

## COMPILER CONSTRUCTION

Tapodhan Singla<sup>1\*</sup>, Varun Vashishtha<sup>2</sup>, Sumeet Singh<sup>3</sup>

<sup>\*1,2,3</sup>Computer Science and Engineering Department, Maharishi Dayanand University, Rohtak, Haryana, India

**\*Corresponding Author: -**

---

### **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. It is intended both 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. We introduce a set of tools especially designed or improved for compiler construction educative projects in C.*

**Keywords: -** Lex, Yacc , Parser, Parser-Lexer, Symptoms & Anomalies.



## 1. INTRODUCTION

A good course in compiler construction is hard to design. The main problem is time. Many courses assume C or some similarly low-level language as both the source and implementation language. This assumption leads in one of two directions. Either a rich source language is defined and the compiler is not completed, or the source and target languages are drastically simplified in order to finish the compiler. Neither solution is particularly satisfying. If the compiler is not completed, the course cannot be considered a success: some topics are left untaught, and the students are left unsatisfied. If the compiler is completed with an oversimplified source language, the compiler is unrealistic on theoretical grounds since the semantics of the language are weak, and if the compiler generates code for a simplified target language, the compiler is unrealistic on practical grounds since the emitted code does not run on real hardware.

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 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.

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:

defined upon them.

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

### What are Compilers?

The compiler accepts a subset of legal Scheme programs as defined in the Revised report, a subset strong enough to compile itself.

- \_ The language is syntactically restricted so that the only numbers accepted are integers in bounded range,
- \_ All lambda expressions have a fixed arity, i.e., no rest arguments.
- \_ Programs cannot have free variables other than references to primitives in operator position,
- \_ Symbols cannot be interned at runtime,
- \_ Rest-class continuations and I/O are not supported,
- \_ Derived syntax is not directly supported,
- \_ Garbage-collection is not provided, and the runtime library is minimal.

### 2. Lex and yacc

#### A. Availability:

Lex and yacc were both developed at Bell Laboratories in the 1970s. Yacc was the first of the two, developed by Stephen C. Johnson. Lex was designed by Mike Lesk and Eric Schmidt to work with yacc. Both lex and yacc have been standard UNIX utilities since 7th Edition UNIX. System V and older versions of BSD use the original AT&T versions, while the latest version of BSD uses flex and Berkeley yacc. The articles written by the developers remain the primary source of information on lex and yacc.

During the first phase the compiler reads the input and converts strings in the source to tokens. With regular expressions we can specify patterns to lex so it can generate code that will allow it to scan and match strings in the input. Each pattern specified in the input to lex has an associated action. Typically an action returns a token that represents the matched string for subsequent use by the parser. Initially we will simply print the matched string rather than return a token value.

The following represents a simple pattern, composed of a regular expression, that scans for identifiers. Lex will read this pattern and produce C code for a lexical analyzer that scans for identifiers. `letter(letter|digit)*`

This pattern matches a string of characters that begins with a single letter followed by zero or more letters or digits. This example nicely illustrates operations allowed in regular expressions:

- repetition, expressed by the "\*" operator
- alternation, expressed by the "|" operator
- concatenation

Any regular expression expressions may be expressed as a finite state automaton (FSA). We can represent an FSA using states, and transitions between states. There is one start state and one or more final or accepting states.

**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 already done may be more than adequate; others need to recognize specific sequences of tokens and perform appropriate actions. Traditionally, a description of such a set of actions is known as a **grammar**.

