1987

# A tool for compiler testing

Chung-Cho Oliver Cheng

*Rochester Institute of Technology*
*School of Computer Science and Technology*

# A Tool for Compiler Testing

By

Cheng, Chung-Cho Oliver

A thesis, submitted to
The Faculty of the School of Computer Science and Technology, in partial fulfillment of the requirements for the degree of Master of Science in Computer Science.

Approved by:

19 Nov 87

Dr. Peter G. Anderson, Chairman

11/19/87

Dr. Peter H. Lutz

11/19/87

Professor John A. Biles

October 26, 1987

# Abstract

A tool for compiler testing is implemented. The tool is driven by a BNF grammar extended by a device called control flags to incorporate those parts of a language that are not controlled by context-free rules (for instance, use of declared names). The tool produces two sets of program strings such that each production in the grammar is used at least once. One set is produced by Purdom's algorithm. The other is produced by a new algorithm designed by the author which builds a tree and then produces a set of strings from it for compiler testing. The program strings produced from the new algorithm differ less in length than those from Purdom's algorithm.



## Key Words and Phrases

Purdom's algorithm, BNF, Control flag, Context-free grammar, Context-sensitive grammar, Context-sensitive restriction.

## Computing Review Subject Codes

D.2.5 Testing and Debugging

D.3.1 Formal Definitions and Theory

F.4.2 Grammars and Other Rewriting Systems

F.4.3 Formal Languages

# Table of Contents

Appendix 2: Pascal Files

Part 1: Pascal's BNF and Control Flags

Part 2: The File, "rule.agr"

Part 3: The Files, "*.ags"

Part 4: The Partial Files, "*.ags"

# 1. Introduction

## 1.1 Problem Statement

Software testing is a vital activity occupying as much as forty percent of project resources. Software testing involves the execution of a software system with data and the comparison of results with the expected output for that data. As such, software testing is both maligned and vitally important activity. The major objection to software testing is that it only reveals the presence of software errors but never guarantees their absence.

However, testing's unpopularity with practitioners stems less from this than from two further reasons. First, software testing draws attention to the fallibility of the practitioners. Secondly, software testing is an extremely time-consuming and tedious activity[8].

The manual construction of test data sets comprises a large part of the validation and verification effort in a software project. It has been a major aim of testing researchers to automate these processes.

A particularly important problem in software production is to control the functional correctness of programs. This control is essential for complex, widely distributed programs such as compilers.

## 1.2 Automatic Generation of Test Data

There are seven categories for automatic generation of test data[8]. This thesis uses the method of syntax-based testing to implement a tool for compiler testing.

### 1.2.1 Random Testing

The automatic production of random test data is the default case by which other methods should be judged. It can only be used during integration testing or unit testing, where the aim is to achieve a high structural coverage.

Random testing has a number of advantages. First, it is inexpensive in terms of software. It only requires a random number generator and a small amount of support software. Secondly, it is straightforward to derive estimates of the operational reliability from random testing results. Thirdly, the use of randomly generated tests can be more stressing to software than manually generated test cases. For this reason it is very useful to employ random test data during the stress testing component of system and acceptance testing.

There are also disadvantages. First, there is no assurance that full coverage can be attained. For example, random inputs may never exercise large sections of program code which require equality between variables to hold. Secondly, it can be expensive in terms of human resources. It may mean examining the output from thousands of tests.

### 1.2.2 Adaptive Testing

Adaptive testing involves monitoring the test effectiveness of a test data set in order to generate further test data, the process being terminated when a maximum value of test effectiveness is achieved. Originally it was used in performance testing. Recent work on adaptive testing has used the number of errors discovered during execution as a test-effectiveness measure.

### 1.2.3 Syntax-Based Testing

This is the earliest reported method of automatically generating test data. Tools based on syntax process a description of test data expressed in a notation such as

BNF. These tools perform the reverse action of a compiler. Given a test data syntax they generate test sets which satisfy the syntax.

Syntax-based data generation does not depend on the structure of the program to be tested. Its major use is on projects which have produced compilers. Compiler developers have normally invested a large amount of effort in defining the syntax of a programming language. It is a natural step to generate test programs from the syntax. However, there is no reason why such methods cannot be used for other application areas. Payne[3] describes their use for the testing of overload conditions in a real-time system.

A major problem in generating test data for compiler testing is that of respecting the contextual dependencies in the language to be compiled. It is relatively easy to generate programs which are syntactically correct but are semantically wrong. There are similar problems when syntax-based test generation methods are used in other application areas. For example, a test generator for producing sequences of message for a real-time system might need to produce data in one message which have to be repeated in a predetermined pattern in later messages.

The advantage of using grammar-based test generation is that it forces the programmer to document test data structure precisely. It thus acts as a valuable formal document used in debugging and maintenance. A second advantage is that syntax-based test generator can produce large quantities of test data. Systems now exist which are able to ensure that each production in a grammar is exercised and that statistical distributions of test data are achieved and allow the tester to proceed in steps from simple general test data to the more complex kind.

There are two disadvantages in using syntaxed-based methods. First, there are classes of data which are impossible to generate. For example, sets of prime numbers cannot be generated using context-free rules. Secondly, writing the syntax rules for complex sets of test data can be a tedious process. This is the reason why syntax-

based automatic test data generation systems have normally been employed in compiler projects. This is because the derivation of a test data syntax is a by-product of the process of developing a compiler and requires little extra work from the tester.

### 1.2.4 Path-Oriented Methods

Path-oriented methods are based on deriving test data which exercise a path or series of paths through a program unit. To derive test data for a path through a program requires solving a set of inequalities. These inequalities represent conditions in constructs such as **if** statements and **repeat** statements. Many path-based data generation techniques depend on the generation of such inequalities and their numerical solution.

### 1.2.5 Specification-Oriented Methods

A program can be throughly structurally tested and still contain major errors. These errors will be due to a partial understanding or misunderstanding of a program unit's specification. Although one can detect some function errors by an examination of inequalities, a more effective strategy involves the generation of test data from a functional specification.

### 1.2.6 Data Flow Methods

There are two disadvantages in path-oriented testing. First, there is a distinct possibility that if sets of data were automatically generated to exercise all the paths through a program some of the data set might not be revealing, i.e., they might provide correct results for a path even when there is an error in a path. The second problem is that programs with loops may contain an infinite number of paths. Thus the criterion that all paths be traversed has to replaced by a weaker criterion that selects a subset of paths.

### 1.2.7 Regression Testing

During the development of a software system a large number of tests will have been performed. During maintenance some of these tests will need to be return in order to check that changes have not affected the original functions of the software.

## 1.3 Previous Work

In "A Formalized Technique for Expressing Compiler Exercisers" [3], Payne gives a formalized description of programs and their contents so that they may be generated in defined proportions for testing compilers. The first part of the paper surveys the previous work that has been done in the field. In the second section they develop a theoretical model of the problem. Part three describes a method for collecting information which can be used to derive the theoretical model. The fourth portion gives the algorithms for generating the test programs.

In "A Sentence Generator for Testing Parsers" [2], Purdom considers only the generation of sentences which comply with the syntax of a context-free grammar; he gives an efficient algorithm to produce a small set of sentences such that each production of the grammar is used at least once.

In "Compiler Testing Using a Sentence Generator" [4], a system for assisting in the testing phase of compilers is described. The definition of the language to be compiled drives an automatic sentence generator. The language is described by an extended BNF grammar which can be augmented by actions to ensure contextual congruence (e.g., between definition and use of identifiers). For deep control of the structure of the produced sample the grammar can be described by step-wise refinements: the generator is iteratively applied to each level of refinement, producing at last compilable, complete programs. The implementation is described

and some experimental results are reported concerning PLZ, MINIPL and some other languages.

In "An Automatic Generator for Compiler Testing" [5], a new method for testing compilers is presented. The compiler is exercised by compilable programs, automatically generated by a test generator. The generator is driven by a tabular description of the source language. This description is in a formalism, called context-free parametric grammars, which nicely extends context-free grammars in a context-dependent direction, but still retains the structure and readability of BNF. The generator produces a set of programs which cover all grammatical constructions of the source language, unless user supplied directives instruct otherwise.

## 1.4 Theoretical and Conceptual Development

This thesis intends to simplify the task of compiler testing. A tool was implemented for this purpose.

The definition of the syntax of a programming language is usually given in two parts. Most of the syntax is specified in the form of a context-free grammar which can be parsed efficiently. The context-free grammar describes the overall structure of a program but allows certain statements which are not allowed in the programming language. The remainder of the specification consists of context-sensitive restrictions placed on the context-free syntax. Such restrictions include type-matching rules for identifiers and the requirement that a call of a procedure contains exactly as many arguments as there are parameters in the definition of the procedure [3].

Each language has different context-sensitive restrictions; even different versions of a language have different context-sensitive restrictions. This thesis will focus on context-free grammars and some context-sensitive restrictions (for instances, use of declared names). Other context-sensitive restrictions will be noted to give users a convenient way to be beware of them. From the viewpoint of engineering economy the notation of context-sensitive restrictions is an efficient way to test compilers, because it is almost impossible to build all context-sensitive restrictions into a testing tool. Another important reason is whether or not context-sensitive restrictions are built into an automatic generator, the outputs are usually not "meaningful"; and if an attempt is made to execute them, the result are unpredictable and uncheckable. For example, I/O commands in a program can monitor the results, but usually the programs that are produced by an automatic sentence generator have no proper I/O commands in the proper place. For this reason some manual work is necessary.

# 2. Function Specification

## 2.1 Function Performed

The tool implemented in this thesis can generate two important output categories. The first category is generated from Purdom's algorithm[2]. Purdom's algorithm generates a set of source strings which is minimal and uses all the productions of the context-free grammar of the language. The second category, generated by a new algorithm derived by the author, first builds a tree and then produces a set of strings from it.

The program strings produced from the new algorithm differ less in length than those from Purdom's algorithm, and control flags are built into the BNF to deal with the restrictions of context-sensitive grammar.

### 2.1.1 Purdom's Algorithm

Algorithm PURDOM

1. Insert the axiom S in STACK, a pushdown stack whose content is the yield.
2. **loop1**: **while** (STACK is not empty)

```
{
    if ( the top of stack is a terminal symbol)
        pop and write it;
    else
        CHOOSE a rule A ->α rewriting the non-terminal symbol A and
            substitute α for A at the top of STACK;
} end of loop1;
```

The choice of a particular rule A-> α is made with the following strategy:

Algorithm CHOOSE

1. **if** (a rule A -> α exist which has not yet been usesd)

choose it;

**else if** (a derivation $A \overset{*}{\Rightarrow} \alpha \Rightarrow \gamma_1 B \gamma_2$ exist such that B is non-terminal symbol not in STACK, and a rule B -> β exists which has not yet been used)

SELECT one of A -> α;

**else**

use the rule which is the first step of a derivation having as its yield the shortest string derivable from A;

2.1.2 The Second Algorithm

The second algorithm builds a tree first, then the system uses the tree to generate program strings. Each tree node has following structure:

```
struct treenode {
    int ruleno;        /* the number of BNF rule */
    int sonsno;        /* the number of sons */
    int sp;            /* stack pointer for the following array */
    struct table *stack[30];    /* stack that stores the addresses of rules in the
                                   BNF rule table maintained by the system */
    int orderno;       /* the number of siblings */
    int active;        /* current active son's number */
    struct treenode *parent;
    struct treenode *sons[MAXSONS];
} *root, *currentnode, *cnode, *newnode;
```

Algorithm **Buildtree**

1. make a root node, and then store the first rule number into it (**root->ruleno** = 0).
2. store the pointer(s), which point to the information location(s) of the nonterminal(s) in the first rule, into the root's stack (**root->rule**).
3. **currentnode** = **root**;
4. **loop1**: **while** (there exists any rule which is not in the tree's **ruleno**)

```
{
    if (no leaf in the currentnode's right hand side)
        conde = left most leaf in the tree;
    else
        cnode = next leaf in the currentnode's right hand side;
    if (cnode is not empty in its stack)
    {
        pop the stack to get a nonterminal information;
        if (the nonterminal is already in the tree)
        {
            make a new node (newnode);
            newnode->ruleno = the shortest rule's number of the nonterminal;
            push the pointer(s), which point to the information location(s) of
                nonterminal(s) in the rule whose rule no. is newnode->ruleno,
                into newnode->stack;
            store necessary information into newnode, and link the newnode as
                the son of cnode;
        } else
```

{

**loop2: while** (there exists any rule of the nonterminal which is not in the tree's **ruleno**)

{

make a new node (**newnode**);

**newnode->ruleno** = one rule's number of the nonterminal;

push the pointer(s), which point to the information locations of the nonterminal(s) in the rule whose rule no. is **newnode->ruleno**, into **newnode->rule**;

store necessary information into **newnode**, and link the **newnode** as the son of **cnode**;

} end of **loop2**;

}

}

**currentnode** = the brand new son;

} end of **loop1**;

5. execute the algorithm **generate-test-data** (following).

Algorithm **generate-test-data**

**loop3**: **while** (all leaves have not been searched)

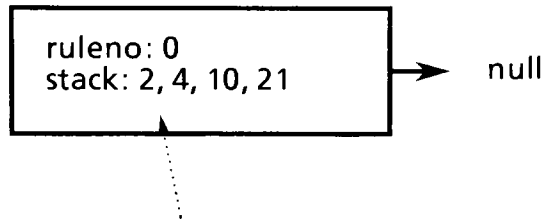
{

find a leaf, then store all of its ancester's rule numbers into a rule array;

use the rule array to generate a program string;

} end of **loop3**;

The following figures from 1 to 7 explain the initial steps of the algorithm **buildtree**; the input file is Prosmall's BNF in Appendix 1 Part 1.

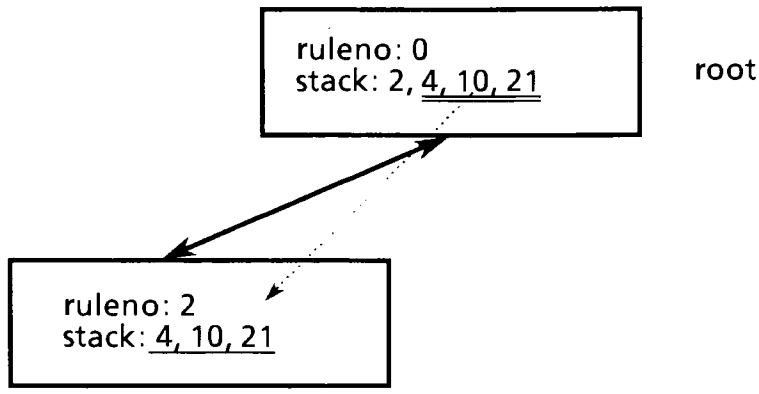



For ease of explanation, rule numbers for the first rule associated with each nonterminal is used instead of pointers to the information locations for the nonterminals.

# Figure 1

Snapshot for algorithm buildtree at step 3. The numbers are the rule number in Appendix 1 Part 2.

Related production rules:
0) <start>: <glbdecl> <procdec> main() <locdecl> begin <stmlst> end #
2) <glbdecl>: #
4) <procdec>: #
10) <locdecl>: #
21) <stmlst>: <stmt> #

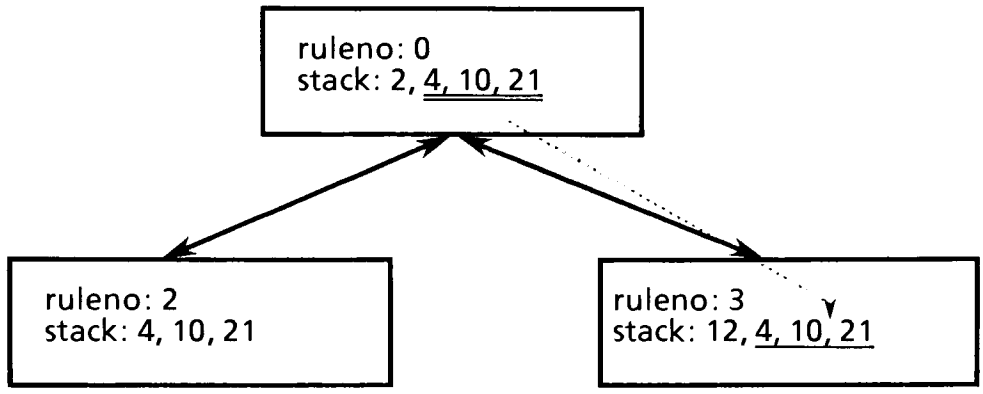



.....➤ Copy the numbers with double underline.

# Figure 2

Snapshot for algorithm **buildtree** in the first loop of step 4.

Related production rule:
2) <glbdecl>: #

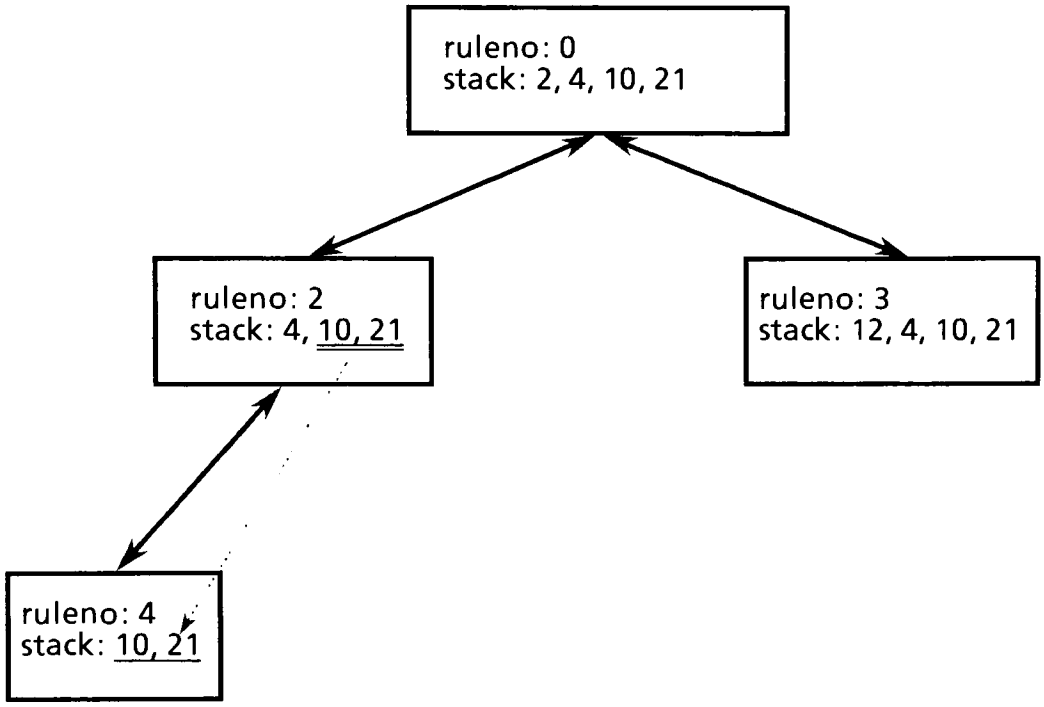



.....➤ Copy the numbers with double underline.

