ISSN 2250-3153

Compiler Construction

Aastha Singh[*], Sonam Sinha[**], Archana Priyadarshi[**]

[*] Department of Computer Science, Institute of Technology& Management,GIDA Gorakhpur (U.P.)

Abstract- Compiler construction is a widely used software engineering exercise, but because most students will not be compiler writers, care must be taken to make it relevant in a core curriculum. The course is suitable for advanced undergraduate and beginning graduate students. Auxiliary tools, such as generators and interpreters, often hinder the learning: students have to fight tool idiosyncrasies, mysterious errors, and other poorly educative issues. We introduce a set of tools especially designed or improved for compiler construction educative projects in C.We also provide suggestions about new approaches to compiler Construction. We draw guidelines from our experience to make tools suitable for education purposes. The final result of this paper is to provide a general knowledge about compiler design and implementation and to serve as a springboard to more advanced courses. Although this paper concentrates on the implementation of a compiler, an outline for an advanced topics course that builds upon the compiler is also presented by us.

Index Terms- Lex,Yacc Parser,Parser-Lexer,Symptoms &Anomalies.

I. INTRODUCTION

Computer programs are formulated in a programming language and specify classes of computing processes. Computers, however, interpret sequences of particular instructions, but not program texts. Therefore, the program text must be translated into a suitable instruction sequence before it can be processed by a computer. This translation can be automated, which implies that it can be formulated as a program itself. The translation program is called a *compiler*, and the text to be translated is called *source text* (or sometimes *source code*). Compilers and operating systems constitute the basic interfaces between a programmer and the machine. Compiler is a program which converts high level programming language into low level programming language or source code into machine code. It focuses attention on the basic relationships between languages and machines. Understanding of these relationships eases the inevitable transitions to new hardware and programming languages and improves a person's ability to make appropriate trade off in design and implementation. Many of the techniques used to construct a compiler are useful in a wide variety of applications involving symbolic data. In particular, every man-machine interface is a form of programming language and the handling of input involves these techniques. The term compilation denotes the conversion of an algorithm expressed in a human-oriented source language to an equivalent algorithm expressed in a hardware-oriented target language.We shall be concerned with the engineering of compilers their organization, algorithms, data structures and user interfaces.The term compilation denotes the conversion of an algorithm expressed in a human-oriented source language to an equivalent algorithm expressed in a hardware-oriented target language. We shall be concerned with the engineering of compilers their organization, algorithms, data structures and user interfaces.It is not difficult to see that this translation process from source text to instruction sequence requires considerable effort and follows complex rules. The construction of the first compiler for the language Fortran(formula translator) around 1956 was a daring enterprise, whose success was not at all assured. It involved about 18 man years of effort, and therefore figured among the largest programming projects of the time.

The term compilation denotes the conversion of an algorithm expressed in a human-oriented source language to an equivalent algorithm expressed in a hardware-oriented target language. We shall be concerned with the engineering of compilers (their organization, algorithms, data structures and user interfaces Programming languages are tools used to construct formal descriptions of finite computations (algorithms). Each computation consists of operations that transform a given initial state into some final state. A programming language provides essentially three components for describing such computations:

- Data types, objects and values with operations defined upon them.
- Rules fixing the chronological relationships among specified operations.
- Rules fixing the (static) structure of a program

II. LEX AND YACC

A. Availability

Lex and yacc were both developed at Bell.T. Laboratories in the 1970s. Yaccwas the first of the two, developed by Stephen C. Johnson. Lex wasdesigned by Mike Lesk and Eric Schmidt to work with yacc. Both lex andyacc have been standard UNIX utilities since 7th Edition UNIX. System Vand older versions of BSD use the original AT&T versions, while the newestversion of BSD uses flex (see below) and Berkeley yacc. The articles writtenby the developers remain the primary source of information on lex andyacc.

