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 RAMM 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'}

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

	→	'(' EXPRESS ')'	{'('}
	→	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'
'+'(binary) '-'(binary) 'or' Decreasing 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.

Pascallite Translation Grammar Stage 1		
Revised Productions:		
1. BEGIN_END_STMT	→	'begin' EXEC_STMTS 'end' '.' *Code('end','.')*
New Productions:		
2. EXEC_STMTS	→	EXEC_STMT EXEC_STMTS
	→	ε
3. EXEC_STMT	→	ASSIGN_STMT
	→	READ_STMT
	→	WRITE_STMT
4. ASSIGN_STMT	→	$NON\_KEY\_ID_x$ *PushOperand(x)* ':=' *PushOperator(':=')* EXPRESS ';' *Code(PopOperator,PopOperand,PopOperand)*
5. READ_STMT	→	'read' READ_LIST ';'
6. READ_LIST	→	'(' IDS_x ')' *Code('read',x)*
7. WRITE_STMT	→	'write' WRITE_LIST ';'
8. WRITE_LIST	→	'(' IDS_x ')' *Code('write',x)*
9. EXPRESS	→	TERM EXPRESSES
10. EXPRESSES	→	REL_OP_x *PushOperator(x)* TERM *Code(PopOperator,PopOperand,PopOperand)* EXPRESSES
	→	ε
11. TERM	→	FACTOR TERMS
12. TERMS	→	$ADD\_LEVEL\_OP_x$ *PushOperator(x)* FACTOR *Code(PopOperator,PopOperand,PopOperand)* TERMS
	→	ε
13. FACTOR	→	PART FACTORS
14. FACTORS	→	$MULT\_LEV\_OP_x$ *PushOperator(x)* PART *Code(PopOperator,PopOperand,PopOperand)* FACTORS
	→	ε
15. PART	→	'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)*)

		→	'-' ('(' EXPRESS ')' *Code('neg',PopOperand)* | INTEGER$_x$ *PushOperand('-'+ x)* | NON_KEY_ID$_x$ *Code('neg',x)*)
		→	INTEGER$_x$ | BOOLEAN$_x$ | NON_KEY_ID$_x$ *PushOperand(x)*
		→	'(' EXPRESS ')'
16.	REL_OP	→	'=' | '<>' | '<=' | '>=' | '<' | '>'
17.	ADD_LEVEL_OP	→	'+' | '-' | 'or'
18.	MULT_LEVEL_OP	→	'*' | 'div' | 'mod' | 'and'

