CATpiler
A Compiler for the LOLCODE language

Stephanie Stroka

stephanie.stroka@sbg.ac.at
Department of Applied Informatics
University of Salzburg

Abstract. This document describes the design and functionality of CATpiler - a compiler for the LOLCODE[1] language. In the following, I would like to give an overview over the input language characteristics, the lexical analysis and the parsing process, as well as the semantic analysis, code generation and memory management done by CATpiler. The project has been realized during the Compiler Construction course lead by Prof. Christoph Kirsch (2010).

1 Language Description

(!!!) LOLCODE Anpassung erklären! LOLCODE can be categorized as an esoteric programming language, which is under construction since May 2007. The main characteristics are the non-classic way of appointing keywords and prefix instead of infix operators and the optional type definition.

1.1 Syntax

The notion of *non-classic way of appointing keywords* gets clear if we have a look at the language comparision between (pseudo) C and LOLCODE syntax, shown in Listing 1. As one can see, keywords can contain blank characters, or in other words, a keyword may consist of several words.
There are two ways of dealing with multiple-word keywords regarding lexical analysis. The first is that we search for the whole keyword, e.g. *"HOW DUZ I"*, which has the advantage of eliminating ambiguous keyword-tokens like *"I"*. The second one is to scan for each single keyword token, e.g. *"HOW"*, *"DUZ"* and *"I"*, respectively, and leave the "real" keyword determination to the parsing process. The benefit that we get by choosing the second approach is to be able to better deal with literal errors and skipped keyword-tokens. I chose the second approach, which results in checking source code tokens against 73 diverse keyword tokens. Thus, the lexical syntax analysis is the process of checking source code words against 73 deterministic finite automatons (DFA) plus three more DFAs for character strings, integers and identifiers.

[1] http://www.lolcode.com

Token scanning is a depth-first search. We, therefore, go through the list of tokens and check the first character of the input string against the first character of the token string. Non-equal characters result in a remove of the token and the checking of the next token. If we found a token that matches the first character, we return it as a candidate token and check the remaining characters. For keywords, which start with equal characters, we may need to revert our choice of the candidate token and chose another one. It may also happen that a source code token starts with the same sequence as keyword token, but exceed the string length and are therefore recognized as identifiers.

1.2 Parsing

CATpiler uses a recursive-decent parser to validate the input source against its desired structure, defined in the Extended Backus-Naur Form (EBNF) (see Table 2). Each non-terminal is represented as a recursive function. To advance the speed of the parsing process, the Scanner assigns a numerical id for each found token, which is than compared in the parser functions. Some of those ids have an important characteristic, which is especially important if they are alternative tokens in an EBNF production. The ids for the keywords *"NUMBR"*, *"CHAR"*, *"TROOF"*, *"NUMBRZ"*, *"CHARZ"*, *"TROOFZ"*, *"SUM"*, *"PRODUKT"*, *"QUOSHUNT"*, *"BIGGR"*, *"SMALLR"* and *"DIFF"* are listed in Table 4 in binary form. EBNF alternative keywords share a common group ID.
The code-optimization trick is similar to the one proposed in [1]. Instead of comparing decimal integer ids by their range (e.g. *"NUMBR"* could have the id 1, *"TROOF"* the id 2 and *"CHAR"* the id 3, so that we would check whether an id is in between the range(1,3)), a logical AND operation is provoked on the keyword id and the check id. The result of the AND operation is either 0, if the keyword does not belong to the group, or the group id respectively.
Identifiers for the other keywords than those listed in Table 4 need to be assigned in a way such that a logical AND operation on the id with the check id does not return any of the group ids. To easily guarantee this behavior, we only assign ids that are bigger than 2^4, or in other words that do not set any bits in the last 4 bits of a word. The remaining 2^{28} bits are more than sufficient to be used as identifiers for other keywords.

1.3 Attributed Grammar

During the lexical analysis and the parsing process, the scanned keywords are extended by attributes represented as string values. Storing identifier names, string and integer values is the preliminary work for building a symbol table. The symbol table contains entries for each source scope, therefore preventing that identifiers are used that have not yet been defined, or that identifiers are defined multiple times. A symbol table entry contains information like `identifier name`, `attribute`, `stack address`, `heap address`, `register`, `type` (i.e. *NUMBR*, *CHARZ*, etc.) and `category` (i.e. *const*, *var*, *reg*, *heap*).

