

DTIC FILE COPY

Final report

(4)

AD-A196 581

Contract N00014-86-K-0204

GENERATING FAST,
ERROR RECOVERING PARSERS

by

Robert W. Gray

B.S., University of Maryland, 1977



A thesis submitted to the
Faculty of the Graduate School of the
University of Colorado in partial fulfillment
of the requirements for the degree of
Master of Science
Department of Computer Science
1987

DISTRIBUTION STATEMENT A

Approved for public release;
Distribution Unlimited

88 5 1 162

This thesis for the Masters of Science degree by

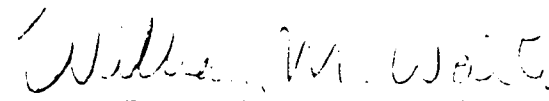
Robert William Gray

has been approved for the

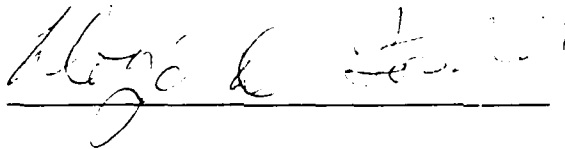
Department of

Computer Science

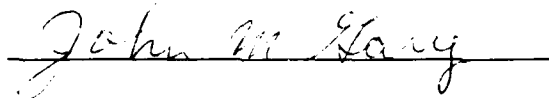
by



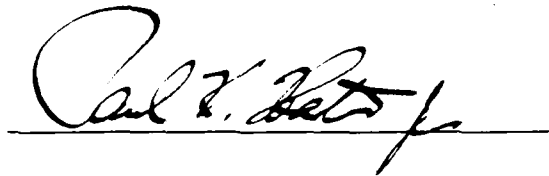
William M. Waite



Lloyd D. Fosdick



John M. Gary



Paul K. Harter

Date 13 April 1987

Gray, Robert William (M.S., Computer Science)

Generating Fast, Error Recovering Parsers

Thesis directed by Professor William M. Waite

Although parser generators have provided significant power for language recognition tasks, many of them are deficient in error recovery. Of the ones that do provide error recovery, many of these produce unacceptably slow parsers. I have designed ^{and} implemented a parser generator that produces fast, error recovering parsers. The high speed of the parser is a result of making the code directly executable, and paying careful attention to implementation details. The error recovery technique guarantees that a syntactically correct parse tree will be delivered after parsing has completed no matter what the input. This is important so that remaining compilation phases will not have to deal with infinitely many special cases of incorrect parse trees. Measurements show that the generated parser runs faster than any other parser examined, including hand-written recursive descent parsers. The cost of this fast parser is a slight increase in space requirements. Although this particular generator requires LL grammars, the ideas can be applied to generators taking LALR grammars. Furthermore, there is evidence that most LALR grammars for programming languages can be automatically converted to equivalent LL grammars.

Accession For	
NTIS GRA&I	☒
DTIC TAB	☐
Unannounced	☐
Justification	
By <i>per letter</i>	
Distribution/	
Availability Codes	
Dist	Avail and/or Special
<i>A-1</i>	

ACKNOWLEDGEMENTS

Professor William Waite has long known that fast parsing speed and reliable error recovery are crucial to practical compilers. His interest and curiosity in this area infected me. I thank him for patiently giving hours of his time, and providing an exciting, stimulating research environment.

Professor Lloyd Fosdick and John Gary graciously allowed me to pursue this thesis while I worked on their supercomputer compiler project.

Mark Hall challenged our Software Engineering Group to construct a generator of fast parsers.

Andreas Lemke helped me with grammar transformations using Ops83. Professor Jon Shultis introduced me to the ML language and additional ML assistance was provided by Hal Eden.

Thanks to Karen and Tori for their company and good food, usually late at night. I regularly counted on the help and support of Carmen, David, Dotty, Elisa, Evi, Francesca, Halfdan, Kathy, Paul, and especially my brothers, sisters and parents.

This work was supported in part by the National Bureau of Standards 70NANB5H0506, the Army Research Office DAAL 03-86-K-0100, and the Office of Naval Research N00014-86-K-0204.

CONTENTS

CHAPTER

1. INTRODUCTION	1
1.1. Background	1
1.2. Tools	5
1.3. A Directly Executable Parser with Error Recovery	9
1.4. Thesis Statement and Organization	9
2. DIRECTLY EXECUTABLE PARSER	10
2.1. Motivation	10
2.2. Recursive Descent Parsing	11
2.3. SYNPUT	13
2.4. Direct Execution	19
2. SYNTACTIC ERROR HANDLING	26
3.1. Overview	26
3.2. Automatic Error Recovery	28
3.3. Implementation	32
4. PERFORMANCE	37
4.1. Performance Details	37
4.2. Comparison	39

	vi
5. CONCLUSIONS	42
BIBLIOGRAPHY	45
APPENDIX	47

CHAPTER 1

INTRODUCTION

A parser is the part of a compiler that recognizes the structure of the input language. It should be time and space efficient. It should be *user friendly*, that is, present clear and concise diagnostics for syntax errors. A parser should also be maintainable; that is, a small change or fix should only require a small amount of effort. Without the aid of tools, it is difficult to build a correct parser that includes all of these attributes. This thesis describes the design and implementation of parser generation software.

Section 1.1 will establish the context for discussing parsing and define the terms used. Section 1.2 describes and comments on the strengths and weaknesses of a number of existing tools for generating parsers that do not meet all of the design criteria. Section 1.3 sketches the design of a parser that meets the criteria. Section 1.4 presents the overall organization of the thesis and the thesis statement.

1.1. Background

Before focusing on the particular problems of parsing, it is important to understand the context in which it occurs. As shown in Figure 1, a compiler is a program that reads a program in a *source* language and translates it into an equivalent program in a *target* language. An important part of this process is the

reporting of errors in the source program to the user. Figure 2 shows the decomposition of a compiler and the flow of data through its constituents. The first level of decomposition divides compilation into analysis and synthesis phases. To determine the meaning of the source program, analysis breaks up the source program into its constituent parts and creates an intermediate representation of the source program called *structure tree*. The name derives from

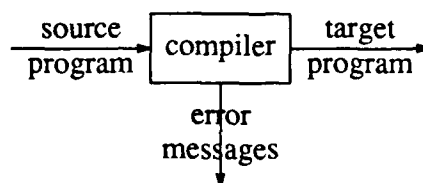


Figure 1. The Compiler

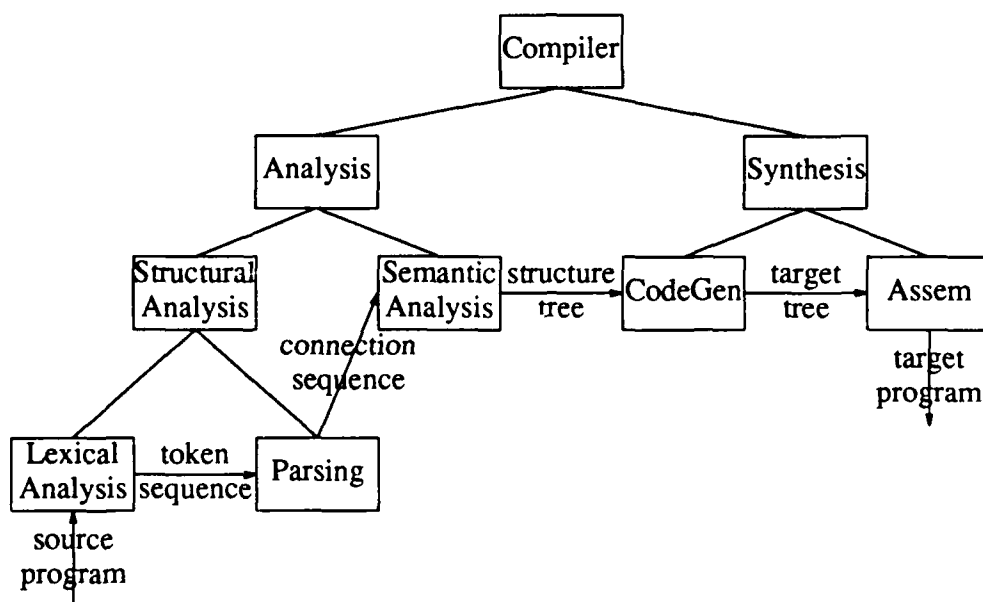


Figure 2. The Decomposition of a Compiler (arrows show data flow)

the fact that it is usually conceptualized as a tree whose structure represents the control and data flow of the program. The synthesis phase constructs the desired equivalent target program from the structure tree.

Analysis is the more formalized of the two major compiler tasks. It is generally broken down into two parts: *structural* analysis and *semantic* analysis. Structural analysis determines the static structure of the source program, and *semantic* analysis checks that substrings of the input are meaningful in their particular context. Structural analysis is further decomposed into two parts. *Lexical analysis* deals with the basic symbols of the source program and is described in terms of a finite-state automaton. Lexical analysis reads the *terminal symbols* of the source program and produces a sequence of *tokens*. *Parsing* deals with the static structure of the program and is described in terms of a pushdown automaton. It determines whether a sequence of tokens forms a structurally correct program. The program can be represented as an explicit *parse tree* or in a linearized form called the *connection sequence* which is input to semantic analysis.

There is little in the way of formal models for the entire synthesis process, although algorithms for various subtasks are known. Synthesis consists of two distinct subtasks: *code generation* and *assembly*. Code generation transforms the structure tree into an equivalent target tree. Assembly resolves all target addressing and converts the target machine instructions into an appropriate output format, namely the target program.

Central to the issues of parsing is a *grammar*, which specifies the structure of a language. For example, an if-else statement in C has the form:

if (expression) statement else statement

The statement is the concatenation of the keyword *if*, a left parenthesis, an

expression, a right parenthesis, a statement, the keyword **else**, and another statement. Using the variable *expr* to denote an expression, and the variable *stmt* to denote a statement, the structuring rule can be expressed as

$$\text{stmt} \rightarrow \text{if (expr) stmt else stmt}$$

in which the arrow can be read as "can have the form". Such a rule is called a *production*. In a production, lexical elements such as identifiers, the keyword **if**, and "(" are the terminal symbols. Variables such as *expr* and *stmt* represent sequences of tokens and are called *non-terminals*.

Grammars can be placed in several classes, of which only *context-free* is of interest here. A context-free grammar has four components:

- (1) A set of *terminal* symbols.
- (2) A set of *non-terminals*.
- (3) A set of productions where each production consists of a non-terminal called the *left-side* of the production, an arrow, and a sequence of tokens and/or non-terminals called the *right-side* of the production.
- (4) A designation of one of the non-terminals as the *start* symbol.

