Chapter 9 The Design of a Full Computer Language

Let's design a simple computer language, that is realistic enough to actually be useful. The language is based on the C language, with classes added. Because it is a bit like C++, but still only a toy language, let us call it B--.

The Lexical Analyser

Programs written in this language are free format. Line breaks, spaces and tabs have no syntactic significance. All three styles of line break (\r, \n, \r\n) are accepted.

Keywords are "int", "char", "bool", "void", "class" (structured types), "extends", "begin", "end", "if", "else", "elif", "while", "do", "for", "return", "this" (the current object), "null".

Special symbols are "(", ")", "[", "]", ";", ",", "." (member selection), "..." (methods with additional parameters), "=", "||", "&&", "<", ">", "<=", ">=", "==", "!=", "+", "-", "*", "/", "%", "!", "++", "--", "&" (address of), "^" (contents of).

Integer literals are decimal only. Character and string literals are as in Java. The keyword "null" represents a null value (for pointers).

Identifiers are represented by a letter, followed by 0 or more letters or digits.

Both "//" and "/*...*/" comments are permitted, but "/*...*/" comments can not be nested.

The code for the lexical analyser is

```
package grammar;

import java.io.*;
import java.util.*;
import java_cup.runtime.*;
import text.*;
import env.*;

%%

%public
%type         Symbol
%char

%{

     public Symbol token( int tokenType, Object value ) {
          Print.error().debugln( "Obtain token "
               + sym.terminal_name( tokenType ) + " \"" + yytext() + "\"" );
          return new Symbol( tokenType, yychar,
               yychar + yytext().length(), value );
          }

     public Symbol token( int tokenType ) {
          return token( tokenType, yytext() );
          }

     private int lineNumber = 1;

     public int lineNumber() { return lineNumber; }

%}
```

```
%init{
     yybegin( NORMALSTATE );
%init}

intconst       =    [0-9]+
octDigit       =    [0-7]
hexDigit       =    [0-9a-fA-F]
escchar        =    \\([ntbrfva\\\'\"\?]|{octDigit}+|[xX]{hexDigit}+)
schar          =    [^\'\"\\\r\n]|{escchar}
charconst      =    \'{schar}\'
stringconst    =    \"{schar}*\"
ident          =    [A-Za-z][A-Za-z0-9]*
space          =    [\ \t]
newline        =    \r|\n|\r\n
%state NORMALSTATE COMMENT1STATE COMMENT2STATE LEXERRORSTATE

%%
<NORMALSTATE> {
     {newline} { lineNumber++; }
     {space}   { }

     if        { return token( sym.IF ); }
     then      { return token( sym.THEN ); }
     else      { return token( sym.ELSE ); }
     elif      { return token( sym.ELIF ); }
     begin     { return token( sym.BEGIN ); }
     end       { return token( sym.END ); }
     while     { return token( sym.WHILE ); }
     do        { return token( sym.DO ); }
     for       { return token( sym.FOR ); }
     return    { return token( sym.RETURN ); }
     null      { return token( sym.NULL ); }
     int       { return token( sym.INT ); }
     char      { return token( sym.CHAR ); }
     bool      { return token( sym.BOOL ); }
     void      { return token( sym.VOID ); }
     class     { return token( sym.CLASS ); }
     extends   { return token( sym.EXTENDS ); }
     this      { return token( sym.THIS ); }

     ";"       { return token( sym.SEMICOLON ); }
     ","       { return token( sym.COMMA ); }
     "..."     { return token( sym.ETC ); }
     "="       { return token( sym.ASSIGN ); }

     "||"      { return token( sym.OR ); }
     "&&"      { return token( sym.AND ); }
     "!"       { return token( sym.NOT ); }
     "<"       { return token( sym.LT ); }
     ">"       { return token( sym.GT ); }
     "<="      { return token( sym.LE ); }
     ">="      { return token( sym.GE ); }
     "=="      { return token( sym.EQ ); }
     "!="      { return token( sym.NE ); }
     "+"       { return token( sym.PLUS ); }
     "-"       { return token( sym.MINUS ); }
     "*"       { return token( sym.TIMES ); }
     "/"       { return token( sym.DIVIDE ); }
     "%"       { return token( sym.MOD ); }
     "++"      { return token( sym.INC ); }
     "--"      { return token( sym.DEC ); }
```

```
     "&"       { return token( sym.AMPERSAND ); }
     "^"       { return token( sym.PTR ); }
     "("       { return token( sym.LEFT ); }
     ")"       { return token( sym.RIGHT ); }
     "["       { return token( sym.LEFTSQ ); }
     "]"       { return token( sym.RIGHTSQ ); }
     "."       { return token( sym.DOT ); }

     {intconst}    { return token( sym.INTVALUE ); }
     {charconst}   { return token( sym.CHARVALUE ); }
     {stringconst} { return token( sym.STRINGVALUE ); }
     {ident}       { return token( sym.IDENT ); }
     "//"          { yybegin( COMMENT1STATE ); }
     "/*"          { yybegin( COMMENT2STATE ); }
     .             {
                   yybegin( LEXERRORSTATE );
                   return token( sym.error );
                   }
     }

<LEXERRORSTATE> {
     ";"           {
                   yybegin( NORMALSTATE );
                   return token( sym.SEMICOLON );
                   }
     {newline}     { lineNumber++; }
     .             { }
     }

<COMMENT1STATE> {
     {newline}     { lineNumber++; yybegin( NORMALSTATE ); }
     .             { }
     }

<COMMENT2STATE> {
     "*/"          { yybegin( NORMALSTATE ); }
     {newline}     { lineNumber++; }
     .             { }
     }

<<EOF>>            { return token( sym.EOF ); }
```

