Overall Compiler Structure - Stage 1

Stage1 adds the assignment statement to Pascallite, enabling you to write very simple but not totally trivial programs. Because arithmetic operations require the use of registers on the computer, you will have to be concerned with the assignment of registers to operands.

Another new concept introduced here is that of *temporary variables*, which are generated by the compiler, not by explicit program direction. The target machine's architecture requires that complex arithmetic expressions be broken into simpler subexpressions, each of which is evaluated separately in turn. The result of one subexpression is an operand of another subexpression. Each result is given a name by the compiler for each reference and is called a *temporary*; for example, the expression:

```
A := B + C * 2;
```

is Pascallite would instead be compiled as if it were:

```
TEMP1 := C * 2;
TEMP1 := B + TEMP1;
A := TEMP1;
```

in which one statement is broken into three.

Pascallite Language Definition - Stage 1

1.	Six new keywords are added: **mod, div, and, or, read, write** The first two are *integer* arithmetic operators, while the next two are *boolean* operators. The last two are used for input and output.
2.	Nine new tokens have been added to Pascallite: **`:= * ( ) <> < <= >= >`** The first of these is a symbol token. This is the first such token which is not also a number or identifier. The scanner will have to be adjusted so that when it encounters ':' it checks whether the next character is '='. This token is the assignment operator. The second token indicates multiplication of *integer* operands. The next two are parentheses for grouping expressions. The last five are relational operators. The operator <> is used for testing inequality and the operator = is used for testing equality.

Pascallite Grammar Stage 1

Revised Productions:

1.	`BEGIN_END_STMT`	→	`'begin' EXEC_STMTS 'end' '.'`	`{'begin'}`

New Productions:

2.	`EXEC_STMTS`	→	`EXEC_STMT EXEC_STMTS`	`{NON_KEY_ID,'read','write'}`
		→	ε	`{'end'}`

<table>
<tr><td>3.</td><td>EXEC_STMT</td><td>→</td><td>ASSIGN_STMT</td><td>{NON_KEY_ID}</td></tr>
<tr><td></td><td></td><td>→</td><td>READ_STMT</td><td>{'read'}</td></tr>
<tr><td></td><td></td><td>→</td><td>WRITE_STMT</td><td>{'write'}</td></tr>
<tr><td>4.</td><td>ASSIGN_STMT</td><td>→</td><td>NON_KEY_ID ':=' EXPRESS ';'</td><td>{NON_KEY_ID}</td></tr>
<tr><td>5.</td><td>READ_STMT</td><td>→</td><td>'read' READ_LIST ';'</td><td>{'read'}</td></tr>
<tr><td>6.</td><td>READ_LIST</td><td>→</td><td>'(' IDS ')'</td><td>{'('}</td></tr>
<tr><td>7.</td><td>WRITE_STMT</td><td>→</td><td>'write' WRITE_LIST ';'</td><td>{'write'}</td></tr>
<tr><td>8.</td><td>WRITE_LIST</td><td>→</td><td>'(' IDS ')'</td><td>{'('}</td></tr>
<tr><td>9.</td><td>EXPRESS</td><td>→</td><td>TERM EXPRESSES</td><td>{'not','true','false','(','+','-',
INTEGER,NON_KEY_ID}</td></tr>
<tr><td>10.</td><td>EXPRESSES</td><td>→</td><td>REL_OP TERM EXPRESSES</td><td>{'<>','=','<=','>=','<','>'}</td></tr>
<tr><td></td><td></td><td>→</td><td>ε</td><td>{')',';'}</td></tr>
<tr><td>11.</td><td>TERM</td><td>→</td><td>FACTOR TERMS</td><td>{'not','true','false','(','+','-',
INTEGER,NON_KEY_ID}</td></tr>
<tr><td>12.</td><td>TERMS</td><td>→</td><td>ADD_LEVEL_OP FACTOR TERMS</td><td>{'-','+','or'}</td></tr>
<tr><td></td><td></td><td>→</td><td>ε</td><td>{'<>','=','<=','>=','<','>',')',';'}</td></tr>
<tr><td>13.</td><td>FACTOR</td><td>→</td><td>PART FACTORS</td><td>{'not','true','false','(','+','-',
INTEGER,NON_KEY_ID}</td></tr>
<tr><td>14.</td><td>FACTORS</td><td>→</td><td>MULT_LEV_OP PART FACTORS</td><td>{'*','div','mod','and'}</td></tr>
<tr><td></td><td></td><td>→</td><td>ε</td><td>{'<>','=','<=','>=','<','>',')',';','-','+','or'}</td></tr>
<tr><td>15.</td><td>PART</td><td>→</td><td>'not'
(
'(' EXPRESS ')'
| BOOLEAN
| NON_KEY_ID
)</td><td>{'not'}

{'('}
{BOOLEAN}
{NON_KEY_ID}
</td></tr>
<tr><td></td><td></td><td>→</td><td>'+'
(
'(' EXPRESS ')'
| INTEGER
| NON_KEY_ID
)</td><td>{'+'}

{'('}
{INTEGER}
{NON_KEY_ID}
</td></tr>
<tr><td></td><td></td><td>→</td><td>'-'
(
'(' EXPRESS ')'
| INTEGER
| NON_KEY_ID
)</td><td>{'-'}

{'('}
{INTEGER}
{NON_KEY_ID}
</td></tr>
<tr><td></td><td></td><td>→</td><td>'(' EXPRESS ')'</td><td>{'('}</td></tr>
</table>

