Overall Compiler Structure – Control Structures - Stage 2

Stage2 adds three new control structures to allow conditional and repeated execution of Pascallite source code.

1. **if-then-else** statement
2. **while-do** statement
3. **repeat-until** statement

The first statement, which permits conditional execution of code, actually appears in two forms, with and without an **else** clause, while the second and third statements, employed for looping, have only one form. With their inclusion, Pascallite will almost become a viable language for simple but nontrivial programs.

Several new concepts will be introduced as well. First, all of these new statements are *compound* statements. As such, you must be able to properly detect the end of each statement which may be arbitrarily far from its beginning. This problem is further compounded by the fact that these statements may be nested inside one another arbitrarily deep.

Pascallite Language Definition - Stage 2

1.	Seven new keywords are added: **if, then, else, while, do, repeat, until** All of these keywords identify clauses of the three control structures augmenting Pascallite. No new tokens are added to Pascallite here.

Pascallite Grammar Stage 2

The context-free grammar for stage2 features along with the selection set of each alternative production is given below. A production is not repeated unless it has somehow changed to accommodate the new language features. *In several cases, selection sets of productions not listed must be augmented.* That task is left to you.

Revised Productions:

1.	BEGIN_END_STMT →	'begin' EXEC_STMTS 'end'	{'begin'}
		(	
		'.'	{'.'}
		| ';'	{';'}
		)	
2.	EXEC_STMT →	ASSIGN_STMT	{NON_KEY_ID}
	→	READ_STMT	{'read'}
	→	WRITE_STMT	{'write'}
	→	IF_STMT	{'if'}
	→	WHILE_STMT	{'while'}
	→	REPEAT_STMT	{'repeat'}
	→	NULL_STMT	{';'}
	→	BEGIN_END_STMT	{'begin'}

New Productions:

3.	IF_STMT →	'if' EXPRESS 'then' EXEC_STMT ELSE_PT	{'if'}
4.	ELSE_PT →	'else' EXEC_STMT	{'else'}
	→	ε	{'end',NON_KEY_ID,';','until','begin', 'while','if','repeat','read','write'}
5.	WHILE_STMT →	'while' EXPRESS 'do' EXEC_STMT	{'while'}
6.	REPEAT_STMT →	'repeat' EXEC_STMTS 'until' EXPRESS ';'	{'repeat'}
7.	NULL_STMT →	';'	{';'}

The only context-sensitive constraint on programs not expressed by the grammar is that the value of the expression in an **if**, **while**, or **repeat** statement must be of type *boolean*.

if statement – both forms

The semantics of the **if** statement is shown below. You should be familiar with this nearly universal statement.

'if'	
express	evaluate express
'then'	branch to L_1 if express is false
EXEC_STMT	execute EXEC_STMT
	L_1:

(a) without **else** clause

'if'	
express	evaluate express
'then'	branch to L_1 if express is false
EXEC_STMT	execute EXEC_STMT
'else'	branch to L_2
	L_1:
EXEC_STMT	execute EXEC_STMT
	L_2:

(b) with **else** clause

The **if** statement presents an unusual situation. If you carefully examine the ELSE_PT production, you will note that part of one selection set is missing. The productions should actually be:

```
ELSE_PT    →  'else'  EXEC_STMT                                  {'else'}
           →   ε                          {'end',NON_KEY_ID,';','until','begin',
                                           'while','if','repeat','read','write',
                                                                          'else'}
```

The 'else' has been omitted from the selection set of production 4. Note that with the full selection set listed, a key conflict results; hence, the grammar is not strictly LL(1), as promised. Here is an explanation of why 'else' has been left out of production 4's selection set. The **if** statement suffers from a classic ambiguity, called the "dangling **else**" problem, which manifests itself in the following code segment:

```
if p then if q then r else s                                        (i)
```

Both *p* and *q* are *boolean* predicates, while *r* and *s* are statements. Which **if** does the "dangling" **else** match? The question becomes clearer when you consider two alternative reformattings of (i) above.