In programs with structured input, two tasks that occur over and over aredividing the input into meaningful units, and then discovering the relationshipamong the units. For a text search program, the units would probablybe lines of text, with a distinction between lines that contain a match of thetarget string and lines that don't. For a C program, the units are variablenames, constants, strings, operators, punctuation, and so forth.This divisioninto units (which are usually called tokens) is

known as lexical analysis,or lexing for short. Lex helps you by taking a set of descriptions of possibletokens and producing a C routine, which we call a lexical analyzer, ora lexer, or a scanner for short, that can identify those tokens. The set ofdescriptions you give to lex is called a lex specification

Sample program. :program for lex

```
%{
#include "y.tab.h"
#include <stdlib.h>
void yyerror(char *);
%}
%%
[0-9]+
{               yylval = atoi(yytext);  return INTEGER; }
[-/*+\n]          { return *yytext; }
[ \t];
               yyerror("Unknown character");
%%
int yywrap(void) {    return 1;}
```

program for independent parser

```
%{
#include <stdio.h>
int yylex(void);
void yyerror(char *);
%}
%token INTEGER
%%
program     :   program expr '\n'        { printf("%d\n", $2); }
          | ;
expr        :   INTEGER
          | expr '+' expr          { $$ = $1 + $3; }
          | expr '*' expr          { $$ = $1 * $3; }
          | expr '-' expr              { $$ = $1 - $3; }
          | expr '/' expr              { $$ = $1 / $3; }
          ;
%%
void yyerror(char *s)
{
printf(stderr, "%s\n", s);
}
int main(void)
{
yyparse();
return 0;
}
```

B. Grammar

For some applications, the simple kind of word recognition we've alreadydone may be more than adequate; others need to recognize specificsequences of tokens and perform appropriate actions. Traditionally, adescription of such a set of actions is known as a ***grammar.***

Parser-Lexer Communication

When you use a lex scanner and a yacc parser together, the parser is thehigher level routine. It calls the lexer **yylex()** whenever it needs a tokenfrom the input. The lexer then scans through the input recognizing tokens.As soon as it finds a token of interest to the parser, it returns to the parser,returning the token's code as the value of **yyfex().**Not all tokens are of interest to the parser-in most programming languagesthe parser doesn't want to hear about comments and whitespace,for example. For these ignored tokens, the lexer doesn't return so that itcan continue on to the next token without bothering the parser.The lexer and the parser have to agree what the token codes are. We solvethis problem by letting yacc define the token codes. The tokens in ourgrammar are the parts of speech: NOUN, PRONOUN, **VERB, ADVERB, ADJECTIVE,**PREPOSITION, and CONJUNCTION. Yacc defines each of these as asmall integer using a preprocessor ***#define,*** Here are the definitions it used

in this example:

```
# define NOUN 257
# define PRONOUN 258
# define VERB 259
# define ADVERB 260
# define ADJECTIVE 261
# define PREPOSITICN 262
# define cXwUNCTICN 263
```

Token code zero is always returned for the logical end of the input. Yaccdoesn't define a symbol for it, but you can yourself if you want.

The Parts of Speech Lexer

Example : shows the declarations and rules sections of the new lexer.

Example : lexer to be called from the parser

```
%{
/*
* We now build a lexical analyzer to be used by a higher-level parser.
* /
#include ""y.tab.hn /* token codes from the parser */
#define LOOKUP 0 /* default - not a defined word type. */
int state;
\n { state = LOOKUP; 1
\.\n I state = LOOKUP;
```