		→	INTEGER	{INTEGER}
		→	BOOLEAN	{'true','false'}
		→	NON_KEY_ID	{NON_KEY_ID}
16.	REL_OP	→	'='	{'='}
		→	'<>'	{'<>'}
		→	'<='	{'<='}
		→	'>='	{'>='}
		→	'<'	{'<'}
		→	'>'	{'>'}
17.	ADD_LEVEL_OP	→	'+'	{'+'}
		→	'-'	{'-'}
		→	'or'	{'or'}
18.	MULT_LEVEL_OP	→	'*'	{'*'}
		→	'div'	{'div'}
		→	'mod'	{'mod'}
		→	'and	{'and'}

The data type of the result of an operation depends upon the operation itself. The result of an arithmetic operation is always *integer*; that of a *boolean* operation is *boolean*; and that of a relational operation is also *boolean*. In addition, there are several context-sensitive constraints on Pascallite programs based on the data types and attributes of operands:

1. Every identifier which is referenced in an assignment statement must have been previously declared as a variable or constant name.
2. The identifier to the left of the assignment operator ':=' must be a variable name and declared to be the same type as the expression to the right of the ':=' operator.
3. The operands of the binary operators '+' '-' 'div' 'mod' '*' must be *integer* valued expressions.
4. The operands of the unary operators '+' '-' must be *integer* valued expressions.
5. The operands of the logical operators 'and' 'or' 'not' must be *boolean* valued expressions.
6. The operands of the relational operators '<' '>' '<=' '>=' must be integer valued but for the two relational operators '=' '<>' the operands may either both be *integer* valued or both be *boolean* valued (but not mixed).

One feature of Pascallite is somewhat unusual--the precedence relationship between operators. For the most part, Pascallite follows the precedence relationships common to most programming languages. These relationships, enforced in the grammar, are shown graphically below. Note that 'and' has the same precedence as '*', 'div', and 'mod', and that 'or' has the same precedence as binary '+' and '–'. This is unusual and can lead to unexpected syntactic errors for a programmer not familiar with this fact. For example, the statement below is illegal:

```
w:= p > q and r > s;
```

where variables *p*,*q*,*r*, and *s* are type *integer* and *w* is type *boolean* because it is equivalent to the parenthesized:

```
w := p < (q and r) > s;
```

rather than the intended:

```
w := (p < q) and (r > s);
```

Operator Precedence

`'not'` `'-'`(unary) `'+'`(unary)	↓
`'*'` `'div'` `'mod'` `'and'`	Decreasing
`'+'`(binary) `'-'`(binary) `'or'`	order of
`'='` `'<'` `'>'` `'<='` `'>='` `'<>'`	Precedence
`':='`	↓

Until a variable has been assigned a value, it is illegal to refer to its value, which is *undefined*. For now, we will assume that no Pascallite program violates this constraint. Although good diagnostic compilers do detect this error, many commercial compilers do not. In such implementations, the value of a variable which is technically undefined is actually just whatever value is left in the storage from some previous usage of that memory location. Other implementations sometimes initialize variables to a special value at compile-time, in which case a variable is never undefined.

In Pascallite, the arithmetic and logical operators obey common mathematical laws:

1. The *integers* obey the commutative law over '+' and '*'.
2. The *integers* obey the associative law over '+' and '*'.
3. If *a* is an *integer* valued expression, then

   `a = -(-a)` and `a = +a`

4. The *booleans* obey the commutative law over 'and' and 'or'.
5. The *booleans* obey the associative law over 'and' and 'or'.
6. If *b* is a *boolean* valued expression, then

   `b = not not b`

This list is obviously incomplete but serves to remind the reader of the flexibility possible in forming equivalent logical and arithmetic expressions.

<table>
<tr><th colspan="4">Pascallite Translation Grammar Stage 1</th></tr>
<tr><td colspan="4">Revised Productions:</td></tr>
<tr><td>1.</td><td>BEGIN_END_STMT</td><td>→</td><td>'begin' EXEC_STMTS 'end' '.' code('end','.')</td></tr>
<tr><td colspan="4">New Productions:</td></tr>
<tr><td>2.</td><td>EXEC_STMTS</td><td>→
→</td><td>EXEC_STMT EXEC_STMTS
ε</td></tr>
<tr><td>3.</td><td>EXEC_STMT</td><td>→
→
→</td><td>ASSIGN_STMT
READ_STMT
WRITE_STMT</td></tr>
<tr><td>4.</td><td>ASSIGN_STMT</td><td>→</td><td>NON_KEY_ID$_x$ pushOperand(x) ':=' pushOperator(':=')
EXPRESS ';'
code(popOperator,popOperand,popOperand)</td></tr>
<tr><td>5.</td><td>READ_STMT</td><td>→</td><td>'read' READ_LIST ';'</td></tr>
<tr><td>6.</td><td>READ_LIST</td><td>→</td><td>'(' IDS$_x$ ')'
code('read',x)</td></tr>
<tr><td>7.</td><td>WRITE_STMT</td><td>→</td><td>'write' WRITE_LIST ';'</td></tr>
<tr><td>8.</td><td>WRITE_LIST</td><td>→</td><td>'(' IDS$_x$ ')'
code('write',x)</td></tr>
<tr><td>9.</td><td>EXPRESS</td><td>→</td><td>TERM EXPRESSES</td></tr>
<tr><td>10.</td><td>EXPRESSES</td><td>→

→</td><td>REL_OP$_x$ pushOperator(x) TERM
code(popOperator,popOperand,popOperand) EXPRESSES
ε</td></tr>
<tr><td>11.</td><td>TERM</td><td>→</td><td>FACTOR TERMS</td></tr>
<tr><td>12.</td><td>TERMS</td><td>→

→</td><td>ADD_LEVEL_OP$_x$ pushOperator(x) FACTOR
code(popOperator,popOperand,popOperand) TERMS
ε</td></tr>
<tr><td>13.</td><td>FACTOR</td><td>→</td><td>PART FACTORS</td></tr>
<tr><td>14.</td><td>FACTORS</td><td>→

