

# An Object-Oriented Compiler Construction Toolkit

Timothy P. Justice  
Department of Computer Science  
Oregon State University  
Corvallis, Oregon  
97331-3202  
justict@storm.cs.orst.edu

March 12, 1993

## Abstract

Although standard tools have been used for lexical and syntactic analysis since the late 1970's, no standard tools exist for the remaining parts of a compiler. Part of the reason for this deficiency is due to the difficulty of producing elegant tools capable of handling the large amount of variation involved in the compiling process. The **Object-oriented Compiler Support** toolkit is a suite of reusable software components designed to assist the compiler writer with symbol management, type checking, intermediate representation construction, optimization, and code generation. A collection of C++ classes defines a common interface to these tools. Variations in implementation are encapsulated in separately compiled modules that are selected and linked into the resulting compiler.

## 1 Introduction

A compiler is a program that translates a source language into an equivalent target language. The process of compilation is well studied [ASU86, FL91, Hol90, Pys88]. Standard tools have been in use since the late 1970's for lexical analysis and syntactic analysis, however no such standard tools exist for the remaining parts of a compiler. Perhaps the reason for this has less to do with our understanding of the problem or even the solution than the difficulty of expressing an elegant solution that encompasses the large amount of variation existing within the problem. Variations in compiler implementation arise from several factors, including:

- Different source languages, such as C or Pascal.
- Different target architectures, such as the Intel 80386 [CG87] or the Motorola 68030 [Wak89].
- Different optimization techniques.
- Different code generation strategies.

The goal of the **Object-oriented Compiler Support** toolkit, OCS (pronounced ox), is the development of a collection of tools that assist the compiler writer with symbol management, type checking, intermediate code construction, optimization, and code generation, using object-oriented techniques. The project focuses on producing a fixed, high-level interface to the tools that is capable of accommodating a variety of source languages. The behavior of the tools is specialized by providing multiple implementations of the functionality described by the interface. The tools are flexible and work well with existing tools such as Lex [LS75] and Yacc [Joh78].

Much work has been done in the area of compiler tools. However, most solutions have used functional decomposition to model the process of compilation. Meyer uses the example of translating a C program to Motorola 68030 code to illustrate functional decomposition and support his assessment that top-down functional design is poorly adapted to the development of significant software systems [Mey88, page 43–49].

The approach taken in the design of OCS is the identification of the objects of compilation, such as symbols, types, expressions, and statements. These objects are assigned responsibilities. For example, a statement is responsible for generating code for itself. The tools take the form of a collection of C++ classes. Figure 1 provides an overview of the tools. The class definitions provide a common interface to the compiler writer. The method implementations provide the variations in behavior, such as whether to generate code for a Motorola 68030 or a National Semiconductor 32032, or what level of intermediate code optimization to use. The different implementations are encapsulated in separately compiled modules that can be selected and linked into the target compiler. In order to keep the size of the research project manageable, its scope is limited to the translation of imperative languages such as Pascal and C into conventional CISC target architectures such as the Motorola 68030 and National Semiconductor 32032.

The remainder of the paper is organized as follows: Section 2 discusses previous work in the area of compiler tools. In Section 3, overall compiler design is investigated. OCS tool usage is demonstrated in Section 4 through an extended example showing the development of a compiler for a simple programming language. The implementation of the tools is the topic of Section 5. Finally, Section 6 provides a conclusion and lists future work.

## 2 Related Work

Much work has been done in the area of compiler tools. However, with the exception of lexical analyzer generators, such as Lex [LS75] and Flex [Pax90], and parser generators, such as Yacc [Joh78] and Bison [DS91], no tool has gained widespread use. Perhaps part of the reason for the widespread use of Lex and Yacc is their distribution as part of Unix [KP84]. The following sections identify a few notable efforts in this area.

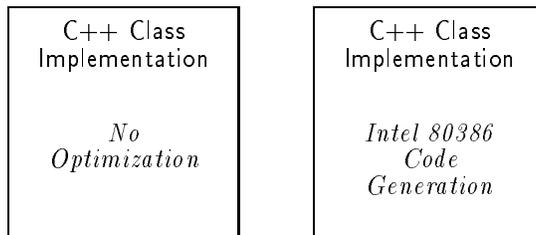
### 2.1 Lex and Yacc

The standard tool for lexical analysis is Lex [LS75]. The input to Lex is a table of regular expressions and corresponding code fragments. Lex generates a subprogram that reads an input stream, breaking it up into tokens as described by the regular expressions. When a particular token is recognized, the corresponding code fragment is executed.

Yacc [Joh78] is the standard tool for syntax analysis or parsing. The input to Yacc is a set of productions and corresponding code fragments. The subprogram generated by Yacc reads tokens from the lexical analyzer

*Compilers Are Built On A Common Interface*

<i>C Compiler</i>		<i>Pascal Compiler</i>		<i>Other language compilers</i>	
<b>C++ Class Interface</b>					
<i>Symbol Management</i>		<i>Intermediate Representation Construction</i>		<i>Type Checking</i>	
<i>Optimization</i>		<i>Code Generation</i>			
C++ Class Implementation	C++ Class Implementation	C++ Class Implementation	C++ Class Implementation	C++ Class Implementation	C++ Class Implementation
<i>Symbol Management</i>	<i>Intermediate Representation Construction</i>	<i>Type Checking</i>	<i>Peephole Optimization</i>	<i>MC68030 Code Generation</i>	



*Multiple Implementations Provide For Variation*

Figure 1: Overview of OCS.

and attempts to match them to the user provided productions. When the right-hand side of a production is recognized, the corresponding code fragment is executed.

Lex and Yacc were designed to work together, although each can be used separately. OCS does not provide tools for lexical or syntactic analysis, so there is no overlap of features. OCS can be used in conjunction with Lex and Yacc to produce complete compilers.

## 2.2 PQCC

The Production-Quality Compiler-Compiler project [LCH<sup>+</sup>80] sought to address the problem of providing a truly automatic compiler-writing system that included not only lexical analysis and syntactic analysis, but also optimization and code generation. The basic design goal was the construction of compilers that would generate highly optimized code from user supplied descriptions of the source language and target computer. These descriptions are processed by the PQCC table generator to produce tables. The resultant tables along with the source program provide the input to the PQC skeleton compiler which produces the object program.

In order to manage the complexity of compilation, the PQCC system decomposes the compilation process into a sequence of phases, each representing an “intellectually manageable subtask.” The intermediate representation was an abstract syntax tree, referred to as TCOL (Tree Computer Oriented Language).

The PQCC project focuses mainly on the implementation of optimization and code generation, demonstrating that retargetability and a high degree of optimization can be successfully achieved. However, the compiler writer interface to the tools still requires precise definition [LCH<sup>+</sup>80]. OCS directly addresses the compiler writer interface issue by providing a fixed, well-defined interface. This interface isolates tool usage from the underlying implementation, permitting aspects of the implementation to be changed without affecting the manner in which the tools are used.

## 2.3 ACK

The Amsterdam Compiler Kit [TSKS83] was based on the idea of UNCOL (UNiversal Computer Oriented Language), a common intermediate language that would be produced as the output of the front end of the compiler and serve as the input to the back end of the compiler. Using this approach, if compilers were needed for  $N$  source languages and  $M$  target machines, only  $N + M$  programs would be required instead on  $N \times M$ . The focus was on the production of a family of compilers, all compatible with one another so that programs written in one source language could call procedures written in another source language.

The original implementation was divided into six passes: front end, peephole optimizer, global optimizer, back end, target optimizer, and assembler. There was also a preprocessor for source languages that allow macro expansion, file inclusion, and conditional compilation. Each pass was a separate program that read the output of the previous pass, performed some further processing, and produced output for the next pass. The intermediate representation used was called EM (Encoding Machine) code, the machine language for a simple stack machine. The first three passes produced EM code as output, the fourth and fifth produced assembly language, and the last pass produced the binary executable program. The compiler writer implementing a new source language is required to write a front end program which reads the source language and produces EM code as output. To produce a compiler for a new target machine, the compiler writer provides machine-dependent driving tables for the back end, target optimizer, and assembler. A later implementation of the toolkit combined the passes into a single program to improve compiler performance [TKLJ89].

While the Amsterdam Compiler Kit provides extensive facilities for optimization and code generation, it does little to assist the compiler writer in producing a front end, beyond defining an intermediate representation. The compiler writer must provide the code for the lexical, syntactic, and semantic analysis phases, construct the intermediate representation in EM code, and pass it to the peephole optimizer phase. In OCS, the compiler writer instantiates objects for the various language constructs. These objects form the intermediate representation. Much of the semantic analysis is performed automatically as each object is created. A binary expression, for example, performs type checking and implicit conversions when it is created. The OCS approach reduces programming effort and enhances robustness by assuring that type checking is performed uniformly on all expression instances.

## 2.4 Eli

Eli [GHL<sup>+</sup>92] was designed to overcome three major deficiencies in current compiler construction tools: the steep learning curve associated with tool usage, the lack of smooth integration of independently developed tools, and the inferior performance of the compilers created by tools compared to hand-coded compilers. Compilation is decomposed into fourteen subproblems, generally grouped into *structuring*, *translation*, and *encoding*. The compiler writer supplies specifications for the source program text, source program tree, target program tree, and machine instruction set as well as the relationships among them. Eli constructs the compiler from these specifications and various library routines by using an expert system. The intent of this approach is to isolate the details of tool usage from the compiler writer and smooth tool interaction.

In order to reduce the learning curve associated with tool usage, Eli provides a number of special-purpose languages for expressing solutions to specific subproblems. These languages are intended to match the compiler writer's understanding of the subproblem. Rather than introduce new notation, OCS provides a collection of C++ classes to represent source language features. These classes provide a natural mapping between the tools and the compiler writer's understanding of the compilation process, without the overhead of the compiler writer learning special languages.

## 2.5 PCCTS

The Purdue Compiler-Construction Tool Set [PDC92] integrates lexical analysis and syntactic analysis more tightly than Lex and Yacc. A PCCTS grammar contains both the lexical and syntactic specifications as well as intermediate-form construction, and error reporting. In order to facilitate its use, PCCTS notation borrows from previous tools such as Yacc. However, there are two key differences between PCCTS and Yacc. First, PCCTS produces an LL(k) parser, whereas Yacc produces an LALR(1) parser. PCCTS's choice of a top-down parsing algorithm allows attributes to be inherited, that is, a production rule can obtain attribute information from the context in which it was invoked. Second, while Yacc grammars are specified using BNF, Backus-Naur Form, PCCTS allows the use of EBNF, Extended Backus-Naur Form. EBNF extends BNF by permitting repetition and alternation, resulting in more concisely expressed grammars.

