Interfacing of a Euclid Compiler to a Pascal Code Generator

by

Pohwa Hsu

B.A., Fu-Jen Catholic University, 1976

-----------------

A MASTER'S REPORT

submitted in partial fulfillment of the requirements for the degree

MASTER OF SCIENCE

Department of Computer Science

KANSAS STATE UNIVERSITY
Manhattan, Kansas

1982

Approved by:

[Signature]
Major Professor
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Introduction</td>
<td>1</td>
</tr>
<tr>
<td>Chapter 1: Background Information</td>
<td>3</td>
</tr>
<tr>
<td>Chapter 2: Design Approach</td>
<td>17</td>
</tr>
<tr>
<td>Chapter 3: Implementation Difficulties</td>
<td>21</td>
</tr>
<tr>
<td>Chapter 4: Final Remark</td>
<td>34</td>
</tr>
<tr>
<td>Appendix A: Modified Aprial Euclid Coder S/SL</td>
<td>38</td>
</tr>
<tr>
<td>Appendix B: New and Modified Euclid Coder Semantic Routine</td>
<td>130</td>
</tr>
<tr>
<td>Appendix C: Modification of April Euclid Allocator Pass</td>
<td>139</td>
</tr>
<tr>
<td>Appendix D: Modification of Pas32 Pass 6, Pass 7 and Pass 8</td>
<td>141</td>
</tr>
<tr>
<td>Reference</td>
<td>144</td>
</tr>
</tbody>
</table>
INTRODUCTION

A compiler for the programming language Euclid [6] is currently available from the Computer Systems Research Group of Toronto. It runs under and generates code for a PDP11 machine running the UNIX Operating System. Euclid is designed primarily for writing system programs and to aid in automatic verification of programs.

The Computer Science Department at Kansas State University has an Interdata 8/32 machine. A project is being conducted in the department to make Euclid available for compilation and execution on the 8/32 machine. This report is part of the design for this compiler modification.

The Euclid compiler now available is written in Euclid. A Pascal compiler, written in Pascal, also exists, which generates 8/32 machine code [5]. Both existing compilers are multi-pass compilers. The intended solution is to build one intermediate pass which serves as a translation from one of the Euclid interpass languages to one of the PASCAL interpass languages. A Euclid program will first be translated by the Syntactic and Semantic Passes of the existing Euclid compiler. It will then be translated by the new pass to an appropriate form. This will finally be translated by the optimization and code generator passes of the Pascal compiler, producing 8/32 machine code.

The report is organized as follows: Chapter 1 gives a general explanation of how the Euclid compiler and the PAS32 Pascal compiler work. It explains the major differences between them and indicates where the actual translation takes place. A design tool used by the Euclid compiler, called Syntax/Semantic Language, is also introduced in this chapter. Chapter 2 specifies the design approach, including
the design of the last pass of the Euclid compiler, called the Coder pass. The Coder is drastically modified to produce the new interface pass. It uses semantic mechanisms written in Euclid and a control portion written in a special compiler writing language called Syntax/Semantic Language (S/SL). The need for modifying the existing Coder semantic routines is also specified. Chapter 3 lists the major problems encountered in the design and describes the solutions for them. Chapter 4 summarizes the work already done and the future work required. Appendix A is the actual S/SL program for the new pass. Appendix B lists new or modified Euclid Coder semantic routines. Appendix C describes the changes which need to be done by Euclid Allocator pass. Appendix D specifies new operators which need to be introduced to Pas32 pass 6.

Thanks are given to Dr. Bates, my major professor, who has patiently directed me toward the objective and has given me much advice in the design process.
CHAPTER 1: BACKGROUND INFORMATION

1.1 The April Euclid Compiler

The existing Euclid compiler accepts a subset of the full Euclid language called April Euclid. The compiler itself is called the April Euclid compiler. Throughout this report, the term "April Euclid" will be used to mean the April Euclid compiler, rather than the April Euclid language.

April Euclid has six passes. These are named the Scanner/Screener, Parser, Builder, Conformance Checker, Allocator and Coder. Each pass performs the functions as described below.

1.1.1 Scanner and Screener

The Scanner is a callable routine with one input stream and three outputs. It reads its input one character at a time until it has recognized a complete elemental token. It will output a token number, the (blank-trimmed) text for the token, and an error message, if necessary.

The Screener provides the token stream to be passed to the Parser. The Screener calls the Scanner to obtain elemental tokens. It resolves identifier tokens into identifiers and word Symbols (keywords) and assigns to identifiers unique identifier numbers. The Screener outputs a token stream consisting of token numbers, constant values and identifier numbers, and token text. The token stream will be passed on to the Parser. Error messages will be passed to the Error Message Handler for later printing of messages. Together, the Scanner and Screener are a classical Lexical Analyzer.
1.1.2 Parser

The second pass of April Euclid checks syntax of the token stream produced by the Screener and emits a translation of the token stream. It converts the token stream to a form which is primarily postfix notation. There are occasional prefix and infix operators, where subsequent passes need them. This token stream is, with minor exceptions, unchanged by all subsequent passes, except, of course, the coder pass.

1.1.3 Builder [ 2 ]

The Symbol Table Builder pass constructs a Symbol Table and replaces all identifier numbers by references to Symbol Table entries. It also constructs a Type Table with an entry for every named or anonymous type. It has the following responsibilities:

-- Allocate a disk entry for each predefined or user-defined identifier. Allocate a disk entry for each predefined or user-defined type. Provide appropriate connections among these entries and fill in various fields in the entries.

-- Replace each use of an identifier by a pointer to the table entry corresponding to its declaration.

-- Allow for inspection of Symbol and Type table entries and insertion of new information into entries by later passes.

1.1.4 Conformance Checker [ 2 ]
The responsibility of the Conformance pass is to enforce many semantic rules of the language, in order to guarantee the integrity of the later passes. Specific functions are:

-- Enforce IMPORT and EXPORT lists as well as various restrictions peculiar to Euclid.

-- Prevent redeclaration in a closed scope, enforce the Free Variable Rule, check that references outside of the local closed scope are to PERVERSIVE identifiers, etc.

-- Check structure in types. Verify that the type constructors such as "..", ARRAY are appropriately used.

-- Check values in types. Values which are not manifest (known at compile time) will require certain run time checks, generated by later passes.

-- Verify that ranges of scalars in records and arrays are manifest.

-- Fold constants and determine sizes of variables.

1.1.5 Allocator [ 2 ]

The primary responsibility of the Allocator pass is to allocate run-time memory space and place this information into the tables. It assigns each non-manifest value a displacement and a base. In the simple situation, the displacement is manifest and the base is a display level which locates a routine's activation record. Generally, the Allocator ignores executable statements and expressions.
1.1.6 Coder (ref 2)

The Euclid Coder pass generates PDP-11 assembly language. It scans the token stream produced by Allocator and uses the Symbol and Type Table to generate the code. It uses semantic data structures of a Case Table Stack, Symbol Stack, Type stack, Checked Stack, Construct Stack, Count Stack, Display Stack, Label Stack, Locator Mechanism and Favor Stack [10]. Many semantic routines operate on these data structures. Some simple optimization is done.

1.2 Syntax/Semantic Language [3]

Syntax/Semantic Language (S/SL) is a language developed by Auturn, Et. Al. as a tool for constructing compilers. It incorporates the following features: sequences, repetitions and selections of actions; input, matching and output of tokens; output of error signals; subprograms (called rules); and invocation of semantic operations, which must be written in some other language. S/SL has no variables or assignments.

Without semantic actions, the language is called SL (Syntax Language). Each SL program consists of a list of executable rules. Each rule consists of a name, an optional return type and a sequence of actions. Each rule has one of these two forms:

name: actions; % procedure rule

name >> type: actions; % choice rule
1.2.1 Summary of SL actions

1.2.1.1 The call action.

The appearance of @ followed by the name of a procedure rule signifies that the rule is to be called. i.e. @BlockBody

1.2.1.2 The return action.

The symbol >> in a rule signifies return before reaching the end of the rule.

1.2.1.3 The input action.

The appearance of an input token name in a rule signifies that the next input should be read and must match the token. For example, the following specifies that the next item in the input stream is to be read and it must be an identifier token

ident

In our translation, we add the symbol "=" together with the input token to make the input action explicit. i.e. =ident

1.2.1.4 The emit action.

The appearance of a dot (a period) followed by an output token signifies that the token is to be emitted to the output stream. For example:
.pushaddr

1.2.1.5 The error action.

The appearance of # followed by an error signal signifies that the error signal name is to be emitted to a special error stream, for example:

#EDeferredMultipleUsedModule

1.2.1.6 The cycle action.

Each cycle is of the form:

```c
{
    actions
}
```

The enclosed actions are repeated until a return (>>) or one of the cycle's exits (>) is encountered.

1.2.1.7 The exit construct.

The appearance of > signifies exit from the most tightly enclosing cycle.

1.2.1.8 The choice action.

The choice action is of the form:

```c
[ selector
    | label, label, ... : actions
    | labels, ... :
```
The selector is optional. If it is omitted, the action is an input choice, which tries to match the current input token to one of the labels. The selector can be of the form \texttt{@identifier} where "identifier" is the name of a choice rule; this gives a rule choice. In a rule choice, the specified rule is called and then the choice tries to match the returned value to one of the labels. There is also a semantic choice, in which the selector is the name of a semantic operation. If no label is matched, the final alternative, labelled by star (\texttt{*}), is executed.

In an input choice, the matching of the input token to a label has the side effect of reading another token; if the token is not matched and the otherwise clause is selected, then reading is not done.

1.2.1.9 Comment.

Everything that follows "#", up to the end of the current line is treated as a comment.

1.2.2 Semantic Mechanisms

SL is sufficient for implementing Parsers for languages
having LR(k) grammars, however, it cannot do semantic analysis unless the ability to call Semantic Mechanisms is added.

Example of a semantic mechanism: A stack of counters.

    mechanism Count:
    PushCount(number) % Push a number onto counter stack
    ChooseCount >> number % Inspect top counter
    ;

This example shows a declaration of a semantic mechanism. It can appear in an S/SL program and it makes pushcount and choosecount available as semantic routines to be called. The semantic mechanism must actually be implemented in some other language which has variables and assignments.

A complete S/SL example:

    SkipExpression: % procedure rule
    {
    % repeat
    [ % input choice
    | EndExpression: % label
    > % return
    | ASubs:
    @SkipSubscripts % call another procedure
    | AIn, ANotIn:
    @SkipExpression % recursive call
    [
S/SL can be viewed as a regular expression-like notation for syntax diagrams, where input and output syntax and semantic routine calls are all combined into one. In addition, choices are made deterministic in S/SL.

Each pass of the Euclid compiler except for the Screener, is written partly in S/SL, with semantic mechanisms written in Euclid.

There are several dialects of S/SL. This project uses the one used in the April Euclid Compiler. Choice rules are not used, and S/SL programs have roughly the form of recursive descent parsers written in S/SL.

1.3 PASCAL Compiler [4, 5]

The Pascal compiler used in this project is a descendent of Hartmann's sequential Pascal compiler. It has been extensively modified by various people. It is called Pas32, and this name will be used throughout this report to refer to it. Pas32 is written in Pascal and generates Interdata 8/32 machine code.

Pas32 is a 9 pass compiler. Pass 1 and pass 2 serve as the syntax
checker. Passes 3, 4 and 5 are semantic analysis. Pass 6 thru pass 9 are optimization and code generation. The compiler makes a very clear division of labor among these passes. With minor exceptions, all interpass communication is through the token streams.

1.3.1 Lexical Analysis

Pass 1 of the Pas32 is the lexical scanner. It converts the character input to the compiler into the tokens recognized by the Parser in pass 2. Each unique identifier is mapped into a spelling index for use by passes 2 and 3. Pass 1 uses a hash table; this is used to perform a hashed lookup of identifiers and to map identifiers into spelling indices.

1.3.2 Syntax Analysis

Pass 2 is a parser which checks the syntax of the first intermediate code generated by pass 1. The Syntax analysis eliminates redundant operators and replaces ambiguous operators by unique ones. It also does the error recovery if syntax errors are detected. The output of the syntax analysis is primarily postfix notation. It is syntactically correct independent of what the input is.

1.3.3 Name Analysis

Pass 3 applies the language scope rules to convert references to identifiers into references to the appropriate unique objects in each part of the program. These are called noun indices. It also collects some structural information about objects which is passed to pass 4 in the intermediate code. It
replaces constant references in the token stream by their values or addresses.

1.3.4 Declaration Analysis

Pass 4 performs storage allocation and outputs data addressing information along with data type information in the intermediate code. The input to pass 4 references objects by noun index, as assigned by pass 3. The output of pass 4 will reference objects by an addressing mode/context/displacement/level 4-tuple. Additionally, references to objects will include the type kind/noun/length information. The output from pass 4 will generally contain only executable instructions; all declaration information will have been distributed among the various executable operators.

1.3.5 Operand Analysis

The function of pass 5 is to perform operand type checking on all of the operators in the program. This pass applies the operand compatibility rules of the language. The input to pass 5 references objects by addressing mode, displacement, declaration context, and lexical level. Additionally, object types are referenced by type noun, kind, and length. The output token stream addresses objects in much the same manner. However, type information is coded into a smaller number of types (word, halfword, byte, real, shortreal, set and structure). The structure type includes a length specification, and the other types have a fixed length. The output is a abstract stack machine language.
1.3.6 Machine Independent Optimization

Pass 6 of Pas32 performs constant folding and some optimizations to simplify the operation of the later passes.

The basic operation of pass 6 is to build an in-memory data structure (a stack of lists of trees) representing one procedure from the input text. Most constant folding is performed while this data structure is built as are many of the code transformations. This data structure is traversed to output code for pass 7. During this traversal, a few optimizations and transformations are performed which require information not conveniently available while the data structure was being built. Also, boclean logic is converted from boolean operators to branching logic as the output is being produced.

1.3.7 Code Generation

Pass 7 of the Pas32 accepts the instructions of the virtual stack machine produced by pass 5. The format has been modified slightly in structure by pass 6. The output is 8/32 instructions ready for assembly by passes 8 and 9.

Pass 7 performs register allocation for general and floating point registers. It selects machine instructions. It generates symbolic references to code labels, procedure labels, statement labels, literal constants, and stack depths. It may also emit output for constant declarations.

1.3.8 Machine Dependent Optimization
The main function of pass 8 is doing "peephole" optimization. It performs many simple, machine dependent optimizations, each of which requires examination of only a small, fixed amount of code. Examples include eliminating redundant labels, eliminating unnecessary branches, removing unused code, etc.

1.3.9 Final Assembly

The final pass of Pas32 assembles the generated code to 8/32 machine language. It translates the previously defined symbolic labels to actual machine addresses. It also changes previously defined instructions to actual machine instructions.

1.4 Differences between the two compilers

Although the two compilers have different numbers of passes, they perform roughly the same compiling sequence as on many multiple pass compilers. The differences are in the data structure and in the operations that manipulate it. These differences imply different intermediate token streams for the two compilers.

The Builder pass of April Euclid forms a Symbol/Type Table as the main data structure. The subsequent passes which do the semantic checking, storage allocation and code generation, all operate on this structure. Each pass of April Euclid just adds to or modifies the existing Symbol/Type Table entries for its own use and for use by the next pass. Nevertheless, the intermediate code is still emitted by each pass. Between the Parser and the Coder, all intermediate token streams are identical, except for minor changes, such as replacement of identifier numbers by table pointers in the builder.
In the Pas32, each pass builds and uses its own data structure, such as the spelling table and the entry records for Name Analysis, noun table for Declaration Analysis, operand stack for Operand Analysis and tree stack for Machine Dependent Optimization. Each pass accepts input tokens together with semantic information from the previous pass, builds its own data structure, analyzes the structure, and produces a new token stream with new semantic information for the next pass. The pass' data structure is then discarded at the end of the pass. Each inter-pass token stream language is significantly different from the others.
CHAPTER 2: DESIGN APPROACH

The interfacing of the two compilers takes place after the Allocator pass of April Euclid and before the Machine Independent Optimization (pass 6) of Pas32. Since this is the common place where both compilers have finished the Syntactic and Semantic analysis and the storage allocation, it minimizes the complexity of the translation and the modification of both compilers, yet still preserves most features of the Euclid language.