→</td><td>MULT_LEV_OP$_x$ pushOperator(x) PART
code(popOperator,popOperand,popOperand) FACTORS
ε</td></tr>
<tr><td>15.</td><td>PART</td><td>→

→</td><td>'not' ('(' EXPRESS ')' code('not',popOperand)|
BOOLEAN$_x$ pushOperand(not x; i.e., 'true' or 'false') |
NON_KEY_ID$_x$ code('not',x))
'+' ('(' EXPRESS ')' |
(INTEGER$_x$ | NON_KEY_ID$_x$) pushOperand(x))</td></tr>
</table>

<table>
<tr><td></td><td></td><td>→</td><td>'-' ('(' EXPRESS ')' code('neg',popOperand) |
INTEGER$_x$ pushOperand('-'+ x) |
NON_KEY_ID$_x$ code('neg',x))</td></tr>
<tr><td></td><td></td><td>→</td><td>INTEGER$_x$ | BOOLEAN$_x$ | NON_KEY_ID$_x$
pushOperand(x)</td></tr>
<tr><td></td><td></td><td>→</td><td>'(' EXPRESS ')'</td></tr>
<tr><td>16.</td><td>REL_OP</td><td>→</td><td>'=' | '<>' | '<=' | '>=' | '<' | '>'</td></tr>
<tr><td>17.</td><td>ADD_LEVEL_OP</td><td>→</td><td>'+' | '-' | 'or'</td></tr>
<tr><td>18.</td><td>MULT_LEVEL_OP</td><td>→</td><td>'*' | 'div' | 'mod' | 'and'</td></tr>
</table>

There are five action routines called in the translation grammar productions:

1. `code(operator,operand1,operand2)`
2. `pushOperator(operator)`
3. `popOperator`
4. `pushOperand(operand)`
5. `popOperand`

The first routine is the code generator. The other four routines are all related to the manipulation of two auxiliary stacks which are needed to process Pascallite source programs: `operatorStk` and `operandStk`, holding a list of operators and operands, respectively.

Details were provided for stage0 to show how the translation grammar would be converted to pseudo-code. By this point, the pattern of conversion should be clear, so from here on only the translation grammar productions will be given, and the relatively mechanical conversion to pseudo-code will be left to you. The scanner modifications will be left to you as well, although you should remember to check whether the next character in the input stream is a continuation of that same token when encountering either ':', '<', or '>'.

operandStk and operatorStk

Complex expressions (whether logical, arithmetic, or relational) cannot be directly evaluated. Complex expressions must be broken into smaller subexpressions which can be directly evaluated. The proper combination of the results of evaluating these subexpressions yields the value of the original expression. The algorithm employed here by stage1 for subexpression evaluation and analysis requires two auxiliary stacks, `operatorStk` and `operandStk`. The first stack holds operators, the second holds the operands of these operators, or more correctly, it holds their names.

To clarify one point about the behavior of `code()`, when `code()` is called upon to emit target text which computes a "result" which must later be referenced (as is the case for unary and binary arithmetic, logical and relational operations), it gives a symbolic name to that result. This name is created internally by stage1 and has no relationship to the names used for identifiers in Pascallite source code. Since the result of such a call is to be referenced later in code generation, `code()` pushes the symbolic name of that result onto `operandStk`. The names stage1 uses for these results have a form similar to the internal names given to external identifiers; i.e., "`Tn`," where *n* is a non-negative integer starting at `0`. With these facts about `code()` in mind, step through the code generation process for the statement below, where all variables are *integer* valued:

```
w := (a + b)*(2 div c);
```

To translate this single assignment statement, begin the derivation with the nonterminal `ASSIGN_STMT` rather than `PROG`. The activity sequence for this derivation, with the actual values of the arguments of the action routines substituted for the variable names is:

```
pushOperand('w')                1
pushOperator(':=')              2
pushOperand('a')                3
pushOperator('+')               4
pushOperand('b')                5
code('+','b','a')               6
pushOperand('T0')               7
pushOperator('*')               8
pushOperand('2')                9
pushOperator('div')            10
pushOperand('c')               11
code('div','c','2')            12
pushOperand('T1')              13
code('*','T1','T0')            14
pushOperand('T0')              15
code(':=','T0','w')            16
```

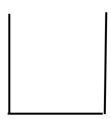
The figures below show the changes to the two stacks during the calls to these action routines. Initially, both stacks are empty. The first five action calls push three operands and two operators onto `operandStk` and `operatorStk`, respectively, shown in (a) through (f). Call (6) adds `a` to `b`, pushing the result `T0` onto `operandStk`. When call (11) is made, code to divide `2` by `c` is emitted, and the result is given the name `T1`, which is pushed onto `operandStk` as shown in figure (m). Because the second operand of a binary operation is on the top of `operandStk`, `code(operator,operand1,operand2)` actually generates code to perform

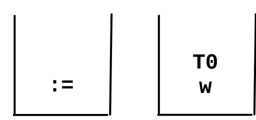
```
operand2 operator operand1
```

Figure (n) shows the result of executing call (12). "`T0`" is reused to name the new result just computed; namely,

```
T1*T0
```

This is possible because the old value of `T0`, `a + b`, cannot be referenced once it is multiplied by the value of variable `b`. After call (12) has been completed, the code which computes the value of the expression on the right hand side of '`:=`' will store its result in a location named `T0`. Call (13) emits code to store that value in variable `w`. The data sets provided address other features, such as logical and relational operations not covered in this example. Note that the correct manipulation of these two push-down stacks is essential to the proper execution of stage1.





operatorStk operandStk
(g)

