NASA Contractor Report 172541

ICASE REPORT NO. 85-4

ICASE

AN ALGORITHM FOR GENERATING ABSTRACT SYNTAX TREES

Robert E. Noonan

Contract No. NAS1-17070

January 1985

INSTITUTE FOR COMPUTER APPLICATIONS IN SCIENCE AND ENGINEERING
NASA Langley Research Center, Hampton, Virginia 23665

Operated by the Universities Space Research Association

National Aeronautics and
Space Administration

Langley Research Center
Hampton, Virginia 23665

Introduction

A compiler takes as input a program in some source language, such as Pascal or C, and produces as output an equivalent program in machine language. Compilers are commonly broken down into phases as shown in Figure 1, where a phase transforms the representation of the source program. Excellent introductions to compilers and their underlying theory can be found in the texts [Aho, 1977], [Barrett, 1979], and [Waite, 1983].

The purpose of the lexical analysis phase or scanner is to read characters from the source file, coalescing them into logically related groups called tokens. The usual tokens are language keywords, integer and real numbers, operator symbols such as ":=", etc. These tokens are passed in a stream to the next phase of the compiler.

The syntactic analysis phase or parser groups tokens into syntactic entities, such as expresssions, statements, etc. The output of this phase is either a parse tree, or more commonly, an abstract syntax tree. In the former, roots of subtrees represent nonterminal symbols of the grammar, while leaves represent terminal grammar symbols. In an abstract syntax tree operators are used as root nodes, while leaves represent operands; furthermore, purely syntactic operators, such as ";" in Pascal, are often discard from the output tree. This latter point accounts for the term "abstract" in the name.

Consider the grammar for arithmetic expressions presented in Figure 2, where <no> represents an arbitrary number. A parse tree for the expression "5 * (4 + 3)" is given in Figure 3, while an abstract syntax tree is given in Figure 4. Notice that the parentheses used for syntactic grouping do not appear in Figure 4.

In this paper we present an algorithm for determining the form of an abstract syntax tree directly from the grammar. We discuss an implementation of this algorithm named TREEGEN and its use in the development of a Modula 2 compiler.

The Algorithm

There is surprisingly little literature on the notion of abstract syntax trees. [Aho, 1977] and [Waite, 1983] each devote less than a page to the subject, while [Barrett, 1979] devotes no space to the concept. Thus, there is no strong basis on which to develop an algorithm, other than experience with various compilers.

Our goal was to be able to derive the abstract syntax directly from a grammar in standard BNF. We chose to ignore various, admittedly useful extensions to BNF grammars so that the grammar used would be acceptable to a wide variety of parser generators, including YACC