# Figure 3

Snapshot for algorithm **buildtree** after the first loop of step 4.

Related production rules:
2) <glbdecl>: #
3) <glbdecl>: global <decl> #
12) <declst>: <decln> #

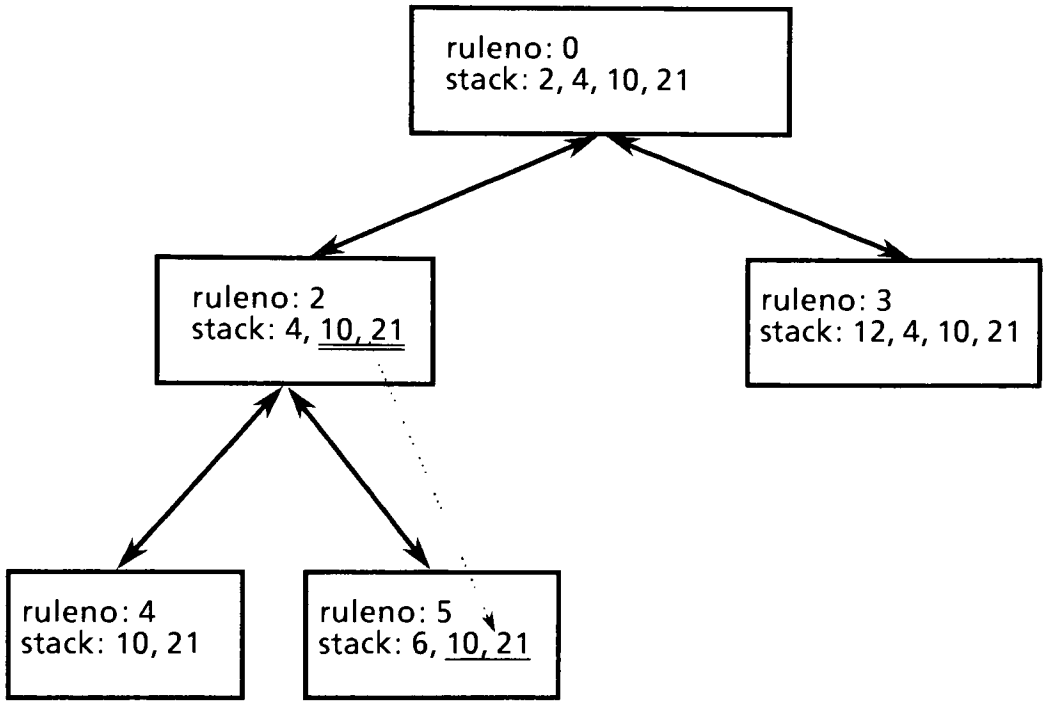



.....➤ Copy the numbers with double underline.

# Figure 4

Snapshot for algorithm **buildtree** in second loop in step 4.

Related production rules:
4) <procdec>: #

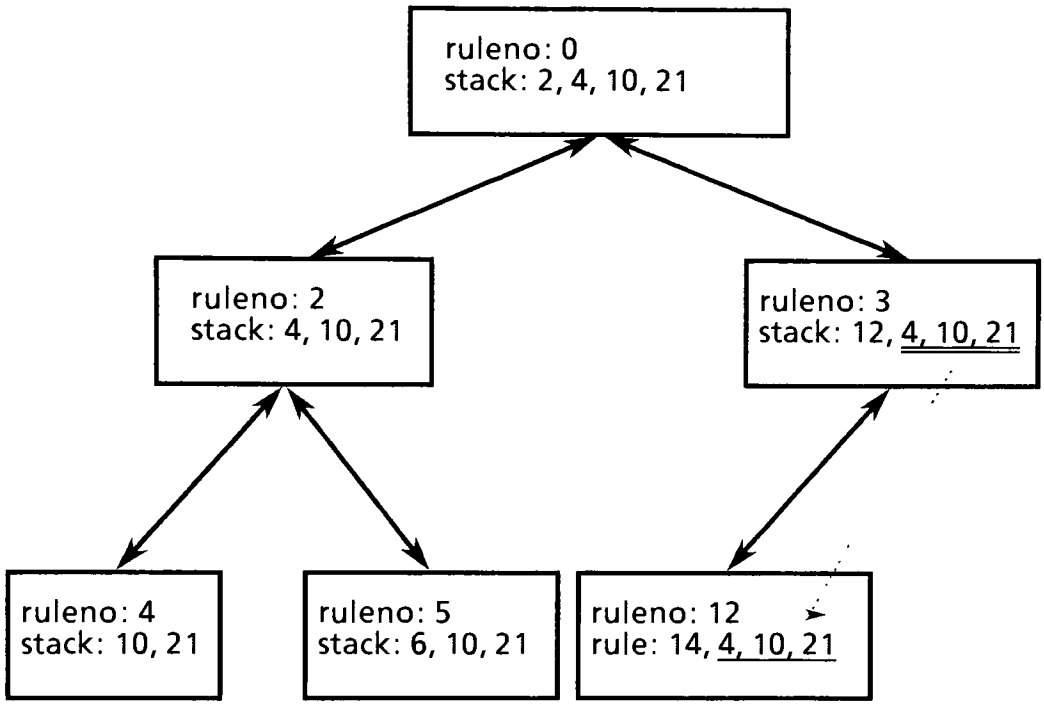



.....➤ Copy the numbers with double underline.

# Figure 5

Snapshot for algorithm **buildtree** after the second loop in step 4.

Related production rules:
4) <procdec>: #
5) <procdec>: <proclst> #
6) <proclst>: <procdef> #

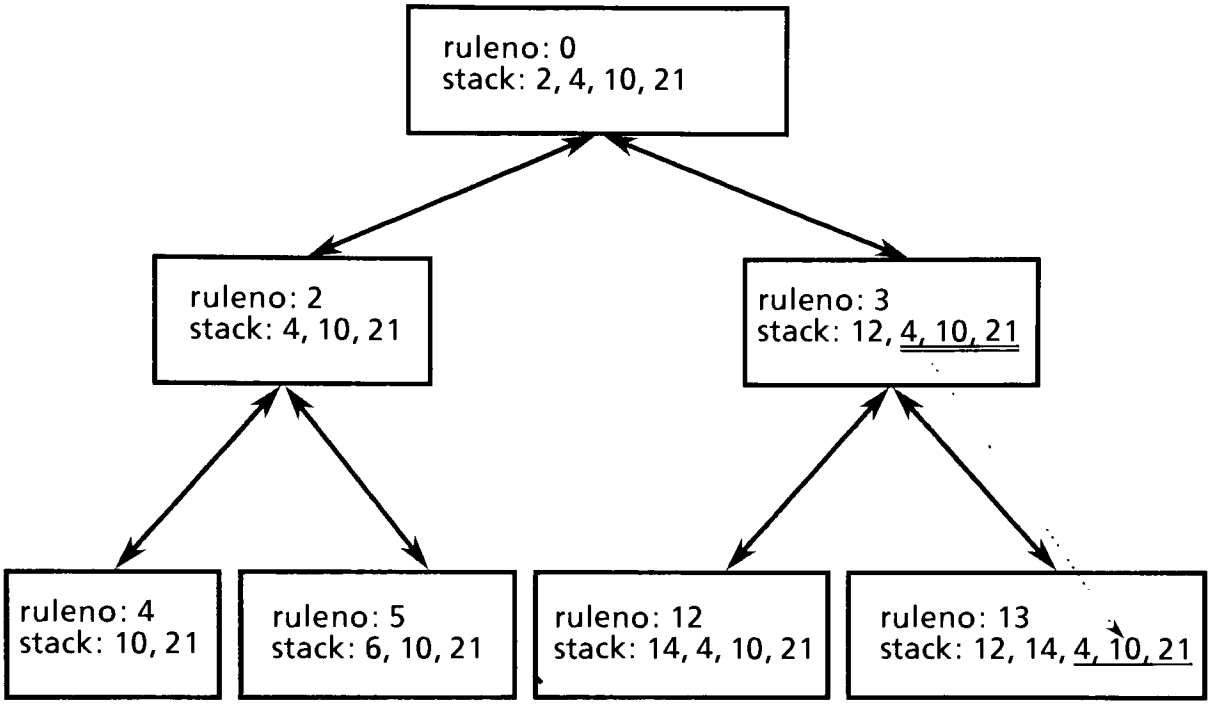



.....➤ Copy the numbers with double underline.

# Figure 6

Snapshot for algorithm **buildtree** in the third loop in step 4.

Related production rules:
12) <declst>: <decln> #
14) <decln>: <typ> <varlst> #




.....➤ Copy the numbers with double underline.

# Figure 7

Snapshot for algorithm **buildtree** after the third loop in step 4, and so on until every rule is in the tree's **ruleno**.

Related production rules:
12) <declst>: <decln> #
13) <declst>: <declst> <decln> #
14) <decln>: <typ> <varlst> #

### 2.1.3 Control Flags

Control flags are built into BNF to deal with the context-sensitive restrictions; they are put into BNF at appropriate places to check whether or not current program string is contradicting to context-sensitive restrictions. If a program string is contradicting to context-sensitive restrictions, then the system will fix the contradiction.

For example, if the running system meets a control flag, $filevariable, then it will do something to make sure the file variable declaration restriction is obeyed. Some tokens may be added to current program string if it is necessary.

This thesis builds control flags into the BNF for Pascal for generating compilable program strings from the second algorithm. Because the second algorithm can handle the control flags easier than Purdom's algorithm, I chose it as an example of implementation. See Chapter 3.

2.2 User Inputs

The principal input is a file which includes the context-free rules and control flags. The control flags are used to make the output readable, to obey context-sensitive restrictions, and to provide the users with messages.

To run this tool one types:

ag file_name

There are other inputs concerning debugging and user selection.

2.3 System Files

There are six groups of system files. They are described as follows:

1) input file

Every reasonable file name will be accepted by this tool. An input file includes context-free rules and control flags. The examples are listed in Appendix 1 Part 1 and Appendix 2 Part 1.

2) rule.ags

This file will be produced from an input file. This file contains a unique index number for every context-free rules in the input file. A user can find a context-free rule easily by using its index number. The examples are listed in Appendix 1 Part 2 and Appendix 2 Part 2.

3) *.agp

The "*" is a wild card symbol. This is a group of output files. They are produced by Purdom's algorithm. The example is listed in Appendix 1 Part 3.

4) *.agpr

A file in this group contains all the context-free rules that are used to produce a corresponding *.agp file. For example: "1.agpr" corresponds to "1.agp". The example is listed in Appendix 1 Part 4.

5) *.ags

This group is produced by the second algorithm which was mentioned in Section 2.1. The examples are listed in Appendix 1 Part 5 and Appendix 2 Part 3.

6) *.agsr

A file in this group contains all the context-free rules that are used to produce a corresponding *.ags file. For example: "23.agsr" corresponds "23.ags". The examples are listed in Appendix 1 Part 6 and Appendix 2 Part 4.

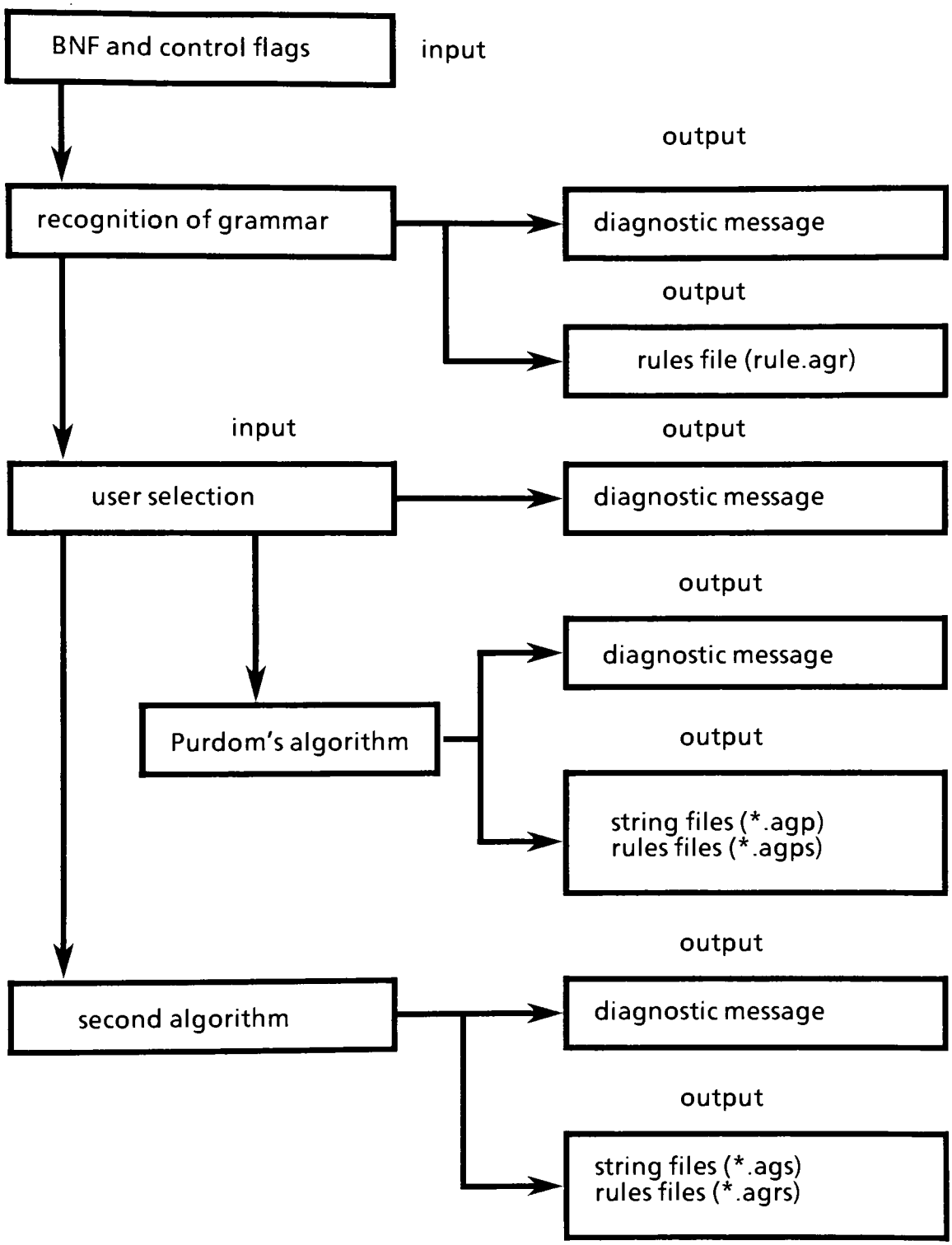
## 2.4 Equipment Configuration

The program was implemented on Pyramid 90X running OSx, a dual part of UNIX System V and 4.2 BSD UNIX. The documents were prepared on Xerox 8010 workstation.

## 2.5 Implementation Tools

All programs were written in C. There was no use of Yacc or Lex. Since C is a versatile language, suitable for system programming, and I am familiar with it, I chose to implement the thesis program in C.

## 2.6 System Data Flow Chart



# 3. Context-Sensitive Restrictions and Control Flags

In programming languages, context-sensitive restrictions usually are placed on the context-free syntax. For our particular purposes, the choice of a formal description tool has to be oriented towards sentence generation rather than recognition. Therefore, control flags are used to represent context-sensitive restrictions. Even though control flags are oriented to sentence generation in this thesis, they can be very useful for programming language specification; that will be discussed in Section 3.3.

A simple example is represented in Section 3.1 to illustrate the use of control flags in a clear way. It illustrates the Pascal rule requiring variable to be declared before use and that "strong typing" be followed. The only types are **int** and **char**; the only statement is assignment.

## 3.1 Simple Example for Control Flags

The BNF for our illustrative language is:

<start>: main() <locdecl> begin <stmlst> end #

<locdecl>: #

<lodecl>: decl <declst> #

<declst>: <decln> #

<declst>: <declst> <decln> #

<decln>: <typ> <varlst> #

<typ>: int #

<typ>: char #

<varlst>: $name #

<varlst>: <varlst>, $name #

<stmlst>: <stmt> #

<stmlst>: <stmlst> <stmt> #

<stmt>: $assgn $name = <exp>; $assgnend #

<exp>: <factor> #

<exp>: <factor> + <factor> #

<exp>: <factor> / <factor> #

<factor>: $number #

<factor>: $letter #

The control flags and their use are as follows.

When the generator reaches $name, if $name is not between $assgn and $assgnend, then the system will call function name() to generate an identifier, and maintain all necessary information in a table, including type and name. If $name is between $assgn and $assgnend, then the $name must have been declared, and the system should check the type matching restriction.

The $number indicates to the system to call function number() to generate a number; the number's type is certainly "int", so the system can check the type matching restriction.

The $letter indicates to the system to call function letter() to generate a letter; the letter's type is certainly "char", so the system can check the type matching restriction.

The C code to achieve this is listed below.

```
/*-------------------------------------------------------------------------------------------------------------
    This is an example program to illustrate how the system deals with the control
flags. Whenever the system generates a reserved word or control flag, it will call
generate().
-------------------------------------------------------------------------------------------------------------*/
int assgn_flag = 0;
struct sentence      /* This structure stores current program string */
{
    char *name;
    struct sentence *next;
} *poghead,          /* Pointer to program string's head */
  *current;          /* Pointer to current node */
struct variables     /* This structure stores the names and types of variables */
{
    char *name;
    int type;        /* type = 1 means "int"; type = 2 means "char" */
    struct sentence *next;
} *head,             /* Pointer to program string's head */
  *curr;             /* Pointer to current node */

int current_type = 1;
int temp_type = 1;
```

```
generate(string)
    char string[40];      /*represent a reserved word or control flag */
{   if (the "string" is first word in a string)
            initial(); /* The initial() initialize all the necessary information. */
    if (string[0] != '$')      /* since every string here is reserved word */
        add the "string" into current program string;
    else    switch (hash(string))   /* The hash() returns a hash number. */
        {   case 1:    /* $name */
                name();   break;
            case 2:    /* $number */
                temp_type = 1;
                number();    break;
            case 4:    /* $letter */
                temp_type = 2;
                letter();
            case 6:     /* $assgn */
                assgn_flag = 1;    break;
            case 9:    /* $assgnend */
                assgn_flag = 0;    break;
            default:
                add the "string" into current program string;      break;
                /* since every string here is a  underfined control flag */
        }
    if ( a program string is finished)
        save the string in a file;
}
```

## 3.2 Complete Example of Pascal Control Flags for the Pascal Language

Pascal is chosen for a complete example of control flags' implementation. I classify the context-sensitive restrictions into three groups, then discuss them with control flags.

### 3.2.1 Type-Matching Restrictions for Identifiers

Pascal requires type matching in assignment statements and expressions. To deal with those restrictions the following control flags are used.