The operations performed by the two compilers before and after this point are of little concern to this project. The design was done primarily by considering how the information in the token stream and Symbol/Type tables produced by the Euclid allocator pass could be transformed into a token stream acceptable to pass 6 of Pas32. A modified April Euclid coder pass [8] was designed to perform this transformation. The following are the considerations for the design.

2.1 Project Scope

The full Euclid language contains many new features which are not available in most computer languages currently used. Some of these cannot be translated directly by the Pas32 code generation passes. The April Euclid language subset of full Euclid is semantically much closer to Pascal. This project is limited to translation of features supported by April Euclid. A few extensions have been considered, but some others such as "parameterized types" [9] and "multiple use modules" [6] are left untouched.

2.2 Flexibility and Understandability
The full implementation of Euclid may be expected in the future. Future expansion of this pass will also be required. It is a major concern that future work can easily be done. Using S/SL, a Euclid feature not implemented by April Euclid can be replaced by a warning or error message. This can be easily replaced later. The S/SL language itself is structured and reasonably self-documenting. The design is largely written in S/SL and will cause no difficulty in understanding for a reader familiar with April Euclid.

2.3 Use of the April Euclid Coder [8]

Another reason that this design is based on the April Euclid Coder pass is that Coder has many useful semantic routines [15] and data structures for the new pass. The new pass uses almost the same semantic mechanisms as the April Euclid Coder. The portion of the Coder written in S/SL also provides an understandable basic logic sequence that the new translation needs. Basing the design on the April Euclid Coder not only increases understandability but reduces the design work.

2.4 Design logic

April Euclid keeps all the information about symbols and types in a Symbol/Type Table; the token stream of April Euclid just represents the syntax of the language. Pas32 keeps all the semantic information in its token stream. The translation logic for the design is a syntax translation with extra information taken from Euclid Symbol/Type Table and put into the output token stream. Most syntax translations are fairly
easy; the difficulty remains in locating the required semantic information and emitting it at the right place.

2.5 Specification of the implementation

Though this pass takes many useful tools directly from the April Euclid Coder, there are still some semantic mechanisms which had to be modified, removed and added. This design gives only the specifications of these altered functions (appendix B). Examples are the New semantic routine "OFushNewLoopLabel", which operates on a new mechanism, the "Loop Stack"; The semantic routine "OADjustRunTimeStackDepthSubroutineBegin", which has been removed because this is not really a run time code generator; and the April Euclid Coder S/SL procedure "EnterTypeInfoIntoSymbol" which takes information from the type stack and enters it into the corresponding symbol stack entry for later assignment use, which has been modified to add more semantic routines to enter additional information needed for the assignments by Pas32.

We have mentioned that some work can not be done by direct translation. Many new semantic routines are designed on the assumption that some changes to the Euclid Symbol/Type Table can be made by modifying earlier passes. These new semantic routine and the required table changes are specified by this design. A typical example of these is to emit semantic information that is not currently available within the April Euclid Symbol/Type Table, such as the parameters length for a routine.
Some changes also have to be made to Pas32 pass 6 and later, in order to accept the new features of the April Euclid language. In most cases, new operators will serve as the communication between the April Euclid and Pas32. We define the function of these new operators in this design: it will be the work the implementor to decide how actually implement them. For instance, a new operator "pop" is emitted to Pas32 pass 6 if an unnecessary expression has been generated by this new pass. When Pas32 pass 6 accepts the operator, it will delete the built tree structure for that expression.

All the design work, including translation S/SL, specification of modified semantic mechanisms, specifications for changes to earlier April Euclid passes and specifications for changes to later Pas32 passes, is listed in appendices.
CHAPTER 3: IMPLEMENTATION DIFFICULTY

In this design, most translations are fairly straightforward. There also exist some translations which need more thought, where some others can not be handled by this pass itself. The main purpose of this chapter is to detail these problems. We have given the solutions to most problems encountered; we also give the suggested solutions to problems not solved by the new pass.

3.1 Scope of a Module

A Euclid module is a generalization of a record type. Module components may be declared as constants, variables, routines or types. Identifiers declared in a module are not known outside unless they are exported explicitly [6]. Since Pas32 does not have a useable equivalent of modules, we remove the boundaries of modules. This means we treat the local identifiers of the modules as local to the same scope as the module itself. The module is invisible to its surrounding block after the translation. An exported routine of this module should be seen as normal routine internal to the surrounding block.

3.2 Defining the outermost scope for entering a program.

In Euclid, a main program is defined as a module type. A problem arises when encountering the main module, since the main module does not have any surrounding block. It would be necessary to define a unique semantic routine that generates level information for main program call and entry. The solution used instead is to push a dummy Symbol Stack entry and to enter the Pas32 program level before we reach the
program module in Euclid. In this way, we can retain consistency in referencing variables which are declared in modules.

3.3 Scope of INVARIANT routine, INITIALLY routine and POST routine of a module

Consider the following example of a Euclid module:

```
VAR modulename: module

CHECKED

CONST stringmaxlength: VALUE := 30;
TYPE stringindex = 1 .. stringmaxlength;
VAR var1: stringindex;
PROCEDURE procedure1 =
  PRE  c / d = 2;
  POST c / d = 2 * k;
  BEGIN
    ...
    ...
    ...
    END procedure1
  ...

INITIALLY
  BEGIN
    ...
    procedure1;
    ...
  END
```
INVARIANT

\[ x + y = a + b; \]

END MODULE

INITIALLY is a parameterless procedure that will be executed when an instance of the module is created. It could call other routines declared inside the module. A call on INVARIANT may be generated by the compiler before exiting from INITIALLY.

INVARIANT is a boolean expression that will always be true during the lifetime of the module variable, except perhaps when one of the procedures of the module has been called and has not yet returned. It is evaluated by INITIALLY and by other procedures of the module (before the execution of the body and after the execution of body). If the expression is false, a runtime error is generated.

POST is also a boolean expression that should be true after the routine body is executed. It is handled similarly to INVARIANT. INVARIANT and POST expressions are translated into procedures. The difficulty is that neither of them has a corresponding Symbol Table entry, which means no information about the scope they belong to can be found in the normal way. The semantic routine "GenerateOneUpperLevel" is therefore added. It will emit one higher level than the level of the top routine on the Symbol Stack.

For INITIALLY, either when compiling a call or when compiling the routine, the top symbol on the Symbol Stack at the time is
the module variable. Since a module does not form its own scope, we must search the second top symbol of the Symbol Stack to locate the routine which surrounds that module and then perform the semantic routine "GenerateOneUpperLevel".

For the INVARIANT routine entry point and for the INVARIANT call statement within INITIALLY, the situation is the same as above, but for the INVARIANT call statement within a local procedure of the module, because the top symbol on Symbol Stack is for the procedure we are inside, we need to find the third top symbol of the Symbol Stack. To make a consistent rule for calling INVARIANT, a dummy symbol stack entry is pushed before calling INVARIANT in the INITIALLY routine and is popped after the call, such that the third entry of the symbol stack will always be the appropriate outer scope of INVARIANT when need.

For the POST routine, the semantic routine "GenerateOneUpperLevel" is just applied to the top of the Symbol Stack.

3.4 Compiling one procedure at a time.

The code generation passes of Pas32 can't compile nested procedures. The token stream emitted by the modified Coder contains nested procedure bodies. So when entering a new routine and when leaving that routine, new tokens "SkipForNesting" and "ResumeForNesting" will be emitted. The Pas32 Code Generator will figure out how to rearrange the procedures.
3.5 Compiling a FORWARD routine.

This problem also arises for because Pas32 compiles a procedure at a time. A routine declaration with FORWARD specified has its body declared somewhere later than the routine call statement. Earlier passes of Pas32 merge the forward declaration with the body, for PASCAL programs. Euclid also has forward declarations, but they are not merged by earlier passes of April Euclid. The token stream emitted by the new pass for forward declarations and their bodies is similar to that of nested routine. We emit a new token "BeginBodySplit" when compiling the "FORWARD", which tells Pas32 pass 6 that the body of the routine is separated from the heading, and emit a new token "EndBodySplit" when reaching the body of this routine. Again, Pas32 must be modified to accept this kind of split procedure body.

3.6 Initial values in variable declarations.

In Euclid, variables can have initial values and these may be nonmanifest. e.g. VAR a,b,c,... : integer := 4 * x; In compiling the statement, it is necessary to process the declarations of variables while deferring code generation until the expression giving the initial value is evaluated. Although the code for evaluating the initial value can be generated after, it is difficult to get multiple copies of the value and assign one copy to each of the variables. In fact, without changes to Pas32, it would be necessary to generate code to evaluate the expression repeatedly. The solution
chosen is to emit the name of each variable declared, and keep track of the number of variables processed by using a counter. After the code for the expression is emitted (only evaluate this once), a new operator "InitialAssign" or "InitialCopy" is produced depending the assign type, also giving the counter value. Pas32 must be able to make multiple assignments from this operator.

3.7 Assigning a value to a constant

In Euclid, a constant value can be a nonmanifest expression. The value for that constant must be decided at block entry, and can not be changed during the lifetime of the block. A defined constant can have different values in different blocks. The coder pass of April Euclid deferred nonmanifest constants. However, this pass translates them anyway. For nonmanifest constant initialization, the translation is just like variable initialization. This means assignment to a constant will be made. A reference to a nonmanifest constant will be treated as variable reference.

3.8 Passing manifest parameter size to routine call and routine entry

Pascal requires the total size of all the parameters to be emitted as a constant value on a routine call and on a routine entry. This design assumes the April Euclid Allocator pass has solved this problem and has put the correct size of parameters into the routine's Symbol Table entry, such that it can be emitted directly from the top Symbol Stack entry.
3.9 Loop stack for loop statements

Euclid defines loop statement as:

```
LOOP
...
statement
...
EXIT
...
IF ... THEN EXIT
EXIT WHEN expression
...
ENDLOOP
```

The statements in the scope are executed repeatedly until control leaves the scope through one of the EXIT statements. A new stack with one entry for each loop statement is added to the modified Coder. The reason for creating this new loop label stack is that an EXIT statement may appear inside an IF statement, and its labels would otherwise be mixed up with the labels created in compiling the IF expression. The loop stack entry will only have a loop entry label and a loop exit label. This is a convenience for EXIT statement processing, which need only pick the top label of the loop stack to jump to.

An alternate method for this translation might be to follow what the April Euclid coder does. Instead of creating a new label stack, April Euclid uses a construct stack. It accumulates all the exit labels of a loop in an appropriate
construct stack entry, and emits them at end of the loop body. A search to the nearest construct stack entry for a loop must be made when exiting and emitting the exit labels.

3.10 "assertion" test

In Section 3.2, we defined the IN Variant and POST assertions. The ASSERT and PRE statements are similar; all contain expressions which are supposed to hold whenever control reaches the appropriate point in the program, and execution of the object program must be terminated if one of these expressions is false.

When the boolean result is false. The new operator "ErrAssert" together with a line number will be emitted to satisfy this requirement. Pas32 must accept this operator and translate it into code which generates an unconditional runtime error "Assertion failure".

3.11 Nonmanifest range checks for assignment

For assignments to enumeration types and subrange types, a "range" operator together with the value of "first" and "last" must be emitted for the Pas32 code generation passes to generate a runtime check. Since "first" and "last" of a enumeration type or a subrange type can be nonmanifest in Euclid, a new operator "varrange" is added. "first" and "last" for this operator will be general expressions preceding "varrange" rather than fixed semantic operands, as they are for "range". The Pas32 code generation passes must accept this operator.
3.12 Referencing an element of nonmanifest sized array

In Pascal, array subscripting requires an operator "index" together with the value of "first", "last" and "size". In Pascal, these values are all manifest, but not in Euclid. We add a new operator "addindex" and emit it if any one of the three is nonmanifest. Similar to "varrange", this operator accepts "first", "last" and "size" as expressions preceding the operator.

3.13 Optimizing logical expressions

Both the April Euclid coder pass and pass 6 of Pas32 optimize boolean expressions by branching logic which avoids evaluation of the second operand of a boolean operator when the first operand determines the result. This optimization is also done in the modified coder, which means pass 6 will find no cases it can optimize.

3.14 New operator "pop"

Consider the following example:

```pascal
TYPE pointer = @mrecord
mrecord = RECORD
  X : INTEGER
  M : MODULE
PROCEDURE a
 ..
END MODULE
```
VAR pt : pointer

After the pointer is created, if we make a reference pt@.m.a, the code for the reference will emit the address for pt, and through indirection get the address of the record and add the field displacement to compute the address of pt@.m. Finally when the translation reaches "a", we discover that it is a procedure call. The translation just generates a procedure call to that a's code, but the useless address computation previously encountered is already emitted and certainly will be translated by Pas32. The way to eliminate this expression is to generate the new operator "pop". The "pop" operator means throw away the most recent expression. Pas32 pass 6 can do this by eliminating the useless expression altogether, so no code for it is generated. The "pop" operator is used in several cases besides the one shown by the example.

3.15 Processing local nonmanifest sized variables at routine entry and at declaration

Pascal requires that the size of variables declared within a routine should be presented at the routine entry point as a constant value, so that the activation record for that routine can be allocated at run time. A problem occurs when nonmanifest sized variables are declared as local to a routine, which Euclid allows. A new operator should be defined for use when compiling a nonmanifest array declaration
(This is not done in our translation). Pas32 will have to generate code to allocate stack space and compute and store an address in the activation record that will point to the allocated space. In the routine's activation record, we only have to reserve the space for the pointer. This means we still have a manifest variables size for the routine entry. Apparently, since we haven't reached the local variable declarations of this routine yet, space required for these pointers will have to be allocated by the Allocator pass. A new field giving variables size should be added to routine's Symbol/Type Table entry, such that Allocator can enter the information, and the Coder can emit the value without difficulty.

3.16 Handling return values in functions

Pas32 was a certain address for storing the result of a function, depending on the type of the return value. In Euclid, each function has an explicitly declared result variable which can be found from the function's Symbol Table entry. In the translation, a proper assignment is made to the result variable when returning from a function. Correction must be made to the result variable's address to match Pas32's definition. The Euclid Allocator pass again must take this responsibility.

3.17 Deleted features

The following are the features in Euclid that can not be implemented using Pas32 without drastic modifications. These
features are deleted from the translation.

Code Blocks

These allow the programmer to write a procedure body in assembly language.

Machine Dependent Record

These allow the programmer to specify bit layouts for record fields.

3.18 Size of nonmanifest parameters in routine formals and routine entry (not solved in this pass)

This problem is similar to problem that described in Section 3.15 Pas32's code generation passes expect a manifest total length for parameters, both in calls and in procedure headings. For any parameter that is passed by address or any nonmanifest-sized parameter, a one word field should be allocated to be the pointer to the actual parameter. In this way, the parameters length at routine entry will be manifest and can be compiled by a previous pass. The routine's Symbol Table Entry must have an added field to store the total parameters' size.

This differs from the translation of local variables, where declarations are processed after the routine entry. The formal parameter declarations are processed earlier than routine entry, which suggests that we can determine the parameters' size in the modified pass. Because too many semantic operations would be involved to do the job within
this pass, the former method is chosen.
CHAPTER 4: FINAL REMARKS

4.1 Previous work.

A translator has been written in Pas32 Pascal language, which translates Euclid Programs to Pascal Programs [7]. This has been used to translate all the passes of April Euclid to Pascal. The translated April Euclid compiler is then compiled by a slightly modified Pas32. This gives the April Euclid compiler written in Interdate 8/32 machine language, which can now be run on a local machine. It can translate Euclid language to PDP11 machine code.

Later, once this design is implemented, this procedure can be used to translate the modified April Euclid passes, when they are available. This, in combination with the modified code generation passes of Pas32 will give an April Euclid compiler which runs on and generates code for the 8/32. This new compiler can then be used to translate the portions of itself written in Euclid, thus completing the bootstrap.

4.2 Work accomplished by this project

This project produced the following design items:

Modified coder SSL (appendix A).

Specification for changes to

-- coder semantic mechanism (appendix B).

-- April Euclid Allocate pass (appendix C).

-- Pas32 pass 6 through 9 (appendix D).

4.3 Deferred features

The following Euclid features are not implemented by this
design. All of these are also not implemented by the April Euclid compiler.

