SPAWK

Project Report

Tad Kellogg

uni: twk2113

COMS W4115 CVN Fall 2015

1. Introduction

The SPAWK language is a compiler that takes as program code source an AWK inspired syntax and produces a as output a target program for execution as a Spark program written in the Python + Spark API. Apache Spark is one of the newer infrastructure as a software systems to implement parallel processing on commodity nodes. The Spark system, along with either Meos or Hadoop for multi-node execution, implements parallel algorithms that can align within the paradigm of a directed acyclic graph programming pattern. Spark uses resilient the distributed dataset (RDD) as the primary data store concept, as such all Spark programs result in a set of transformation and action expressions applied to RDDs. Inspiration for the SPAWK language comes from the Hadoop based interpreter Pig and the Pig Latin language, where pig latin programs are translated into a set of Map/Reduce java programs for execution in the Hadoop computing platform. Just as the AWK language was able to bring programming simplification for the areas of data extraction, manipulation and reporting, so too will SPAWK simplify the programming required for similar operations using the Spark cluster computing platform for Big Data. In more broad terms, SPAWK could be used as the sole constructor or a component constructor for Big Data Extract Transform Load (ETL) systems.

Language Highlights

Just as the AWK language was able to bring programming simplification for the areas of data extraction, manipulation and reporting, so too will Spawk simplify the programming required for similar operations using the Spark cluster computing platform for Big Data. In more broad terms, Spawk could be used as the sole constructor or a component constructor for Big Data Extract Transform Load (ETL) systems.

As in the case of the Hadoop Pig interpreter where many sequences of Map/Reduce jobs are produced for execution of a single pig latin program, the Spawk compiler may have to produce a sequence of Spark programs in order to fully accomplish the desired intent expressed in a single Spawk language program.

2. Language Tutorial

The SPAWK language requires a system with Spark version 1.3 or higher and python 2.7 or higher installed (python is a requirement for Spark). One can obtain Spark here: http://spark.apache.org

The SPAWK compiler , spawk.native, requires two command line arguments:

```
spawk.native <program file> <data file>
```

1. <program file > The filename for the file containing the user created program following the SPAWK syntax.

2. <data file> The filename of the data file for processing as an RDD dataset in the Spark environment.

example: from the unix command shell prompt:

```
./spawk.native gcd.spawk ../wordcount_test.txt
```

The compiler then produces three output files:

1. The <program file>.py , this is the python program suitable for execution within the spark system, for this initial version of SPAWK, the compiler pre configures the python execution for a local spark system and sets the spark application name to "spawk", for example:

```
conf = (SparkConf()
     .setMaster("local")
     .setAppName("spawk")
     .set("spark.executor.memory", "1g"))

sc = SparkContext(conf = conf)
```

Future versions should incorporate command line options for alternative spark runtime environment settings.

2. The <program file>.pyc is a product of running the produced python code through the py_compile library with the python command. This causes python to check for syntax errors and they are reported in the following final production file.

3. The <program file>.warn contains a printout of the AST version of the SPAWK source code, any warning messages from the SAST checker of the AST version and any syntax warnings from the python compiler.

To execute the SPAWK produced python code one must use a spark execution command, as of version 1.4 the recommended command is the spark-submit command:

spark-submit <progam file>.py

3. Language Manual

3.1. Lexical Conventions

3.2. Tokens

SPAWK has six types of tokens: identifiers, keywords, constants, operators, separators and white space.