The intermediate representation produced is an abstract syntax tree, AST, in a LISP-like notation.

```
( root, child-1, child-2, ..., child-N )
```

AST's can be created automatically by PCCTS or explicitly by the compiler writer. A library of routines is provided for manipulating AST's. For example, there are routines for adding a leaf to an AST, duplicating an AST, and performing a preorder traversal of an AST.

The output is a parser program in C source code that recognizes the language described by the input grammar. The C code is designed to be human readable to allow the use of standard debugging tools.

PCCTS Version 1.00 focuses primarily on lexical and syntactic analysis, which is not part of OCS. PCCTS's intermediate representation is more primitive than that of OCS, requiring the compiler writer to perform direct manipulations, such as adding nodes and performing traversals. OCS presents the compiler writer with a high-level intermediate representation. Instead of creating nodes in a tree, the compiler writer creates expressions and statements, which correspond more closely to the actual source language constructs being translated. PCCTS does not currently provide optimization or code generation, while OCS does. Future versions of PCCTS will reportedly add these features [PDC92].

## 3 Compiler Design

### 3.1 The Classical Approach

Modern compilers are modeled as a sequence of phases, each accepting the source program in one representation, transforming it in some manner, and producing a new representation [ASU86, FL91, Hol90, Pys88]. Figure 2 shows how the phases are connected as well as the input and output of each. This model lends itself to the classical top-down functional design method which is based on stepwise refinement of the abstract functionality of a system. Meyer identifies four major flaws with top-down design [Mey88, page 44]:

- **The evolutionary nature of software systems is not taken into account.** The focus is on external interfaces which are particularly susceptible to change and represent the more superficial aspects of the system. As the abstract functionality is refined into a more detailed control structure, ordering constraints are imposed between elements of the structure. This premature emphasis on temporal relations limits the flexibility of the system.
- **Not all software systems can be described by a single abstract function.** Many software systems are more realistically viewed as a set of services. Formulating a single abstract function for these systems imposes an artificial structure that can yield a poor design.
- **Data structure is neglected.** Typically many functions must interact with the data. Centering the design around the functions can weaken the data structure by distributing its description among the functions.
- **Reusability is not promoted.** Each refinement is designed to satisfy a specific functional requirement. As a result software elements tend to be narrowly defined, inhibiting their reuse.

### 3.2 An Object-Oriented View

Object-oriented programming is a new programming paradigm, a new way of viewing computation. Various authors have provided definitions of object-oriented programming, each with his or her own emphasis [CN91, KL89, Mic88, Mey88, PW88, Weg86]. The key aspects of object-oriented programming integral to this project are:

- A design methodology that views software in terms of a group of interacting agents responsible for providing specific services to one another.

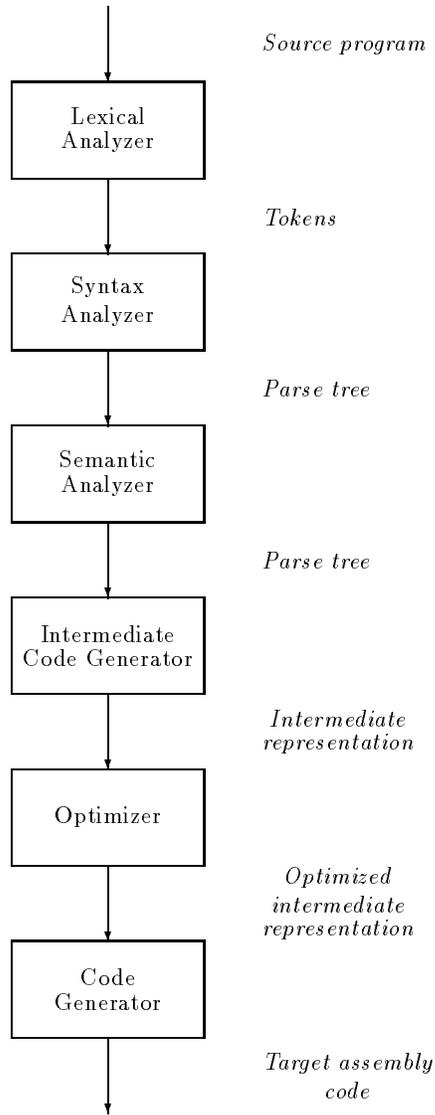


Figure 2: Phases of a compiler.

- An implementation technique that defines agents as encapsulations of knowledge and behavior supplying both generalization through inheritance and specialization through overriding, and agent interaction via message passing.

An object-oriented view of compiler design focuses on the objects associated with compilation and their corresponding responsibilities rather than particular phases of the compilation process. Figure 3 illustrates an object-oriented compiler model. Arcs with solid lines represent requests made by the compiler writer. Arcs with dotted lines represent requests made by the underlying implementation.

Action is initiated by requesting the syntax analyzer to translate the source code. The syntax analyzer requests tokens from the lexical analyzer, which scans the source code searching for instances of user defined patterns and delivers the corresponding tokens. As the syntax analyzer recognizes various programming language constructs it makes a series of requests specified by the compiler writer. There is no semantic analyzer, intermediate code generator, optimizer, or code generator. Instead, the compiler writer requests the creation of symbol, type, expression, and statement objects that represent the language constructs recognized, and then asks these objects to generate target code for themselves. The complexity of semantic analysis and code synthesis is therefore managed by delegating the responsibility to the objects themselves. For example, an expression representing the addition of two integer values knows how to generate code for itself, but it does not need to know how to generate code for a function call.

## 4 Using The Tools

OCS comprises several class hierarchies representing various programming language features. Figure 4 illustrates a portion these hierarchies. The compiler writer instantiates objects for declarations, expressions, statements, and so on, as each construct is recognized. Collectively these objects are the intermediate form of the source language. Much of the semantic analysis is performed by the constructors for the various classes. When a `FunctionCall` is created, for example, the constructor sends a message to a `Scope` object to look up the symbol table entry for the function and perform type checking on the actual parameters. Target code is generated when the compiler writer sends the `genCode` message to the statements.

The grammar shown in Figure 5 is provide to illustrate tool usage. The language described by the grammar, which we call *MINPAS*, is essentially a simplified Pascal [JW85]. Comments are enclosed within braces, `{ }`, and cannot be nested. Identifiers must begin with a letter and may be followed by any combination of letters and digits. Upper and lower case letters are considered different.

A compiler has been implemented for this language using OCS in conjunction with Lex and Yacc. The complete Lex and Yacc specifications are shown in Appendix B and Appendix C, respectively. Excerpts from these specification are highlighted in the following sections.

### 4.1 Declarations

In the Lex specification, a `Symbol` is created when an identifier is recognized. When `integer` or `real` is recognized an `IntegerType` or `FloatType` is created.

```
id          [A-Za-z][0-9A-Za-z]*
```

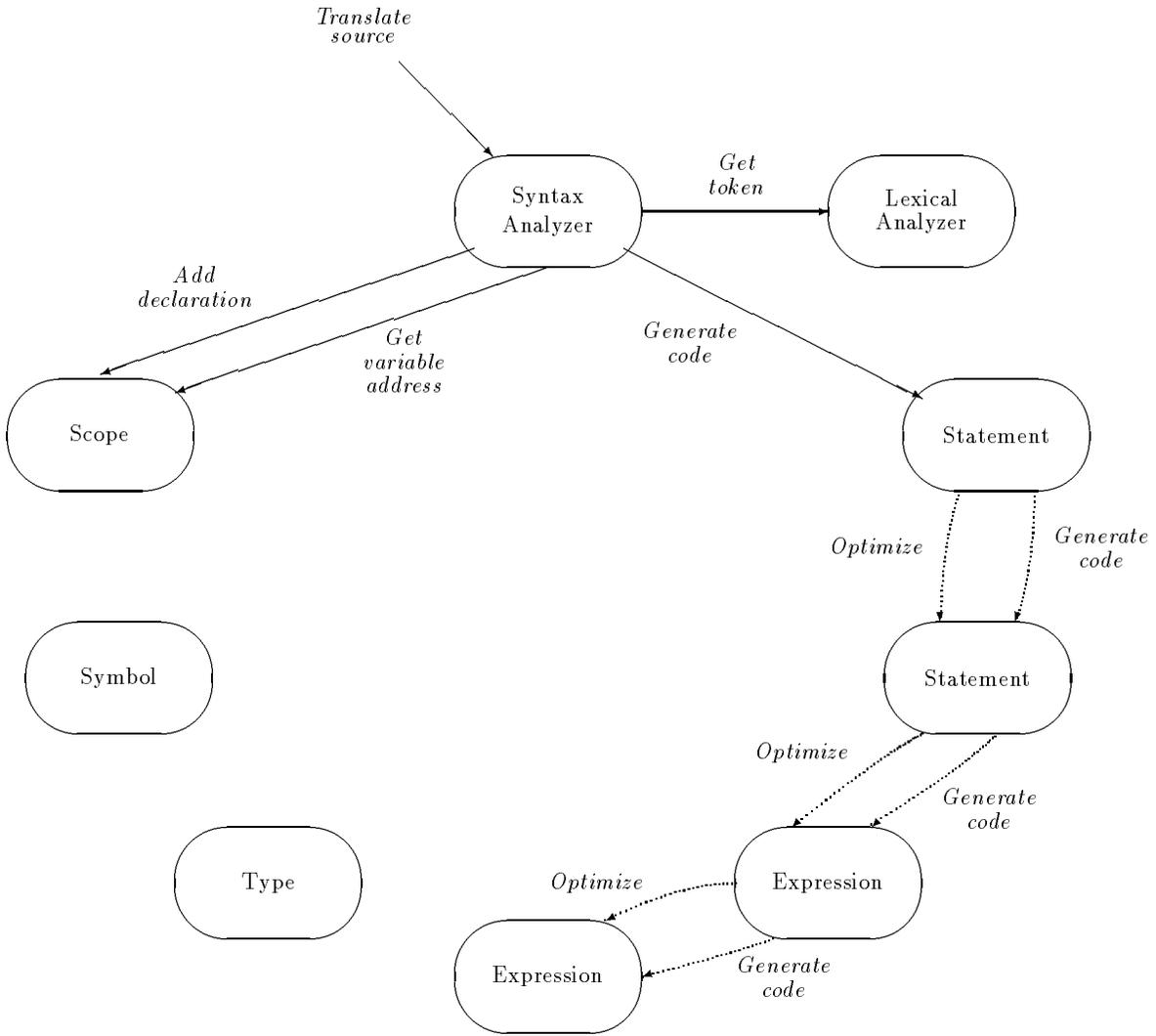


Figure 3: An object-oriented compiler model.