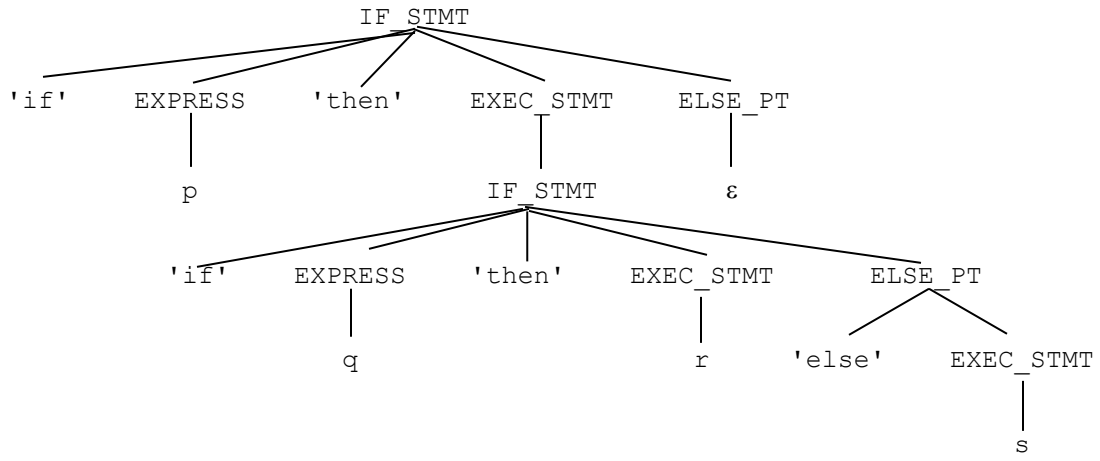
```
if p then                                                           (ii)
  if q then r
  else s
```

and

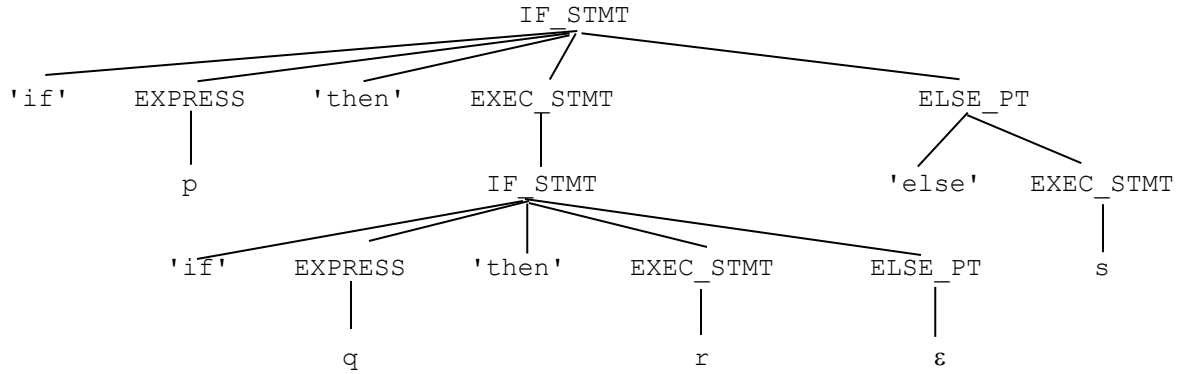
```
if p then                                                           (iii)
  if q then r
else s
```

The first binds (in appearance) the **else** to the inner and closest **if**, while the second binds the **else** clause to the outer and farthest **if**. The parse trees for these two bindings are shown below. The grammar is ambiguous, with these two trees expressing the nature of the ambiguity.

The ambiguity in the *language* must be resolved first. This will dictate how to resolve the ambiguity in the grammar. Pascallite, like most languages, opts for the interpretation shown in the parse tree (ii). No known language opts for the parse tree (iii). The grammar must allow only for this interpretation, but unfortunately, there is no unambiguous grammar for Pascallite which has this optional **else** clause. You should rely upon an alternative which is occasionally taken by compiler writers—to force the choice of one production over another by deleting elements from selection sets, production 4 being an embodiment of this strategy. A willingness to resolve selection set conflicts in this manner significantly increases the class of grammars to which the LL(1) parsing algorithm is applicable, even though these grammars are not strictly LL(1).



parse tree (ii)



parse tree (iii)

while statement

The **while** statement provides for conditional repetitive execution of a single (possibly compound) statement. When the expression following `'while'` is evaluated, if it is false, control resumes following the end of the **while** statement. If it is true, the statement following `'do'` is executed. When its execution is complete, control returns back to the point where the expression is again evaluated, and the entire process is repeated until the expression eventually turns false (or else the program is in an infinite loop).