Token definitions:

```
 | '='    { ASSIGN }
 | '+'    { PLUS }
 | '-'    { MINUS }
 | '*'    { TIMES }
 | ','    { COMMA }
 | ';'    { SEMICOLON }
 | '('    { LEFTPAREN }
 | ')'    { RIGHTPAREN }
 | '{'    { LEFTBRACE }
 | '}'    { RIGHTBRACE }
 | '['    { LEFTBRACK }
 | ']'    { RIGHTBRACK }
 | '/'    { DIVIDE }
 | '%'    { MOD }
 | '$'    { DOLLAR }
 | "#"    { comment lexbuf }          (* Comments *)
 | '"'    { read_string (Buffer.create 17) lexbuf}
 | "=="   { EQ }
 | "!="   { NEQ }
 | '<'    { LT }
```

```
| "<="    { LEQ }
| ">"     { GT }
| ">="    { GEQ }
| "if"    { IF }
| "else"  { ELSE }
| "for"   { FOR }
| "in"    { IN }
| "while" { WHILE }
| "return" { RETURN }
| "mapReduceByKey" { MRBK }
| "emptyPattern" { EP }
| "var"   { VAR }
| iden as i {
   (* try keywords if not found then it's identifier *)
   let l = String.lowercase i in
   try List.assoc l keywords
   with Not_found -> IDENTIFIER i
}
| number as d {
   (* parse number *)
   NUMBER  d
}
| '\n'    { incr line_num; micro lexbuf } (* counting new line characters *)
| blank   { micro lexbuf } (* skipping blank characters *)
| _       { syntax_error "couldn't identify the token" }
| eof     { EOF } (* no more tokens *)
and read_string buf =
   parse
   | '"'     { STRING (Buffer.contents buf) }
   | '\\' '/'  { Buffer.add_char buf '/'; read_string buf lexbuf }
   | '\\' '\\' { Buffer.add_char buf '\\'; read_string buf lexbuf }
```

```
        | '\\' 'b'  { Buffer.add_char buf '\b'; read_string buf lexbuf }
        | '\\' 'f'  { Buffer.add_char buf '\012'; read_string buf lexbuf }
        | '\\' 'n'  { Buffer.add_char buf '\n'; read_string buf lexbuf }
        | '\\' 'r'  { Buffer.add_char buf '\r'; read_string buf lexbuf }
        | '\\' 't'  { Buffer.add_char buf '\t'; read_string buf lexbuf }
        | [^ '"' '\\']+
          { Buffer.add_string buf (Lexing.lexeme lexbuf);
              read_string buf lexbuf
           }
        | _ { raise (Syntax_error ("Illegal string character: " ^ Lexing.lexeme
lexbuf)) }
        | eof { raise (Syntax_error ("String is not terminated")) }


    and comment = parse
    '\n' { micro lexbuf }
    | _    { comment lexbuf }
```

3.3.Comments

SPAWK uses the comment convention of a line starting with the '#' character and ending with a newline character. e.g.

# a comment line

3.4.Identifiers

In SPAWK in identifier is the name of a variable, array, built-in function or user-defined function. The name of an identifier is composed of a sequence of alphanumeric characters, starting with a letter (A-Za-z0-9, case-sensitive). The identifier name can not match a keyword name.

3.5.Keywords

The following keywords and names of built-in functions are considered reserved words:

BEGIN break continue do else END for function if in print while

index length split map reduceByKey

3.6.Constants

3.6.1. SPAWK has four types of constants: numeric, string, regular expression and pattern statements.

3.6.2.Numbers

Numbers can be an integer, a decimal fraction, or a number in scientific (exponential) notation. Some examples of numeric constants, which all have the same value:

205

2.05e+2

2050e-1

```
let digit = ['0'-'9']
let int = '-'? digit digit*
let digits = digit*
let frac = '.' digit*
let exp = ['e' 'E']['-' '+']? digit+
let float = digit* frac? exp?
let number = (int | float)
```

3.6.3.Strings

A string constant consists of a sequence of characters enclosed in double-quote marks. e.g.:

"Ocaml"

represents the string whose contents are ‘Ocaml’.

Newlines “\n” are allowed and contained within the string constant, enabling paragraphs for string constants.

One uses backslash as an escape sequence to include special character in a string constant.:

\”

Represents a literal double-quote, “”.

\\

Represents a literal backslash, ‘\’.

\a

Represents the "alert" character, control-g, ASCII code 7.

\b

Represents a backspace, control-h, ASCII code 8.

\f

Represents a formfeed, control-l, ASCII code 12.

\n

Represents a newline, control-j, ASCII code 10.

\r

Represents a carriage return, control-m, ASCII code 13.

\t

Represents a horizontal tab, control-i, ASCII code 9.