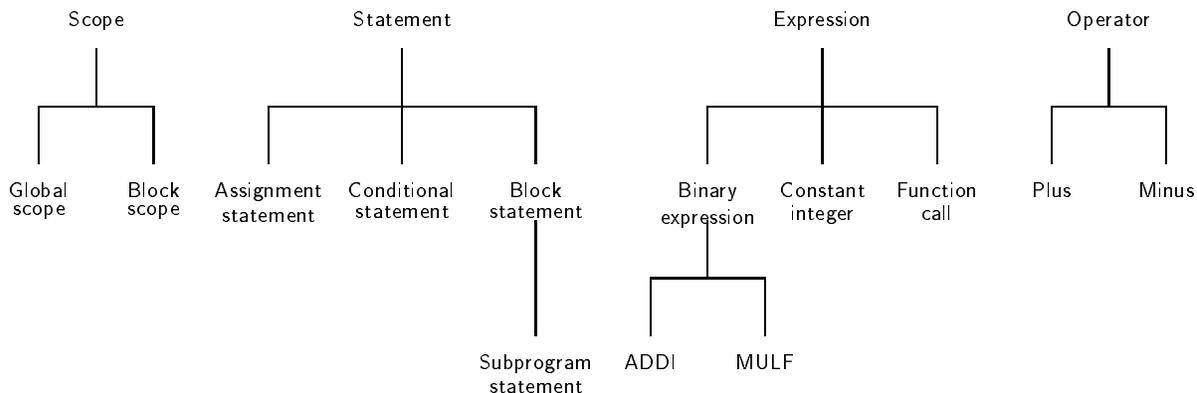


Figure 4: Partial OCS Class Hierarchies.

```

...
{id}          { yylval.sym = new Symbol(yytext);
               return ID; }
"integer"    { yylval.type = new IntegerType();
               return TYPE; }
"real"       { yylval.type = new FloatType();
               return TYPE; }
  
```

A **Declaration** is a combination of a **Symbol** and an **Attribute**. A **Symbol** represents an identifier in the source program while an **Attribute** characterizes identifier usage. In general, identifiers can be constants, variables, subprograms, types, or labels. *MINPAS* limits identifier usage to variable names and function names. A **Type** object returned by Lex is used in the Yacc specification to make an instance of **VariableAttribute** which, combined with a **Symbol**, produces a **Declaration** for a variable. Multiple variable declarations are built into a **DeclarationList**.

```

variable_decl : VAR variable_list SEMICOLON
               { currentScope->insert($2); }
               ...
               ;

variable_list : decl
               { $$ = new DeclarationList($1); }
               | variable_list SEMICOLON decl
               { $$ = $1->addLast($3); }
               ;

decl          : ID COLON TYPE
  
```

$\langle \text{program} \rangle$	$\rightarrow$	<b>program</b> $\langle \text{block} \rangle$ .
$\langle \text{block} \rangle$	$\rightarrow$	$\langle \text{var\_decl} \rangle$ $\langle \text{fun\_decl} \rangle$ $\langle \text{compound\_stmt} \rangle$
$\langle \text{var\_decl} \rangle$	$\rightarrow$	<b>var</b> $\langle \text{var\_list} \rangle$ ;   $\epsilon$
$\langle \text{var\_list} \rangle$	$\rightarrow$	$\langle \text{decl} \rangle$   $\langle \text{var\_list} \rangle$ ; $\langle \text{decl} \rangle$
$\langle \text{decl} \rangle$	$\rightarrow$	$\langle \text{id} \rangle$ : $\langle \text{type} \rangle$
$\langle \text{fun\_decl} \rangle$	$\rightarrow$	$\langle \text{fun\_decl} \rangle$ $\langle \text{function} \rangle$ ;   $\epsilon$
$\langle \text{function} \rangle$	$\rightarrow$	<b>function</b> $\langle \text{id} \rangle$ ( $\langle \text{var\_list} \rangle$ ) : $\langle \text{type} \rangle$ ; $\langle \text{block} \rangle$
$\langle \text{stmt\_list} \rangle$	$\rightarrow$	$\langle \text{stmt} \rangle$   $\langle \text{stmt\_list} \rangle$ $\langle \text{stmt} \rangle$
$\langle \text{stmt} \rangle$	$\rightarrow$	$\langle \text{compound\_stmt} \rangle$   $\langle \text{assign\_stmt} \rangle$   $\langle \text{if\_stmt} \rangle$   $\langle \text{while\_stmt} \rangle$   $\langle \text{read\_stmt} \rangle$   $\langle \text{write\_stmt} \rangle$   $\langle \text{return\_stmt} \rangle$
$\langle \text{compound\_stmt} \rangle$	$\rightarrow$	<b>begin</b> $\langle \text{stmt\_list} \rangle$ <b>end</b>
$\langle \text{assign\_stmt} \rangle$	$\rightarrow$	$\langle \text{id} \rangle$ := $\langle \text{expr} \rangle$ ;
$\langle \text{if\_stmt} \rangle$	$\rightarrow$	<b>if</b> $\langle \text{expr} \rangle$ <b>then</b> $\langle \text{stmt} \rangle$ <b>else</b> $\langle \text{stmt} \rangle$   <b>if</b> $\langle \text{expr} \rangle$ <b>then</b> $\langle \text{stmt} \rangle$
$\langle \text{while\_stmt} \rangle$	$\rightarrow$	<b>while</b> $\langle \text{expr} \rangle$ <b>do</b> $\langle \text{stmt} \rangle$
$\langle \text{read\_stmt} \rangle$	$\rightarrow$	<b>read</b> $\langle \text{id} \rangle$ ;
$\langle \text{write\_stmt} \rangle$	$\rightarrow$	<b>write</b> $\langle \text{expr} \rangle$ ;
$\langle \text{return\_stmt} \rangle$	$\rightarrow$	<b>return</b> $\langle \text{expr} \rangle$ ;
$\langle \text{expr\_list} \rangle$	$\rightarrow$	$\langle \text{expr} \rangle$   $\langle \text{expr\_list} \rangle$ , $\langle \text{expr} \rangle$
$\langle \text{expr} \rangle$	$\rightarrow$	$\langle \text{simple\_expr} \rangle$   $\langle \text{simple\_expr} \rangle$ $\langle \text{relop} \rangle$ $\langle \text{simple\_expr} \rangle$
$\langle \text{simple\_expr} \rangle$	$\rightarrow$	$\langle \text{term} \rangle$   $\langle \text{simple\_expr} \rangle$ $\langle \text{addop} \rangle$ $\langle \text{term} \rangle$
$\langle \text{term} \rangle$	$\rightarrow$	$\langle \text{factor} \rangle$   $\langle \text{term} \rangle$ $\langle \text{mulop} \rangle$ $\langle \text{factor} \rangle$
$\langle \text{factor} \rangle$	$\rightarrow$	$\langle \text{integer\_const} \rangle$   $\langle \text{real\_const} \rangle$   $\langle \text{id} \rangle$ ( $\langle \text{expr\_list} \rangle$ )   $\langle \text{id} \rangle$   ( $\langle \text{expr} \rangle$ )
$\langle \text{relop} \rangle$	$\rightarrow$	=   <>   <   <=   >   >=
$\langle \text{addop} \rangle$	$\rightarrow$	+   -
$\langle \text{mulop} \rangle$	$\rightarrow$	*   /   mod
$\langle \text{type} \rangle$	$\rightarrow$	<b>integer</b>   <b>real</b>

Figure 5: Grammar for *MINPAS*.

```

        { $$ = new Declaration($1, new VariableAttribute($3)); }
    ;

```

The global variable `currentScope` is used to keep track of the open scopes. At the beginning of a program, a `GlobalScope` is created and assigned to `currentScope`. When a function is entered, a new `SubprogramScope` becomes the `currentScope`. Variable declarations, collected into a `DeclarationList`, are inserted into the `currentScope`. The `Scope` referenced at `currentScope` assigns storage to the variables and adds them to its symbol table. At the end of a function or the entire program, the `Scope` is closed by assigning the parent of `currentScope` to `currentScope`. Scopes can be nested to an arbitrary depth.

```

program_hdg      : PROGRAM
                  { ...
                    currentScope = new GlobalScope(); }
                  ;

block            : variable_decl function_decl BEGIN_STMT stmt_list END_STMT
                  { ...
                    currentScope = currentScope->parent(); }
                  ;

variable_decl    : VAR variable_list SEMICOLON
                  { currentScope->insert($2); }
                  | /* empty */
                  { }
                  ;

function_decl    : /* empty */
                  { $$ = new StatementList(); }
                  | function_decl function SEMICOLON
                  { $$ = $1->addLast($2); }
                  ;

function_hdg     : FUNCTION ID LPAREN variable_list RPAREN COLON TYPE
                  { $$ = new SubprogramType($2, $4, $7);
                    currentScope->insert($2, new SubprogramAttribute($$));
                    currentScope = new SubprogramScope(currentScope, $$); }
                  ;

```

Functions declarations are also inserted into the `currentScope`. A `SubprogramType` includes the function name, the formal parameter declarations, and the return type. A `SubprogramAttribute` is grouped with identifier `Symbol` to produce the declaration.