`'while'`	L_1:
`express`	evaluate `express`
`'do'`	branch to L_2 if `express` is false
`EXEC_STMT`	execute `EXEC_STMT`
	branch to L_1
	L_2:

repeat statement

The **repeat** statement is quite similar to the **while** statement in that both are used for looping, but with two major differences: the predicate is evaluated *after* the loop body has been executed, not *before*, as in a **while** statement. Consequently, the loop body (statements between `'repeat'` and `'until'`) is always executed at least once, even if the predicate is initially false. For a **while** statement, this is not true. If the predicate is initially false, the loop body (statement following `'do'`) is skipped entirely. The second difference is that the **while** loop is executed repeatedly until the predicate is false, while the **repeat** statement is executed repeatedly until the predicate is true.

`'repeat'`	L_1:
`EXEC_STMTS`	execute `EXEC_STMTS`
`'until'`	
`express`	evaluate `express`
`';'`	branch to L_1 if `express` is false

Pascallite Translation Grammar Stage 2

Revised Productions:

1. BEGIN_END_STMT → 'begin' EXEC_STMTS 'end'
   ('.'$_x$ | ';'$_x$) *code('end',x)*

2. EXEC_STMT → ASSIGN_STMT
   → READ_STMT
   → WRITE_STMT
   → IF_STMT
   → WHILE_STMT
   → REPEAT_STMT
   → NULL_STMT
   → BEGIN_END_STMT

New Productions:

3. IF_STMT → 'if' EXPRESS 'then'
   code('then',popOperand())
   EXEC_STMT ELSE_PT

4. ELSE_PT → 'else' *code('else',popOperand())*
   EXEC_STMT *code('post_if',popOperand())*
   → ε *code('post_if',popOperand())*

5. WHILE_STMT → 'while' *code('while')* EXPRESS 'do'
   code('do',popOperand()) EXEC_STMT
   code('post_while',popOperand(),popOperand())

6. REPEAT_STMT → 'repeat' *code('repeat')* EXEC_STMTS
   'until' EXPRESS *code('until',popOperand(),popOperand())*
   ';'

7. NULL_STMT → ';'

Pascallite Stage 2 Header File (/usr/local/4301/include/stage2.h)

```
#ifndef STAGE2_H
#define STAGE2_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

  void ifStmt();          // stage 2, production 3
  void elsePt();          // stage 2, production 4
  void whileStmt();       // stage 2, production 5
  void repeatStmt();      // stage 2, production 6
  void nullStmt();        // stage 2, production 7

  // 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();

  // Emit Functions for Stage 1
  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

  // Emit functions for Stage 2
  void emitThenCode(string operand1, string = "");
  // emit code which follows 'then' and statement predicate
  void emitElseCode(string operand1, string = "");
  // emit code which follows 'else' clause of 'if' statement
  void emitPostIfCode(string operand1, string = "");
  // emit code which follows end of 'if' statement
  void emitWhileCode(string = "", string = "");
  // emit code following 'while'
  void emitDoCode(string operand1, string = "");
  // emit code following 'do'
  void emitPostWhileCode(string operand1, string operand2);
  // emit code at end of 'while' loop;
  // operand2 is the label of the beginning of the loop
  // operand1 is the label which should follow the end of the loop
  void emitRepeatCode(string = "", string = "");
  // emit code which follows 'repeat'
  void emitUntilCode(string operand1, string operand2);
  // emit code which follows 'until' and the predicate of loop
  // operand1 is the value of the predicate
  // operand2 is the label which points to the beginning of the loop

  // 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
  bool isLabel(string s) const;     // determines if s represents a label

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 2 main() (/usr/local/4301/src/stage2main.C)