We assume that the production listed first contains the start symbol and that digits, special symbols such as "(", and boldface strings are terminals. For notational convenience productions with the same non-terminal on the left can have their right-sides grouped together with the alternative right-sides separated by the symbol "I" which can be read as "or."

Parsing methods can be classified according to the way the parse tree is constructed. Top-down parsers build parse trees from the root (start symbol) to the leaves (terminal symbols) while bottom-up parsers start from the leaves and work to the root. In both cases, the input to the parser is scanned from left to right, one token at a time. The two types of parsers are capable of recognizing different classes of languages defined by properties of the grammar. The

language recognized by a top-down parser can be specified by an LL grammar. An LALR grammar can be used to specify a bottom-up parser.

Discussion can be limited to the LL(1) and the LALR(1) grammars because parsers for their languages:

- (1) Are deterministic and hence parse in linear time,
- (2) Never accept a symbol that cannot continue a correct parse,
- (3) Cover almost all programming language constructs,
- (4) Can be analyzed by well-known tools, and
- (5) Have well-known algorithms for error recovery.

Parsers implemented by hand, which work with LL(1) grammars, are usually directly executable code. Generated parsers are usually table driven and are specified with LL(1) grammars or the larger set, LALR(1) grammars.

1.2. Tools

A *parser generator* is the tool that takes a grammar as an input, and produces a parser (see Figure 3).

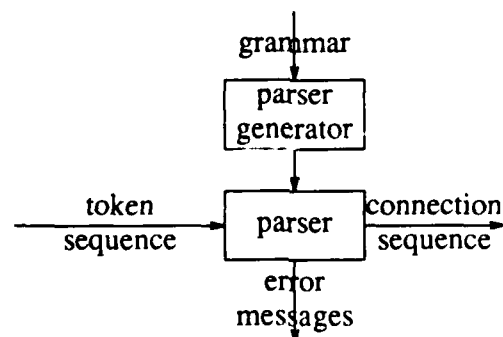


Figure 3. Parser generator

Using a parser generator is preferred over "hand coding" because the resulting parser is more reliable, maintainable and often faster. Further, it can be produced with less effort. By reliable, I mean that the parser produces useful results under the weakest possible assumptions about the quality of the input. A reliable parser will accept all legal programs and reject any illegal ones. Building such a parser "by hand" is tedious and error prone.

Even without error recovery, building and maintaining a parser by hand is an error prone process. A hand written parser is usually written from a grammar specification. The most common technique, recursive descent, requires an LL grammar. Just determining that the grammar is LL is tedious. If the parser does not work properly, it will need to be changed, but the grammar specification may or may not be correct. The real definition of the language that the parser accepts becomes embodied in the code of the parser routine. The correspondence between the grammar and the parser is not guaranteed. On the other hand, a generated parser is constructed directly from the grammar specification. Assuming the tool works properly, the parser *must* perform according to the grammar. Changes in parsing behavior are always a result of changing the grammar. The tool ensures that the generated parser will always correspond exactly to the grammar.

Almost by definition, error handling involves a mass of special cases and exceptions to rules. Therefore, it is very difficult to attempt to handle all errors with ad-hoc techniques. It is difficult to prove, or even convince someone, that a hand written parser is correct and can deal robustly with illegal input. If the source program contains structural errors, the parser must indicate the problem. It is usually unacceptable for the parser to "give up" at the first error and abort the compilation process. We desire parsers that will "recover" from an error by

altering the remaining input and continuing. This has to be carried out in a very careful and systematic way to insure that the entire compiler remains in a consistent state. Fortunately, there is a theoretic foundation that provides an error recovery technique that is guaranteed to handle all syntactic errors. Automating this method ensures that the error recovery will always work properly.

Parser generators gained popularity in the early 1970's [Aho1986]. YACC [Johnson1979] for example has been used to help implement hundreds of compilers. Initially, just being able to construct a parser from a grammar was sufficient motivation to use a parser generator. As languages become more powerful and complex, we need to place more emphasis on parsing speed and automatic error recovery. Each of the parser or parser generators in the following list have different strengths and weaknesses, but none of them have all of the desired attributes.

YACC is a parser generator well known in the UNIX community. Many UNIX tools such as *pic*, *make*, *config*, *awk* and *cc* use YACC to construct their parsers. YACC produces a medium speed, table driven parser (C code). For most applications the parsing speed is not a major bottleneck in the overall program. The space requirements of a YACC parser are very reasonable. YACC produces a parser from an LALR(1) grammar. This is an advantage over parser generators that only work from LL(1) grammars. It is an inconvenience that YACC does not accept extended BNF notation (EBNF) [Waite1983]. YACC has directives that can be used to specify precedence and associativity. The major drawback of this tool is that there is no automatic error recovery. Unless the compiler writer is willing to do extra work, the first syntax error will terminate parsing.

The PGS [Dencker1985] parser generator also uses an LALR(1) grammar to generate a table driven parser. Unfortunately, the parser runs unacceptably slowly. PGS spends a lot of effort compacting the parse tables. This combined with inflexibility of PASCAL (the implementation language of the generated parser) results in extremely costly parser table access. The major advantage of a PGS generated parser is that it is guaranteed to recover from syntactic errors. PGS accepts an extended BNF notation, but has no directives for precedence or associativity.

The SYNPUT generator [Dunn1981] produces parsing tables from an LL(1) grammar. SYNPUT accepts extended BNF and has the same automatic error recovery as PGS. The generated parser runs at about the same speed as a YACC parser.

Performance is always an issue for production compilers. Measurements show that parsing time often represents a significant percentage of compilation time and furthermore that hand-written parsers run faster than parsers generated from tools [Gray1985b]. This is unnecessary and these tools are deficient. Production compilers often use directly-executable recursive descent parsers. Many of the tool-generated parsers are table driven thus leading to reduced speed.

If hand written parsers were always much faster than generated parsers, there would be trouble justifying using the latter. Although none of the above parser generators offers both speed and automatic error recovery (AER), as compiler writers, we desire such a tool. The produced parser should be reasonably space efficient. It is desirable for the tool to accept the LALR(1) class of grammars.

1.3. A Directly Executable Parser with Error Recovery

Waite and Hall [Waite1985b] have devised a method of generating a top-down parser directly from an LL(1) grammar. Their directly executable parser runs even faster than a recursive descent parser because they have eliminated unnecessary procedure call overhead; nevertheless, we still need automatic error recovery.

This thesis describes the design and implementation of a tool that generates a Directly Executable parser with Error Recovery (DEER). The error recovery has a negligible effect on the parsing speed of syntactically correct programs. When a syntactic error is present in the input program, the parser invokes an error recovery automaton which deletes and/or generates tokens to get the parse back on track. Normal parsing then continues.

1.4. Thesis Statement and Organization

This thesis is concerned with generating fast, error recovering parsers. An earlier project [Gray1985b] investigated why tool generated compilers [Kastens1982] ran so much slower than hand written ones and found that parsing time accounted for the major difference. Not all generated parsers are slow [Gray1985a]. With the exception of this work, other generated parsers do not provide both high speed and error recovery. I present the design and implementation of a tool that produces fast, error correcting parsers. The generated parser is efficient, user-friendly, and maintainable.

Chapter 2 explains the generation of directly executable parsers. Chapter 3 shows how automatic error recovery has been incorporated into the fast parser. Chapter 4 presents time and space measurements of the DEER, YACC and PGS generated parsers. The last chapter discusses areas of further study.

CHAPTER 2

DIRECTLY EXECUTABLE PARSER

This chapter explains the concepts of the Waite and Hall parser and the refinements I have made to it. The conventional recursive descent parser is explained first. Then I draw a parallel between the recursive descent parser and SYNPUT tables. These tables can be interpreted or they can be transformed into directly executable code.

2.1. Motivation

There are numerous cases where the efficiency of a tool is crucial to its acceptance. The PGS tool has the error recovery feature that we desire, but is often not used because of the slow running speed of the generated parser. As both Waite [1985a] and Pennello [1986] hypothesized, high parsing speed can be achieved when the parser is directly executable. Hall and Waite [Waite1985b] performed an experiment comparing a YACC generated parser for Pascal to their directly executable parser for the same language. As Figure 4 shows, at a cost of twice the total space the speedup is almost a factor of ten. Although the respective grammars are somewhat different, it is clear that direct execution offers the opportunity to achieve large speed improvements.

parser	time	text	data	bss	total space
YACC	631	664	3916	608	5188
Waite/Hall	65	9660	436	0	10096

Figure 4. Waite-Hall parser vs YACC

2.2. Recursive Descent Parsing

A recursive descent parser has a procedure for each left-hand non-terminal of the grammar. Each procedure recognizes the respective right-hand side. The technique requires an LL(1) grammar. Briefly, for LL(1) grammars, it is always possible to determine which production will be required based only on the current terminal symbol. A complete discussion of LL(1) grammars and recursive descent parsing can be found in the literature [Waite1983, Aho1986]. The LL(1) condition can be verified by hand, or by using a tool such as SYNPUT. Recursive descent parsing has been used for years in production compilers [McClure1972]. It is the best general method of writing parsers by hand. Even when coded in a high level language, the recursive descent parser is very efficient.

Figure 5 gives a simple LL(1) grammar that will be used in examples throughout this chapter.

type	→	simple
		^ id
		array [simple] of type
simple	→	int
		char
		num .. num

Figure 5. Sample grammar

Figure 6 gives some example sentences of the language specified by the sample grammar.

```

num .. num
int
array [ int ] of ^id
char

```

Figure 6. Sample input sentences

Aho, Sethi and Ullman [Aho1986] give a recursive descent parser for this grammar which is condensed in Figure 7. *Match* is the procedure that consumes the expected token and gets the next token from the lexical analyzer. This is known as the *lookahead* (*la*) symbol (or token) because it has not yet been accepted by the parser, but it is available for inspection. Parsing begins by setting *la* to the first token of an input, and then calling the procedure corresponding to the *start* of the grammar (e.g. "type" in this grammar). Figure 8 traces the recursive descent execution on the input sentence: **array [int] of ^ id**.

```

procedure match(t:token)
  if la = t then la := nextterm( ); else error( );

procedure type
  if la is in {int char num} then simple( )
  elseif la = ^ then match(^); match(id);
  elseif la = array then match(array); match([);
    simple( ); match(]); match(of); type( );
  else error( );

procedure simple
  if la = int then match(int);
  elseif la = char then match(char);
  elseif la = num then match(num); match(..); match(num);
  else error( );

```

Figure 7. Recursive descent parser

Procedure	Code	Lookahead token
		array
type	if la is in {int char num} elseif la = ^ elseif la = array then match(array);	
match		[
match		int
simple	if la = int then match(int)	
match		]
match		of
match		^
type	if la is in {int char num} elseif la = ^ then match(^);	
match		id
match		EOF

Figure 8. Recursive descent trace

2.3. SYNPUT

SYNPUT is a tool that produces a top-down table driven (ie. not directly executable) parser from an LL(1) grammar. The parsing tables are divided into six parts.