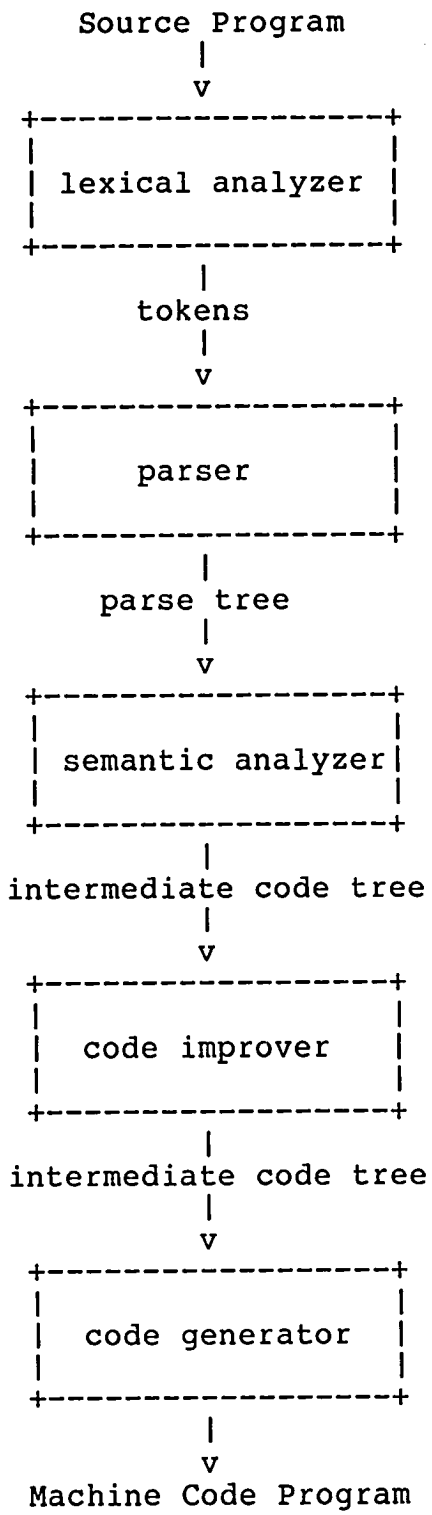

Figure 1: Phases of a Compiler

[Johnson, 1979], Mystro [Collins, 1980], LR [Wetherell, 1981], etc. Because of the manner in which LR and LL parsers in these systems apply semantic actions, there must be a close connection between the rules of the grammar and the abstract syntax.

In developing an algorithm, we first established the following principles.

-- Following [Aho, 1977], the abstract syntax is a tree with operators as root nodes and operands as leaf nodes.

-- Terminal symbols representing only punctuation are to be discarded.

-- As noted above, there must be a close tie between the production rules of the grammar and the abstract syntax. This imposes a burden on the grammar writer no greater than the one imposed by use of a parser generator.

-- Each operator should have a unique interpretation. In particular, the number of operands must be uniquely determinable from the operator. It would be a violation of this principle to use an operator named "case" to represent both a case statement and the variant part of a record in Pascal [Jensen, 1976], since these are distinct constructs in the language.

-- Subject to the above, the total number of operators should be minimized. Different forms of the same basic construct should all use the same operator.

Producing an abstract syntax tree directly from a grammar requires that operand terminal symbols be distinguishable from operator terminal symbols. For most programming language grammars the set of operand terminal symbols is identical to the set of terminal symbols in angle brackets, i.e., enclosed in "<" and ">". An implementation could provide for the addition and deletion of symbols to this set.

A naive algorithm can be based on a simple analysis of individual grammar rules or productions:

(1) If the righthand side of the production is empty, then the subtree associated with this production is the nil-ary operator "nil_root".

(2) If the righthand side of the production consists of a single terminal symbol, then the subtree associated with this production is the nil-ary operator representing the rule, if the terminal is an operator, or the operand representing the terminal symbol.

(3) If the righthand side consists of a single nonterminal, then the lefthand side nonterminal inherits the tree of the righthand side

```
<goal>   ::= <expr>
<expr>   ::= <expr> + <term>
<expr>   ::= <term>
<term>   ::= <term> * <factor>
<term>   ::= <factor>
<factor> ::= ( <expr> )
<factor> ::= <no>
```

Figure 2: Arithmetic Expression Grammar

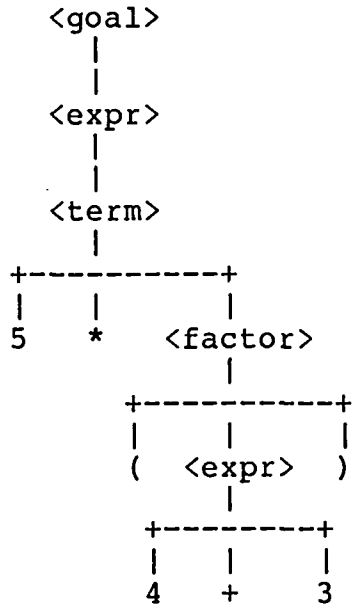

Figure 3: Parse Tree of 5*(4+3)

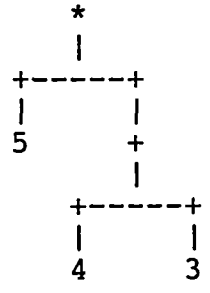

Figure 4: Abstract Syntax Tree of 5+(4*3)

nonterminal.

(4) Whenever the righthand side consists of more than a single symbol, the operands are the nonterminals and operand terminals in the righthand side and the operator name is chosen to uniquely represent the rule. The latter can be accomplished as follows. If the first terminal operator symbol in the righthand side is unique in the grammar then it can be used as the operator name. Otherwise, the operator name is derived from the lefthand side symbol.