```
#include <stage2.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;
}
```

Code Generation - Stage 2

Add new alternatives to the **case** statement for `code` for the new arguments with which it can legitimately be called.

Begin with the revisions to `code` for the **end** statement since these are quite trivial. In revised production number one, if the punctuation following the keyword `'end'` is a period, then the entire program has terminated; otherwise, the punctuation is a semicolon, and a nested **begin-end** block has just been terminated, but the program should continue. Recall that all programs must end with a period and that the only legal way in which a

statement may end with a period is if it is the final `'end'` of the program. No special action is required for a **begin-end** statement terminating with a semicolon.

In the previous stages, *operandStk* has held only operands of the various operators found in Pascallite source code such as `':='`, `'='`, `'div'`, or `'and'`, among many others. We now extend the possible elements of *operandStk* to include *labels* as well. Labels which stage2 generates will have the form *`'.Ln'`*, where *n* is a non-negative integer. Initially, *n* is `-1` and is incremented by one with each call for a new label, so that the first label is `'.L0'`. The need for labels is illustrated back in the figures for the **if**, **repeat**, and **while** structures, which show effectively how the implementations of these three statements appear. Branch instructions are issued either to skip over code that is to be conditionally executed or to return to the beginning of a loop from its end. In either case, the operand of the branch is a label of some other statement that must be pushed onto *operandStk*.

Pseudocode for Selected Emit Member Functions

emitThenCode()

```
void emitThenCode(string operand1, string operand2)
//emit code that follows 'then' and statement predicate
{
  string tempLabel
  if the type of operand1 is not boolean
    processError(if predicate must be of type boolean)
  assign next label to tempLabel
  if operand1 is not in the A register then
    emit instruction to move operand1 to the A register
  emit instruction to compare the A register to zero (false)
  emit code to branch to tempLabel if the compare indicates equality
  push tempLabel onto operandStk so that it can be referenced when emitElseCode() or
    emitPostIfCode() is called
  if operand1 is a temp then
    free operand's name for reuse
  deassign operands from all registers
}
```

emitElseCode()

```
void emitElseCode(string operand1, string operand2)
//emit code that follows else clause of if statement
{
  string tempLabel
  assign next label to tempLabel
  emit instruction to branch unconditionally to tempLabel
  emit instruction to label this point of object code with the argument operand1
  push tempLabel onto operandStk
  deassign operands from all registers
}
```

emitPostIfCode

```
void emitPostIfCode(string operand1, string operand2)
//emit code that follows end of if statement
{
  emit instruction to label this point of object code with the argument operand1
  deassign operands from all registers
}
```

EmitWhileCode()

```
void emitWhileCode(string operand1, string operand2)  //emit code that follows while
{
  string tempLabel
  assign next label to tempLabel
  emit instruction to label this point of object code as tempLabel
  push tempLabel onto operandStk
  deassign operands from all registers
}
```

emitDoCode()

```
void emitDoCode(string operand1, string operand2)  //emit code that follows do
{
  string tempLabel
  if the type of operand1 is not boolean
    processError(while predicate must be of type boolean)
  assign next label to tempLabel
  if operand1 is not in the A register then
    emit instruction to move operand1 to the A register
  emit instruction to compare the A register to zero (false)
  emit code to branch to tempLabel if the compare indicates equality
  push tempLabel onto operandStk
  if operand1 is a temp then
    free operand's name for reuse
  deassign operands from all registers
}
```

emitPostWhileCode()

```
void emitPostWhileCode(string operand1, string operand2)
//emit code at end of while loop, operand2 is the label of the beginning of the loop,
//operand1 is the label which should follow the end of the loop
{
  emit instruction which branches unconditionally to the beginning of the loop, i.e., to the
    value of operand2
  emit instruction which labels this point of the object code with the argument operand1
  deassign operands from all registers
}
```

emitRepeatCode()

```
void emitRepeatCode(string operand1, string operand2)  //emit code that follows repeat
{
  string tempLabel
  assign next label to tempLabel
  emit instruction to label this point in the object code with the value of tempLabel
  push tempLabel onto operandStk
  deassign operands from all registers
}
```

emitUntilCode()