$condition: This flag indicates to the system that a conditional expression follows; i.e., an expression whose type must be boolean.

$conditionend: This flag indicates the end of conditional expression. Every $condition must be followed by a $conditionend.

$checkfield: This flag indicates the system to check the restrictions of record's fields. Every record's field should be declared before use.

$in: This flag indicates to the system that the "in" operator follows.

$filevariable: This flag indicates to the system that a file variable follows.

$real: This flag indicates to the system that current type is real.

$assgn: This flag indicates to the system that the current statement is assignment.

$assgnend: This flag indicates the end of assignment statement. Every $assign must be followed by a $assgnend.

$with: This flag indicates the system current statement is "with".

$withend: This flag indicates the end of a "with" command. Every $with must be followed by a $withend.

$name: This flag indicates to the system to generate an identifier and to maintain necessary information.

$array: This flag indicates to the system that current type is array.

$arrayend: This flag indicates the end of array type. Every $array must be followed a $arrayend.

$record__variable: This flag indicates to the system that the current type is record.

$recordend: This flag indicates the end of a record type.

$boolean: This flag indicates to the system that the current type is boolean.

$pointer: This flag indicates that the current type is pointer.

$constantname: This flag indicates to the system to generate a constant name and check its restriction.

### 3.2.2 Context-Sensitive Restrictions for Functions and Procedures

Pascal requires that a call of a procedure or function contains exactly as many arguments as there are parameters in the definition of the procedure or function, and that the arguments have the same type as in the definition.

Pascal also requires that a function return a value and that the return value must have same type as its definition.

To deal with those restrictions the following control flags are used.

$paraflag: This flag indicates that function or procedure's parameter(s) follow.

$paraflagend: This flag indicates the end of parameter(s).

$funname: This flag indicates to the system to generate a function name.

$proname: This flag indicates to the system to generate a procedure name.

$checkfun: This flag indicates to the system to check whether or not a function has assigned a value to the function name, if not, an appropriate assignment statement will be added into the program string.

$proflag: This flag indicates to the system that a procedure follows.

### 3.2.3 Context-Sensitive Goto Restriction

Pascal requires that every label be declared properly. To deal with the restriction following control flags are used.

$label_integer: This flag indicates to the system to generate a unique label number.

$goto: This flag indicates to the system a goto statement follows so that the system can check the necessary restrictions.

$label: This flag indicates to the system a labeled statement follows.

### 3.2.4 Other Control Flags

In addition to the above control flags, I use some other control flags not related to context-sensitive restrictions. They are described here.

Some reserved words contain '<' which must be distinguished from the symbol for nonterminals; therefore, control flags are used to replace those reserved words.

$>: This flag stands for "<>".

$Less: This flag stands for "<".

$=: This flag stands for "<=".

Some control flags just indicate a beginning position of a structure, so the system can know critical positions in the program strings.

$typestart: This flag indicates the position of <type definition part>.

$labelstart: This flag indicates the position of <label declaration part>.

$pfstart: This flag indicates the position of < procedure and function declaration part>.

$constart: This flag indicates the position of <constant definition part>.

Two control flags, $number and $letter, replace the BNF rules for numbers and letters, because the rules are not necessary to include in the input files so that users can more efficiently test their compiler. For example, some important numbers like "0" and "0.0" can not be generated properly from BNF at the proper places in program strings. Users may prefer to arrange suitable numbers by themselves rather than by automatic generation.

The "$L" is used to indicate a new line for readability.

The other system keywords are identified by using '$' as initial character. They will appear in program strings as well as reserved words so that users can recognize them.

### 3.3 Control Flags and Programming Language Specification

There are some grammars that have been designed to incorporate context-sensitive restrictions; but up to now, context-free grammars are usually used in compilers. It is interesting to know the reasons why compiler implementors do not use context-sensitive grammars. One reason is that context-sensitive grammars are more complicated than context-free grammars. But context-free grammars present problems too. Those problems make users prefer to learn a compiler's context-sensitive restrictions from practice in exercising their compilers rather than from the user manuals. Also the user manuals often do not specify the context-sensitive restrictions very clearly.

Control flags can help to solve these problems. The definition of the syntax of a programming language is usually given in two parts. The context-sensitive restrictions are placed on the context-free syntax at certain positions; These positions can be considered as control flags positions. Usually, compiler implementors do not give any information about where and how context-sensitive

restrictions are checked; they prefer to explain their context-sesitive restrictions in the user manual.

The user manuals are often very hard to understand for a novice, sometimes they are very hard to understand for senior programmers also. That is the reason why users often learn a compiler's context-sensitive restrictions from practice, exercising compilers.

Therefore, this thesis renders a suggestion for programming languages: that control flags should be put into language specifications.

Putting control flags into BNF can tell users where and how context-sensitive restrictions are checked by the compiler, that let users know what kind of context-sensitive restrictions are built into the language. A user manual also can explain context-sensitive restrictions easier with the aid of control flags.

For compiler implementors, control flags not only can help them to maintain the compiler easier but also can save labor when implementing a compiler.

If control flags were become part of a programming language definition, then compiler implementors don't need spend time to figure out where context-sensitive restrictions should be checked and what kind of context-sensitive restrictions should be built into the context-free syntax; those can be very clearly understood.

# 4. Verification and Validation

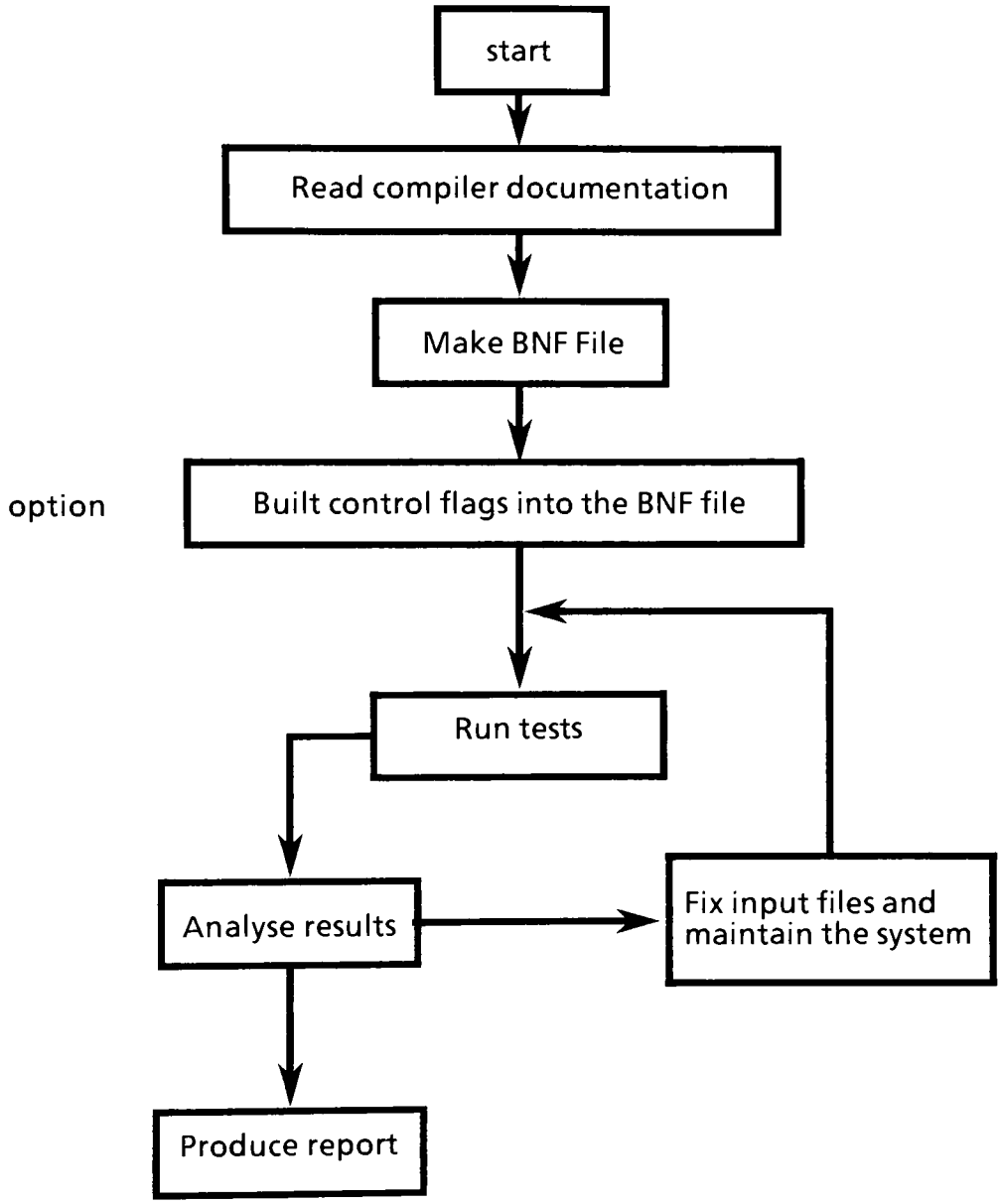
## 4.1 Test Plan

In order to test this compiler testing system, I chose Prosmall, C, Pascal, and Ada's BNF as input files to test the sentence generator algorithms. Then, I added the control flags into Pascal's BNF to generate compilable programs.

Because implementing functions for the control flags is very complicated, I only implemented functions for the context-sensitive restrictions mentioned in Section 3.1, and 3.2. When the output was correct for one restriction, I added the next restriction until all program strings were compilable.

## 4.2 Test and Development Procedure

The following flow chart shows my procedure of test and development.



## 4.3 Test Result

This test was executed using the Pyramid 90X. The results are discussed below.

### 4.3.1 The Outputs from Purdom's Algorithm

In this thesis, Purdom's algorithm can accept an entered number to change the number of program strings from a input file. The entered number should be larger than 0. If the entered number is 1, then it means that every program string uses at least one brand new production rule; if the entered number is 2, then it means that every program string use at least two brand new production rules, and so on. Therefore, if the entered number is 1, then the system generates the maximum set of program strings. Because all languages' BNF, in this thesis, have fewer than 1000 rules, if the entered number is 1000, then the system generates the minimum set of program strings.

In using syntax-based testing, some production rules must be applied together, that is why this system can apply more new production rules than the limit that was entered. But this system will apply exactly the number of new production rule(s) as the entered number if it is possible.

In the output group (*.agpr), some rule numbers are negative, which means that the rules are applied the first time, so that a user can pay attention to those newly applied production rules.

Table 1 is a statistical table for the outputs of Purdom's algorithm.

| entered # | Pascal | C | Ada | Prosmall |
|---|---|---|---|---|
| 1 | 31 | 29 | 110 | 7 |
| 2 | 25 | 22 | 82 | 5 |
| 3 | 23 | 18 | 67 | 5 |
| 4 | 20 | 16 | 59 | 4 |
| 5 | 17 | 15 | 53 | 4 |
| 6 | 15 | 14 | 47 | 4 |
| 7 | 15 | 14 | 41 | 4 |
| 20 | 8 | 9 | 21 | 3 |
| 50 | 5 | 7 | 13 | 3 |
| 1000 | 2 | 6 | 6 | 3 |

# Table 1

The numbers of the program strings generated by Purdom's algorithm with different entered numbers and languages.

4.3.2 The Outputs From The Second Algorithm

Table 2 is a statistical table for the outputs of the second algorithm.

| | Pascal | C | Ada | Prosmall |
|---|---|---|---|---|
| number of program strings | 99 | 137 | 337 | 36 |

Table 2

The numbers of their program strings generated by the second algorithm with different languages.

4.3.3 The Compilable Program Strings of Pascal

The compilable program strings of Pascal generated from the second algorithm are listed in Appendix 2, Part 3. All of these compilable program strings have been compiled. The compiling procedure for every program string is:

1. change program file name from <file>.ags to <file>.p

   (Because the Pascal compiler can compile only a program named with postfix ".p")

2. pascal <file>.p

3. check whether or not there is compiler error.

4.3.4 Discussion

From table 1 and table 2, we know that the second algorithm can generate more program strings than Purdom's algorithm. Hence the average length of program strings from the second algorithm is shorter than from Purdom's algorithm. See Table 3. Therefore, I suggest that the second algorithm is suitable for initial compiler testing. Because a compiler may have a lot of bugs at the beginning, the simpler the test data is, the easier it is to know where the bugs are.

| From | Pascal | C | Ada | Prosmall |
|---|---|---|---|---|
| Purdom's algorithm | 266 | 112 | 334 | 149 |
| The second algorithm | 165 | 38 | 131 | 70 |

# Table 3

The average lengths from Purdom's and the second algorithm. The entered number is 1 for Purdom's algorithm.

Average length = Total characters / number of program strings.

From Table 1, we know that for all languages, the number of program strings can be less than seven; even a very big language likes Ada generated only six program strings.

Therefore, we can use this characteristic to generate random testing data very easily, and the set of program strings will not be too large. Because the system now maintains an array in which element values are initialized as 1 to trace how many times a production rule was already used. If a production rule was used once, then the corresponding element value is decreased by one. If a element value is less than 1, then its corresponding rule will not be used if this rule is not necessary; so a random number generator can assign initial random numbers into every element to generate random testing data. If a user doesn't want to generate a lot of program strings with a seed number, then he can enter a very big number for the entered number to decrease the quantity of program strings. Because the big entered number makes the system generate a few program strings, those generated strings can be very complicated and unpredictable.

# 5. Experiences and Conclusions

In this thesis, I use some complicated recursive functions to generated program strings. Those recursive functions can produce a very long and complicated output. Therefore, the task of debugging and tracing become very difficult.

sometimes the program may have already executed for several hours, but it may still be hard to say whether or not the functions are executing perfectly. Therefore, I chose a very small language (Prosmall which was used in a class on compiler construction) to perform the initial test. At first I derived a result by hand from Prosmall's BNF and then compared the result with the output from the system. After the program passed the initial test, C, Pascal and Ada were used to test the program.

At the beginning I paid most attention to context-sensitive restrictions, and tried to use only Purdom's algorithm to generate program strings. Later I found that there should be some variety for a sentence generator.

It is impossible to generate all possible combinations from a BNF of programming language, because there are a lot of infinite loops in a BNF of programming language.

Up to now, sentence generation is not a hot topic in software testing, but it is very important in syntax-based testing because different algorithms can generate different sets of test data. Different sets of test data can be used to test software in different ways to find bugs. Therefore, additional research for sentence generator algorithms is to be encouraged.

As we know, context-free grammars are not sufficient to represent computer languages, so many methods have been rendered, but there is no single method that is widely accepted. It is hard to predict when there will be a widely accepted way to deal with context-sensitive restrictions in automatic tools for compiler testing. Therefore to derive a new and better method to represent context-sensitive restrictions in compiler testing is a challenge field of research.

This thesis uses the method of control flags to deal with context-sensitive restrictions. The control flags are oriented to sentence generation, but the method has the potential to help in the definition of programming languages. Therefore, additional research in the use of control flags in programming language definition is suggested.

# 6. Bibliography

[1] K. V. Handford, "Automatic Generation of Test Cases", IBM Systems Journal 9(4), pp. 242-257, 1970.

[2] P. Purdom, "A Sentence Generator for Testing Parsers", BIT, Vol. 12, pp. 366-375, July 1972.

[3] A. J. Payne, "A Formalized Technique for Expressing Compiler Exercisers", SIGPLAN Notices 13(1), pp. 59-69, 1978.

[4] A. Celentano, S. Crespi Reghizzi, P. Della Vigna, C, Ghezzi, G. Granta and F. Savoretti, "Compiler Testing Using a Sentence Generator", Software-Practice and Experience 10(11), pp. 897-918, 1980.

[5] Franco Bazzichi and Ippolito Spadafora, "An Automatic Generator for Compiler Testing", IEEE Transactions on Software Engineering, Vol. SE-8, No. 4, pp. 343-353, July 1982.

[6] Jean-Paul Tremblay and Paul G. Sorenson, "The Theory and Practice of Compiler Writing", 1982.

[7] Brian W. Kernighn and Dennis M. Ritchie, "The C Programming Language", Bell Laboratories Murray Hill, New Jersey, 1978.

[8] D. C. Ince, "The Automatic Generation of Test Data", The Computer Journal, Vol. 30 No. 1, 1987.

[9] Glenford J. Myers, "The Art of Software Testing", 1979.

[10] Michale Marcotty, Henry F. Ledgard and Gregor V. Bochmann, "A Sampler of Formal Definitions", ACM Computing Surveys, Vol. 8 Number 2, pp. 191-276, June 1976.

[11] Wolfgang Polk, "Compiler Specification and Verification", Lecture Notes in Computer Science, 1981.

[12] Brain A. Wichmann, Z. J. Ciechanowicz, "Pascal Compiler Validation", John Wiley & Sons, 1983.

# Appendix 1: The Input and Outputs of Prosmall

Part 1: the Prosmall's BNF
Part 2: the file of "rule.agr"
Part 3: the files of "*.agp"; the entered number is 1.
Part 4: the files of "*.agpr"
Part 5: the files of "*.ags"
Part 6: the files of "*.agsr"

# Appendix 1

# Part 1: the Prosmall's BNF

Prosmall is a subset of combination of both Pascal and C. Prosmall was used in the Spring 1987 class of compiler construction in RIT Graduate Computer Science Department.

```
<start> : <glbdecl>  <procdec>  main() $L  <locdecl> begin $L
          <stmlst> end $L #
<iter> : while ( <bexp> ) $L loop $L <stmlst> pool  #
<glbdecl> :#
<glbdecl> :  global <declst> $L #
<procdec> :#
<procdec> :  <proclst> $L #
<proclst> : <procdef>#
<proclst> : <proclst> <procdef>#
<procdef> : proc $ID <locdecl> $L  begin $L  <stmlst> end $L #
<procdef> : proc $ID ( <declst>  ) <locdecl> $L  begin $L
          <stmlst> end $L #
<locdecl> : #
<locdecl>  : decl <declst> #
<declst>  : <decln> #
<declst>  : <declst> <decln> #
<decln> : <typ> <varlst> ;#
<typ> : int#
<typ> : char#
<varlst>  : <vardecl>#
<varlst> : <varlst> ,  <vardecl>#
<vardecl> : $ID#
<vardecl> : array $ID [ $NUM ]#
<stmlst> : <stmt>#
<stmlst> : <stmlst> <stmt>#
<stmt> : <io> ; $L  #
<stmt> : <asgn> ; $L #
<stmt> : <procall> ; $L #
<stmt> : <condl> ; $L #
<stmt> : <iter> ; $L #
<procall> : $ID ( <parlst> )#
<procall> : $ID ()#
<parlst> : $ID#
<parlst> : <parlst> , $ID#
<io> : read ( <var> )#
<io> : write( <exp> )#
<asgn> : <var> = <exp>#
<condl> : if ( <bexp> ) $L  then $L  <stmlst> fi #
<condl> : if ( <bexp> ) $L then $L  <stmlst> $L  else $L
        <stmlst> $L fi #
<bexp> : $NUM #
<bexp> : <bexp> or <bexp>#
<bexp> : not <bexp>#
<bexp> : <exp> == <exp>#
<bexp> : <exp> >= <exp>#
<bexp> : <exp> > <exp>#
<bexp> : <exp> <= <exp>#
<bexp> : <exp> < <exp>#
<bexp> : <bexp> and <bexp>#
<exp> : <term>#
<exp> : <exp> + <term>#
<exp> : <exp> - <term>#
<term> : <factor>#
```

<term> : <term> * <factor>#
<term> : <term> / <factor> #
<term> : <term> % <factor>#
<factor> : $NUM#
<factor> : <var>#
<factor> : $CH#
<factor> : ( <exp> )#
<factor> : - <factor>#
<var> : $ID#
<var> : $ID [ <exp> ]#

# Appendix 1

# Part 2: the file of "rule.agr"

This file was produced form the input file (Prosmall's BNF). This file contains a unique index number for every context-free grammar rule in the input file. A user can find a context-free grammar rule easily by using its index number.