The Grammar

```
...

parser code
     {:
     private Yylex lexer;
     private File file;

     public parser( File file ) {
          this();
          this.file = file;
          try {
               lexer = new Yylex( new FileReader( file ) );
               }
          catch ( IOException exception ) {
               throw new Error( "Unable to open file \"" + file + "\"" );
               }
          }
...
```

```
     :};

scan with
     {:
     return  lexer.yylex();
     :};

terminal
     IF, THEN, ELSE, ELIF, LEFT, RIGHT, SEMICOLON,
     COMMA, PLUS, MINUS, TIMES, DIVIDE, MOD, ASSIGN,
     OR, AND, NOT, LT, GT, LE, GE, EQ, NE,
     INC, DEC, AMPERSAND, PTR, LEFTSQ,
     RIGHTSQ, BEGIN, END, WHILE, DO, FOR,
     RETURN, DOT, NULL, INT, CHAR,  BOOL,
     VOID, CLASS, ETC, THIS, EXTENDS;

terminal String INTVALUE;
terminal String CHARVALUE;
terminal String STRINGVALUE;
terminal String IDENT;

nonterminal TypeNode
     ActualType, FormalType, ClassType, ArrayType, PtrType, BasicType;
nonterminal DeclStmtListNode
     MethodDeclBody, GlobalDeclStmtList, LocalDeclStmtList,
     MemberDeclList, FormalParamDeclList, StmtList;
 nonterminal DeclStmtNode
     GlobalDeclStmt, LocalDeclStmt, MemberDecl, ClassTypeDecl, MethodDecl,
     VarDecl, InitVarDecl, ClassInstanceDecl, ArrayInstanceDecl, Stmt;
non terminal String
     ExtendsOpt;
nonterminal DeclaratorListNode
     IdentList, LocalVarList;
nonterminal DeclaratorNode
     LocalVar;
nonterminal ElseOptNode
     ElseOpt;
nonterminal ExprListNode
     ExprListOpt, ExprList;
nonterminal ExprNode
     Expr, AssignExpr, OrExpr, AndExpr, RelExpr,
     AddExpr, MulExpr, PrefixExpr, PostfixExpr, Primary, LiteralValue;
nonterminal MethodNameNode
     MethodName;
nonterminal VariableNode
     Variable;

start with GlobalDeclStmtList;
```

Global Declarations and Statements

A program is a sequence of global declarations and statements to be executed. There is no notion of main method to invoke.

```
GlobalDeclStmtList::=
          GlobalDeclStmtList:globalDeclList GlobalDeclStmt:globalDecl
          {:
          globalDeclList.addElement( globalDecl );
          RESULT = globalDeclList;
          :}
     |
          /* Empty */
```

```
          {:
          RESULT = new DeclStmtListNode();
          :}
     ;
```

At the global level it is possible to declare class types, methods, and variables.

Declarations and statements can be intermingled, although variables should be declared before they are used (however the compiler/interpreter might not detect failure to do so).

Mutually recursive methods are permitted, which means the compiler must be multi-pass.