## 4.2 Expressions and Statements

Integer and real constants are recognized by Lex and returned to Yacc as instances of `ConstantInteger` and `ConstantFloat`.

```

integerNumber      [+\\-]?[0-9]+
realNumber         [+\\-]?[0-9]+(\\. [0-9]+)?(E[+\\-]?[0-9]+)?
...
{integerNumber}   { yylval.expr = new ConstantInteger(atoi(yytext));
                  return INTEGER; }
{realNumber}      { yylval.expr = new ConstantFloat(atof(yytext));
                  return REAL; }

```

*MINPAS* defines five arithmetic operators and six relational operators. In the Lex specification, occurrences of these operators cause the creation of the corresponding **Operator** objects. The primary behavior provided by an **Operator** is the creation the appropriate expression, based on operand type. Since *MINPAS* allows mixed-mode expressions, the necessary coercions are also performed. For example, sending the message `expression` to an instance of `Plus` with an integer expression and a real expression as arguments returns an `ADDF` expression, after coercing the integer expression to a float expression.

```

"+"               { yylval.op = new Plus();           return ADDOP; }
"-               { yylval.op = new Minus();          return ADDOP; }
"*               { yylval.op = new Multiply();       return MULOP; }
"/               { yylval.op = new Divide();         return MULOP; }
"mod"           { yylval.op = new Modulo();          return MULOP; }
"="             { yylval.op = new Equal();           return RELOP; }
"<>"            { yylval.op = new NotEqual();        return RELOP; }
"<"             { yylval.op = new LessThan();        return RELOP; }
"<="           { yylval.op = new LessOrEqual();     return RELOP; }
">"             { yylval.op = new GreaterThan();    return RELOP; }
">="           { yylval.op = new GreaterOrEqual(); return RELOP; }

```

When an identifier is encountered in an expression, the message `identifierAddress` is sent to the `currentScope` to retrieve an expression representing the address of that identifier. If a declaration for the identifier cannot be found in any of the open scopes, OCS issues an error message. An expression for the value stored in an identifier is obtained by sending the `dereference` message to its address expression.

```

expr             : simple_expr
                  { $$ = $1; }
                | simple_expr RELOP simple_expr
                  { $$ = $2->expression($1, $3); }
                ;

simple_expr       : term
                  { $$ = $1; }
                | simple_expr ADDOP term
                  { $$ = $2->expression($1, $3); }
                ;

```

```

term          : factor
               { $$ = $1; }
| term MULOP factor
               { $$ = $2->expression($1, $3); }
;

factor        : INTEGER
               { $$ = $1; }
| REAL
               { $$ = $1; }
| ID LPAREN expr_list RPAREN
               { $$ = new FunctionCall($1, $3); }
| id_addr
               { $$ = $1->dereference(); }
| LPAREN expr RPAREN
               { $$ = $2; }
;

id_addr       : ID
               { $$ = currentScope->identifierAddress($1); }
;

```

A **Statement** is a combination of expressions and other statements. An **AssignmentStatement**, for example, is created from an address expression and a value expression. The **AssignmentStatement** constructor performs type checking and the necessary coercions. A statement providing input into a variable can be created by sending the **inputStatement** message to an expression for the variable's address. A statement providing output for an expression can be created by sending the **outputStatement** message to the expression. Multiple statements can be combined into a **StatementList**.

```

stmt_list     : stmt
               { $$ = new StatementList($1); }
| stmt_list stmt
               { $$ = $1->addLast($2); }
;

stmt          : compound_stmt
               { $$ = $1; }
| assign_stmt
               { $$ = $1; }
| if_stmt
               { $$ = $1; }
| while_stmt
               { $$ = $1; }
| read_stmt
               { $$ = $1; }
;

```

```

| write_stmt
  { $$ = $1; }
| return_stmt
  { $$ = $1; }
;

compound_stmt : BEGIN_STMT stmt_list END_STMT
  { $$ = new CompoundStatement($2); }
;

assign_stmt : id_addr ASSIGN expr SEMICOLON
  { $$ = new AssignmentStatement($1, $3); }
;

if_stmt : IF expr THEN stmt ELSE stmt
  { $$ = new ConditionalStatement($2, $4, $6); }
| IF expr THEN stmt
  { $$ = new ConditionalStatement($2, $4); }
;

while_stmt : WHILE expr DO stmt
  { $$ = new PretestLogicalLoop($2, $4, false); }
;

read_stmt : READ id_addr SEMICOLON
  { $$ = $2->inputStatement(); }
;

write_stmt : WRITE expr SEMICOLON
  { $$ = $2->outputStatement(); }
;

return_stmt : RETURN expr SEMICOLON
  { $$ = new ReturnStatement($2); }
;

```

A **Block** represents a section of code with optional variable and function declarations. A **Subprogram** is created from a **SubprogramType** and a **Block**.

```

block : variable_decl function_decl BEGIN_STMT stmt_list END_STMT
  { $$ = new Block(currentScope, $4, $2);
    ... }
;

function : function_hdg SEMICOLON block
  { $$ = new Subprogram($1, $3); }

```

```

;
program      : program_hdg block PERIOD
              { $$ = new Subprogram($1, $2);
                ... }
;

```

### 4.3 Generating Code

Code generation is initiated by sending the `genCode` message to a `Subprogram`. When the `Subprogram` receives the message, it sends the `optimize` message to itself before actually outputting the assembly code.

```

program      : program_hdg block PERIOD
              { $$ = new Subprogram($1, $2);
                $$->genCode(); }
;

```

### 4.4 Compiling a Program

The *MINPAS* program `area.mp` shown below computes the area of a circle.

```

program
var
    radius : real;

begin
    read radius;
    write 3.14159 * radius * radius;
end.

```

Figure 6 shows the sequence of actions, as defined in the Lex and Yacc specifications, that occur during compilation of this program.

Appendix D provides a *MINPAS* source program for computing the greatest common divisor and Appendix E shows the corresponding assembly code generated for the Motorola 68030.

## 5 Implementation

The implementations of the methods in the OCS classes are designed to be replaceable, while the interface to the methods remains fixed. In fact the true flexibility of the tools is exhibited by having multiple implementations of the methods. The current version contains two implementations of the methods that generate code, one for the Motorola 68030 and one for the National Semiconductor 32032. Only a change

<i>Action Number</i>	<i>Construct<sup>a</sup> Recognized</i>	<i>Action</i>
1	"program"	new Symbol("main")
2	<program_hdr>	new SubprogramType(#1)
3		currentScope = new GlobalScope()
4	"radius"	new Symbol("radius")
5	"real"	new FloatType()
6	<decl>	new VariableAttribute(#5)
7		new Declaration(#4, #6)
8	<variable_list>	new DeclarationList(#7)
9	<variable_decl>	currentScope→insert(#8)
10	<function_decl>	new StatementList()
11	"radius"	new Symbol("radius")
12	<id_addr>	currentScope→identifierAddress(#11)
13	<read_stmt>	#12→inputStatement()
14	<stmt_list>	new StatementList(#13)
15	"3.14159"	new ConstantFloat(3.14159)
16	"*"	new Multiply()
17	"radius"	new Symbol("radius")
18	"*"	new Multiply()
19	<id_addr>	currentScope→identifierAddress(#17)
20	<factor>	#19→dereference()
21	<term>	#16→expression(#15, #20)
22	"radius"	new Symbol("radius")
23	<id_addr>	currentScope→identifierAddress(#22)
24	<factor>	#23→dereference()
25	<term>	#18→expression(#21, #24)
26	<write_stmt>	#25→outputStatement()
27	<stmt_list>	#14→addLast(#26)
28	<block>	new Block(currentScope, #27, #10)
29		currentScope = currentScope→parent()
30	<program>	new Subprogram(#2, #28)
31		#30→genCode()

<sup>a</sup>"construct1" indicates a construct recognized by Lex. (construct2) indicates a construct recognized by Yacc.

Figure 6: Action Sequence when Compiling the area.mp Program.

in a command to the linker is required to create a compiler for a different target language. This reuse of interface is a powerful feature of object-oriented programming. The following subsections describe the current implementations.

## 5.1 Symbol Management

A `Symbol` exhibits limited behavior. Relational operators provide various comparison operations between two symbols. Two implementations of the operators are available, offering either case-sensitive or case-insensitive comparisons. A `Symbol` can be cast to an integer. The implementation of this method is actually a hash function designed to allow symbols to be inserted into a hash table.

The information associated with an identifier depends upon its usage. A constant identifier has a type and a value. A variable has a type and possibly its location represented as an offset into an activation record. `Attribute` has several subclasses to accommodate these variations including `ConstantAttribute`, `LabelAttribute`, `SubprogramAttribute`, `TypeAttribute`, and `VariableAttribute`.

Scope refers to the text of a program unit, such as a function or procedure, in which a set of identifiers is visible. At a particular line in a program the innermost scope enclosing the line is the current scope. The scope immediately enclosing the current scope is its parent scope. When an identifier is encountered, a search for the declaration begins in the current scope. If no declaration is found the parent scope is searched. The process continues until a declaration is found or the outermost scope is reached without finding a declaration, in which case the identifier is undefined.

`Scope` is an abstract superclass with several subclasses.

- `GlobalScope` represents the outermost scope. Identifiers declared in a `GlobalScope` are available to all other scopes.
- `SubprogramScope` represents a procedure or function. At runtime `SubprogramScope` generates an activation record for storing parameters and local identifiers.
- `BlockScope` represents a section of code that allows declarations but do not cause an activation record to be created. Blocks as defined in the C programming language are an example of this.
- `RecordScope` represents a scope in which the fields of a record can be accessed by the field name alone with the record variable name implied. The Pascal `WITH` statement is an example.
- `NullScope` indicates that no scopes are currently open.

A global variable `currentScope` always points to the current scope. Initially, it points to an instance `NullScope`. Opening a new scope is accomplished by creating a new instance of the appropriate scope class and assigning it to the variable `currentScope`. For example, when the syntax analyzer recognizes the beginning of the program code, a `GlobalScope` is created. When a procedure is recognized, a `SubprogramScope` is created which maintains a pointer to its parent scope. Scopes may be nested in this manner to an arbitrary depth, creating, in effect, a linked list of open scopes. Closing a scope is simply a matter of sending the message `parent` to `currentScope`, assigning the result to `currentScope`.

