Overall Compiler Structure - Stage 0

Following is the overall framework for stage0, the first stage of the Pascallite compiler, including the main routine and the interfaces between that routine and its major components. All of the stages are organized as a translation grammar processor. The grammar given below is an LL(1) grammar so that the processor can generate a leftmost derivation of programs without backtracking. The grammar given below includes the *action* symbols needed to build the symbol table (with the selection sets omitted).

Pascallite Grammar Stage 0

1.	PROG	→	PROG_STMT CONSTS VARS BEGIN_END_STMT
2.	PROG_STMT	→	'program' $NON\_KEY\_ID_x$ ';'
			code('program', x); insert(x,PROG_NAME,CONSTANT,x,NO,0)
3.	CONSTS	→	'const' CONST_STMTS
		→	ε
4.	VARS	→	'var' VAR_STMTS
		→	ε
5.	BEGIN_END_STMT	→	'begin' 'end' '.' *code('end', '.')*
6.	CONST_STMTS	→	$NON\_KEY\_ID_x$ '='($NON\_KEY\_ID_y$ | 'not' $NON\_KEY\_ID_y$ | LIT_y) ';'
			insert(x,whichType(y),CONSTANT,whichValue(y),YES,1)
			(CONST_STMTS | ε)
7.	VAR_STMTS	→	IDS_x ':' $TYPE_y$ ';'
			insert(x,y,VARIABLE,ε,YES,1)
			(VAR_STMTS | ε)
8.	IDS	→	NON_KEY_ID (',' IDS | ε)
9.	TYPE	→	'integer'
		→	'boolean'
10.	LIT	→	INTEGER | BOOLEAN | 'not' BOOLEAN | '+' INTEGER | '-' INTEGER
11.	BOOLEAN	→	'true' | 'false'

Note that in production 6, subscript *y* is used twice. This does not contradict restriction 7 on the form of valid productions of a translation grammar, however, since *y* is subscripting alternatives. Production 6 is actually an abbreviation for *two* productions; therefore, *y* can never be the value of both alternatives simultaneously and no conflict can arise.

There are just four action routines called in this simple translation grammar:

1. `insert(externalName,storeType,mode,value,allocate,units)`
2. `whichType(externalName)`
3. `whichValue(externalName)`
4. `code(op, operand1, operand2)`

These routines are explained further in the following pages.

Following is the pseudo code for the main program for stage0. It is extremely simple, reflecting the fact that most of the actual processing is performed by the parser. The symbol table is defined because this data structure is so pervasively referenced throughout the compiler. To reference an entry in the table for the external name 'trivia', the pseudo code simply writes `symbolTable['trivia']`. If there is no entry under that index, then the value referenced is *undefined*. The detail of how the look-up of entries is handled is left to the programmer.

Pascallite Stage 0 Header File (`/usr/local/4301/include/stage0.h`)

```
#ifndef STAGE0_H
#define STAGE0_H

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

using namespace std;

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

enum storeTypes {INTEGER, BOOLEAN, PROG_NAME};
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

  // 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 = "");

  // Emit Functions
  void emit(string label = "", string instruction = "", string operands = "",
            string comment = "");
  void emitPrologue(string progName, string = "");
  void emitEpilogue(string = "", string = "");
  void emitStorage();
```

```
  // 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);

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
};

#endif
```

Pascallite Stage 0 main() (/usr/local/4301/src/stage0main.C)

```
#include <stage0.h>

int main(int argc, char **argv)
{
  // This program is the stage0 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 Some Member Functions

```
Compiler(char **argv) // constructor
{
  open sourceFile using argv[1]
  open listingFile using argv[2]
  open objectFile using argv[3]
}

~Compiler() // destructor
{
  close all open files
}

void createListingHeader()
{
  print "STAGE0:", name(s), DATE, TIME OF DAY
  print "LINE NO:", "SOURCE STATEMENT"
    //line numbers and source statements should be aligned under the headings
}

void parser()
{
  nextChar()
  //ch must be initialized to the first character of the source file
  if (nextToken() != "program")
    processError(keyword "program" expected)
  //a call to nextToken() has two effects
  //  (1) the variable, token, is assigned the value of the next token
  //  (2) the next token is read from the source file in order to make
  //      the assignment.  The value returned by nextToken() is also
  //      the next token.
  prog()
  //parser implements the grammar rules, calling first rule
}

void createListingTrailer()
{
  print "COMPILATION TERMINATED", "# ERRORS ENCOUNTERED"
}

void processError(string err)
{
  Output err to listingFile
  Call exit(EXIT_FAILURE) to terminate program
}
```

Grammar Rules

prog() - production 1