- (1) A summary of the number of terminal symbols, director sets, actions, and parsing rules.
- (2) A bit matrix representing the director sets (explained below).
- (3) The terminal symbols of the grammar with their assigned token codes.
- (4) The terminal symbols that have intrinsic attributes — such as identifiers and numbers.
- (5) The non-terminals.
- (6) The parsing rules.

Given these tables, the interpreter part of the parser will recognize sentences of the language. When the interpreter needs the next terminal symbol, it calls the lexical analyzer which returns a token code. SYNPUT encodes the terminal symbols of the sample grammar as given in Figure 9.

1	^
2	id
3	array
4	[
5	]
6	of
7	int
8	char
9	num
10	..

Figure 9. Token codes for the terminal symbols

Briefly, the *director sets* are sets of terminal symbols that enable the parser to determine quickly what action to take next. Figure 10 gives the director sets for the grammar in both symbolic and integer form.

D1 =	{int char num} =	{7 8 9}
D2 =	{^ array int char num} =	{1 3 7 8 9}

Figure 10. Director sets

Notice in Figure 7 that the first line of procedure *type* is checking whether the lookahead is a member of the director set D1. Furthermore, if the lookahead is not a member of director set D2, then calling *type* will cause a call to the procedure *error*. This is how these sets are used to direct parsing action. In procedure *simple*, the first test can be considered a test in a director set with a single element (ie. a test for symbol *int*). It is more efficient to avoid explicit singleton director sets.

The SYNPUT produced parsing rules are given in Figure 11 (symbolic codes have replaced integer codes to make the table easier to follow).

A portion of the interpreter for SYNPUT tables is given in Figure 12. Parsing operation begins with the lookahead symbol, *la*, initialized to the first terminal of the program, and with the stack, *Stk*, containing a single element zero.

Rule	Op	S	L/A
L0	JE	{^ array int char num}	L1
L1	NC		L14
L2	NR		
L3	JO	int	L6
L4	RD	int	
L5	JP		L13
L6	JO	char	L9
L7	RD	char	
L8	JP		L13
L9	JO	num	L4
L10	RD	num	
L11	RE	..	
L12	RE	num	
L13	NR		
L14	JO	{int char num}	L17
L15	NC		L3
L16	JP		L28
L17	JO	^	L21
L18	RD	^	
L19	RE	id	
L20	JP		L28
L21	JO	array	L15
L22	RD	array	
L23	RE	[	
L24	NE	{int char num}	L3
L25	RE	]	
L26	RE	of	
L27	NE	{^ array int char num}	L14
L28	NR		

Figure 11. SYNPUT produced parsing tables

The first table entry is a jump (JE) to the code that corresponds to the start symbol of the grammar. Parsing is complete when the entire input is consumed and the stack is empty. Every state transition requires a cycle through the loop. The current parse state is maintained on the top of the stack. The PUSH macro pushes a new state onto the top of the stack. The JMP macro changes the top of the stack to the target, thus affecting a jump. In the interpreter, the three arrays Op, Dset and Datum correspond to the columns of the table for opcodes, symbols or

director sets and labels (rules) respectively. The opcode definitions are given in Figure 13. *S* refers to a set of terminal symbols (possibly a singleton set). Later, it will be more convenient to express singleton sets by suffixing the opcode with the letter "S". For example, "JO" is equivalent to "JOS". *L* refers to a rule (also a state of the parser), and *A* always refers to an action to be performed upon recognition of a grammar production. (There are no actions in the sample grammar.) Each opcode requires specific operand types, so operand interpretation is unambiguous.

Consider the trace given in Figure 14 for the parsing the following sentence:

Interpretation begins at rule 0 of Figure 11. The lookahead symbol has been initialized to the first token, that is, $la = 3$. la is a member of D_2 so goto L_1 .

```

do {
rule =
switch
    Stk[Sp]++;
    (Op[rule])
    RDS RD    { /* case labels */
               la=lexAttr( ); break;

    RE        IF_NOT(Dset[rule],la) (void)parErr(la,rule);
               else la=lexAttr( ); break;

    AC        action(Datum[rule]); break;
    RA        IF_NOT(Dset[rule],la) (void)parErr(la,rule);
               else { action(Datum[rule]);la=lexAttr(); } break;

    NC        PUSH break;

    JP        JMP; break; /* set stack to Datum[rule] */

    JI        IF_IN(Dset[rule],la) JMP; break;

    ...
    }
}
while(Sp>0);

```

Figure 12. Table interpreter

AC	A	Perform semantic action A.
AR	S,A	Perform action A and read a new symbol.
EO	S	Error if la not in S.
JE	S,L	If la in S jump to L, else error.
JI	S,L	If la in S jump to L.
JO	S,L	If la not in S jump to L
JP	L	Unconditionally jump to L.
NC	L	Call nonterminal L.
NE	S,L	If la in S call nonterminal L, else error.
NR		Return
RD	S	Read new symbol.
RE	S	If la in S read a new one, else error.
RA	S,A	If la in S perform A and read, else error.

Figure 13. Definition of SYNPUT codes

rule L	Op[rule]	Dset[rule] S	Datum[rule] L/A	Stack	lookahead
L0	JE	{D2}	L1	0 1	array
L1	NC		L14	0 2	
L14	JO	{D1}	L17	0 2 15	push(14)
L17	JOS	^	L21	0 2 18	
L21	JOS	array	L15	0 2 22	
L22	RDS	array		0 2 23	
L23	RES	[		0 2 24	[
L24	NE	{D1}	L3	0 2 25	int
L3	JOS	int	L6	0 2 25 4	push
L4	RDS	int		0 2 25 5	
L5	JP		L13	0 2 25 6	]
L13	NR			0 2 25 14	
L25	RES	]		0 2 26	
L26	RES	of		0 2 27	of
L27	NE	{D2}	L14	0 2 28	^
L14	JO	{D1}	L17	0 2 28 15	push(14)
L17	JOS	^	L21	0 2 28 18	
L18	RDS	^		0 2 28 19	
L19	RES	id		0 2 28 20	id
L20	JP		L28	0 2 28 21	EOF
L28	NR			0 2 28 29	
L28	NR			0 2 29	
L2	NR			0 3	

Figure 14. Interpretive parser trace of `array [ int ] of ^id`

Now make a subroutine call (L14) to process the non-terminal *simple*. *la* is not in D1 so goto L17. *la* is not equal to 1 so goto L21. *la* is equal to 3 so continue at L22. Read the next symbol, 4. *la* is 4 so read next symbol: *la* = 7. The process is continued until the stack is empty.

2.4. Direct Execution

SYNPUT has performed the work of producing a set of parsing tables that tell how to recognize sentences of a language. These tables can be translated into directly executable code. Figure 15 shows the steps to generate a DEER parser from an LL(1) grammar. A set of programs transform the SYNPUT tables into files of C code. *Direc.h* and *Direc.c* encode the director sets. There are three files for the arrays Op, Dset and Datum. Finally the directly executable code is in the file *tbl.i*. There are also a set of files that remain constant for all grammars. These, along with the six generated C files, can be compiled together yielding a complete parser.

The structure of the directly executable parser shown in Figure 16 is similar to a an assembly language version of a recursive descent parser. A call is

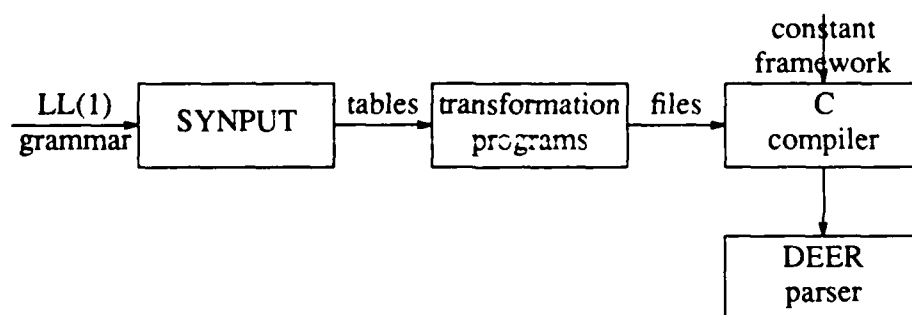


Figure 15. Steps to build a directly executable parser

implemented as the sequence *PU(); goto L;* and a return is *goto pppop*. Notice that the case labels are compact. This yields an improvement in execution speed of up to 25%.

```

goto L0;
pppop:      switch (--*DEPSp) {
case 0:      break;

              /* Code included from tbl.i */
              L0:  IF_NOT(D2,la) la=parErr(la,L0);
                  goto L1;
              L1:  PU(1); goto L14;
case 1:      L2:  goto pppop;
              L3:  if (la != int) goto L6;
              L4:  la = nexterm( );
              L5:  goto L13;
              ...
              ...
case 3:      L25:  if (la != ]) la=parErr(la,L25);
                  la = nexterm( );
              L26:  if (la != of) la=parErr(la,L26);
                  la = nexterm( );
              L27:  IF_NOT(D2,la) la=parErr(la,L27);
                  PU(4); goto L14;
case 4:      L28:  goto pppop;
              }

```

Figure 16. Directly Executable Parser

At first glance, the directly executable parser appears to have a structure similar to the interpretive parser. However, the interpretive parser requires a loop iteration for every state change: the directly executable parser makes transitions with goto statements. A switch statement is required for only a fraction of the cases. There are roughly four times as many instructions required for a state transition for the interpretive case. The trace in Figure 17 illustrates the parsing for the sentence

array [int] of ^id.

I use a two step translation process to convert SYNPUT tables into C code. The SYNPUT tables distinguish between multiple element sets and singleton sets by using a star (*). It is more efficient to test for a particular symbol than to test whether a set contains the symbol. As shown in the example in Figure 18, in step 1, any instruction that is dealing with a single terminal symbol (starred integers in Figure 11), is translated into a new instruction, whose name has an *S* suffix. Figure 19 gives a sample of the transform pattern.

PU is a macro that pushes its argument onto the parser stack if there is room. This is part of the mechanism used to simulate procedure calls. For example, *NC* (the other procedure call instructions are *NE* and *NES*) is a call to a non-terminal production of the grammar. First *NC* remembers the location to which control should return (*PU*). A goto is executed and the return position is marked with a case label. For efficiency, the case labels should be compact, thus *current* is an integer from 1 to the number of calls in the parse tables. *L* is a label attached to the code for the entry corresponding to the jump target; *IF_IN* and *IF_NOT* are macros that test if *la* is a member (not a member) of director set *S*.