Figure 7 illustrates the scope objects that would represent the following C code fragment:

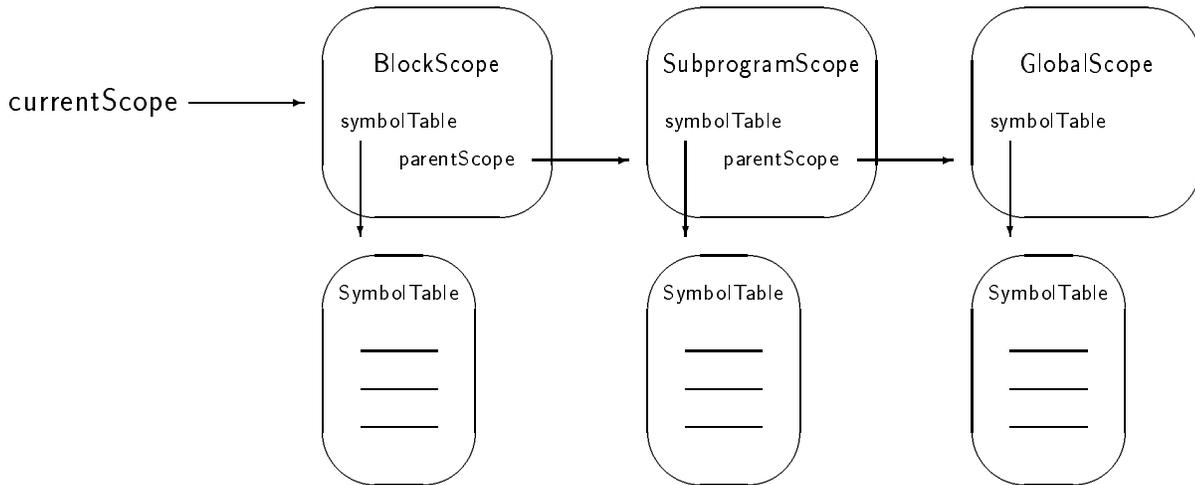


Figure 7: Scope Implementation.

```

#include <stdio.h>

int a;                /* declared in the GlobalScope */

main()
{
    int b;            /* declared in the SubprogramScope */
    ...
    {
        int c;        /* declared in the BlockScope */
    }
}
  
```

In order to fulfill its responsibilities, a class often elicits the services of Facilitator or Helper classes. Indifferent to the problem domain, these classes provide general services that can be shared among projects. This allows code reusability at the module level. Managing symbols in a compiler requires a number of rather general services. Associating two objects as a key-value pair, maintaining a list of objects, and maintaining a dictionary of associated key-value objects are some of the services required. Modeled after similar classes in Smalltalk [GR83], OCS includes four Facilitator classes: `List`, `Association`, `AssociationList`, and `Dictionary`. These are implemented as template classes. To make it more convenient to use these classes, type definitions are provided.

```

typedef List<Declaration *>      DeclarationList;
typedef List<Expression *>      ExpressionList;
typedef List<Statement *>       StatementList;
  
```

```

typedef List<Symbol *>           SymbolList;
typedef List<Type *>           TypeList;

typedef Association<Symbol *, Attribute *> SymbolEntry;
typedef AssociationList<Symbol *, Attribute *> SymbolEntryList;
typedef Dictionary<Symbol *, Attribute *> SymbolTable;

```

Consider the following ANSI C code fragment:

```

int foo(int a, double b)
{
    int i, j, k;
    double x, y, z;
    ...
}

```

Two groups of variable declarations are present, the formal parameters for the function `foo` and the automatic variables declared local to `foo`. Each variable has three objects directly related to it: a `Symbol`, a `Type`, and a `VariableAttribute`. The `Type` is maintained as an instance variable in `VariableAttribute`. The `Symbol` and `VariableAttribute` are then combined into a `Declaration` and multiple declarations are collected into a `DeclarationList`.

Up to this point the processing for both variable groups is identical, with a separate `DeclarationList` created for each group. The handling of each list can now be specialized. The list of formal parameters serves two distinct purposes. The first purpose is type checking. A `SubprogramType` is created for the function `foo`. The formal parameter list is maintained as an instance variable in the `SubprogramType` so that subsequent invocations of the function can be checked for the correct count and type of arguments. The second purpose of the formal parameter list is to provide the information required at runtime. At runtime the actual parameters are stored in an activation record at offsets from the frame pointer. The `VariableAttribute` for each variable contains the offset. The parameters for a `SubprogramScope` are set by the constructor for `SubprogramScope`, which sends itself the `setParameters` message. The `SubprogramScope` sets the offset of each variable, based on its size and position in the list and adds the variables to its symbol table. The `SymbolTable` is a `Dictionary` whose key is a `Symbol` and value is an `Attribute`. The dictionary is implemented as an array of lists. This implementation provides a natural representation for a hash table using separate chaining to resolve collisions.

Local variables are also stored in an activation record at runtime, although they begin at a different offset than parameters. The local variables are added to the scope using the `insert` message. When this message is sent the scope sets the offset of each variable and adds them to the symbol table.

Producing an expression for the address of a variable is one of the most common services provided by `Scope`. The method `identifierAddress` is specified in `Scope` and implemented in each of its subclasses. In `GlobalScope`, if the `Symbol` is found in the symbol table a `GlobalIdentifier` is created and returned; otherwise an error condition is generated for an undefined variable. In `SubprogramScope` if the `Symbol` is found a `LocalIdentifier` is created and returned. A `LocalIdentifier` maintains an instance variable indicating the number of static

levels to be traversed so that the appropriate code can be generated. If the `Symbol` is not found, the message is passed to the parent `Scope`.

`RecordScope` provides an elegant solution to the `WITH` statement in Pascal. In a record type definition the fields of a record are handled in the same fashion as other variables and placed in a `DeclarationList` and stored in a `RecordType`. Instead of representing the offset from the frame pointer, the offset in the `VariableAttribute` for a field is the offset from the beginning of the record. When the `identifierAddress` message is sent to a `RecordScope` the search begins in the `DeclarationList` for the `RecordType`. If the `Symbol` is found an expression for the record field is created and returned; otherwise the message is simply passed on to the parent scope.

## 5.2 Intermediate Representation

The compiler writer creates an intermediate representation of the source program during syntax analysis. Creating an intermediate representation rather than target code directly permits the isolation of machine dependent details, easing the task of retargeting a compiler and permitting machine independent code optimizations. Several forms of representation have been used including abstract syntax trees, three-address code, and tuples. Figure 8 illustrates three internal representation for the Pascal assignment statement:

```
H := H + 1 / N;
```

Both variables `H` and `N` are declared as type `real`.

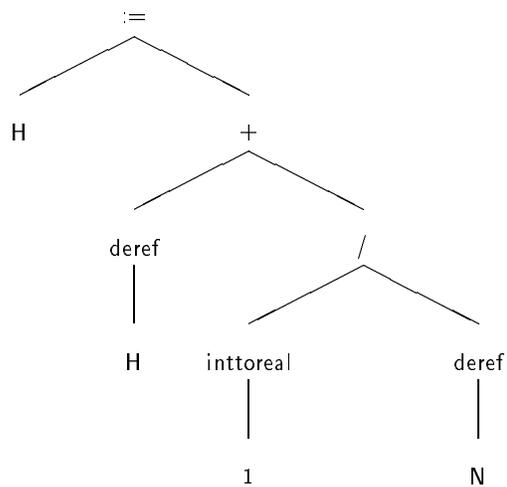
Each representation presents a particular view of the translated program that dictates the manner in which further processing will proceed. The abstract syntax tree presents a hierarchical view of the program. Optimization and code generation are a matter of traversing the tree. The three-address code presents the assembly language of a virtual machine. Further processing isolates blocks of code for transformation. The object-oriented representation can be viewed as nested levels of encapsulation. The outer level objects effectively hide the details of the inner level objects.

The internal representation used by OCS is divided primarily into two class hierarchies, `Expression` and `Statement`. An `Expression` represents an expression in a programming languages such as the multiplication of two floating point values, a global variable, or a function call. An expression is responsible for knowing its type, optimizing itself, and generating code for itself. A `Statement`, likewise, represents a statement in a programming language and is responsible for optimizing itself and generating code for itself. Both `Expression` and `Statement` have a number of subclasses. The subclasses are designed to be general enough to handle a wide variety of source language constructs.

Repetition statements provide a good example of variability. The categories of iterative statements center around two design questions: [Seb89]

1. How is iteration controlled?
2. Where does the control mechanism appear?

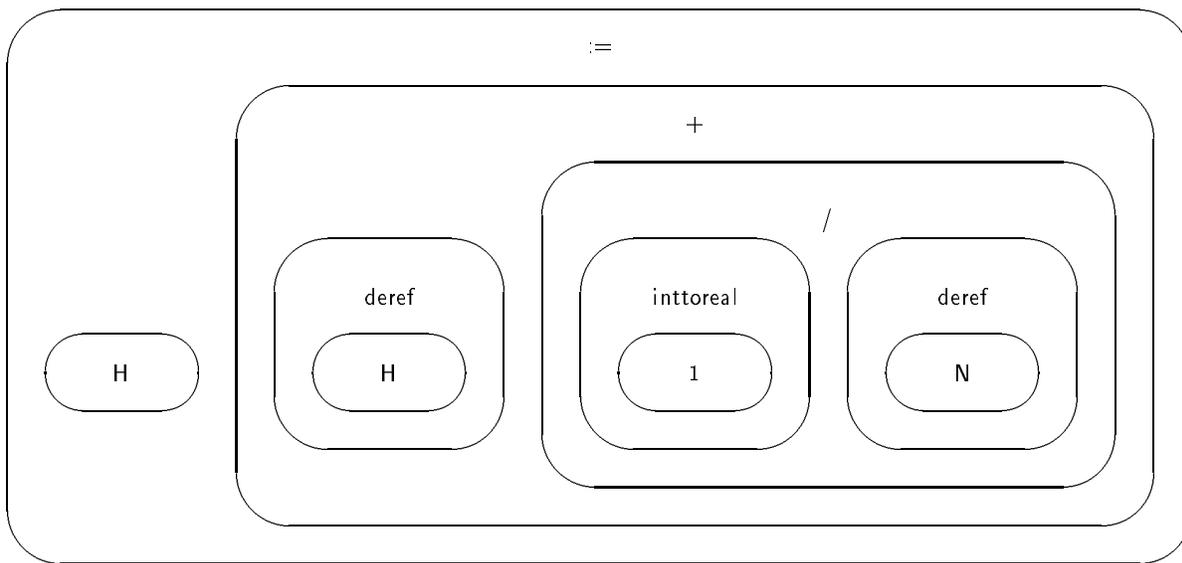
Iteration can be controlled by a counting mechanism or the evaluation of a Boolean expression. The control mechanism can be located at the beginning of the loop, the end of the loop, or even at some user specified location within the loop.