Class types can only be declared at the global level.

```
GlobalDeclStmt::=
          ClassTypeDecl:decl
          {:
          RESULT = decl;
          :}
     |
          MethodDecl:decl
          {:
          RESULT = decl;
          :}
     |
          LocalDeclStmt:stmt
          {:
          RESULT = stmt;
          :}
     ;
```

Types and Values

```
ActualType::=
          BasicType:type
          {:
          RESULT = type;
          :}
     |
          PtrType:type
          {:
          RESULT = type;
          :}
     |
          ClassType:type
          {:
          RESULT = type;
          :}
     |
          ArrayType:type
          {:
          RESULT = type;
          :}
     ;

FormalType::=
          BasicType:type
          {:
          RESULT = type;
          :}
     |
          PtrType:type
          {:
          RESULT = type;
```

```
          :}
     ;

ClassType::=
          IDENT:ident
          {:
          RESULT = new TypeIdentNode( ident );
          :}
     ;

ArrayType::=
          LEFTSQ INTVALUE:size RIGHTSQ ActualType:type
          {:
          RESULT = new ArrayTypeNode( Integer.parseInt( size ), type );
          :}
     ;

PtrType::=
          PTR ActualType:type
          {:
          RESULT = new PtrTypeNode( type );
          :}
     ;

BasicType::=
          INT
          {:
          RESULT = new IntTypeNode();
          :}
     |
          CHAR
          {:
          RESULT = new CharTypeNode();
          :}
     |
          BOOL
          {:
          RESULT = new BoolTypeNode();
          :}
     ;
```

The language has basic types int and char, that are effectively just integers. There is no need for an explicit conversion between these types. However, a char takes up only one byte of memory, while an int takes up a quadword. The bool type is independent of int and char, and cannot be cast to an int. The language also has the void (missing value), and null (zero address) type. String literals exist, but are pointers to null terminated sequences of chars.

The language has pointer types. A pointer can point to a variable of any type (including an array or class instance). Pointers are useful for creating complex data structures, and provide the equivalent of var parameters.

The language has array types, including arrays of arrays, etc. The array size must be a constant, so the compiler knows how much space to allocate. The language also has class types.

Like in C, it is possible to declare a variable of array or class type. The variable actually is the array or class instance, not a pointer to an array or class instance. There is no need for the equivalent of “new” to allocate space. The declaration itself allocates space for the array or class instance. (If you want to dynamically allocate class instances, you have to create an array of instances, and do your own allocation from this array.)

However, all uses of an array variable are converted into a pointer to the 0th element of the array, and all uses of a class instance variable are converted into a pointer to the class instance. It is not possible to copy array or class instances, except by explicitly copying each component.

Class type declarations

```
ClassTypeDecl::=
          CLASS IDENT:ident ExtendsOpt:extendsIdent
          BEGIN
          MemberDeclList:memberDecls
          END
          {:
          RESULT = new ClassDeclNode( ident, extendsIdent, memberDecls );
          :}
     ;

ExtendsOpt::=
          EXTENDS IDENT:superClass
          {:
          RESULT = superClass;
          :}
     |
          /* Empty */
          {:
          RESULT = null;
          :}
     ;

MemberDeclList::=
          MemberDeclList:declList MemberDecl:decl
          {:
          declList.addElement( decl );
          RESULT = declList;
          :}
     |
          /* Empty */
          {:
          RESULT = new DeclStmtListNode();
          :}
     ;

MemberDecl::=
          MethodDecl:decl
          {:
          RESULT = decl;
          :}
     |
          LocalDeclStmt:declStmt
          {:
          RESULT = declStmt;
          :}
     ;
```

A class type can extend another class type. The body of the class is composed of declarations of instance fields and methods, and statements for performing initialisation. However, there is no notion of static fields or methods. There is no notion of constructors. Fields can include array or class instances.

Method Declarations

```
MethodDecl::=
          FormalType:returnType IDENT:ident
          LEFT FormalParamDeclList:formalParams RIGHT
```

```
          MethodDeclBody:methodDeclBody
          {:
          RESULT =
               new MethodDeclNode( returnType, ident,
                    formalParams, methodDeclBody );
          :}
     |
          VOID IDENT:ident
          LEFT FormalParamDeclList:formalParams RIGHT
          MethodDeclBody:methodDeclBody
          {:
          RESULT =
               new MethodDeclNode( new VoidTypeNode(), ident,
                    formalParams, methodDeclBody );
          :}
     ;

MethodDeclBody::=
          BEGIN LocalDeclStmtList:localDeclStmtList END
          {:
          RESULT = localDeclStmtList;
          :}

     |
          SEMICOLON
          {:
          RESULT = null;
          :}
     ;

 FormalParamDeclList::=
          VarDecl:decl FormalParamDeclList:declList
          {:
          declList.prependElement( decl );
          RESULT = declList;
          :}
     |
          ETC
          {:
          RESULT = new DeclStmtListNode( new EtcDeclNode() );
          :}
     |
          /* Empty */
          {:
          RESULT = new DeclStmtListNode();
          :}
     ;

LocalDeclStmtList::=
          LocalDeclStmtList:declStmtList LocalDeclStmt:declStmt
          {:
          declStmtList.addElement( declStmt );
          RESULT = declStmtList;
          :}
     |
          /* Empty */
          {:
          RESULT = new DeclStmtListNode();
          :}
     ;
```

```
LocalDeclStmt::=
          InitVarDecl:decl
          {:
          RESULT = decl;
          :}
     |
          ClassInstanceDecl:decl
          {:
          RESULT = decl;
          :}
     |
          ArrayInstanceDecl:decl
          {:
          RESULT = decl;
          :}
     |
          Stmt:stmt
          {:
          RESULT = stmt;
          :}
     ;
```

Method declarations without a body correspond to library methods, declared outside of the language, or abstract methods in a class.

