The source program consists of a stream of tokens, so object-orientation has little to do with the code for the parser. Coming out of the parser, the source program consists of a syntax tree, with constructs or nodes implemented as objects. These objects deal with all of the following: construct a syntax-tree node, check types, and generate three-address intermediate code.

A.l The Source Language

A program in the language consists of a block with optional declarations and statements. Token *basic* represents basic types.

program -> block

block -> { decls stmts }

decls -> decls decl | e

decl type -> id ;

type -> type [num] | basic

stmts -> stmts stmt | e

Treating assignments as statements, rather than as operators within expressions, simplifies translation.

stmt -» loc = bool;

| if (bool) stmt

| if (bool) stmt else stmt

| while (bool) stmt

| do stmt while (bool);

|for (stmt ; bool ; stmt ;) stmt

| break;

| continue;

| block

loc -> loc [bool] | id

The productions for expressions handle associativity and precedence of operators. They use a nonterminal for each level of precedence and a nonterminal, factor, for parenthesized expressions, identifiers, array references, and constants.

bool -> bool || join | join

join -> join && equality | equality

equality -> equality == rel | equality ! = rel | rel

rel -> expr < expr | expr <= expr | expr >= expr |

expr > expr | expr

expr -> expr + term | expr - term | term

*term -> term * unary | term / unary | unary*

unary -> !unary | - unary | factor

factor -> (bool) | loc | ***num*** *|* ***real*** *|* ***true*** *|* ***false***

A. 2 Main

Execution begins in method main in class **Main**. Method main creates a lexical analyzer and a parser and then calls method program in the parser.

A.3 Lexical Analyzer

Class **Tag** defines constants for tokens. Three of the constants, INDEX, MINUS, and TEMP, are not lexical tokens; they will be used in syntax trees.

Class **Word** manages lexemes for reserved words, identifiers, and composite tokens like '&&'. It is also useful for managing the written form of operators in the intermediate code like unary minus; for example, the source text '-2' has the intermediate form 'minus 2'.

Class **Real** is for floating point numbers and class **Num** is for integers.

The main method in class **Lexer**, function *scan*, recognizes numbers, identifiers, and reserved words. Also in class **Lexer** have been reserved selected keywords. Objects Word.True and Word.False are defined in class **Word**. Objects for the basic types int, char, bool, and float are defined in class **Type**, a subclass of **Word**.

Function *readch()* is used to read the next input character into variable peek. The name readch is overloaded to help recognize composite tokens. For example, once input '<' is seen, the call *readch*('=') reads the next character into peek and checks whether it is '='.

Function *scan()* begins by skipping white space. It recognizes composite tokens like '<=' and numbers like '365' and '3.14', before collecting words. Finally, any remaining characters are returned as tokens.

A.4 Symbol Tables and Types

Class **Env** implements symbol tables. Whereas class **Lexer** maps strings to words, class **Env** maps word tokens to objects of class **Id**.

We define class **Type** to be a subclass of **Word** since basic type names like int are simply reserved words, to be mapped from lexemes to appropriate objects by the lexical analyzer. The objects for the basic types are Type. Int, Type. Float, Type.Char, and Type.Bool. All of them have inherited field tag set to Tag.BASIC, so the parser treats them all alike.

Functions *numeric* and *max* are useful for type conversions. Conversions are allowed between the "numeric" types Type.Char, Type.Int, and Type.Float. When an arithmetic operator is applied to two numeric types, the result is the "max" of the two types.

Arrays are the only constructed type in the source language. The call to parent constructor sets field width, which is essential for address calculations. It also sets *lexeme* and *tok* to default values that are not used.

A.5 Intermediate Code for Expressions

Here are represent the **Node** class hierarchy. **Node** has two subclasses: **Expr** for expression nodes and **Stmt** for statement nodes. This section introduces **Expr** and its subclasses. Some of the methods in **Expr** deal with booleans and jumping code; they will be discussed in Section A.6, along with the remaining subclasses of **Expr**.

Nodes in the syntax tree are implemented as objects of class **Node**. For error reporting, field *lexline* saves the source-line number of the construct at this node.

Expression constructs are implemented by subclasses of **Expr**. Class **Expr** has fields *op* and *type*, representing the operator and type, respectively, at a node.

Method *gen* returns a "term" that can fit the right side of a three-address instruction. Given expression E = E1 + E2, method *gen* returns a term x1+x2, where x1 and x2 are addresses for the values of E1 and E2, respectively. The return value **this** is appropriate if this object is an address; subclasses of **Expr** typically reimplement *gen*.