Multiple Use Modules
FINALLY routines
BIND variables
Nonmanifest array bounds
Variant Records
Counted Collections
Zones
Parameterized types
FOR loop statements
RETURN WHEN statements
Nonmanifest constants
Sets
Long compares

4.4 Future work

As mentioned in the first paragraph, the final work of this project is to actually translate Euclid programs using the new compiler. The design work for the translation is mostly done. The next steps should be:

4.4.1 Write the modified semantic routines

The semantic routines used in this translation are mostly taken from the April Euclid coder. A Loop Label Stack mechanism has been added, and the Display Stack and Favor Stack have been taken out. Some extra fields have been added to the Extended Symbol Stack. Some other fields in
the Symbol/Type Stack of Euclid may not be required by our pass. The simplest way to write the modified routines is probably just to leave the useless things along and add new things. The necessary changes are specified in appendix B.

4.4.2 Implement new operators

New input operators for the Pas32 code generation passes have been introduced into the design. Passes 6 and 7 of Pas32 must be modified to handle these. Pass 8 and Pass 9 may also need minor modification. These are specified in appendix D.

4.4.3 Modify April Euclid Allocator

Some information that the Allocator presently can not supply is required by the modified coder. This concerns adding new fields to the Symbol Table entry and supplying them with manifest values. Assignment of displacements to fields of activation records must also be changed to match Pas32's conventions. The required changes are specified in appendix D.

4.4.4 Translate and compile the modified compiler

See 4.1

4.4.5 Extend the design

Many features have been deferred by this translation as indicated above. Unfortunately, these features also are the main features which make Euclid different from Pascal.
language. On the other hand, since the Euclid compiler itself hasn't implemented those yet, this extension work will be a long term goal.
APPENDIX A

Modified April Euclid Code S/SL
Program:

0InitialStacks % Initial all working area
@Pervasive % Skip token p140
=AType
=AIIdent
0PushIdentSymbol

[    | APrams: #EAbortParmsOnMainModule
    | * :
]

[    | APre: #EAbortPreMainOnModule
    | * :
]

[    | AForward: #EAbortForwardOnMainModule
    | * :
]

[    | APacked: #EAbortPackckedOnMainModule
    | * :
]

[    | AModule:
        OPushSymbolType % Type entry of main module
        @GenerateProgramHeader % Enter main program p105
        @ModuleBody % Single Used p110,e868
        @ModuleTrailer % Skip 'endmodule' p105
        @GenerateProgramTrailer % Call to main program p105
        .ecm % Output end
        OPoptype % Pop module type
    | * : #EAbortProgramMustBeModule
]

0PopSymbol % Pop module symbol
0CheckStacks % Check if stacks
; % are all popped
GenerateProgramHeader: % Generate program entry.
  % Deferred external program call.
  % Length of local variables are in module's type
  % Need to fixed allocator to contain the info.
  % Push extra symbol representing the scope
  % before main module.

  .skipfornesting
  .startlist
  .program
  OPushNewLabel
  OGenerateLabel
  OGenerateConstant[zero]
  OGenerateMainVarLength % Take from module type
  OPushNewSymbol
  OEnterProgramLevel % stack and enter program level
  OGenerateLevel
  OSwapSymbol
  ;

GenerateProgramTrailer: % Call to main program

  .return
  .program
  .resume fornesting
  .startlist
  .call
  .program
  OGenerateLabel
  OGenerateConstant[zero]
  OSwapSymbol
  OGenerateLevel
  OPopSymbol
  ;

ModuleTrailer:

  =AE ndModule
  [  
    |AE ndIdent:
        =AI dent
    |*
  ];
ModuleBody:

OSelectTypeAttribute[TySingleUse]
  [OTestTypeAttribute
   | On : OPushConstruct[CsingleUseModule]
   ]

  | * : #EDeferredMultipleUseModule
   OPushConstruct[CmultipleUseModule]
]

  | AModuleIdent :=Ident
  | *
]
@ImportClause `% Skip iden p140
@ExportClause `% Skip iden p145
@PushAppropriateCheck % p115
[ OIsChecked % Checks Check Stack
  | Yes:
  [ OIsModuleTypeInvariantPresent % Entered by previous % pass.
    [Yes: OPushNewLabel[C0Invariant] % Invariant label
     OEnterLabelTypeInvariantAddress
     OPopLabel
     ]
    | *:
  ]
|*::
]
@Declarations & p155, eu1101
@OptionalInitially % p120, eu926
[
  | AAbstraction:
    #EAbortAbstraction % Allocator removes Abstractions
  | *:
]
@OptionalInvariant % p115,
@OptionalFinally % eu958. Deferred. p120
OPopConstruct
OPopChecked
;
PushAppropriateCheck:

[  
|ANotChecked: OPushNotChecked  
|AChecked: OPushChecked  
|*: OPushCopyChecked  
]

OptionalInvariant: % Top symbol, top type : module

[  
|AInvariant:
    |OIschecked:
        |Yes: .skipfornesting %
    .startlist % Generate entry for
    .enter % Invariant
    .proc_mode % treat as proc mode
    OGenerateInvariantLabel % in module. see p110
    OGenerateConstant[zero] % No parameters
    OGenerateConstant[zero] % No variables
    OSwapSymbol % before module on top
    OGenerateOneUpperLevel % then top symbol
    OSwapSymbol % module on top
    @BooleanControlExpression % p410, e4654
    @ResolveSymbolBooleanTrue % jump falselab. truedef
    OPushNewLabel % Emit jump to
    .jump % true
    OGenerateLabel % label
    CGenerateSymbolBooleanShunt % False labels
    .errassert % New operator to PASCAL
    CGenerateNewLine % Emit run time error
    % and line number
    .labeldef % emit label definition
    OGenerateLabel % for true label, from
    OPCpLabel % top label stack
    OPCpSymbol % Boolean result
    .return % Emit exit invariant
    .proc_mode % It is proc mode.
  |*: @SkipExpression % p150
  |*:  
]

42
OptionalInitially:
% Pre: Top symbol: module; top type: module, initially
% Post: Top symbol and top type is module

[ 
|AInitially:
OPushInitiallyType   % Find initially routine type table
% index and push to type stack
@InitiallyRoutineDeclaration  % p125, e1707
OSwapType            % type are: initial routine, module
OSelectTypeAttribute[TySingleUse]
|
|On:
OSwapType  % type are: module, initial routine
.startlist  % Generate call to initially
call
.proc_mode
OGenerateRoutineLabel  % from initially type
OGenerateConstant[Zero]  % no parameters
OSwapSymbol  % Symbol before module on top
OGenerateOneUpperLevel  % from top symbol
CSwapSymbol  % Module on top
OSwapType  % Initially routine, module
|*:
  #EdeferredMultipleUseModule
]
OSwapType
OPopType  % Type are: module
|*:
];

OptionalFinally:  % Top type describes module

[ 
|AFinally:
  #EDeferredFinally
%  OPushFinallyType
%  @FinallyRoutineDeclaration
%  OGenerateFinallyEndComment
%  OPopType
|*:
];
InitiallyRoutineDeclaration: % Pre and Post:
% Type are: initially routine
% Symbol are: ... module

.skipfornesting
.GenerateInitiallyBeginComment
.PushNewLabel[CoRoutine] % For INITIALLT routine address
.EnterLabelTypeEntryAddress % Enter label to top Type Stack
.startlist
.enter
.GenerateLabel % from top table stack
.GenerateConstant[Zero] % parameters
.GenerateVarLength % The field to store the length of
% variables should be already added to
% Type Table. Allocator should enter
% the value.
.SwapSymbol % Symbol before module on top
.GenerateOneUpperLevel % from outer block
.SwapSymbol % module on top
.PopLabel % Initially routine label
.SwapSymbol % Symbol: module, routine before module
.RoutinePreAndPostAndImports % p135 e1926
.SwapSymbol % Symbol: routine before module, module
[
  |ABegin:  .PushConstruct[CoRoutine]
  .BlockbodyForRoutine % p230
  .GenerateConstructRoutineLabel % for embedded return
  .CleanUpBlock % p130, currently, just skip
  .SwapSymbol % Symbol: module,
% routine before module
  .CallPostIfNeed % p130
  .SwapSymbol % Symbol: routine before module,
% module
  .PushNewSymbol % Dummy symbol, see p132
  .CallInvariantIfNeed % p132
  .PopSymbol % Pop dummy symbol
  .return % return to initially call
  .proc_mode
  .CodeConstruct
  |ACode:  .CodeBlock % Code block is eliminated
  ]
.resumefornesting %
CleanUpBlock:

[ 0IsConstructCleanUpNeedForBlock  % c2137
  |Yes:  #EdeferredRobberVarsAndFinally
  |*: ]

CallPostIfNeed:  % top type: normal routine or initially routine
                 % top symbol: normal routine
                 %  or
                 %  ... module, routine before module

[ 0ChooseTypePostAddressKind
  |ConcKind:  % Do nothing
  |*:    .startlist  % Generate call to post routine
           .call
           .proc mode
           OGeneratePostLabelAddress  % Post address in
           OGenerateConstant[Zero]  % No parameters
           OGenerateOneUpperLevel  % from outer block
  ];
CallInvariantIfNeed:

% Type: ... module, routine
% Symbol: before module, module, routine or dummy
% Invariant will be called by Initially routine or
% normal routine,
% When called by normal routine, Symbol are: before
% module, module, routine.
% When called by Initially, Symbol are: before module,
% module, dummy symbol.
% The reason for pushing a dummy symbol inp125
% is for consistence in retrieving the scope information.

OSwapType           % Type are: ... routine, module
[OChooseTypeKind
 |TyModule:
   [OChooseTypeInvariantAddressKind
    |ConKind: % Do nothing
     |*: OSeselectSymbolAttribute[SyExported]
    [OTestSymbolAttribute
     |Cn: .startlist % Generate call to invariant
        .call .proc_mode
        OGenerateInvariantLabel % From module type
        OGenerateConstant[Zero] % No parameters
        OSwapSymbolLetTargetBeTop % Symbol: module, routine or dummy,
                                  % before module.
                                 OGenerateOneUpperLevel % From top symbol
                                 OSwapSymbolLetTopBeTarget % Symbol: before
                                                  % module, module,
                                                  % routine or dummy
     |*:]
   ]
|*:
]
OSwapType          % module, routine
;
RoutinePreAndPostAndImports:

% Top type: initially routine or normal
% routine
% Top symbol: .. module, routine before module
% or
% ... normal routine

@ImportClauses       % p140, just skip token
[
  |APre:  @AssertionExpression  % p410, e2384
  |*
];
[
  |APost:
  |[0Ischecked
  |[Yes:
    .Skipfromnesting  %
    OPushNewLabel[CoPost]  % Label are: post
    OEnterLabelTypePostAddress  % Post label enter Type
    .startlist  % Entry for a post
    .enter
    .proc_mode
    OGenerateLabel  % Post label, % from label stack
    OGenerateConstant[Zero]  % No parameter
    OGenerateConstant[Zero]  % No variable
    OGenerateOneUpperLevel  % From top symbol stack
    OPopLabel  % Pop post label
    @AssertionExpression  % p410
    .return  % From top Symbol
    @GenerateMode
    .resumeformnesting  %
  |*
];  @SkipExpression  % p150
|*
|  % No post
]
ImportClauses:

{ [ @Imports: [ AEndImport: >
  |*: @ImportItem % p140
] ] [*: > ] ];

ImportItem:

@Pervasive % p140
@BindingCondition % p140
=AIdent ;

Pervasive:

[ @Pervasive:
  |*: ];

BindingCondition:

[ @Var:
  @ReadOnly:
  @Const:
  |*: ] ;
ExportClause:

  [  
    |AExports: exportList  % p145
    |*: 
  ];

ExportList:

exportItem  % p145
{
  [  
    |AEndExports: >
    |*: exportItem  % p145
  ]
};

ExportItem:

  [  
    |AAssign:
    |AEqual:
    |APointer:
    |AIdent:
      [  
        |ATo: =AIdent
        |*: @WithClause  % p150
      ]
    |*: 
      @BindingCondition  % p140
      =AIdent
      @WithClause  % p150
  ];
WithClause:

[ [AWith:
   [{
     [AEndWith: >
     [*: @ExportItem % p145
     ]
   }
   [*: ]];

SkipExpression:

{ [ [AEndExpression: >
    [Subs: @SkipSubscripts % p150
    [AIn, ANotIn: @SkipExpression % p150
      [ATo: @SkipExpression % p150
      [*: ]
    [*: 0FlushInputToken

SkipSubscripts:

{ [ [AEndSubs: >
    [ASubs: @SkipSubscripts % p150
    [*: 0FlushInputToken

]};
Declarations:

{
    [
        [APervasive:
            |*:
        ]
    [
        AInline:
            |*:
    ]
    [AVar:
        @VariableDeclaration % 160
    ABind:
        #EDeferredbind
    AAssert:
        @AssertionExpression % p410
    AConst:
        @ConstantDeclaration % p190
    AType:
        @TypeDeclaration % p195
    AConverter:
        @ConverterDeclaration % p185, e1266
    APredicate:
        @PredicateDeclaration % p200, e1669
    AFunction:
        @FunctionDeclaration % p200
    |*:
        >
    ]
};
VariableDeclaration:

=AIIdent
OPushIdentSymbol
OPushSymbolType
@ResolveToLiteralType % p245
OPushCount[One] % Count the declared identifiers
[
  |AAnt: @SkipExpression % p150
  |*:
    [
      |
      |AIIdent: OPushIdentSymbol
        OIncrementCount
      |*: >
    ]
]

=AVarType
@TypeDefininition % p165, e1306 % For nonmefifest array,
@PushSizeSymbol % p345 % New operator should be
[0IsSymbolManifestValue % emitted here. together
  |Yes:
    
    |*:
      #EAbortNonManifestArrayBound % address[as a pointer],
  |*:
    OPopSymbol % Pop size % size. Code should be
  |
  |AInitial:
    OSwapType % Type surrounding declaration goes on top
[0ChooseTypeKind
  |TyModule:
    OSel ectTypeAttribute[ TySingleUse]
    [0TestTypeAttribute
      |On: % Good
      |*:
        #EDeferredInitialInMultipleUseTypes
    ]
    |TyProcedure, TyFunction: % Good
    |TyRecord, TyMRecord:
      #EDeferredInitialInMultipleUseTypes
  ]
OSwapType % variable goes on top
OPushCopyCount % save one for input to PASCAL
@AssignInitialValue % p162
% no initial value
{
[ OChooseCount
 ]One:
>
|*:
  OPopSymbol
  ODecrementCount

}
OPopCount         % Original count
OPopType          % Variable type
OPopSymbol        % The last variable symbol
};
AssignInitialValue:

[ OChooseCount ]

One:
OFixVarSymbolDataDescr
@EnterTypeInfoIntoSymbol   % p340
.pushadrr      % Declarad variable
@GenerateMode      % p415
OGenerateDisp
OGenerateLevel
@AssignableExpression % Symbol are: var., expr.  p360
@UnsignedIndication % p380
@ResolveBooleanJumpInSymbol % p405
OSwapSymbol      % Symbol: init value, variable
@GenerateRangeCheck   % p385
[ OChooseSymbolExtensionKind
 | SxScalar:
   .assign
   */:
   .copy
   OGenerateCopyLength   % Manifest only, see p375
]

OPopSymbol        % Variable Symbol
OPopSymbol        % Initial value
OPopType          % Variable type
OPopCount         % Pop the copy count

*/:

[ OChooseTypeCount ]

One:
OFixVarSymbolDataDescr   % p340
OEnterTypeInfoIntoSymbol   % p340
.pushadrr
@GenerateMode
OGenerateDisp
OGenerateLevel
OPopCount        % Pop copy count

*/:
OFixVarSymbolDataDescr
OEnterTypeInfoIntoSymbol   % p340
.pushadrr
@GenerateMode   % p415
OGenerateDisp
OGenerateLevel
ODecrementCount
OPopSymbol

]
% One count on counter stack; symbol is: variable;
% type is: variable type;
% Generate new operator to pascal, solve the initial
% statement for Var a, b, c, d: sometype := expression
% New operator 'Initialassign[number]', giving the
% counter value as the number, that tell the PASAL how
% many variable should be given assignment operation.
% 'Initialcopy[number]' replace the 'copy' operator.