A portion of Figure 7 has been transformed into directly executable code in Figure 20.

Rule	Code	Lexical array
L0:	IF_NOT(D2,la) la=parErr(la,L0); goto L1;	
L1:	PU(1); goto L14;	push(1) 0 1
L14:	IF_NOT(D1,la) goto L17;	
L17:	if (la != ^) goto L21;	
L21:	if (la != array) goto L15;	
L22:	la = nexterm();	[
L23:	if (la != [) la=parErr(la,L23); la = nexterm();	int
L24:	IF_NOT(D1,la) la=parErr(la,L24); PU(3); goto L3;	push(3) 0 1 3
L3:	if (la != int) goto L6;	
L4:	la = nexterm();	]
L5:	goto L13;	
L13:	goto pppop;	pppop: 0 1 3
case 3:		
L25:	if (la !=]) la=parErr(la,L25); la = nexterm();	of
L26:	if (la != of) la=parErr(la,L26); la = nexterm();	^
L27:	IF_NOT(D2,la) la=parErr(la,L27); PU(4); goto L14;	push(4) 0 1 4
L14:	IF_NOT(D1,la) goto L17;	
L17:	if (la != ^) goto L21;	
L18:	la = nexterm();	id
L19:	if (la != id) la=parErr(la,L19); la = nexterm();	EOF
L20:	goto L28;	
case 4:		
L28:	goto pppop;	pppop: 0 1 4
case 4:		
L28:	goto pppop;	pppop: 0 1
case 1:		
L2:	goto pppop;	pppop: 0

Figure 17. DEER parser trace of array [int] of 'id'

	JO	7*,6
step 1	JOS	7,6
step 2	if(la !=7) goto L6	

Figure 18. Translation example

RD		la = nexterm();
AC	A	action(A);
NC	L	PU(current); goto L; case current:
NR		goto pppop;
JP	L	goto L;
JI	S,L	IF_IN(S, la) goto L;
JO	S,L	IF_NOT(S, la) goto L;
EO	S	IF_NOT(S, la) la=parErr(la,current);
JIS	S,L	if (la == S) goto L;
JOS	S,L	if (la != S) goto L;
NE	S,L	IF_NOT(S,la) la=parErr(la,L);
		PU(current); goto L; case current:

Figure 19. SYNPUT codes to C code

Rule	Op	S	L/A	Directly Executable Code
L0	JE	D2	L1	IF_NOT(D2,la) la=parErr(la,L0); goto L1;
L1	NC		L14	PU(1); goto L14; case 1:
L2	NR			goto pppop;
L3	JO	int	L6	if(la != int) goto L6;
L4	RD	int		la = nexterm();
L5	JP		L13	goto L13;
...	...	...	...	...
L25	RE	]		if(la !=]) la=parErr(la,L25); la = nexterm();
L26	RE	of		if(la != of) la=parErr(la,L26); la = nexterm();
...	...	...	...	...

Figure 20. Directly Executable Code

The most straight-forward data structure used to represent director sets is a two dimensional array, indexed by director set number and token code. For our grammar, it might look like Figure 21.

set #	token codes									
	1	2	3	4	5	6	7	8	9	10
D1	0	0	0	0	0	0	1	1	1	0
D2	1	0	1	0	0	0	1	1	1	0

Figure 21. Naive director set encoding

Since most machines are byte or word addressable, we waste lots of space when representing director sets. Packing the bits has the disadvantage of requiring a much longer access time. A set membership test would require first a two dimensional array access, and then a bit test, depending on the value of the terminal symbol.

The key insight for better access efficiency is that for each rule the director set S is a constant for a particular grammar and thus fixed at generation time; only la varies with different input programs. Access will be faster if the director sets are stored *vertically*, with each byte holding bits from 8 sets and successive bits of a set occupying successive bytes. When there are more than 8 sets, another block of bytes will represent the next 8 sets. The length of the blocks matches the number of terminal symbols (ie. 11 for the sample grammar). Figure 22 shows that D1 contains a bit representing symbols 7, 8 and 9 which correspond to the director set $D1 = \{\text{int char num}\}$.

sym	D7	D6	D5	D4	D3	D2	D1	D0
0								
1						1		
2								
3						1		
4								
5								
6								
7						1	1	
8						1	1	
9						1	1	
10								

Figure 22. Director set access and storage

The macros (Figure 23) generate an index into a one dimensional array, and generate the right bit mask for the test.

```

#define WD 8
#define NS 11 /* ten symbols plus the zeroth entry */
#define IF_NOT(set,sym) if(!(Dir[sym+(set/WD)*NS]&(1<<(set%WD))))
#define IF_IN(set,sym) if( (Dir[sym+(set/WD)*NS]&(1<<(set%WD))))

char Dir[] = {
0x00,      /* sym=0 sets 7-0 */
0x04,      /* sym=1 sets 7-0 */
0x00,      /* sym=2 sets 7-0 */
0x04,      /* sym=3 sets 7-0 */
0x00,      /* sym=4 sets 7-0 */
0x00,      /* sym=5 sets 7-0 */
0x00,      /* sym=6 sets 7-0 */
0x06,      /* sym=7 sets 7-0 */
0x06,      /* sym=8 sets 7-0 */
0x06,      /* sym=9 sets 7-0 */
0x00};     /* sym=10 sets 7-0 */

```

Figure 23. Director set storage

CHAPTER 3

SYNTACTIC ERROR HANDLING

The quality of its syntactic error messages is an indication of the user friendliness of a compiler. In the best case, the user is immediately led to all syntactic errors of his program. In the worst case, if error recovery operates incorrectly, the compiler may crash, leaving the user helpless. Section 3.1 gives a brief overview of syntactic error recovery techniques. Section 3.2 presents the theory behind the error recovery method selected. Implementation details are presented in Section 3.3.

3.1. Overview

As a parser operates, it consumes input, token by token. The consumed input, also called *accepted* input, drives the parser into a particular state. Deterministic (non-backtracking) parsers never accept a token that cannot legally continue what has already been accepted. This is one of the principle merits of LL and LALR parsing techniques — they are guaranteed to detect errant tokens as soon as they are encountered.

The user should receive as much information as possible from each compilation attempt. It is unacceptable just to detect the first error and quit. The parser should repair errors and continue parsing. Finally, it should deliver a valid

parse tree or connection sequence to the rest of the compiler. If the parser tries to recover but delivers a faulty parse tree, the remaining phases of the compiler could crash and leave the user helpless.

Gries [1976] gives an excellent annotated bibliography for error handling. Also, Horning [1976] presents an overview of various techniques of error handling. There are many strategies a parser can employ for error recovery: panic mode, phrase level, error productions, global, and automatic.

One of the simplest language independent recovery techniques is the *panic mode*. When an error is detected the input is skipped until one of a predefined set of "special" symbols such as *begin* or ";" is encountered. The parsing stack is popped until the special symbol can be accepted. Unfortunately, this method has many shortcomings. It frequently results in deleting large portions of the source text. In addition, semantic information depending on the erased part of the stack becomes inconsistent. Finally, the set of special symbols must be determined by hand.

On detection of an error, *phrase-level* recovery makes a backward move in the parse stack and a forward move in the remaining input. This isolates a phrase which is likely to contain the error. Then a weighted minimum distance correction is carried out at the phrase level.

Joy, Graham and Haley [Graham1982b] use *error productions* for their production Pascal compiler. First the compiler writer needs to predict the most likely kinds of errors expected. Then error productions must be written by hand. It is impossible to foresee all error conditions. In practice, the parser must be exercised to see how well the error recovery works. Further tuning is likely to be needed. There is another shortcoming of this method: the added error productions could make the grammar ambiguous.

Global error recovery attempts to find the smallest set of changes that will make a given program syntactically correct. It is impractical from the perspective of efficiency due to the exponential number of corrections that must be considered. The smallest set of changes if there are more than two errors is to enclose the error portion with comment brackets. This is usually not a desirable recovery.

Röhrich [1980] has implemented *automatic* construction of error handling parsers for LALR(1) grammars, and Fischer [1980] has done it for LL(1) grammars. The technique is based on a sound theoretic foundation. The resulting parsers are capable of correcting all syntax error by insertion and/or deletion of tokens to the right of the error location. Therefore, no backtracking is needed, and the output of the parser always corresponds to a syntactically valid program. This contributes significantly to the reliability and robustness of a compiler. The speed of parsing correct parts of a program is not affected by the presence of the error handling capability.

Röhrich's technique of automatic error recovery was chosen for incorporation in the directly executable parser because it automatically derives the error recovery directly from the grammar. Many other techniques require the manual specification of error recovery.

3.2. Automatic Error Recovery

A parser for language L will *accept* input strings (i.e. programs) in L . Let T be the set of terminal symbols of the language L ; then $T^* - L$ is the set of all erroneous programs. Let $\omega t \chi$ be an erroneous program, where ω is an initial string that is syntactically correct and has been accepted by the parser and symbol t cannot be accepted by the parser. The rest of the program is the string χ . We say that t is a *parser-defined* error.

If $\omega t \chi \in (T^* - L)$ is an erroneous program with parser-defined error t , then to effect recovery the parser must alter either ω or $t \chi$ such that $\omega' t \chi \in L$ or $\omega t' \chi' \in L$. Alteration of ω is undesirable since it may involve undoing the effects of previous actions. It is too expensive to retain information in case backtracking is needed. Thus, we consider the alteration of only t and χ .

The basic idea is as follows. A fixed terminal symbol $f(q)$ is associated with each state q of the parser. When an error is detected, the parse stack is copied. Then a "continuation parse" is carried out using the copied stack and $f(q_i)$ as input at each state q_i . In addition the set of allowable terminals (Director set) of each state q_i is added to the *anchor* set which is the set of all terminals that could be accepted during this continuation parse. The function f is chosen such that this process would terminate the parse rapidly, driving the parser through states $q_1, \dots, q_n$. Next zero or more of the actual input symbols are discarded until an input symbol t'' is found which is in the anchor set. The state for which t'' is acceptable is q_i . Then, the error is corrected by inserting $f(q_1) \dots f(q_{i-1})$ into the input stream to the left of t'' while adjusting the original stack. Finally, normal parsing is resumed.

More formally, the steps are:

- Associating: Associate a terminal symbol $f(q)$ with each parse state q at generation time.
- Detecting: Detect the unacceptable token t .
- Anchors: Determine a *continuation*, μ , such that $\omega \mu \in L$. In other words, find a string that legally completes the accepted input ω . Construct a set of *anchors* $D = \{d \in T \mid v \text{ is a head of } \mu \text{ and } \omega v d \text{ is a head of some string in } L\}$.
- Discarding: Discard from the input the shortest string $\eta \in T^*$ such that $t\chi = \eta t''\mu', t'' \in D$.
- Generating: Generate the shortest string $\sigma \in T^*$ such that $\omega \sigma t''$ is a head of some string in L . Note, σ is one particular v of step 3.
- Resuming: Resume the normal parse with input $t''\mu'$.