$t1 := \text{inttoreal } 1$   
 $t2 := t1 / N$   
 $t3 := H + t2$   
 $H := t3$

*Three-Address Code*

*Abstract Syntax Tree*



*Object-oriented Representation*

Figure 8: Intermediate Representations.

The class `LogicallyControlledLoop` has two subclasses: `PretestLogicalLoop` and `PosttestLogicalLoop` as shown by the following class definitions:

```
class LogicallyControlledLoop : public Statement
{
public:
    virtual Statement * optimize();
    virtual void      genCode();

protected:
    Expression      * expression;
    Statement       * statement;
    int             terminateCondition;
};

class PretestLogicalLoop : public LogicallyControlledLoop
{
public:
    PretestLogicalLoop(Expression *, Statement *, int);
    virtual void      genCode();
};

class PosttestLogicalLoop : public LogicallyControlledLoop
{
public:
    PosttestLogicalLoop(Expression *, Statement *, int);
    virtual void      genCode();
};
```

`LogicallyControlledLoop` contains three instance variables:

- `expression` points to an instance of a Boolean expression to be tested on each iteration of the loop.
- `statement` points to an instance of a statement to be executed on each iteration.
- `terminateCondition` contains a Boolean value specifying that loop termination should occur when the expression evaluates to either true or false.

Two specialized subclasses are provided, `PretestLogicalLoop` and `PosttestLogicalLoop`, which inherit all their behavior except code generation. The table in Figure 9 shows how these classes can be used to represent loop statements in various languages.

<i>Language</i>	<i>Statement</i>	<i>Class</i>	<i>terminateCondition</i>	<i>Example</i>
Ada	while	PretestLogicalLoop	False	while i < j loop i := i * 2; end loop;
C	while	PretestLogicalLoop	False	while (i < j) i = i * 2;
	do	PosttestLogicalLoop	False	do i = i * 2; while (i < j);
Pascal	while	PretestLogicalLoop	False	while i < j do i := i * 2;
	repeat	PosttestLogicalLoop	True	repeat i := i * 2; until i >= j;

Figure 9: Examples of logical loop statements.

### 5.3 Type Checking

Type checking tests the compatibility of identifier usage in programming language constructs. Common forms include checking the types of the arguments to subprogram calls with the subprogram declarations and checking expressions involving arithmetic operators. OCS isolates the details of type checking in the implementation of the class methods. For example when a `ProcedureCall` is created, the constructor automatically checks the number and type of each argument expression with the declaration for the subprogram type.

A mixed-mode expression is a single expression that contains operands of different types. Many languages allow this feature, including FORTRAN, C, and Pascal. To resolve this problem OCS uses *coercive generality*, a technique used in Smalltalk-80 [GR83]. The goal is to minimize the loss of information and assure that commutative operations produce the same result regardless of operand order. Each numeric data type is assigned a unique generality number. Rather than having a significance on its own, the generality number imposes a linear ordering on the types with the most general type having the largest number. To build an arithmetic expression, the message `expression` is sent to an `Operator` with the right and left hand expressions passed as arguments. The two arguments are passed as reference parameters to the function `matchTypes`. If the generality numbers for the two expressions are equal, no coercion is necessary. If the numbers are not equal, the expression with the lower generality number is coerced to the type of the expression with the higher number.

The following C code fragment serves as an example:

```
main()
{
    double x, y;
```

```

    int i;
    ...
    y = x * i;
    ...
}

```

The message `expression` is sent to an instance of `Multiply` with an expression for the value of `x` as the left operand and an expression for the value of `i` as the right operand. The two operands are passed to the `matchTypes` function which sends the `generality` message to each of them. The left operand returns the greater generality number so the message `coerce` is sent to the type of the left operand with the right operand as an argument. The `FloatType` object sends the message `expression` to an instance of `ToFloat`. The message returns a new expression for converting the value in `i` from an integer to a float. Figure 10 illustrates the implementation of the methods.

## 5.4 Optimization

Optimization attempts to improve the intermediate representation of the source program so that better, i.e., faster or smaller, code can be generated. In OCS statements and expressions optimize themselves automatically by sending themselves the `optimize` message. The compiler writer chooses the level of optimization desired by selecting the appropriate compiled implementation.

As currently implemented, OCS provides two implementations of optimization. The first actually provides no optimization. The `optimize` message sent to an expression or statement merely returns the receiver. The second implementation provides peephole optimizations, such as constant folding, algebraic simplification, and reduction in strength. The table in Figure 11 lists examples of some of these optimizations.

Some object-oriented languages provide features for testing the dynamic class of an object. Object Pascal [Tes85] provides the function `member(anObject, aClass)` that returns true if `anObject` is a member of `aClass`. In Smalltalk [GR83], all objects respond to the message `isMemberOf`. C++ [Str91] provides no such feature. When performing optimizations it is often necessary to know an object's dynamic class. Every class in OCS is assigned a unique integer value defined as a global constant whose name is the class name, beginning with a lower case letter, followed by 'Class'. For example, the unique identifier constant for the `ProcedureCall` class is `procedureCallClass`. This variable is returned in response to the message `classId`. The superclass for each hierarchy defines a method `isMemberOf` which can be used to test the dynamic class of an object. Figure 12 shows how this method is used to perform constant folding on an `ADDI` expression. First, the message `isMemberOf(constantIntegerClass)` is sent to the left and right operand expressions. If both messages return true, the `integerValue()` is sent to the expressions to obtain the integer values. The left and right operand expressions are deleted and a new `ConstantInteger` expression is created for the sum of the values.

## 5.5 Code Generation

During code generation the intermediate representation is translated into a sequence of instructions in the target language. Target machine dependencies are isolated in the implementation of the code generation methods. This means that a compiler can be retargeted for a new machine by creating a new implementation of only these methods. The interface, i.e., the class definitions, remain fixed. This reuse of interface allows reuse of high level specifications while replacing low level details.

```

Expression * Multiply::expression(Expression * left, Expression * right)
{
    Type * type = matchTypes(left, right);

    if (type->isMemberOf(floatTypeClass))
        return new MULF(left, right);
    else ...
}

static Type * matchTypes(Expression *& left, Expression *& right)
{
    Type * leftType = left->type();
    Type * rightType = right->type();

    if (leftType->generality() > rightType->generality())
    {
        right = leftType->coerce(right);
        return leftType;
    }
    else if (leftType->generality() < rightType->generality())
    {
        left = rightType->coerce(left);
        return rightType;
    }
    else
        return leftType;
}

int FloatType::generality()
{ return floatGenerality; }

int IntegerType::generality()
{ return integerGenerality; }

Expression * FloatType::coerce(Expression * expr)
{ return ToFloat().expression(expr); }

```

Figure 10: Implementing coercive generality.

<i>Optimization</i>	<i>Expression Before Optimization</i>	<i>Expression After Optimization</i>
Constant Folding	$4 + 8$	12
	$8 * 2$	16
Algebraic Simplification	$x + 0$	$x$
	$x * 1$	$x$
Reduction in Strength	$x * 2$	$x \ll 1$
	$x / 16$	$x \gg 4$

Figure 11: Peephole Optimizations.

```

if (leftOpExpr->isMemberOf(constantIntegerClass) &&
    rightOpExpr->isMemberOf(constantIntegerClass))
{
    value = leftOpExpr->integerValue() + rightOpExpr->integerValue();
    delete leftOpExpr;
    delete rightOpExpr;
    return new ConstantInteger(value);
}

```

Figure 12: Constant folding.

The following discussion illustrates one implementation of the code generation methods. This implementation generates assembly code for a Motorola 68030. Code generation is initiated by sending the `genCode` message to a `SubprogramStatement`. Each `SubprogramStatement` object maintains three instance variables:

- `scope` which is a pointer to the corresponding scope object for the subprogram
- `subprograms` which is a pointer to a list of subprograms declared within the subprogram
- `statements` which is a pointer to a list of statements representing the body of the subprogram.

The `genCode` method begins by sending the `genCode` message to `subprograms` to generate their code. The subroutine header is output, including alignment and global directives and the subroutine label. Local storage is allocated on the activation record. At this point, the `genCode` message is sent to `statements` to generate the code for the body of the subroutine. The subroutine footer is output, which deallocates the stack space used by local variables, restores the old frame pointer, and returns execution to the point of the call. Finally, the message `genStorage` is sent to `scope`. If `scope` is an instance of `GlobalScope`, a define constant assembler directive is output for each constant in the scope's symbol table and a define storage directive is output for each variable. If `scope` is an instance of `SubprogramScope`, a define constant assembler directive is output for each constant, after mangling its name in order to avoid conflicts with global identifiers.

Statements are composed of expressions and other statements. It is at the expression level that the real work of code generation is performed. The implementation being discussed generates a simple stack machine code. As an example, consider the following Pascal procedure.

```
procedure swap(var a, b : integer);
var
    tmp : integer;

begin
    tmp := a;
    a := b;
    b := tmp
end;
```

Since no procedures and no functions are defined within this procedure, the subprogram list is empty. The subroutine header is output as follows:

```
version 2
text
    lalign 2
    global _swap
_swap:
    link.w %a6,&-4
```

The last line listed above sets up the activation record for the procedure, allocating four bytes of storage for the local variable `tmp`.

The three assignment statements in the `swap` procedure generate the following three assembly instructions:

```
mov.l  ([12,%a6]),-4(%a6)
mov.l  ([16,%a6]),([12,%a6])
mov.l  -4(%a6),([16,%a6])
```

For an assignment statement, the assembly instructions generated depend upon the type of the source and destination expressions. In the `swap` procedure, `a`, `b`, and `tmp` are declared as type `integer`. Therefore the assignment statements must generate code for integer assignments. OCS provides separate assignment expressions for each type. The `AssignmentStatement` maintains a pointer to an appropriate assignment expression object, in this case an instance of `ASSIGNI`. When generating code, `ASSIGNI` sends the `lvalue` message to the destination expression and the `rvalue` message to the source expression to obtain the code for the left side operand and right side operand, respectively.