Applying the above algorithm to the arithmetic expression grammar of Figure 2, we get the trees shown in Figure 5. Note that in Figure 5, the trees are given immediately after each production; a "tree" consisting of a single nonterminal means that the lefthand side inherits the tree of the associated righthand side nonterminal.

As can be seen, this naive algorithm works fairly well for this simple grammar. Only for the sixth rule does it generate an undesired tree, namely, the "(<expr>" rather than merely inheriting the subtree for <expr>. In this work it was felt that it would be easier for the user to delete unwanted trees, than to add missing subtrees.

For larger, more complex grammars this simple set of rules proves to be inadequate. In many programming language grammars, many rules are used to capture lists of various nonterminals, e.g., identifier lists, parameter lists, statement lists, etc. In order to cut down on the number of operators generated, all such lists can use the same generic operator.

(5) A recursive rule can be mapped to the n-ary operator "listof" whenever all of the operator terminals represent punctuation. Since there is no syntactic way to determine which symbols these are, an implementation must provide a way for the user to enumerate punctuation symbols.

A more complex problem arises from a language construct having several different forms. Consider the case of a set expression in Modula [Wirth, 1983] as shown below:

```
<set>  ::= { }
<set>  ::= { <element_list> }
<set>  ::= <qualident> { }
<set>  ::= <qualident> { <element_list> }
```

The naive algorithm above would generate four distinct operators for these rules, when, in fact, the rules are merely variants of a single, general form. One way to solve this problem is to rewrite the grammar as follows:

```
<goal>   ::= <expr>
   <expr>
<expr>   ::= <expr> + <term>
   + <expr> <term>
<expr>   ::= <term>
   <term>
<term>   ::= <term> * <factor>
   * <term> <factor>
<term>   ::= <factor>
   <factor>
<factor> ::= ( <expr> )
   ( <expr>
<factor> ::= <no>
   no
```

Figure 5: Abstract Syntax Trees for Expression Grammar

```
<set>           ::= <qualident?> { <element_list?> }
<qualident?>    ::=
<qualident?>    ::= <qualident>
<element_list?> ::=
<element_list?> ::= <element_list>
```

This poses an unnecessary burden on the grammar writer.

Thus, we were led to add the following rule:

(6) For a given nonterminal, collect all its nonrecursive productions. Pattern match each nonempty production against each longer one, starting with the longest. If the production is a special case of a longer one, the same abstract syntax is used with each missing operand replaced by the subtree "nil_root".

Applying this to the first set grammar given above, the trees shown will be generated:

```
<set>  ::= { }
       set nil_root nil_root
<set>  ::= { <element_list> }
       set nil_root <element_list>
<set>  ::= <qualident> { }
       set <qualident> nil_root
<set>  ::= <qualident> { <element_list> }
       set <qualident> <element_list>
```

The final rule attempts to further minimize the number of operators:

(7) For a given nonterminal which has no recursive productions, collect all of the productions for which rule 6 was not applicable and which have at most one operand. Distinct operators need not be generated if the abstract syntax is already distinct for each production.

Consider the example below taken from the grammar for Modula [Wirth, 1983]. Since a <qualident> can never be a "nil_root", the second rule can inherit the tree for <qualident> without introducing the operator "return_part."

```
<return_part> ::=
        nil_root
<return_part> ::= : <qualident>
        return_part <qualident>
<qualident>   ::= <ident>
        <ident>
<qualident>   ::= <qualident> . <ident>
        qualident <qualident> ident
```

The Program TREEGEN

The rules given above were implemented as a program named TREEGEN. The rules are applied in order except that rule 4 is applied last. The input to TREEGEN is a grammar in standard BNF, except that alternation within a single production is not allowed.

TREEGEN uses a column one convention for its input. A "<" in column one indicates the start of a new production, a "+" the continuation of a production, a "*" a line of comment, and a "#" a program directive.