@AssignableExpression  % The init expr on top,  p360
@ResolveBooleanJumpInSymbol  % p405
OPopSymbol  % Pop expr, variable on top
@GenerateRangeCheck  % for variable;  p385
[ OChooseSymbolExtensionKind  %
  |SxScalar,SxRange:
    .initialassign  % New operator
    OGenerateCountValue
  |*:  %
    .initialcopy  % New operator. Manifest
    OGenerateCountValue  % value only
    OGenerateCopyLength  % Manifest value only
]
OPopSymbol  % The last variable symbol
OPopType  % Variable type
]
OPopCount  % Original count
TypeDefinition:  % top type table entry is the type
  % e1306

[  
  APacket:  
  *:
  ]

  |AEnum:  @EnumeriatedTypeDefintion  % p710, e1368; just skip
  |AArray:  OPushIndexType  % Type are: array, index
  |  @ResolveToLiteralType  % p245
  |  @IndexType  % p170, e1358
  |  OPopType  % Index type
  |  OPushComponentType  % Type: array, component
  |  @ResolveToLiteralType  % p245
  |  @TypeDefintion  % p165
  |  OPopType  % Component

  |ARecord:  @RecordBody  % p170
  |  @RecordTrailer  % p710

  |AModule:  @ModuleBody  % p110
  |  @ModuleTrailer  % p105

  |ASET:  OPushBaseType  
  |  @ResolveToLiteralType  % p245
  |  @IndexType  % p170
  |  OPopType

  |AMDRecord:  
  |  #DELETEMachineDependentRecord

  |AMDModule:  
  |  #DELETEMachineDependentModule

  |ACollection:  @CollectionTypeDefintion  % p185

  |APointer:  
  |   =AIdent
  |   OPushInputToken  % Collection variable
  |*:
  |  @NameCrRangeTypeDefintion  % p180
]


IndexType:

[  
  |AEnum: @EnumeratedTypeDefinition % p170  
  |*: @NamedOrRangeTypeDefinition % p715, e1379 
];

EnumeratedTypeDefinition:

=AIIdent 
{  
  [  
    |AEndEnum: >  
    |* =AIIdent  
  ]
};

RecordBody:

@DeclarationsInRecord % p175 
[  
  |ACase: #EAbortVariantRecord % deferred 
  |*:  
];
DeclarationsInRecord:

{ [ 
  |A Var: 
    |AVariableDeclaration % p 160
  |A Const: 
    |AConstantDeclaration % p 190
  |A Pervasive: 
    |ACConst 
    |AConstantDeclaration % p 190
  |A Assert: 
    |AAssertionExpression % p 410.
  |*: > 
};

RecordTrailer:

=AEndRecord 
[ 
  |AEndIdent: 
    =AIdent
  |*: 
];
NameOrRangeTypeDefinition:

% e1379, A great difference from EUCLID.
% An expression assign to last or first,
% the last or the first symbol must be
% emitted first. For Manifest last or
% first, don't generate anything.
% Pre: type are array, index.
% symbol are array, array... array.
% Post: the same as above.

OPushTypeFirstSymbol   % p345    Symbol are: ... array, first
OFixConstSymbolDataDescr
[OIsSymbolManifestValue
 [OYes: OPopSymbol   % First
 [ [AIdent: OPushIdentSymbol   % Symbol: .. array, indnt
 [OChooseSymbolKind
 [SyType: OPopsymbol   % typename symbol
 [/*:
 OPopsymbol   % typename symbol
 @SkipExpression   % p150

 [/*:
 @SkipExpression   % p150

[ [AIdent: OPushIdentSymbol   % Symbol: array, first, ident
 [OChooseSymbolKind
 [SyType: OPopsymbol
 [/*: >>

 ]
]

% A variable or expression is assigned to first symbol
% Symbol: array, first, ident

OSwapSymbol   % Symbol: array, ident, first
.pushaddr    % for first address
.sccnst_mode
OGenerateDisp
OGenerateLevel
OSwapSymbol   % Symbol: array, first, ident
OPushSymbolType   % Type: array, index, ident
@VariableValueWithoutPush   % p250, ident already on top
@Expression   % e8900, p360
@TranslateAssign   % p375
]
% Starting translate last. Symbol: array. Type: array,index

=ATo
0PushTypeLastSymbol            % p345
0FixConstSymbolDataDescr
[0IsSymbolManifestValue
 |Yes:
 |   0PopSymbol               % Last
 @SkipExpression            % p150
 |*:
 .pushaddr                  % Emit last address
 0GenerateConstant[ sconst_mode ]
 0GenerateDisp
 0GenerateLevel
 @Expression                % p360
 @TranslateAssign           % p375
];
CollectionTypeDefinition: % e1518

[  
  |ACounted:  
    #EDeferredCountedCollections  
    [  
      |ACountMax:  
        @SkipExpression % p150  
      |*:  
    ]  
  |ACheckable:  
    #EWarningNoCodeToCheckCollections  
  |*:  
]  
=AVarType
OPushObjectType
@ResolveToLiteralType % p245
@TypeDefin Type % p165
OPopType
[  
  |AIn:  
    #EDeferredUserZones  
    =AIIdent  
    @SkipVariable  
  |*:  
];

ConverterDeclaration: % Just skip the converter declaration

% Converter should be already be solved previously  
=AIIdent
[  
  |AIIdent:  
    |AProcedure: #EDeferredConverterCfRoutine  
    |AFunction: #EDeferredConverterOfRoutine  
]  
=AReturns  
=AIIdent;
ConstantDeclaration:

=AIIdent
OPushIdentSymbol
OPushSymbolType
@ResolveToLiteralType
OPushCount[One]
{
  [AIIdent:
    OPushIdentSymbol
    OIncrementCount
  ]*: >
}
]

[AIConsType:
  ]*
]
@TypeDefination

[AVAlues:
  @ConstValueList % p195, Just skip value
  {
    [OChooseTypeCount
    One:
      OPopSymbol
    ]*
      OPopSymbol
    ]
  }
OPopCount
OPopType % Constant type
| AInitial:
  [ CIsSymbolManifestValue
    | yes:
      % Manifest scalar constant, No code emit
      @SkipExpression % p150
      @UnseignedIndication % p380
    { [ CChooseTypeCount
      | One:
        OPopSymbol
      >
      |*
        OPopSymbol
    ]
  }
  OPopCount
  OPopType % Constant type
| *:
  CPushCopyCount
  @AssignInitialValue % p162, Symbol, type and count
  % are popped
];
ConstValueList:

{ % manifest structure constant only.
  [ 
    |AEndValues: >
    |AValues:
      |ConstValueList % p195
      |*: 
      |SkipExpression % p150
      |UnsignedIndication % p380
  ]
};

TypeDeclaration:

=AIIdent
OPushIdentSymbol
OPushSymbolType
@ResolveToLiteralType % p245
[
  |APrams:
    #EAbortParameterizedTypeDeferred
    |*: 
];
[
  |APre:
    @AssertionExpression % p410
    |*: 
];
[
  |AForward:
    |*: 
      @TypeDefiniton % p165
] OPPopType
OPopSymbol
;
RoutineDeclaration: % e1669

=Alldent
OPushIdentSymbol
OPushSymbolType
OSelectTypeRoutineAttribute[TyExternal]
[OTestTypeRoutineAttribute
  |On:  @ExternalRoutineDeclaration % p200, e2012
  |*:  @InternalRoutineDeclaration % p200, e1689
]
OPopType % Routine
OPopSymbol
;

InternalRoutineDeclaration: % e1689, p200
% Top symbol, top type are routine

.skipfornesting
OGenerateRoutineBeginComment % Preserve from Euclid coder
[OChooseTypeForwardBodyAddressKind
  |CONoKind:  @VirginRoutine % p205
  |*:  @AlreadyForwardRoutine % p210
]
.resumefornesting
;

ExternalRoutineDeclaration:

{
  |AExternal: >
  |AModule, AModule:
    #ELiteralModuleAsExternalFormalType
    |*:  OFlushInputToken
};
VirginRoutine: % Not previous encountered with forward. e1769
  % Top type, symbol: routine

OPushNewLabel[ CoRoutine]
OEnterLabelTypeEntryAddress % Enter label represents routine
  % entry address to top type
  % stack and its type table
RoutineFormals % p215 e1876
  .enter
  GenerateMode % p415
GenerateLabel % From label stack
GenerateParamLength % Emit total length of parameters from
  % routine's Symbol Stack. Assume Euclid
  % allocator pass has added this new field
  % to routine's Symbol table entry. and
  % has entered the length information.
  % either parameters are manifest or
  % nonmanifest
GenerateVarLength % Use the same technique as
  % GenerateParametersLength

GenerateLevel

  [ |AReturns: FunctionResultDeclaration % e1913 p210
    |*
    ]
CallInvariantIfNeed % p132
RoutinePreAndPostAndImports % p135

  [ |AForward:
    OPushNewLabel[ CoForwardRoutine]
    BeginBodySplit % New operator to PAS32, with the label
    GenerateLabel % of the splitted body. inform PASCAL to
    % joint the body elsewhere.
    OEnterLabelTypeForwardBodyAddress % Enter label to type
    % stack and type table
    OPopLabel
  ]
  |AExternal:
    OAAbort % Already tested type external
  |*
  RoutineBody % p220, e1824
];
FunctionResultDeclaration:

=AIdent
OPushIdentSymbol
OPushSymbolType
@ResolveToLiteralType % p245
@TypeDefinitioin % p165
OPopType
OPopSymbol

AlreadyForwardRoutine:

.EndBodySplit % To join a beginbodysplit
OGenerateForwardBodyAddressLabel % Emit .deflabel also
[
 |AFoward, AExternal:
 |OAbort
 |
 ]
@RoutineBody % p220
RoutineFormals:

.startlist
[
  |AParms:
    @RoutineFormal    % p215
    {
      [  
        |AEndParms: >
        |*:   @RoutineFormal % p215
      ]
    }
  ]  % no formals
];

Routineformal:

@Pervasive          % p140
@BindingCondition   % p140
{
  [  
    |AIdent:
      OPushIdentSymbol  % Symbol and type: routine, parameter
      OPushSymbolType
      .parm
      @GenerateMode      % p415
      @GenerateContext   % p415
      OGenerateDisp
      OGenerateTypeLength
      % Assum allocator pass of Euclid compiler
      % has calculated the length and added to
      % formal's Symbol tables entry. either
      % manifest or nonmanifest
      OPopSymbol
      OPopType
    |*:   >
  ]
} =AParmType
@SkipTypeDefininition ;
RoutineBody % Called by virgineroutine. Already forward. % Body not external.

[ ]
|ABegin:
    |ImmediateTypeUpdate % make sure top type stack is rewritten
    % in the type table, incase a
    % recursive call needs the addr of
    % the present routine.
    OPushCnstruct[CoRoutine]
    @BlockBodyFcrRoutine % p230, e2088
    OGenerateCnstructRoutineLabel % For embedded return, if any
    @CleanUpBlock % Small EUCLID, skip p130
    @CallPostIfNeed % p130
    [OCnoseTypeKind
     |TyProcedure: @CallInvariantIfNeed % p132
     |TyFunction: % Function no side effects
    ]
    .return
    OSnarchAndGenerateReturnMode
        % Search Symbol Stack from
        % the second top entry, until
        % find a procedure or a
        % function symbol, return the mode
    OPopCnstruct
    |ACode: #EDeletedCodeBlock
    |AForward, AXternal: 0Abort
];
BlockBody:

@PushAppropriateChecked % p115
@Declarations % p155
{
  [  
    |AIIdent:
      @AssignmentOrCallStatement % p245
    |AExit:
      @ExitStatement % p246
    |AReturn:
      @EmbeddedReturnValue % p235
    |AAssert:
      @AssertionExpression % p410
    |ABegin:
      @BeginStatement % p247
    |AIf:
      @IfStatement % p235
    |ALoop:
      @LoopStatement % p246
    |AFor:
      @ForLoopStatement
    |ACase:
      @CaseStatement % p247
    |*: >
  ]
}@CleanupBlock % p130
0PopChecked;
BlockBodyForRoutine:

@PushAppropriateCheck % p115
@Declarations % p155
{
  [  
    |@If:
    |  @IfStatement % p235, e2316
    |@Assert:
    |  @AssertionExpression % p410, e2384
    |@Loop:
    |  @LoopStatement % p246, e2147
    |@Begin:
    |  @BeginStatement % p247, e2045
    |@Case:
    |  @CaseStatement % p247, e2411
    |@Ident:
    |  @AssignmentOrCallStatement % p245, e3645
    |@For:
    |  @ForLoopStatement % Not implemented yet
    |@Endbegin:
    |  % Do nothing, implicit return
  ]
  @OptionalEndIdent % p230
  @CleanUpBlock % p130
  OPopChecked;

OptionalEndIdent:

  [  
    |@EndIdent:
    |  [  
    |    | @*:
    |  ];
  ];
IfStatement:

@BooleanControlExpression % p410, e4654. Translate expr
@ResolveSymbolBooleanTrues % Generate False jump. p405, e4579
@BlockBody % THEN statements % p225
[
    |AElse:
      OPushNewLabel % Exit label
      .jump
      OGenerateLabel .deflabel
      OGenerateSymbolBooleanShunts % Generate false labels
      @BlockBody % ELSE statement p225 .deflabel
      OGenerateLabel % Exit label definition
      OPopLabel =aendif
    ]
  ]
OPopSymbol % Boolean result symbol ;

EmbeddedReturnStatement:

@HandleReturnStatementValueIfFunction % p240
[
    |AWhen:
      #2DeferredReturnWhen
      *:
    ]
@CleanUpForReturnStatement % p235
OPushNewLabel[CoReturn]
 .jump
OGenerateLabel
OMergeLabelIntoRoutineConstruct % find the first encounter % routine construct
OPopLabel;
CleanUp\ForReturnStatement:

[ \IsConstruct\CleanUp\NeededForReturn
 | Yes:
   \#\Deferred\Robber\VarsAndFinally
 | *:
 ];
ExplicitReturnStatementForEndOfRoutine:

@HandleReturnStatementIfFunction
[
  |AWhen:
  |  #2Deferred
  |  [*:
];

HandleReturnStatementIfFunction:  
  % copy expr in return statement to
  % result variable

[ OChooseTypeKind
  |TyFunction:
  |
    |AReturnValue:
      @PushResultVariableSymbol        % p240, e1972
      @AssignableExpression            % p360
      @TranslateAssign                 % p375
    |  [*:  % Do nothing, since result variable
        % should already be given value
    |TyProcedure:          % Do nothing
];

PushResultVariableSymbol:        % e1972

OPushTypeResultVariableSymbol
OPushSymbolType
@ResolveToLiteralType            % p245
OPushVarSymbolDataDescr          % p340
.pushaddr
@GenerateMode                     % p415
OGenerateDisp                     % Result variable should be already
OGenerateLevel                    % fixed to Pascal convention
OPopType                         % result type
ResolveToLiteralType: % e3407

{
  [OChooseTypeKind
   |TyInstance:
     [ IsTypeParameterized
       |Yes:
         #EDeferredparameterizetype
     |*:
       OPushOfType
       GSwapType
       OPopType
     ]
   |* >
  ]
};

AssignmentOrCallStatement:

@VariableAddress % p250, e2552
[
  |AAssign:
    @Expression % p360, e3660
    @TranslateAssign % p375
    |*:
      OPopSymbol % not assignment
];
LoopStatement: % Need a loop stack

OPushNewLabel % Push new label into loop label stack
.deflabel % for entering loop
OGenerateNewLabel % for exit
@PushNewLabel % p225
@BlockBody
OSwapNewLabel % Loop label: exit, enter
.jump
OGenerateNewLabel % Pop enter
OPopNewLabel % exit label definition
OPopNewLabel % Pop exit
=AEndLoop
;

ExitStatement:
% Exit statement is not allowed in a blockbody for % routine. It only allows in a beginstatement and % other statement within the loop

[
|AWhen:
@BooleanControlExpression % p410
@ResolveSymbolBooleanTrues % Emit false jump p405
.jump
OGenerateNewLabel % Exit label
OGenerateSymbolBooleanShunts % Emit false jump labels
|*:
.jump
OGenerateNewLabel
];
BeginStatement:

OPushConstruct[CBegin]
@BlockBody % p225
=AEndBegin
OPopConstruct ;

CaseStatement:

[|AConst, AVar, ABreadOnly: % not implement
|*: % Simple case statement
@AssignableExpression % p360
.case_expr
@CaseBody % p248
RearrangeCaseTable
  % Pre: labelstack: exit, else or errlabel
  % Sort the top case table, enter else
  % label or error label in missing slots
  % after sorted, insert else label or
  % error label as the first label in case
  % table. also makes available the min
  % and max value of case table labels
OPopLabel % Pop else label
.case_jump
GGenerateAndPopCaseTable
.deflabel
GGenerateLabel % exit label
OPopLabel
];
CaseBody:

