Theory of Computation and Compiler Design

PROJECT TITLE: DEEP ANALYSIS ON TINY-C COMPILER

Faculty: Boominathan P Sir
(School of Computer Engineering)

Submitted by-

Archit Jain
16BCE2318
G1+TG1 SLOT

Deep Analysis on Tiny-C Compiler

ABSTRACT

In this project, we are going to discuss about a compiler for Tiny-C language (simplified version of language C). We will be talking about its various phases from converting a simplified C language into x86 assembly code. So. basically a compiler is a software that converts computer code or language from one programming language (source language) into the other programming language (target language). They are basically a type of translator, as it translates a high-level programming language to a lower-level language, so that an executable program can be created. There are different types of compiler available for different OS or CPU. If a compiler runs on different CPU and OS, it is called as cross compiler.
So, a compiler performs many operations, such as **preprocessing, lexical analysis, parsing, semantic analysis, generation of intermediate representation, code optimization and code generation.**
Tiny-C compiler is a small but hyper fast compiler for C language. It is a self-relying compiler, as there is no need for you to connect any external assembler or linker because Tiny C compiler do itself.
So basically Tiny-C compiler works upon these steps:

- • Step1: File Reading.
- • Step 2: lexical analysis.
- • Step 3: Parsing the Grammar
- • Step 4: Generating the Three-Address Code
- • Step 5: Assembly code generation and executable file generation.
- • Step 6: Optimization is also done.

So, in this project we are going to learn this about, how a Tiny-C compiler works, its source code, and different parts of this compiler.

1. Lexer for TinyC

Notation

In the syntax notation used here, syntactic categories (non-terminals) are indicated by italic type, and literal words and character set members (terminals) by bold type. A colon (:) following a non-terminal introduces its definition.
Alternative definitions are listed on separate lines, except when prefaced by the words "one of". An optional symbol is indicated by the subscript "opt", so that the following indicates an optional expression enclosed in braces.
{ expression$_{opt}$ }

Lexical Grammar of TinyC

I. **Lexical elements**
   1. **Keywords**
   2. **Identifiers**
   3. **Constants**
   4. **String literals**
   5. **Punctuators**

II. **Comments**

Keywords

keyword: one of

auto	enum	restrict	unsigned
break	extern	return	void
case	float	short	volatile
char	for	signed	while
const	goto	sizeof	_Bool
continue	if	static	_Complex
default	inline	struct	_Imaginary
do	int	switch	
double	long	typedef	
else	register	union	

Identifiers

1. **Non-Digit Identifiers**

_	a	b	c	d	e	f	g	h	i	j	k	l	m
	n	o	p	q	r	s	t	u	v	w	x	y	z
	A	B	C	D	E	F	G	H	I	J	K	L	M
	N	O	P	Q	R	S	T	U	V	W	X	Y	Z

2. **Digit-Identifiers**

0 1 2 3 4 5 6 7 8 9

Constants

1. **Integer-Constant**
2. **Floating-Constant**
3. **Enumeration-constant**
4. **Character-constant**

String Literals

string-literal:
" *s-char-sequence*$_{opt}$ "
s-char-sequence:
s-char
s-char-sequence s-char
s-char:
any member of the source character set except
the double-quote ", backslash \, or new-line character
escape-sequence

Punctuators

```
[   ]   (   )   {   }   .   ->
++   --   &   *   +   -   ~   !
/   %   <<   >>   <   >   <=   >=   ==   !=   ^   |   &&   ||
?   :   ;   ...
=   *=   /=   %=   +=   -=   <<=   >>=   &=   ^=   |=
,   #
```

Comments

- **Single-Line comment**

The characters // introduce a comment that includes all multibyte characters up to, but not including, the next-line character.

- **Multi-line comment**

The character /* introduce a comment and */ is used to terminate it.
Thus, /*…. */ comments do not nest.