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 on the RAMM computer. 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
```

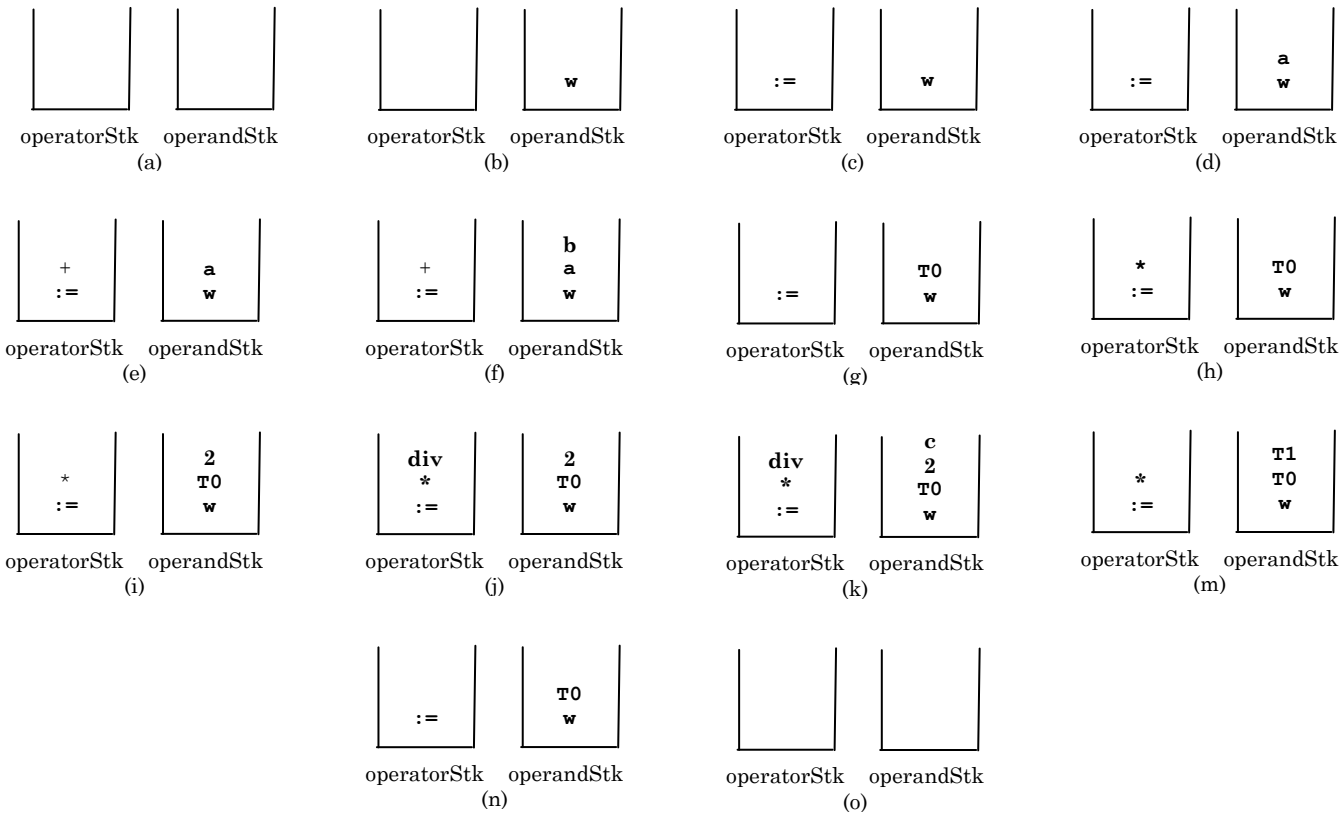
The figures below show the changes to the two stacks during the calls to these action routines. Initially, both stacks are empty. The first five action calls push three operands and two operators onto `operandStk` and `operatorStk`, respectively, shown in (a) through (f). Call (6) adds `a` to `b`, pushing 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.



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  process error: operator stack underflow;
}

string PopOperand()  //pop name from operandStk
{
  if operandStk is not empty
    return top element removed from stack;
  else  process 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', 'or', '=', and ':='. Each operation is from a different class or illustrates some new aspect of compilation not revealed in the others.

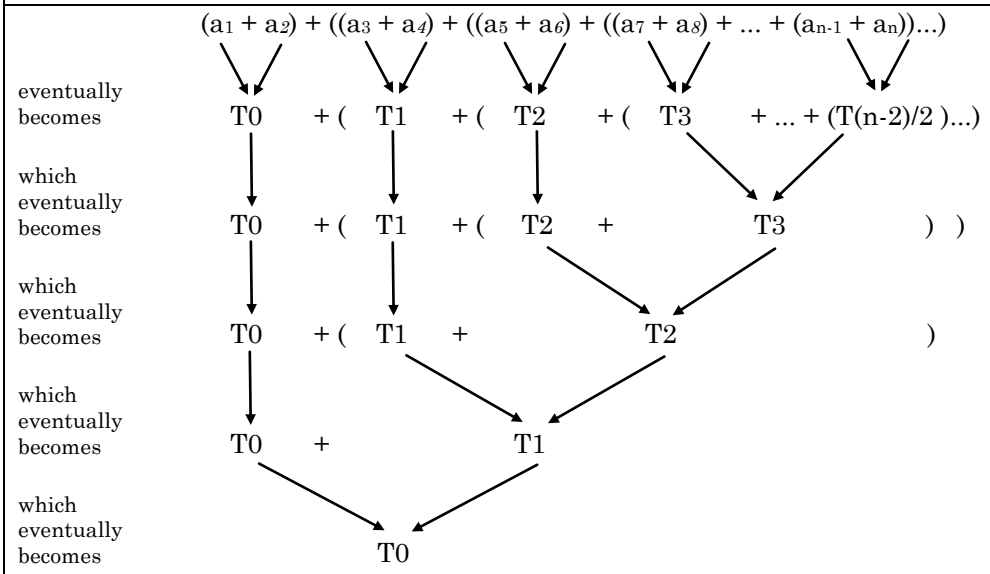
```
void Code(string operator, string operand1 = "", string operand2 = "")
{
  switch(operator)
  {
    case 'program': emit first RAMM instruction STRT NOP …
    case 'end'    : emit HLT, BSS and DEC pseudo ops, and END
    case 'read'   : emit read code;
    case 'write'  : emit write code
    '+'           : emit addition code;     //this must be binary '+'
    '-'           : emit subtraction code;  //this must be binary '-'
    'neg'         : emit negation code;     //this must be unary '-'
    'not'         : emit not code;
    '*'           : emit multiplication code;
    'div'         : emit division code;
    'mod'         : emit modulo code;
    'and'         : emit and code;
    ...
    '='           : emit equality code;
    ':='          : emit assignment code;
    default       : process error: undefined operation
  }
```

Register Allocation and Assignment - Stage1

In stage1 register assignment scheme, both registers `A` and `Q` will be used, but with `Q` 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.