In the first assignment the source operand is `([12,%a6])`, representing the parameter variable `a`. The variable `a` is stored in the activation record at an offset of 12 from the frame pointer, `a6`. Since `a` is passed as a `VAR` parameter, its occurrences within the procedure are treated as dereferenced pointers. The memory indirect with base displacement addressing mode provides a concise representation for `a`. The destination operand, `-4(%a6)`, represents the local variable `tmp`, which is stored in the activation record at an offset of `-4` from the frame pointer. The based addressing mode is used to reference `tmp`. Finally, since the values stored are integers, the `mov.l` instruction is used.

After generating code for the three assignment statements, `genCode` outputs the subroutine footer code which frees the activation record and returns control to the caller:

```
L2:      unlk    %a6
         rts
```

Having output the code for the procedure, the `genCode` method for `SubprogramStatement` concludes by sending the `genStorage` message to `scope`. Had any constants been declared in `swap`, statements would have been output to allocate storage for them.

## 6 Conclusion

There exists a conflict between generalization and specialization in the development of significant software systems. Generalization seeks to maximize reusability. Specialization seeks to maximize applicability. Object-oriented programming supports several approaches to resolving this tension.

- Parameterized classes provide high-level reusability by defining a template for a generic class which can then be instantiated for various types. A linked list, for example, requires the same functionality regardless of the type of elements stored in the list.

- Class hierarchies provide the structure for both inheritance (generalization) and overriding (specialization). Common behavior is factored toward the root of the tree while exceptional behavior is drawn toward the leaves. Extensibility is enhanced by permitting new subclasses to be defined by specifying only the exceptional behavior.
- The separation of interface and implementation allows a different type of reuse. A single interface can be shared by a number of implementations. Generalization is provided at compile time by the class definition while specialization is deferred until link time.

OCS demonstrates how compiler construction can benefit from these approaches.

- C++ template classes are used for various container classes. For example, the template class `List` is used to instantiate lists for `Symbol`'s, `Type`'s, `Declaration`'s, `Expression`'s, and `Statement`'s.
- Class hierarchies are used extensively to capture commonality. The class `Subprogram`, for example, shares the same data and much of the behavior of its superclass `Block`. The use of classes also allows compiler writers to easily extend the tools to accommodate new language features.
- Multiple implementations of the methods that generate code allow a compiler to be retargeted by simply changing the link command.

In OCS, compilation is centered around the objects being manipulated, e.g., expressions and statements, instead of the processes being performed, e.g., optimization and code generation. This distribution of control reduces complexity. To implement the code generation for a conditional statement, it is only necessary to consider that one particular statement type. Extendibility is enhanced for the same reason. If a new statement or expression type is needed it can be added without affecting existing code. Multiple method implementations allow low level parts to be replaced without affecting the high level parts, namely the interface.

At present OCS consists of 174 classes. Figure A shows the complete list. Compilers have been constructed for large subsets of Pascal and ANSI C for the Motorola 68030 and the National Semiconductor 32032.

## 6.1 Future Work

So far, only imperative languages have been addressed, with some preliminary work on providing for object-oriented language features. The full spectrum of object-oriented features should be examined. Non-traditional target architectures should also be addressed, particularly RISC-based systems. The current implementation allows code generation at the subprogram level. This should be enhanced to allow code generation at the statement level. Much work needs to be done to more fully develop the optimization methods.

## 6.2 Acknowledgements

I would like to thank my major professor Timothy Budd for providing the original ideas for this project and for his help throughout the project. I thank Jim Shur for the stimulating conversations we had at the beginning of the project. I would also like to thank Rajeev Pandey for the many suggestions he offered on earlier drafts of this paper.

## References

- [ASU86] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley Publishing Company, Reading, Massachusetts, 1986.
- [CG87] John H. Crawford and Patrick P. Gelsinger. *Programming the 80386*. SYBEX Inc., Alameda, California, 1987.
- [CN91] Brad J. Cox and Andrew J. Novobilski. *Object-Oriented Programming: An Evolutionary Approach*. Addison-Wesley Publishing Company, Reading, Massachusetts, second edition, 1991.
- [DS91] Charles Donnelly and Richard Stallman. Bison: The yacc-compatible parser generator. Technical report, Free Software Foundation, Cambridge, MA, 1991.
- [FL91] Charles N. Fischer and Richard J. LeBlanc, Jr. *Crafting A Compiler With C*. The Benjamin/Cummings Publishing Company, Inc., Redwood City, California, 1991.
- [GHL<sup>+</sup>92] Robert W. Gray, Vincent P. Heuring, Steven P. Levi, Anthony M. Sloane, and William M. Waite. Eli: A complete, flexible compiler construction system. *Communications of the ACM*, 35(2):121–131, February 1992.
- [GR83] Adele Goldberg and David Robson. *Smalltalk-80: The Language and Its Implementation*. Addison-Wesley Publishing Company, Reading, Massachusetts, 1983.
- [Hol90] Allen I. Holub. *Compiler Design In C*. Prentice Hall, Englewood Cliffs, New Jersey, 1990.
- [Joh78] Stephen C. Johnson. Yacc: Yet another compiler-compiler. Technical report, Bell Laboratories, Murray Hill, New Jersey, 1978.
- [JW85] Kathleen Jensen and Niklaus Wirth. *Pascal User Manual and Report*. Springer-Verlag, New York, third edition, 1985.
- [KL89] Won Kim and Frederick H. Lochovsky, editors. *Object-Oriented Concepts, Databases, and Applications*. Addison-Wesley Publishing Company, Reading, Massachusetts, 1989.
- [KP84] Brian W. Kernighan and Rob Pike. *The UNIX Programming Environment*. Prentice Hall Incorporated, New York, 1984.
- [LCH<sup>+</sup>80] Bruce W. Leverett, Roderic G. G. Cattell, Steven O. Hobbs, Joseph M Newcomer, Andrew H. Reiner, Bruce R. Schatz, and William A. Wulf. An overview of the production-quality compiler-compiler project. *IEEE Computer*, pages 38–49, August 1980.
- [LS75] M. E. Lesk and E. Schmidt. Lex - a lexical analyzer generator. Technical report, Bell Laboratories, Murray Hill, New Jersey, 1975.
- [Mey88] Bertrand Meyer. *Object-Oriented Software Construction*. Prentice Hall Incorporated, New York, 1988.
- [Mic88] Josephine Micallef. Encapsulation, reusability and extensibility in object-oriented programming languages. *Journal of Object-Oriented Programming Languages*, 1(1):12–35, 1988.

- [Pax90] Vern Paxson. Flex - fast lexical analyzer generator. Technical report, Free Software Foundation, Cambridge, MA, 1990.
- [PDC92] T. J. Parr, H. G. Dietz, and W. E. Cohen. PCCTS reference manual version 1.00. *SIGPLAN Notices*, 27(2):88–165, February 1992.
- [PW88] Lewis J. Pinson and Richard S. Wiener. *An Introduction to Object-Oriented Programming and Smalltalk*. Addison-Wesley Publishing Company, Reading, Massachusetts, 1988.
- [Pys88] Arthur B. Pyster. *Compiler Design and Construction*. Van Nostrand Reinhold Company, New York, 1988.
- [Seb89] Robert W. Sebesta. *Concepts Of Programming Languages*. The Benjamin/Cummings Publishing Company, Inc., Redwood City, California, 1989.
- [Str91] Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley Publishing Company, Reading, Massachusetts, second edition, 1991.
- [Tes85] Larry Tesler. Object pascal report. Technical report, Apple Computers, Inc., 1985.
- [TKLJ89] Andrew S. Tanenbaum, M. Frans Kaashoek, Koen G. Langendoen, and Criel J. H. Jacobs. The design of very fast portable compilers. *SIGPLAN Notices*, 24(11):125–131, November 1989.
- [TSKS83] Andrew S. Tanenbaum, Hans Van Staveren, E. G. Keizer, and Johan W. Stevenson. A practical tool kit for making portable compilers. *Communications of the ACM*, 26(9):654–660, September 1983.
- [Wak89] John F. Wakerly. *Microcomputer Architecture and Programming: The 68000 Family*. John Wiley and Sons, Inc., New York, 1989.
- [Weg86] Peter Wegner. Classification in object-oriented systems. *SIGPLAN Notices*, 21(10):173–182, October 1986.

## A OCS Class Hierarchy

Scope	BlockScope GlobalScope SubprogramScope NullScope RecordScope	LogicallyControlledLoop PosttestLogicalLoop PretestLogicalLoop
Declaration		NullStatement
Specifier	TypeSpecifier UserTypeSpecifier AccessSpecifier SignSpecifier SizeSpecifier StorageSpecifier StructSpecifier	ProcedureCall ReturnStatement
Declarator		Expression
AbstractDeclarator	SpecifierDeclarator PointerDeclarator ArrayDeclarator FunctionDeclarator	BinaryExpression
Symbol		ADDC SUBC MULC DIVC MODC EQC NEC LTC LEC GTC GEC ASSIGNC ADDF SUBF MULF DIVF EQF NEF LTF LEF GTF GEF ASSIGNF ADDI SUBI MULI DIVI MODI EQI NEI LTI LEI GTI GEI ASSIGNI SHLI SHRI ADDP SUBP MULP DIVP MODP EQP NEP LTP LEP GTP
Attribute	ConstantAttribute FloatAttribute IntegerAttribute StringAttribute LabelAttribute SubprogramAttribute TypeAttribute VariableAttribute	
Type	ArrayType StringType CharType ClassType FloatType IntegerType PointerType RecordType SubprogramType UnknownType VoidType	
Statement	ExpressionStatement AssignmentStatement Block Subprogram CompoundStatement ConditionalStatement CounterControlledLoop MultipleEvalCounterLoop SingleEvalCounterLoop	

	GEP	PreIncrement
	ASSIGNP	PostIncrement
	LAND	PreDecrement
	LOR	PostDecrement
	ConstantChar	
	ConstantFloat	
	ConstantInteger	
	ConstantString	
	ExpressionSequence	
	FunctionCall	
	GlobalIdentifier	
	LocalIdentifier	
	NullExpression	
	UnaryExpression	
	CTOI	
	DEREFC	
	NEGF	
	FTOI	
	DEREFF	
	NEGI	
	ITOC	
	ITOF	
	ITOP	
	DEREFI	
	PREINCI	
	POSTINCI	
	PREDECI	
	POSTDECI	
	DEREFP	
	LNOT	
	ADDR	
Operator		
	Plus	
	Minus	
	Multiply	
	Divide	
	IntegerDivide	
	RealDivide	
	Modulo	
	LogicalAnd	
	LogicalOr	
	LogicalNot	
	Equal	
	NotEqual	
	LessThan	
	LessOrEqual	
	GreaterThan	
	GreaterOrEqual	
	Assign	
	AddressOf	
	Dereference	
	ToChar	
	ToFloat	
	ToInteger	
	ToPointer	