Example : lexer to be called from the parser (continued)

```
return 0; /* end of sentence */
1
lrerb ( state = VERB; 1
^adj { state = ADJECTIVE; 1
"adv { state = ADVERB; 1
"noun { state = NOUN; 1
Prep { state = PREPOSITION; 1
pron { state = FTUXOUN; 3
"conj { state = CONJUNCTI~; 1
[a-zA-Z]+ {
if (state != LOOKUP) {
add-word(state, yytext) ;
) else I
switch (lookUpPword (yytext) ) {
case VERB:
```

```
return (VERB) ;
case ALXEXTIVE:
return (ALUBTIVE) ;
case ADVERB:
return (ADVERB);
case NOUN:
return (NOUN) ;
case PREPOSITION:
return (PREPOSITION);
case PRONOUN:
return (PRONOUN) ;
case CONmION:
return (CONJUWTION) ;
default :
printf("%s: don't reccgnize\nu, yytext):
/* don't return, just ignore it */
}
   }
      }
% %
... same add-word() and lookup.word() as before ...
```

There are several important differences here. We've changed the part ofspeech names used in the lexer to agree with the token names in theparser. We have also added return statements to pass to the parser thetoken codes for the words that it recognizes. There aren't any return statementsfor the tokens that define new words to the lexer, since the parserdoesn't care about them.

A Yacc Parser

Example 1-7 introduces our first cut at the yacc grammar.

Example 1-7: Simple yacc sentence parser

```
% t
/*
* A lexer for the basic g r m to use for recognizing mlish sentences.
/
#include <stdio.h>
% 1
%token NOUN PRCXWUN VERB AIXlERB ADJECl'IVE J3EPOSITIM CONJUNCTIM
% %
sentence: subject VERB object( printf("Sentence is valid.\nn); )
subject: NOUN
I PRONOUN
object: NOUN
extern FILE win;
main ( )
(
while ( !f eof (yyin)) {
yparse( ) ;
```

example : Simple yacc sentence parser (continued)

```
yyerror ( s)
char *s;
fprintf (stderr, "%s\na , s) ;
}
```

The structure of a yacc parser is, not by accident, similar to that of a lexlexer. Our first section, the definition section, has a literal code block,enclosed in "%{" and "%I". We use it here for a C comment (as with lex, Ccomments belong inside C code blocks, at least within the definition section)and a single include file.

The Rules Section

In our grammar we use the special character " I ", which introduces a rulewith the same left-hand side as the previous one. It is usually read as "or,"e.g., in our grammar a subject can be either a NOUN or a PRONOUN. Theaction part of a rule consists of a C block, beginning with "{" and endingwith "{". The parser executes an action at the end of a rule as soon as the
rule matches. In our sentence rule, the action reports that we've successfullyparsed a sentence. Since sentence is the top-level symbol, the entireinput must match a sentence. The parser returns to its caller, in this casethe main program, when the lexer reports the end of the input. Subsequentcalls to yyparse() reset the state and begin processing again. Our exampleprints a message if it sees a "subject VERB object" list of input tokens. Whathappens if it sees "subject subject" or some other invalid list of tokens? Theparser calls yyerroro, which we provide in the user subroutines section,and then recognizes the special rule error. You can provide error recoverycode that tries to get the parser back into a state where it can continue parsing, If error recovery fails or: as is the case here, there is no error recovery
code, yyparse() returns to the caller after it finds an error.

C. Storage Management

In this section weshall discuss management of storage for collections of objects, including temporary variables,during their lifetimes. The important goals are the most economical use of memory and thesimplicity of access functions to individual objects. Source language properties govern thepossible approaches, as indicated by the following questions :

- Is the exact number and size of all objects known at compilation time?
- Is the extent of an object restricted, and what relationships hold between the extentsof distinct objects (e.g. are they nested)?
- Does the static nesting of the program text control a procedure's access to global objects,or is access dependent upon the dynamic nesting of calls?