OPushNewLabel % Exit label
.startlist
OPushNewCaseTable % A empty case table to top case table stack
{
  |
|A Labels:
  OPushNewLabel: % Label: exit, new
  .deflabel
  OGenerateLabel % Emit new
  OEnterVariantLabelCaseTable % enter above new label into case
  % table to be a new variant
  OPopLabel % Pop new. Label stack now is: exit
  @CaseLabel % p249
  |
  |AEndLabels:
  >
  |*: @CaseLabel % p249
|
} @BlockBody % p225
.jump
OGenerateLabel % exit label
@SkipCaselabel % just the extra token % p249
|*: |
|AEndCase: CFpushErrLabel % An error label which will cause % run time error
|AOtherwise: OPushNewLabel % Otherwise label
  .deflabel
  OGenerateLabel @BlockBody % p225
  OSwapLabel % otherwise, exit
  .jump
  OGenerateLabel % exit label
  CSwapLabel % exit, otherwise
  =AEndCase
|
> ];
CaseLabel:

@Expression % p360
  .pop % New operator to Pass32 pass 6
  [ % Delete the expression from Pass32
    |ATo:
      @expression
      .pop
      OEnterRangeCaseTable
      OPopSymbol
      |*:
        OEnterValueCaseTable % Enter case table the range of top
        % two symbols' value and labels
        % Pop one expr
        OPopSymbol % Enter case table the value of
        % top symbols' value
      ] % Pop evaluated expression
  ]

SkipCaseLabel:

@SkipExpression % p150
  [ %010
    |ATo:
      @SkipExpression % p150
      |*:
    ];
VariableAddress:

OPushIdentSymbol
OPushSymbolType
[ OChooseSymbolKind
  |SyVariable: OFixVarSymbolDataDescr  % See Euclid semantic
       @VarAddress    % p270
  |SyConstant: % Only scalar constant can be nonmanifest, and
               % its address can be referenced. Structure
               % constant should not be appeared here
       OFixedConstSymbolDataDescr % Become sydata
       @VarAddress    % p270
  |*:            @RoutineCallOrSytype % p255
]
@AssertSymbolNotFunctionOrConverter % p335 e2627
@EnterTypeInfoIntoSymbol   % For later assignment p340
OPoptype;

VariableValue:

OPushIdentSymbol
OPushSymbolType
[ OChooseSymbolKind
  |SyVariable: OFixVarSymbolDataDescr  % See Euclid Coder
       @VarValue     % p280
  |SyConstant: OFixedConstSymbolDataDescr % become sydata
       @ConstantValue % p290
  |*:            @RoutineCallOrSytype % p255
]
@AssertSymbolNotFunctionOrConverter % p335
@EnterTypeInfoIntoSymbol   % For later assignment p340
OPoptype;

VariableValueWithoutPush:

[ OChooseSymbolKind
  |SyVariable: OFixVarSymbolDataDescr  %
       @VarValue     % p280
  |SyConstant: OFixedConstSymbolDataDescr % become sydata
       @ConstantValue % p290
  |*:            @RoutineCallOrSytype % p255
]
@AssertSymbolNotFunctionOrConverter % p335
@EnterTypeInfoIntoSymbol   % p340
OPoptype;
RoutineCallOrSytype:

[ OChooseSymbolKind
 |SyProcedure: @ProcedureCall % p255 e3185
 |SyFunction: @FunctionCall % p255
 |SyType: @PossibleSetConstructor % p335
 |SyParameterizedType: @ParameterizedInstanceDefinition % p335
 |SyConverter: @ConverterCall % p257
];

ProcedureCall:

OSelectTypeRoutineAttribute[ TyBuiltin ]
[ OTestTypeRoutineAttribute
 |On: @NonBuiltinProcedureCall % p255
 |*: % No
]

FunctionCall:

OSelectTypeRoutineAttribute[ TyBuiltin ]
[ OTestTypeRoutineAttribute
 |On: @NonBuiltinFunctionCall % p260, e3299
 |*: @BuiltinFunctionCall % only ord, chr. p265
]

NonBuiltinProcedureCall:
% Pre and post: type and symbol are procedure

@Arglist % p262
 call
 .proc_mode
 GenerateRoutineLabel % In routine type, top type stack
 GenerateParamsLength % see VirginRoutine p205
 GenerateLevel
ConverterCall:

% Pre: Top type and symbol are for converter
% Post: Top type[unsolved] and symbol are for converted value
% No code generated
OPushParameterType
[ OChooseTypeKind
  |TyProcedure,TyFunction: % Get address of routine
    #EDeferredConverterOfRoutine
  %
  OPopType % Parameter type
  OPopSymbol % Converter
  OPushNullSymbol % Place holder till routine ident found
  =ASubs
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
%
NonBuiltInFunctionCall: % e3299

OPushTypeResultVariableSymbol
OPushSymbolType % Type of function result
OPushSymbolType % Extra copy to resolve
@ResolveToLiteralType % p245
[OChooseTypeKind
 | TyEnumerated,TySubrange,TySet,TyPointer,TyInteger,TyChar,
 | TyBoolean,TyStorageUnit:
 | TyArray,TyRecord,TyModule,TyMIDRecord:
 #EDeferredNonScalarFunctions
]
.Ofuncvalue % Result' mode
OGenerateMode % from resolved result variable
OGenerateType % Manifest value only
OGenerateSize % Alignment of the result variable
OGenerateAlign % New field in type.
OPopType % Resolved copy of result type
OSwapSymbol % Symbol: result, function
OPopType % Type: function
@Arglist % p262
.call 
.Ofunc_mode % See Appendix B, number 1
OGenerateLabelTypeAssress % Pop function symbol. Result
OGenerateParamsLength % symbol and result type now on top
OGenerateLevel
OPopSymbol ;
Arglist:

% Top symbol and top type is routine
% Manifest parameters only,
% For nonmanifest parameters, pass A/F record also.
.

.startlist
[
 |ASubs:
  OPushTypeFormalsLocator
    % Formals symbol entry on locator stack
  OPushCount[Zero] % Will be used to retrieve a particular
    % formal in locator stack
  |
  |
  |AEndSubs: >
  |
  |
  |*:  
  |
  |OPushCount
  |
  |OPushCountLocatorSymbol
    % A formal on top symbol stack
  |
  |OPushSymbolType
  |
  |OSelectSymbolAttribute[SyPassByValue]
  |
  |CTestSymbolAttribute
  |On:
    |
    | assignableExpression % Actual p360
  |[*:  
  |
  |OVariableAddress % Pass by var, actual is
  |  | UNSIGNEDINDICATION
  |  | % Just skip token p380
  |  | @PassActualTypeAERecordIfNeeded % if nonmanifest,
  |  | % deferred. p265
  |  | OPopSymbol % Pop actual,
  |  | .arglist % formal on top symbol
  |  | @GenerateNode % p415
  |  | CGenerateDisp
  |  | @GenerateContext % p415
  |  | CGenerateSize
  |  | @GenerateTypeRangeCheck % p263, Note: different
  |  | % from @GenerateRangeCheck
  |  | OPopSymbol
  |  | OPopType
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
GenerateTypeRangeCheck:

% Top Symbol and top type is for the symbol

[ ChooseTypeKind
  |TyEnumerated, TySubrange:
    OPushTypeFirstSymbol
    OPushTypeLastSymbol
    [ OIsSymbolManifestValue
      |Yes:
        OSwapSymbol
      [ OIsSymbolManifestValue
        |Yes:
          .range
          OGenerateSymbolValue
          OSwapSymbol
          CGenerateSymbolValue
        |*
        @NewOperatorForRangeCheck
      |*
      @NewOperatorForRangeCheck
    ]
  |*
  OSwapSymbol
  @NewOperatorForRangeCheck
  ]
OPopSymbol
OPopSymbol
|*
% No range check
];
BuiltinFunctionCall: % only chr and ord is implemented,
    % How PASCAL do it? This may not work.

[0IsRoutineSymbolChr
 |Yes:
    0PopSymbol     % Pop Chr
    =ASubs
  .startlist
  @AssignableExpression   % p360, Actual on top symbol
  =AEndSubs
  .arglist
    % The mode, disp, context, type and
  0GenerateChrMode  % length of a certain built in routine
  0GenerateChrDisp  % should be fixed and already known.
  0GenerateChrContext % Just delete these semantic routine
  0Generate_chrType % and output the actual values if
  0GenerateChrLength % found.
  .function
    .chr1
    .byte_type % byte
  0GenerateChrSize % Replace these two semantic routine
  0GenerateChrAlign % to output action if the value known
 |*:
[0IsRoutineSymbolOrd
 |Yes:
    0PopSymbol
    =ASubs
  .startlist
  @AssignableExpression   % p360
  .arglist
    % The mode, disp, context, type
  0GenerateOrdMode  % and length of a certain builtin
  0GenerateOrdDisp  % should be fixed and known
  0GenerateOrdContext % Just change these semantic
  0GenerateOrdType  % routine to the actual values if
  0GenerateOrdLength % found.
  .function
    .ord1
    .word_type    % ORD data type
  0GenerateOrdSize % Replace these two to output
  0GenerateOrdAlign % action if the value known
 |*:  % other not implement
]

0EnterSymbolKind[sysconstant] % Into top result symbol
0EnterResultTypeIntoSymbol    % From top type: builtin function
0PopType
0PushSymbolType    % result type
;

PassActualAllRecordsIfNeeded:

% Support for "Parameter" deferred
VarAddress: % e2732, Generate address reference
% for variable or constant.
% Symbol and type is variable or constant
[
  |ASubs: % Item is array or collection
  @AssertSymbolVariableOrConstant % p341, can not be const
  @AssertSymbolNotPackingInEffect % p335
  @ResolveToLiteralType % p245
  [OChooseTypeKind
    |TyArray: .pushaddr % Emit address of array
      .lconst_mode
      OGenerateDisp
      OGenerateLevel
      @ArraySubscription % p8540, emit pushind
      @ContinueVarAddress % p275
    ]
  ]
  |TyCollection:
    OPopSymbol % Pop collection symbol. Type remain
    @AssignableExpression % Symbol: pointer. p360
    -=AEndSubs % Type: collection
    @Pointing % p8545
    @ContinueVarAddress % p275
  ]
  |APointer: % Emit pushaddr, pushind but not pushvar
    @AssertSymbolVariableOrConstant % p341
    @AssertSymbolNotPackingInEffect % p335
    @ResolveToLiteralType % Pointer type, p245
    .pushaddr
    .lconst_mode % Pointer mode
    OGenerateDisp
    OGenerateLevel
    OPushTypeCollectionSymbol % Symbol: pointer, collection
    OPopType % Pop pointer type
    OPushSymbolType % Type: collection
    OPopSymbol % Symbol: pointer
    @ResolveToLiteralType % Collection type, p245
    @Pointing % p8545 emit pushind
    @ContinueVarAddress % p275
  ]
  |AField:
    @ResolveToLiteralType % p245
    .pushaddr
    @GenerateMode % p415
    OGenerateDisp
    OGenerateLevel
    @DotAddress % p300
  ]*
  % A simple variable without subscript, dot or point
  % or a nonmanifest scalar constant
  .pushaddr
  @GenerateMode % p415
  OGenerateDisp
  OGenerateLevel
]
};
ContinueVarAddress:  % more subscripting, pointing and dotting

[
  [A Subs:
    @$AssertSymbolVariableOrConstant  % p341
    @$AssertSymbolNotPackingInEffect  % 335
    @$ResolveToLiteralType  % p245
    [O ChooseTypeKind
      |TyArray:  @$ArraySubscripting  % p350
      |TyCollection:  % tycollection should not appear in here
        .pop  % since it make no sense
        @$PopSymbol
        @$AssignableExpression  % p360
        =$EndSubs
    ]
  ]
  [A Pointer:
    @$AssertSymbolVariableOrConstant  % p341
    @$AssertSymbolNotPackingInEffect  % p335
    @$ResolveToLiteralType  % Pointer type p245
    @$PushTypeCollectionSymbol  % symbol: pointer, collection
    @$PopType  % Pop pointer type
    @$PushSymbolType  % Type: collection
    @$PopSymbol  % Symbol: pointer
    @$ResolveToLiteralType  % p245
    @$Pointing  % Emit pushind
  ]
  [A Field:
    @$ContinueDotAddress  % p300
  ]
  [*
    >  % End Emit address reference for variable or
  ]
  % nonmanifest scalar constant.
};
VarValue:

[  |ASubs:  % Item is array or collection
   @AssertSymbolVariableOrConstant  % p341
   @AssertSymbolNotPackingInEffect  % p335
   @ResolveToLiteralType  % p245
   @ChooseTypeKind
   |TyArray:  .pushaddr  % Emit address of array
   .lconst_mode
   0GenerateDisp
   0GenerateLevel
   @ArraySubscription  % p8540, emit pushind
   @ContinueVarValue  % p285

   |TyCollection:
   0PopSymbol  % Pop collection symbol. Type remain
   @AssignableExpression  % Symbol: pointer, p360
   @EndSubs  % Type: collection
   @Pointing  % p8545
   @ContinueVarValue  % p285

  ]

|APointer:  % Emit pushaddr, pushind but not pushvar
   @AssertSymbolVariableOrConstant  % p341
   @AssertSymbolNotPackingInEffect  % p335
   @ResolveToLiteralType  % Pointer type, p245
   .pushaddr
   .lconst_mode  % Pointer mode
   0GenerateDisp
   0GenerateLevel
   0PushTypeCollectionSymbol  % Symbol: pointer, collection
   0PopType  % Pop pointer type
   0PushSymbolType  % Type: collection
   0PopSymbol  % Symbol: pointer
   @ResolveToLiteralType  % Collection type, p245
   @Pointing  % p8545  emit pushind
   @ContinueVarValue  % p285
|AField:
   @ResolveToLiteralType % p245
   .pushaddr
   @GenerateMode % p415
   OGenerateDisp
   OGenerateLevel
   @DotValue % p315
|*:
   [OChooseTypeKind
    |TyEnumerated, TySubrange, TyInteger, TyChar,
      TyPointer, TyBoolean:
       .pushvar
       OGenerateType
       .sconst_mode
       OGenerateDisp
       OGenerateLevel
   |*:
     .pushaddr
     .lconst_mode
     OGenerateDisp
     OGenerateLevel
   ]
};
ContinueVarValue: % more subscirping, pointing and dotting

[
  |ASubs:
    @AssertSymbolVariableOrConstant % p341
    @AssertSymbolNotPackingInEffect % p335
    @ResolveToLiteralType % p245
  |OChooseTypeKind
  |TyArray: @ArraySubscripting % p8540
            @ContinueVarValue % p285
  |TyCollection: % tycollection should not appear in here
    pop % since it make no sense
    OPcpSymbol
    @AssignableView % p360
    =AEndSubs
    @ContinueVarValue % p285
]

|APointer:
  @AssertSymbolVariableOrConstant % p341
  @AssertSymbolNotPackingInEffect % p335
  @ResolveToLiteralType % Pointer type p245
  OPpushTypeCollectionSymbol % symbol: pointer, collection
  OPpushSymbolType % Type: collection
  OPpushSymbol % Symbol: pointer
  @ResolveToLiteralType % p245
  @Pointing
  @ContinueVarValue % p285

|AField:
  @ContinueDctValue % p325
|*: [OChooseTypeKind % end of emit variable value
    |SyEnumerated, TySubrange, TyInteger, TyChar,
      TyPointer, TyBoolean:
      .pushind % Get value
      |*: % Do noting, structure keep its address
]
];
ConstantValue: % e2732, Generate value of a constant.
% Symbol and type is constant
% For nonmanifest scalar constant, treat as
% variable reference. Structure constant
% should be always manifest

[ASubs: % Item is array or collection
  @AssertSymbolVariableOrConstant % p341
  @AssertSymbolNotPackingInEffect % p335
  @ResolveToLiteralType % p245