## B Lex Specification for *MINPAS*

```
%{
#include <stdio.h>
#include <stdlib.h>
#include "yacc.h"
#include "ocs.h"
#include "yac_mp.h"
%}

comment          "{ "[^"]*" }"
id               [A-Za-z][0-9A-Za-z]*
integerNumber    [+\\-]?[0-9]+
realNumber       [+\\-]?[0-9]+(\\. [0-9]+)?(E[+\\-]?[0-9]+)?
whiteSpace       [ \\t\\n]

%%

{whiteSpace}+   { /* ignore white space */ }
{comment}+     { /* ignore comments */ }
"begin"        { return BEGIN_STMT; }
"end"          { return END_STMT; }
"function"     { return FUNCTION; }
"read"         { return READ; }
"write"        { return WRITE; }
"if"           { return IF; }
"then"         { return THEN; }
"else"         { return ELSE; }
"while"        { return WHILE; }
"do"           { return DO; }
"return"       { return RETURN; }
"var"          { return VAR; }
"program"      { yylval.sym = new Symbol("main"); return PROGRAM; }
"integer"      { yylval.type = new IntegerType(); return TYPE; }
"real"         { yylval.type = new FloatType(); return TYPE; }
"+"           { yylval.op = new Plus(); return ADDOP; }
"-"           { yylval.op = new Minus(); return ADDOP; }
"*"           { yylval.op = new Multiply(); return MULOP; }
"/"           { yylval.op = new Divide(); return MULOP; }
"mod"         { yylval.op = new Modulo(); return MULOP; }
"="           { yylval.op = new Equal(); return RELOP; }
"<>"         { yylval.op = new NotEqual(); return RELOP; }
"<"          { yylval.op = new LessThan(); return RELOP; }
"<="         { yylval.op = new LessOrEqual(); return RELOP; }
```

```

">"      { yylval.op = new GreaterThan();    return RELOP; }
">="     { yylval.op = new GreaterOrEqual(); return RELOP; }
":="     { return ASSIGN; }
":"      { return COLON; }
";"      { return SEMICOLON; }
","      { return COMMA; }
"."      { return PERIOD; }
"("      { return LPAREN; }
")"      { return RPAREN; }
{id}     { yylval.sym = new Symbol((char *) yytext);
          return ID; }
{integerNumber} { yylval.expr = new ConstantInteger(atoi(yytext));
                  return INTEGER; }
{realNumber}   { yylval.expr = new ConstantFloat(atof(yytext));
                  return REAL; }
.              { errorAt("unknown character"); }
%%

```

## C Yacc Specification for *MINPAS*

```
%token PROGRAM BEGIN_STMT END_STMT FUNCTION READ WRITE
%token IF THEN ELSE WHILE DO RETURN
%token COLON SEMICOLON COMMA LPAREN RPAREN PERIOD
%token VAR TYPE INTEGER REAL ID
%token ASSIGN ADDOP MULOP RELOP
```

```
{
#include <stdio.h>
#include <stdlib.h>
#include "ocs.h"
#include "yacc.h"
}
```

```
%union
{
Block * block;
Declaration * decl;
DeclarationList * declList;
Expression * expr;
ExpressionList * exprList;
Statement * stmt;
StatementList * stmtList;
Operator * op;
Symbol * sym;
Type * type;
SubprogramType * subpgmType;
}
```

```
%type <stmt> program
%type <subpgmType> program_hdg
%type <block> block
%type <declList> variable_list
%type <decl> decl
%type <stmtList> function_decl
%type <stmt> function
%type <subpgmType> function_hdg
%type <stmtList> stmt_list
%type <stmt> stmt
%type <stmt> compound_stmt
%type <stmt> assign_stmt
%type <stmt> if_stmt
%type <stmt> while_stmt
```

```

%type <stmt>      read_stmt
%type <stmt>      write_stmt
%type <stmt>      return_stmt
%type <exprList>  expr_list
%type <expr>      expr
%type <expr>      simple_expr
%type <expr>      term
%type <expr>      factor
%type <expr>      id_addr
%type <op>        ADDOP
%type <op>        MULOP
%type <op>        RELOP
%type <sym>       PROGRAM
%type <sym>       ID
%type <type>      TYPE
%type <expr>      INTEGER
%type <expr>      REAL

```

```
%start program
```

```
%%
```

```

program      : program_hdg block PERIOD
              { $$ = new Subprogram($1, $2);
                $$->genCode(); }
              ;

program_hdg  : PROGRAM
              { $$ = new SubprogramType($1);
                currentScope = new GlobalScope(); }
              ;

block        : variable_decl function_decl BEGIN_STMT stmt_list END_STMT
              { $$ = new Block(currentScope, $4, $2);
                currentScope = currentScope->parent(); }
              ;

variable_decl : VAR variable_list SEMICOLON
              { currentScope->insert($2); }
              | /* empty */
              { }
              ;

```

```

variable_list : decl
               { $$ = new DeclarationList($1); }
| variable_list SEMICOLON decl
               { $$ = $1->addLast($3); }
;

decl : ID COLON TYPE
      { $$ = new Declaration($1, new VariableAttribute($3)); }
;

function_decl : /* empty */
               { $$ = new StatementList(); }
| function_decl function SEMICOLON
               { $$ = $1->addLast($2); }
;

function : function_hdg SEMICOLON block
          { $$ = new Subprogram($1, $3); }
;

function_hdg : FUNCTION ID LPAREN variable_list RPAREN COLON TYPE
              { $$ = new SubprogramType($2, $4, $7);
                currentScope->insert($2, new SubprogramAttribute($$));
                currentScope = new SubprogramScope(currentScope, $$); }
;

stmt_list : stmt
           { $$ = new StatementList($1); }
| stmt_list stmt
           { $$ = $1->addLast($2); }
;

stmt : compound_stmt
      { $$ = $1; }
| assign_stmt
      { $$ = $1; }
| if_stmt
      { $$ = $1; }
| while_stmt
      { $$ = $1; }
| read_stmt
      { $$ = $1; }
| write_stmt

```

```

        { $$ = $1; }
| return_stmt
    { $$ = $1; }
;

compound_stmt : BEGIN_STMT stmt_list END_STMT
               { $$ = new CompoundStatement($2); }
;

assign_stmt   : id_addr ASSIGN expr SEMICOLON
               { $$ = new AssignmentStatement($1, $3); }
;

if_stmt       : IF expr THEN stmt ELSE stmt
               { $$ = new ConditionalStatement($2, $4, $6); }
| IF expr THEN stmt
               { $$ = new ConditionalStatement($2, $4); }
;

while_stmt    : WHILE expr DO stmt
               { $$ = new PretestLogicalLoop($2, $4, false); }
;

read_stmt     : READ id_addr SEMICOLON
               { $$ = $2->inputStatement(); }
;

write_stmt    : WRITE expr SEMICOLON
               { $$ = $2->outputStatement(); }
;

return_stmt   : RETURN expr SEMICOLON
               { $$ = new ReturnStatement($2); }
;

expr_list     : expr
               { $$ = new ExpressionList($1); }
| expr_list COMMA expr
               { $$ = $1->addLast($3); }
;

expr          : simple_expr
               { $$ = $1; }
| simple_expr RELOP simple_expr

```

```

        { $$ = $2->expression($1, $3); }
    ;

simple_expr : term
           { $$ = $1; }
    | simple_expr ADDOP term
           { $$ = $2->expression($1, $3); }
    ;

term : factor
     { $$ = $1; }
    | term MULOP factor
     { $$ = $2->expression($1, $3); }
    ;

factor : INTEGER
       { $$ = $1; }
    | REAL
       { $$ = $1; }
    | ID LPAREN expr_list RPAREN
       { $$ = new FunctionCall($1, $3); }
    | id_addr
       { $$ = $1->dereference(); }
    | LPAREN expr RPAREN
       { $$ = $2; }
    ;

id_addr : ID
        { $$ = currentScope->identifierAddress($1); }
    ;

%%

main()
{
    return yyparse();
}

```

## D A *MINPAS* Source Program

```
{ gcd.mp - calculate greatest common divisor }
```

```
program
var
  x : integer;
  y : integer;

  function gcd(a : integer; b: integer) : integer;
begin
  if b = 0 then
    return a;
  else
    return gcd(b, a mod b);
end;

begin
  read x;
  read y;
  write gcd(x, y);
end.
```

## E Generated M68030 Assembly Code

```
        version 2
text
        lalign 2
        global _gcd
_gcd:
        link.w  %a6,&-0
        mov.l   &0,-(%sp)
        mov.l   16(%a6),%d0
        cmp.l   %d0,(%sp)+
        bne     L3
        mov.l   12(%a6),%d0
        bra     L2
        bra     L4
L3:
        mov.l   16(%a6),-(%sp)
        mov.l   12(%a6),%d0
        divsl.l (%sp)+,%d1:%d0
        mov.l   %d1,%d0
        mov.l   %d0,-(%sp)
        mov.l   16(%a6),-(%sp)
        mov.l   8(%a6),-(%sp)
        jsr     _gcd
        add.l   &12,%sp
        bra     L2
L4:
L2:
        unlk   %a6
        rts
        version 2
text
        lalign 2
        global _main
_main:
        link.w  %a6,&-0
        pea.l   _x
        mov.l   %a6,-(%sp)
        jsr     _inInteger
        add.l   &8,%sp
        pea.l   _y
        mov.l   %a6,-(%sp)
        jsr     _inInteger
        add.l   &8,%sp
```

```

    mov.l  _y,-(%sp)
    mov.l  _x,-(%sp)
    mov.l  %a6,-(%sp)
    jsr    _gcd
    add.l  &12,%sp
    mov.l  %d0,-(%sp)
    mov.l  %a6,-(%sp)
    jsr    _outInteger
    add.l  &8,%sp
L1:
    unlk   %a6
    rts
data
    comm   _x,4
    comm   _y,4

```