Like in C, there is only one parameter passing mechanism - pass by value. However, since the value can be the address of a variable, we can effectively have var parameters.

The expression corresponding to the actual parameter is evaluated, and used to initialise the formal parameter.

```
int fact( int n; )
     begin
          if n == 0 then
               return 1;
          else
               return n * fact( n - 1 );
          end
     end
int i;
for i = 0; i < 10; i = i + 1 do
     printf( "%d factorial = %d\n", i, fact( i ) );
end
```

A var parameter represents the address of a variable passed as an actual parameter. The formal parameter points to the address of the actual parameter. An extra level of dereferencing is needed to access the value of the variable.

```
void inc( ^int a; )
     begin
          a^++;
     end
int i;
printf( "Checking var parameters\n" );
for i = 0; i < 10; inc( &i ) do
     printf( "%d\n", i );
end
```

Variable Declarations

Simple variable declarations, without initialisation, can be used to declare formal parameters. A formal type can only be a basic type or pointer type, not an array or class type.

```
VarDecl::=
```

```
          FormalType:type IdentList:identList SEMICOLON
          {:
          RESULT = new VarDeclNode( type, identList );
          :}
     ;
```

Formal parameter declarations do not specify an initial value. (The initial value comes from the invocation.)

```
IdentList::=
          IdentList:identList COMMA IDENT:ident
          {:
          identList.addElement( new UninitDeclaratorNode( ident ) );
          RESULT = identList;
          :}
     |
          IDENT:ident
          {:
          RESULT = new DeclaratorListNode( new UninitDeclaratorNode( ident ) );
          :}
     ;
```

Variable declarations, possibly with initialisation, can be used to declare global and local variables, and instance fields of a class.

When a variable is declared, it can be given a default value, or explicitly initialised.

Variables of type int, char, and bool have a default value of 0, '\000', false, respectively. Variables of pointer type have a default value of null.

```
InitVarDecl::=
          FormalType:type LocalVarList:localVarList SEMICOLON
          {:
          RESULT = new VarDeclNode( type, localVarList );
          :}
     ;

LocalVarList::=
          LocalVarList:localVarList COMMA LocalVar:localVar
          {:
          localVarList.addElement( localVar );
          RESULT = localVarList;
          :}
     |
          LocalVar:localVar
          {:
          DeclaratorListNode localVarList = new DeclaratorListNode( localVar );
          RESULT = localVarList;
          :}
     ;

LocalVar::=
          IDENT:ident
          {:
          RESULT = new UninitDeclaratorNode( ident );
          :}
     |
          IDENT:ident ASSIGN Expr:expr
          {:
          RESULT = new InitDeclaratorNode( ident, expr );
          :}
     ;
```

For example, we could write

```
int i = 3, j;
^char s = "hello";
```

Array and class instance declarations, that allocate space for the array can be used to declare global and local variables, and instance fields of a class. But it is not possible to specify an initial value for the array or class instance. It is also not possible to pass an array or class instance as a parameter, only a pointer to one.

```
ArrayInstanceDecl::=
          ArrayType:type IdentList:identList SEMICOLON
          {:
          RESULT = new VarDeclNode( type, identList );
          :}
     ;

ClassInstanceDecl::=
          ClassType:type IdentList:identList SEMICOLON
          {:
          RESULT = new VarDeclNode( type, identList );
          :}
     ;
```

Statements

There are all the usual kinds of statements.

Expressions with side effects can be used as statements, and the result is ignored.

There are no compound statements. Instead, control statements allow a statement list rather than just a single substatement. To avoid ambiguities in the grammar, all control statements are terminated by the keyword “end”.