The following two examples illustrate error recovery for the cases of only token deletion and only token generation. The other possible case is token deletion and generation.

3.3. Implementation

This section first describes the straight-forward task of implementing error recovery in a table driven parser. Next we address the problems encountered in incorporating error recovery in a directly executable parser and describe their solutions which form the contributions of this work. The Appendix contains the source code for both types of parsers. Between the two parsers there are many routines that are identical and a number that are similar. The C preprocessor's conditional directives handle the few differences in the latter case. The Makefile can be directed to produce either a table driven parser ("make itr") or a directly executable parser ("make dep"). When a routine name is mentioned, it can be found in a C file of the same name in the Appendix.

Chapter 2 explained the workings of the table driven parser on correct input programs. This section explains parser behavior on incorrect programs. Figure 26 is a block diagram showing the interactions of the various parts of the table-driven parser during error recovery. The labels on the arcs are used in the text to refer to the corresponding invocations during processing.

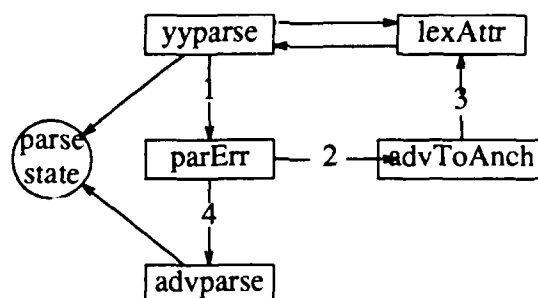


Figure 26. Error Recovery in a Table Driven Parser

During normal operation, the parser requests tokens from lexAttr. When an error

is detected, `yyparse` calls `parErr` (1) which conducts the entire error recovery. By the time `parErr` returns, the error has been corrected and `yyparse` continues normally with the next token.

In handling error correction, `parErr` calculates the anchor set (D) using the technique described in the previous section and calls `advToAnch` and `advparse` to delete and insert the appropriate strings. After computing D , `parErr` calls `advToAnch` (2) which discards tokens from the input by calling `lexAttr` (3) until an input token (t'') is found which is in D . Finally `parErr` calls `advparse` (4) to continue the parse generating tokens until reaching a state in which t'' can be accepted. For this last step, `advparse` uses the actual parse stack and modifies the parser state. When `advparse` reaches a state where t'' is acceptable, error recovery is complete, `parErr` returns and normal parsing continues. Both `yyparse` and `advparse` are automata that share the same state, but differ only in their input (actual input for `yyparse`, generated input for `advparse`). For either automaton, parsing and semantic actions will be identical for identical input. The source code with annotations is given in the Appendix.

Whereas the state in the table driven parser is entirely contained in the stack, in a DEER parser the state is represented by both the program counter and the stack. If error recovery can change the parse state, then this change needs to be reflected back in the directly executable parser. This could be accomplished by `parErr` "jumping" to the appropriate place in the parser which would be difficult and machine dependent. Instead, the approach taken is to have error recovery drive the directly executable parser by manipulating the tokens it receives. Normally, the scanner gives tokens directly to the parser through calls to `lexAttr`. However, under error recovery, the parser may get generated tokens (ie. from the string σ if it is non-empty). Control over this is exercised by

interposing a routine between the parser and *lexAttr*, which returns either input tokens (from *lexAttr*) or generated tokens (from *advparse*) depending on the error correcting state of the system.

There are only a few differences between the error recovery routines of the DEER parser and the table driven parser.

<i>nexterm</i> ()	Obtains a token from <i>lexAttr</i> or <i>advparse</i> depending on the parser state.
<i>parErr</i> (<i>la</i> , <i>rule</i>)	has the additional task of building a call stack in a form that is expected by error recovery. This requires a mapping between compact case labels and rules (states).
<i>advparse</i> ()	The directly executable version performs no actions. Instead the actions are performed in the actual parser as <i>advparse</i> provides generated tokens.

Figure 27 shows the control flow of the DEER parser. A dashed arrow represents a function call, and a solid arrow represents a token returned by a function. The parser operates in either normal mode or error recovering mode. This state is indicated by the global boolean variable [*ErrRecovering*] (set by *parErr*, cleared

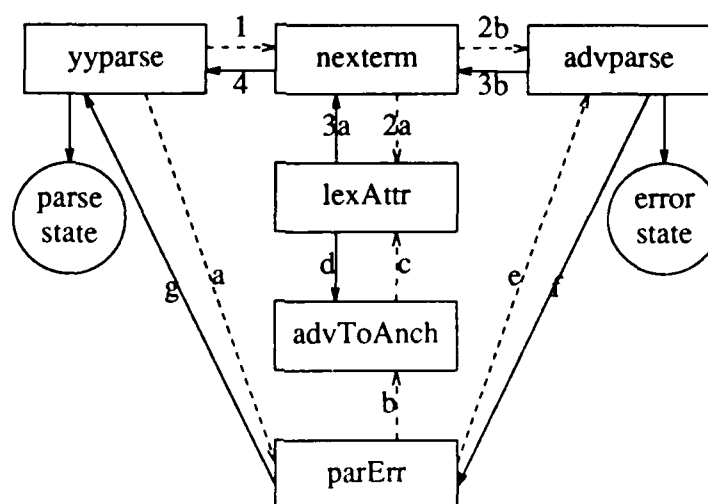


Figure 27. Logical structure of the error recovering parser

by *advparse*, and examined by *nexterm*). During normal parsing, *yyparse* requests a token from *nexterm* (1). *nexterm* sees that [ErrRecovering] is false so it returns the value obtained from *lexAttr* (2a,3a,4). The cost of testing [ErrRecovering] which should be insignificant compared to the cost of obtaining a token from the lexical analyzer, is the only extra cost when parsing a correct program.

The following conditions are required for correct behavior. For every call to *parErr* that occurs, there must be at least one call to *advparse*, the first call is made by *parErr* itself, while possible additional calls will be made by *nexterm* (when *ErrRecovering* = true).

The last call to *advparse*(from *parErr* or *nexterm*) must never generate a synthetic token but must return t'' . This is the exact condition under which *ErrRecovering* gets reset to false. *advparse* will be called exactly $\text{length}(\sigma) + 1$ times. Furthermore, if $\sigma = \emptyset$, then $\eta \neq \emptyset$. (Note that the first and last call of *advparse* could be the same call). There are two cases for the first call to *advparse* from *parErr*. 1) The token t'' matches what *advparse* was expecting so *ErrRecovering* is reset to false and t'' is returned. This happens when tokens are only discarded from input. 2) t'' does not match what *advparse* was expecting so a token must be generated.

The token returned by *parErr* must be one such that the condition which caused the call to *parErr* is corrected - for example

if($la \neq 5$) $la = \text{parErr}(\dots)$ or

IF_IN(2, la) $la = \text{parErr}(\dots)$

parErr must return a token la such that the condition is false; that is, *parErr* returns 5 or something in director set 2 respectively.

Actions will occur at the correct place because only the parser can invoke actions.

CHAPTER 4

PERFORMANCE

The time and space requirements of a parser can vary widely. Seemingly small details can make huge differences. In this chapter, I bring out some of the performance issues and then compare DEER, SYNPUT and YACC generated parsers recognizing PASCAL.

4.1. Performance Details

Conventional wisdom for software tuning is to build a system, measure it, and then work on the areas which can yield the largest payoffs. I have used this approach. The design and implementation of DEER has been heavily biased toward fast parsing. Often, clever data structures reduce both time and space requirements; however, when there has been a conflict, I have chosen to trade off some extra space for higher speed.

I will first discuss instruction space efficiency. Figure 28 gives the static frequency distribution of code to parse PASCAL. There are total of 730 such instructions.

174	la = nexterm();
108	IF_NOT() la=parErr();
92	if (!=) goto L;
88	if (!=) la=parErr();
85	goto L;
82	PU(); goto L
36	goto pppop;
...	...

Figure 28. Frequency distribution of code

In terms of instruction space efficiency, the macro IF_NOT occurs very frequently and should therefore be as compact as possible. One of the original implementations expanded this macro into about 5 machine instructions. The current version, (Figure 22), expands the test into one instruction. This saves about 1000 bytes for the pascal language (2 bytes per instruction * 5 instructions per test * 108 tests).

The director set representation is crucial to data space efficiency. There are roughly 64 director sets and 64 symbols for the PASCAL grammar. The naive implementation would require about 4096 bytes. Bit packing reduces this to 512 bytes.

Time efficiency is facilitated by the vertical director set storage technique. The macro IF_NOT(la,2) checks membership of the lookahead in director set 2. It is expanded as

if(! (Dir[la + 2/8 * 11] & (1<< 2%8)))

The single 68020 instruction required to carry out this test is:

btst #2,a5@(0,d7:w)

The lookahead is in register d7, and the base of the director sets is in register a5. The WIDTH of a byte is 8 and there are 11 terminal symbols (The extra is the zeroth array entry). Various less optimal implementations require a runtime

computation of the array index and the bit offset before the test can be made. This triples the time required. On a 68020, byte tests are significantly cheaper than long tests. Also, placing the array base in a register saves a load instruction for every director set test. Taken together, these details represent more than an order of magnitude speed improvement for director set testing.

The static frequency distribution of code gives us no clue as to how often these statements are executed. It turns out that director set membership is heavily used. For the input program described in the next section, the IF_NOT macro is used 37,267 times and the PU macro is used 19,590 times.

A naive representation of Op, Dset, and Datum would use integer arrays requiring a total of 8760 bytes ($730 \text{ rules} * 3 \text{ arrays} * 4 \text{ bytes per integer}$). There are only 24 different kinds of *Operations* (Figure 12 gives a partial list); therefore, a byte is sufficient to represent them. Symbols and director sets could exceed 255 and rule numbers certainly do (730 rules for pascal) so a short integer is used. More than 4000 bytes are saved by this representation ($730 + 1460 + 1460$).

There are a number of other examples where careful tuning can yield substantial time and efficiency payoffs. These include choosing registers for heavily used objects, such as the lookahead symbol. The stack pointer is also places in a register.

4.2. Comparison

This section compares the time and space requirements of the parsers. All measurements were carried out on a SUN 3/75 running SUN UNIX 3.2. The parsers, which are written in C, were compiled with the optimize flag (-O). The call graph execution profiler *gprof* [Graham1982a] and *time* provided the speed

measurements. The input used was the distributed SYNPUT pascal program. The file which is 105,813 bytes long, consists of 16,170 tokens. The breakdown of this input program is given in Figure 29.

