A PARSER MODIFICATION OF THE EUCLID COMPILER: AUTOMATIC GENERATION OF SYNTAX ERROR RECOVERY

by

GRACE EVANS

B. M., University of Michigan, 1962

M. M., University of Michigan, 1963

A MASTER'S REPORT

submitted in partial fulfillment of the

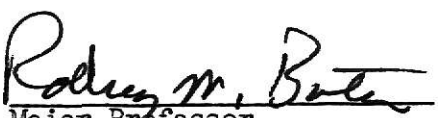
requirements for the degree

MASTER OF SCIENCE

KANSAS STATE UNIVERSITY
Manhattan, Kansas

1982

Approved by:



Major Professor

ACKNOWLEDGEMENTS

With grateful thanks to Dr. Rodney Bates for his patience and help and to my husband, Bill, for his constant support and encouragement.

THIS BOOK CONTAINS NUMEROUS PAGES WITH DIAGRAMS THAT ARE CROOKED COMPARED TO THE REST OF THE INFORMATION ON THE PAGE.

THIS IS AS RECEIVED FROM CUSTOMER.

TABLE OF CONTENTS

INTRODUCTION

Euclid is a relatively new programming language, designed to enable the construction of verifiable systems programs. [3] This project modified the Euclid compiler which was developed at the University of Toronto in 1977. [1] The original Toronto compiler provided minimal syntax error recovery in the parser, and this project was undertaken with three aims:

1) to provide more efficient syntax error recovery,
2) to provide a syntactically correct output token stream to the new pass, and
3) to demonstrate that the first two aims can be automatically generated.

Hartmann's error recovery scheme serves as the foundation for the generation of syntax error recovery. But in order to provide flexibility for potential future changes in the actual parsing program, the intent of this project is to provide the capability for automatic generation of the syntax error recovery, rather than being locked into any one particular language being parsed. The Euclid parser modules were modified with this end in mind.

Any parser written in S/SL consists of four parts: 1) an S/SL program, 2) an S/SL assember, 3) output from the S/SL assembler in table form, and 4) a table interpreter. The S/SL program contains the actual parsing procedures, and the other three parts implement its actions.

SYNTAX/SEMANTIC LANGUAGE

Syntax/Semantic Language (S/SL) is a modest programming language which was designed at the University of Toronto for use in the implementation of compilers. [4] It was developed from the use of syntax and semantic charts, which contain inputs, outputs, error messages, and semantic action calls all in the same diagram. [2] These charts could only be hand read and checked, and S/SL is a textual notation equivalent to them. Implemented by a translator/interpreter combination, as in the Euclid compiler, an S/SL program representing the syntax of a language can be computerized, which greatly simplifies compiler development.

LANGUAGE FEATURES

Much of the following background information is derived from a paper written by the authors of S/SL. [4]

S/SL is used for control only; the language contains no data or assignment statements. It consists only of sequences, repetitions (cycles), and choice of action statements which are combined to form rules, or procedures. A rule terminates with a semi-colon. Input, matching, or output of tokens and emission of error signals are the other features of S/SL. In some instances, an S/SL program may invoke semantic operations written in another language, such as Pascal, to manipulate data, but the parser pass of the Euclid compiler does not utilize a semantic mechanism, relying only on S/SL to perform the parsing and translation to a suitable output token stream for later passes of the Euclid compiler. Typical S/SL programs are, in fact, recursive descent

parsers written in S/SL.

The S/SL language dialect used by the Euclid parser has seven S/SL actions or statements:

1) the call action: "@" followed by a rule name indicates the calling of a rule, e.g., @TypeDeclaration,
2) the return action: "<" is found at the end of a rule and indicates that control should return to the point at which the rule was called,
3) the cycle (repeat) action: actions inside the braces are repeated until a cycle exit symbol (">") is encountered

```
{
  statements
}
```

4) the input (match) action: an input token name indicates the next input token read should match the token,
5) the input choice action: the brackets indicate a choice of actions, and the "=" indicates that the next input token should match one of the labels in this choice. The statements following the matched label and colon are then performed. An asterisk ("*") indicates an otherwise clause, which is taken if the current input token does not match a label