Method *reduce* computes or "reduces" an expression down to a single address; that is, it returns a constant, an identifier, or a temporary name. Given expression **E**, method *reduce* returns a temporary **t** holding the value of **E**. Again, this is an appropriate return value if this object is an address.

Methods *jumping* and *emitJumps* generate jumping code for boolean expressions.

Class **Id** inherits the default implementations of *gen* and *reduce* in class **Expr**, since an identifier is an address.

The node for an identifier of class **Id** is a leaf. The call parent constructor **Expr**(w, t) saves *w* and *t* in inherited fields *op* and *type*, respectively. Field *offset* holds the relative address of this identifier.

Class **Op** provides an implementation of reduce that is inherited by subclasses **Arith** for arithmetic operators, **Unary** for unary operators, and **Access** for array accesses. In each case, *reduce* calls *gen* to generate a term, emits an instruction to assign the term to a new temporary name, and returns the temporary.

Class **Arith** implements binary operators like + and *. Constructor Arith begins by calling parent constructor (tok, nullptr), where *tok* is a token representing the operator and *nullptr* is a placeholder for the type. The type is determined by using *type->max*, which checks whether the two operands can be coerced to a common numeric type; the code for *type->max* is in Section A.4. If they can be coerced, type is set to the result type; otherwise, a type error is reported. This simple compiler checks types, but it does not insert type conversions.

Method *gen* constructs the right side of a three-address instruction by reducing the subexpressions to addresses and applying the operator to the addresses. For example, suppose *gen* is called at the root for a+b*c. The calls to *reduce* return ***a*** as the address for subexpression ***a*** and a temporary ***t*** as the address for b*c. Meanwhile, *reduce* emits the instruction t=b*c. Method *gen* returns a new Arith node, with operator * and addresses ***a*** and ***t*** as operands.

It is worth nothing that temporary names are typed, along with all other expressions. The constructor **Temp** is therefore called with a type as a parameter.

Class **Unary** is the one-operand counterpart of class **Arith**.

A.6 Jumping Code for Boolean Expressions

Jumping code for a boolean expression ***B*** is generated by method *jumping*, which takes two labels ***t*** and ***f*** as parameters, called the true and false exits of ***B***, respectively. The code contains a jump to ***t*** if **B** evaluates to true, and a jump to ***f*** if **B** evaluates to false. By convention, the special label 0 means that control falls through **B** to the next instruction after the code for **B**.

We begin with class **Constant**. The constructor Constant takes a token *tok* and a type *t* as parameters. It constructs a leaf in the syntax tree with label *tok* and type *t*. For convenience, the constructor Constant is overloaded to create a constant object from an integer.

Method *jumping* takes two parameters, labels ***t*** and ***f***. If this constant is the static object True and ***t*** is not the special label 0, then a jump to ***t*** is generated. Otherwise, if this is the object False and ***f*** is nonzero, then a jump to ***f*** is generated.

Class **Logical** provides some common functionality for classes **Or**, **And**, and **Not**. Fields *expr1* and *expr2* correspond to the operands of a logical operator. (Although class **Not** implements an unary operator, for convenience, it is a subclass of Logical.) The constructor Logical (tok, x1, x2) builds a syntax node with operator *tok* and operands *expr1* and *expr2*. In doing so it uses function *check* to ensure that both *expr1* and *expr2* are booleans. Method *gen* will be discussed at the end of this section.

In class **Or**, method *jumping* generates jumping code for a boolean expression B = B1 || B2. For the moment, suppose that neither the true exit ***t*** nor the false exit ***f*** of B is the special label 0. Since B is true if B1 is true, the true exit of B1 must be ***t*** and the false exit corresponds to the first instruction of B2. The true and false exits of B2 are the same as those of B.

In the general case, ***t***, the true exit of B, can be the special label 0. Variable *label* ensures that the true exit of B1 is set properly to the end of the code for B. If *t* is 0, then label is set to a new label that is emitted after code generation for both B1 and B2.

The code for class **And** is similar to the code for **Or**. Class **Not** has enough in common with the other boolean operators that we make it a subclass of **Logical**, even though **Not** implements a unary operator. The parent class expects two operands, so *x2* appears twice in the call to parent. Only *expr2* is used in the methods *jumping* and *toString*. Method *jumping* simply calls *expr2-> jumping* with the true and false exits reversed.