The push routines include a check for stack overflow, the pop routines for stack underflow. These checks for compiler errors, not errors in the Pascallite source code are more examples of defensive programming. How the stacks are themselves implemented is left open here. The external, not the internal, form of names is pushed onto the operand stack. Since external names can be arbitrarily long in most programming languages, it is unlikely that a commercial compiler would use external name here; rather, for the sake of time and space, the shorter internal names would instead be pushed onto the stack, or, as suggested for operands in class, the index of the symbolTable entry for `name`.

pushOperator(), pushOperand(), popOperator(), popOperand()

```
void pushOperator(string name)  //push name onto operatorStk
{
  push name onto stack;
}

void pushOperand(string name)  //push name onto operandStk
               //if name is a literal, also create a symbol table entry for it
{
  if name is a literal and has no symbol table entry
    insert symbol table entry, call whichType to determine the data type of the literal
  push name onto stack;
}

string popOperator()  //pop name from operatorStk
{
  if operatorStk is not empty
    return top element removed from stack;
  else
    processError(compiler error; operator stack underflow)
}

string popOperand()  //pop name from operandStk
{
  if operandStk is not empty
    return top element removed from stack;
  else
    processError(compiler error; operand stack underflow)

}
```

code()

For stage1, `code()` will be expanded as alternatives to handle the various features of Pascallite. Pay attention to the order that the operands are popped from the stack and then passed along as parameters to the appropriate emit functions.

Pseudo code will be provided for several of the new operations added to stage1: '+', 'div', 'and', '=', ':=', 'read', and 'write'. Each operation is from a different class or illustrates some new aspect of compilation not revealed in the others.

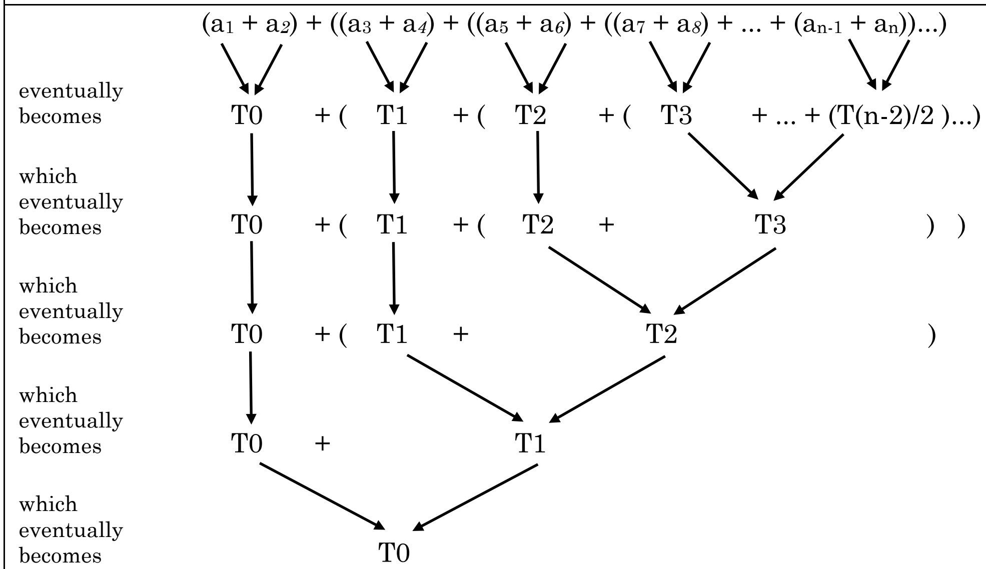
```
void code(string op, string operand1, string operand2)
{
  if (op == "program")
    emitPrologue(operand1)
  else if (op == "end")
    emitEpilogue()
  else if (op == "read")
    emit read code
  else if (op == "write")
    emit write code
  else if (op == "+") // this must be binary '+'
    emit addition code
  else if (op == "-") // this must be binary '-'
    emit subtraction code
  else if (op == "neg") // this must be unary '-'
    emit negation code;
  else if (op == "not")
    emit not code
  else if (op == "*")
    emit multiplication code
  else if (op == "div")
    emit division code
  else if (op == "mod")
    emit modulo code
  else if (op == "and")
    emit and code
  …
  else if (op == "=")
    emit equality code
  else if (op == ":=")
    emit assignment code
  else
    processError(compiler error since function code should not be called with
                 illegal arguments)
}
```

Register Allocation and Assignment - Stage1

In stage1 register assignment scheme, both registers `A` (`eax`) and `D` (`edx`) will be used, but with `D` used mainly for the remainder of mod division. This scheme will avoid needless stores into main memory and avoid needless loads into registers.

A register will be assigned to at most one operand at a time.

To evaluate $a_1 + a_2$ requires one temporary location to hold the result of the addition (where a_1 is a general quantity and not an identifier). The evaluation of $(a_1 + a_2) + (a_3 + a_4)$ requires two temporary locations, one for each parenthesized subexpression. The evaluation of $(a_1 + a_2) + ((a_3 + a_4) + (a_5 + a_6))$ requires three temporary locations. In general, the evaluation of $(a_1 + a_2) + ((a_3 + a_4) + ((a_5 + a_6) + ((a_7 + a_8) + \ldots + (a_{n-1} + a_n))\ldots)$ requires $n/2$ temporaries to store all of the intermediate results. The following diagram shows the assignment of temporaries for the last expression. Note that temporaries are reused wherever possible.



Since the number of temporary locations required depends upon the expressions being compiled, stage1 cannot allocate storage for these temporaries in advance with assurance of allocating enough; rather, it must allocate storage as needed, or allocate a fixed number and risk not having enough. In practice, only a handful of temporaries would be used, so that it is fairly safe to allocate 10 temporaries. However, it is not much harder to handle the general case, so you should do that.