Variables cannot be declared within a statement.

```
int comb( int n, r; )
     begin
          if r < 0 || r > n then
               return 0;
          elif r == 0 || r == n then
               return 1;
          else
               return comb( n - 1, r - 1 ) + comb( n - 1, r );
          end
     end
int n, r;
for n = 0; n < 10; n = n + 1 do
     for r = 0; r < 10 - n; r = r + 1 do
          printf( "    " );
     end
     for r = 0; r <= n; r = r + 1 do
          printf( "%8d", comb( n, r ) );
     end
     printf( "\n" );
end
```

The grammar for statements is:

```
StmtList::=
          StmtList:stmtList Stmt:stmt
          {:
          stmtList.addElement( stmt );
          RESULT = stmtList;
          :}
     |
```

```
          /* Empty */
          {:
          RESULT = new DeclStmtListNode();
          :}
     ;

Stmt::=
          SEMICOLON
          {:
          RESULT = new NullStmtNode();
          :}
     |
          Expr:expr SEMICOLON
          {:
          RESULT = new ExprStmtNode( expr );
          :}
     |
          IF Expr:expr THEN StmtList:stmtList ElseOpt:elseOpt END
          {:
          RESULT = new IfStmtNode( expr, stmtList, elseOpt );
          :}
     |
          WHILE Expr:expr DO StmtList:stmtList END
          {:
          RESULT = new WhileStmtNode( expr, stmtList );
          :}
     |
          DO StmtList:stmtList END WHILE Expr:expr SEMICOLON
          {:
          RESULT = new DoStmtNode( stmtList, expr );
          :}
     |
          FOR Expr:initial SEMICOLON Expr:cond SEMICOLON Expr:increment
          DO StmtList:stmtList END
          {:
          RESULT = new ForStmtNode( initial, cond, increment, stmtList );
          :}
     |
          RETURN SEMICOLON
          {:
          RESULT = new ReturnStmtNode();
          :}
     |
          RETURN Expr:expr SEMICOLON
          {:
          RESULT = new ReturnExprStmtNode( expr );
          :}
     |
          error END
          {:
          RESULT = new ErrorDeclStmtNode();
          :}
     |
          error SEMICOLON
          {:
          RESULT = new ErrorDeclStmtNode();
          :}
     ;

ElseOpt::=
          /* Empty */
```

```
          {:
          RESULT = new ElseOpt0Node();
          :}
     |
          ELSE StmtList:stmtList
          {:
          RESULT = new ElseOpt1Node( stmtList );
          :}
     |
          ELIF Expr:expr THEN StmtList:stmtList ElseOpt:elseOpt
          {:
          RESULT = new ElseOpt2Node( expr, stmtList, elseOpt );
          :}
     ;
```