```
void emitUntilCode(string operand1, string operand2)
//emit code that follows until and the predicate of loop.  operand1 is the value of the
//predicate.  operand2 is the label that points to the beginning of the loop
{
  if the type of operand1 is not boolean
    processError(if predicate must be of type boolean)
  if operand1 is not in the A register then
    emit instruction to move operand1 to the A register
  emit instruction to compare the A register to zero (false)
  emit code to branch to operand2 if the compare indicates equality

  if operand1 is a temp then
    free operand1's name for reuse
  deassign operands from all registers
}
```

Commands to compile, link, and run Stage 2

```
mmotl@csunix ~/4301> # Create a folder for stage2 and change into it
mmotl@csunix ~/4301> mkdir stage2
mmotl@csunix ~/4301> cd stage2
mmotl@csunix ~/4301/stage2> cp /usr/local/4301/src/Makefile .
mmotl@csunix ~/4301/stage2> # Edit Makefile adding a target of
mmotl@csunix ~/4301/stage2> # stage2 to targets2srcfiles
mmotl@csunix ~/4301/stage2> cp /usr/local/4301/include/stage2.h .
mmotl@csunix ~/4301/stage2> cp /usr/local/4301/src/stage2main.C .
mmotl@csunix ~/4301/stage2> make stage2
g++ -g -Wall -std=c++11 -c stage2main.C -I/usr/local/4301/include/ -I.
g++ -g -Wall -std=c++11 -c stage2.cpp -I/usr/local/4301/include/ -I.
g++ -o stage2 stage2main.o stage2.o -L/usr/local/4301/lib/ -lm
mmotl@csunix ~/4301/stage2> # There are numerous data files in
mmotl@csunix ~/4301/stage2> # /usr/local/4301/data/stage2/
mmotl@csunix ~/4301/stage2> ls /usr/local/4301/data/stage2/
201.dat  204.lst  208.dat  214.dat  219.dat  225.dat  230.dat  237.dat  245.dat
202.asm  205.dat  208.lst  215.dat  220.dat  225.lst  230.lst  237.lst  246.dat
202.dat  206.asm  209.dat  216.dat  221.dat  226.asm  231.dat  238.dat  247.dat
202.lst  206.dat  210.dat  217.asm  222.dat  226.dat  232.dat  239.dat  248.dat
203.asm  206.lst  211.dat  217.dat  223.dat  226.lst  233.dat  240.dat  249.dat
203.dat  207.asm  212.asm  217.lst  224.asm  227.dat  234.dat  241.dat
203.lst  207.dat  212.dat  218.asm  224.dat  228.dat  235.dat  242.dat
204.asm  207.lst  212.lst  218.dat  224.lst  229.dat  236.dat  243.dat
204.dat  208.asm  213.dat  218.lst  225.asm  230.asm  237.asm  244.dat
mmotl@csunix ~/4301/stage2> # Copy as many or as few as you like
mmotl@csunix ~/4301/stage2> # Let's copy dataset 202 as an example
mmotl@csunix ~/4301/stage2> cp /usr/local/4301/data/stage2/202.dat .
mmotl@csunix ~/4301/stage2> cat 202.dat
program stage2no202;
  var y,z:integer;
  begin
    read(y);
    while (y < 10) do
      y:=y+1;
    z:=y+1;
    write(y);
    write(z);
  end.
mmotl@csunix ~/4301/stage2> # Execute your stage2 compiler on 202.dat
mmotl@csunix ~/4301/stage2> ./stage2 202.dat 202.lst 202.asm
```

```
mmotl@csunix ~/4301/stage2> cat 202.lst
STAGE2:  YOUR NAME(S)       Thu Apr 22 10:40:16 2021

LINE NO.              SOURCE STATEMENT

    1|program stage2no202;
    2|  var y,z:integer;
    3|  begin
    4|    read(y);
    5|    while (y < 10) do
    6|      y:=y+1;
    7|    z:=y+1;
    8|    write(y);
    9|    write(z);
   10|  end.

