is your name, human?”

“Dent. Arthur Dent,” said Arthur.

“Late, as in the late Dentarthurdent,” said the old man, sternly. “It’s a sort of threat you see.” Another wistful look came into his tired old eyes. “I’ve never been very good at them myself, but I’m told they can be very effective.”

HHG 1:22

## 8 Executable Code

### 8.1 Introduction

We have everything we need in order to execute our program, except.. executable code. We do have intermediate code and it’s already very close to the stuff our virtual machine understands. So all we have to do is a quick translation step between intermediate and executable code.

Why was this separate step necessary again? As you will see, the “translation” really comes down to putting our strings in an array and referring to them as indices instead of pointers to the symbol table. We already did jump targets last time, so they won’t change anymore. So this is a short part and the code changes aren’t big.

Maybe it wasn’t strictly necessary for us to create intermediate code. But when writing a more advanced compiler it’s useful to have a separate, more ‘conceptual’ stage before the actual machine code: it makes it easier to optimize the code, and you can retarget your compiler to another machine without much difficulties.

### 8.2 The Final Step

As you read the code for this part you’ll notice my extreme laziness took control at some point and made me write really evil code..

For example, I’ve combined the compiler and virtual machine into one program and I’m passing the `*intermediate*` code to the virtual machine, which isn’t the proper way of doing things. You’ll probably want your compiler to handle everything up to the executable generation, then maybe store the executable in a file and have your VM read & execute that file.

In our case, the `Read()` function in `VMachine` first gets all the strings from our symbol table and puts it in its string array. Then it walks through the code in a linear way and ‘translates’ it line by line. The special jump target instructions we used are just converted to `NOP` instructions, which should really be optimized away.

Oh, one particularly disgusting thing I did was store the string table index from the virtual machine in the symbol table from the compiler (using the symbol table’s new `PutNo()/GetNo()` members).. It’s a very easy way to find the string index later, but you’ll agree that modular programming is something entirely different..

### 8.3 It works! I can't believe it!

Hey, you can actually run a program now with the compiler/virtual machine combination! You probably nearly gave up on that, didn't you? Well, go ahead and try `example.str` on it, or `ex2.str` provided with this part's source download.. They should execute correctly. Is that fun or what?

Well, that's what we've been working for all this time. A tiny language that, while it isn't useful in itself, represents something very cool - you now know enough to write your own simple scripting engine!

### 8.4 So what happens now?

Well, after such an incredible climax (ahem) I'm sure you feel a bit empty and bewildered. Where do we go from here?

I will probably do one more part about some more advanced subjects, maybe talking about adding functions, classes, polymorphism, etc. to the language. Let me know what you're interested in.

There won't be any supporting code anymore though - everyone should be able to take the example compiler and expand it. Or, much better, write your own from scratch and get it right. The world's your oyster!

### 8.5 Quote

“More importantly, a towel has immense psychological value. For some reason, if a strag (strag: non-hitch hiker) discovers that a hitch hiker has his towel with him, he will automatically assume that he is also in possession of a toothbrush, face flannel, soap, tin of biscuits, flask, compass, map, ball of string, gnat spray, wet weather gear, space suit etc., etc. Furthermore, the strag will then happily lend the hitch hiker any of these or a dozen other items that the hitch hiker might accidentally have “lost”. What the strag will think is that any man who can hitch the length and breadth of the galaxy, rough it, slum it, struggle against terrible odds, win through, and still knows where his towel is is clearly a man to be reckoned with.”

HHG 1:3

## 9 Advanced Subjects

### 9.1 Introduction

Now that you have been playing with the finished scripting sample for a bit, and maybe have implemented some new features, you're probably wondering when we're getting to the good stuff.

Allow me to warn you that most this good stuff is a LOT of work (which is why I won't present any more sample code :-). I will discuss several advanced scripting subjects and give a general idea how (I think) they're implemented.

But first:

## 9.2 A lockup-resistant VM

Some time ago Joseph Hall gave me a great tip for dealing with infinite-looping script code. His idea is as follows: give the virtual machine a maximum number of opcodes to execute each time you call it, and let it resume next frame if it's not done yet; this is the virtual equivalent to a CPU-burst in pre-emptive multitasking. This way your game engine can keep running even when your script code hangs; it could even automatically detect that the script is constantly looping (i.e. stuck in one part of the code) and reset the VM.

Now, let's see how we could expand our language:

## 9.3 Functions

Adding functions to your scripting language isn't very hard, but it introduces the new concepts of parameters and local variables. For both of these, the stack is used. Before a function call the application pushes the parameters onto the stack. The function then reserves space on the same stack for its local variables. Then the function executes, using the reserved stack space to read values from and write values into. In our sample compiler, we've only pushed to and popped from the top of the stack, but now you also have to access memory addresses somewhere in the middle of the stack.