0) <start> : <glbdecl> <procdec> main() $L <locdecl> begin $L
<stmlst> end $L #
1) <iter> : while ( <bexp> ) $L loop $L <stmlst> pool #
2) <glbdecl> :#
3) <glbdecl> : global <declst> $L #
4) <procdec> :#
5) <procdec> : <proclst> $L #
6) <proclst> : <procdef>#
7) <proclst> : <proclst> <procdef>#
8) <procdef> : proc $ID <locdecl> $L begin $L <stmlst> end $L #
9) <procdef> : proc $ID ( <declst> ) <locdecl> $L begin $L
<stmlst> end $L #
10) <locdecl> : #
11) <locdecl> : decl <declst> #
12) <declst> : <decln> #
13) <declst> : <declst> <decln> #
14) <decln> : <typ> <varlst> ;#
15) <typ> : int#
16) <typ> : char#
17) <varlst> : <vardecl>#
18) <varlst> : <varlst> , <vardecl>#
19) <vardecl> : $ID#
20) <vardecl> : array $ID [ $NUM ]#
21) <stmlst> : <stmt>#
22) <stmlst> : <stmlst> <stmt>#
23) <stmt> : <io> ; $L #
24) <stmt> : <asgn> ; $L #
25) <stmt> : <procall> ; $L #
26) <stmt> : <condl> ; $L #
27) <stmt> : <iter> ; $L #
28) <procall> : $ID ( <parlst> )#
29) <procall> : $ID ()#
30) <parlst> : $ID#
31) <parlst> : <parlst> , $ID#
32) <io> : read ( <var> )#
33) <io> : write( <exp> )#
34) <asgn> : <var> = <exp>#
35) <condl> : if ( <bexp> ) $L then $L <stmlst> fi #
36) <condl> : if ( <bexp> ) $L then $L <stmlst> $L else $L
<stmlst> $L fi #
37) <bexp> : $NUM #
38) <bexp> : <bexp> or <bexp>#
39) <bexp> : not <bexp>#
40) <bexp> : <exp> == <exp>#
41) <bexp> : <exp> >= <exp>#
42) <bexp> : <exp> > <exp>#
43) <bexp> : <exp> <= <exp>#
44) <bexp> : <exp> < <exp>#
45) <bexp> : <bexp> and <bexp>#
46) <exp> : <term>#
47) <exp> : <exp> + <term>#
48) <exp> : <exp> - <term>#
49) <term> : <factor>#

50) <term> : <term> * <factor>#
51) <term> : <term> / <factor> #
52) <term> : <term> % <factor>#
53) <factor> : $NUM#
54) <factor> : <var>#
55) <factor> : $CH#
56) <factor> : ( <exp> )#
57) <factor> : - <factor>#
58) <var> : $ID#
59) <var> : $ID [ <exp> ]#

# Appendix 1

# Part 3: the files of "*.agp"; the entered number is 1

This is a group of output files. They are produced by the Purdom's algorithm.

```
::::::::::::::
0.agp
::::::::::::::
main()
begin
read ( $ID ) ;
end
::::::::::::::
1.agp
::::::::::::::
global int $ID ;
proc $ID decl char array $ID [ $NUM ] , $ID ; int $ID ;
begin
$ID [ $NUM ] = $ID % $CH / ( - $NUM ) * $NUM - $NUM
      + $NUM ;
$ID ( $ID ) ;
end
main()
begin
if ( $NUM )
then
while ( not $NUM == $NUM or $NUM >= $NUM )
loop
write( $NUM ) ;
pool ;
fi ;
end
::::::::::::::
2.agp
::::::::::::::
proc $ID ( int $ID ; )
begin
read ( $ID ) ;
end
proc $ID
begin
read ( $ID ) ;
end
main()
begin
read ( $ID ) ;
end
::::::::::::::
3.agp
::::::::::::::
main()
begin
while ( $NUM > $NUM )
loop
$ID () ;
pool ;
read ( $ID ) ;
end
```

::::::::::::::
4.agp
::::::::::::::
```
main()
begin
$ID ( $ID , $ID ) ;
end
```
::::::::::::::
5.agp
::::::::::::::
```
main()
begin
if ( $NUM <= $NUM )
then
read ( $ID ) ;
else
read ( $ID ) ;
fi ;
read ( $ID ) ;
end
```
::::::::::::::
6.agp
::::::::::::::
```
main()
begin
if ( $NUM < $NUM )
then
if ( $NUM and $NUM )
then
read ( $ID ) ;
else
read ( $ID ) ;
fi ;
fi ;
read ( $ID ) ;
end
```

# Appendix 1

# Part 4: the files of "*.agpr"

A file in this group contains all the context-free rules that are used to produce a corresponding *.agp file. For example, the rules file "1.agpr" corresponds to produced program file "1.agp". In this group some rule numbers are negative, which means that the rules are applied the first time, so that a user can pay attention to those newly applied production rules.

```
::::::::::::::
0.agpr
::::::::::::::
**************** applied production rules *****************
0)      <start> : <glbdecl>  <procdec>  main() $L  <locdecl> begin $L
          <stmlst> end $L #
-2)     <glbdecl> :#
-4)     <procdec> :#
-10)    <locdecl> : #
-21)    <stmlst> : <stmt>#
-23)    <stmt> : <io> ; $L  #
-32)    <io> : read ( <var> )#
-58)    <var> : $ID#
::::::::::::::
1.agpr
::::::::::::::
**************** applied production rules *****************
0)      <start> : <glbdecl>  <procdec>  main() $L  <locdecl> begin $L
          <stmlst> end $L #
-3)     <glbdecl> :  global <declst> $L #
-12)    <declst>  : <decln> #
-14)    <decln> : <typ> <varlst> ;#
-15)    <typ> : int#
-17)    <varlst>  : <vardecl>#
-19)    <vardecl> : $ID#
-5)     <procdec> :  <proclst> $L #
-6)     <proclst> : <procdef>#
-8)     <procdef> : proc $ID <locdecl> $L  begin $L  <stmlst> end $L #
-11)    <locdecl>  : decl <declst> #
-13)    <declst>  : <declst> <decln> #
12)     <declst>  : <decln> #
14)     <decln> : <typ> <varlst> ;#
-16)    <typ> : char#
-18)    <varlst> : <varlst> ,  <vardecl>#
17)     <varlst>  : <vardecl>#
-20)    <vardecl> : array $ID [ $NUM ]#
19)     <vardecl> : $ID#
14)     <decln> : <typ> <varlst> ;#
15)     <typ> : int#
17)     <varlst>  : <vardecl>#
19)     <vardecl> : $ID#
-22)    <stmlst> : <stmlst> <stmt>#
21)     <stmlst> : <stmt>#
-24)    <stmt> : <asgn> ; $L #
-34)    <asgn> : <var> = <exp>#
-59)    <var> :  $ID [ <exp> ]#
-46)    <exp> : <term>#
-49)    <term> : <factor>#
-53)    <factor> : $NUM#
-47)    <exp> : <exp> + <term>#
-48)    <exp> : <exp> - <term>#
46)     <exp> : <term>#
-50)    <term> : <term> * <factor>#
```

-51) <term> : <term> / <factor> #
-52) <term> : <term> % <factor>#
49) <term> : <factor>#
-54) <factor> : <var>#
58) <var> : $ID#
-55) <factor> : $CH#
-56) <factor> : ( <exp> )#
46) <exp> : <term>#
49) <term> : <factor>#
-57) <factor> : - <factor>#
53) <factor> : $NUM#
53) <factor> : $NUM#
49) <term> : <factor>#
53) <factor> : $NUM#
49) <term> : <factor>#
53) <factor> : $NUM#
-25) <stmt> : <procall> ; $L #
-28) <procall> : $ID ( <parlst> )#
-30) <parlst> : $ID#
10) <locdecl> : #
21) <stmlst> : <stmt>#
-26) <stmt> : <condl> ; $L #
-35) <condl> : if ( <bexp> ) $L then $L <stmlst> fi #
-37) <bexp> : $NUM #
21) <stmlst> : <stmt>#
-27) <stmt> : <iter> ; $L #
-1) <iter> : while ( <bexp> ) $L loop $L <stmlst> pool #
-38) <bexp> : <bexp> or <bexp>#
-39) <bexp> : not <bexp>#
-40) <bexp> : <exp> == <exp>#
46) <exp> : <term>#
49) <term> : <factor>#
53) <factor> : $NUM#
46) <exp> : <term>#
49) <term> : <factor>#
53) <factor> : $NUM#
-41) <bexp> : <exp> >= <exp>#
46) <exp> : <term>#
49) <term> : <factor>#
53) <factor> : $NUM#
46) <exp> : <term>#
49) <term> : <factor>#
53) <factor> : $NUM#
21) <stmlst> : <stmt>#
23) <stmt> : <io> ; $L #
-33) <io> : write( <exp> )#
46) <exp> : <term>#
49) <term> : <factor>#
53) <factor> : $NUM#
::::::::::::::
2.agpr
::::::::::::::
********** applied production rules **********

```
0)      <start> : <glbdecl>  <procdec>  main() $L  <locdecl> begin $L
          <stmlst> end $L #
2)      <glbdecl> :#
5)      <procdec> :  <proclst> $L #
-7)     <proclst> : <proclst> <procdef>#
6)      <proclst> : <procdef>#
-9)     <procdef> : proc $ID ( <declst>  ) <locdecl> $L  begin $L
         <stmlst> end $L #
12)     <declst>  : <decln> #
14)     <decln> : <typ> <varlst> ;#
15)     <typ> : int#
17)     <varlst>  : <vardecl>#
19)     <vardecl> : $ID#
10)     <locdecl> : #
21)     <stmlst> : <stmt>#
23)     <stmt> : <io> ; $L  #
32)     <io> : read ( <var> )#
58)     <var> : $ID#
8)      <procdef> : proc $ID <locdecl> $L  begin $L  <stmlst> end $L #
10)     <locdecl> : #
21)     <stmlst> : <stmt>#
23)     <stmt> : <io> ; $L  #
32)     <io> : read ( <var> )#
58)     <var> : $ID#
10)     <locdecl> : #
21)     <stmlst> : <stmt>#
23)     <stmt> : <io> ; $L  #
32)     <io> : read ( <var> )#
58)     <var> : $ID#
::::::::::::::
::::::::::::::
3.agpr
::::::::::::::
**************** applied production rules *****************
0)      <start> : <glbdecl>  <procdec>  main() $L  <locdecl> begin $L
          <stmlst> end $L #
2)      <glbdecl> :#
4)      <procdec> :#
10)     <locdecl> : #
22)     <stmlst> : <stmlst> <stmt>#
21)     <stmlst> : <stmt>#
27)     <stmt> : <iter> ; $L #
1)      <iter> : while ( <bexp> ) $L loop $L <stmlst> pool  #
-42)    <bexp> : <exp> > <exp>#
46)     <exp> : <term>#
49)     <term> : <factor>#
53)     <factor> : $NUM#
46)     <exp> : <term>#
49)     <term> : <factor>#
53)     <factor> : $NUM#
21)     <stmlst> : <stmt>#
25)     <stmt> : <procall> ; $L #
-29)    <procall> : $ID ()#
23)     <stmt> : <io> ; $L  #
```

```
32)     <io> : read ( <var> )#
58)     <var> : $ID#
::::::::::::::
4.agpr
::::::::::::::
**************** applied production rules *****************
0)      <start> : <glbdecl>  <procdec>  main() $L  <locdecl> begin $L
          <stmlst> end $L #
2)      <glbdecl> :#
4)      <procdec> :#
10)     <locdecl> : #
21)     <stmlst> : <stmt>#
25)     <stmt> : <procall> ; $L #
28)     <procall> : $ID ( <parlst> )#
-31)    <parlst> : <parlst> , $ID#
30)     <parlst> : $ID#
::::::::::::::
5.agpr
::::::::::::::
**************** applied production rules *****************
0)      <start> : <glbdecl>  <procdec>  main() $L  <locdecl> begin $L
          <stmlst> end $L #
2)      <glbdecl> :#
4)      <procdec> :#
10)     <locdecl> : #
22)     <stmlst> : <stmlst> <stmt>#
21)     <stmlst> : <stmt>#
26)     <stmt> : <condl> ; $L #
-36)    <condl> : if ( <bexp> ) $L then $L  <stmlst> $L  else $L
        <stmlst> $L fi #
-43)    <bexp> : <exp> <= <exp>#
46)     <exp> : <term>#
49)     <term> : <factor>#
53)     <factor> : $NUM#
46)     <exp> : <term>#
49)     <term> : <factor>#
53)     <factor> : $NUM#
21)     <stmlst> : <stmt>#
23)     <stmt> : <io> ; $L  #
32)     <io> : read ( <var> )#
58)     <var> : $ID#
21)     <stmlst> : <stmt>#
23)     <stmt> : <io> ; $L  #
32)     <io> : read ( <var> )#
58)     <var> : $ID#
23)     <stmt> : <io> ; $L  #
32)     <io> : read ( <var> )#
58)     <var> : $ID#
::::::::::::::
6.agpr
::::::::::::::
**************** applied production rules *****************
0)      <start> : <glbdecl>  <procdec>  main() $L  <locdecl> begin $L
```

```
          <stmlst> end $L #
2)        <glbdecl> :#
4)        <procdec> :#
10)       <locdecl> : #
22)       <stmlst> : <stmlst> <stmt>#
21)       <stmlst> : <stmt>#
26)       <stmt> : <condl> ; $L #
35)       <condl> : if ( <bexp> ) $L  then $L  <stmlst> fi #
-44)      <bexp> : <exp> < <exp>#
46)       <exp> : <term>#
49)       <term> : <factor>#
53)       <factor> : $NUM#
46)       <exp> : <term>#
49)       <term> : <factor>#
53)       <factor> : $NUM#
21)       <stmlst> : <stmt>#
26)       <stmt> : <condl> ; $L #
36)       <condl> : if ( <bexp> ) $L then $L  <stmlst> $L  else $L
          <stmlst> $L fi #
-45)      <bexp> : <bexp> and <bexp>#
37)       <bexp> : $NUM #
37)       <bexp> : $NUM #
21)       <stmlst> : <stmt>#
23)       <stmt> : <io> ; $L  #
32)       <io> : read ( <var> )#
58)       <var> : $ID#
21)       <stmlst> : <stmt>#
23)       <stmt> : <io> ; $L  #
32)       <io> : read ( <var> )#
58)       <var> : $ID#
23)       <stmt> : <io> ; $L  #
32)       <io> : read ( <var> )#
58)       <var> : $ID#
```

# Appendix 1

# Part 5: the files of "*.ags"

This group is produced by the second algorithm which was mentioned in Section 2.1.

```
::::::::::::::
0.ags
::::::::::::::
main()
begin
read ( $ID ) ;
end
::::::::::::::
1.ags
::::::::::::::
main()
begin
read ( $ID [ $NUM ] ) ;
end
::::::::::::::
10.ags
::::::::::::::
main()
begin
write( $NUM + $NUM ) ;
end
::::::::::::::
11.ags
::::::::::::::
main()
begin
write( $NUM - $NUM ) ;
end
::::::::::::::
12.ags
::::::::::::::
main()
begin
$ID = $NUM ;
end
::::::::::::::
13.ags
::::::::::::::
main()
begin
$ID ( $ID ) ;
end
::::::::::::::
14.ags
::::::::::::::
main()
begin
$ID ( $ID , $ID ) ;
end
::::::::::::::
15.ags
::::::::::::::
main()
```

```
begin
$ID () ;
end
:::::::::::::::
16.ags
:::::::::::::::
main()
begin
if ( $NUM )
then
read ( $ID ) ;
fi ;
end
:::::::::::::::
17.ags
:::::::::::::::
main()
begin
if ( $NUM or $NUM )
then
read ( $ID ) ;
fi ;
end
:::::::::::::::
18.ags
:::::::::::::::
main()
begin
if ( not $NUM )
then
read ( $ID ) ;
fi ;
end
:::::::::::::::
19.ags
:::::::::::::::
main()
begin
if ( $NUM == $NUM )
then
read ( $ID ) ;
fi ;
end
:::::::::::::::
2.ags
:::::::::::::::
main()
begin
write( $NUM ) ;
end
:::::::::::::::
20.ags
:::::::::::::::
```

```
main()
begin
if ( $NUM >= $NUM )
then
read ( $ID ) ;
fi ;
end
```

21.ags

```
main()
begin
if ( $NUM > $NUM )
then
read ( $ID ) ;
fi ;
end
```

22.ags

```
main()
begin
if ( $NUM <= $NUM )
then
read ( $ID ) ;
fi ;
end
```

23.ags

```
main()
begin
if ( $NUM < $NUM )
then
read ( $ID ) ;
fi ;
end
```

24.ags

```
main()
begin
if ( $NUM and $NUM )
then
read ( $ID ) ;
fi ;
end
```

25.ags

```
main()
begin
if ( $NUM )
```

```
then
read ( $ID ) ;
else
read ( $ID ) ;
fi ;
end
```

26.ags

```
main()
begin
while ( $NUM )
loop
read ( $ID ) ;
pool ;
end
```

27.ags

```
main()
begin
read ( $ID ) ;
read ( $ID ) ;
end
```

28.ags

```
main()
decl int $ID ; begin
read ( $ID ) ;
end
```

29.ags

```
proc $ID
begin
read ( $ID ) ;
end
main()
begin
read ( $ID ) ;
end
```