5638	Name
1869	;
965	:=
834	,
740	) (
400-523	Int : ^ .
200-368	END BEGIN String THEN IF] [=
70-182	+ NIL DO <> ELSE VAR
40-61	- PROCEDURE WITH WHILE
...	...

Figure 29. Lexical classification of input

The *size* command provided the space information for *text*, (the executable code), *data*, (the initialized data), and *bss* (the uninitialized data, zero fill on demand).

Both SYNPUT and DEER parsers have automatic error recovery; the YACC parser has no error recovery. Figure 30 compares overall time and space requirements of the parsers for the input. (The link editor rounds up sizes to the next 2k byte boundary).

Parser	Time	Space			
		text	data	bss	total
DEER	2.0	40960	24576	6148	71684
SYNPUT	3.9	32768	24576	5968	63312
YACC	3.2	24576	24576	5884	55036

Figure 30. Time and space requirements

In Figure 31, the gprof tool extracts the parsing time from the overall time.

DEER	0.32
SYNPUT	2.20
YACC	1.24

Figure 31. Parsing time reported by gprof

DEER parses about four times faster than YACC. This speed advantage plus error recovery costs about 25% more space (71K :: 55K).

CHAPTER 5

CONCLUSIONS

I have built a generator that produces a very fast parser with error recovery. Based only on the grammar, the parser is guaranteed to recover from any syntactic input error and will output a correct structure tree. In terms of efficiency, user friendliness and maintainability, the generated parser contributes significantly to the quality of software containing it. These design goals have been met with only a very modest space cost over parsers that have no error recovery.

The DEER parser generator requires an LL(1) grammar. Many existing grammars are LALR(1) and compiler writers have come to rely on the power of the LALR technique. Often, simple mechanical transformations can yield an LL(1) grammar from an LALR(1) grammar [Griffiths1976]; however, there are clearly cases where this is neither possible nor desirable. There seem to be two areas inviting further research — first, the need for a quality LALR(1) parser generator. Second, there is a need for programs to assist in the grammar transformation process.

I believe there is need for a new LALR(1) parser generator. It could be based on an existing generator such as PGS or YACC. Corbett's [1985] Bison, which is similar to YACC, holds promise as a starting base because of its

efficiency and clarity. Furthermore, it is in the public domain. The major question is whether it would be easier to add error recovery to YACC or Bison, or make PGS faster. Pennello's [1986] work on making LALR parsing very fast should be considered before such a project is undertaken.

Automating the grammar transformation process is desirable because tedious, error prone work is eliminated. A parser generated from the transformed grammar may be able to run significantly faster than a parser generated from the LALR(1) grammar.

There are several problems with transforming grammars. The most important problem is that, LL(1) languages are a proper subset of LALR(1) languages, thus there must exist LALR(1) grammars that cannot be transformed. Furthermore, it is undecidable whether an arbitrary LALR(1) grammar has an equivalent LL(1) grammar; therefore, no program can tell us if the transformation can be accomplished.

The transformed grammar may not be transparent to the compiler writer. It would be much better to work only with the original grammar and insist that the transforming tool preserve actions. This is easily accomplished by planting connection points in the grammar before transforming.

Early work in the area was performed by Foster [Foster1968] who built a program call SID. Its main purpose was to assist language designers in producing an equivalent grammar which could be parsed by a simple one-track parsing algorithm. The parser also had reliable error detection capability whereas hand written parsers had no assurances. SID first removes left recursion. Next a series of substitutions and factorings is performed in hopes of producing an LL grammar. If this step succeeds, a final pass is made for parsing efficiency.

Based on some of Foster's ideas, I implemented some code to perform the same three transformations: left recursion removal, left factoring and corner substitution. The programs work on small simple test cases. Due to bugs in our ML compiler, the programs cannot transform larger grammars such as PASCAL and ADA. It will be interesting to learn whether complex grammars such as ADA can successfully be transformed.

BIBLIOGRAPHY

- [Aho1986] Aho, A. V., R. Sethi and J. D. Ullman, *Compilers Principles, Techniques, and Tools*, Addison Wesley, 1986.
- [Corbett1985] Corbett, R. P., "Static Semantics and Compiler Error Recovery", UCB/Computer Science Dpt. 85/251, Berkeley, June 1985.
- [Dencker1985] Dencker, P., *User Description of the Parser Generating System PGS*, Institut fur Informatik Universitat karlsruhe, 1985.
- [Dunn1981] Dunn, D. and W. M. Waite, SYNPUT A Tool for Processing Programming Language Syntax, Feb. 1981.
- [Fischer1980] Fischer, C. N., D. R. Milton and S. B. Quiring, "Efficient LL(1) Error Correction and Recovery Using Only Insertions", *Acta Informatica 13* (1980), University of Wisconsin-Madison.
- [Foster1968] Foster, J. M., "A Syntax Improving Device", *Computer Journal*, May 1968.
- [Graham1982a] Graham, S. L., P. B. Kessler and M. K. McKusick, "gprof: a Call Graph Execution Profiler", *UNIX PROGRAMMER'S MANUAL 4.2BSD, Vol 2C*, 1982.
- [Graham1982b] Graham, S. L., C. B. Haley and W. N. Joy, "Practical LR Error Recovery", *SIGPLAN Notices*, 1982.
- [Gray1985a] Gray, R. W., *A Brief Comparison of the Time and Space Requirements of two LALR(1) Parser Generators- PGS and YACC*, University of Colorado, December 1985.
- [Gray1985b] Gray, R. W., "Comparing Semantic Analysis Efficiency of a GAG Generated Compiler vs Hand Written Compilers", ECE690 Report, Department of Electrical and Computer Engineering, University of Colorado, Boulder, CO, Dec. 1985.
- [Gries1976] Gries, D., "ERROR RECOVERY and CORRECTION - An Introduction to the Literature", *Compiler Construction - An Advanced Course Edited 2nd Ed*, 1976.
- [Griffiths1976] Griffiths, M., "LL(1) Grammars and Analysers", *Compiler Construction - An Advanced Course Edited 2nd Ed*, 1976.
- [Horning1976] Horning, J. J., "What the Compiler Should Tell the User", *Compiler Construction - An Advanced Course Edited 2nd Ed*, 1976.
- [Johnson1979] Johnson, S. C., "YACC- Yet Another Compiler Compiler", *UNIX PROGRAMMER'S MANUAL Seventh Edition, Vol 2B*, Jan 1979.
- [Kastens1982] Kastens, U., B. Hutt and E. Zimmermann, *GAG: A Practical Compiler Generator*, Springer Verlag, 1982.

- [McClure1972] McClure, R. M., "An Appraisal of Compiler Technology", in *Spring Joint Computer Conf.*, vol. 40 , AFIPS Press, Montvale, NJ, 1972.
- [Pennello1986] Pennello, "Very Fast LR Parsing", *SIGPLAN Notices* 21, 7 (July 1986).
- [Rohrich1980] Rohrich, J., "Methods for the Automatic Construction of Error Correcting Parsers", *Acta Informatica*, 1980.
- [Waite1983] Waite, W. M. and G. Goos, *Compiler Construction*, Springer-Verlag, 1983.
- [Waite1985a] Waite, W. M., "The Cost of a Generated Parser", *Software—Practice & Experience* 15, 3 (March 1985), 221-237.
- [Waite1985b] Waite, W. M. and M. Hall, Private Discussions, 1985.

APPENDIX

This annotation pertains to the source code which follows. In this description, program variables will appear in square brackets. The first step of the formal description is to associate a terminal symbol with each parser state. This is a shortest valid continuation of the string being parsed can be obtained by advancing the parser to an accepting state. The continuation is defined by the sequence of terminal symbols specified by AR, RD, RE, and RA instructions encountered during the advance. To advance the parser to the end of parsing, increment the state at each AC, AR, EO, RD, RE, or RA instruction. The state should be set to L at a JE, JO, or JP instruction. At NC or NE, the state is incremented and the new value pushed onto the parse stack. At NR, the state is set to the value of the top element of the parse stack and this element is popped off.

When an error is detected, *parErr* is called with two arguments: the errant token t [la] and, the parsing rule where the error was detected [rule]. *parErr* controls the recovery by arranging for the Anchor, Discarding and Generating steps of the formal description to be carried out. First a call is made to *getanch* to compute the anchor set D . Next η is deleted from the input by calling *advToAnch* with the argument t [la]. It returns the first acceptable token t'' which is saved in [AnchLa]. Finally, *advparse* is called to generate the string of terminal symbols σ .

getanch is the automaton that carries out step 3 of the recovery. It initializes its state by making a private copy [dstack] of the global parse stack [Stk]. *getanch* then simulates parsing by following the shortest continuation μ

and simultaneously building the anchor set D [Anchors]. There are two steps for each state transition cycle. First, look at the current state q [rule], and if the automaton is in a state that requires looking at a director set, then add this set to the anchor set. Second, perform a state transition.

advToAnch carries out step 4. It receives the errant token t [errLa] and returns the first acceptable token t'' [la]. As a side effect, zero or more calls have been made to the lexical analyzer interface *lexAttr*. Note, if zero calls were made to *lexAttr*, then it must have been the case that $t = t''$.

advparse carries out step 5. It is an automaton very similar to the regular parser automaton (*parseLTR*). It operates on the actual parse stack [Stk] with the following two differences. At each state transition cycle, a check is made to determine if the anchor token t'' [AnchLa] can be accepted in the current state. If so, the automaton is terminated and the token t'' is returned. The second difference is that when a token needs to be read, one is instead generated from the continuation $(\mu), f(q)$.

When *yyparse* receives an unacceptable token, it calls *parErr* (a) to initiate the error recovery automaton. *parErr* receives a copy of the errant token t [la] and the place where the error occurred [rule]. *parErr* sets *ErrRecovering* to true, constructs a stack for the automaton *advparse* to simulate the state of the parser, computes the anchor set D [Anchors] by calling *getanch*, advances the input to the first token of the anchor set, by calling *advToAnch* (b), saves this token in t'' [AnchLa] and, finally, returns an acceptable token to *yyparse* by calling *advparse* (e,f,g).