Some directives are used to control the output of the program itself; these include a list of all the operators generated, a cross reference, etc. Another directive controls the addition and deletion of symbols to the set of operators representing punctuation. Another allows for the specification of a name for various terminal symbols such as "+", "-", etc.

After TREEGEN reads the directives, the user is presented with a menu of on-off directives and their current settings, so that he may invert any option. Next, TREEGEN reads, stores, and analyzes the entire grammar.

Then, each rule and its associated abstract syntax tree is presented to the user. Using rule 6 of Figure 2, this would appear as:

```
[6] <factor> ::= ( <expr> )
        factor <expr>
```

The user can either accept this tree or enter one of his own choice. To minimize typing, operands are indicated by their position in the righthand side of the rule. Thus, the desired choice of having <factor> inherit the abstract tree of <expr> can be entered by merely typing "2"; TREEGEN verifies that each number is entered at most once and corresponds to an operand. In the event of a question, TREEGEN prints the abstract syntax tree and asks the user to confirm it.

In generating operator names, TREEGEN uses heuristics for recognizing the use of the relational operators and for distinguishing between unary and binary "+" and "-". For these operators TREEGEN has default names wired into it.

In addition to the listings, there are two outputs from TREEGEN. One is a table that is used by the parser to automatically construct the abstract syntax tree associated with the production. The tree is constructed prior to any semantics associated with the rule being executed.

The other output is a set of constants, types, and routines for constructing the abstract syntax trees. For example, one of the types declared is an enumeration of all of the operator names generated. This text is automatically merged into a skeletal compiler which the user provides. It is our experience that this automatic text management is one of the most useful features of TREEGEN.

Experience with Modula

Appendix A presents a full Modula grammar and its associated abstract syntax. For the case of a production in which a nonterminal derives a single nonterminal the abstract syntax, was omitted for the sake of brevity. The grammar shown was taken directly from [Wirth, 1983] and run through a translator that converts extended BNF to standard BNF. The grammar appears to reflect the fact that it was derived from a recursive descent compiler.

The first ten rules of the Modula grammar were omitted since they define lexical tokens. A new goal rule was inserted as the first rule of the grammar. The syntax of <const_expr> was deleted and <expr> substituted for <const_expr> for the sake of brevity. Finally, the syntax for <expr> was rewritten with each operator being substituted for the appropriate nonterminal. It was interesting that on the parser generator being used, this had the effect of greatly increasing the state tables of the parser generator without increasing the resulting parse tables (due to optimization of the latter).

TREEGEN appears to do extremely well with this grammar. One problem already noted is that an extraneous operator is generated for the production in which <factor> derives a parenthesized expression.

This abstract syntax is being used in a Modula compiler under development. Most of the problems encountered appear to be due to the fact that the grammar is not always in the most appropriate form for an LR parser, rather than problems with the abstract syntax itself.

Conclusions

In this paper we have presented an algorithm for automatically deriving the form of the abstract syntax directly from the grammar. The algorithm was implemented and used to derive the abstract syntax of Modula for a compiler currently under development. The abstract syntax derived appears to have very few problems.

Acknowledgements

The naive algorithm originally presented was derived jointly with John Knight on the back of an airline boarding pass. Dewey Allen implemented the naive algorithm.

References

1. Aho, Alfred V., and Ullman, Jeffrey D. Principles of Compiler Design. Addison-Wesley, 1977.

2. Barrett, William A., and Couch, John D. Compiler Construction: Theory and Practice. SRA, 1979.

3. Collins, W. Robert, Knight, John. C., and Noonan, Robert E. A translator writing system for micro-computer high-level languages and assemblers, NASA-AIAA Workshop on Aerospace Applications of Microcomputers, (November 1980), 179-186.