Another part of attributing source code is the collection of function definitions and function calls. Instead of storing information about called and implemented functions temporarily in the RAM during compilation, as we do with symbol tables, we write that information into the object code file, to enable separate compilation and late function implementation checking by the Linker.

1.4 Error Handling

Code generation is just performed, if the source code has not been marked as faulty. On the other hand it is important to continue parsing if an error has been detected, since it is more convenient for the programmer to come to know all his errors in the source code, and not just the first one.
Error handling is considered difficulty for the LOLCODE language, because multiple-word keywords provide many options to forget keywords or to introduce typing errors. CATpiler handles programmatic errors pragmatically and considers that keyword skipping happens more frequently. Therefore, a lot of lookahead operations, which give information about the next keyword without adjusting the source code pointer, are necessary. If a lookahead returns a keyword that is not expected, we simply check if the next keyword fits again. An improved version, e.g. to adjust the source pointer if the next keyword also does not fit, and therefore to assume that the keyword contains a typing error, remains future work.
If the error extend is to big to detect a certain statement production, we look up source code tokens until we find the next start of a statement, or until we reach the end of the file, flow-control statement or function. Thus, we can also say that statement begin and end keywords are the *strong symbols* of the LOLCODE language.

2 Code Generation

Immediate code generation is part of the parsing process. Every EBNF-production function immeditaley generates code for the output language, therefore enabling compilation in a single source code iteration. The following sections present the target language and code optimizations done by CATpiler.

2.1 Target Language

Although there exist a variety of interpreters and one compiler for the .NET platform, CATpiler is the first compiler for the MIPS32 architecture. If computes instructions in MIPS assembler, which can be interpreted and run on any MIPS simulator. The code has been tested for the MARS assembler and simulator platform[2].