3.ags

```
main()
begin
write( $ID ) ;
end
```

30.ags

```
proc $ID ( int $ID ; )
```

```
 begin
 read ( $ID ) ;
 end
 main()
 begin
 read ( $ID ) ;
 end
:::::::::::::::
31.ags
:::::::::::::::
 proc $ID
 begin
 read ( $ID ) ;
 end
 proc $ID
 begin
 read ( $ID ) ;
 end
 main()
 begin
 read ( $ID ) ;
 end
:::::::::::::::
32.ags
:::::::::::::::
 global int $ID ;
 main()
 begin
 read ( $ID ) ;
 end
:::::::::::::::
33.ags
:::::::::::::::
 global int array $ID [ $NUM ] ;
 main()
 begin
 read ( $ID ) ;
 end
:::::::::::::::
34.ags
:::::::::::::::
 global int $ID , $ID ;
 main()
 begin
 read ( $ID ) ;
 end
:::::::::::::::
35.ags
:::::::::::::::
 global char $ID ;
 main()
 begin
 read ( $ID ) ;
```

```
 end
::::::::::::::
36.ags
::::::::::::::
 global int $ID ; int $ID ;
 main()
 begin
 read ( $ID ) ;
 end
::::::::::::::
4.ags
::::::::::::::
 main()
 begin
 write( $CH ) ;
 end
::::::::::::::
5.ags
::::::::::::::
 main()
 begin
 write( ( $NUM ) ) ;
 end
::::::::::::::
6.ags
::::::::::::::
 main()
 begin
 write( - $NUM ) ;
 end
::::::::::::::
7.ags
::::::::::::::
 main()
 begin
 write( $NUM * $NUM ) ;
 end
::::::::::::::
8.ags
::::::::::::::
 main()
 begin
 write( $NUM / $NUM ) ;
 end
::::::::::::::
9.ags
::::::::::::::
 main()
 begin
 write( $NUM % $NUM ) ;
 end
```

# Appendix 1

# Part 6: the files of "*.agsr"

A file in this group contains all the context-free rules that are used to produce a corresponding "*.ags" file. For example: "3.agsr" corresponds "3.ags".

```
::::::::::::::
0.agsr
::::::::::::::
********** applied production rules **********
0)      <start> : <glbdecl>  <procdec>  main() $L  <locdecl> begin $L
          <stmlst> end $L #
2)      <glbdecl> :#
4)      <procdec> :#
10)     <locdecl> : #
21)     <stmlst> : <stmt>#
23)     <stmt> : <io> ; $L  #
32)     <io> : read ( <var> )#
58)     <var> : $ID#
::::::::::::::
1.agsr
::::::::::::::
********** applied production rules **********
0)      <start> : <glbdecl>  <procdec>  main() $L  <locdecl> begin $L
          <stmlst> end $L #
2)      <glbdecl> :#
4)      <procdec> :#
10)     <locdecl> : #
21)     <stmlst> : <stmt>#
23)     <stmt> : <io> ; $L  #
32)     <io> : read ( <var> )#
59)     <var> :  $ID [ <exp> ]#
46)     <exp> : <term>#
49)     <term> : <factor>#
53)     <factor> : $NUM#
::::::::::::::
10.agsr
::::::::::::::
********** applied production rules **********
0)      <start> : <glbdecl>  <procdec>  main() $L  <locdecl> begin $L
          <stmlst> end $L #
2)      <glbdecl> :#
4)      <procdec> :#
10)     <locdecl> : #
21)     <stmlst> : <stmt>#
23)     <stmt> : <io> ; $L  #
33)     <io> : write( <exp> )#
47)     <exp> : <exp> + <term>#
46)     <exp> : <term>#
49)     <term> : <factor>#
53)     <factor> : $NUM#
49)     <term> : <factor>#
53)     <factor> : $NUM#
::::::::::::::
11.agsr
::::::::::::::
********** applied production rules **********
0)      <start> : <glbdecl>  <procdec>  main() $L  <locdecl> begin $L
          <stmlst> end $L #
```

```
2)      <glbdecl> :#
4)      <procdec> :#
10)     <locdecl> : #
21)     <stmlst> : <stmt>#
23)     <stmt> : <io> ; $L  #
33)     <io> : write( <exp> )#
48)     <exp> : <exp> - <term>#
46)     <exp> : <term>#
49)     <term> : <factor>#
53)     <factor> : $NUM#
49)     <term> : <factor>#
53)     <factor> : $NUM#
::::::::::::::
12.agsr
::::::::::::::
********** applied production rules **********
0)      <start> : <glbdecl>  <procdec>  main() $L  <locdecl> begin $L
          <stmlst> end $L #
2)      <glbdecl> :#
4)      <procdec> :#
10)     <locdecl> : #
21)     <stmlst> : <stmt>#
24)     <stmt> : <asgn> ; $L #
34)     <asgn> : <var> = <exp>#
58)     <var> : $ID#
46)     <exp> : <term>#
49)     <term> : <factor>#
53)     <factor> : $NUM#
::::::::::::::
13.agsr
::::::::::::::
********** applied production rules **********
0)      <start> : <glbdecl>  <procdec>  main() $L  <locdecl> begin $L
          <stmlst> end $L #
2)      <glbdecl> :#
4)      <procdec> :#
10)     <locdecl> : #
21)     <stmlst> : <stmt>#
25)     <stmt> : <procall> ; $L #
28)     <procall> : $ID ( <parlst> )#
30)     <parlst> : $ID#
::::::::::::::
14.agsr
::::::::::::::
********** applied production rules **********
0)      <start> : <glbdecl>  <procdec>  main() $L  <locdecl> begin $L
          <stmlst> end $L #
2)      <glbdecl> :#
4)      <procdec> :#
10)     <locdecl> : #
21)     <stmlst> : <stmt>#
25)     <stmt> : <procall> ; $L #
28)     <procall> : $ID ( <parlst> )#
```

```
31)     <parlst> : <parlst> , $ID#
30)     <parlst> : $ID#
::::::::::::::
::::::::::::::
15.agsr
::::::::::::::
::::::::::::::
********** applied production rules ***********
0)      <start> : <glbdecl>  <procdec>  main() $L  <locdecl> begin $L
          <stmlst> end $L #
2)      <glbdecl> :#
4)      <procdec> :#
10)     <locdecl> : #
21)     <stmlst> : <stmt>#
25)     <stmt> : <procall> ; $L #
29)     <procall> : $ID ()#
::::::::::::::
::::::::::::::
16.agsr
::::::::::::::
::::::::::::::
********** applied production rules ***********
0)      <start> : <glbdecl>  <procdec>  main() $L  <locdecl> begin $L
          <stmlst> end $L #
2)      <glbdecl> :#
4)      <procdec> :#
10)     <locdecl> : #
21)     <stmlst> : <stmt>#
26)     <stmt> : <condl> ; $L #
35)     <condl> : if ( <bexp> ) $L  then $L  <stmlst> fi #
37)     <bexp> : $NUM #
21)     <stmlst> : <stmt>#
23)     <stmt> : <io> ; $L  #
32)     <io> : read ( <var> )#
58)     <var> : $ID#
::::::::::::::
::::::::::::::
17.agsr
::::::::::::::
::::::::::::::
********** applied production rules ***********
0)      <start> : <glbdecl>  <procdec>  main() $L  <locdecl> begin $L
          <stmlst> end $L #
2)      <glbdecl> :#
4)      <procdec> :#
10)     <locdecl> : #
21)     <stmlst> : <stmt>#
26)     <stmt> : <condl> ; $L #
35)     <condl> : if ( <bexp> ) $L  then $L  <stmlst> fi #
38)     <bexp> : <bexp> or <bexp>#
37)     <bexp> : $NUM #
37)     <bexp> : $NUM #
21)     <stmlst> : <stmt>#
23)     <stmt> : <io> ; $L  #
32)     <io> : read ( <var> )#
58)     <var> : $ID#
::::::::::::::
::::::::::::::
18.agsr
::::::::::::::
```

```
********** applied production rules **********
0)      <start> : <glbdecl>  <procdec>  main() $L  <locdecl> begin $L
          <stmlst> end $L #
2)      <glbdecl> :#
4)      <procdec> :#
10)     <locdecl> : #
21)     <stmlst> : <stmt>#
26)     <stmt> : <condl> ; $L #
35)     <condl> : if ( <bexp> ) $L  then $L  <stmlst> fi #
39)     <bexp> : not <bexp>#
37)     <bexp> : $NUM #
21)     <stmlst> : <stmt>#
23)     <stmt> : <io> ; $L  #
32)     <io> : read ( <var> )#
58)     <var> : $ID#
::::::::::::::
19.agsr
::::::::::::::
********** applied production rules **********
0)      <start> : <glbdecl>  <procdec>  main() $L  <locdecl> begin $L
          <stmlst> end $L #
2)      <glbdecl> :#
4)      <procdec> :#
10)     <locdecl> : #
21)     <stmlst> : <stmt>#
26)     <stmt> : <condl> ; $L #
35)     <condl> : if ( <bexp> ) $L  then $L  <stmlst> fi #
40)     <bexp> : <exp> == <exp>#
46)     <exp> : <term>#
49)     <term> : <factor>#
53)     <factor> : $NUM#
46)     <exp> : <term>#
49)     <term> : <factor>#
53)     <factor> : $NUM#
21)     <stmlst> : <stmt>#
23)     <stmt> : <io> ; $L  #
32)     <io> : read ( <var> )#
58)     <var> : $ID#
::::::::::::::
2.agsr
::::::::::::::
********** applied production rules **********
0)      <start> : <glbdecl>  <procdec>  main() $L  <locdecl> begin $L
          <stmlst> end $L #
2)      <glbdecl> :#
4)      <procdec> :#
10)     <locdecl> : #
21)     <stmlst> : <stmt>#
23)     <stmt> : <io> ; $L  #
33)     <io> : write( <exp> )#
46)     <exp> : <term>#
49)     <term> : <factor>#
53)     <factor> : $NUM#
```

```
::::::::::::::
::::::::::::::
20.agsr
::::::::::::::
::::::::::::::
********** applied production rules **********
0)      <start> : <glbdecl>  <procdec>  main() $L  <locdecl> begin $L
          <stmlst> end $L #
2)      <glbdecl> :#
4)      <procdec> :#
10)     <locdecl> : #
21)     <stmlst> : <stmt>#
26)     <stmt> : <condl> ; $L #
35)     <condl> : if ( <bexp> ) $L  then $L  <stmlst> fi #
41)     <bexp> : <exp> >= <exp>#
46)     <exp> : <term>#
49)     <term> : <factor>#
53)     <factor> : $NUM#
46)     <exp> : <term>#
49)     <term> : <factor>#
53)     <factor> : $NUM#
21)     <stmlst> : <stmt>#
23)     <stmt> : <io> ; $L  #
32)     <io> : read ( <var> )#
58)     <var> : $ID#
::::::::::::::
::::::::::::::
21.agsr
::::::::::::::
::::::::::::::
********** applied production rules **********
0)      <start> : <glbdecl>  <procdec>  main() $L  <locdecl> begin $L
          <stmlst> end $L #
2)      <glbdecl> :#
4)      <procdec> :#
10)     <locdecl> : #
21)     <stmlst> : <stmt>#
26)     <stmt> : <condl> ; $L #
35)     <condl> : if ( <bexp> ) $L  then $L  <stmlst> fi #
42)     <bexp> : <exp> > <exp>#
46)     <exp> : <term>#
49)     <term> : <factor>#
53)     <factor> : $NUM#
46)     <exp> : <term>#
49)     <term> : <factor>#
53)     <factor> : $NUM#
21)     <stmlst> : <stmt>#
23)     <stmt> : <io> ; $L  #
32)     <io> : read ( <var> )#
58)     <var> : $ID#
::::::::::::::
::::::::::::::
22.agsr
::::::::::::::
::::::::::::::
********** applied production rules **********
0)      <start> : <glbdecl>  <procdec>  main() $L  <locdecl> begin $L
          <stmlst> end $L #
2)      <glbdecl> :#
```

```
4)      <procdec> :#
10)     <locdecl> : #
21)     <stmlst> : <stmt>#
26)     <stmt> : <condl> ; $L #
35)     <condl> : if ( <bexp> ) $L  then $L  <stmlst> fi #
43)     <bexp> : <exp> <= <exp>#
46)     <exp> : <term>#
49)     <term> : <factor>#
53)     <factor> : $NUM#
46)     <exp> : <term>#
49)     <term> : <factor>#
53)     <factor> : $NUM#
21)     <stmlst> : <stmt>#
23)     <stmt> : <io> ; $L  #
32)     <io> : read ( <var> )#
58)     <var> : $ID#
::::::::::::::
23.agsr
::::::::::::::
********** applied production rules ***********
0)      <start> : <glbdecl>  <procdec>  main() $L  <locdecl> begin $L
          <stmlst> end $L #
2)      <glbdecl> :#
4)      <procdec> :#
10)     <locdecl> : #
21)     <stmlst> : <stmt>#
26)     <stmt> : <condl> ; $L #
35)     <condl> : if ( <bexp> ) $L  then $L  <stmlst> fi #
44)     <bexp> : <exp> < <exp>#
46)     <exp> : <term>#
49)     <term> : <factor>#
53)     <factor> : $NUM#
46)     <exp> : <term>#
49)     <term> : <factor>#
53)     <factor> : $NUM#
21)     <stmlst> : <stmt>#
23)     <stmt> : <io> ; $L  #
32)     <io> : read ( <var> )#
58)     <var> : $ID#
::::::::::::::
24.agsr
::::::::::::::
********** applied production rules ***********
0)      <start> : <glbdecl>  <procdec>  main() $L  <locdecl> begin $L
          <stmlst> end $L #
2)      <glbdecl> :#
4)      <procdec> :#
10)     <locdecl> : #
21)     <stmlst> : <stmt>#
26)     <stmt> : <condl> ; $L #
35)     <condl> : if ( <bexp> ) $L  then $L  <stmlst> fi #
45)     <bexp> : <bexp> and <bexp>#
37)     <bexp> : $NUM #
```

```
37)     <bexp> : $NUM #
21)     <stmlst> : <stmt>#
23)     <stmt> : <io> ; $L  #
32)     <io> : read ( <var> )#
58)     <var> : $ID#
::::::::::::::
::::::::::::::
25.agsr
::::::::::::::
::::::::::::::
********** applied production rules ***********
0)      <start> : <glbdecl>  <procdec>  main() $L  <locdecl> begin $L
          <stmlst> end $L #
2)      <glbdecl> :#
4)      <procdec> :#
10)     <locdecl> : #
21)     <stmlst> : <stmt>#
26)     <stmt> : <condl> ; $L #
36)     <condl> : if ( <bexp> ) $L then $L  <stmlst> $L  else $L
        <stmlst> $L fi #
37)     <bexp> : $NUM #
21)     <stmlst> : <stmt>#
23)     <stmt> : <io> ; $L  #
32)     <io> : read ( <var> )#
58)     <var> : $ID#
21)     <stmlst> : <stmt>#
23)     <stmt> : <io> ; $L  #
32)     <io> : read ( <var> )#
58)     <var> : $ID#
::::::::::::::
::::::::::::::
26.agsr
::::::::::::::
::::::::::::::
********** applied production rules ***********
0)      <start> : <glbdecl>  <procdec>  main() $L  <locdecl> begin $L
          <stmlst> end $L #
2)      <glbdecl> :#
4)      <procdec> :#
10)     <locdecl> : #
21)     <stmlst> : <stmt>#
27)     <stmt> : <iter> ; $L #
1)      <iter> : while ( <bexp> ) $L loop $L <stmlst> pool  #
37)     <bexp> : $NUM #
21)     <stmlst> : <stmt>#
23)     <stmt> : <io> ; $L  #
32)     <io> : read ( <var> )#
58)     <var> : $ID#
::::::::::::::
::::::::::::::
27.agsr
::::::::::::::
::::::::::::::
********** applied production rules ***********
0)      <start> : <glbdecl>  <procdec>  main() $L  <locdecl> begin $L
          <stmlst> end $L #
2)      <glbdecl> :#
4)      <procdec> :#
10)     <locdecl> : #
```

```
22)     <stmlst> : <stmlst> <stmt>#
21)     <stmlst> : <stmt>#
23)     <stmt> : <io> ; $L  #
32)     <io> : read ( <var> )#
58)     <var> : $ID#
23)     <stmt> : <io> ; $L  #
32)     <io> : read ( <var> )#
58)     <var> : $ID#
::::::::::::::
::::::::::::::
28.agsr
::::::::::::::
::::::::::::::
********** applied production rules ***********
0)      <start> : <glbdecl>  <procdec>  main() $L  <locdecl> begin $L
          <stmlst> end $L #
2)      <glbdecl> :#
4)      <procdec> :#
11)     <locdecl>  : decl <declst> #
12)     <declst>  : <decln> #
14)     <decln> : <typ> <varlst> ;#
15)     <typ> : int#
17)     <varlst>  : <vardecl>#
19)     <vardecl> : $ID#
21)     <stmlst> : <stmt>#
23)     <stmt> : <io> ; $L  #
32)     <io> : read ( <var> )#
58)     <var> : $ID#
::::::::::::::
::::::::::::::
29.agsr
::::::::::::::
::::::::::::::
********** applied production rules ***********
0)      <start> : <glbdecl>  <procdec>  main() $L  <locdecl> begin $L
          <stmlst> end $L #
2)      <glbdecl> :#
5)      <procdec> :  <proclst> $L #
6)      <proclst> : <procdef>#
8)      <procdef> : proc $ID <locdecl> $L  begin $L  <stmlst> end $L #
10)     <locdecl> : #
21)     <stmlst> : <stmt>#
23)     <stmt> : <io> ; $L  #
32)     <io> : read ( <var> )#
58)     <var> : $ID#
10)     <locdecl> : #
21)     <stmlst> : <stmt>#
23)     <stmt> : <io> ; $L  #
32)     <io> : read ( <var> )#
58)     <var> : $ID#
::::::::::::::
::::::::::::::
3.agsr
::::::::::::::
::::::::::::::
********** applied production rules ***********
0)      <start> : <glbdecl>  <procdec>  main() $L  <locdecl> begin $L
          <stmlst> end $L #
2)      <glbdecl> :#
```

```
21)     <stmlst> : <stmt>#
23)     <stmt> : <io> ; $L  #
32)     <io> : read ( <var> )#
58)     <var> : $ID#
10)     <locdecl> : #
21)     <stmlst> : <stmt>#
23)     <stmt> : <io> ; $L  #
32)     <io> : read ( <var> )#
58)     <var> : $ID#
::::::::::::::
::::::::::::::
32.agsr
::::::::::::::
********** applied production rules **********
0)      <start> : <glbdecl>  <procdec>  main() $L  <locdecl> begin $L
          <stmlst> end $L #
3)      <glbdecl> :  global <declst> $L #
12)     <declst>  : <decln> #
14)     <decln> : <typ> <varlst> ;#
15)     <typ> : int#
17)     <varlst>  : <vardecl>#
19)     <vardecl> : $ID#
4)      <procdec> :#
10)     <locdecl> : #
21)     <stmlst> : <stmt>#
23)     <stmt> : <io> ; $L  #
32)     <io> : read ( <var> )#
58)     <var> : $ID#
::::::::::::::
::::::::::::::
33.agsr
::::::::::::::
********** applied production rules **********
0)      <start> : <glbdecl>  <procdec>  main() $L  <locdecl> begin $L
          <stmlst> end $L #
3)      <glbdecl> :  global <declst> $L #
12)     <declst>  : <decln> #
14)     <decln> : <typ> <varlst> ;#
15)     <typ> : int#
17)     <varlst>  : <vardecl>#
20)     <vardecl> : array $ID [ $NUM ]#
4)      <procdec> :#
10)     <locdecl> : #
21)     <stmlst> : <stmt>#
23)     <stmt> : <io> ; $L  #
32)     <io> : read ( <var> )#
58)     <var> : $ID#
::::::::::::::
::::::::::::::
34.agsr
::::::::::::::
********** applied production rules **********
0)      <start> : <glbdecl>  <procdec>  main() $L  <locdecl> begin $L
          <stmlst> end $L #
3)      <glbdecl> :  global <declst> $L #
12)     <declst>  : <decln> #
```

```
14)     <decln> : <typ> <varlst> ;#
15)     <typ> : int#
18)     <varlst> : <varlst> ,  <vardecl>#
17)     <varlst>  : <vardecl>#
19)     <vardecl> : $ID#
19)     <vardecl> : $ID#
4)      <procdec> :#
10)     <locdecl> : #
21)     <stmlst> : <stmt>#
23)     <stmt> : <io> ; $L  #
32)     <io> : read ( <var> )#
58)     <var> : $ID#
::::::::::::::
35.agsr
::::::::::::::
********** applied production rules **********
0)      <start> : <glbdecl>  <procdec>  main() $L  <locdecl> begin $L
          <stmlst> end $L #
3)      <glbdecl> :  global <declst> $L #
12)     <declst>  : <decln> #
14)     <decln> : <typ> <varlst> ;#
16)     <typ> : char#
17)     <varlst>  : <vardecl>#
19)     <vardecl> : $ID#
4)      <procdec> :#
10)     <locdecl> : #
21)     <stmlst> : <stmt>#
23)     <stmt> : <io> ; $L  #
32)     <io> : read ( <var> )#
58)     <var> : $ID#
::::::::::::::
36.agsr
::::::::::::::
********** applied production rules **********
0)      <start> : <glbdecl>  <procdec>  main() $L  <locdecl> begin $L
          <stmlst> end $L #
3)      <glbdecl> :  global <declst> $L #
13)     <declst>  : <declst> <decln> #
12)     <declst>  : <decln> #
14)     <decln> : <typ> <varlst> ;#
15)     <typ> : int#
17)     <varlst>  : <vardecl>#
19)     <vardecl> : $ID#
14)     <decln> : <typ> <varlst> ;#
15)     <typ> : int#
17)     <varlst>  : <vardecl>#
19)     <vardecl> : $ID#
4)      <procdec> :#
10)     <locdecl> : #
21)     <stmlst> : <stmt>#
23)     <stmt> : <io> ; $L  #
32)     <io> : read ( <var> )#
58)     <var> : $ID#
```

```
::::::::::::::
4.agsr
::::::::::::::
********** applied production rules **********
0)      <start> : <glbdecl>  <procdec>  main() $L  <locdecl> begin $L
          <stmlst> end $L #
2)      <glbdecl> :#
4)      <procdec> :#
10)     <locdecl> : #
21)     <stmlst> : <stmt>#
23)     <stmt> : <io> ; $L  #
33)     <io> : write( <exp> )#
46)     <exp> : <term>#
49)     <term> : <factor>#
55)     <factor> : $CH#
::::::::::::::
5.agsr
::::::::::::::
********** applied production rules **********
0)      <start> : <glbdecl>  <procdec>  main() $L  <locdecl> begin $L
          <stmlst> end $L #
2)      <glbdecl> :#
4)      <procdec> :#
10)     <locdecl> : #
21)     <stmlst> : <stmt>#
23)     <stmt> : <io> ; $L  #
33)     <io> : write( <exp> )#
46)     <exp> : <term>#
49)     <term> : <factor>#
56)     <factor> : ( <exp> )#
46)     <exp> : <term>#
49)     <term> : <factor>#
53)     <factor> : $NUM#
::::::::::::::
6.agsr
::::::::::::::
********** applied production rules **********
0)      <start> : <glbdecl>  <procdec>  main() $L  <locdecl> begin $L
          <stmlst> end $L #
2)      <glbdecl> :#
4)      <procdec> :#
10)     <locdecl> : #
21)     <stmlst> : <stmt>#
23)     <stmt> : <io> ; $L  #
33)     <io> : write( <exp> )#
46)     <exp> : <term>#
49)     <term> : <factor>#
57)     <factor> : - <factor>#
53)     <factor> : $NUM#
::::::::::::::
7.agsr
::::::::::::::
********** applied production rules **********
```

```
0)      <start> : <glbdecl>  <procdec>  main() $L  <locdecl> begin $L
          <stmlst> end $L #
2)      <glbdecl> :#
4)      <procdec> :#
10)     <locdecl> : #
21)     <stmlst> : <stmt>#
23)     <stmt> : <io> ; $L  #
33)     <io> : write( <exp> )#
46)     <exp> : <term>#
50)     <term> : <term> * <factor>#
49)     <term> : <factor>#
53)     <factor> : $NUM#
53)     <factor> : $NUM#
::::::::::::::
8.agsr
::::::::::::::
********** applied production rules **********
0)      <start> : <glbdecl>  <procdec>  main() $L  <locdecl> begin $L
          <stmlst> end $L #
2)      <glbdecl> :#
4)      <procdec> :#
10)     <locdecl> : #
21)     <stmlst> : <stmt>#
23)     <stmt> : <io> ; $L  #
33)     <io> : write( <exp> )#
46)     <exp> : <term>#
51)     <term> : <term> / <factor> #
49)     <term> : <factor>#
53)     <factor> : $NUM#
53)     <factor> : $NUM#
::::::::::::::
9.agsr
::::::::::::::
********** applied production rules **********
0)      <start> : <glbdecl>  <procdec>  main() $L  <locdecl> begin $L
          <stmlst> end $L #
2)      <glbdecl> :#
4)      <procdec> :#
10)     <locdecl> : #
21)     <stmlst> : <stmt>#
23)     <stmt> : <io> ; $L  #
33)     <io> : write( <exp> )#
46)     <exp> : <term>#
52)     <term> : <term> % <factor>#
49)     <term> : <factor>#
53)     <factor> : $NUM#
53)     <factor> : $NUM#
```