Boolean and *integer* values both occupy one full-word, so temporary locations may freely be used to hold either value type. Hence, only one type of temporary storage will have to be allocated--full word temporaries. If we had stored *boolean* values in more compact storage, such as one byte or even one bit, then separate temporaries would have to be maintained for *integer* and *boolean* values, complicating the allocation and assignment scheme further. Since we shall be using one temporary location for both types of storage, we shall have to adjust the symbol table entry for the temp whenever the storage type changes. This is necessary in order to test whether the data-types of operands are correct for the operators they appear with.

Pascallite Stage 1 Header File (`/usr/local/4301/include/stage1.h`)

```
#ifndef STAGE1_H
#define STAGE1_H

#include <iostream>
#include <fstream>
#include <string>
#include <map>
#include <stack>

using namespace std;

const char END_OF_FILE = '$';      // arbitrary choice

enum storeTypes {INTEGER, BOOLEAN, PROG_NAME, UNKNOWN};
enum modes {VARIABLE, CONSTANT};
enum allocation {YES, NO};

class SymbolTableEntry
{
public:
  SymbolTableEntry(string in, storeTypes st, modes m,
                   string v, allocation a, int u)
  {
    setInternalName(in);
    setDataType(st);
    setMode(m);
    setValue(v);
    setAlloc(a);
    setUnits(u);
  }

  string getInternalName() const
  {
    return internalName;
  }

  storeTypes getDataType() const
  {
    return dataType;
  }

  modes getMode() const
  {
    return mode;
  }

  string getValue() const
  {
    return value;
  }

  allocation getAlloc() const
  {
    return alloc;
```

```
  }

  int getUnits() const
  {
    return units;
  }

  void setInternalName(string s)
  {
    internalName = s;
  }

  void setDataType(storeTypes st)
  {
    dataType = st;
  }

  void setMode(modes m)
  {
    mode = m;
  }

  void setValue(string s)
  {
    value = s;
  }

  void setAlloc(allocation a)
  {
    alloc = a;
  }

  void setUnits(int i)
  {
    units = i;
  }

private:
  string internalName;
  storeTypes dataType;
  modes mode;
  string value;
  allocation alloc;
  int units;
};

class Compiler
{
public:
  Compiler(char **argv); // constructor
  ~Compiler();           // destructor

  void createListingHeader();
  void parser();
  void createListingTrailer();

  // Methods implementing the grammar productions
```

```
  void prog();            // stage 0, production 1
  void progStmt();        // stage 0, production 2
  void consts();          // stage 0, production 3
  void vars();            // stage 0, production 4
  void beginEndStmt();    // stage 0, production 5
  void constStmts();      // stage 0, production 6
  void varStmts();        // stage 0, production 7
  string ids();           // stage 0, production 8

  void execStmts();       // stage 1, production 2
  void execStmt();        // stage 1, production 3
  void assignStmt();      // stage 1, production 4
  void readStmt();        // stage 1, production 5
  void writeStmt();       // stage 1, production 7
  void express();         // stage 1, production 9
  void expresses();       // stage 1, production 10
  void term();            // stage 1, production 11
  void terms();           // stage 1, production 12
  void factor();          // stage 1, production 13
  void factors();         // stage 1, production 14
  void part();            // stage 1, production 15

  // Helper functions for the Pascallite lexicon
  bool isKeyword(string s) const;  // determines if s is a keyword
  bool isSpecialSymbol(char c) const; // determines if c is a special symbol
  bool isNonKeyId(string s) const; // determines if s is a non_key_id
  bool isInteger(string s) const;  // determines if s is an integer
  bool isBoolean(string s) const;  // determines if s is a boolean
  bool isLiteral(string s) const;  // determines if s is a literal

  // Action routines
  void insert(string externalName, storeTypes inType, modes inMode,
              string inValue, allocation inAlloc, int inUnits);
  storeTypes whichType(string name); // tells which data type a name has
  string whichValue(string name); // tells which value a name has
  void code(string op, string operand1 = "", string operand2 = "");
  void pushOperator(string op);
  string popOperator();
  void pushOperand(string operand);
  string popOperand();

  // Emit Functions
  void emit(string label = "", string instruction = "", string operands = "",
            string comment = "");
  void emitPrologue(string progName, string = "");
  void emitEpilogue(string = "", string = "");
  void emitStorage();
  void emitReadCode(string operand, string = "");
  void emitWriteCode(string operand, string = "");
  void emitAssignCode(string operand1, string operand2);         // op2 = op1
  void emitAdditionCode(string operand1, string operand2);       // op2 +  op1
  void emitSubtractionCode(string operand1, string operand2);    // op2 -  op1
  void emitMultiplicationCode(string operand1, string operand2); // op2 *  op1
  void emitDivisionCode(string operand1, string operand2);       // op2 /  op1
  void emitModuloCode(string operand1, string operand2);         // op2 %  op1
  void emitNegationCode(string operand1, string = "");           // -op1
  void emitNotCode(string operand1, string = "");                // !op1
```

```
  void emitAndCode(string operand1, string operand2);            // op2 && op1
  void emitOrCode(string operand1, string operand2);             // op2 || op1
  void emitEqualityCode(string operand1, string operand2);       // op2 == op1
  void emitInequalityCode(string operand1, string operand2);     // op2 != op1
  void emitLessThanCode(string operand1, string operand2);       // op2 <  op1
  void emitLessThanOrEqualToCode(string operand1, string operand2); // op2 <=
op1
  void emitGreaterThanCode(string operand1, string operand2);    // op2 >  op1
  void emitGreaterThanOrEqualToCode(string operand1, string operand2); // op2
>= op1

  // Lexical routines
  char nextChar(); // returns the next character or END_OF_FILE marker
  string nextToken(); // returns the next token or END_OF_FILE marker

  // Other routines
  string genInternalName(storeTypes stype) const;
  void processError(string err);
  void freeTemp();
  string getTemp();
  string getLabel();
  bool isTemporary(string s) const; // determines if s represents a temporary

private:
  map<string, SymbolTableEntry> symbolTable;
  ifstream sourceFile;
  ofstream listingFile;
  ofstream objectFile;
  string token;              // the next token
  char ch;                   // the next character of the source file
  uint errorCount = 0;       // total number of errors encountered
  uint lineNo = 0;           // line numbers for the listing

  stack<string> operatorStk;  // operator stack
  stack<string> operandStk;   // operand stack
  int currentTempNo = -1;     // current temp number
  int maxTempNo = -1;         // max temp number
  string contentsOfAReg;      // symbolic contents of A register
};

#endif
```