[2] `http://courses.missouristate.edu/KenVollmar/MARS/`

3 Memory Management

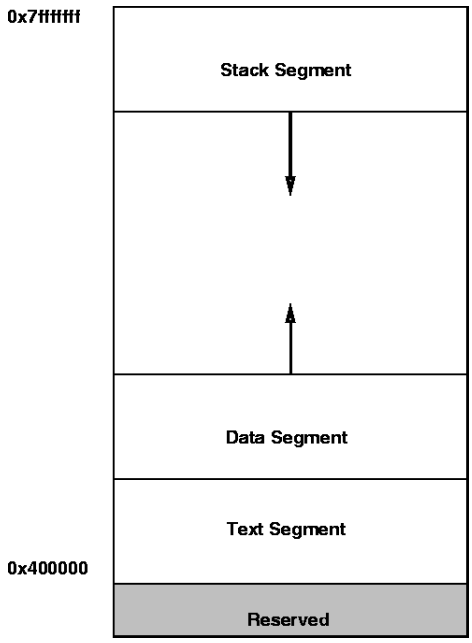

Fig. 1: MIPS Memory Layout

4 Separate Compilation

5 Testing and Execution

6 Conclusion

References

1. N. Wirth. *Grundlagen und Techniken des Compilerbaus*. Oldenbourg Wissenschaftsverlag, 2008.

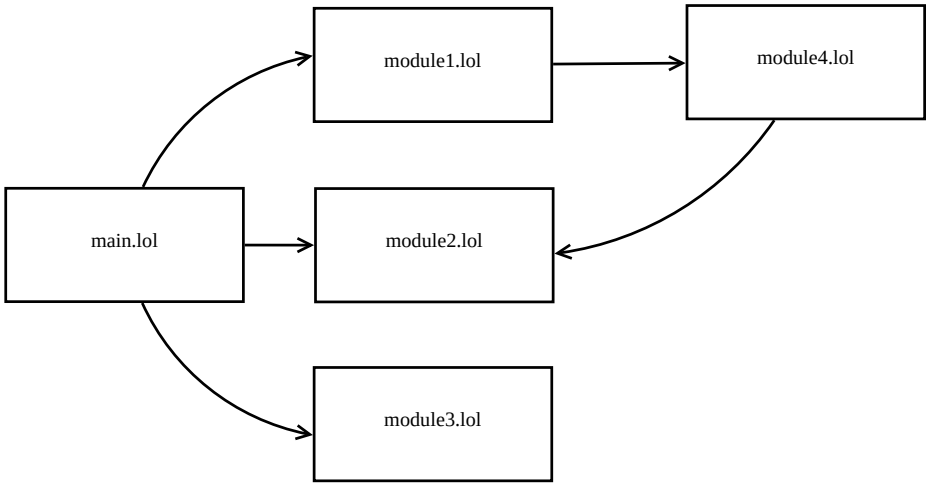

Fig. 2: Recursive Import of Module Files

C-alike syntax	**LOLCode syntax**
//	BTW
/* ... */	OBTW ... TLDR
#include < module.h >	CAN HAS module
main()	HAI
exit()	KTHXBYE
< type >< var >	I HAS A < var >
...	< var > IS NOW A < type >
< var >=< value >	< var > R < value >
char	CHAR
char[ ]	CHARZ
int	NUMBR
int[ ]	NUMBRZ
boolean	TROOF
boolean[ ]	TROOFZ
i[4]	i.4
someStruct.element	someStruct− >element
untyped	NOOB
true	WIN
false	FAIL
\n	:)
”	:”
< x > + < y >	SUM OF < x > AN < y >
< x > − < y >	DIFF OF < x > AN < y >
< x > ∗ < y >	PRODUKT OF < x > AN < y >
< x > / < y >	QUOSHUNT OF < x > AN < y >
max(< x >, < y >)	BIGGR OF < x > AN < y >
min(< x >, < y >)	SMALLR OF < x > AN < y >
< x > && < y >	BOTH OF < x > AN < y >
< x > || < y >	EITHER OF < x > AN < y >
! < x >	NOT < x >
$< x_1 >$ && $< x_2 >$ &&...&& $< x_i >$	ALL OF $< x_1 >$ AN $< x_2 >$ AN ... AN $< x_i >$ MKAY
$< x_1 >$ || $< x_2 >$ ||...|| $< x_i >$	ANY OF $< x_1 >$ AN $< x_2 >$ AN ... AN $< x_i >$ MKAY
< x >==< y >	BOTH SAEM < x > AN < y >
< x >! =< y >	DIFFRINT < x > AN < y >
< x >>=< y >	BOTH SAEM < x > AN BIGGR OF < x > AN < y >
< x ><=< y >	BOTH SAEM < x > AN SMALLR OF < x > AN < y >
< x >>< y >	DIFFRINT < x > AN BIGGR OF < x > AN < y >
< x ><< y >	DIFFRINT < x > AN SMALLR OF < x > AN < y >
if	ORLY?
then	YA RLY
elseif	MEBBE
else	NO WAI
end − of − if	OIC
loop	IM IN YR < label > YR < var > [TIL—WILE < expr >]
loop − end	IM OUTTA YR < label >
function_label(< arg1 >, arg2 > ...)	CAN U < function_label > [YR < arg1 > AN YR < arg2 > ...]
void function_label(< arg1 >, arg2 > ...){	HOW DUZ I < function_label > [YR < arg1 > AN YR < arg2 > ...]
} (function − end)	IF YOU SAY SO
struct < label >	STUFF < label >
struct − end	THATSIT
malloc()	DOWANT