# Appendix 2: Pascal Files

Part 1: Pascal's BNF and control flags
Part 2: the file, "rule.agr"
Part 3: the files, "*.ags"
Part 4: the partial files, "*.agsr"

# Appendix 2

# Part 1: Pascal's BNF and control flags

This file was produced using Pascal's BNF and control flags. This file contains a unique index number for every context-free rule in the input file. A user can find a context-free rule easily by using its index number.

```
<program_heading> : program tester ( input , output ) ; $L
    <block> $$.  $L #
<block> : $labelstart <label_declaration_part> $constart
     <constant_definition_part> $typestart
     <type_definition_part> $varstart <variable_declaration_part>
     $pfstart  <pro_and_fun_declaration_part> <statement_part> #
<label_declaration_part> : #
<label_declaration_part> : label $decllabel <re_label> ; $L #
<re_label> : <label> #
<re_label> : <re_label> , <label> #
<label> : $label_integer #
<constant_definition_part> : #
<constant_definition_part> : const <re_constant_definition> ; $L  #
<re_constant_definition> : <constant_definition> #
<re_constant_definition> : <re_constant_definition> ; $L
                  <constant_definition> #
<constant_definition> : $name = <constant> #
<constant> : <unsigned_number> #
<constant> : <sign> <unsigned_number> #
<constant> : <constant_identifier> #
<constant> : <sign> <constant_identifier> #
<constant> : <string> #
<unsigned_number> : <unsigned_integer> #
<unsigned_number> : <unsigned_real> #
<unsigned_integer> : $number #
<unsigned_real> : <unsigned_integer> $$. $number  #
<unsigned_real> : <unsigned_integer> $$. $number $$E <scale_factor> #
<unsigned_real> : <unsigned_integer> $$E <scale_factor> #
<scale_factor> : <unsigned_integer> #
<scale_factor> : <sign> <unsigned_integer> #
<sign> : $$+ #
<sign> : $$- #
<constant_identifier> : $constantname #
<string> : $$' <re_character> $$' #
<re_character> : $letter #
<re_character> : <re_character> $letter #
<type_definition_part> : #
<type_definition_part> : type $L <re_type_definition> #
<re_type_definition> :  <type_definition> ; $L #
<re_type_definition> : <re_type_definition>  <type_definition> ; $L #
<type_definition> : $name =  <type>  #
<type> : <simple_type> #
<type> : <structured_type> #
<type> : <pointer_type> #
<simple_type> : <type_identifier> #
<simple_type> : <scalar_type> #
<simple_type> : <subrange_type> #
<scalar_type> : ( <re_identifier> ) #
<re_identifier> :  $name #
<re_identifier> : <re_identifier> , $name  #
<subrange_type> : <constant> .. <constant> #
<type_identifier> : integer #
<type_identifier> : real #
```

```
<structured_type> : <unpacked_structured_type> #
<structured_type> : packed <unpacked_structured_type> #
<unpacked_structured_type> : <array_type> #
<unpacked_structured_type> : <record_type> #
<unpacked_structured_type> : <set_type> #
<unpacked_structured_type> : <file_type> #
<array_type> : array [ <re_index_type> ] of <component_type> #
<re_index_type> : <index_type> , <index_type> #
<index_type> : 1 .. 2 #
<component_type> : <type> #
<record_type> : record $L  <field_list>  $L end  #
<field_list> : <fixed_part> #
<field_list> : <fixed_part> ; $L  <variant_part> #
<field_list> : <variant_part> #
<fixed_part> : <re_record_section> #
<re_record_section> : <record_section> #
<re_record_section> : <re_record_section> ; $L  <record_section> #
<record_section> : #
<record_section> : <re_field_identifier> : <type> #
<re_field_identifier> :  <field_identifier> #
<re_field_identifier> : <re_field_identifier> , <field_identifier> #
<variant_part> : case <tag_field> <type_identifier> of <re_variant> #
<re_variant> :  <variant> #
<re_variant> : <re_variant> ; $L  <variant> #
<tag_field> :  #
<tag_field> : <field_identifier> : $L #
<variant> : #
<variant> : <case_label_list> : ( <field_list> ) #
<case_label_list> : <re_case_label> #
<re_case_label> : <case_label> #
<re_case_label> : <re_case_label> , <case_label> #
<case_label> : <constant> #
<set_type> : set of <base_type> #
<base_type> : <simple_type> #
<file_type> : file of <type> #
<pointer_type> : ^ <type_identifier> #
<variable_declaration_part> : var <re_variable_declaration>  ;
    $L #
<variable_declaration_part> : #
<re_variable_declaration> :
    <variable_declaration> #
<re_variable_declaration> : <re_variable_declaration> ; $L
    <variable_declaration> #
<variable_declaration> : <re_identifier> : <type> #
<pro_and_fun_declaration_part> : <re_pro_or_fun_declaration>  #
<re_pro_or_fun_declaration> :  #
<re_pro_or_fun_declaration> : <pro_or_fun_declaration> ; $L  #
<re_pro_or_fun_declaration> : <re_pro_or_fun_declaration>
     <pro_or_fun_declaration> ; $L  #
<pro_or_fun_declaration> : <procedure_declaration> #
<pro_or_fun_declaration> : <function_declaration> #
<procedure_declaration> : <procedure_heading> <block> #
<procedure_heading> : procedure   $name ; $L  #
```

```
<procedure_heading> : procedure  $name  (
   <re_formal_parameter_section> ) ; $L  #
<re_formal_parameter_section> : <formal_parameter_section> #
<re_formal_parameter_section> : <re_formal_parameter_section> ;
         <formal_parameter_section> #
<formal_parameter_section> : <parameter_group> #
<formal_parameter_section> : var <parameter_group> #
<formal_parameter_section> : function  <parameter_group>   #
<formal_parameter_section> : procedure <re_identifier>  #
<parameter_group> : <re_identifier> : <type_identifier> #
<function_declaration> : <function_heading> <block> #
<function_heading> : function  $funname $function  :
          <result_type> ; $L  #
<function_heading> : function  $funname $function (
   <re_formal_parameter_section> ) :  <result_type> ; $L  #
<result_type> : <type_identifier> #
<statement_part> : <compound_statement> #
<statement> : <unlabelled_statement> #
<statement> : $label <label> : <unlabelled_statement> #
<unlabelled_statement> : <simple_statement> #
<unlabelled_statement> : <structured_statement> #
<simple_statement> : <assignment_statement> #
<simple_statement> : #
<simple_statement> : $assg  <procedure_statement> $assgend  #
<simple_statement> : <go_to_statement> #
<assignment_statement> : $assg <variable> :=  <expression> $assgend #
<variable> : <entire_variable> #
<variable> : <referenced_variable> #
<variable> : <component_variable> #
<entire_variable> : <variable_identifier> #
<variable_identifier> : $name #
<component_variable> : <indexed_variable> #
<component_variable> : <field_designator> #
<component_variable> : <file_buffer> #
<indexed_variable> : $array <array_variable> [ <re_expression> ]
                $arrayend  #
<re_expression> : 2  #
<re_expression> : 2 , 2 $array  #
<array_variable> : <variable> #
<field_designator> : <record_variable> . $checkfield <field_identifier>  #
<record_variable> : $record_variable #
<field_identifier> : $name #
<file_buffer> : <file_variable> ^ #
<file_variable> : $file_variable #
<referenced_variable> : <pointer_variable> ^ $pointer #
<pointer_variable> : <variable> #
<expression> : <simple_expression> #
<expression> : <simple_expression> <relational_operator>
     <simple_expression> #
<relational_operator> : $boolean =  #
<relational_operator> : $boolean $> #
<relational_operator> :  $boolean $ #
<relational_operator> : $boolean $Less= #
```

```
<relational_operator> : $boolean >= #
<relational_operator> : $boolean > #
<relational_operator> :  in  $in  #
<simple_expression> : <term> #
<simple_expression> : <sign> <term> #
<simple_expression> : <simple_expression> <adding_operator> <term> #
<adding_operator> : + #
<adding_operator> : or #
<term> : <factor> #
<term> : <term> <multiplying_operator> <factor> #
<multiplying_operator> : * #
<multiplying_operator> : / $real  #
<multiplying_operator> : div #
<multiplying_operator> : mod #
<multiplying_operator> : and #
<factor> : <variable> #
<factor> : <unsigned_constant> #
<factor> : ( <expression> ) #
<factor> : <function_designator> #
<factor> : <set> #
<factor> : not <factor> #
<unsigned_constant> : <unsigned_number> #
<unsigned_constant> : <string> #
<unsigned_constant> : <constant_identifier> #
<unsigned_constant> : nil $pointer   #
<function_designator> : <function_identifier> #
<function_designator> : $paraflag <function_identifier> (
      <re_actural_parameter>   ) $paraend #
<re_actural_parameter> :  <actural_parameter>   #
<re_actural_parameter> : <re_actural_parameter> , <actural_parameter> #
<function_identifier> : $funname #
<set> : [ $element_list <element_list> $element_end ]  #
<element_list> :  #
<element_list> : <re_element> #
<re_element> : <element> #
<re_element> : <re_element> , <element> #
<element> : <expression> #
<element> : <expression> .. <expression> #
<procedure_statement> : <procedure_identifier> #
<procedure_statement> : $paraflag <procedure_identifier> (
       <re_actural_parameter>  ) $paraend  #
<procedure_identifier> : $proname #
<actural_parameter> : <variable> #
<go_to_statement> : goto $goto  <label> ; $L #
<structured_statement> : <compound_statement> #
<structured_statement> : <conditional_statement> #
<structured_statement> : <repetitive_statement> #
<structured_statement> : <with_statement> #
<compound_statement> : begin $L <re_statement> $L $checkfun  end   #
<re_statement> :  <statement> #
<re_statement> : <re_statement> ; $L  <statement> #
<conditional_statement> : <if_statement> #
<conditional_statement> : <case_statement> #
```

```
<if_statement> : if $assg $condition <con_expression> $assgend
            $conditend then $L <statement> #
<if_statement> : if $assg $condition <con_expression> $assgend $conditend
      then $L <statement> else <statement>  #
<case_statement> : case $assg $name $assgned of $L
         0: ; $L <re_case_list_element> $L end   #
<re_case_list_element> : <case_list_element>  #
<re_case_list_element> : <re_case_list_element> ; $L  <case_list_element> #
<case_list_element> : #
<case_list_element> : <case_label_list> : <statement> #
<repetitive_statement> : <while_statement> #
<repetitive_statement> : <repeat_statement> #
<repetitive_statement> : <for_statement> #
<while_statement> : while $assg $condition <con_expression> $assgend
    $conditend do $L <statement> #
<repeat_statement> : repeat $L <re_statement> $L until $assg $condition
         <con_expression> $assgend $conditend #
<con_expression> : $name #
<con_expression> : $name <relational_operator> $name #
<for_statement> : for $assg <control_variable> := <for_list> $assgend
    do $L <statement> #
<for_list> : <initial_value> to <final_value> #
<for_list> : <initial_value> downto <final_value> #
<control_variable> : $name #
<initial_value> : <expression> #
<final_value> : <expression> #
<with_statement> : with $with <record_variable_list> $withend
      do $L  <statement> #
<record_variable_list> : <re_record_variable> #
<re_record_variable> : <record_variable> #
<re_record_variable> : <re_record_variable> , <record_variable> #
```

# Appendix 2

## Part 2: the file, "rule.agr"

This file was produced using Pascal's BNF and control flags. This file contains a unique index number for every context-free rule in the input file. A user can find a context-free rule easily by using its index number.

```
0) <program_heading> : program tester ( input , output ) ; $L
     <block> $$.  $L #
1) <block> : $labelstart <label_declaration_part> $constart
     <constant_definition_part> $typestart
     <type_definition_part> $varstart <variable_declaration_part>
     $pfstart  <pro_and_fun_declaration_part> <statement_part> #
2) <label_declaration_part> : #
3) <label_declaration_part> : label $decllabel <re_label> ; $L #
4) <re_label> : <label> #
5) <re_label> : <re_label> , <label> #
6) <label> : $label_integer #
7) <constant_definition_part> : #
8) <constant_definition_part> : const <re_constant_definition> ; $L  #
9) <re_constant_definition> : <constant_definition> #
10) <re_constant_definition> : <re_constant_definition> ; $L
                  <constant_definition> #
11) <constant_definition> : $name = <constant> #
12) <constant> : <unsigned_number> #
13) <constant> : <sign> <unsigned_number> #
14) <constant> : <constant_identifier> #
15) <constant> : <sign> <constant_identifier> #
16) <constant> : <string> #
17) <unsigned_number> : <unsigned_integer> #
18) <unsigned_number> : <unsigned_real> #
19) <unsigned_integer> : $number #
20) <unsigned_real> : <unsigned_integer> $$. $number  #
21) <unsigned_real> : <unsigned_integer> $$. $number $$E <scale_factor> #
22) <unsigned_real> : <unsigned_integer> $$E <scale_factor> #
23) <scale_factor> : <unsigned_integer> #
24) <scale_factor> : <sign> <unsigned_integer> #
25) <sign> : $$+ #
26) <sign> : $$- #
27) <constant_identifier> : $constantname #
28) <string> : $$' <re_character> $$' #
29) <re_character> : $letter #
30) <re_character> : <re_character> $letter #
31) <type_definition_part> : #
32) <type_definition_part> : type $L <re_type_definition> #
33) <re_type_definition> :  <type_definition> ; $L #
34) <re_type_definition> : <re_type_definition>  <type_definition> ; $L #
35) <type_definition> : $name =  <type>  #
36) <type> : <simple_type> #
37) <type> : <structured_type> #
38) <type> : <pointer_type> #
39) <simple_type> : <type_identifier> #
40) <simple_type> : <scalar_type> #
41) <simple_type> : <subrange_type> #
42) <scalar_type> : ( <re_identifier> ) #
43) <re_identifier> :  $name #
44) <re_identifier> : <re_identifier> , $name  #
45) <subrange_type> : <constant> .. <constant> #
46) <type_identifier> : integer #
47) <type_identifier> : real #
```