For functions you need two special opcodes: `CALL` and `RETURN`. `CALL` is an unconditional jump that saves the instruction pointer on the stack. `RETURN` reads the stored instruction pointer and jumps back to the instruction after `CALL`.

The most logical thing to do is to let the caller (not the function) remove the parameters from the stack; after all, the caller put them there in the first place. This also allows for a simple mechanism of "output parameters": the function changes one of its parameters and the caller stores this value into a variable afterwards. A function's return value can also be seen as an output parameter.

Function headers can be stored in the symbol table. With them you could store links to its parameters and local variables (which can each be separate symbol table entries). During code generation you can store the start address of the function in the symbol table as well.

## 9.4 Quote

"He stared at it for some time as things began slowly to reassemble themselves in his mind. He wondered what he should do, but he only wondered it idly. Around him people were beginning to rush and shout a lot, but it was suddenly very clear to him that there was nothing to be done, not now or ever. Through the new strangeness of noise and light he could just make out the shape of Ford Prefect sitting back and laughing wildly.

A tremendous feeling of peace came over him. He knew that at last, for once and for ever, it was now all, finally, over."

HHG 5:25

## 9.5 Overloading

Function overloading can be a very nice feature in a language, but implementing it can be tricky. The problem is finding the “right” function to call in case none of the available function headers match exactly with the supplied parameter types. In this case, you will have to coerce some of the parameters to different types to get a complete match. The question is, of course, what parameters to coerce and what type to coerce them to. Most compilers try to match the call to each of the alternatives and choose the one with the smallest number of coercions. Some compilers allow double coercions (i.e. `bool -> int`, then `int -> unsigned`), further complicating matters. Keep it simple is my advice.

Operators can just be seen as functions that are called using a different syntax; if you treat all your operators this way (not making them actual function (slow) but rather inline functions or macros), you can easily extend function overloading to operator overloading.

## 9.6 Classes

If you want to implement classes in your language, decide exactly which features you want to support. Supporting full C++ classes, including multiple inheritance, access restrictions, dynamic casts, virtual functions, etc. is very hard and I don’t recommend starting with all that. A simple class system with single inheritance is a good starting point. You can always expand it later if the need arises.

Classes, and structs, are compound data types: they contain a number of data members, and are linked to a number of methods or member functions. You could store a member list in your symbol table, which links to other symbol table entries that are the separate members. This allows you to easily find the offset of a member in the structure.

### 9.6.1 Inheritance

Single inheritance is relatively easy: when you’re looking for a member in an object, check whether the member’s in the child class; if not, check its parent class. The memory layout for child classes is very simple: first you store the parent, then its child, another child, etc. So downcasting is implicit: if you treat a Cat pointer as an Animal pointer, that simply means your program has access to fewer members, but the address of the pointer need not change.

Multiple inheritance introduces ambiguity problems when calling member functions or accessing data members. Consider this: Two classes B and C are child classes of the same class A. Then someone creates a class D that’s derived from both B and C. Now, if class A has a public member function `DoSomething` and the programmer calls `DoSomething` on an object of type D, you don’t know which of the two `DoSomething`’s to call: the one that acts on the A part of B or on the A part of C.. Okay, maybe a picture will make it clearer:

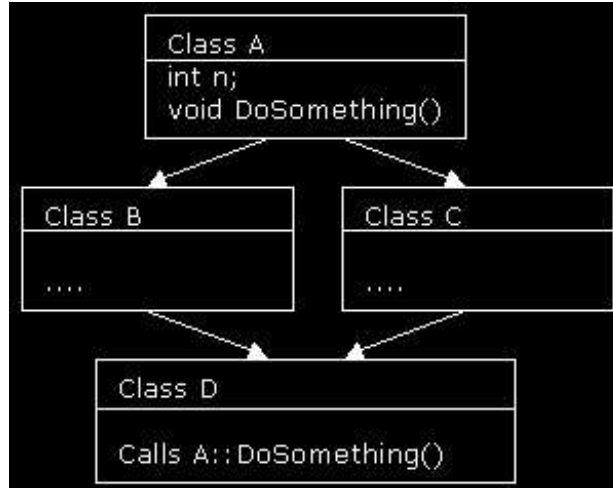


Figure 3: Inheritance.