[OChooseTypeKind
  |TyArray: .pushaddr % Emit address of array
    .lconst_mode
    .generateDisp
    .generateLevel
    @ArraySubscription % p8540, emit pushind
    @ContinueConstantValue % p295

|TyCollection:
% Tycollection should not appear here,
  OPopSymbol
  @AssignableExpression % p360
  =AEndSubs % Type: collection
  @Pointing % p8545
  @ContinueConstantValue % p295
]

|APointer: % Emit pushaddr, pushind but not pushvar
  @AssertSymbolVariableOrConstant % p341
  @AssertSymbolNotPackingInEffect % p335
  @ResolveToLiteralType % Pointer type, p245
  .pushaddr
  .lconst_mode % Pointer mode
    .generateDisp
    .generateLevel
    OPushTypeCollectionSymbol % Symbol: pointer, collection
    OPopType % Pop pointer type
    OPushSymbolType % Type: collection
    OPopSymbol % Symbol: pointer
    @ResolveToLiteralType % Collection type, p245
    @Pointing % p8545 % emit pushind
    @ContinueConstantValue % p295
FIELD:
  @ResolvedToLiteralType % p245
  .pushaddr
  @OGenerateMode % p415
  OGenerateDisp
  OGenerateLevel
  @DotValue % p315

[OChooseTypeKind % A scalar or structure constant
TyEnumerated, Ty subrange, TyInteger, TyChar,
TyPointer, TyBoolean:
  [OIsSymbolManifestValue
    |Yes:
      .pushconst
      OGenerateSymbolValue
    |]*:
      .pushvar
      OGenerateType
      @GenerateMode % p 415
      OGenerateDisp
      OGenerateLevel
    ]

|*:
  % Structure constant, always manifest
  % Previous pass of Euclid should fixed the
  % address of manifest structure constant
  % to match that of Pas32
  @EmitAddrAndValueForStructureConstant p420
  @EmitStructureConstantAddrReference p420

]:
ContinueConstantValue: % more subcscripting, pointing and dotting
    % Field of a structure constant

{
    [ ASubs:
        @AssertSymbolVariableOrConstant % p341
        @AssertSymbolNotPackingInEffect % p335
        @ResolveToLiteralType % p245
    ]

    [ OChooseTypeKind
        |TyArray: @ArraySubscripting % p8540
        |TyCollection:
            % This code may never be enter
            .pop % New operator to PASCAL
            OPopSymbol % Address of collection var is
            @AssignableExpression % p360 % not useful
            =EndSubs
    ]

|APointer:
    @AssertSymbolVariableOrConstant % p341
    @AssertSymbolNotPackingInEffect % p335
    @ResolveToLiteralType % Pointer type p245
    OPushTypeCollectionSymbol % symbol: pointer, collection
    OPopType % Pop pointer type
    OPushSymbolType % Type: collection
    OPopSymbol % Symbol: pointer
    @ResolveToLiteralType % p245
    @Pointing % Emit pushind

|AField:
    @ContinueDotValue % The same as DotValue
|*: [0ChooseTypeKind % End emit const value
|TyEnumerated, TySubrange, TyInteger, TyChar,
|TyPointer, TyBoolean:
| % A element of a structure constant.
| % should be always manifest
| .pop % New operator, address calculated is
| .pushconst % useless.
| OGenerateSymbolValue % from symbol

|*: % This is a structure constant inside a structure
% constant. The address reference for the inner
% structure constant is already emitted, but this
% address and its value have not been introduced
% to Pas32 yet. To reference a structure constant
% in Pas32, it must first know the address and
% the value of that constant by using the operator
% "lit_const", then the reference can be made by
% "pushaddr" operator. The problem encountered
% here is not solved

@InnerStructureConstantValue % p420

] >

]}

}];
DotAddress: % first call on dotting

[|AIdent:
  0PushIdentSymbol % Symbol: preceding dot, after dot
  0SelectSymbolAttribute[SyTypeActure]
  0TestSymbolAttribute
  |On: @DctTypeActual % Deferred p300
  |*: % Field of record or module
    [0ChooseSymbolKind
      |SyVable:
        @DotVariableAddress % p305
      |SyConstant:
        @DotContinueConstant % p302
        @ContinueVarAddress % p275
      |*: @DotRoutineAndOther % p353
    ]
  ]
|AStandardComponent:
  @DotStandardComponent % Won't occurred, p353
];

DctTypeActure:

   #EDeferredParameterizedTypes;

ResolveExtendParameter:

   % No action, since parameterized type are deferred
ContinueDotAddress: % for address reference

[|
  |AIdent:
  |OPushIdentSymbol
  |OSelectSymbolAttribute[SyTypeActual]
  |OTestSymbolAttribute
  |On: @DotTypeActual % Deferred p300
  |*: [OChooseSymbolKind
        |SyVarivale:
        |@DotContinueVariableAddress % p310
        |SyConstant:
        |@DotContinueConstant % Scalar nonmanifest
        % constant reference. p302
        |*:
        |@RoutineCallOrOther % p352
  |]
  |AStandardComponent:
  |@DotStandardComponent % p353, won't occurs
];

DotContinueConstant:

% Scalar nonmanifest constant only
% It is address reference
@ResolveExtendParameter % Deferred p300
OFixConstSymbolDataDescr
OPcpType % Preceding dot
OPushSymbolType % Unresolved type following dot
OSwapSymbol
OPopSymbol % Preceding dot
.field
OGenerateDisp
;
DotVariableAddress: % Pre: symbols are: item before dot, variable
top type is unresolved type before dot
% Post: symbols are: the same as above
top type is unresolved type after dot

@ResolveToLiteralType   % p245
[0ChooseTypeKind
[0TyModule:
  I0SelectTypeAttribute[TySingleUse]
[0TestTypeAttribute
  J0n: % Type before dot is a module, does not add address
   .pop
   0PopType       % Type before dot
   0PushSymbol1Type
   0FixVarSymbolDataDescr
   0SwapSymbol
   0PopSymbol     % Pop module
  @VarAddress     % p270
  >>
] *
]*:
]
]
% item before dot isn't a module

0PopType                  % Pop record type
0PushSymbol1Type          % Type of item following dot
0FixDotVarSymbolDataDescr %
.field
0GeneratedDisp
0SwapSymbol
0PopSymbol                 % Symbols: After dot, record
0PopSymbol                % Pop record symbol
@ContinueVarAddress       % p275
;
DotContinueVariableAddress:

ResolveToLiteralType % p245
[OCchooseTypeKind
 |TyModule:
 OSelectTypeAttribute[TySingleUse]
 [OTestTypeAttribute
 |On:
   #ErrorModuleInRecord
   .pop % Multiple used
   OPopType % module is not allowed, so
   OPushSymbolType % record.module.variable should
   OFixVarSymbolDataDescr % not be appeared.
   OSwapSymbol % If multiple used module is allowed
   OPopSymbol % the allocate pass should be fixed
   @VarAddress % let the access method be the same
   >> % as record's field. if this is true
 |]*:
] % the following should be enough.
]*:
]

% Item before dot is not module
OSwapSymbol % Symbol: After dot, Before dot
OPopSymbol % The address before dot has already been generated
OPopType % Type before dot
OPushSymbolType % Type after dot
OFixDotVariableSymbolDataDescr .field
OGenerateDisp
DotValue:       % first call on dotting value

[  
  |ADecl:  
    OPushIdentSymbol  
    SelectSymbolAttribute[SyTypeActure]  
    CTestSymbolAttribute  
    |on:  @DotTypeActual      % Deferred. p300  
    |*:  % Field of record or module  
    [CChooseSymbolKind  
      |SysVariable:  
        @DotVariableValue % p320  
      |SyConstant:  
        OSwapSymbol       % Symbol: constant, before dot  
        OPopSymbol        % Symbol: constant  
        @ResolveTcliteralType % Type before dot p245  
        @ResolveExtendedParameter % Deferred p300  
        OFixConstSymbolDataDescr  
        OPopType           % Type before dot  
        OPushSymbolType    % Const type  
        .field  
        OGenerateDisp  
        @ContinueConstantValue % p295  
      |*:  @DotRoutineAndOther % p353  
    ]  
  ]  
  |AStandardComponent  
    @DotStandardComponent % p353  
];
DotVariableValue: % Pre: symbols are: item before dot, variable
top type is unresolved type before dot
% Post: symbols are: the same as above
top type is unresolved type after dot

@ResolveToLiteralType % p245
[0ChooseTypeKind
]TyModule:
OSelectTypeAttribute[TySingleUse]
[0TestTypeAttribute
]On: % Type before dot is a module, does not add address
   .pop
   OPopType % Type before dot
   OPushSymbolType
   OFixVarSymbolDataDescr
   OSwapSymbol
   OPopSymbol % Pop module
   @VarValue % p280
>>
]*:
]
]*:
]

% item before dot isn't a module

OPopType % Pop record type
OPushSymbolType % Type of item following dot
OFixDotVarSymbolDataDescr %
.field % after dot
OGenerateDisp
OSwapSymbol % Symbols: After dot, record
OPopSymbol % Pop record symbol
@ContinueVarValue % p285
ContinueDotValue: % for address reference

[  
|AIdent:
 OPushIdentSymbol
 OSelectSymbolAttribute[SyTypeActual]
 [OTestSymbolAttribute
  |On: @DotTypeActual % Deferred
  |*: [OChocseSymbolKind
   |SyVarivale:
    @ContinueDotVariableValue % p330
  |AyConstant:
    OSwapSymbol % Symbol: const, before dot
    OPopSymbol % Symbol: const
    @ResolveToLiteralType % Type before dot p245
    @ResolveExtendedParameter % Deferred
    OFixConstSymbolDataDescr
    OPopType % Type before dot
    OPushSymbolType % Constant type
     .field
    OGenerateDisp
    @ContinueConstantValue % p295
  |*: @OoutineCallOrOther % p352
 ]
]

|AStandardComponent:
 @DotStandardComponent % p353
];
ContinueDotVariableValue:

ResolveToLiteralType % p245
[ OChooseTypeKind
 |TyModule:
 | OSelectTypeAttribute
 | OTestTypeAttribute
 | On: % Now, for a record.module.variable;
 | #ErrorModuleInRecord
 | .pop
 | OPopType % Doesn't add module address
 | OPushSymbolType
 | OFixVarSymbolDataDescr
 | OSwapSymbol
 | OPopSymbol
 | @VarValue % p280
 | >>
 | *:
 | [*:
 | ]
 | ]

% Item before dot is not module
OSwapSymbol % Symbol: After dot, Before dot
OPopSymbol % The address before dot has already been generated
OPopType % Type before dot
OPushSymbolType % Type after dot
OFixDotVariableSymbolDataDescr
.field
OGenerateDisp
@ContinueVarValue % p285
AssertSymbolNotFunctionOrConverter:

[ 0ChooseSymbolKind
    |SyVariable, SyConstant, SyType, SyParameterizedType,
    |SyProcedure:
        |SyFunction, SyConverter: OAbort
];

PossibleSetConstructor:

% See Euclid Coder 3433
[
    |ASubs: #EDeferredPowerSets
    |*:
];

ParameterizedInstanceDefinition: % e1546

#EDeferredParameterizedTypes
[ 0ChooseTypeKind
    |TyParameterizedType:
    |*: OAbort
];

AssertSymbolNotPackingInEffect: % e2647

0SelectSymbolExtensionAttribute[ SxPackingInEffect ]
[ 0TestSymbolExtensionAttribute
    |On: OAbort
    |*:
];
EnterTypeInfoIntoSymbol:
% End of variable has been found. Enter type
% information into symbol entry to be used
% for handling expressions and assignments
% e2657
[0ChooseSymbolKind
 |SysVariable, SysConstant:
 | OEnterSymbolKind[SyData]
 |@ResolveToLiteralType % p245
 |[0ChooseTypeKind
 | TyCollection: % No size
 |*: @PushSizeSymbol % p345
 | OEnterSymbolValueLeftSymbolSize
 | OPopSymbol % size
 ]
[0ChooseTypeKind
 |TyEnumerated, TySubrange: % for later
 | OEnterSymbolExtensionKind[SxRange] % range check
 | OEnterSymbolExtensionTypeFirstSymbol
 | OEnterSymbolExtensionTypeLastSymbol
 | OSelectSymbolExtensionAttributeName[ SxPackingInEffect ]
 | OTestSymbolExtensionAttributeName
 | On: % Symbol is packable cmpt of packed array
 | OPushSymbolManifestValue[One]
 | OEnterSymbolValueLeftSymbolSize
 | OPopSymbol
 |*: ]
|0IsTypeUnsigned
 |Yes:
 | OSelectSymbolExtensionAttributeName[ SyUnsigned ]
 | OTurnOnSymbolExtensionAttributeName
 |*: ]
|TyChar, TyBoolean:
 | OEnterSymbolExtensionKind[ SxScalar ]
 | OSelectSymbolExtensionAttributeName[ SxPackingInEffect ]
 | OTestSymbolExtensionAttributeName
 | On: % Symbol is packable cmpt of packed array
 | OPushSymbolManifestValue[One]
 | OEnterSymbolValueLeftSymbolSize
 | OPopSymbol
 |*: ]
|0IsTypeUnsigned
 |Yes:
 | OSelectSymbolExtensionAttributeName[ SyUnsigned ]
 | OTurnOnSymbolExtensionAttributeName
 |*: ]
#DeferredSet
OEnterSymbolExtensionKind[ SxScalar ]
OSelectSymbolExtensionAttribute[ SyUnsigned ]
CTurnOnSymbolExtensionAttribute

|TyInteger:
OEnterSymbolExtensionKind[ SxScalar ]
|TyArray, TyRecord, TyModule, TyMDRecord:
OEnterSymbolExtensionKind[ SxArrayRecordModule ]

|TyCollection:
#EDeferredCollections
|TyProcedure, TyFunction, TyConverter,
TyParameterizedType, TyInstance:
OAbort

|SyType, SyParameterizedType, SyProcedure:
% no type need, it never occur in expressions
|SyFunction, SyConverter:
OAbort

AssertSymbolVariableOrConstant:

[ OChooseSymbolKind
  |SyVariable, SyConstant: % Good
  |*: OAbort
];
PushTypeLastSymbol:

OPushTypeLastSymbol
OFixConstSymbolDataDescr

PushTypeFirstSymbol:

OPushTypeFirstSymbol
OFixConstSymbolDataDescr

Pointing: % Pre: symbol > pointer; type > resolved collection type
% post: symbol > object; type > unsolved object type

OPeEnterObjectIntoSymbol
OFixType
OPopType
OPushSymbolType
.pushind
OGenerateType

PushSizeSymbol:

[ OChooseTypeKind
|TyInstance:
  OPushTypeSizeSymbol % e8629, Instance size as a symbol push
  @ResolveExtendedParameter
  OFixConstSymbolDataDescr % into symbol stack
  [ CISSymbolManifestValue
  |Yes:
  |*: OPopSymbol
  |*: @ResolveToLiteralType % p245
  OPushTypeSizeSymbol
  @ResolveExtendedParameter
  OFixConstSymbolDataDescr
  ]
|*: OPushTypeSizeSymbol
  @ResolveExtendedParameter % e3398, Deferred
  OFixConstSymbolDataDescr
];
ArraySubscripting: % Pre: symbol: array; type: resolved array type
    % Post: symbol: component; type: unsolved component
  @EnterComponentTypeIntoSymbol % Change symbol array to component
  @PushIntType
  @ResolveToLiteralType % Enum or subrange p245
  @PushTypeLastSymbol % p345
  @PushTypeFirstSymbol % p345
  @PopType % Pop index type
  @PushComponentType
  @PushCopyType   % Type are: array, comp, comp
  @PushSizeSymbol % Symbol: comp, last, first, size of comp p345
  [0伊斯SymbolManifestValue
   |yes:
  ];
];
@ResolveToLiteralType % To solve the problem, push address of every
 % non manifest into PASCAL, and emit new operator
 % into PASCAL to resolve it. see SubscriptingNonManifest
 % Type are: array, unsolved comp, resoled comp, array.
 @ResolveToLiteralType % Post: type are: unsolved comp, resolved comp, array. p245
 @ReplaceIfPacked       % Pop array type
 @PopType               % Type are: unsolved comp
 @AssignableExpression  % p360
 @UnsignedIndication    % p380
 @PopSymbol             % Pop expr
 =EndSubs
 @SolvingSubscripting   % p356
 @PopSymbol             % Pop last, first and size
 @PopSymbol             %
 @PopSymbol             % Symbol are: comp
 ;
ReplaceIfPacked:       % Pre: type: array, comp, resolvedcomp
    % Post: comp, resolvedcomp, array