```
void prog()  //token should be "program"
{
  if (token != "program")
    processError(keyword "program" expected)
  progStmt()
  if (token == "const")
    consts()
  if (token == "var")
    vars()
  if (token != "begin")
    processError(keyword "begin" expected)
  beginEndStmt()
  if (token != END_OF_FILE)
    processError(no text may follow "end”)
}
```

progStmt() - production 2

```
void progStmt()  //token should be "program"
{
  string x
  if (token != "program")
    processError(keyword "program" expected)
  x = nextToken()
  if (token is not a NON_KEY_ID)
    processError(program name expected)
  if (nextToken() != ";")
    processError(semicolon expected)
  nextToken()
  code("program", x)
  insert(x,PROG_NAME,CONSTANT,x,NO,0)
}
```

consts() - production 3

```
void consts()  //token should be "const"
{
  if (token != "const")
    processError(keyword "const" expected)
  if (nextToken() is not a NON_KEY_ID)
    processError(non-keyword identifier must follow "const")
  constStmts()
}
```

`vars()` - production 4

```
void vars()  //token should be "var"
{
  if (token != "var")
    processError(keyword "var" expected)
  if (nextToken() is not a NON_KEY_ID)
    processError(non-keyword identifier must follow "var")
  varStmts()
}
```

`beginEndStmt()` - production 5

```
void beginEndStmt()  //token should be "begin"
{
  if (token != "begin")
    procesError(keyword "begin" expected)
  if (nextToken() != "end")
    processError(keyword "end" expected)
  if (nextToken() != ".")
    processError(period expected)
  nextToken()
  code("end", ".")
}
```

constStmts() - production 6

```
void constStmts()  //token should be NON_KEY_ID
{
  string x,y
  if (token is not a NON_KEY_ID)
    processError(non-keyword identifier expected)
  x = token
  if (nextToken() != "=")
    processError("=" expected)
  y = nextToken()
  if (y is not one of "+","-","not",NON_KEY_ID,"true","false",INTEGER)
    processError(token to right of "=" illegal)
  if (y is one of "+","-")
  {
    if (nextToken() is not an INTEGER)
      processError(integer expected after sign)
    y = y + token;
  }
  if (y == "not")
  {
    if (nextToken() is not a BOOLEAN)
      processError(boolean expected after "not")
    if (token == "true")
      y = "false"
    else
      y = "true"
  }
  if (nextToken() != ";")
    processError(semicolon expected)
  if (the data type of y is not INTEGER or BOOLEAN)
    processError(data type of token on the right-hand side must be INTEGER or
                 BOOLEAN)
  insert(x,whichType(y),CONSTANT,whichValue(y),YES,1)
  x = nextToken()
  if (x is not one of "begin","var",NON_KEY_ID)
    processError(non-keyword identifier, "begin", or "var" expected)
  if (x is a NON_KEY_ID)
    constStmts()
}
```

varStmts() - production 7

```
void varStmts()  //token should be NON_KEY_ID
{
  string x,y
  if (token is not a NON_KEY_ID)
    processError(non-keyword identifier expected)
  x = ids()
  if (token != ":")
    processError(":" expected)
  if (nextToken() is not one of "integer","boolean")
    processError(illegal type follows ":")
  y = token
  if (nextToken() != ";")
    processError(semicolon expected)
  insert(x,y,VARIABLE,"",YES,1)
  if (nextToken() is not one of "begin",NON_KEY_ID)
    processError(non-keyword identifier or "begin" expected)
  if (token is a NON_KEY_ID)
    varStmts()
}
```

ids() - production 8