Class **Rel** implements the operators <, <=, ==, !=, >=, and >. Function *check* checks that the two operands have the same type and that they are not arrays. For simplicity, coercions are not permitted.

Method *jumping* begins by generating code for the subexpressions *expr1* and *expr2*. It then calls method *emitjumps*. If neither *t* nor *f* is the special label 0, then *emitjumps* executes the following

```
emit("if " + test + " goto L" + std::to_string(t));    // file Expr.h
emit("goto L" + std::to_string(f));
```

At most one instruction is generated if either *t* or *f* is the special label 0 (again, from file Expr.h):

```
else if (t != 0)
        emit("if " + test + " goto L" + std::to_string(t));
else if (f != 0)
        emit("iffalse " + test + " goto L" + std::to_string(f));
else;                 // nothing since both t and f fall through
```

For another use of *emitjumps*, consider the code for class **Access**. The source language allows boolean values to be assigned to identifiers and array elements, so a boolean expression can be an array access. Class **Access** has method *gen* for generating "normal" code and method *jumping* for jumping code. Method *jumping* calls *emitjumps* after reducing this array access to a temporary. The constructor is called with a flattened array ***a***, an index ***i***, and the type ***t*** of an element in the flattened array. Type checking is done during array address calculation.

Jumping code can also be used to return a boolean value. Class **Logical**, earlier in this section, has a method *gen* that returns a temporary *temp*, whose value is determined by the flow of control through the jumping code for this expression. At the true exit of this boolean expression, temp is assigned true; at the false exit, temp is assigned false. The temporary is declared on line 45. Jumping code for this expression is generated on line 48 with the true exit being the next instruction and the false exit being a new label ***f***. The next instruction assigns true to temp (line 52), followed by a jump to a new label ***a***. The code on lines 54-55 emits label ***f*** and an instruction that assigns false to temp. The code fragment ends with label ***a***, generated on line 56. Finally, *gen* returns temp.

A.7 Intermediate Code for Statements

Each statement construct is implemented by a subclass of **Stmt**. The fields for the components of a construct are in the relevant subclass; for example, class **While** has fields for a test expression and a substatement, as we shall see.

The constructor *Stmt()* does nothing, since the work is done in the subclasses. The static object Stmt::Null represents an empty sequence of statements.

The method *gen* is called with two labels ***b*** and ***a***, where ***b*** marks the beginning of the code for this statement and ***a*** marks the first instruction after the code for this statement. Method *gen* is a placeholder for the gen methods in the subclasses. The subclasses **While**, **Do and For** save their label ***a*** in the field *after* so it can be used by any enclosed break statement to jump out of its enclosing construct, and label ***b*** in the field *begin* to mark begin of loop instructions. The object Stmt::Enclosing is used during parsing to keep track of the enclosing construct.

The constructor for class **If** builds a node for a statement if (E) S. Fields *expr* and *stmt* hold the nodes for *E* and *S*, respectively. Note that *expr* in lower-case letters names a field of class **Expr**; similarly, *stmt* names a field of class **Stmt**.

The code for an **If** object consists of jumping code for expr followed by the code for stmt. As discussed in Section A.6, the call expr->jumping(0, a) specifies that control must fall through the code for expr if expr evaluates to true, and must flow to label *a* otherwise.

The implementation of class **Else**, which handles conditionals with else parts, is analogous to that of class **If**.

The construction of a **While** object is split between the constructor *While()*, which creates a node with null children, and an initialization function *init(x, s)*, which sets child *expr* to *x* and child *stmt* to *s*. Function *gen(b,a)* for generating three-address code is in the spirit of the corresponding function *gen()* in class **If**. The difference is that label ***a*** is saved in field *after* and that the code for stmt is followed by a jump to ***b*** for the next iteration of the while loop.

Class **Do** is very similar to class **While**.

Class **For** is also splits between it`s constructor and function *init*, which sets stmt1 to *init_val*, boolen expr to *condition*, stmt2 to *step* and stmt3 to *instruction*. Class **For** is also containes **Id** object variable to verify the equality condition of initial value and step variable. In *gen*, after generation code for init_val, *label* marks the start of condition check code and *label2* marks incremention of step variable code. After generation code of instruction, performed the transition to *label2* and then, to *label*.

Class **Set** implements assignments with an identifer on the left side and an expression on the right. Most of the code in class **Set** is for constructing a node and checking types. Function *gen* emits a three-address instruction.