### 9.6.2 Virtual functions

Virtual functions are a way of creating polymorphic (lit. “of many forms”) types; e.g. an `Animal` class that contains a virtual function `MakeSound()` and child classes `Cat` and `Dog` that each implement these functions in a different way (I’ll let you figure out how exactly they would implement them ;-). So when you call the `MakeSound` function of an `Animal` object, you don’t know (and don’t need to know) what kind of animal is making a sound.

Virtual functions are implemented by creating a so-called vtable. When the parent class declares a function as being virtual, it’s added to the vtable for that class. Each child class now gets its own version of the vtable, so that, although the caller sees no difference between the tables, different functions are called based on the object’s actual type.

### 9.6.3 Dynamic casts

Dynamic casts can be handy: for example, in UnrealScript you can not only downcast an object (cast it to its parent type), but also upcast it (cast it to an object of a child class), if the object is indeed an object of the child class. This means you need a way of determining whether an object of type `Parent` is really a `Child1` object that’s been cast down (in which case it can be cast up) or a `Child2` object (in which case it can’t be cast up). In the latest C++ compilers you can do the same with the `dynamic_cast<...>` operator. How to determine this? Each object will have to have a unique identification number, perhaps an index to a table of classes and their parents. By using this number you can always tell what kind of object it really is.

### 9.6.4 Type variables

Type variables Another fun thing is to allow type variables. This allows you to dynamically create objects of variable types. An example. Say you have a game with a cloning machine. An enemy walks in and two identical enemies walk out. You could use a big switch statement that contains all possible enemies, but that's not a very extensible approach. So you store the enemy's type and tell the game to create a monster of that type. In some imaginary language's code:

```
TypeVar<Enemy> enemytype;           // A type variable
enemytype = typeof (monster);       // Get the monster's type
Enemy *newmonster = new enemytype;  // Create a new monster of the same type
```

You can pass type variables to functions; this will make them very flexible indeed, as you can use the same function to create and manipulate many different kinds of objects!

For type variables you need to expand the table of classes and their parent class to include every class' size; otherwise you have no way of dynamically creating them.

## 9.7 Game-specific language constructs

UnrealScript was (as far as I know) the first language to offer two language features that are very useful in games: states and latent code.

### 9.7.1 States

Classes in UnrealScript can have several states; an object is always in exactly one state. Based on which state the object is in, different functions are executed for that object. So if the object is an enemy and it's in the Angry state, the Angry version of the function SeePlayer would be executed and the enemy would start attacking the player. If the enemy is in the Frightened state, another SeePlayer function (with the same parameter types) would be called and would make the enemy flee.

States aren't very hard to add, although it requires some work; the state is an extra (invisible) class member and whenever a state-specific function is called the appropriate version of the function is executed. This could easilt be done with a jump table, using the state number as an index.

States can have their own out-of-function code, known in UnrealScript as state code. This is handy in combination with the next construct: latent functions.

### 9.7.2 Latent functions

Latent functions are quite hard to implement, but very cool: a latent function takes some game time to execute; in other words, the process can start a function

Wait or Animate that starts waiting or animating the creature, and when the animation is done the code resumes execution. This is a great feature to have for AI scripts.

The problem with (and strength of) latent code is that it's essentially running in parallel with the other code. Every now and then a piece of latent code is executed, then it's stopped again. So we have to remember the latent code instruction pointer. And when the object changes states, you'll also want it to execute other latent code.

We can see the reason UnrealScript only allow latent functions to be called from state code, not from normal functions: if latent functions could be called from anywhere, every object could essentially have many 'threads' running in parallel.. this would require a lot of bookkeeping and would become slow. Also, synchronisation problems would occur: one object thread would set a member variable to a certain value, then another thread would become active and modify it again.. We would need to implement a full threading system if we wanted to allow this.

## 9.8 That's it for now...

Well, I hope this got your imagination going. There's lots of features your scripting language could offer; but you're going to have to limit yourself to the ones you really need if you ever want to finish it.

This was probably the last part of this tutorial series. I really enjoyed writing it! If you feel some aspect has not gotten enough attention, let me know, and maybe I'll do an extra part. Of course, if you have any other questions I'd also like to hear from you.

Good luck, and keep on scripting! ;-)

## 9.9 Quote

“He stared at it for some time as things began slowly to reassemble themselves in his mind. He wondered what he should do, but he only wondered it idly. Around him people were beginning to rush and shout a lot, but it was suddenly very clear to him that there was nothing to be done, not now or ever. Through the new strangeness of noise and light he could just make out the shape of Ford Prefect sitting back and laughing wildly.

A tremendous feeling of peace came over him. He knew that at last, for once and for ever, it was now all, finally, over.”

HHG 5:25