4. Jensen, Kathleen, and Wirth, Niklaus. Pascal User Manual and Report. Springer-Verlag, 1975.

5. Johnson, S. C. YACC -- Yet another compiler-compiler. UNIX Programmer's Manual, Bell Laboratories, (January 1979).

6. Waite, William M., and Goos, Gerhard. Compiler Construction. Springer-Verlag, 1983.

7. Wetherell, Charles, and Shannon, Alfred. IEEE Trans. Soft. Engr., 7 (May 1981), 274-278.

8. Wirth, Niklaus. Programming in Modula-2. Springer-Verlag, 1983.

Appendix A: Abstract Syntax of Modula

```
# punctuation , ; |
# name <> not_eq
# name # not_eq
# name & and
* the above are program directives; this is a comment
*
<compilation>       ::= <comp_unit>
<qualident>         ::= <ident>
        ident
<qualident>         ::= <qualident> . <ident>
        qualident <qualident> ident
<const_decl>        ::= <ident> = <expr>
        const_decl ident <expr>
<element_list>      ::= <element>
<element_list>      ::= <element_list> , <element>
        listof <element_list> <element>
<set>               ::= { }
        set nil_root nil_root
<set>               ::= <qualident> { }
        set <qualident> nil_root
<set>               ::= { <element_list> }
        set nil_root <element_list>
<set>               ::= <qualident> { <element_list> }
        set <qualident> <element_list>
<element>           ::= <expr>
        element <expr> nil_root
<element>           ::= <expr> .. <expr>
        element <expr-1> <expr-2>
<type_decl>         ::= <ident> = <type>
        type_decl ident <type>
<type>              ::= <simple_type>
<type>              ::= <array_type>
<type>              ::= <record_type>
<type>              ::= <set_type>
<type>              ::= <pointer_type>
<type>              ::= <proc_type>
<simple_type>       ::= <qualident>
<simple_type>       ::= <enumeration>
<simple_type>       ::= <subrange_type>
<enumeration>       ::= ( <ident_list> )
        enumeration <ident_list>
<ident_list>        ::= <ident>
        ident
<ident_list>        ::= <ident_list> , <ident>
        listof <ident_list> ident
<subrange_type>     ::= [ <expr> .. <expr> ]
        subrange_type <expr-1> <expr-2>
<index_list>        ::= <simple_type>
```

```
<index_list>        ::= <index_list> , <simple_type>
        listof <index_list> <simple_type>
<array_type>        ::= ARRAY <index_list> OF <type>
        array_type <index_list> <type>
<record_type>       ::= RECORD <field_list_seq> END
        record <field_list_seq>
<field_list_seq>    ::= <field_list>
<field_list_seq>    ::= <field_list_seq> ; <field_list>
        listof <field_list_seq> <field_list>
<variant_list>      ::= <variant>
<variant_list>      ::= <variant_list> | <variant>
        listof <variant_list> <variant>
<field_else_part>   ::=
        nil_root
<field_else_part>   ::= ELSE <field_list_seq>
        field_else_part <field_list_seq>
<field_list>        ::=
        nil_root
<field_list>        ::= <ident_list> : <type>
        field_list1 <ident_list> <type>
<field_list>        ::= CASE <qualident> OF <variant_list>
+                       <field_else_part> END
        field_list2 nil_root <qualident> <variant_list>
                  <field_else_part>
<field_list>        ::= CASE <ident> : <qualident> OF <variant_list>
+                       <field_else_part> END
        field_list2 nil_root <qualident> <variant_list>
                  <field_else_part>
<variant>           ::= <case_label_list> : <field_list_seq>
        variant <case_label_list> <field_list_seq>
<case_label_list>   ::= <case_labels>
<case_label_list>   ::= <case_label_list> , <case_labels>
        listof <case_label_list> <case_labels>
<case_labels>       ::= <expr>
        case_labels <expr> nil_root
<case_labels>       ::= <expr> .. <expr>
        case_labels <expr-1> <expr-2>
<set_type>          ::= SET OF <simple_type>
        set_type <simple_type>
<pointer_type>      ::= POINTER TO <type>
        pointer <type>
<proc_type>         ::= PROCEDURE
        proc_type nil_root
<proc_type>         ::= PROCEDURE <formal_type_list>
        proc_type <formal_parm_list>
<formal_type_part>  ::= <formal_type>
<formal_type_part>  ::= VAR <formal_type>
        formal_type_part <formal_type>
<formal_type_seq>   ::= <formal_type_part>
<formal_type_seq>   ::= <formal_type_seq> , <formal_type_part>
```

```
        listof <formal_type_seq> <formal_type_part>
<return_part>       ::=
        nil_root
<return_part>       ::= : <qualident>
        <qualident>
<formal_type_list> ::= ( <formal_type_seq> ) <return_part>
        formal_type_list <formal_type_seq> <return_part>
<var_decl>          ::= <ident_list> : <type>
        var_decl <ident_list> <type>