Class **SetElem** implements assignments to an array element.

Class **Seq** implements a sequence of statements. The tests for null statements are for avoiding labels. Note that no code is generated for the null statement, Stmt::Null, since method *gen* in class **Stmt** does nothing.

A break statement sends control out of an enclosing loop or switch statement. Class **Break** uses field *stmt* to save the enclosing statement construct (the parser ensures that Stmt::Enclosing denotes the syntax-tree node for the enclosing construct). The code for a Break object is a jump to the label *stmt->after*, which marks the instruction immediately after the code for stmt.

Class **Continue** is similar to **Break**. The difference is that in *gen* invoke *emit* with pointer to *stmt->begin* for return to begin of loop statements.

A. 8 Parser

The parser reads a stream of tokens and builds a syntax tree by calling the appropriate constructor functions from Sections A.5-A.7.

Like the simple expression translator in Section 2.5, class **Parser** has a procedure for each nonterminal. The procedures are based on a grammar formed by removing left recursion from the source-language grammar in Section A.l.

Parsing begins with a call to procedure program, which calls *block()* to parse the input stream and build the syntax tree. Lines 87-94 generate intermediate code.

Symbol-table handling is shown explicitly in procedure block. Variable *top* holds the top symbol table; variable *savedEnv* is a link to the previous symbol table.

Declarations result in symbol-table entries for identifiers. Declarations can also result in instructions to reserve storage for the identifiers at run time.

Procedure *stmt* has a switch statement with cases corresponding to the productions for nonterminal Stmt. Each case builds a node for a construct, using the constructor functions discussed in Section A.7. The nodes for *while*, *do* and *for* statements are constructed when the parser sees the opening keyword. The nodes are constructed before the statement is parsed to allow any enclosed *break*- or *continue statements* to point back to its enclosing loop. Nested loops are handled by using variable Stmt::Enclosing in class **Stmt** and savedStmt to maintain the current enclosing loop.

For convenience, the code for assignments appears in an auxiliary procedure, *assign*. The parsing of arithmetic and boolean expressions is similar. In each case, an appropriate syntax-tree node is created. Code generation for the two is different, as discussed in Sections A.5-A.6.

The rest of the code in the parser deals with "factors" in expressions. The auxiliary procedure *offset* generates code for array address calculations, as discussed in Section 6.4.3.

A.9 Creating the Front End

The front end of current program was made by using WinAPI and based on POPPAD by C.Petzold in “Programming Windows”.

All major source programs are written to the Main file. In the file PopFile.h there is a code for calling windows dialog File Open and File Save, as well as input / output functions. The PopFind file contains the search and replace logic, and the file PopFont - the logic for selecting the font. There is almost nothing in the PopPRNT0 file: print is not supported.

First consider the file Main. In WndProc, when processing a message WM_CREATE, the specified file name is passed to the *PopFileRead* function, which is located in the PopFile. In the Main, two lines are supported for storing the file name: in the first one with the name *szFileName*, located in WndProc, completely specifies the drive, path and file name. In the second, the name *szTitleName* is specified only the name itself. In the program, it is used in the *DoCaption* function to display in the title of the window the file name; In addition, the second line is used in the *OKMessage* and *AskAboutSave* functions to output to the screen message box for the user.

There are several functions in the PopFile for displaying the File Open and File Save dialog windows, and implementation of input / output files. The dialog boxes are displayed using the functions *GetOpenFileName* and *GetSaveFileName*, which are in a dynamically-connected library of dialogs windows of common use (COMDLG32.DLL). Both of these functions use a structure of type OPENFILENAME, defined in the header file COMMDLG.H. The PopFile for this structure uses global variable ***ofn***. Most of the fields of the ofn structure are initialized in the *PopFileInitialize* function, which is called in the Main file when processing the WM_CREATE message in WndProc.

The structural variable ***ofn*** is convenient to make static and global, because the functions GetOpenFileName and GetSaveFileName returns some information to the structure that is needed for subsequent calls these functions. In this program only the basic features of the File Open and File Save dialog boxes are used. Defined structure fields OPENFILENAME are: lStructSize (size of the structure), hwndOwner (owner of the dialog box), lpstrFilter (about which conversation ahead), lpstrFile and nMaxFile (pointer to the buffer in which the full filename is given, taking into account the path, and the size of this buffer), lpstrFileTitle and nMaxFileTitle (the buffer and its size is only for the file name), Flags (for set options for the dialog window) and lpstrDefExt (here you specify a text string with the default extension file name if the user does not specify his own when typing the file name in the dialog box).