2. Parser for TinyC

1. **Expressions**

primary-expression:
identifier
constant
string-literal
(*expression*)
postfix-expression:
primary-expression
postfix-expression [*expression*]
postfix-expression ($argument\text{-}expression\text{-}list_{opt}$)
postfix-expression . *identifier*
postfix-expression $->$ *identifier*
postfix-expression ++
postfix-expression −−
(*type-name*) { *initializer-list* }
(*type-name*) { *initializer-list* , }
argument-expression-list:
assignment-expression
argument-expression-list , *assignment-expression*
unary-expression:
postfix-expression
++ *unary-expression*
−− *unary-expression*
unary-operator cast-expression
sizeof *unary-expression*
sizeof (*type-name*)
unary-operator: one of

& * + - ~ !

cast-expression:
unary-expression
(*type-name*) *cast-expression*
multiplicative-expression:
cast-expression
multiplicative-expression * *cast-expression*
multiplicative-expression / *cast-expression*
multiplicative-expression % *cast-expression*

additive-expression:
multiplicative-expression
additive-expression + *multiplicative-expression*
additive-expression − *multiplicative-expression*
shift-expression:
additive-expression
shift-expression << *additive-expression*
shift-expression >> *additive-expression*
relational-expression:
shift-expression
relational-expression < *shift-expression*
relational-expression > *shift-expression*
relational-expression <= *shift-expression*
relational-expression >= *shift-expression*
equality-expression:
relational-expression
equality-expression == *relational-expression*
equality-expression ! = *relational-expression*
AND-expression:
equality-expression
AND-expression & *equality-expression*
exclusive-OR-expression:
AND-expression
*exclusive-OR-expression*ˆ *AND-expression*
inclusive-OR-expression:
exclusive-OR-expression
inclusive-OR-expression | *exclusive-OR-expression*
logical-AND-expression:
inclusive-OR-expression
logical-AND-expression && *inclusive-OR-expression*
logical-OR-expression:
logical-AND-expression
logical-OR-expression || *logical-AND-expression*
conditional-expression:
logical-OR-expression
logical-OR-expression ? *expression* : *conditional-expression*
assignment-expression:
conditional-expression
unary-expression assignment-operator assignment-expression
assignment-operator: one of

```
= *= /= %= += -= <<= >>= &= ^= |=
```

expression:
assignment-expression
expression , *assignment-expression*
constant-expression:
conditional-expression

2. Declarations

declaration:
declaration-specifiers *init-declarator-list*$_{opt}$;
declaration-specifiers:
storage-class-specifier *declaration-specifiers*$_{opt}$
type-specifier *declaration-specifiers*$_{opt}$
type-qualifier *declaration-specifiers*$_{opt}$
function-specifier *declaration-specifiers*$_{opt}$
init-declarator-list:
init-declarator
init-declarator-list , *init-declarator*
init-declarator:
declarator
declarator = *initializer*

storage-class-specifier:
extern
static
auto
register
type-specifier:
void
char
short
int
long
float
double
signed
unsigned
_Bool
_Complex
_Imaginary
enum-specifier
specifier-qualifier-list:
type-specifier *specifier-qualifier-list*$_{opt}$
type-qualifier *specifier-qualifier-list*$_{opt}$
enum-specifier:
enum *identifier*$_{opt}$ { *enumerator-list* }
enum *identifier*$_{opt}$ { *enumerator-list* , }
enum *identifier*
enumerator-list:
enumerator
enumerator-list , *enumerator*
enumerator:
enumeration-constant
enumeration-constant = *constant-expression*

type-qualifier:
const
restrict
volatile
function-specifier:
inline
declarator:
pointer$_{opt}$ *direct-declarator*
direct-declarator:
identifier
(*declarator*)
direct-declarator [*type-qualifier-list*$_{opt}$ *assignment-expression*$_{opt}$]
direct-declarator
[**static** *type-qualifier-list*$_{opt}$ *assignment-expression*]
direct-declarator [*type-qualifier-list* **static** *assignment-expression*]
direct-declarator [*type-qualifier-list*$_{opt}$ *]
direct-declarator (*parameter-type-list*)
direct-declarator (*identifier-list*$_{opt}$)
pointer:
* *type-qualifier-list*$_{opt}$
* *type-qualifier-list*$_{opt}$ *pointer*
type-qualifier-list:
type-qualifier
type-qualifier-list type-qualifier
parameter-type-list:
parameter-list
parameter-list , ...
parameter-list:
parameter-declaration
parameter-list , *parameter-declaration*

parameter-declaration:
declaration-specifiers declarator
declaration-specifiers
identifier-list:
identifier
identifier-list , *identifier*
type-name:
specifier-qualifier-list
initializer:
assignment-expression
{ *initializer-list* }
{ *initializer-list* , }
initializer-list:
designation$_{opt}$ *initializer*
initializer-list , *designation*$_{opt}$ *initializer*
designation:
designator-list =

designator-list:
designator
designator-list designator
designator:
[*constant-expression*]
. *identifier*