Static Storage Management

We speak of static storage management if the compiler can provide fixed addresses for allobjects at the time the program is translated (here we assume that translation includesbinding), i.e. we can answer the first question above with 'yes'. Arrays with dynamic bounds,recursive procedures and the use of anonymous objects are prohibited. The condition is fulfilled for languages like FORTRAN and BASIC, and for the objects lying on the outermostcontour of an ALGOL 60 or Pascal program. (In contrast, arrays with dynamic bounds canoccur even in the outer block of an ALGOL 68 program.)If the storage for the elements of an array with dynamic bounds is managed separately,the condition can be forced to hold in this case also. That is particularly interesting when wehave additional information that certain procedures are not recursive, for example becauserecursivity must be noted specially (as in PL/1) or because we have determined it fromanalysis of the procedure calls. We can then allocate storage statically for contours

otherthan the outermost.Static storage allocation is particularly valuable on computers that allow access to anylocation in main memory via an absolute address in the instruction. Here, static storage correspondsexactly to the class of objects with direct access paths .If, however, it is unknown during code generation whether or not an object is directly addressable(as on the IBM 370) because this depends upon the _nal addressing carried outduring binding, then we must also access statically-allocated objects via a base register. Theonly advantage of static allocation then consists of the fact that no operations for storagereservation or release need be generated at block or procedure entry and exit.

Dynamic Storage Management Using a Stack

All declared values in languages such as Pascal andSIMULA have restricted lifetimes. Further, the environments in these languages are nested:The extent of all objects belonging to the contour of a block or procedure ends before that ofobjects from the dynamically enclosing contour. Thus we can use a stack discipline to managethese objects: Upon procedure call or block entry, the activation record containing storage forthe local objects of the procedure or block is pushed onto the stack. At block end, procedurereturn or a jump out of these constructs the activation record is popped of the stack. (Theentire activation record is stacked, we do not deal with single objects individually!)An object of automatic extent occupies storage in the activation record of the syntacticconstruct with which it is associated. The position of the object is characterized by the base address, b, of the activation record and the relative location offset), R, of its storage withinthe activation record. R must be known at compile time but b cannot be known (otherwisewe would have static storage allocation). To access the object, b must be determined at runtime and placed in a register. R is then either added to the register and the result usedas an indirect address, or R appears as the constant in a direct access function of the form'register+constant'.The extension, whichmay vary in size from activation to activation, is often called the second order storage of theactivation record. Storage within the extension is always accessed indirectly via informationheld in the static part; in fact, the static part of an object may consist solely of a pointer tothe dynamic part.

Dynamic Storage Management Using a Heap

The last resort is to allocate storage on a heap: The objectsare allocated storage arbitrarily within an area of memory. Their addresses are determined atthe time of allocation, and they can only be accessed indirectly. Examples of objects requiringheap storage are anonymous objects such as those created by the Pascal new function andobjects whose size changes unpredictably during their lifetime. (Linked lists and the exiblearrays of ALGOL 68 belong to the latter class.). The use of a stack storage discipline is notrequired, but simply provides a convenient mechanism for reclaiming storage when a contouris no longer relevant. By storing the activation records on a heap, we broaden the possibilitiesfor specifying the lifetimes of objects. Storage for an activation record is analyze and understand all the provided review comments thoroughly. Now make the required amendments in your paper. If you are not confident about any review comment, then don't forget to get clarity about that comment. And in some cases there could be chances where your paper receives number of critical remarks. In that cases don't get disheartened and try to improvise the maximum.released only if the program fragment (block, procedure, class) to which it belongs has beenleft and no pointers to objects within this activation record exist.Heap allocation is particularly simple if all objects required during execution can ‘t intothe designated area at the same time. In most cases, however, this is not possible. Eitherthe area is not large enough or, in the case of virtual storage, the working set becomes toolarge. We shall only sketch threepossible recycling strategies for storage and indicate the support requirements placed uponthe compiler by these strategies.

Storage can be recycled automatically by a process known as garbage collection, whichoperates in two steps:

- Mark. All accessible objects on the heap are marked as being accessible.
- Collect. All heap storage is scanned. The storage for unmarked objects is recycled, and

all marks are erased.