Operators

There are operators “=”, “||”, “&&”, “<”, “>”, “<=”, “>=”, “==”, “!=”, “+”, “-”, “*”, “/”, “%”, “!”, “&” (address of) “++”, “--”, These operators have much the same precedence as in C, except the relational operators are nonassociative. There is no operator for string concatenation (but you can get the equivalent by using the sprintf() library method).

```
ExprListOpt::=
          ExprList:exprList
          {:
          RESULT = exprList;
          :}
     |
          /* Empty */
          {:
          RESULT = new ExprListNode();
          :}
     ;

ExprList::=
          ExprList:exprList COMMA Expr:expr
          {:
          exprList.addElement( expr );
          RESULT = exprList;
          :}
     |
          Expr:expr
          {:
          ExprListNode exprList = new ExprListNode();
          exprList.addElement( expr );
          RESULT = exprList;
          :}
     ;

Expr::=
          AssignExpr:expr
          {:
          RESULT = expr;
          :}
     ;

AssignExpr::=
          Variable:variable ASSIGN OrExpr:expr
          {:
          RESULT = new AssignNode( variable, expr );
          :}
```

```
     |
          OrExpr:expr
          {:
          RESULT = expr;
          :}
     ;

OrExpr::=
          OrExpr:expr1 OR AndExpr:expr2
          {:
          RESULT = new OrNode( expr1, expr2 );
          :}
     |
          AndExpr:expr
          {:
          RESULT = expr;
          :}
     ;

AndExpr::=
          AndExpr:expr1 AND RelExpr:expr2
          {:
          RESULT = new AndNode( expr1, expr2 );
          :}
     |
          RelExpr:expr
          {:
          RESULT = expr;
          :}
     ;

RelExpr::=
          AddExpr:expr1 LT AddExpr:expr2
          {:
          RESULT = new LessThanNode( expr1, expr2 );
          :}
     |
          AddExpr:expr1 GT AddExpr:expr2
          {:
          RESULT = new GreaterThanNode( expr1, expr2 );
          :}
     |
          AddExpr:expr1 LE AddExpr:expr2
          {:
          RESULT = new LessEqualNode( expr1, expr2 );
          :}
     |
          AddExpr:expr1 GE AddExpr:expr2
          {:
          RESULT = new GreaterEqualNode( expr1, expr2 );
          :}
     |
          AddExpr:expr1 EQ AddExpr:expr2
          {:
          RESULT = new EqualNode( expr1, expr2 );
          :}
     |
          AddExpr:expr1 NE AddExpr:expr2
          {:
          RESULT = new NotEqualNode( expr1, expr2 );
          :}
```

```
     |
          AddExpr:expr
          {:
          RESULT = expr;
          :}
     ;

AddExpr::=
          AddExpr:expr1 PLUS MulExpr:expr2
          {:
          RESULT = new PlusNode( expr1, expr2 );
          :}
     |
          AddExpr:expr1 MINUS MulExpr:expr2
          {:
          RESULT = new MinusNode( expr1, expr2 );
          :}
     |
          MulExpr:expr
          {:
          RESULT = expr;
          :}
     ;

MulExpr::=
          MulExpr:expr1 TIMES PrefixExpr:expr2
          {:
          RESULT = new TimesNode( expr1, expr2 );
          :}
     |
          MulExpr:expr1 DIVIDE PrefixExpr:expr2
          {:
          RESULT = new DivideNode( expr1, expr2 );
          :}
     |
          MulExpr:expr1 MOD PrefixExpr:expr2
          {:
          RESULT = new ModNode( expr1, expr2 );
          :}
     |
          PrefixExpr:expr
          {:
          RESULT = expr;
          :}
     ;

PrefixExpr::=
          MINUS PrefixExpr:expr
          {:
          RESULT = new NegateNode( expr );
          :}
     |
          NOT PrefixExpr:expr
          {:
          RESULT = new NotNode( expr );
          :}
     |
          LEFT FormalType:type RIGHT PrefixExpr:expr
          {:
          RESULT = new ExplicitCastNode( type, expr );
          :}
```

```
     |
          INC Variable:variable
          {:
          RESULT = new PreIncNode( variable );
          :}
     |
          DEC Variable:variable
          {:
          RESULT = new PreDecNode( variable );
          :}
     |
          AMPERSAND Variable:variable
          {:
          RESULT = new AddressNode( variable );
          :}
     |
          PostfixExpr:expr
          {:
          RESULT = expr;
          :}
     ;

PostfixExpr::=
          Variable:variable INC
          {:
          RESULT = new PostIncNode( variable );
          :}
     |
          Variable:variable DEC
          {:
          RESULT = new PostDecNode( variable );
          :}
     |
          Primary:expr
          {:
          RESULT = expr;
          :}
     ;
```

Primaries

Primaries can be parenthesised expressions, literals, “this”, variables, and method invocations.

```
Primary::=
          LEFT Expr:expr RIGHT
          {:
          RESULT = expr;
          :}
     |
          LiteralValue:value
          {:
          RESULT = value;
          :}
     |
          THIS
          {:
          RESULT = new ThisNode();
          :}
     |
          Variable:variable
          {:
          RESULT = new VarExprNode( variable );
```

```
          :}
     |
          MethodName:methodName LEFT ExprListOpt:actualParams RIGHT
          {:
          RESULT = new InvocationNode( methodName, actualParams );
          :}
     ;
```

Literal values

Literals can be integer, character or string literals, or “null”.

```
LiteralValue::=
          INTVALUE:value
          {:
          RESULT = new IntValueNode( new Integer( value ) );
          :}
     |
          CHARVALUE:value
          {:
          RESULT = new CharValueNode(
               Convert.parseChar( value.substring( 1, value.length() - 1 ) ) );
          :}
     |
          STRINGVALUE:value
          {:
          RESULT = new StringValueNode(
               Convert.parseString( value.substring( 1, value.length() - 1 ) ) );
          :}
     |
          NULL
          {:
          RESULT = new NullValueNode();
          :}
     ;
```

Variables

Variables can be simple identifiers, indexed variables, field selections, or indirection of a pointer.

```
Variable::=
          IDENT:ident
          {:
          RESULT = new IdentVariableNode( ident );
          :}
     |
          Primary:array LEFTSQ Expr:index RIGHTSQ
          {:
          RESULT = new IndexVariableNode( array, index );
          :}
     |
          Primary:object DOT IDENT:member
          {:
          RESULT = new MemberVariableNode( object, member );
          :}
     |
          Primary:object PTR
          {:
          RESULT = new IndirectVariableNode( object );
          :}
     ;
```

Method names

A method name can be a simple identifier, or an instance method of a class instance.

```
MethodName::=
          IDENT:ident
          {:
          RESULT = new IdentMethodNameNode( ident );
          :}
     |
          Primary:object DOT IDENT:member
          {:
          RESULT = new MemberMethodNameNode( object, member );
          :}
     ;
```