<designator_tail>   ::= . <ident>
        designator_tail1 ident
<designator_tail>   ::= [ <expr_list> ]
        designator_tail2 <expr_list>
<designator_tail>   ::= ^
        caret
<designator_seq>    ::=
        nil_root
<designator_seq>    ::= <designator_seq> <designator_tail>
        listof <designator_seq> <designator_tail>
<designator>        ::= <qualident> <designator_seq>
        designator <qualident> <designator_seq>
<expr_list>         ::= <expr>
<expr_list>         ::= <expr_list> , <expr>
        listof <expr_list> <expr>
<expr>              ::= <simple_expr>
<expr>              ::= <simple_expr> = <simple_expr>
        eq <simple_expr-1> <simple_expr-2>
<expr>              ::= <simple_expr> # <simple_expr>
        not_eq <simple_expr-1> <simple_expr-2>
<expr>              ::= <simple_expr> <> <simple_expr>
        not_eq <simple_expr-1> <simple_expr-2>
<expr>              ::= <simple_expr> < <simple_expr>
        lt <simple_expr-1> <simple_expr-2>
<expr>              ::= <simple_expr> <= <simple_expr>
        lt_eq <simple_expr-1> <simple_expr-2>
<expr>              ::= <simple_expr> > <simple_expr>
        gt <simple_expr-1> <simple_expr-2>
<expr>              ::= <simple_expr> >= <simple_expr>
        gt_eq <simple_expr-1> <simple_expr-2>
<expr>              ::= <simple_expr> IN <simple_expr>
        in <simple_expr-1> <simple_expr-2>
<simple_expr>       ::= <simple_expr_tail>
<simple_expr>       ::= + <simple_expr_tail>
        plus <simple_expr_tail>
<simple_expr>       ::= - <simple_expr_tail>
        negate <simple_expr_tail>
<simple_expr_tail> ::= <term>
<simple_expr_tail> ::= <simple_expr_tail> + <term>
        add <simple_expr_tail> <term>
<simple_expr_tail> ::= <simple_expr_tail> - <term>
```

```
        minus <simple_expr_tail> <term>
<simple_expr_tail> ::= <simple_expr_tail> OR <term>
        or <simple_expr_tail> <term>
<term>             ::= <factor>
<term>             ::= <term> * <factor>
        multiply <term> <factor>
<term>             ::= <term> / <factor>
        divide <term> <factor>
<term>             ::= <term> DIV <factor>
        div <term> <factor>
<term>             ::= <term> MOD <factor>
        mod <term> <factor>
<term>             ::= <term> AND <factor>
        and <term> <factor>
<term>             ::= <term> & <factor>
        and <term> <factor>
<factor>           ::= <number>
        number
<factor>           ::= <string>
        string
<factor>           ::= <set>
<factor>           ::= <designator> <opt_actual_parm>
        factor1 <designator> <opt_actual_parm>
<factor>           ::= ( <expr> )
        factor2 <expr>
<factor>           ::= NOT <factor>
        not <factor>
<opt_actual_parm>  ::=
        nil_root
<opt_actual_parm>  ::= <actual_parms>
<actual_parms>     ::= ( <expr_list> )
        actual_parms <expr_list>
<actual_parms>     ::= ( )
        actual_parms nil_root
<opt_expr>         ::=
        nil_root
<opt_expr>         ::= <expr>
        <expr>
<statement>        ::=
        nil_root
<statement>        ::= <assignment>
<statement>        ::= <procedure_call>
<statement>        ::= <if_stmt>
<statement>        ::= <case_stmt>
<statement>        ::= <while_stmt>
<statement>        ::= <repeat_stmt>
<statement>        ::= <loop_stmt>
<statement>        ::= <for_stmt>
<statement>        ::= <with_stmt>
<statement>        ::= EXIT
```

```
        exit
<statement>         ::= RETURN <opt_expr>
        RETURN <opt_expr>
<assignment>        ::= <designator> := <expr>
        colon_eq <designator> <expr>
<procedure_call>    ::= <designator> <opt_actual_parm>
        procedure_call <designator> <opt_actual_parm>
<stmt_seq>          ::= <stmt>
<stmt_seq>          ::= <stmt_seq> ; <stmt>
        listof <stmt_seq> <stmt>
<elsif_part>        ::= ELSIF <expr> THEN <stmt_seq>
        elsif <expr> <stmt_seq>
<elsif_seq>         ::=
        nil_root
<elsif_seq>         ::= <elsif_seq> <elsif_part>
        listof <elsif_seq> <elsif_part>
<else_part>         ::=
        nil_root
<else_part>         ::= ELSE <stmt_seq>
        else_part <stmt_seq>
<if_stmt>           ::= IF <expr> THEN <stmt_seq> <elsif_part>
+                       <else_part> END
        if <expr> <stmt_seq> <elsif_part> <else_part>
<case_seq>          ::= <case>
<case_seq>          ::= <case_seq> | <case>
        listof <case_seq> <case>
<case_stmt>         ::= CASE <expr> OF <case_seq> <else_part> END
        case_stmt <expr> <case_seq> <else_part>
<case>              ::= <case_label_list> : <stmt_seq>
        case <case_label_list> <stmt_seq>
<while_stmt>        ::= WHILE <expr> DO <stmt_seq> END
        while <expr> <stmt_seq>