```
string ids()  //token should be NON_KEY_ID
{
  string temp,tempString
  if (token is not a NON_KEY_ID)
    processError(non-keyword identifier expected)
  tempString = token
  temp = token
  if (nextToken() == ",")
  {
    if (nextToken() is not a NON_KEY_ID)
      processError(non-keyword identifier expected)
    tempString = temp + "," + ids()
  }
  return tempString
}
```

Parser

Starting in `main()` in the parser, the action calls have been inserted into the productions. In the coding, the art of "defensive" programming is practiced. In particular, each parser routine expects the current token to be among a certain set of values when that routine is called. If the parser is performing properly (i.e., has no bugs), then each routine's input will be what it should be. However if there are any errors in the compiler, a routine could be called under improper conditions; e.g., `prog()` could be called with the current token something other than "`program`". Such an erroneous call could propagate errors indefinitely through any number of other routines until it were caught (if at all). Rather than assume the compiler is correct, you should presume it might very well have bugs and test whether each parser routine is being called under the right circumstances. If not, an error processing routine is called to handle the problem, otherwise, compilation continues unabated. The price paid for this additional check is the added cost to test the value of the current token against the set of expected tokens, a small price to pay during development of the additional error detection capability. If stage0 were installed as a working compiler, the compiler implementor could choose to remove these additional checks prior to installation if he felt the performance would be unduly limited by their inclusion.

Action Routines

insert()

`insert()` creates entries in the symbol table. It has six arguments:

1. a list of external names
2. the type of the list members
3. the mode of the list members
4. the value of the list members
5. whether or not storage will be emitted
6. the number of storage units to be emitted (if any)

Note that `insert()` calls `genInternalName()`, a function that has one argument, the type of the name being inserted. `genInternalName()` returns a unique internal name each time it is called, a name that is known to be a valid symbolic name. As a visual aid, we use different forms of internal names for each data-type of interest. The general form is:

`dn`

where `d` denotes the data-type of the name ("`I`" for *integer*, "`B`" for *boolean*) and `n` is a non-negative integer starting at `0`. The generated source code for `001.dat` clearly shows the effects of calling `genInternalName()`. The compiler itself will also need to generate names to appear in the object code, but since the compiler is defining these itself, there is no need to convert these names into any other form. The external and internal forms will be the same. The code for `insert()` treats any external name beginning with an uppercase character as defined by the compiler.

```
void insert(string externalName,storeType inType, modes inMode, string inValue,
            allocation inAlloc, int inUnits)
    //create symbol table entry for each identifier in list of external names
    //Multiply inserted names are illegal
{
  string name
  while (name broken from list of external names and put into name != "")
  {
    if (symbolTable[name] is defined)
      processError(multiple name definition)
    else if (name is a keyword)
      processError(illegal use of keyword)
    else  //create table entry
    {
      if (name begins with uppercase)
        symbolTable[name]=(name,inType,inMode,inValue,inAlloc,inUnits)
      else
        symbolTable[name]=(genInternalName(inType),inType,inMode,inValue,
                           inAlloc,inUnits)
    }
}
```

whichType(), whichValue()

```
storeTypes whichType(string name)        //tells which data type a name has
{
  if (name is a literal)
    if (name is a boolean literal)
      data type = BOOLEAN
    else
     data type = INTEGER
  else      //name is an identifier and hopefully a constant
    if (symbolTable[name] is defined)
      data type = type of symbolTable[name]
    else
      processError(reference to undefined constant)
  return data type
}

string whichValue(string name)        //tells which value a name has
{
  if (name is a literal)
    value = name
  else      //name is an identifier and hopefully a constant
    if (symbolTable[name] is defined and has a value)
      value = value of symbolTable[name]
    else
      processError(reference to undefined constant)
  return value
}
```

code()

```
void code(string op, string operand1, string operand2)
{
  if (op == "program")
    emitPrologue(operand1)
  else if (op == "end")
    emitEpilogue()
  else
    processError(compiler error since function code should not be called with
                 illegal arguments)
}
```

emit(), emitPrologue(), emitEpilogue(), emitStorage()

```
void emit(string label, string instruction, string operands, string comment)
{
  Turn on left justification in objectFile
  Output label in a field of width 8
  Output instruction in a field of width 8
  Output the operands in a field of width 24
  Output the comment
}

void emitPrologue(string progName, string operand2)
{
  Output identifying comments at beginning of objectFile
  Output the %INCLUDE directives
  emit("SECTION", ".text")
  emit("global", "_start", "", "; program " + progName)
  emit("_start:")
}

void emitEpilogue(string operand1, string operand2)
{
  emit("","Exit", "{0}")
  emitStorage();
}

void emitStorage()
{
  emit("SECTION", ".data")
  for those entries in the symbolTable that have
    an allocation of YES and a storage mode of CONSTANT
  { call emit to output a line to objectFile }
  emit("SECTION", ".bss")
  for those entries in the symbolTable that have
    an allocation of YES and a storage mode of VARIABLE
  { call emit to output a line to objectFile }
}
```

Lexical Scanner

The lexical scanner, `nextToken()`, of stage0 is referenced repeatedly in functions which define the parser. `nextToken()` is a function which always returns the next token; in addition, it always assigns the value it returns to the variable `token`, so that the value is easily referenced after the call is completed. The scanner itself calls a routine which returns characters to it, called `nextChar()`; `nextChar()` also assigns the value it returns to a variable for each referencing, `ch`. `nextChar()` can be used to print the listing file as well as returning the current character to `nextToken()`.

nextToken(), nextChar()