When you use a lex scanner and a yacc parser together, the parser is the higher level routine. It calls the lexer **yylex()** whenever it needs a token from 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 **yylex()**. Not all tokens are of interest to the parser—in most programming languages the parser doesn't want to hear about comments and white space, for example. For these ignored tokens, the lexer doesn't return so that it can continue on to the next token without bothering the parser. The lexer and the parser have to agree what the token codes are. We solve this problem by letting yacc define the token codes. The tokens in our grammar are the parts of speech: **NOUN**, **PRONOUN**, **VERB**, **ADVERB**, **ADJECTIVE**, **PREPOSITION**, and **CONJUNCTION**. Yacc defines each of these as a small 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 PREPOSITION 262
# define CONJUNCTION 263
```

Token code zero is always returned for the logical end of the input. Yacc doesn'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 newlexer.

*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.h" /* 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 */
I
lverb ( state = VERB; 1
^adj { state = ADJECTIVE; 1
"adv { state = ADVERB; 1
"noun { state = NOUN; 1
Prep { state = PREPOSITION; 1
pron { state = PRONOUN; 3
"conj { state = CONJUNCTION; 1
[a-zA-Z]+ {
if (state != LOOKUP) {
add-word(state, yytext);
} else I
switch (lookUpPword (yytext)) {
case VERB:
return (VERB);
case ADJECTIVE:
return (ADJECTIVE);
case ADVERB:
return (ADVERB);
case NOUN:
return (NOUN);
case PREPOSITION:
return (PREPOSITION);
case PRONOUN:
return (PRONOUN);
case CONJUNCTION:
return (CONJUNCTION);
default :
printf("%s: don't recognize\n", 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 of speech names used in the lexer to agree with the token names in the parser. We have also added return statements to pass to the parser the token codes for the words that it recognizes. There aren't any return statements for the tokens that define new words to the lexer, since the parser doesn'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 grammar to use for recognizing english sentences.
 */
#include <stdio.h>
% 1
%token NOUN PRONOUN VERB ADJECTIVE
ADJECTIVE PREPOSITION CONJUNCTION
```

```

% %
sentence: subject VERB object (printf("Sentence is
valid.\n\n);)
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\n", s) ;
}

```

The structure of a yacc parser is, not by accident, similar to that of a lexer. Our first section, the definition section, has a literal code block, enclosed in "%{" and "%}". We use it here for a C comment (as with lex, C comments 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 rule with 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. The action part of a rule consists of a C block, beginning with "{" and ending with "}". 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 successfully parsed a sentence. Since sentence is the top-level symbol, the entire input must match a sentence. The parser returns to its caller, in this case the main program, when the lexer reports the end of the input. Subsequent calls to yyparse() reset the state and begin processing again. Our example prints a message if it sees a "subject VERB object" list of input tokens. What happens if it sees "subject subject" or some other invalid list of tokens? The parser calls yyerror, which we provide in the user subroutines section, and then recognizes the special rule error. You can provide error recovery code 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 we shall 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 the simplicity of access functions to individual objects. Source language properties govern the possible 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 extents of 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 all objects at the time the program is translated (here we assume that translation includes binding), 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 outermost contour of an ALGOL 60 or Pascal program. (In contrast, arrays with dynamic bounds can occur 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 we have additional information that certain procedures are not recursive, for example because recursivity must be noted specially (as in PL/1) or because we have determined it from analysis of the procedure calls. We can then allocate storage statically for contours other than the outermost. Static storage allocation is particularly valuable on computers that allow access to any location in main memory via an absolute address in the instruction. Here, static storage corresponds exactly 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 out during binding, then we must also access statically allocated objects via a base register. The only advantage of static allocation then consists of the fact that no operations for storage reservation 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 and SIMULA 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 of objects from the dynamically enclosing contour. Thus we can use a stack discipline to manage these objects: Upon procedure call or block entry, the activation record containing storage for the local objects of the procedure or block is pushed onto the stack. At block end, procedure return or a jump out of these constructs the activation record is popped of the stack. (The entire 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 syntactic construct 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 within the activation record.  $R$  must be known at compile time but  $b$  cannot be known (otherwise we 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 used as an indirect address, or  $R$  appears as the constant in a direct access function of the form 'register+constant'. The extension, which may vary in size from activation to activation, is often called the second order storage of the activation record. Storage within the extension is always accessed indirectly via information held in the static part; in fact, the static part of an object may consist solely of a pointer to the dynamic part.