<repeat_stmt>       ::= REPEAT <stmt_seq> UNTIL <expr>
        repeat <stmt_seq> <expr>
<increment>         ::=
        nil_root
<increment>         ::= BY <expr>
        <expr>
<for_stmt>          ::= FOR <ident> := <expr> TO <expr> <increment> DO
+                       <stmt_seq> END
        for ident <expr-1> <expr-2> <increment> <stmt_seq>
<loop_stmt>         ::= LOOP <stmt_seq> END
        loop <stmt_seq>
<with_stmt>         ::= WITH <designator> DO <stmt_seq> END
        with <designator> <stmt_seq>
<proc_decl>         ::= <proc_heading> ; <block> <ident>
        proc_decl <proc_heading> <block> <ident>
<proc_heading>      ::= PROCEDURE <ident>
        proc_heading ident nil_root
<proc_heading>      ::= PROCEDURE <ident> <formal_parms>
```

```
        proc_heading ident <formal_parms>
<decl_seq>          ::=
        nil_root
<decl_seq>          ::= <decl_seq> <decl>
        listof <decl_seq> <decl>
<body_part>         ::=
        nil_root
<body_part>         ::= BEGIN <stmt_seq>
        begin <stmt_seq>
<block>             ::= <decl_seq> <body_part> END
        block <decl_part> <body_part>
<const_decl_seq>    ::=
        nil_root
<const_decl_seq>    ::= <const_decl_seq> <const_decl> ;
        listof <const_decl_seq> <const_decl>
<type_decl_seq>     ::=
        nil_root
<type_decl_seq>     ::= <type_decl_seq> <type_decl> ;
        listof <type_decl_seq> <type_decl>
<var_decl_seq>      ::=
        nil_root
<var_decl_seq>      ::= <var_decl_seq> <var_decl> ;
        listof <var_decl_seq> <var_decl>
<decl>              ::= CONST <const_decl_seq>
        decl1 <const_decl_seq>
<decl>              ::= TYPE <type_decl_seq>
        decl2 <type_decl_seq>
<decl>              ::= VAR <var_decl_seq>
        decl3 <var_decl_seq>
<decl>              ::= <proc_decl> ;
<decl>              ::= <module_decl> ;
<fp_seq>            ::= <fp_section>
<fp_seq>            ::= <fp_seq> ; <fp_section>
        listof <fp_seq> <fp_section>
<formal_parms>      ::= ( ) <return_part>
        formal_parms nil_root <return_part>
<formal_parms>      ::= ( <fp_seq> ) <return_part>
        formal_parms <fp_seq> <return_part>
<fp_section>        ::= <opt_var> <ident_list> : <formal_type>
        fp_section <opt_var> <ident_list> <formal_type>
<formal_type>       ::= <qualident>
<formal_type>       ::= ARRAY OF <qualident>
        formal_type <qualident>
<opt_priority>      ::=
        nil_root
<opt_priority>      ::= <priority>
<import_seq>        ::=
        nil_root
<import_seq>        ::= <import_seq> <import>
        listof <import_seq> <import>
```

```
<export_part>       ::=
        nil_root
<export_part>       ::= <export>
<module_decl>       ::= MODULE <ident> <opt_priority> ; <import_seq>
+                       <export_part> <block> <ident>
        module_decl ident <opt_priority> <import_seq> <export_part>
                    <block> ident
<priority>          ::= [ <expr> ]
        <expr>
<export>            ::= EXPORT <ident_list> ;
        export nil_root <ident_list>
<export>            ::= EXPORT QUALIFIED <ident_list> ;
        export qualified <ident_list>
<import_source>     ::=
        nil_root
<import_source>     ::= FROM <ident>
        ident
<import>            ::= <import_source> IMPORT <ident_list> ;
        import <import_source> <ident_list>
<defn_seq>          ::=
        nil_root
<defn_seq>          ::= <defn_seq> <defn>
        listof <defn_seq> <defn>
<defn_module>       ::= DEFINITION MODULE <ident> ; <import_seq>
+                       <export_part> <defn_seq> END <ident>
        defn_module ident <import_seq> <export_part>
                    <defn_seq> ident
<type_defn_part>    ::= <ident>
        type_defn_part ident nil_root
<type_defn_part>    ::= <ident> = <type>
        type_defn_part ident <type>
<type_defn_seq>     ::=
        nil_root
<type_defn_seq>     ::= <type_defn_seq> <type_defn_part>
        listof <type_defn_seq> <type_defn_part>
<defn>              ::= CONST <const_decl_seq>
        defn1 <const_decl_seq>
<defn>              ::= TYPE <type_defn_seq>
        defn2 <type_defn_seq>
<defn>              ::= VAR <var_decl_seq>
        defn3 <var_decl_seq>
<defn>              ::= <proc_heading> ;
        <proc_heading>
<program_module>    ::= MODULE <ident> <opt_priority> ; <import_seq>
+                       <export_part> <block> <ident> .
        program_module ident <opt_priority> <import_seq> <export_part>
                       <block> ident
<comp_unit>         ::= <defn_module>
<comp_unit>         ::= <program_module>
<comp_unit>         ::= IMPLEMENTATION <program_module>
        implementation <program_module>
```