```
string nextToken()         //returns the next token or end of file marker
{
  token = "";
  while (token == "")
  {
    switch(ch)
    {
      case '{'               : //process comment
                               while (nextChar() is not one of END_OF_FILE, '}'}
                               { //empty body }
                               if (ch == END_OF_FILE)
                                 processError(unexpected end of file)
                               else
                                 nextChar()

      case '}'               : processError('}' cannot begin token)

      case isspace(ch)       : nextChar()

      case isSpecialSymbol(ch): token = ch
                                nextChar()

      case islower(ch)       : token = ch
                               while (nextChar() is one of letter, digit, or
                                      '_' but not END_OF_FILE)
                               {
                                 token+=ch
                               }
                               if (ch is END_OF_FILE)
                                 processError(unexpected end of file)

      case isdigit(ch)       : token = ch
                               while (nextChar() is digit but not END_OF_FILE)
                               {
                                 token+=ch
                               }
                               if (ch is END_OF_FILE)
                                 processError(unexpected end of file)

      case END_OF_FILE       : token = ch

      default                : processError(illegal symbol)
```

```
  }
  return token
}

char nextChar()   //returns the next character or end of file marker
{
  read in next character
  if end of file
    ch = END_OF_FILE     //use a special character to designate end of file
                         // END_OF_FILE is defined in stage0.h
  else
  {
    ch = next character
    print to listing file (starting new line if necessary)
  }
  return ch
}
```

Commands to compile, link, and run Stage 0

```
mmotl@csunix ~/4301> # Create a folder for stage0
mmotl@csunix ~/4301> mkdir stage0
mmotl@csunix ~/4301/stage0> cp /usr/local/4301/src/Makefile .
mmotl@csunix ~/4301/stage0> # Edit Makefile adding a target of
mmotl@csunix ~/4301/stage0> # stage0 to targets2srcfiles
mmotl@csunix ~/4301/stage0> cp /usr/local/4301/include/stage0.h .
mmotl@csunix ~/4301/stage0> cp /usr/local/4301/src/stage0main.C .
mmotl@csunix ~/4301/stage0> # Edit stage0.cpp
mmotl@csunix ~/4301/stage0> make stage0
g++ -g -Wall -std=c++11 -c stage0main.C -I/usr/local/4301/include/ -I.
g++ -g -Wall -std=c++11 -c stage0.cpp -I/usr/local/4301/include/ -I.
g++ -o stage0 stage0main.o stage0.o -L/usr/local/4301/lib/ -lm
mmotl@csunix ~/4301/stage0> # There are numerous data files in
mmotl@csunix ~/4301/stage0> # /usr/local/4301/data/stage0/
mmotl@csunix ~/4301/stage0> ls /usr/local/4301/data/stage0/
001.asm  004.asm  011.dat  018.lst  026.dat  030.asm  033.asm  036.asm  043.dat
001.dat  004.dat  012.dat  019.dat  026.lst  030.dat  033.dat  036.dat  044.dat
001.lst  004.lst  013.dat  020.dat  027.dat  030.lst  033.lst  036.lst  045.dat
002.asm  005.dat  014.dat  021.dat  028.asm  031.asm  034.asm  037.dat  046.dat
002.dat  006.dat  015.dat  022.dat  028.dat  031.dat  034.dat  038.dat  047.dat
002.lst  007.dat  016.dat  023.dat  028.lst  031.lst  034.lst  039.dat  048.dat
003.asm  008.dat  017.dat  024.dat  029.asm  032.asm  035.asm  040.dat  049.dat
003.dat  009.dat  018.asm  025.dat  029.dat  032.dat  035.dat  041.dat  050.dat
003.lst  010.dat  018.dat  026.asm  029.lst  032.lst  035.lst  042.dat  051.dat
mmotl@csunix ~/4301/stage0> # Copy as many or as few as you like
mmotl@csunix ~/4301/stage0> cp /usr/local/4301/data/stage0/001.dat .
mmotl@csunix ~/4301/stage0> cat 001.dat
program    stage0no001;   {here is a comment}
     const yes=true;no=false;
           small=0;smalleryet=-1;
           big = 1;  biggeryet = 2;  large = biggeryet;
           maybe = not true;
     var   some,many:integer;
           right, wrong : boolean;
     begin
     end.
mmotl@csunix ~/4301/stage0> # Execute your compiler on one of the
mmotl@csunix ~/4301/stage0> # datasets. The compiler is invoked with four
mmotl@csunix ~/4301/stage0> # command-line arguments. They are:
mmotl@csunix ~/4301/stage0> #   1) the executable of your compiler
mmotl@csunix ~/4301/stage0> #   2) the Pascallite source file
mmotl@csunix ~/4301/stage0> #   3) the listing file generated by your compiler
mmotl@csunix ~/4301/stage0> #   4) the object file (x86 assembly code)
mmotl@csunix ~/4301/stage0> #      generated by your compiler
mmotl@csunix ~/4301/stage0> ./stage0 001.dat 001.lst 001.asm
mmotl@csunix ~/4301/stage0> cat 001.lst
STAGE0:  YOUR NAME(S)       Mon Oct 19 17:19:53 2020