48) <structured_type> : <unpacked_structured_type> #
49) <structured_type> : packed <unpacked_structured_type> #
50) <unpacked_structured_type> : <array_type> #
51) <unpacked_structured_type> : <record_type> #
52) <unpacked_structured_type> : <set_type> #
53) <unpacked_structured_type> : <file_type> #
54) <array_type> : array [ <re_index_type> ] of <component_type> #
55) <re_index_type> : <index_type> , <index_type> #
56) <index_type> : 1 .. 2 #
57) <component_type> : <type> #
58) <record_type> : record $L <field_list> $L end #
59) <field_list> : <fixed_part> #
60) <field_list> : <fixed_part> ; $L <variant_part> #
61) <field_list> : <variant_part> #
62) <fixed_part> : <re_record_section> #
63) <re_record_section> : <record_section> #
64) <re_record_section> : <re_record_section> ; $L <record_section> #
65) <record_section> : #
66) <record_section> : <re_field_identifier> : <type> #
67) <re_field_identifier> : <field_identifier> #
68) <re_field_identifier> : <re_field_identifier> , <field_identifier> #
69) <variant_part> : case <tag_field> <type_identifier> of <re_variant> #
70) <re_variant> : <variant> #
71) <re_variant> : <re_variant> ; $L <variant> #
72) <tag_field> : #
73) <tag_field> : <field_identifier> : $L #
74) <variant> : #
75) <variant> : <case_label_list> : ( <field_list> ) #
76) <case_label_list> : <re_case_label> #
77) <re_case_label> : <case_label> #
78) <re_case_label> : <re_case_label> , <case_label> #
79) <case_label> : <constant> #
80) <set_type> : set of <base_type> #
81) <base_type> : <simple_type> #
82) <file_type> : file of <type> #
83) <pointer_type> : ^ <type_identifier> #
84) <variable_declaration_part> : var <re_variable_declaration> ;
$L #
85) <variable_declaration_part> : #
86) <re_variable_declaration> :
<variable_declaration> #
87) <re_variable_declaration> : <re_variable_declaration> ; $L
<variable_declaration> #
88) <variable_declaration> : <re_identifier> : <type> #
89) <pro_and_fun_declaration_part> : <re_pro_or_fun_declaration> #
90) <re_pro_or_fun_declaration> : #
91) <re_pro_or_fun_declaration> : <pro_or_fun_declaration> ; $L #
92) <re_pro_or_fun_declaration> : <re_pro_or_fun_declaration>
<pro_or_fun_declaration> ; $L #
93) <pro_or_fun_declaration> : <procedure_declaration> #
94) <pro_or_fun_declaration> : <function_declaration> #
95) <procedure_declaration> : <procedure_heading> <block> #
96) <procedure_heading> : procedure $name ; $L #

```
97) <procedure_heading> : procedure  $name  (
   <re_formal_parameter_section> ) ; $L  #
98) <re_formal_parameter_section> : <formal_parameter_section> #
99) <re_formal_parameter_section> : <re_formal_parameter_section> ;
        <formal_parameter_section> #
100) <formal_parameter_section> : <parameter_group> #
101) <formal_parameter_section> : var <parameter_group> #
102) <formal_parameter_section> : function  <parameter_group>   #
103) <formal_parameter_section> : procedure <re_identifier>  #
104) <parameter_group> : <re_identifier> : <type_identifier> #
105) <function_declaration> : <function_heading> <block> #
106) <function_heading> : function  $funname $function  :
         <result_type> ; $L  #
107) <function_heading> : function  $funname $function (
   <re_formal_parameter_section> ) :  <result_type> ; $L  #
108) <result_type> : <type_identifier> #
109) <statement_part> : <compound_statement> #
110) <statement> : <unlabelled_statement> #
111) <statement> : $label <label> : <unlabelled_statement> #
112) <unlabelled_statement> : <simple_statement> #
113) <unlabelled_statement> : <structured_statement> #
114) <simple_statement> : <assignment_statement> #
115) <simple_statement> : #
116) <simple_statement> : $assg  <procedure_statement> $assgend  #
117) <simple_statement> : <go_to_statement> #
118) <assignment_statement> : $assg <variable> :=  <expression> $assgend #
119) <variable> : <entire_variable> #
120) <variable> : <referenced_variable> #
121) <variable> : <component_variable> #
122) <entire_variable> : <variable_identifier> #
123) <variable_identifier> : $name #
124) <component_variable> : <indexed_variable> #
125) <component_variable> : <field_designator> #
126) <component_variable> : <file_buffer> #
127) <indexed_variable> : $array <array_variable> [ <re_expression> ]
                $arrayend  #
128) <re_expression> : 2  #
129) <re_expression> : 2 , 2 $array  #
130) <array_variable> : <variable> #
131) <field_designator> : <record_variable> . $checkfield <field_identifier>  #
132) <record_variable> : $record_variable #
133) <field_identifier> : $name #
134) <file_buffer> : <file_variable> ^ #
135) <file_variable> : $file_variable #
136) <referenced_variable> : <pointer_variable> ^ $pointer #
137) <pointer_variable> : <variable> #
138) <expression> : <simple_expression> #
139) <expression> : <simple_expression> <relational_operator>
    <simple_expression> #
140) <relational_operator> : $boolean =  #
141) <relational_operator> : $boolean $> #
142) <relational_operator> :  $boolean $ #
143) <relational_operator> : $boolean $Less= #
```

144) <relational_operator> : $boolean >= #
145) <relational_operator> : $boolean > #
146) <relational_operator> : in $in #
147) <simple_expression> : <term> #
148) <simple_expression> : <sign> <term> #
149) <simple_expression> : <simple_expression> <adding_operator> <term> #
150) <adding_operator> : + #
151) <adding_operator> : or #
152) <term> : <factor> #
153) <term> : <term> <multiplying_operator> <factor> #
154) <multiplying_operator> : * #
155) <multiplying_operator> : / $real #
156) <multiplying_operator> : div #
157) <multiplying_operator> : mod #
158) <multiplying_operator> : and #
159) <factor> : <variable> #
160) <factor> : <unsigned_constant> #
161) <factor> : ( <expression> ) #
162) <factor> : <function_designator> #
163) <factor> : <set> #
164) <factor> : not <factor> #
165) <unsigned_constant> : <unsigned_number> #
166) <unsigned_constant> : <string> #
167) <unsigned_constant> : <constant_identifier> #
168) <unsigned_constant> : nil $pointer #
169) <function_designator> : <function_identifier> #
170) <function_designator> : $paraflag <function_identifier> (
<re_actural_parameter> ) $paraend #
171) <re_actural_parameter> : <actural_parameter> #
172) <re_actural_parameter> : <re_actural_parameter> , <actural_parameter> #
173) <function_identifier> : $funname #
174) <set> : [ $element_list <element_list> $element_end ] #
175) <element_list> : #
176) <element_list> : <re_element> #
177) <re_element> : <element> #
178) <re_element> : <re_element> , <element> #
179) <element> : <expression> #
180) <element> : <expression> .. <expression> #
181) <procedure_statement> : <procedure_identifier> #
182) <procedure_statement> : $paraflag <procedure_identifier> (
<re_actural_parameter> ) $paraend #
183) <procedure_identifier> : $proname #
184) <actural_parameter> : <variable> #
185) <go_to_statement> : goto $goto <label> ; $L #
186) <structured_statement> : <compound_statement> #
187) <structured_statement> : <conditional_statement> #
188) <structured_statement> : <repetitive_statement> #
189) <structured_statement> : <with_statement> #
190) <compound_statement> : begin $L <re_statement> $L $checkfun end #
191) <re_statement> : <statement> #
192) <re_statement> : <re_statement> ; $L <statement> #
193) <conditional_statement> : <if_statement> #
194) <conditional_statement> : <case_statement> #

```
195) <if_statement> : if $assg $condition <con_expression> $assgend
          $conditend then $L <statement> #
196) <if_statement> : if $assg $condition <con_expression> $assgend $conditend
     then $L <statement> else <statement>  #
197) <case_statement> : case $assg $name $assgned of $L
        0: ; $L <re_case_list_element> $L end   #
198) <re_case_list_element> : <case_list_element>  #
199) <re_case_list_element> : <re_case_list_element> ; $L  <case_list_element> #
200) <case_list_element> : #
201) <case_list_element> : <case_label_list> : <statement> #
202) <repetitive_statement> : <while_statement> #
203) <repetitive_statement> : <repeat_statement> #
204) <repetitive_statement> : <for_statement> #
205) <while_statement> : while $assg $condition <con_expression> $assgend
   $conditend do $L <statement> #
206) <repeat_statement> : repeat $L <re_statement> $L until $assg $condition
        <con_expression> $assgend $conditend #
207) <con_expression> : $name #
208) <con_expression> : $name <relational_operator> $name #
209) <for_statement> : for $assg <control_variable> := <for_list> $assgend
   do $L <statement> #
210) <for_list> : <initial_value> to <final_value> #
211) <for_list> : <initial_value> downto <final_value> #
212) <control_variable> : $name #
213) <initial_value> : <expression> #
214) <final_value> : <expression> #
215) <with_statement> : with $with <record_variable_list> $withend
     do $L  <statement> #
216) <record_variable_list> : <re_record_variable> #
217) <re_record_variable> : <record_variable> #
218) <re_record_variable> : <re_record_variable> , <record_variable> #
```

# Appendix 2

# Part 3: the files, "*.ags"

This group is produced by the second algorithm which was mentioned in Section 2.1.

::::::::::::::
0.ags
::::::::::::::
```
program tester ( input , output ) ;
var seventeen, sixteen: integer;
fifteen : integer ;
begin
sixteen := seventeen
end .
```
::::::::::::::
1.ags
::::::::::::::
```
program tester ( input , output ) ;
var two, one: integer;
eighteen , zero : integer ;
begin
one := two
end .
```
::::::::::::::
10.ags
::::::::::::::
```
program tester ( input , output ) ;
var one, eighteen: integer;
function zero ( one: integer): integer;
begin
zero := 0;
end;
begin
eighteen := zero ( one )
end .
```
::::::::::::::
11.ags
::::::::::::::
```
program tester ( input , output ) ;
type
sett = ( see );
order = set of sett;
var two: order;
begin
two := [ ]
end .
```
::::::::::::::
12.ags
::::::::::::::
```
program tester ( input , output ) ;
type
sett = ( see, four );
order = set of sett;
var three: order;
begin
three := [ four ]
end .
```
::::::::::::::

13.ags

```
program tester ( input , output ) ;
type
sett = ( see, six, seven );
order = set of sett;
var five: order;
begin
five := [ six .. seven ]
end .
```

14.ags

```
program tester ( input , output ) ;
type
sett = ( see, nine, ten );
order = set of sett;
var eight: order;
begin
eight := [ nine , ten ]
end .
```

15.ags

```
program tester ( input , output ) ;
var twelve, eleven: integer;
begin
eleven := not twelve
end .
```

16.ags

```
program tester ( input , output ) ;
var fifteen, fourteen, thirteen: integer;
begin
thirteen := fourteen * fifteen
end .
```

17.ags

```
program tester ( input , output ) ;
var eighteen, seventeen, sixteen: real;
begin
sixteen := seventeen / eighteen
end .
```

18.ags

```
program tester ( input , output ) ;
var two, one, zero: integer;
begin
zero := one div two
end .
```

:::::::::::::::
19.ags
:::::::::::::::
```
program tester ( input , output ) ;
var five, four, three: integer;
begin
three := four mod five
end .
```
:::::::::::::::
2.ags
:::::::::::::::
```
program tester ( input , output ) ;
var six, five: integer;
three : integer ;
four : integer ;
begin
five := six
end .
```
:::::::::::::::
20.ags
:::::::::::::::
```
program tester ( input , output ) ;
var eight, seven, six: integer;
begin
six := seven and eight
end .
```
:::::::::::::::
21.ags
:::::::::::::::
```
program tester ( input , output ) ;
var ten, nine: integer;
begin
nine := + ten
end .
```
:::::::::::::::
22.ags
:::::::::::::::
```
program tester ( input , output ) ;
var thirteen, twelve, eleven: integer;
begin
eleven := twelve + thirteen
end .
```
:::::::::::::::
23.ags
:::::::::::::::
```
program tester ( input , output ) ;
var sixteen, fifteen, fourteen: integer;
begin
fourteen := fifteen or sixteen
end .
```
:::::::::::::::
24.ags
:::::::::::::::

```
program tester ( input , output ) ;
var zero, eighteen, seventeen: boolean;
begin
seventeen := eighteen = zero
end .
```

25.ags

```
program tester ( input , output ) ;
type pointer = ^ integer;
var two: integer;
one: pointer;
begin
one ^ := two
end .
```

26.ags

```
program tester ( input , output ) ;
var four: integer;
three: array[1..2] of integer;
begin
three [ 2 ] := four
end .
```

27.ags

```
program tester ( input , output ) ;
var six: integer;
five: array[1..2, 1..2] of integer;
begin
five [ 2 , 2 ] := six
end .
```

28.ags

```
program tester ( input , output ) ;
type
exam = record
score : integer;
end;
var eight: integer;
seven: exam;
begin
seven . score := eight
end .
```

29.ags

```
program tester ( input , output ) ;
type integerfile = file of integer;
var ten: integer;
nine: integerfile;
```

```
begin
nine ^ := ten
end .
:::::::::::::::
3.ags
:::::::::::::::
program tester ( input , output ) ;
var eight, seven: integer;
begin
seven := eight
end .
:::::::::::::::
30.ags
:::::::::::::::
program tester ( input , output ) ;
begin

end .
:::::::::::::::
31.ags
:::::::::::::::
program tester ( input , output ) ;
procedure eleven ;
begin
end;
begin
eleven
end .
:::::::::::::::
32.ags
:::::::::::::::
program tester ( input , output ) ;
var thirteen: integer;
procedure twelve ( thirteen :integer ) ;
begin
end;
begin
twelve ( thirteen )
end .
:::::::::::::::
33.ags
:::::::::::::::
program tester ( input , output ) ;
var sixteen, fifteen: integer;
procedure fourteen ( sixteen , fifteen :integer ) ;
begin
end;
begin
fourteen ( fifteen , sixteen )
end .
:::::::::::::::
34.ags
:::::::::::::::
```

```
program tester ( input , output ) ;
label 91;
begin
goto 91;
91:
;

end .
```

35.ags

```
program tester ( input , output ) ;
var eighteen, seventeen: integer;
begin
begin
seventeen := eighteen
end
end .
```

36.ags

```
program tester ( input , output ) ;
var two, one: integer;
zero: boolean;
begin
if zero then
one := two
end .
```

37.ags

```
program tester ( input , output ) ;
var six, five: integer;
four, three: boolean;
begin
if three = four then
five := six
end .
```

38.ags

```
program tester ( input , output ) ;
var ten, nine: integer;
eight, seven: boolean;
begin
if seven <> eight then
nine := ten
end .
```

39.ags

```
program tester ( input , output ) ;
var fourteen, thirteen: integer;
```

```
twelve, eleven: boolean;
begin
if eleven < twelve then
thirteen := fourteen
end .
```

4.ags

```
program tester ( input , output ) ;
var nine: integer;
begin
nine := 3
end .
```

40.ags

```
program tester ( input , output ) ;
var eighteen, seventeen: integer;
sixteen, fifteen: boolean;
begin
if fifteen <= sixteen then
seventeen := eighteen
end .
```

41.ags

```
program tester ( input , output ) ;
var three, two: integer;
one, zero: boolean;
begin
if zero >= one then
two := three
end .
```

42.ags

```
program tester ( input , output ) ;
var seven, six: integer;
five, four: boolean;
begin
if four > five then
six := seven
end .
```

43.ags

```
program tester ( input , output ) ;
var ten, nine: integer;
eight: integer;
begin
if eight in [1,2] then
nine := ten
end .
```

```
::::::::::::::
44.ags
::::::::::::::
program tester ( input , output ) ;
var fifteen, fourteen: integer;
thirteen, twelve: integer;
eleven: boolean;
begin
if eleven then
twelve := thirteen else fourteen := fifteen
end .
::::::::::::::
45.ags
::::::::::::::
program tester ( input , output ) ;
var sixteen: integer;
begin
case sixteen of
0: ;

end
end .
::::::::::::::
46.ags
::::::::::::::
program tester ( input , output ) ;
var zero, eighteen: integer;
seventeen: integer;
begin
case seventeen of
0: ;
24 : eighteen := zero
end
end .
::::::::::::::
47.ags
::::::::::::::
program tester ( input , output ) ;
var one: integer;
begin
case one of
0: ;
;

end
end .
::::::::::::::
48.ags
::::::::::::::
program tester ( input , output ) ;
var four, three: integer;
two: boolean;
begin
```

```
while two do
three := four
end .
```

49.ags

```
program tester ( input , output ) ;
var seven: boolean;
six, five: integer;
begin
repeat
five := six
until seven
end .
```

5.ags

```
program tester ( input , output ) ;
var ten: char;
begin
ten := 'Z'
end .
```

50.ags

```
program tester ( input , output ) ;
var twelve, eleven: integer;
ten, nine, eight: integer;
begin
for eight := nine to ten do
eleven := twelve
end .
```

51.ags

```
program tester ( input , output ) ;
var seventeen, sixteen: integer;
fifteen, fourteen, thirteen: integer;
begin
for thirteen := fourteen downto fifteen do
sixteen := seventeen
end .
```

52.ags

```
program tester ( input , output ) ;
type
exam = record
score : integer;
end;
var one, zero: integer;
eighteen: exam;
begin
```

```
with eighteen do
zero := one
end .
```