3. Statements

statement:
labeled-statement
compound-statement
expression-statement
selection-statement
iteration-statement
jump-statement
labeled-statement:
identifier : *statement*
case *constant-expression* : *statement*
default : *statement*
compound-statement:
{ $block\text{-}item\text{-}list_{opt}$ }
block-item-list:
block-item
block-item-list block-item
block-item:
declaration
statement
expression-statement:
$expression_{opt}$;
selection-statement:
if (*expression*) *statement*
if (*expression*) *statement* **else** *statement*
switch (*expression*) *statement*
iteration-statement:
while (*expression*) *statement*
do *statement* **while** (*expression*) ;
for ($expression_{opt}$; $expression_{opt}$; $expression_{opt}$) *statement*
for (*declaration* $expression_{opt}$; $expression_{opt}$) *statement*
jump-statement:
goto *identifier* ;
continue ;
break ;
return $expression_{opt}$;

4. **External definitions**

translation-unit:
external-declaration
translation-unit external-declaration
external-declaration:
function-definition
declaration
function-definition:
declaration-specifiers declarator declaration-list$_{opt}$ compound-statement
declaration-list:
declaration
declaration-list declaration

3. Machine Independent Code Generator for TinyC

Scope of Machine-Independent Translation

Expression Phase

Support all arithmetic, shift, relational, bit, logical (Boolean), and assignment expressions excluding:

1. sizeof operator
2. Comma (,) operator
3. Compound assignment operators

*= /= %= += -= <<= >>= &= ^= |=

Support only simple assignment operator (=)

4. Structure component expression

Declaration Phase

Support for declarations should be provided as follows:

1. Simple variable, pointer, array, and function declarations should be supported. For example, the following would be translated:

double d = 2.3;
int i, w [10];
*int a = 4, *p, b;*
void func (int i, double d);
char c;

2. Consider only **void, char, int**, and **double** type-specifiers. As specified in C, char and int are to be taken as signed.

For computation of offset and storage mapping of variables, assume the following sizes (in bytes) of types:

Type	Size	Remarks
void	*undefined*	
char	1	
int	4	
double	8	
void *	4	*All pointers have same size*

It may also help to support an implicit bool (Boolean) type with constants 1 (TRUE) and 0 (FALSE). This type may be inferred for a logical expression or for an int expression in logical context. Note that the users cannot define, load, or store variables of bool type explicitly, hence it is not storable and does not have a size.

3. Initialization of arrays may be skipped.

4. storage-class-specifier, enum-specifier, type-qualifier, and function-specifier may be skipped.

5. Function declaration with only parameter type list may be skipped. Hence,
void func (int i, double d);
should be supported while
void func (int, double);
may not be.

Statement Phase

Support all statements excluding:
1. Declarations within **compound-statement** (block).
2. Declaration within **for**.
3. All Labelled statements (**labeled-statement**).
4. **Switch** in selection-statement.
5. All Jump statements (jump-statement) except **return**.

The 3-Address Code

Use the 3-Address Code specification as discussed in the class. For easy reference the same is reproduced here. Every 3-Address Code:

1. Uses only up to 3 addresses.
2. Is represented by a quad comprising - opcode, argument 1, argument 2, and result; where argument 2 is optional.

Address Type

Name: Source program names appear as addresses in 3-Address Codes.

1. Constant: Many different types and their (implicit) conversions are allowed as deemed addresses.
2. Compiler-Generated Temporary: Create a distinct name each time a temporary is needed - good for optimization.

Instruction Types

For Addresses x, y, z, and Label L

1. Binary Assignment Instruction: For a binary op (including arithmetic, shift, relational, bit, or logical operators):

x = y op z

2. Unary Assignment Instruction: For a unary operator op (including unary minus or plus, logical negation, bit, and conversion operators):

x = op y

3. Copy Assignment Instruction:

x = y

4. Unconditional Jump:

goto L

5. Conditional Jump:

~ Value-based:

if x goto L
ifFalse x goto L

~ Comparison-based: For a relational operator **op** (including <, >, == ,! =, >=, <=):

if x relop y goto L

6. Procedure Call: A procedure call ***p(x1, x2, ..., xN)*** having N>= 0 parameters is coded as (for addresses p, x1, x2, and xN):

param x1
param x2
...
param xN
y = call p, N

Note that N is not redundant as procedure calls can be nested.

7. Return Value: Returning a return value and / or assigning it is optional.
If there is a return value v it is returned from the procedure p as:

return v

8. Indexed Copy Instructions:

x = y[z]
x[z] = y

9. Address and Pointer Assignment Instructions:

x = &y
x = *y
****x = y***

Symbol table

Symbol Table Use symbol tables for user-defined (including arrays and pointers) variables, temporary variables and functions.