0SelectTypeAttribute[TyPackable]
[0TestTypeAttribute
  |on:
];
% Type are: unresolved comp,
OSwapTypesMakingTargetBeTop % array, resolved comp
OSelectTypeAttribute[Typackable]
[0TestTypeAttribute
 |on: CPopSymbol
 OPushSymbolManifestValue
 CTurnOnPackingInEffectSymbol
 |*: ]
 |*: OSwapTypesMakingTargetBeTop
];
DotRoutineAndOther:

[ @ChooseSymbolKind
    @SyType: % p352
    @DotType
    @SyParameterizedType: % p352
    @DotParameterizedInstanceDefinition
    @SyProcedure,SyFunction: % p355
    @DotRoutineCall
    @SyConverterCall: % p355
    @DotConverterCall
    @*:
    OAbort
];

DotType: % Pre: Symbol are: .. module value, type ident
         %   Top type is unresolved type preceding dot
         % Post: Symbols are : type ident
         %   Top type is for top symbol [type ident]

@ResolveToLiteralType
OPopType % of item before dot
OPushSymbolType % of new type ident
@PossibleSetConstructor % p335 Deferred
OSwapSymbol % Symbol are: .. type ident, module value
OPopSymbol % Pop module value
;

DotParameterizedInstanceDefinition:

% Pre: Symbol are: module value, param type ident;
%   Top type is module
% Post: Symbol are param type ident;
%   Top type is for defined instance
@ResolveToLiteralType
@ParameterizedInstanceDefinition % p356 Deferred
OSwapSymbol
OPopSymbol
;
DotStandardComponent:  % Pre: Top symbol and top type are for item before dot; top type is unresolved % post: Top symbol and top type are for standard function result, top type % is unresolved.
% Treat as Standard Function Call in Pas32

#EWarningTemporaryDeferred
.pop % Pop the previous address emitted
[OChooseStandardComponent
|CSucc:
|OPopSymbol % Pop the symbol before dot
|=ASubs
.|startlist
@Assign ableExpression
|=AEndSubs
.|arglist
.|succ1 % Pas32' standard num
|OGenerateSuccType % All these three should
|OGenerateSuccSize % be known previously
|OGenerateSuccAlign

|OPushCopyType % copy to be resolved
@ResolveToLiteralType
[OChooseTypeKind
|TySubrange: % Type of value is parent type
|OPushCfType
|OSwapTypes
|OPopType
|OSwapType % Type are: result type, unsolved expr type
|*
]
OEnterSymbolKind[SyConstant]

OPopType % Type are: ... result type
|CPred: |
|OPopSymbol % Pop the symbol before dot |
| = ASubs |
| .startlist |
| @AssignableExpression |
| = AEndSubs |
| .arglist |
| .pred1 % Pas32' standard num |
| OGeneratePredType % All these three should |
| OGeneratePredSize % be known previously |
| OGeneratePredAlign |
| OPushCopyType % copy to be resolved |
| @ResolveToLiteralType |
| [OChooseTypeKind |
| |TySubrange: % Type of value is parent type |
| | OPushOfType |
| | CSwapTypes |
| | OPCpType |
| | OSwapTypes % Type are: result type, |
| | % unresolved expression type |
| |] |
| ] |
| OEnterSymbolKind[ SyConstant ] |
| OPopType % Type are: ... result type |
| |CORD: |
| #EDeleteCORD |
| | |
| | *: |
| | #EDefferedStandardComponents |
| |] |
DotRoutineCall:  % Pre: Symbol are: module value, routine
%   Top type is for module value
% Post: If procedure then symbol and type are for
%   procedure indent
%   else symbol is result value, top type is
%   for unresolved result
% Does not support parameterized module

@ResolveToLiteralType
OSelectTypeAttribute[TySingleUse]
[ OTypAttribute
  |on: - pop  % No need the address previous emitted
  |*: #EDeferredMultipleUseModules
]
OOpqType  % Module type
OPushSymbolType  % Routine type
[ OChooseSymbolKind
  |SyProcedure:
    @NonBuiltinProcedureCall
  |*:
    @NonBuiltinFunctionCall
]

DotConverterCall:  % Pre: Symbol are: module value, converter
%   Top type is for item before dot
% Post: Symbol are: converter value
%   Top type [unsolved] is for converted value

OOpqType  % of item before dot
OPushSymbolType  % converter type entry
@ConverterCall
OSwapSymbol
OOpqSymbol
% Pop module value


SolvingSubscripting:  % Symbol are: comp, last, first, size
    % Size is manifest

OSwapSymbolsMakingTopBeTarget % Symbol: comp, size, last, first
    [OIsSymbolManifestValue  % First
        |yes: OSwapSymbol % Symbol: comp, size, first, last
            [OIsSymbolManifestValue  % Last
                |Yes: OSwapSymbol % Symbol: comp, size, last, first
                    @SubscriptingAllManifest % p356
                |*:  OSwapSymbol % Symbol: comp, size, last, first
            ]
        |*:  @SubscriptingNonManifest % p356
    ]

SubscriptingAllManifest:  % Pre: symbol are comp, size, last, first

 .index
OSwapSymbolsMakingTargetBeTop  % Symbol: comp, last, first, size
OSwapSymbol % Symbol: comp, size, first, last
OSwapSymbolsMakingTargetBeTop % Symbol: comp, first, last, size
OSwapSymbol % size

SubscriptingNonManifest:  % Pre: Symbol: comp, size, last, first
    % New operator 'ADDRINDEX' to
    % PASCAL, PASCAL pass 6 should
    % tell which one of min, max and
    @GeneratePushvarIfNonmanifest % size is nonmanifest by finding
OSwapSymbol % token 'pushvar' and 'pushconst'.
@GeneratePushvarIfNonmanifest % Size [length] of the component
OSwapSymbolsMakingTargetBeTop % should be manifest.
.pushconst
OSwapSymbolsMakingTargetBeTop % For size
OSwapSymbol % size
ADDRIINDEX

;
GeneratePushvarIfNonManifest:

[ IsSymbolManifestValue
  | Yes:
    .pushconst
    CGenerateSymbolValue
  |*:
    .pushvar
    .sconst_mode   % value should be integer
    CGenerateDisp
    CGenerateLevel
];

AssignableExpression:                     % e3672

@Expression                                     % p360
@ResolveBooleanJumpsInSymbol

Expression:                      % e3660

{ [ [AEndExpression:
    >
  |*:
    @ExpressionItem  % p365, e3782
]
};
ExpressionItem:                  % e3782

[  
|AIdent:              
  @VariableValue    % p250  
|ANumber:          
  .pushconst       
  CGenerateLiteralNumberSymbol  
  OPushLiteralNumberSymbol   
|AChar:             
  .pushconst       
  CGenerateLiteralCharSymbol  
  OPushLiteralCharSymbol   
|AMDChar:          
  #EDeleted        
|AString:          
  % No Symbol table entry for the literal,  
  % Not solved in this translation  
  @SolvingStringConstant   % p420  
|*:                 
  @Operator         % p370  
];
OPERATOR: % e3883

[
| AMinus:
  @TranslateMinus % p375
| AParens:
| ADD:
  @UnsignedIndication % p380
  @TranslateAdd % p380
| ASubtract:
  @UnsignedIndication % p380
  @TranslateSubtract % p380
| AXor:
  @SetOpIndication % p390
  @TranslateXor % e4916, Deferred. The % same as 'not_equal'
| AMultiply:
  @TranslateMultiply % p390
| ADiv:
  @TranslateDivid % p395
| AMod:
  @TranslateMod % p395
| AInfixCompare:
  @ResolveBooleanJumpsInSymbol % p405
| AEqual:
  @UnsignedOrSetOpIndication % p375, just skip token
  @ResolveBooleanJumpInSymbol % p405
  @TranslateEqual % p395
| ANotEqual:
  @UnsignedOrSetOpIndication % Emit EQ, NOT p375
  @ResolveBooleanJumpInSymbol % p405
  @TranslateEqual % p395
  .not
  @TranslateNot % p405
| ALess:
  @ResolveBooleanJumpInSymbol % Emit GE, NOT p405
  @TranslateEqual % p395
  .not
  @TranslateNot % p405
| ALessEqual:
  [ASetOp:
    @TranslatePowerSetLessEqual % Deferred p420
  | *:
    @ResolveBooleanJumpInSymbol % p405
    @TranslateGreater % p400, Emit GT, NOT .not
    @TranslateNot % p405
]
AGreater:
  @ResolveBooleanJumpInSymbol % p405
  @TranslateGreater % p400
AGreaterEqual:
  [ASetOp:
   @TranslatePowerSetGreaterEqual % Deferred p420
  ] *
   @ResolveBooleanJumpInSymbol % p405
   @TranslateGreaterEqual % p400
AIn:
  @ResolveBooleanJumpInSymbol % p405
  @TranslateIn % EDeferred, e4893
ANotIn:
  @ResolveBooleanJumpsInSymbol
  @TranslateIn
  .not
  @TranslateNot % p405
  @UnsignedIndication % p380
ANot:
  .not
  @TranslateNot % p405
AInfixOr:
  .not
  @TranslateInfixOr % p400, e4586
AOr:
  @TranslateOr % p400, e4586
AInfixAnd:
  @TranslateNot
  @TranslateInfixOr % p405
  @TranslateInfixOr % p400, e4586
AAnd:
  @TranslateOr % p400
  @TranslateNot % p405
AInfixImply:
  @TranslateNot
  @TranslateInfixOr % Same as infixand p405
  @TranslateInfixOr % p400, e4586
AImply:
  @TranslateOr % Same as aor. p400
};
TranslateAssign:  % Bit assign should not appear

@ResolveBooleanJumpInSymbol  % p405
OSwapSymbol  % Symbol: expr, left hand side variable
@GenerateRangeCheck  % p385
[0ChooseSymbolExtensionKind  %
  |SxSet, SxArrayRecordModule:  % For manifest length only
    .copy  % If nonmanifest array. need to
    OGenerateCopyLength  % Emit new operator as follow
    |*:  % push var[A/E record] "newcopy"
  ]
*assign

OGenerateType  % Assign type, Emit from top symbol
OPopSymbol  % left hand side variable
OPopSymbol  % expr

UnsignedOrSetOpIndication:

[
  [AUnsigned:  %
   ASetOp:  %
  ];

TranslateMinus:

@SkipUnsigned  % p380
.neg
OGenerateOperatorType  % From top symbol
;
UnsignedIndication:

[  |AUnsigned:
  |*:
];

SkipUnsigned:

[  |AUnsigned:
  |*:
]

TranslateAdd:

[  |ASetOp:  @TranslatePowerSetAdd % e4904, EDeferred p420
  |*:
  @TranslateNumericAdd % P 380
];

TranslateNumericAdd:

@SkipUnsigned % p380
  .add
  OPopSymbol
  OGenerateOperatorType
;

TranslateSubtract:

[  |ASetOp:
  @TranslatePowerSetSubtract % p420
  |*:
  @SkipUnsigned % p380
  @TranslateNumericSubtract % p390
];
GenerateRangeCheck:
  % Introduce new operator into PASCAL If range are
  % manifest, New operator named 'varrange', PASCAL
  % should treat the range as variable address when
  % encounter the operator.

[0ChooseSymbolExtensionKind % See modified semantic routine
 | 5xRange: % "OEnterSymbolExtensionKind"
  OPushExtensionTypeFirstSymbol % from top symbol
  OPushExtensionTypeLastSymbol
  [0IsSymbolManifestValue
   |Yes:
     OSwapSymbol % Symbol: variable, last, first
     [0IsSymbolManifestValue
      |Yes:
        .range % Manifest
        0GenerateSymbolValue % First
        OSwapSymbol % Symbol: variable, first, last
        0GenerateSymbolValue % Last
      |*:
        @NewOperatorForRangeCheck % p387
     ]
   |*:
     OSwapSymbol % variable, last, first
     @NewOperatorForRangeCheck % p387
  ]
  0PopSymbol % last
  0PopSymbol % first
|*
  % No range check
];
NewOperatorForRangeCheck: % pre: symbol: variable, last, first
% post: symbol: variable, first, last

[ OIsSymbolManifestValue % for first
 |Yes:
  .pushconst
  OGenerateSymbolValue
  OSwapSymbol
 [ OIsSymbolManifestValue % for last
  |Yes:
   .pushconst
   OGenerateSymbolValue
  |*:
   .pushvar
   OGenerateType
   @GenerateMode
   OGenerateDisp
   OGenerateLevel
   ]
 ]
 |*:
  .pushvar
  OGenerateType
  @GenerateMode
  OGenerateDisp
  OGenerateLevel
  OSwapSymbol
 [ OIsSymbolManifestValue
  |Yes:
   .pushconst
   OGenerateSymbolValue
  |*:
   .pushvar
   OGenerateType
   @GenerateMode
   OGenerateDisp
   OGenerateLevel
   ]
 ]
 .vrange
 ;
TranslateNumericSubtract:

@SkipUnsigned

.sub
OGenerateSubtractType % new semantic
OPopSymbol
;

TranslateMultiple:

[
 |ASetOp: % Deferred, e4906
   @TranslatePowerSetMultiply % p420
 |*:
   @TranslateNumericMultiple % p390
 ];

TranslateNumericMultiple:

.mut
OGenerateMultipleType % New semantic
OPopSymbol
;

SetOpIndication:

[
 |ASetOp:
 |*:
 ];
TranslateDivid:

.div
OGenerateDividType % New semantic
OPopSymbol
;

TranslateMod:

.mod
OGenerateModType % New semantic
OPopSymbol
;

TranslateEqual:

[ OChooseSymbolExtensionKind
 | SxScalar, SxRange, SxSet: % Set not implement
   .compare
   .eql
   OGenerateEqualType % New semantic
   OPopSymbol
   @ForceSymbolToBeBooleanJump % p405
   [*:
     @TranslateLongEqual % e4729, EDeferredLongCompare
    ];
TranslateGreaterEqual: % operands is scala. e4754

@SkipUnsigned
 .compare
 .geq
 OGenerateGreaterThanEqualType % New semantic
 OPopSymbol
 @ForceSymbolToBeBooleanJump % p405
 ;

TranslateGreater: % Scalar operands e4735

@SkipUnsigned % p380
 .ccmpare
 .gt
 OGenerateCompareType % New semantic
 OPopSymbol
 @ForceSymbolToBeBooleanJump % p405
 ;

TranslateInfixOr: % e4586

@ForceSymbolToBeBooleanJump % p405
 @ResolveSymbolBooleanFalse % p405
 ;

TranslateOr: % e4586

@ForceSymbolToBeBooleanJump % p405
 OMergeSymbolBooleanLeftShuntIntoRightTrues % Merge left false
 OSwapSymbol % labels to top true
 OPopSymbol % labels
 ;
ResolveBooleanJumpInSymbol:  % e4628

[ OChooseSymbolExtensionKind
 | SxBooleanJump:
   @ResolveSymbolBooleanTrues  % p405
   OGenerateSymbolBooleanShunts
   |*:  % Shunt still on top symbol
];

ResolveSymbolBooleanTrues:

@TranslateNot          % p405
@TranslateSymbolBooleanFalse % p405

TranslateNot:

@ForceSymbolToBeBooleanJump  % p405
OExchangeTrueLabelAndFalseLabel
;

ResolveSymbolBooleanFalse:

@ForceSymbolToBeBooleanJump
OPushNewLabel
    .false jump
OGenerateLabel
OGenerateSymbolBooleanFalse % 5. Nulled true label shunt
OPopLabel
;

ForceSymbolToBeBooleanJump:  % Replace original symbol kind to bo
% boolean symbol kind. Enter shunt.
% Initial true and false shunt.