While in error recovery mode, *yyparse* calls *nexterm* (1), who calls *advparse* (2b). Either a token is generated, or the global t'' [AnchLa] is returned (3b,4). The global variable [AnchLa] holds the first input token matching some

member of the anchor set. This must be saved because error recovery may need to generate synthetic tokens before t'' [AnchLa] can be accepted.

advToAnch, must possibly discard the string of tokens η from the input string by calling *lexAttr* (c,d). The automaton *advparse* must generate the shortest string σ up to the anchor t'' [AnchLa]. If σ is empty no tokens will be generated.

parErr should return t [la] if only discarding has occurred otherwise *parErr* should return the head of σ as obtained from *advparse*(.).

advparse.c

```

#include "err.h"
extern POSITION curpos;
#include "macros.h" /* PUSH POP ACTION ... */
short
choose(set)
short set;
/* Pick the first terminal of a of Director set, set.
** On entry-
**   set=Director set from which to choose
** On exit-
**   returns terminal code for the chosen terminal
*/
{
register DECTYPE *dirSet = virtDir;
short sym;

for(sym=1; sym<=NSYMS; sym++)
    IF_IN(set,sym)
        break;

If(sym > NSYMS)
    PANIC("choose: terminal not found");
DB(CHOOSE,("choose: returns token %s(%d) from set %d\n",tokCod(sym), sym,set));
return(sym);
}

void tokgen() { DB(TOKGEN,("tokgen\n"));}
char *malloc();

short
advparse()
/*
** Advance the parser to accept the anchor
** On entry-
**   Stk[1..Sp]=parser stack
**   Stk[Sp] indexes the rule detecting the error
** On exit-
**   Stk[1..Sp]=repaired parser stack
**   Stk[Sp] indexes the rule to be interpreted
**
** DEP:
** On entry-
**   AnchLa contains first token matching an anchor. Advparse found
**   it by discarding input up to anchor token.
** On exit-
**   if AnchLa is member of director set- ErrRecovering=0
**   return(AnchLa)
**   else loop until we tokgen (nT==1), then return it (genla).
*/
{
register DECTYPE *dirSet = virtDir;
static char emsg[] = "Generating token \"%s\"";
char *p;
short genla; /* look ahead generated */
short rule,i;
short nT=0; /* no newToken generated yet, used only by DEP */

DB(ADVPAR,("AdvPar: rule Op[rule] Dset[] Datum[] Stk befor switch\n"));
do {
    rule = Stk[Sp];

    DB(ADVPAR,("%d\\%s\\%d\\%d\\",
        rule, instrCod(Op[rule]), Dset[rule],Datum[rule]));
    DBFOR(i=0,i<=Sp,i++)
        DB(ADVPAR,(" %d",Stk[i]));

```

```

DB(ADVPAR,("\n"));

#include "opcase.i" /* shorthand for case labels below */
switch( Op[rule]) /* CHECK SYMBOL */
{
    AR JE JI JO RD RE RA

        IF_IN(Dset[rule],AnchLa) goto stopFound;
        break;
    ARS JES JIS JOS RDS RES RAS
        If(Dset[rule]==AnchLa) goto stopFound;
        break;
    AC EO JP NC NE NR NES NRS EOS
        break;
    default:
        PANIC("advparse: default case");
}

#ifdef DEP /* all actions are done co-routine in directexec parser */
#define ACTION
#else
#define ACTION action(Datum[rule])
#endif
#define JMP Stk[Sp]=Datum[rule]

    Stk[Sp]++; /* next rule */
    switch( Op[rule])
    {
        AC ACTION; break;
        RD RE genla=choose(Dset[rule]);tokgen();nT=1; break;
        AR RA genla=choose(Dset[rule]);tokgen();nT=1; ACTION; break;
        RDS RES genla = Dset[rule]; tokgen();nT=1; break;
        ARS RAS genla = Dset[rule]; tokgen();nT=1; ACTION; break;
        NES NC NE PUSH; break;
        NRS NR POP; break;
        JOS JES JP JO JE JMP; break;
        JIS EOS JI EO /*cont*/break;
        default:
            PANIC("advparse: default of parsop[rule]");
    }

#ifdef DEP
}while(Sp!=0 && nT==0);
If(Sp==0) printf("WARNING!! advparse: exit from while loop Sp==0\n");
DB(ADVPAR,("ADVPAR, genLa=%s(%d)\n",tokCod(genla),genla));
If( (p=malloc(sizeof(msg)+10)) == NULL) /* for token expansion !!!! */
    printf("parErr: malloc failed\n");
else
    {
        sprintf(p, msg, tokCod(genla));
        message(WARNING,p,0,&curpos);
    }
return(genla);
#else
}while(Sp!=0);
printf("WARNING!! advparse: Exit from while loopSP==0\n");
#endif

stopFound:
ErrRecovering = 0; /* can continue normal parse now */
DB(ADVPAR,("ADVPAR End Recovery,returns genLa=%s(%d)\n",tokCod(AnchLa),AnchLa));
return(AnchLa);
}

```

getanch.c

```

#include "macros.h"

void
getanch()
/* Compute the anchor set
** On entry-
**   Stk[1..Sp]=parser stack
**   Stk[Sp] indexes the rule detecting the error
** On exit-
**   Anchors=anchor set
*/
{
    register DECTYPE *dirSet = virtDir;
    short dsiktop; /* dummy parse stack */
    short dstack[MAXSTK];
    short rule,i;

    for(i=1; i<=NSYMS; i++) Anchors[i] = 0; /* zero Anchors */

    dsiktop = 0; /* copy valid portion of Stk */
    DB(GETANCH,("getanch:stack= "));
    while( dsiktop<Sp ) {
        dsiktop++;
        dstack[dsiktop]=Stk[dsiktop];
        DB(GETANCH,("%d ",Stk[dsiktop]));
    }
    DB(GETANCH,("\n"));

    do {
        rule = dstack[dsiktop]++; /* assume get ready for next instr */

#include "opcase.i"
        switch(Op[rule]) /* SEARCH - build Anchor Set for this production */
        {
            /* does JJ belong here??? */
            AR JE JI JO RD RE RA
                for(i=1; i<=NSYMS; i++) /* add elements of Dir set to Anchors */
                    IF_IN(Dset[rule], i)
                        Anchors[i] = 1;
                break;
            ARS JES JIS JOS RDS RES RAS
                Anchors[Dset[rule]] = 1; /* add this symbol to Anchors
                break;

            AC EO JP NC NE NR NES NRS EOS
                break;
            default:
                PANIC("getanch: default");
        }

        switch(Op[rule]) /* MOVE - shortest continuation */
        {
            JOS JES JP JO JE
                dstack[dsiktop] = Dnum[rule]; /* Jump */
                break;
            NES NC NE
                dsiktop++; dstack[dsiktop] = Dnum[rule]; /* Call
                break;
            NRS NR
                dsiktop--; /* Return */
                break;
            ARS EOS JIS RDS RES RAS RD RE AC AR RA JI EO

```

```

        break;          /* does JI belong here??? */
    default:
        PANIC("getanch: default case");
    }

    }while(dstktop!=0);

#ifdef DEBUG
DB(GETANCH,("anchor set: "));
    for(i=1; i<=NSYMS; i++)if(Anchors[i]) DB(GETANCH,("%s(%d) ",tokCod(i),i));
DB(GETANCH,("\n"));
#endif
}

```

advToAnch.c

```

#include "err.h"
extern POSITION curpos;
#include "macros.h"
short
advToAnch(errLa)
short errLa;
/*
** Advance the input text to an anchor
** On entry-
**     errLa=first unaccepted token
**     Anchors=set of possible anchors
** On exit-
**     return first member of Anchors encountered
*/
{
    short la = errLa;
    while(!Anchors[la])
    {
        message(WARNING, "Discarding token",0,&curpos);
        DB(ADVANCH, ("ADVANCH: discarding token %s(%d)\n",tokCod(la),la));
        la = lexAttr();
    }
    return(la);
}

```

parErr.c

```

#include "err.h"
extern POSITION curpos;
#include "macros.h"
char *malloc();
/*
** returns the token that was expected when error occurred
**
** entry- la is errant token
** exit- AnchLa has first usable token of input, ie: in the anchor set
**       return a token that the parser would have accepted if parser
**       saw that token instead of la.
*/
short
parErr(la,rule)
short rule,la;
{
    static char errmsg[] = "Parse error in rule %d";
    char *p;
    int i,cas;
    static short *caseMap;
    #ifdef DEP
    short *s;
    ErrRecovering = 1;          /* nextterm examines, advparse clears it */
    /*
    ** Provide a compatible stack for the error recovery automaton.
    ** Need to translate between compact case labels, into
    ** sparse states. This implies a mapping table. We built it
    ** once per run if error recovery is needed.
    */
    #define NC 7
    #define NE 8
    #define NES 20
    If( (caseMap=(short *)malloc(300)) == NULL)
        printf("parErr: malloc failed\n");
    cas=0;
    for(i=0; i< 728; i++)
        If(Op[i]==NE || Op[i]==NES || Op[i]==NC)
            caseMap[cas++] = i;

    s = DEPStack;          /* provide a stack for automaton */
    Sp = 0;
    while( s<DEPSp )
    {
        printf(" %d ", caseMap[*s]);
        Stk[Sp++] = caseMap[*s++];
    }
    printf("\n");
    #endif
    If( (p=malloc(sizeof(errmsg)+8)) == NULL) /* 8 for decimal expansion */
        printf("parErr: malloc failed\n");
    else
    {
        sprintf(p, errmsg, rule-1);
        message(ERROR,p,0,&curpos);
        /* message(ERROR, &curpos, "Syntax error"); */
    }

    DB(PARERR,("Parse error in rule %d la=%d\n", rule-1,la) );
    Stk[Sp] = rule;          /* overwrite top with current rule */
    getanch();
    AnchLa = advToAnch(la); /* save the first useful token after discarding */
    return(advparse());
}

```

parseDEP.c

```

#define SYNPUTEOF 0
#include "macros.h"
void
poflo(x)
short r;
{
    printf("***Stack overflow pushing %d\n", r );
    exit(3);
}

short DEPStack[MAXSTK], *DEPSp;

short
nextterm()
{
    return( ErrRecovering?advparse():lexAttr() );
}

void
yyparse()
{
    register DECTYPE *dirSet = virtDir;
    register short *p;
    register short la;

#ifdef DEBUG
#define PU(x) {if (DEPSp == &DEPStack[MAXSTK]) poflo(x);\
                *DEPSp++ = x;\
                DB(PARDEP,("PARDEP: push(%d) ",x));\
                DBFOR(p=DEPStack,p<DEPSp,p++)\
                    DB(PARDEP,("%d ",*p));\
                DB(PARDEP,("\n"));\
            }
#else
#define PU(x) *DEPSp++ = x
#endif

    if(SERIALnum != serialNum)
        printf("mismatch between Direc.h and Direc.c\n");

    DEPSp = DEPStack; *DEPSp++ = 0; la = nextterm();
    goto L0;

pppop:
#ifdef DEBUG
    DB(PARDEP,("PARDEP: pppop: "));
    DBFOR(p=DEPStack,p<DEPSp,p++)
        DB(PARDEP,("%d ",*p));
    DB(PARDEP,("case: %d\n",*(DEPSp)));
#endif

    switch (*(--DEPSp)) {
    case 0: printf("\ncase 0:\n");break;
#include "tbl.i"
    default: printf("default case, \n");
    }
    if(la != SYNPUTEOF)
        printf("extra stuff \n");
    DB(PARDEP,("PARDEP: PARSE SUCCESSFUL\n"));
}