Pascallite Stage 1 main() (`/usr/local/4301/src/stage1main.C`)

```
#include <stage1.h>

int main(int argc, char **argv)
{
  // This program is the stage1 compiler for Pascallite.  It will accept
  // input from argv[1], generate a listing to argv[2], and write object
  // code to argv[3].

  if (argc != 4)         // Check to see if pgm was invoked correctly
  {
    // No; print error msg and terminate program
    cerr << "Usage:  " << argv[0] << " SourceFileName ListingFileName "
         << "ObjectFileName" << endl;
    exit(EXIT_FAILURE);
  }

  Compiler myCompiler(argv);

  myCompiler.createListingHeader();
  myCompiler.parser();
  myCompiler.createListingTrailer();

  return 0;
}
```

Pseudocode for Selected Emit Member Functions

emitAdditionCode()

```
void emitAdditionCode(string operand1,string operand2) //add operand1 to operand2
{
  if type of either operand is not integer
    processError(illegal type)
  if the A Register holds a temp not operand1 nor operand2 then
    emit code to store that temp into memory
    change the allocate entry for the temp in the symbol table to yes
    deassign it
  if the A register holds a non-temp not operand1 nor operand2 then deassign it
  if neither operand is in the A register then
    emit code to load operand2 into the A register
  emit code to perform register-memory addition
  deassign all temporaries involved in the addition and free those names for reuse
  A Register = next available temporary name and change type of its symbol table entry to integer
  push the name of the result onto operandStk
}
```

emitDivisionCode()

Division is slightly more complex because the x86 instruction set requires a double register in order to perform this operation, the A-D pair. *Operand2* is loaded into the A register. The contents of this register is then divided by *operand1*, leaving the quotient in the A register. At this point in the compiler, you should presume that no program ever attempts to divide by zero, which is, of course, illegal and is trapped by the hardware.

```
void emitDivisionCode(string operand1,string operand2) //divide operand2 by operand1
{
  if type of either operand is not integer
    processError(illegal type)
  if the A Register holds a temp not operand2 then
     emit code to store that temp into memory
     change the allocate entry for it in the symbol table to yes
     deassign it
  if the A register holds a non-temp not operand2 then deassign it
  if operand2 is not in the A register
    emit instruction to do a register-memory load of operand2 into the A register
  emit code to extend sign of dividend from the A register to edx:eax
  emit code to perform a register-memory division
  deassign all temporaries involved and free those names for reuse
  A Register = next available temporary name and change type of its symbol table entry to integer
  push the name of the result onto operandStk
}
```

emitAndCode()

```
void emitAndCode(string operand1,string operand2) //and operand1 to operand2
{
  if type of either operand is not boolean
    processError(illegal type)
  if the A Register holds a temp not operand1 nor operand2 then
    emit code to store that temp into memory
    change the allocate entry for the temp in the symbol table to yes
    deassign it
  if the A register holds a non-temp not operand1 nor operand2 then deassign it
  if neither operand is in the A register then
    emit code to load operand2 into the A register
  emit code to perform register-memory and
  deassign all temporaries involved in the and operation and free those names for reuse
  A Register = next available temporary name and change type of its symbol table entry to boolean
  push the name of the result onto operandStk
}
```

emitEqualityCode()

The relational operation '=' is performed between any two operands of the same type.

```
void emitEqualityCode(string operand1,string operand2) //test whether operand2 equals operand1
{
  if types of operands are not the same
    processError(incompatible types)
  if the A Register holds a temp not operand1 nor operand2 then
     emit code to store that temp into memory
     change the allocate entry for it in the symbol table to yes
     deassign it
  if the A register holds a non-temp not operand2 nor operand1 then deassign it
  if neither operand is in the A register then
    emit code to load operand2 into the A register
  emit code to perform a register-memory compare
  emit code to jump if equal to the next available Ln (call getLabel)
  emit code to load FALSE into the A register
  insert FALSE in symbol table with value 0 and external name false
  emit code to perform an unconditional jump to the next label (call getLabel should be L(n+1))
  emit code to label the next instruction with the first acquired label Ln
  emit code to load TRUE into A register
  insert TRUE in symbol table with value -1 and external name true
  emit code to label the next instruction with the second acquired label L(n+1)
  deassign all temporaries involved and free those names for reuse
  A Register = next available temporary name and change type of its symbol table entry to boolean
  push the name of the result onto operandStk
}
```

emitAssignCode()

```
void emitAssignCode(string operand1,string operand2) //assign the value of operand1 to operand2
{
  if types of operands are not the same
    processError(incompatible types)
  if storage mode of operand2 is not VARIABLE
    processError(symbol on left-hand side of assignment must have a storage mode of VARIABLE)
  if operand1 = operand2 return
  if operand1 is not in the A register then
    emit code to load operand1 into the A register
  emit code to store the contents of that register into the memory location pointed to by
    operand2
  set the contentsOfAReg = operand2
  if operand1 is a temp then free its name for reuse
  //operand2 can never be a temporary since it is to the left of ':='
}
```