When the user selects the **Open** option of the File menu, the program calls the *PopFileOpenDlg* function from file PopFile, while the functions are passed a window handle, a pointer to the buffer of the full file name and pointer to the buffer only the file name. The function *PopFileOpenDlg* places into the structure OPENFILENAME respectively, the fields hwndOwner, lpstrFile and lpstrFileTitle, sets Flags to the value OFN_HIDEREADONLY |OFN_CREATEPROMPT, and then calls the *GetOpenFileName* function, which prints dialog box. In the default window of the File Open dialog, there is a flag that allows the user to designate that the file should be opened only for reading; at setting the OFN_HIDEREADONLY flag, the *GetOpenFileName* function does not display this flag.

When the user closes the dialog box, the function *GetOpenFileName* completes its work. If the file specified by the user does not exist, then the OFN_CREATEPROMPT flag causes the function *GetOpenFileName* to display a message window in which the user is asked about the need create a new file.

In the function *PopFileInitialize* from the PopFile in the variable *szFilter* filter is defined for three types of files: text files with .TXT extension, ASCII files with extension .ASC and files of all types. This value is set in the *lpstrFilter* field of the structure OPENFILENAME.

If the user changes the filter when the dialog box is active, then in the nFilterIndex field of the OPENFILENAME structure the user-made selection is reflected. Because the structure is stored as a static variable, when the next time the dialog box is called, the filter will be set in accordance with the selected file type. Function *PopFileSaveDlg* works similarly. It sets the Flags parameter to OFN_OVERWRITEPROMPT and to display the File Save dialog box calls the *GetSaveFileName* function. If the file selected by the user already exists, the OFN_OVERWRITEPROMPT flag causes the window message, asking if you want to overwrite an existing file. Other functions of the PopFile implement the input / output files using standard library functions of the C++ language.

Change font

When processing the WM_CREATE message in the program, the *PopFontInitialize* function is called from the file PopFont. This function gets the LOGFONT structure, formed on the basis of the system font, creates a font on its base and sends a message to the child edit window to install a new font WM_SETFONT. (Although the default font of the child editing window is the system font, the function *PopFontInitialize* creates for him a new additional font, because in the end all the font will be deleted, which would be unreasonable for a standard system font.)

When the program receives a WM_COMMAND message when the font option is selected, it is called function *PopFontChooseFont*. This function initializes the CHOOSEFONT structure, and then for display the font selection dialog box calls the *ChooseFont* function. If the user clicks the **OK** button, the return value of the *ChooseFont* function is TRUE. Then the program calls the function *PopFontSetFont* for installation in the child window for editing a new font. The old font is deleted.

Finally, when processing the WM_DESTROY message, the program calls the function *PopFontDeinitialize* to remove the last font created by the *PopFontSetFont* function.

Search and replace

The public dialog box library also includes two dialog windows for execution search and replace text functions. Both of these functions (*FindText* and *ReplaceText*) use a type structure FINDREPLACE. In the PopFind there are two other functions (*PopFindFindDlg* and *PopFindReplaceDlg*); In addition, it also has functions for searching and replacing text in the child edit window.

There are several remarks related to the use of search and replace functions. First, the dialog windows, which they call are non-modal, which means, in the case of active dialog windows, change the message processing loop to call the *IsDialogMessage* function. Secondly, the structure FINDREPLACE, passed to the *FindText* and *ReplaceText* functions, must be specified as a static variable; and since the dialog windows are non-modal, the functions should finish their work after that, as dialog boxes are displayed, and not after they are closed. And despite this, it is necessary to continue provide the ability to access the structure from the dialog window procedure.

Third, as long as dialog windows remain on the screen, the *FindText* and *ReplaceText* functions interact with window by means of a special message. The number of this message can be obtained by call the *RegisterWindowMessage* function with the FINDMSGSTRING parameter. This is done when processing in WndProc message WM_CREATE, and the received message number is stored in a static variable.

When processing the next message in WndProc, the message variable is compared with the value, returned by the *RegisterWindowMessage* function. The *lParam* parameter of the message is a pointer to the structure FINDREPLACE, the Flags field that indicates whether the user used the dialog box to search and replace text or it is closed. To directly implement the search and replace in program *The PopFindFindDlg* and *PopFindReplaceDlg* functions are located in the PopFind file.