53.ags

```
program tester ( input , output ) ;
type
exam = record
score : integer;
end;
var five, four: integer;
three, two: exam;
begin
with two , three do
four := five
end .
```

54.ags

```
program tester ( input , output ) ;
label 91;
var seven, six: integer;
begin
91 : six := seven
end .
```

55.ags

```
program tester ( input , output ) ;
var eleven, ten: integer;
nine, eight: integer;
begin
eight := nine ;
ten := eleven
end .
```

56.ags

```
program tester ( input , output ) ;
var seventeen, sixteen: integer;
fifteen, fourteen: integer;
procedure twelve ;
var thirteen : integer ;
begin
fourteen := fifteen
end ;
begin
sixteen := seventeen
end .
```

57.ags

```
program tester ( input , output ) ;
var five, four: integer;
three, two: integer;
procedure eighteen ( zero : integer ) ;
var one : integer ;
begin
two := three
end ;
begin
four := five
end .
```

58.ags

```
program tester ( input , output ) ;
var twelve, eleven: integer;
ten, nine: integer;
procedure six ( var seven : integer ) ;
var eight : integer ;
begin
nine := ten
end ;
begin
eleven := twelve
end .
```

59.ags

```
program tester ( input , output ) ;
var zero, eighteen: integer;
seventeen, sixteen: integer;
procedure thirteen ( function fourteen : integer ) ;
var fifteen : integer ;
begin
sixteen := seventeen
end ;
begin
eighteen := zero
end .
```

6.ags

```
program tester ( input , output ) ;
const twelve = 5;
var eleven: integer;
begin
eleven := twelve
end .
```

60.ags

```
program tester ( input , output ) ;
var seven, six: integer;
```

```
five, four: integer;
procedure one ( procedure two ) ;
var three : integer ;
begin
four := five
end ;
begin
six := seven
end .
```

61.ags

```
program tester ( input , output ) ;
var fifteen, fourteen: integer;
thirteen, twelve: integer;
procedure eight ( nine : integer ; ten : integer ) ;
var eleven : integer ;
begin
twelve := thirteen
end ;
begin
fourteen := fifteen
end .
```

62.ags

```
program tester ( input , output ) ;
var two, one: integer;
zero, eighteen: integer;
function sixteen : integer ;
var seventeen : integer ;
begin
eighteen := zero
;
sixteen := 0;
end ;
begin
one := two
end .
```

63.ags

```
program tester ( input , output ) ;
var nine, eight: integer;
seven, six: integer;
function three ( four : integer ) : integer ;
var five : integer ;
begin
six := seven
;
three := 0;
end ;
begin
```

```
 eight := nine
 end .
:::::::::::::::
64.ags
:::::::::::::::
 program tester ( input , output ) ;
 var fifteen, fourteen: integer;
 thirteen, twelve: integer;
 procedure ten ;
 var eleven : integer ;
 begin
 twelve := thirteen
 end ;
 begin
 fourteen := fifteen
 end .
:::::::::::::::
65.ags
:::::::::::::::
 program tester ( input , output ) ;
 type
 sixteen = integer ;
 var zero, eighteen: integer;
 seventeen : integer ;
 begin
 eighteen := zero
 end .
:::::::::::::::
66.ags
:::::::::::::::
 program tester ( input , output ) ;
 type
 one = real ;
 var four, three: integer;
 two : integer ;
 begin
 three := four
 end .
:::::::::::::::
67.ags
:::::::::::::::
 program tester ( input , output ) ;
 type
 five = ( six ) ;
 var nine, eight: integer;
 seven : integer ;
 begin
 eight := nine
 end .
:::::::::::::::
68.ags
:::::::::::::::
 program tester ( input , output ) ;
```

```
type
ten = 35 .. 12 ;
var thirteen, twelve: integer;
eleven : integer ;
begin
twelve := thirteen
end .
```

69.ags

```
program tester ( input , output ) ;
type
fourteen = array [ 1 .. 2 , 1 .. 2 ] of integer ;
var seventeen, sixteen: integer;
fifteen : integer ;
begin
sixteen := seventeen
end .
```

7.ags

```
program tester ( input , output ) ;
type pointer = ^ integer;
var thirteen: pointer;
begin
thirteen := nil
end .
```

70.ags

```
program tester ( input , output ) ;
type
eighteen = record
end ;
var two, one: integer;
zero : integer ;
begin
one := two
end .
```

71.ags

```
program tester ( input , output ) ;
type
three = record
four : integer
end ;
var seven, six: integer;
five : integer ;
begin
six := seven
end .
```

72.ags

```
program tester ( input , output ) ;
type
eight = record
nine , ten : integer
end ;
var thirteen, twelve: integer;
eleven : integer ;
begin
twelve := thirteen
end .
```

73.ags

```
program tester ( input , output ) ;
type
fourteen = record
;
end ;
var seventeen, sixteen: integer;
fifteen : integer ;
begin
sixteen := seventeen
end .
```

74.ags

```
program tester ( input , output ) ;
type
eighteen = record
;
case integer of
end ;
var two, one: integer;
zero : integer ;
begin
one := two
end .
```

75.ags

```
program tester ( input , output ) ;
type
three = record
case integer of
end ;
var six, five: integer;
four : integer ;
begin
five := six
end .
```

76.ags

```
program tester ( input , output ) ;
type
seven = record
case integer of 89 : ( )
end ;
var ten, nine: integer;
eight : integer ;
begin
nine := ten
end .
```

77.ags

```
program tester ( input , output ) ;
type
eleven = record
case integer of 10 , 999 : ( )
end ;
var fourteen, thirteen: integer;
twelve : integer ;
begin
thirteen := fourteen
end .
```

78.ags

```
program tester ( input , output ) ;
type
fifteen = record
case integer of ;
end ;
var eighteen, seventeen: integer;
sixteen : integer ;
begin
seventeen := eighteen
end .
```

79.ags

```
program tester ( input , output ) ;
type
zero = record
case one :
integer of
end ;
var four, three: integer;
two : integer ;
begin
three := four
end .
```

8.ags

```
program tester ( input , output ) ;
var fifteen, fourteen: integer;
begin
fourteen := ( fifteen )
end .
```

80.ags

```
program tester ( input , output ) ;
type
five = set of integer ;
var eight, seven: integer;
six : integer ;
begin
seven := eight
end .
```

81.ags

```
program tester ( input , output ) ;
type
nine = file of integer ;
var twelve, eleven: integer;
ten : integer ;
begin
eleven := twelve
end .
```

82.ags

```
program tester ( input , output ) ;
type
thirteen = packed array [ 1 .. 2 , 1 .. 2 ] of integer ;
var sixteen, fifteen: integer;
fourteen : integer ;
begin
fifteen := sixteen
end .
```

83.ags

```
program tester ( input , output ) ;
type
seventeen = ^ integer ;
var one, zero: integer;
eighteen : integer ;
begin
zero := one
end .
```

84.ags

```
::::::::::::::
program tester ( input , output ) ;
type
two = integer ;
three = integer ;
var six, five: integer;
four : integer ;
begin
five := six
end .
::::::::::::::
85.ags
::::::::::::::
program tester ( input , output ) ;
const seven = 11 ;
var ten, nine: integer;
eight : integer ;
begin
nine := ten
end .
::::::::::::::
86.ags
::::::::::::::
program tester ( input , output ) ;
const eleven = 0.3 ;
var fourteen, thirteen: integer;
twelve : integer ;
begin
thirteen := fourteen
end .
::::::::::::::
87.ags
::::::::::::::
program tester ( input , output ) ;
const fifteen = 24.35E12 ;
var eighteen, seventeen: integer;
sixteen : integer ;
begin
seventeen := eighteen
end .
::::::::::::::
88.ags
::::::::::::::
program tester ( input , output ) ;
const zero = 89.10E+999 ;
var three, two: integer;
one : integer ;
begin
two := three
end .
::::::::::::::
89.ags
::::::::::::::
```

```
program tester ( input , output ) ;
const four = 11E0 ;
var seven, six: integer;
five : integer ;
begin
six := seven
end .
:::::::::::::::
9.ags
:::::::::::::::
program tester ( input , output ) ;
var sixteen: integer;
function seventeen : integer;
begin
seventeen := 0;
end;
begin
sixteen := seventeen
end .
:::::::::::::::
90.ags
:::::::::::::::
program tester ( input , output ) ;
const eight = +3 ;
var eleven, ten: integer;
nine : integer ;
begin
ten := eleven
end .
:::::::::::::::
91.ags
:::::::::::::::
program tester ( input , output ) ;
const twelve = -24 ;
var fifteen, fourteen: integer;
thirteen : integer ;
begin
fourteen := fifteen
end .
:::::::::::::::
92.ags
:::::::::::::::
program tester ( input , output ) ;
const seventeen = 8;
sixteen = seventeen ;
var one, zero: integer;
eighteen : integer ;
begin
zero := one
end .
:::::::::::::::
93.ags
:::::::::::::::
```

```
program tester ( input , output ) ;
const three = 4;
two = + three ;
var six, five: integer;
four : integer ;
begin
five := six
end .
:::::::::::::
94.ags
:::::::::::::
program tester ( input , output ) ;
const seven = 'Z' ;
var ten, nine: char;
eight : integer ;
begin
nine := ten
end .
:::::::::::::
95.ags
:::::::::::::
program tester ( input , output ) ;
const eleven = 'ZZ' ;
var fourteen, thirteen: char;
twelve : integer ;
begin
thirteen := fourteen
end .
:::::::::::::
96.ags
:::::::::::::
program tester ( input , output ) ;
const fifteen = 35 ;
sixteen = 12 ;
var zero, eighteen: integer;
seventeen : integer ;
begin
eighteen := zero
end .
:::::::::::::
97.ags
:::::::::::::
program tester ( input , output ) ;
label 89 ;
var three, two: integer;
one : integer ;
begin
89 :
two := three
end .
:::::::::::::
98.ags
:::::::::::::
```

```
program tester ( input , output ) ;
label 10 , 999 ;
var six, five: integer;
four : integer ;
begin
999 :
10 :
five := six
end .
```

# Appendix 2

# Part 4: the Partial files, "*.agsr"

A file in this group contains all the context-free rules that are used to produce a corresponding *.ags file. For example: "3.agsr" corresponds "3.ags".

```
::::::::::::::
0.agsr
::::::::::::::
********** applied production rules **********
0)      <program_heading> : program tester ( input , output ) ; $L
   <block> $$.  $L #
1)      <block> : $labelstart <label_declaration_part> $constant
    <constant_definition_part> $typestart
    <type_definition_part> $varstart <variable_declaration_part>
    $pfstart  <pro_and_fun_declaration_part> <statement_part> #
2)      <label_declaration_part> : #
7)      <constant_definition_part> : #
31)     <type_definition_part> : #
84)     <variable_declaration_part> : var <re_variable_declaration>  ;
   $L #
86)     <re_variable_declaration> :
   <variable_declaration> #
88)     <variable_declaration> : <re_identifier> : <type> #
43)     <re_identifier> :  $name #
36)     <type> : <simple_type> #
39)     <simple_type> : <type_identifier> #
46)     <type_identifier> : integer #
89)     <pro_and_fun_declaration_part> : <re_pro_or_fun_declaration>  #
90)     <re_pro_or_fun_declaration> :  #
109)    <statement_part> : <compound_statement> #
190)    <compound_statement> : begin $L <re_statement> $L $checkfun  end   #
191)    <re_statement> :  <statement> #
110)    <statement> : <unlabelled_statement> #
112)    <unlabelled_statement> : <simple_statement> #
114)    <simple_statement> : <assignment_statement> #
118)    <assignment_statement> : $assg <variable> :=  <expression> $assgend #
119)    <variable> : <entire_variable> #
122)    <entire_variable> : <variable_identifier> #
123)    <variable_identifier> : $name #
138)    <expression> : <simple_expression> #
147)    <simple_expression> : <term> #
152)    <term> : <factor> #
159)    <factor> : <variable> #
119)    <variable> : <entire_variable> #
122)    <entire_variable> : <variable_identifier> #
123)    <variable_identifier> : $name #

::::::::::::::
1.agsr
::::::::::::::
********** applied production rules **********
0)      <program_heading> : program tester ( input , output ) ; $L
   <block> $$.  $L #
1)      <block> : $labelstart <label_declaration_part> $constant
    <constant_definition_part> $typestart
    <type_definition_part> $varstart <variable_declaration_part>
    $pfstart  <pro_and_fun_declaration_part> <statement_part> #
2)      <label_declaration_part> : #
```

```
7)      <constant_definition_part> : #
31)     <type_definition_part> : #
84)     <variable_declaration_part> : var <re_variable_declaration>  ;
   $L #
86)     <re_variable_declaration> :
   <variable_declaration> #
88)     <variable_declaration> : <re_identifier> : <type> #
44)     <re_identifier> : <re_identifier> , $name  #
43)     <re_identifier> :  $name #
36)     <type> : <simple_type> #
39)     <simple_type> : <type_identifier> #
46)     <type_identifier> : integer #
89)     <pro_and_fun_declaration_part> : <re_pro_or_fun_declaration>  #
90)     <re_pro_or_fun_declaration> :  #
109)    <statement_part> : <compound_statement> #
190)    <compound_statement> : begin $L <re_statement> $L $checkfun  end   #
191)    <re_statement> :  <statement> #
110)    <statement> : <unlabelled_statement> #
112)    <unlabelled_statement> : <simple_statement> #
114)    <simple_statement> : <assignment_statement> #
118)    <assignment_statement> : $assg <variable> :=  <expression> $assgend #
119)    <variable> : <entire_variable> #
122)    <entire_variable> : <variable_identifier> #
123)    <variable_identifier> : $name #
138)    <expression> : <simple_expression> #
147)    <simple_expression> : <term> #
152)    <term> : <factor> #
159)    <factor> : <variable> #
119)    <variable> : <entire_variable> #
122)    <entire_variable> : <variable_identifier> #
123)    <variable_identifier> : $name #

::::::::::::::
10.agsr
::::::::::::::
********** applied production rules ***********
0)      <program_heading> : program tester ( input , output ) ; $L
   <block> $$.  $L #
1)      <block> : $labelstart <label_declaration_part> $constart
    <constant_definition_part> $typestart
    <type_definition_part> $varstart <variable_declaration_part>
    $pfstart  <pro_and_fun_declaration_part> <statement_part> #
2)      <label_declaration_part> : #
7)      <constant_definition_part> : #
31)     <type_definition_part> : #
85)     <variable_declaration_part> : #
89)     <pro_and_fun_declaration_part> : <re_pro_or_fun_declaration>  #
90)     <re_pro_or_fun_declaration> :  #
109)    <statement_part> : <compound_statement> #
190)    <compound_statement> : begin $L <re_statement> $L $checkfun  end   #
191)    <re_statement> :  <statement> #
110)    <statement> : <unlabelled_statement> #
112)    <unlabelled_statement> : <simple_statement> #
```

```
114)    <simple_statement> : <assignment_statement> #
118)    <assignment_statement> : $assg <variable> :=  <expression> $assgend #
119)    <variable> : <entire_variable> #
122)    <entire_variable> : <variable_identifier> #
123)    <variable_identifier> : $name #
138)    <expression> : <simple_expression> #
147)    <simple_expression> : <term> #
152)    <term> : <factor> #
162)    <factor> : <function_designator> #
170)    <function_designator> : $paraflag <function_identifier> (
      <re_actural_parameter>   ) $paraend #
173)    <function_identifier> : $funname #
171)    <re_actural_parameter> :  <actural_parameter>   #
184)    <actural_parameter> : <variable> #
119)    <variable> : <entire_variable> #
122)    <entire_variable> : <variable_identifier> #
123)    <variable_identifier> : $name #

::::::::::::::
11.agsr
::::::::::::::
********** applied production rules **********
0)      <program_heading> : program tester ( input , output ) ; $L
    <block> $$.  $L #
1)      <block> : $labelstart <label_declaration_part> $constart
     <constant_definition_part> $typestart
     <type_definition_part> $varstart <variable_declaration_part>
     $pfstart  <pro_and_fun_declaration_part> <statement_part> #
2)      <label_declaration_part> : #
7)      <constant_definition_part> : #
31)     <type_definition_part> : #
85)     <variable_declaration_part> : #
89)     <pro_and_fun_declaration_part> : <re_pro_or_fun_declaration>  #
90)     <re_pro_or_fun_declaration> :  #
109)    <statement_part> : <compound_statement> #
190)    <compound_statement> : begin $L <re_statement> $L $checkfun  end   #
191)    <re_statement> :  <statement> #
110)    <statement> : <unlabelled_statement> #
112)    <unlabelled_statement> : <simple_statement> #
114)    <simple_statement> : <assignment_statement> #
118)    <assignment_statement> : $assg <variable> :=  <expression> $assgend #
119)    <variable> : <entire_variable> #
122)    <entire_variable> : <variable_identifier> #
123)    <variable_identifier> : $name #
138)    <expression> : <simple_expression> #
147)    <simple_expression> : <term> #
152)    <term> : <factor> #
163)    <factor> : <set> #
174)    <set> : [ $element_list <element_list> $element_end ] #
175)    <element_list> :  #

::::::::::::::
12.agsr
```

```
::::::::::::::
::::::::::::::
********** applied production rules **********
0)      <program_heading> : program tester ( input , output ) ; $L
    <block> $$.  $L #
1)      <block> : $labelstart <label_declaration_part> $constart
     <constant_definition_part> $typestart
     <type_definition_part> $varstart <variable_declaration_part>
     $pfstart  <pro_and_fun_declaration_part> <statement_part> #
2)      <label_declaration_part> : #
7)      <constant_definition_part> : #
31)     <type_definition_part> : #
85)     <variable_declaration_part> : #
89)     <pro_and_fun_declaration_part> : <re_pro_or_fun_declaration>  #
90)     <re_pro_or_fun_declaration> :  #
109)    <statement_part> : <compound_statement> #
190)    <compound_statement> : begin $L <re_statement> $L $checkfun  end   #
191)    <re_statement> :  <statement> #
110)    <statement> : <unlabelled_statement> #
112)    <unlabelled_statement> : <simple_statement> #
114)    <simple_statement> : <assignment_statement> #
118)    <assignment_statement> : $assg <variable> :=  <expression> $assgend #
119)    <variable> : <entire_variable> #
122)    <entire_variable> : <variable_identifier> #
123)    <variable_identifier> : $name #
138)    <expression> : <simple_expression> #
147)    <simple_expression> : <term> #
152)    <term> : <factor> #
163)    <factor> : <set> #
174)    <set> : [ $element_list <element_list> $element_end ]  #
176)    <element_list> : <re_element> #
177)    <re_element> : <element> #
179)    <element> : <expression> #
138)    <expression> : <simple_expression> #
147)    <simple_expression> : <term> #
152)    <term> : <factor> #
159)    <factor> : <variable> #
119)    <variable> : <entire_variable> #
122)    <entire_variable> : <variable_identifier> #
123)    <variable_identifier> : $name #
```