1. Report No. NASA CR-172541 ICASE Report No. 85-4	2. Government Accession No.	3. Recipient's Catalog No.
4. Title and Subtitle AN ALGORITHM FOR GENERATING ABSTRACT SYNTAX TREES		5. Report Date January 1985 6. Performing Organization Code
7. Author(s) Robert E. Noonan		8. Performing Organization Report No. 85-4 10. Work Unit No.
9. Performing Organization Name and Address Institute for Computer Applications in Science and Engineering Mail Stop 132C, NASA Langley Research Center Hampton, VA 23665		11. Contract or Grant No. NAS1-17070, 13. Type of Report and Period Covered Contractor Report
12. Sponsoring Agency Name and Address National Aeronautics and Space Administration Washington, D.C. 20546		14. Sponsoring Agency Code 505-31-83-01

15. Supplementary Notes

Langley Technical Monitor: J. C. South, Jr.
Final Report

16. Abstract

In this paper, we discuss the notion of an abstract syntax. An algorithm is presented for automatically deriving an abstract syntax directly from a BNF grammar. The implementation of this algorithm and its application to the grammar for Modula are discussed.

17. Key Words (Suggested by Author(s))	18. Distribution Statement
abstract syntax parsing grammars	61 - Computer Programming and Software Unclassified - Unlimited

19. Security Classif. (of this report)	20. Security Classif. (of this page)	21. No. of Pages	22. Price
Unclassified	Unclassified	19	A02