This has the advantage that no access paths can exist to recycled storage, but it requiresconsiderable support from the compiler and leads to periodic pauses in program execution. Inorder to carry out the mark and collect steps, it must be possible for the run-time system to find all pointers into the heap from outside, find all heap pointers held within a given objecton the heap, mark an object without destroying information, and find all heap objects on alinear sweep through the heap. Only the questions of finding pointers affect the compiler;

there are three principal possibilities for doing this:

1. The locations of all pointers are known beforehand and coded into the marking algorithm.
2. Pointers are discovered by a dynamic type check. (In other words, by examining a

storage location we can discover whether or not it contains a pointer.)

3. The compiler creates a template for each activation record and for the type of every

object that can appear on the heap. Pointer locations and (if necessary) the object

length can be determined from the template.

Pointers in the stack can also be indicated by linking them together into a chain, but thiswould certainly take too much storage on the heap

.

D. *Error Handling*

Error Handling is concerned with failures due to many causes: errors in the compiler or itsenvironment (hardware, operating system), design errors in the program being compiled, anincomplete understanding of the source language, transcription errors, incorrect data, etc.The tasks of the error handling process are to detect each error, report it to the user, andpossibly make some repair to allow processing to continue. It cannot generally determinethe cause of the error, but can only diagnose the visible symptoms. Similarly, any repaircannot be considered a correction (in the sense that it carries out the user's intent); it merelyneutralizes the symptom so that processing may continue.

The purpose of error handling is to aid the programmer by highlighting inconsistencies.It has a low frequency in comparison with other compiler tasks, and hence the time requiredto complete it is largely irrelevant, but it cannot be regarded as an 'add-on' feature of acompiler. Its inuence upon the overall design is pervasive, and it is a necessary debugging
tool during construction of the compiler itself. Proper design and implementation of an errorhandler, however, depends strongly upon complete understanding of the compilation process.This is why we have deferred consideration of error handling until now

Errors, Symptoms, Anomalies and Limitations
We distinguish between the actual error and its symptoms. Like a physician, the error handlersees only symptoms. From these symptoms, it may attempt to diagnose the underlyingerror. The diagnosis always involves some uncertainty, so we may choose simply to report thesymptoms with no further attempt at diagnosis. Thus the word 'error' is often used when'symptom' would be more appropriate.A simple example of the symptom/error distinction is the use of an undeclared identifierin LAX. The use is only a symptom, and could have arisen in several ways:

- The identifier was misspelled on this use.
- The declaration was misspelled or omitted.
- The syntactic structure has been corrupted, causing this use to fall outside of the scopeof the declaration.

Most compilers simply report the symptom and let the user perform the diagnosis.An error is detectable if and only if it results in a symptom that violates the definition ofthe language. This means that the error handling procedure is dependent upon the language definition, but independent of the particular source program being analyzed. For example,the spelling errors in an identifier will be detectable in LAX (provided that they do not result in another declared identifier) but not in FORTRAN, which will simply treat the misspellingas a new implicit declaration.
We shall use the term anomaly to denote something that appears suspicious, but that wecannot be certain is an error. Anomalies cannot be derived mechanically from the language definition, but require some exercise of judgement on the part of the implementor. As experienceis gained with users of a particular language, one can spot frequently-occurring errors and report them as anomalies before their symptoms arise.

III. CONCLUSION

This report outlines a course in compiler construction. The implementation and source language is Scheme, and the target language is assembly code. This choice of languages allows a direct-style,stack-based compiler to be implemented by an undergraduate in one semester that touches onmore aspects of compilation than a student is likely to see in a compiler course for more traditionall Languages. Furthermore, expressiveness is barely sacrificed; the compiler can be bootstrappedprovided there is enough run-timesupport. Besides covering basic compilation issues, the course yields an implemented compiler that can serve as a testbed for coursework language implementation. The compiler has beenused, for example, tostudy advanced topics such as the implementation of first-class continuations and register allocation.