```

parseITR.c

```

#define SYNPUTEOF 0
#define JMP Stk[Sp]=Datum[rule]
#include "macros.h"
void
yyvsparse()
{
    register DECTYPE *dirSet = virtDir;
    register short la,rule,i;
        if(SERIALnum != serialNum)
            printf("mismatch between Direc.h and Direc.c\n");
        la = lexAttr(); /* init LookAhead */
        DB(PARITR,("rule Op[rule] Dset[] Datum[] Stk befor switch\n"));
        Sp = 1; Stk[Sp] = 0;
        do {
            rule = Stk[Sp]++; /* get current rule and prepare for next */
            DB(PARITR,("%d\t%s\t%d\t%d\n",
                rule, instrCod(Op[rule]), Dset[rule],Datum[rule]));
            DBFOR(i=0,i<=Sp,i++) DB(PARITR,(" %d",Stk[i]));
            DB(PARITR,("\n"));

#include "opcase.i" /* shorthand for case statement */

            switch (Op[rule]) {
                ARS AR    action(Datum[rule]);
                        /* Fall thru */
                RDS RD    la=lexAttr();
                        break;
                RE        IF_NOT(Dset[rule],la)
                        la = parErr(la,rule);
                        else la=lexAttr();
                        break;
                AC        action(Datum[rule]);
                        break;
                RA        IF_NOT(Dset[rule],la) la = parErr(la,rule);
                        else { action(Datum[rule]); la=lexAttr(); }
                        break;
                NC        PUSH
                        break;
                NE        IF_NOT(Dset[rule],la) la = parErr(la,rule);
                        else PUSH
                        break;
                NR        If (Dset[rule] != 0)
                        IF_NOT(Dset[rule],la) la = parErr(la,rule);
                        else POP;
                        /****** /
                        break;
                JP        JMP;
                        break;
                JI        IF_IN(Dset[rule],la) JMP;
                        break;
                JO        IF_NOT(Dset[rule],la) JMP;
                        break;
                JE        IF_NOT(Dset[rule],la) la = parErr(la,rule);
                        else JMP;
                        break;
                EO        IF_NOT(Dset[rule],la) la = parErr(la,rule);
                        break;
                RES        If(Dset[rule]!=la) la = parErr(la,rule);
                        else la=lexAttr();
                        break;
                RAS        If(Dset[rule]!=la) la = parErr(la,rule);
                        else { action(Datum[rule]); la=lexAttr(); }
                        break;
                NES        If(Dset[rule]!=la) la = parErr(la,rule);

```

```

                                else PUSH                                break;
NRS      if (Dset[rule] != 0)
                                if(Dset[rule]!=la) la = parErr(la,rule);
                                else POP;
                                break;
JIS      if(Dset[rule]==la) JMP;                                break;
JOS      if(Dset[rule]!=la) JMP;                                break;
JES      if(Dset[rule]!=la) la = parErr(la,rule);
                                else JMP;                                break;
EOS      if(Dset[rule]!=la) la = parErr(la,rule);                                break;
                                default: PANIC("default of parser");
                                }
} while (Sp > 0);
if(la != SYNPUTEOF)
    printf("extra stuff after complete program\n");
DB(PARITR,("PARITR: PARSE SUCCESSFUL\n"));
}

```

macros.h

```

#include <stdio.h>
#define LEX          0x0001
#define ACT          0x0002
#define ADVANCH      0x0004
#define CHOOSE       0x0008
#define TOKGEN       0x0010
#define ADVPAR       0x0020
#define GETANCH      0x0040
#define PARERR       0x0080
#define PARDEP       0x0100
#define PARITR       0x0200
#ifdef DEBUG
#define DB(what,rest)      if(Debug&what) printf rest
#define DBFOR(a,b,c)      for(a;b;c)
#else
#define DB(what,rest)
#define DBFOR(a,b,c)
#endif

#define PANIC(c) {printf("PANIC: %s, aborting\n", c); fflush(stdout);abort();}
#define MAXSTK 128
#define PUSH
/* DBFOR(p=DEPStack,p<DEPStackSize) */
{if(Sp>=MAXSTK) PANIC("Parse stk overflow\n");\
Sp++; Stk[Sp] = Datum[rule];\
DB(PARITR,("PARITR: push(%d) ",Stk[Sp]));\
DB(PARITR,("\n"));\
}
#define POP      Sp--
/*
#define POP      {Sp--; \
DB(PARITR,("PARITR: pop(%d)\n",Stk[Sp]));\
}
*/

#include "Direc.h"          /* parameters of director set— NUMSYMS and WIDTH */
/* Check membership in director set by set number and symbol */
#define IF_NOT(set,sym) if(!(dirSet[sym+(set /WIDTH)*NSYMS]&(1<<(set%WIDTH))))
#define IF_IN(set,sym)  if( (dirSet[sym+(set /WIDTH)*NSYMS]&(1<<(set%WIDTH))))

/* EXTERNAL DECLARATIONS */
void getanch(), action(), yyparse(), tokgen();
short parErr(), nexterm(), lexAttr(), advToAnch(), advparse(), choose();
char *instrCod(), *tokCod();

extern short Anchors[];
extern short Op[], Dset[], Datum[];

extern short Sp, AnchLa, ErrRecovering;
extern short Stk[];
extern short DEPStack[], *DEPStackSize;
extern short Debug, serialNum;

```

Makefile

```

# ITR is interpretive parser, DEP is directly executing parser
GRAM=      pascal.g
CTL=       pascal.ctl
CF         = -O -DDEBUG
CFLAGS=    -I. -IS(INCL) $(CF)
LIBS=      $$CST/lib/frame.a
INCL=      $$CST/include
XSRC=      $$CST/lib/lex.c $$CST/lib/idn.c $$CST/src/lib/lnt.c $$CST/src/lib/str.c \
          $$CST/src/lib/fpt.c $$CST/src/lib/src.c $$CST/src/lib/err.c $$CST/lib/csm.c

HDRS=      Direc.h macros.h
GENSRC=    lexAttr.c advparse.c advToAnch.c parseDEP.c instrCod.c \
          getanch.c main.c parErr.c tokCod.c action.c \
          data.c parErr.c advparse.c Direc.c
DIRSRC=    parseDEP.c
ITRSRC=    parseITR.c
GENOBJ=    lex.o idn.o csm.o lexAttr.o advToAnch.o instrCod.o \
          getanch.o main.o tokCod.o action.o Direc.o data.o \
          Op.o Dset.o Datum.o
DIROBJ=    parErrDEP.o advparseDEP.o parseDEP.o
ITROBJ=    parErrITR.o advparseITR.o parseITR.o
SCRIPTS=   OpDsetDatum.awk action.awk dir.awk t1.awk t3.awk tok.lex.awk

dep:       Direc.h $(GENOBJ) $(DIROBJ)
           cc $(CFLAGS) -DDEP $(DIROBJ) $(GENOBJ) $(LIBS)
           mv a.out dep

itr:       $(GENOBJ) $(ITROBJ) macros.h
           cc $(CFLAGS) $(ITROBJ) $(GENOBJ) $(LIBS)
           mv a.out itr

yac:       ylex.o driver.o y.tab.o yylex.o idn.o csm.o
           cc $(CFLAGS) ylex.o driver.o y.tab.o yylex.o idn.o csm.o $(LIBS) -o yac

symbols:   pascal.par
           sed -f $$HOME/bin/yacc.sed pascal.par | sort | uniq > symbols

parser.y:  newcodes pascal.par
           $$HOME/bin/mkyacc.sh newcodes pascal.par

y.tab.c y.tab.h: parser.y
           yacc -d -v parser.y

print:
           print Makefile $(HDRS) $(GENSRC) $(DIRSRC) $(ITRSRC) $(SCRIPTS)

lexOut:    tables
           cvtdir <tables
CTL.x:     $(CTL) lexOut
           (cat $(CTL); sort lexOut) >CTL.x

term.h:    CTL.x
           -ln $$HOME/bin/V$FILE .
           -ln $$HOME/bin/ZERDAT .
           bgla CTL.x
           rm $FILE ZERDAT

tokCod.o:  tokCod.c tokCod.i
           cc -c tokCod.c

lexMap.i:  newcodes
           sort +2 -n newcodes/ awk -f tok.lex.awk

tokCod.i:  newcodes

```

```

sort +l -n newcodes| awk -f tok.lex.awk

action.o:          action.c action.i
cc -c action.c
cc -c $(CFLAGS) $<

idntbl.h csmtbl.h: newcodes
adinit -f -c newcodes

lex.o: term.h lexMap.i
cc -c $(CFLAGS) lex.c
main.o: tbl.i
parseDEP.o: tbl.i

lintDEP:
lint -I. -I$(INCL) -DDEP $(XSRC) $(GENSRC) $(DIRSRC)

lintITR:
lint -I. -I$(INCL) $(XSRC) $(GENSRC) $(ITRSRC)

parErrDEP.o: parErr.c
cc -c $(CFLAGS) -DDEP parErr.c
mv parErr.o parErrDEP.o
parErrITR.o: parErr.c
cc -c $(CFLAGS) parErr.c
mv parErr.o parErrITR.o

advparseDEP.o: advparse.c
cc -c $(CFLAGS) -DDEP advparse.c
mv advparse.o advparseDEP.o
advparseITR.o: advparse.c
cc -c $(CFLAGS) advparse.c
mv advparse.o advparseITR.o

csm.o: csmtbl.h
cc $(CFLAGS) -DINIT -c $$CST /lib/csm.c

idn.o: idntbl.h
cc $(CFLAGS) -DINIT -c $$CST /lib/idn.c

tables: $(GRAM)
synput $(GRAM) > $(GRAM).out
tbl.i: t1.awk t3.awk tables
# expand RD 5* to RDS 5
awk -f t1.awk tables >xtbl
awk -f t3.awk xtbl >tbl.i;
# to get Op.c Dset.c Datum.c
awk -f OpDsetDatum.awk xtbl
rm -f xtbl

Direc.h: tables
cvtdir <tables
Direc.c: tables
cvtdir <tables

cleanall: clean
rm -f cvtdir tables newcodes tokCod.i lexMap.i parser synput.h

clean:
rm -f tbl.i dep itr $(GENOBJ) $(ITROBJ) $(DIROBJ) \

```