The Library

The language has a standard library (globalLibrary.in), which declares the Boolean constants true and false, and methods exit (terminate execution), print (print a simple string), vsprintf, sprintf (formatted printing to a char array), and printf (formatted printing to standard output). The methods work much the same as in C.

```
bool true = 0 == 0;
bool false = 0 != 0;

void exit( int value; );
void print( ^char s; );
void vsprintf( ^char dest, s; ^^char args; );

void sprintf( ^char dest, s; ... )
     begin
          vsprintf( dest, s, &s + 1 );
     end

void printf( ^char s; ... )
     begin
          [ 200 ] char dest;
          vsprintf( dest, s, &s + 1 );
          print( dest );
     end
```

The B-- Directory Structure

At the top level, we have the files

- Documentation — Documentation for the compiler/interpreter.
- Source — Source code for the compiler/interpreter.
- Classes — The directory in which the compiled class files for the compiler/interpreter are stored.
- run.jar — A jar file containing all the class files.
- Programs — A directory containing all the B-- sample programs, and the files generated by compiling, interpreting and executing them.
- Bash scripts to compile the compiler/interpreter.

  createcompiler.bash compiles the compiler/interpreter, by invoking createlexer.bash, createparser.bash, createclass.bash, createjar.bash.

createlexer.bash runs JFlex, to generate the Java code for the lexical analyser.

createparser.bash runs CUP, to generate the Java code for the parser.

createclass.bash runs javac, to compile the Java source for the compiler/interpreter, and generate class files.

createjar.bash runs jar, to build the jar file run.jar.

- jflex.error, parser.cup.error, javac.error

  Error messages created when compiling the compiler/interpreter.

- Bash scripts to compile/interpret B-- programs.

This compiler/interpreter has been written so that you can specify how much of it is run. It is a combined interpreter and compiler.

At the bottom level, the compiler/interpreter can be run from Java:

```
java -classpath "run.jar$CPSEP$CCUPJAR" Main \
     -dir  "$DIR" -debug -reprint -check -interp -compile
```

The various options and flags are:

-dir DirName

Specify the directory containing the program to be interpreted/compiled. e.g., Programs/array.

-debug Turn on debugging.

The method Print.error().debugln(...); can be used to print diagnostics.

The method Print.error().debugStackTrace(...); can be used to print a stack trace.

Normally these have no effect.

-reprint Reprint the program.

-check Type check the program.

-interp Interpret the program.

-compile Compile the program.

At a higher level, the program is run via a shell script.

run.bash

Execute the compiler/interpreter. It takes a directory and flags as parameters.

To interpret/compile a program, you actually use a command such as

runPrint.bash, runCheck.bash, runInterp.bash, runDebugInterp.bash, runCompile.bash, runBoth.bash

Run a specific program, with the appropriate options. The directory containing the program is passed as an argument. e.g., Programs/array.

runSim.bash

Run the alpha simulator on the generated assembly language in batch mode.You can also run them in GUI mode, in the normal way. The directory containing the program is passed as an argument. e.g., Programs/array.

Note that run.bash and runSim.bash have a resource limit specified by the ulimit command. This does not work on Cygwin, and should be commented out to avoid a warning message.

runall.bash, runallSim.bash, runallBoth.bash, runallCompile.bash, runallInterp.bash, runallPrint.bash

Run all programs in a specified directory, with the specified options. The directory containing the program directories is passed as an argument. e.g., Programs.

The source code is split into a number of files and directories

- Main.java — The main program.
- text — Support code for printing text suitable indented, etc.
- grammar — The lexical analyser and parser definitions.
- env — Classes for managing the compile-time environments and declarations.
- type — Classes to represent types.
- runEnv — Classes to represent run-time values and run-time environments in the interpreter.
- node — Classes for each node in the abstract syntax tree.
- builtin — A class to implement the builtin methods for the interpreter, and library code for the simulator.
- code — Classes for generating code, representing operands for instructions, usage of registers, etc.

Within the node directory, we have the following files and directories

- Node.java — All nodes of the tree extend this abstract class.
- ProgramNode.java — A class to represent the root node of the abstract syntax tree.
- declNode — Classes for nodes corresponding to declarations.
- typeNode — Classes for nodes corresponding to types.
- exprNode — Classes for nodes corresponding to expressions.
- stmtNode — Classes for nodes corresponding to statements.

Division into passes

The compiler/interpreter is multi-pass. The passes are:

- parse — Lexical analysis and parsing.
- toString — Conversion of the program back into text. This is purely to check that it has been parsed correctly.

  The textual representation of a node is a synthesized attribute.
- genEnv — Generation of the compile-time environments. Generation and insertion of declarations in the appropriate compile-time environments.