Name	Type	Initial Value	Size	Offset	Nested Table
...	...	...	...	...	...

Global Functions:

Following (or similar) global functions and more may be needed to implement the semantic actions:

`makelist(i)` A global function to create a new list containing only `i`, an index into the array of `quad`'s, and to return a pointer to the newly created list.
`merge(p1, p2)` A global function to concatenate two lists pointed to by `p1` and `p2` and to return a pointer to the concatenated list.
`backpatch(p, i)` A global function to insert `i` as the target label for each of the `quad`'s on the list pointed to by `p`.
`typecheck(E1, E2)` A global function to check if `E1` & `E2` have same types (that is, if `<type_of_E1> = <type_of_E2>(E)`). If not, then to check if they have compatible types (that is, one can be converted to the other), to use an appropriate conversion function `conv<type_of_E1>2<type_of_E2>(E)` or `conv<type_of_E2>2<type_of_E1>(E)` and to make the necessary changes in the Symbol Table entries. If not, that is, they are of incompatible types, to throw an exception during translation.
`conv<type1>2<type2>(E)` A global function to convert[a] an expression `E` from its current type `type1` to target type `type2`, to adjust the attributes of `E` accordingly, and finally to generate additional codes, if needed.

4. Target Code Generator for TinyC

In this we will write a target code translator from the array of 3{Address quad's (with the supporting symbol table, and other auxiliary data structures) to the assembly language of x86-32. The translation is now machine-specific and your generated assembly code would be translated with the gcc assembler to produce the final executable codes for the TinyC program.

Scope of Target Translation

For simplicity restrict TinyC further:
1. Skip shift and bit operators.
2. Support only void, int, and char types. Skip double type.
3. Support only one-dimensional arrays.
4. Support only void, int, char, void*, int*, and char* types for returns types of functions.
5. No type conversion to be supported.

For I/O, provide a library (as created in Assignment 2) using in-line assembly language program of x86-32 along with int $128 (software interrupt) for gcc assembler.:
- ***int prints(char *)*** - prints a string of characters. The parameter is terminated by `\0'. The return value is the number of characters printed.
- ***int printi(int n)*** - prints the integer value of n (no newline). It returns the number of characters printed.
- ***int readi(int *eP)*** - reads an integer (signed) and returns it. The parameter is for error (ERR = 1, OK = 0).

The header file myl.h of the library will be as follows:

```
#ifndef _MYL_H
#define _MYL_H
#define ERR 1
#define OK 0
int prints(char *);
int printi(int);
int readi(int *eP); // *eP is for error, if the input is not an integer
#endif
```

Design of a translator

The translation comprises the following major steps only:

Memory Binding: This deals with the design of the allocation schema of variables (including parameters and constants) that associates each variable to the respective address expression or register. This needs to handle the following:

Handle local variables, parameters, and return value for a function.
These are automatic and reside in the Activation Record (AR) of the function. Various design schema for AR are possible based on the calling sequence protocol. A sample AR design could be as follows:

Offset	Stack Item	Responsibility
–ve	Saved Registers	Callee Saves & Restores
–ve	Callee Local Data	Callee defines and uses
0	Base Pointer of Caller	Callee Saves & Restores
	Return Address	Saved by call, used by ret
+ve	Return Value	Callee writes, Caller reads
+ve	Parameters	Caller writes, Callee reads

Handle global variables (note that local static variables are not allowed in TinyC) as static and generate allocations in static area.
1. Generate Constants from Table of Constants - handle string constants as assembler symbols in Data Segments and integer constants as parts of target code (Text Segment)
2. Register Allocations & Assignment: Create memory binding for variables in registers:
- After a load / store the variable on the activation record and the register have identical values
- Registers can be used to store temporary computed values
- Register allocations are often used to pass int or pointer parameters
- Register allocations are often used to return int or pointer values.

Code Translation
This deals with the translation of 3{Address quad's to x86-32 assembly code. This needs to handle:
1. Generation of Function Prologue - few lines of code at the beginning of a function, which prepare the stack and registers for use within the function.
2. Generate Function Epilogue - appears at the end of the function, and restores the stack and registers to the state they were in before the function was called.
3. Map 3{Address Code to Assembly - to translate the function body do:
- Choose optimized assembly instructions for every expression, assignment and control quad.
- Use algebraic simplification & reduction of strength for choice of assembly instructions from a quad.
- Use Machine Idioms (**like inc for i++ or ++i in place of add reg, 1**).