Table 1: Sytax accepted by CATpiler

Non-Terminal	Production
<LETTER>	::= "a" | ... | "z" | "A" | ... | "Z" .
<DIGIT_NO_ZERO>	::= "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9" .
<DIGIT>	::= "0" | <DIGIT_NO_ZERO> .
<NUM>	::= <DIGIT_NO_ZERO> { <DIGIT> } .
<UNDERSCORE>	::= "_" .
<SPECIAL_CHAR>	::= " "| <UNDERSCORE> | <CTRL_CHAR> |" − "|"."|","|";"|" :: "|" :)"| "!"|" : ""|"$"|"%"|"&"|"/"|"("|")"|" = "|"?"|" \ "|"'"|" ∗ "|" + "|" > "|" < ".
<STRING>	::= """({<LETTER> | <DIGIT> | <SPECIAL_CHAR> }) """ .
<BOOL>	::= "WIN" | "FAIL" .
<IDENTIFIER>	::= <LETTER> { <LETTER> | <DIGIT> | <UNDERSCORE> } .
<TYPE>	::= "TROOF" | "NUMBR" | "CHAR" . "TROOFZ" | "NUMBRZ" | "CHARZ" .
<GEN_EXPR>	::= ("BOTH SAEM" | "DIFFRINT") <OPERATION> "AN" <OPERATION> .
<INF_EXPR>	::= ("ALL OF" | "ANY OF") <BOOL_OP> "AN" <BOOL_OP> ::= { "AN" <BOOL_OP> } "MKAY" .
<BI_EXPR>	::= <BOOL_OP> | <GEN_EXPR> .
<EXPR>	::= <BI_EXPR> | <INF_EXPR> | (<BOOL>| <IDENTIFIER>) .
<BOOL_OP>	::= (("BOTH OF" | "EITHER OF") <EXPR> "AN" <EXPR>) | ("NOT" <EXPR>) .
<STR_OP>	::= <STRING> | <IDENTIFIER> .
<NUM_OP>	::= ("SUM OF" | "DIFF OF" | "PRODUKT OF" | "QUOSHUNT OF" | "BIGGR OF" | "SMALLR OF") <NUM_OP> "AN" <NUM_OP> | (<NUM> | <IDENTIFIER>) .
<OPERATION>	::= <NUM_OP> | <BOOL_OP> | <STR_OP> .
<VAR_INIT>	::= "I HAS A" <IDENTIFIER> .
<VAR_DECL>	::= <IDENTIFIER> "IS NOW A" <TYPE> .
<VAR_ASSIGN>	::= <IDENTIFIER> "R" <OPERATION> .
<IF>	::= <EXPR> ORLY? YA RLY { <STATEMENT> } { MEBBE <EXPR> { <STATEMENT> } } [NO WAI { <STATEMENT> }] } OIC .
<LOOP>	::= "IM IN YR" <IDENTIFIER> [YR <IDENTIFIER>] [WILE|TIL <EXPR>] { <STATEMENT> } "IM OUTTA YR" <IDENTIFIER> .
<FLOW_CONTROL>	::= <IF> | <LOOP> .
<FUNC_CALL>	::= <IDENTIFIER> { <EXPR> } .
<STATEMENT>	::= <VAR_INIT> | <VAR_DECL> | <VAR_ASSIGN> | <OPERATION> | <FLOW_CONTROL> | <FUNC_CALL> .
<FUNCTION>	::= "HOW DUZ I" <IDENTIFIER> [YR <IDENTIFIER> {AN YR <IDENTIFIER> }] { <STATEMENT> } { "FOUND YR" <EXPR> | "GTFO"} "IF YOU SAY SO" .
<MODULE>	::= "CAN HAS" <IDENTIFIER> "?" .
<STRUCT>	::= "STUFF" { <VAR_INIT> "THATSIT" .
<MAIN>	::= "HAI" { <STATEMENT> } "KTHXBYE" .
<PROGRAMM>	::= { <MODULE> } { <STRUCT> } [<MAIN>] { <FUNCTION> } .

Table 2: Extended Backus-Naur-Form for LOLCODE

Keyword	Assigned Id	Group Id	Check Id
NUMBR	0000 0000 0000 0001	0000 0000 0000 0001	1111 1111 1111 1001
CHAR	0000 0000 0000 0011	0000 0000 0000 0001	1111 1111 1111 1001
TROOF	0000 0000 0000 0101	0000 0000 0000 0001	1111 1111 1111 1001
NUMBRZ	0000 0000 0000 0010	0000 0000 0000 0010	1111 1111 1111 0011
CHARZ	0000 0000 0000 0110	0000 0000 0000 0010	1111 1111 1111 0011
TROOFZ	0000 0000 0000 1010	0000 0000 0000 0010	1111 1111 1111 0011
SUM	0000 0000 0000 0100	0000 0000 0000 0100	1111 1111 1100 0111
PRODUKT	0000 0000 0001 0100	0000 0000 0000 0100	1111 1111 1100 0111
QUOSHUNT	0000 0000 0010 0100	0000 0000 0000 0100	1111 1111 1100 0111
DIFF	0000 0000 0000 1100	0000 0000 0000 0100	1111 1111 1100 0111
BIGGR	0000 0000 0001 1100	0000 0000 0000 0100	1111 1111 1100 0111
SMALLR	0000 0000 0010 1100	0000 0000 0000 0100	1111 1111 1100 0111
DIFF	0000 0000 0000 1100	0000 0000 0000 1100	1111 1111 1100 1111
BIGGR	0000 0000 0001 1100	0000 0000 0000 1100	1111 1111 1100 1111
SMALLR	0000 0000 0010 1100	0000 0000 0000 1100	1111 1111 1100 1111