LINE NO.              SOURCE STATEMENT

    1|program    stage0no001;   {here is a comment}
    2|     const yes=true;no=false;
    3|           small=0;smalleryet=-1;
    4|           big = 1;  biggeryet = 2;  large = biggeryet;
    5|           maybe = not true;
    6|     var   some,many:integer;
```

```
    7|           right, wrong : boolean;
    8|     begin
    9|     end.

COMPILATION TERMINATED      0 ERRORS ENCOUNTERED
mmotl@csunix ~/4301/stage0> cat 001.asm
; YOUR NAME(S)       Mon Oct 19 17:19:53 2020
%INCLUDE "Along32.inc"
%INCLUDE "Macros_Along.inc"

SECTION .text
global  _start                          ; program stage0no001

_start:
        Exit    {0}

SECTION .data
I2      dd      1                       ; big
I3      dd      2                       ; biggeryet
I4      dd      2                       ; large
B2      dd      0                       ; maybe
B1      dd      0                       ; no
I0      dd      0                       ; small
I1      dd      -1                      ; smalleryet
B0      dd      -1                      ; yes

SECTION .bss
I6      resd    1                       ; many
B3      resd    1                       ; right
I5      resd    1                       ; some
B4      resd    1                       ; wrong
mmotl@csunix ~/4301/stage0> # Edit the Makefile to add a target
mmotl@csunix ~/4301/stage0> # of 001 (or any other dataset) to
mmotl@csunix ~/4301/stage0> # targetsAsmLanguage
mmotl@csunix ~/4301/stage0> make 001
nasm -f elf32  -o 001.o 001.asm -I/usr/local/4301/include/ -I.
ld -m elf_i386 --dynamic-linker /lib/ld-linux.so.2 -o 001 001.o \
/usr/local/4301/src/Along32.o -lc
mmotl@csunix ~/4301/stage0> ls 001*
001  001.asm  001.dat  001.lst  001.o
mmotl@csunix ~/4301/stage0> # Note that 001.asm assembled and linked
mmotl@csunix ~/4301/stage0> # with no errors
mmotl@csunix ~/4301/stage0> # Execute ./001 to ensure it runs without
mmotl@csunix ~/4301/stage0> # errors
mmotl@csunix ~/4301/stage0> ./001
mmotl@csunix ~/4301/stage0> # You can diff the .asm and .lst files
mmotl@csunix ~/4301/stage0> diff /usr/local/4301/data/stage0/001.lst 001.lst
1c1
< STAGE0:  YOUR NAME(S)       Mon Oct 19 17:06:18 2020
---
> STAGE0:  YOUR NAME(S)       Mon Oct 19 17:19:53 2020
mmotl@csunix ~/4301/stage0> diff /usr/local/4301/data/stage0/001.asm 001.asm
1c1
< ; YOUR NAME(S)       Mon Oct 19 17:06:18 2020
---
> ; YOUR NAME(S)       Mon Oct 19 17:19:53 2020
mmotl@csunix ~/4301/stage0> # dataset 001 looks like a success!
mmotl@csunix ~/4301/stage0> # 50 more datasets to go!
mmotl@csunix ~/4301/stage0>
```