emitReadCode()

```
void emitReadCode(string operand, string operand2)
{
  string name
  while (name is broken from list (operand) and put in name != "")
  {
    if name is not in symbol table
      processError(reference to undefined symbol)
    if data type of name is not INTEGER
      processError(can't read variables of this type)
    if storage mode of name is not VARIABLE
      processError(attempting to read to a read-only location)
    emit code to call the Irvine ReadInt function
    emit code to store the contents of the A register at name
    set the contentsOfAReg = name
  }
}
```

emitWriteCode()

```
void emitWriteCode(string operand, string operand2)
{
  string name
  static bool definedStorage = false
  while (name is broken from list (operand) and put in name != "")
  {
    if name is not in symbol table
      processError(reference to undefined symbol)
    if name is not in the A register
      emit the code to load name in the A register
      set the contentsOfAReg = name
    if data type of name is INTEGER
      emit code to call the Irvine WriteInt function
    else // data type is BOOLEAN
    {
      emit code to compare the A register to 0
      acquire a new label Ln
      emit code to jump if equal to the acquired label Ln
      emit code to load address of TRUE literal in the D register
      acquire a second label L(n + 1)
      emit code to unconditionally jump to label L(n + 1)
      emit code to label the next line with the first acquired label Ln
      emit code to load address of FALSE literal in the D register
      emit code to label the next line with the second acquired label L(n + 1)
      emit code to call the Irvine WriteString function
      if static variable definedStorage is false
      {
        set definedStorage to true
        output an endl to objectFile
        emit code to begin a .data SECTION
        emit code to create label TRUELIT, instruction db, operands 'TRUE',0
        emit code to create label FALSELIT, instruction db, operands 'FALSE',0
        output an endl to objectFile
        emit code to resume .text SECTION
      } // end if
    } // end else
    emit code to call the Irvine Crlf function
  } // end while
}
```

freeTemp(), getTemp()

All code emitting procedures just given refer to "books" which must be "adjusted" to reflect changing uses of temporary locations. When a new temporary is needed, one must be created. When an old temporary is no longer needed, it must be discarded. There are two utility routines for this called `getTemp()` and `freeTemp()`, respectively. `getTemp()` returns the name of the next temporary, and if necessary, will force allocation of a full-word to correspond to that name. `freeTemp()` releases the name of the last temporary created by a call to `getTemp()`; however, the storage (if any) allocated by the earlier call to `getTemp()` remains allocated after the call to `freeTemp()`. Once storage for a temporary has been allocated, it may not be de-allocated. Note that both `freeTemp()` and `getTemp()` refer to private data within a `Compiler` object, `currentTempNo` and `maxTempNo`. Both should be initialized to `-1` to reflect the fact that initially no temporaries are allocated (`T0` will be the first temporary name used).

```
void freeTemp()
{
  currentTempNo--;
  if (currentTempNo < -1)
      processError(compiler error, currentTempNo should be ≥ -1)
}

string getTemp()
{
  string temp;
  currentTempNo++;
  temp =  "T" + currentTempNo;
  if (currentTempNo > maxTempNo)
    insert(temp, UNKNOWN, VARIABLE, "", NO, 1)
    maxTempNo++
  return temp
}
```