#### **D. Error Handling:**

Error handling is concerned with failures due to many causes: errors in the compiler or its environment (hardware, operating system), design errors in the program being compiled, an incomplete 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, and possibly make some repair to allow processing to continue. It cannot generally determine the cause of the error, but can only diagnose the visible symptoms. Similarly, any repair cannot be considered a correction (in the sense that it carries out the user's intent); it merely neutralizes 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 required to complete it is largely irrelevant, but it cannot be regarded as an 'add-on' feature of a compiler. Its influence 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 error handler, however, depends strongly upon complete understanding of the compilation process. This is why we have deferred consideration of error handling until now. It is perhaps useful to make a distinction between the correctness of a system and its reliability. The former property is derived from certain assumptions regarding both the primitives upon which the system is based and the inputs that drive it. For example, program verification techniques might be used to prove that a certain compiler will produce correct object programs for all source programs obeying the rules of the source language. This would not be a useful property, however, if the compiler collapsed whenever some illegal source program was presented to it. Thus we are more interested in the reliability of the compiler: its ability to produce useful results under the weakest possible assumptions about the quality of the environment, input data and human operator. Proper error handling techniques contribute to the reliability of a system by providing it with a means for dealing with violations of some assumptions on which its design was based.

#### **Errors, Symptoms, Anomalies and Limitations**

We distinguish between the actual error and its symptoms. Like a physician, the error handler sees only symptoms. From these symptoms, it may attempt to diagnose the underlying error. The diagnosis always involves some uncertainty, so we may choose simply to report the symptoms 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 identifier in LAX. The use is only a symptom, and could have arisen in several ways:

- The identifier as misspelled on this use.
- The identifier was misspelled or omitted.
- The syntactic structure has been corrupted, causing this use to fall outside of the scope of 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 of the 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 misspellings as a new implicit declaration. We shall use the term anomaly to denote something that appears suspicious, but that we cannot 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 experience is gained with users of a particular language, one can spot frequently occurring errors and report them as anomalies before their symptoms arise.

#### **7. 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 on more aspects of compilation than a student is likely to see in a compiler course for more traditional Languages. Furthermore, expressiveness is barely sacrificed; the compiler can be bootstrapped provided there is enough run-time support. Besides covering basic compilation issues, the course yields an implemented compiler that can serve as a test bed for coursework language implementation. The compiler has been used, for example, to study advanced topics such as the implementation of first-class continuations and register allocation.

#### **8. REFERENCES**

- [1]. William M. Waite Department of Electrical Engineering University of Colorado Boulder, Colorado 80309 USA email: William.Waite@colorado.edu.
- [2]. Gerhard Goos Institut Programmstrukturen und Datenorganisation Fakultät für 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., Hopcroft, J. E., and Ullman, Jeffrey D. [1974]. The Design and Analysis of Computer Algorithms. Addison 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, and Compiling. Prentice-Hall, Englewood Cliffs.
- [8]. Aho, Alfred V. and Ullman, Jeffrey D. [1977]. Principles of Compiler Design. Addison [
- [9]. Wesley, Reading, MA. Ross, D. T. [1967]. The AED free storage package. Communications of the ACM, 10(8):481492.
- [10]. Rutishauser, H. [1952]. Automatische Rechenplanfertigung bei Programm-gesteuerten [13] Rechenmaschinen. Mitteilungen aus dem Institut für Angewandte Mathematik der ETHZurich, 3.
- [11]. Sale, Arthur H. J. [1971]. The classification of FORTRAN statements. Computer Journal, 14:1012.
- [12]. Sale, Arthur H. J. [1977]. Comments on 'report on the programming language Euclid'. ACM SIGPLAN Notices, 12(4):10.
- [13]. Sale, Arthur H. J. [1979]. A note on scope, one-pass compilers, and Pascal. Pascal News, 15:6263.
- [14]. Salomaa, Arto [1973]. Formal Languages. Academic Press, New York.
- [15]. Samelson, K. and Bauer, Friedrich L. [1960]. Sequential formula translation. Communications of the ACM, 3(2):7683.
- [16]. Satterthwaite, E. [1972]. Debugging tools for high level languages. Software {Practice and Experience, 2:197217.
- [17]. Scarborough, R. G. and Kolsky, H. G. [1980]. Improved optimization of FORTRAN [21] object programs. IBM Journal of Research and Development, 24(6):660676.
- [18]. Schulz, Waldemar A. [1976]. Semantic Analysis and Target Language Synthesis in a Translator. Ph.D. thesis, University of Colorado, Boulder, CO.
- [19]. Seegmüller, G. [1963]. Some remarks on the computer as a source language machine.
- [20]. In Popplewell, C.M., editor, Information processing 1962, pages 524{525. NorthHolland, Amsterdam, NL.