COMPILATION TERMINATED      0 ERRORS ENCOUNTERED
mmotl@csunix ~/4301/stage2> cat 202.asm
; YOUR NAME(S)       Thu Apr 22 10:40:16 2021
%INCLUDE "Along32.inc"
%INCLUDE "Macros_Along.inc"

SECTION .text
global  _start                          ; program stage2no202

_start:
        call    ReadInt                 ; read int; value placed in eax
        mov     [I0],eax                ; store eax at y
.L0:                                    ; while
        mov     eax,[I0]                ; AReg = y
        cmp     eax,[I2]                ; compare y and 10
        jl      .L1                     ; if y < 10 then jump to set eax to TRUE
        mov     eax,[FALSE]             ; else set eax to FALSE
        jmp     .L2                     ; unconditionally jump
.L1:
        mov     eax,[TRUE]              ; set eax to TRUE
.L2:
        cmp     eax,0                   ; compare eax to 0
        je      .L3                     ; if T0 is false then jump to end while
        mov     eax,[I0]                ; AReg = y
        add     eax,[I3]                ; AReg = y + 1
        mov     [I0],eax                ; y = AReg
        jmp     .L0                     ; end while
.L3:
        mov     eax,[I0]                ; AReg = y
        add     eax,[I3]                ; AReg = y + 1
        mov     [I1],eax                ; z = AReg
        mov     eax,[I0]                ; load y in eax
        call    WriteInt                ; write int in eax to standard out
        call    Crlf                    ; write \r\n to standard out
        mov     eax,[I1]                ; load z in eax
        call    WriteInt                ; write int in eax to standard out
        call    Crlf                    ; write \r\n to standard out
        Exit    {0}

SECTION .data
I3      dd      1                       ; 1
I2      dd      10                      ; 10
FALSE   dd      0                       ; false
TRUE    dd      -1                      ; true

SECTION .bss
I0      resd    1                       ; y
I1      resd    1                       ; z
```

```
mmotl@csunix ~/4301/stage2> # diff the .lst and .asm files
mmotl@csunix ~/4301/stage2> diff 202.lst /usr/local/4301/data/stage2/202.lst
1c1
< STAGE2:  YOUR NAME(S)       Thu Apr 22 10:40:16 2021
---
> STAGE1:  YOUR NAME(S)       Mon Feb  8 14:20:33 2021
mmotl@csunix ~/4301/stage2> diff 202.asm /usr/local/4301/data/stage2/202.asm
1c1
< ; YOUR NAME(S)       Thu Apr 22 10:40:16 2021
---
> ; YOUR NAME(S)       Mon Feb  8 14:20:33 2021
mmotl@csunix ~/4301/stage2> # Edit the Makefile to add a target of 202
mmotl@csunix ~/4301/stage2> # (or any other dataset) to
mmotl@csunix ~/4301/stage2> # targetsAsmLanguage
mmotl@csunix ~/4301/stage2> make 202
nasm -f elf32  -o 202.o 202.asm -I/usr/local/4301/include/ -I.
ld -m elf_i386 --dynamic-linker /lib/ld-linux.so.2 -o 202 202.o \
/usr/local/4301/src/Along32.o -lc
mmotl@csunix ~/4301/stage2> ls 202*
202  202.asm  202.dat  202.lst  202.o
mmotl@csunix ~/4301/stage2> # Note that 202 assembled (created 202.o)
mmotl@csunix ~/4301/stage2> # and linked (created 202) with no errors
mmotl@csunix ~/4301/stage2> # Execute ./202 to ensure it runs without
mmotl@csunix ~/4301/stage2> # errors. This program reads an integer
mmotl@csunix ~/4301/stage2> # value for y (I'll enter a value of 4),
mmotl@csunix ~/4301/stage2> # enters a while loop to increment y to 10,
mmotl@csunix ~/4301/stage2> # places 11 (y + 1) in z, and then prints
mmotl@csunix ~/4301/stage2> # y and z (10 and 11).
mmotl@csunix ~/4301/stage2> ./202
4
+10
+11
mmotl@csunix ~/4301/stage2> # It works!
mmotl@csunix ~/4301/stage2> # Move on to the next dataset.
mmotl@csunix ~/4301/stage2>
```