Commands to compile, link, and run Stage 1

```
mmotl@csunix ~/4301> # Create a folder for stage1 and change into it
mmotl@csunix ~/4301> mkdir stage1
mmotl@csunix ~/4301> cd stage1
mmotl@csunix ~/4301/stage1> cp /usr/local/4301/src/Makefile .
mmotl@csunix ~/4301/stage1> # Edit Makefile adding a target of
mmotl@csunix ~/4301/stage1> # stage1 to targets2srcfiles
mmotl@csunix ~/4301/stage1> cp /usr/local/4301/include/stage1.h .
mmotl@csunix ~/4301/stage1> cp /usr/local/4301/src/stage1main.C .
mmotl@csunix ~/4301/stage1> make stage1
g++ -g -Wall -std=c++11 -c stage1main.C -I/usr/local/4301/include/ -I.
g++ -g -Wall -std=c++11 -c stage1.cpp -I/usr/local/4301/include/ -I.
g++ -o stage1 stage1main.o stage1.o -L/usr/local/4301/lib/ -lm
mmotl@csunix ~/4301/stage1> # There are numerous data file in
mmotl@csunix ~/4301/stage1> # /usr/local/4301/data/stage1/
mmotl@csunix ~/4301/stage1> ls /usr/local/4301/data/stage1/
101.asm  112.dat  119.asm  124.lst  133.lst  142.dat  149.asm  160.dat  173.dat
101.dat  113.dat  119.dat  125.dat  134.asm  143.dat  149.dat  161.dat  174.dat
101.lst  114.dat  119.lst  126.dat  134.dat  144.dat  149.lst  162.dat  175.dat
102.dat  115.dat  120.dat  127.dat  134.lst  145.dat  150.dat  163.dat  176.dat
103.dat  116.asm  121.dat  128.dat  135.asm  146.asm  151.dat  164.dat  177.dat
104.dat  116.dat  122.asm  129.dat  135.dat  146.dat  152.dat  165.dat  178.dat
105.dat  116.lst  122.dat  130.dat  135.lst  146.lst  153.dat  166.dat
106.dat  117.asm  122.lst  131.dat  136.dat  147.asm  154.dat  167.dat
107.dat  117.dat  123.asm  132.asm  137.dat  147.dat  155.dat  168.dat
108.dat  117.lst  123.dat  132.dat  138.dat  147.lst  156.dat  169.dat
109.dat  118.asm  123.lst  132.lst  139.dat  148.asm  157.dat  170.dat
110.dat  118.dat  124.asm  133.asm  140.dat  148.dat  158.dat  171.dat
111.dat  118.lst  124.dat  133.dat  141.dat  148.lst  159.dat  172.dat
mmotl@csunix ~/4301/stage1> # Copy as many or as few as you like
mmotl@csunix ~/4301/stage1> cp /usr/local/4301/data/stage1/101.dat .
mmotl@csunix ~/4301/stage1> cat 101.dat
program stage1no101;
{demonstrate correct usage of Pascallite features.}
const zero = 0; five = 5;
var a,b : integer;
    c   : integer;
begin
   read(a);
   read(b,c);
   write(a);
   write(b);
   a := five * (3 + 34);
   b := a +a;
   write(five);
   write(a, b, c, five, zero);
end .
mmotl@csunix ~/4301/stage1> # Execute your compiler on dataset 101
mmotl@csunix ~/4301/stage1> ./stage1 101.dat 101.lst 101.asm
mmotl@csunix ~/4301/stage1> cat 101.lst
STAGE1:  YOUR NAME(S)        Wed Nov  4 15:53:39 2020

LINE NO.              SOURCE STATEMENT

    1|program stage1no101;
    2|{demonstrate correct usage of Pascallite features.}
    3|const zero = 0; five = 5;
    4|var a,b : integer;
    5|    c   : integer;
    6|begin
    7|   read(a);
    8|   read(b,c);
```

```
    9|   write(a);
   10|   write(b);
   11|   a := five * (3 + 34);
   12|   b := a +a;
   13|   write(five);
   14|   write(a, b, c, five, zero);
   15|end .

COMPILATION TERMINATED      0 ERRORS ENCOUNTERED
mmotl@csunix ~/4301/stage1> cat 101.asm
; YOUR NAME(S)       Wed Nov  4 15:53:39 2020
%INCLUDE "Along32.inc"
%INCLUDE "Macros_Along.inc"

SECTION .text
global  _start                          ; programstage1no101

_start:
        call    ReadInt                 ; read int; value placed in eax
        mov     [I2],eax                ; store eax at a
        call    ReadInt                 ; read int; value placed in eax
        mov     [I3],eax                ; store eax at b
        call    ReadInt                 ; read int; value placed in eax
        mov     [I4],eax                ; store eax at c
        mov     eax,[I2]                ; load a in eax
        call    WriteInt                ; write int in eax to standard out
        call    Crlf                    ; write \r\n to standard out
        mov     eax,[I3]                ; load b in eax
        call    WriteInt                ; write int in eax to standard out
        call    Crlf                    ; write \r\n to standard out
        mov     eax,[I5]                ; AReg = 3
        add     eax,[I6]                ; AReg = 3 + 34
        imul    dword [I1]              ; AReg = T0 * five
        mov     [I2],eax                ; a = AReg
        add     eax,[I2]                ; AReg = a + a
        mov     [I3],eax                ; b = AReg
        mov     eax,[I1]                ; load five in eax
        call    WriteInt                ; write int in eax to standard out
        call    Crlf                    ; write \r\n to standard out
        mov     eax,[I2]                ; load a in eax
        call    WriteInt                ; write int in eax to standard out
        call    Crlf                    ; write \r\n to standard out
        mov     eax,[I3]                ; load b in eax
        call    WriteInt                ; write int in eax to standard out
        call    Crlf                    ; write \r\n to standard out
        mov     eax,[I4]                ; load c in eax
        call    WriteInt                ; write int in eax to standard out
        call    Crlf                    ; write \r\n to standard out
        mov     eax,[I1]                ; load five in eax
        call    WriteInt                ; write int in eax to standard out
        call    Crlf                    ; write \r\n to standard out
        mov     eax,[I0]                ; load zero in eax
        call    WriteInt                ; write int in eax to standard out
        call    Crlf                    ; write \r\n to standard out
        Exit    {0}

SECTION .data
I5      dd      3                       ; 3
I6      dd      34                      ; 34
I1      dd      5                       ; five
I0      dd      0                       ; zero

SECTION .bss
I2      resd    1                       ; a
```

```
I3      resd    1                       ; b
I4      resd    1                       ; c
mmotl@csunix ~/4301/stage1> # Edit the Makefile to add a target
mmotl@csunix ~/4301/stage1> # of 101 (or any other dataset) to
mmotl@csunix ~/4301/stage1> # targetsAsmLanguage
mmotl@csunix ~/4301/stage1> make 101
nasm -f elf32  -o 101.o 101.asm -I/usr/local/4301/include/ -I.
ld -m elf_i386 --dynamic-linker /lib/ld-linux.so.2 -o 101 101.o \
/usr/local/4301/src/Along32.o -lc
mmotl@csunix ~/4301/stage1> ls 101*
101  101.dat  101.lst  101.asm  101.o
mmotl@csunix ~/4301/stage1> # Note that 101 assembled and linked
mmotl@csunix ~/4301/stage1> # with no errors
mmotl@csunix ~/4301/stage1> # Execute ./101 to ensure it runs without
mmotl@csunix ~/4301/stage1> # errors. Note that this program reads
mmotl@csunix ~/4301/stage1> # three integers from the keyboard: a,b,c
mmotl@csunix ~/4301/stage1> # It then writes a and b. It computes new
mmotl@csunix ~/4301/stage1> # values for a and b and then writes
mmotl@csunix ~/4301/stage1> # five (5), a (185), b (370), c, five (5),
mmotl@csunix ~/4301/stage1> # and zero (0)
mmotl@csunix ~/4301/stage1> ./101
-4301
3304
2336
-4301
+3304
+5
+185
+370
+2336
+5
+0
mmotl@csunix ~/4301/stage1> # It works!
mmotl@csunix ~/4301/stage1> # You can diff the .asm and .lst files
mmotl@csunix ~/4301/stage1> # Move on to the next dataset
mmotl@csunix ~/4301/stage1>
```