Table 3: Identifier Assignment for EBNF alternative keywords

Instruction	Description	Operation	Syntax
add	Adds two registers	$\$d = \$s + \$t$	add $d $s $t
addi	Adds immediate value to register	$\$d = \$s + i$	addi $d $s i
sub	Subtracts $t from $s	$\$d = \$s - \$t$	sub $d $s $t
subi	Subtracts immediate value from register	$\$d = \$s - i$	subi $d $s i
mul	Multiplies two registers	$\$d = \$s * \$t$	mul $d $s $t
muli	Multiplies immediate value to register	$\$d = \$s * i$	muli $d $s i
div	Divides $t from $s	$\$d = \$s \ / \ \$t$	div $d $s $t
divi	Divides immediate value from register	$\$d = \$s \ / \ i$	divi $d $s i
and	AND-operation on two registers	$\$d = \$s \ \&\& \ \$t$	and $d $s $t
andi	AND-operation on register and immediate value	$\$d = \$s \ \&\& \ i$	andi $d $s i
or	OR-operation on two registers	$\$d = \$s \ \| \| \ \$t$	and $d $s $t
ori	OR-operation on register and immediate value	$\$d = \$s \ \| \| \ i$	andi $d $s i
sll	Shift left-operation on two registers	$\$d = \$s \ \| \| \ \$t$	and $d $s $t
lw	Loads word from address	$\$d = address$	lw $d a
lb	Loads byte from address	$\$d = address$	lb $d a
sw	Stores word in address	$\$d = address$	sw $d a
sb	Stores byte in address	$\$d = address$	sb $d a
slt	If $s ¡ $t, set $d = 1, otherwise $d = 0		slt $d $s $t
beq	Branches if both registers are equal		beq $s $t label
bne	Branches if both registers are not equal		bne $s $t label
bgez	Branches if the register is greater than or equal zero		bgez $s label
bgtz	Branches if the register is greater than zero		bgtz $s label
bltz	Branches if the register is less than zero		bltz $s label
j	Jumps to label		j label
jal	Stores pc in $ra register and jumps to label		jal label

Table 4: MIPS Instructions

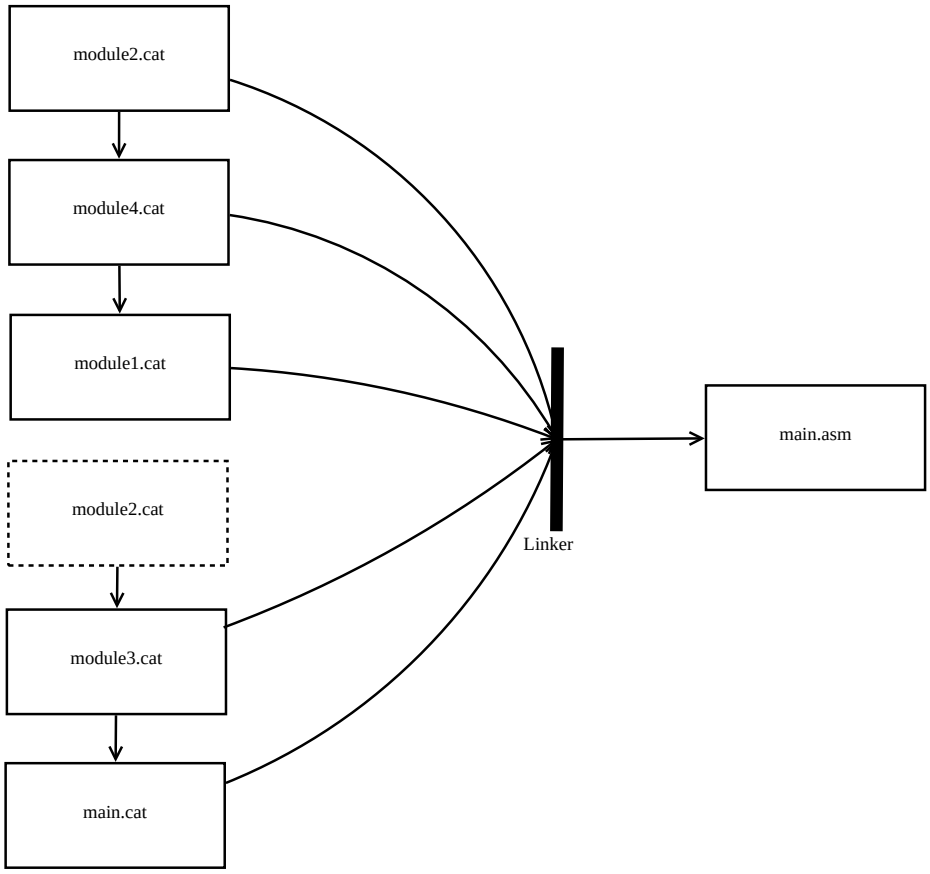

Fig. 3: The Linking Process