[ OChooseSymbolExtensionKind
 | SxBooleanJump:
 |*:  OEnterExtensionKind[ SxBooleanJump]
];
AssertionExpression:

[ 0IsChecked
    |Yes:
        @BooleanControlExpression % p410, e4654
        .not
        @ResolveSymbolBooleanFalse % p405, e4579
        .errassert
        .newline
        OGenerateLineNumber
        OGenerateSymbolBooleanShunts
        OPopSymbol % Boolean result symbol
    |*:
        @SkipExpression % p150
    ];

BooleanControlExpression:

@Expression % p360
@ForceSymbolToBeBooleanJump % p405
;
GenerateMode: % Top symbol is the one need

[OCChooseSymbolKind
 |SyFunction:
   .func_mode
 |SyProcedure:
   .proc_mode % Really should be emitted inline
 |*:
   [OCChooseTypeKind
    |TyEnumerated, TySubrange, TyChar, TyBoolean, TyPointer, TyInteger:
     .sconst_mode
    |*:
     .lconst_mode
   ]
];

GenerateContext: % Top symbol is a formal parameter
 % No other place call this SSL, except from routine
 % call and routine enter
OSelectSymbolAttribute[SyPassByValue]
[OTestSymbolAttribute
 |On:
   .const_parm
 |*:
   .var_parm
];
EmitAddrAndValueForStructureConstant:
    #ETemporaryDeferredStructureConstant;

EmitStructureConstantAddrReference:
    #ETemporaryDeferredStructureConstant;

InnerStructureConstantValue:
    #EDeferredInnerStructureConstantValue;

SolvingStringLengthConstant:
    #EDeferredStringLengthLiteralConstant;

TranslateLongEqual:
    #EDeferredLongEqual;

TranslatePowerSetMultiply:
    #EDeferredPowerSets;

TranslatePowerSetSubtract:
    #EDeferredPowerSets;

TranslatePowerSetAdd:
    #EDeferredPowerSets;

TranslateIn:
    #EDeferredInOperator;

TranslatePowerSetGreaterEqual:
    #EDeferredPowerSets;

TranslatePowerSetLessEqual:
    #EDeferredPowerSets;

TranslateXor:
    #EDeferredPowerSets;
New and Modified Euclid Coder Semantic Routines
1. Add a new semantic mechanism, Loop Label Stack, for Loop statement and Exit statement.

2. New semantic routine, "OPushNewLoopLabel". Generate a new label and push it into top Loop Label Stack.

3. New semantic routine, "OSwapLoopLabel". Swap the top two labels of the Loop Label Stack.

4. New semantic routine, "OGenerateLoopLabel". Emit the top label of the Loop Label Stack.

5. New semantic routine, "OPopLoopLabel". Pop the top label of the Loop Label Stack.

6. New semantic routine, "ORearrangeCaseTable". Which sorts the top Case Table, enters the else label or error label in missing slots (else label or error label at the time should be on top of the Label Stack). After sorting, insert the else label or error label as the first label in Case Table. Also make available the "min" and "max" value of Case Table labels.

7. New semantic routine, "OGenerateAndPopCaseTable". Emit "min", "max" value and all the labels from top Case Table Stack in the sorted sequence.

8. New semantic routine, "OPushErrLabel". Push an error label onto the Label Stack. The error label should be able to bring a run time error message.

9. New semantic routine, "OEnterRangeCaseTable". Enter into the
top Case Table, the range of top two Symbol Stack entries, Also enter a new label taken from the Label Generator.

10. New semantic routine, "OEnterValueCaseTable". Enter into the top Case Table, the value of the top Symbol Stack entry. Also enter a new label.

11. New semantic routine, "OGenerateParamsLength". Emit the Parameters' length which is in the top Type Stack entry. The top Type Stack entry must be a routine. The parameters' length field must be added to routine's Type Table entry. Its value must be entered by Allocator pass of April Euclid (see Appendix C, No. 1).

12. New semantic routine, "OGenerateModType". Emit Pas32's mod operator type. The top two symbols are for the operands.

13. New semantic routine, "OGenerateLevel". Emit Pas32's level information from the top Symbol Stack entry. It may need to take into consideration on how Pas32's levels differ from April Euclid'.

14. New semantic routine, "OGenerateType". Emit Pas32's type information for the symbol from the top Type Stack entry.

15. New semantic routine, "OGenerateSize". Emit Pas32's size information for the symbol from the top Type Stack entry.

16. New semantic routine, "OGenerateAlign". Emit Pas32's alignment information of a function result symbol from the top Type Stack entry. Assume April Euclid already added this new field to the Type Table of the result symbol and filled in the value in the
Allocator pass. See Appendix C No. 2.

17. New semantic routine, "OGenerateCompareType". Emit Pas32's compare operator type. The top two symbols are for the operands.

18. New semantic routine, "OGenerateDisp". Emit Pas32's displacement information for a symbol from the top Symbol Stack entry.

19. Modified semantic routine. "OEnterSymbolExtensionKind". SymbolExtensionKind is a field of the Extended Symbol Stack. The type of this field is Scalar .. Dummy. In April Euclid, every scalar type symbol will have the SymbolExtensionKind equal to 'scalar' in the Symbol Stack entry, after translating the symbol's input tokens. When translation reaches the assignment operator, this information about the operand symbol won't tell whether a range check needs to be emitted, since at that time, the types for the symbols are already popped. We add a new value 'syrange' to the domain of SymbolExtensionKind. A subrange type symbol and an enumeration type symbol will have its SymbolExtensionKind equal to 'syrange' when Translation reaches the assignment operator, so that a proper range check for Pas32 assignment can be emitted. This is done by the semantic routine "OEnterSymbolExtensionKind", which is called within the design SSL procedure "EnterTypeInfoIntoSymbol".

20. New semantic routine, "OGenerateSymbolValue". Emit the value of the symbol which is on top of the Symbol Stack. The value should be a manifest value.
21. **New semantic routine, "OGenerateLiteralNumberSymbol".** Emit the numeric literal to Pas32 directly from the Coder's input stream.

22. **New semantic routine, "OGenerateLiteralCharSymbol".** Emit the single character literal to Pas32 from the Coder's input stream.

23. **New semantic routine, "OGenerateSubtractType".** Emit the Pas32 subtract operator type from the top Symbol Stack entry.

24. **New semantic routine, "OPushExtensionTypeFirstSymbol".** Push type first symbol from top symbol stack's extensionkind to top of the Symbol Stack. Also see 26.

25. **New semantic routine, "OPushExtensionTypeLastSymbol".** Push type last symbol from top symbol stack's extensionkind to top of the Symbol Stack. Also see 27.

26. **New semantic routine, "OEnterSymbolExtensionTypeFirstSymbol".** A new field should be added to the Symbol Extension Stack entry for the enumeration type and the subrange type. This new field will hold type first symbol of the symbol with enumeration or subrange type. This routine enters the type first symbol into the top of Symbol Extension Stack entry. It will be used by no. 24 when translating the assignment and emitting Pas32's runtime range check.

27. **New semantic routine, "OEnterSymbolExtensionTypeLastSymbol".** A new field for the type last symbol should be added to the Symbol Extension Stack entry for the enumeration type and the
 subrange type. This routine enters the type last symbol into the top of the Symbol Extension Stack entry. It will be used by no. 25 when translating assignment and emitting Pas32's run time range check.

28. Modified semantic routine, "0GenerateSymbolBooleanFalse". The operations are: 1. Enter a new label from the top Label Stack entry into the true label shunts of top Symbol Stack entry. 2. Emit the false label shunts from the top Symbol Stack entry. 3. Empty the false label shunts of top Symbol Stack entry. 4. False label shunts <-- True label shunts. 5. Empty the true label shunts of top Symbol Stack entry.

29. New semantic routine, "0GenerateNewLine". Get the line number from defined line number generator and emit it to output. This routine is used after the new operator "errassert" is emitted. Also see Appendix D, number 5.

30. Modified semantic routine, "0GenerateSymbolBooleanShunt". Just emit the false label shunts from top Symbol Stack entry.

31. New semantic routine, "0InitialStack". Initialize all the stacks which are used by this translation. They are Symbol Stack, Type stack, Construct Stack, Count Stack, Label stack, Check Stack and Loop Label Stack.

32. New semantic routine, "0GenerateLabel". Emit the label to output stream from the top of the Label Stack.

33. New semantic routine, "0GenerateConst(zero)". Emit the the constant value zero to the output stream.
34. New semantic routine, "OGenerateMainVarLength". The top symbol stack and top type stack should be main module. This routine takes the length of declared variables which are inside the main module and outputs it. Also see Appendix C, No. 3.

35. New semantic routine, "OEnterProgramLevel". Enter Pas32's program level into the top symbol stack entry.

36. New semantic routine, "OGenerateEqualType". Emit Pas32's equal operator type. The top two symbols are for the operands.

37. New semantic routine, "OGenerateGreaterEqualType". Emit Pas32's greater-equal operator type. The top two symbols are for the operands.

38. New semantic routine, "OGenerateInvariantLabel". Emit the label which represents the INVARIANT routine entry. It is in the top Type Stack entry. The top Symbol Stack entry should be a module.

39. New semantic routine, "OGenerateOneUpperLevel". Emit One higher than Pas32's level from the top Symbol Stack entry. The top entry should be a routine.

40. New semantic routine, "OGenerateRoutineLabel". Emit the label which represents the entry address of a routine. The label is in the top Type Stack entry.

41. New semantic routine, "OGeneratePostLabelAddress". Emit the post label which is on the top Type Stack entry and represents
the POST routine address.

42. New semantic routine, "OPushCopyCount". Push a copy of the current top Count Stack entry to the Count Stack.

43. New semantic routine, "OGenerateCountValue". Emit a counter value to the output stream. The value is on top of the count stack.

44. New semantic routine, "OGenerateParamsLength". Emit the parameters length field to the output stream from the top of the Type Stack, which is a routine. Also see Appendix C, No. 4.

45. New semantic routine, "OGenerateVarsLength". Emit the length of local variables of a routine from the top of the Type Stack. Also see Appendix C, No. 5.

46. New semantic routine, "OGenerateForwardBodyAddressLabel". Emit the FORWARD label from the top Type Stack entry to output stream.

47. New semantic routine, "OGenerateTypeLength". Emit the type length information from the top Type Stack entry. The top Type Stack entry must be a routine formal Parameter. Also see Appendix C, No. 6.

48. New semantic routine, "OGenerateConstructRoutineLabel". Emit labels of the top construct stack. The entry must be a routine, otherwise a search needs to be done to locate the routine first encountered.
49. New semantic routine, "0SearchAndGenerateReturnMode". Search the Symbol stack from the second top for a procedure symbol or function symbol. Emit 'func_mode' if found a function; emit 'proc_mode' if found a procedure.

50. New semantic routine, "0GenerateCopyLength". Emit the length of the long variable from the top Symbol Stack entry. This is emitted following a copy operator. The length must be a manifest value.

51. New semantic routine, "0GenerateMultipleType". Emit Pas32's multiply operator type. The top two symbols are the operands.

52. New semantic routine, "0GenerateDividType". Emit Pas32's divide operator type. The top two symbols are for the operands.
APPENDIX C

Modifications to April Euclid Allocator
1. Add a new field, parameters' length, to routine's Type table entry. Also, the Allocator pass of April Euclid must enter the value. This new field is required for translating a routine call and a routine entry into Pas32.

2. Add a new field, Pas32's "Align", to the Type Table entry of April Euclid's function result variable. The Allocator pass must enter the needed value also.

3. Add a new field which represents the length of the variables declared in a main module to the main module's type table entry. Also fix Allocator pass to enter the proper value. This is for the calling statement of the main program.

4. Add a new field to a routine's Type Table entry and Type Stack Entry which represents the parameters length of the routine. The information should be entered by Allocator pass.

5. Add a new field to a routine's Type Table entry and Type Stack Entry which represents the length of local declared variables of the routine. The information also should be entered by the Allocator pass of Euclid.

6. Add a new field to a routine formal's Type table entry which represents the type length of the formal. This is either a manifest or nonmanifest value. The Allocator must enter the manifest value which represents the passed length.
APPENDIX D

Modifications to Pas32 Code Generation
1. Accept new operator "popexpr". After reading the new operator, pass 6 of Pas32 must delete the most recently built tree of an expression.

2. Accept new operator "pop". Pass 6 of Pas32 must delete the most recently built tree of a variable reference.

3. Accept new operator "varrange". Pas32 will generate a run time check for the assignment of nonmanifest variable which is the subrange type or enumeration type.

4. Accept new operator "addrindx". Pas32 will generate a run time check and array subscript computation for a reference to a nonmanifest array element.

5. Accept New operator "errassert". Pas32 will generate a run time error message when encountering this operator. The operator will appear when an assertion test fail.

6. Accept new operator "initialassign". Pas32 takes the top symbol of Symbol Stack as an expression's result and assigns it to next several symbols of the symbol stack. The counter value following the "initialassign" operator will determine how many symbols are to be assigned the value. This is for the scalar type variables initial value.

7. Accept new operator "initialCopy". The operations performed by Pas32 are the same as above, except this is a structured value to be assigned.

8. Accept new operator "beginsplit". This operator is emitted when encounter a FORWARD routine declaration. Pas32 should
find the body of the routine and make it compilable. also see next.

9. Accept new operator "endbodysplit". This operator is emitted when encounter the body of a FORWARD routine declaration. Together with above new operator, Pas32 must make the FORWARD routine compilable.

10. Accept new operator "skipfornestings" when entering a new routine, and accept new operator "resumeofnestings" after comming out from a routine. Pas32 pass 6 should rearrange the routine block, to make available for compiling one routine at a time.
REFERENCES

1. Wortman, D. B.,
"Structure of the Scanner and Screener",
Euclid working paper 3,
Computer Science Research Group of the University of Toronto,
Toronto, Canada, 1977

2. Holt, R. C.,
"Notes on the Structure of the Euclid Translator",
Euclid working paper 27.
Computer Science Research Group of the University of Toronto,
Toronto, Canada, 1977

"Introduction to S/SL: Syntax/Semantic Language",
Computer Science Research,
Group of the University of Toronto,
Toronto, Canada, 1980

4. Hartmann, Alfred C.,
"A Concurrent Pascal Compiler for Minicomputers",
PhD Thesis,
California Institute of Technology,
Pasadena, California, 1976

5. Young, Robert,
"Internal documentation for the Pascal/32 compiler",
Computer Science, Kansas State University
Manhattan, Kansas, 1979

Mitchell, J. G. and Popek, G. J.
REFERENCES

"Report on the Programming Language Euclid",
SIGPLAN Notices, vol. 12, no. 2 (Feb. 1977)

7. Roeseven, D.
"Euclid to Pascal Translator",
Masters Report, Computer Science, Kansas State University,
Manhattan, Kansas, 1980

8. Holt, R.C.
"April Euclid Coder S/SL",
Computer Science Research Group of the University of Toronto,
Toronto, Canada, June, 1979

9. Holt, R.C.,
"Compiling Parameterized Types",
Euclid Working Paper 34,
Computer Science Research Group of the University of Toronto,
Toronto, Canada, June, 1978

10. Holt, R.C.,
"Semantic Mechanisms and Operations for the Coder",
Euclid Working Paper 76,
Computer Science Research Group of the University of Toronto,
Toronto, Canada, 1978

11. Holt, R.C.
"Addressing and Routine Calling in Euclid/11",
Euclid working paper 45,
Computer Science Research Group of the University of Toronto,
Toronto, Canada, 1978
REFERENCES

12. Cordy, J.R.,
"Revised Contents of the Symbol Table",
Euclid Working Paper 57,
Computer Science Research Group of the University of Toronto,
Toronto, Canada, 1978

13. Cordy, J.R.,
"Revised Contents of the Type Table",
Euclid Working Paper 63,
Computer Science Research Group of the University of Toronto,
Toronto, Canada, 1978

14. Hclt, R.C.,
"Data Descriptors (revised) for use by the Emitter",
Euclid Working Paper 74,
Computer Science Research Group of the University of Toronto,
Toronto, Canada, 1978

15. Hclt, R.C.,
"April Euclid Coder Semantic Skeleton",
Computer Science Research Group of the University of Toronto,
Toronto, Canada, June, 1979
Interfacing of a Euclid Compiler to a Pascal Code Generator
by
Pohwa Hsu

E.A., Fu-Jen Catholic University, 1976

-------------------

AN ABSTRACT OF A MASTER'S REPORT

submitted in partial fulfillment of the

requirements for the degree

MASTER OF SCIENCE

Department of Computer Science

KANSAS STATE UNIVERSITY
Manhattan, Kansas

1982
A project is being conducted in the department to make Euclid available for compilation and execution on the 8/32 machine. This report is part of the design for this compiler modification.

The Euclid compiler now available is written in Euclid. A Pascal compiler, written in Pascal, also exists, which generates 8/32 machine code. Both existing compilers are multi-pass compilers. The intended solution is to build one intermediate pass which serves as a translation from one of the Euclid interpass languages to one of the PASCAL interpass languages. A Euclid program will first be translated by the Syntactic and Semantic Passes of the existing Euclid compiler. It will then be translated by the new pass to an appropriate form. This will finally be translated by the optimization and code generator passes of the Pascal compiler, producing 8/32 machine code.