ACKNOWLEDGMENT

No individual effort, big or small can ever be realized without collective effort of proverbial Friends or guide. We are highly obliged and thankful to almighty who has provided us with theOpportunity to thank all the kins who stood by us in the process of working on this project. we would like to thank Mr.Abhishek Srivastava our honorable faculty memberand project guide who was of utmost help till we have proceeded. His valuableguidance is unmatched.

REFERENCES

[1] William M. WaiteDepartment of Electrical EngineeringUniversity of ColoradoBoulder, Colorado 80309USAemail: William.Waite@colorado.edu.
[2] GerhardGoosInstitutProgrammstrukturen und DatenorganisationFakultat fur Informatik
[3] Universit□at KarlsruheD-76128 KarlsruheGermanyemail: ggoos@ipd.info.uni-karlsruhe.de
[4] Niklaus WirthThis is a slightly revised version of the book published by Addison-Wesley in 1996ISBN 0-201-40353-6Zürich, November 2005.
[5] Aho, Alfred V., Hop croft, J. E., and Ullman, Jeffrey D. [1974]. The Design andAnalysis of Computer Algorithms.Addision Wesley, Reading, MA.
[6] Aho, Alfred V. and Johnson, Stephen C. [1976]. Optimal code generation for expression trees. Journal of the ACM, 23(3):488501.
[7] Aho, Alfred V. and Ullman, Jeffrey D. [1972]. The Theory of Parsing, Translation,
[8] and Compiling. Prentice-Hall, Englewood Cliffs.
[9] Aho, Alfred V. and Ullman, Jeffrey D. [1977]. Principles of Compiler Design.Addision
[10] Wesley, Reading, MA.
[11] Ross, D. T. [1967]. The AED free storage package. Communications of the ACM, 10(8):481492.
[12] Rutishauser, H. [1952]. Automatische Rechenplanfertigung bei Programm-gesteuerten
[13] Rechenmaschinen. Mitteilungen aus dem Institut f□ur Angewandte Mathematik der ETHZurich, 3.
[14] Sale, Arthur H. J. [1971]. The classi_cation of FORTRAN statements. Computer Journal,14:1012.
[15] Sale, Arthur H. J. [1977]. Comments on 'report on the programming language Euclid'.ACM SIGPLAN Notices, 12(4):10.
[16] Sale, Arthur H. J. [1979]. A note on scope, one-pass compilers, and Pascal. Pascal News,15:6263.
[17] Salomaa, Arto [1973]. Formal Languages. Academic Press, New York.
[18] Samelson, K. and Bauer, Friedrich L. [1960]. Sequential formula translation. Communicationsof the ACM, 3(2):7683.
[19] Satterthwaite, E. [1972]. Debugging tools for high level languages. Software{Practice andExperience, 2:197217.
[20] Scarborough, R. G. and Kolsky, H. G. [1980]. Improved optimization of FORTRAN
[21] object programs. IBM Journal of Research and Development, 24(6):660676.
[22] Schulz, Waldean A. [1976]. Semantic Analysis and Target Language Synthesis in a Translator.Ph.D. thesis, University of Colorado, Boulder, CO.
[23] Seegm□uller, G. [1963]. Some remarks on the computer as a source language machine.
[24] In Popplewell, C.M., editor, Information processing 1962, pages 524{525. North-Holland,Amsterdam, NL.

AUTHORS

First Author – Aastha Singh,Btech final year,Institute of Technology & Management ,Gida Gorakhpur (UP),E-mail: Aasthasingh158@gmail.com

Second Author –Sonam Sinha,Btech final year,Institute of Technology & Management ,Gida Gorakhpur (UP),E-mail: Sonamsinha28@gmail.com

Third Author –Archana Priyadarshi final year,Institute of Technology & Management ,Gida Gorakhpur (UP),E-mail: Archanapriya258@gmail.com