  The compile-time environment is primarily an inherited attribute, passed down the tree as a parameter to genEnv(). However, the portion of the compile-time environment corresponding to the current block is technically

really a threaded attribute, because declarations are added to it as it is threaded through the declarations.

- getType, setType

  Lookup of type identifiers, building of type information for type constructs, and setting of type information for declarations.

  The type of a type construct is a synthesized attribute returned as the result of getType(). It is then used as an inherited attribute by setType(), passed down the tree as a parameter to setType(), and used to set the type information for each declaration.

- checkType Determination and checking of the types of expressions.

  The type of the expression is a synthesized attribute returned as the result of checkType().

- genOffset Determination of the offset of variables and methods in activation records, field and method tables, etc.

- eval, evalAddr Execution of the program by performing a treewalk.

  The run-time environment is primarily an inherited attribute, passed down the tree as a parameter to eval(). However, assignment statements can modify the run-time environment, which means it is technically really a threaded attribute.

- genCode, evalCode, boolCode, evalAddrCode

  Generation of assembly language, so the program can be run on the Alpha simulator.

  Rather than returning the assembly language as text, it is directly output to a file.

The main program

Essentially each node of the abstract syntax tree has a method corresponding to each pass (apart from the parsing phase), that recursively invokes the corresponding method for its children.

```
public class Main {

    public static void main( String[] argv ) {
        try {
            boolean reprint = false;
            boolean check = false;
            boolean interp = false;
            boolean compile = false;
            String dirName = null;
            for ( int i = 0; i < argv.length; i++ ) {
                if ( argv[ i ].equals( "-debug" ) ) {
                    Print.DEBUG = true;
                    }
                else if ( argv[ i ].equals( "-dir" ) ) {
                    i++;
                    if ( i >= argv.length )
                        throw new Error( "Missing directory name" );
                    dirName = argv[ i ];
                    }
                else if ( argv[ i ].equals( "-reprint" ) ) {
                    reprint = true;
```

```
                    }
                else if ( argv[ i ].equals( "-check" ) ) {
                    check = true;
                    }
                else if ( argv[ i ].equals( "-interp" ) ) {
                    interp = true;
                    }
                else if ( argv[ i ].equals( "-compile" ) ) {
                    compile = true;
                    }
                else {
                    throw new Error(
                        "Usage Error: java Main -dir directory [-debug] [-
reprint] [-check] [ -interp ] [ -compile ] on argument " + i );
                    }
                }

            if ( dirName == null )
                throw new Error( "Directory not specified" );

            if ( interp || compile )
                check = true;

            if ( check )
                reprint = true;

            Print.setError( new File( dirName, "program.err" ) );

            parser libraryParser = new parser(
                new File( "LIBRARY", "globalLibrary.in" ) );
            DeclStmtListNode globalLibrary =
                ( DeclStmtListNode ) ( libraryParser.parse().value );

            File inputFile = new File( dirName, "program.in" );

            parser programParser = new parser( inputFile );
            DeclStmtListNode declStmtList =
                ( DeclStmtListNode ) ( programParser.parse().value );
            if ( globalLibrary == null )
                Print.error().println( "globalLibrary == null" );
            if ( declStmtList == null )
                Print.error().println( "declStmtList == null" );
            ProgramNode program = new ProgramNode(
                globalLibrary, declStmtList );

            if ( reprint ) {
                Print.error().println( "Reprinting ... " );
                Print.setReprint( new File( dirName, "program.print" ) );
                Print.reprint().println( program );
                }

            if ( check ) {
                Print.error().println( "Generate Environments ... " );
                program.genEnv();
                Print.error().println( "Set Types for declarations ... " );
                program.setType();
                Print.error().println( "Check Types ... " );
                program.checkType();
                Print.error().println( "Generate Offsets ... " );
                program.genOffset();
                }
```

```
            if ( interp ) {
                 Print.error().println( "Evaluate ... " );
                 Print.setInterp( new File( dirName, "interp.out" ) );
                 try {
                      program.eval();
                      }
                 catch ( ExitException exception ) {
                      Print.error().println(
                           "Exit with status " + exception.returnValue() );
                      }
                 catch ( Error exception ) {
                      Print.error().println(
                           "User Error " + exception.getMessage() );
                      Print.error().printStackTrace( exception );
                      }
                 }

            if ( compile ) {
                 Print.error().println( "Generate Code ... " );
                 Print.setCompile( new File( dirName, "usercode.user.s" ) );
                 program.genCode();
                 }
            }
       catch ( Throwable exception ) {
            exception.printStackTrace();
            Print.error().println( "Exception in Main " + exception );
            Print.error().printStackTrace( exception );
            System.exit( -1 );
            }
       }
  }
```