\v

Represents a vertical tab, control-k, ASCII code 11.

\nnn

Represents the octal value nnn, where nnn are one to three digits between 0 and 7. See unix ascii man page for octal codes.

\xhh...

Represents the hexadecimal value hh, where hh are hexadecimal digits (`0' through `9' and either `A' through `F' or `a' through `f').

let blank = [' ' '\r' '\t']

3.6.4.Regular Expressions

A regular expression is enclosed in slashes prior to the action statement as the pattern statement, such as

/^start and end of line$/ { action }

Python Regular expression syntax is supported, as this is a pass through string.

character : meaning

———————————————

. :Any character except newline

a :The character a

ab :The string ab

a|b :a or b

a* :0 or more a's

\ :Escapes a special character

* :0 or more

+ :1 or more

? :0 or 1

(…) : Capturing group

(?P<Y>…) :Capturing group named Y

(?:…) :Non-capturing group

\Y :Match the Y'th captured group

(?P=Y) :Match the named group Y

(?#…) :Comment

^ :Start of string

\A :Start of string, ignores m flag

$:End of string

\Z :End of string, ignores m flag

\b : Word boundary

\B : Non-word boundary

(?=…) :Positive lookahead

(?!…) :Negative lookahead

(?<=…) :Positive lookbehind

(?<!…) :Negative lookbehind

(?()|) :Conditional

3.6.5. Operators

The operators in SPAWK, in order of increasing precedence, are:

= += -= *= /= %= ^=

Assignment. Both absolute assignment (var=value) and operator assignment are supported.

||

Logical "or".

&&

Logical "and".

< <= > >= != ==

Relational operators.

blank

String concatenation.

+ -

Addition and subtraction.

* / %

Multiplication, division, and modulus.

+ - !

Unary plus, unary minus, and logical negation.

^

Exponentiation.

++ --

Increment and decrement, both prefix and postfix.

$

Field reference.

( )

Grouping

{ }

Action statements, where action is a sequence of statements

3.7. Field separators.

The field separator, which is a single character, controls the way SPAWK splits an input record into fields. SPAWK scans the input record for character sequences that match the separator; the fields themselves are the text between the matches.

3.8. White space

Blanks used as default field separator character.

horizontal and vertical tabs, blanks are ignored except as they separate tokens. Blank space is required to separate otherwise adjacent identifiers, keywords, and constants.

let blank = [' ' '\r' '\t']

3.9. Variables

3.10. User defined

A variable name is a valid expression by itself; it represents the variable's current value. Variables are given new values with assignment operators, increment operators and decrement operators.

Variables in SPAWK can be assigned either numeric, string values or spark RDD pointers. Resilient Distributed Dataset (RDD), which is a fault-tolerant collection of elements that can be operated on in parallel in the spark environment. By default, variables are initialized to the empty string, which is zero if converted to a number. Pre-initialize of each variable explicitly in SPAWK is not needed.

3.11. Statements

3.12. Action Statements

Most often, each line in an SPAWK program is a separate statement or separate rule, like this:

```
/42/  { print $0 }
```

To split a single statement into two lines at a point where a newline would terminate it, you can continue it by ending the first line with a backslash character (‘\’):

```
{ \
 print \\
      "hello, world" \
 }
```

3.13.Selection Statements

If statement is used to check the conditions, if the condition returns true, it performs its corresponding action(s), where {} are needed:

```
if (conditional-expression)
{
        action1;
        action2;
}
```

If Else statement you can give the list of action to perform if the condition is false:

```
if (conditional-expression)
        action1
else
        action2
```

3.14.Iteration Statements

While statement is the simplest looping statement. It repeatedly executes a statement as long as a condition is true. It looks like this:

```
while (condition)
  {
        action1;
        action2;
  }
```

The do loop executes the action statement once, and then repeats action statement as long as condition is true. It looks like this:

```
do
  {
        action1;
        action2;
}
while (condition)
```

The for statement looks like this:

```
for (initialization; condition; increment)
  {
        action statements
 }
```

The initialization, condition and increment parts are arbitrary expressions.

The for statement starts by executing initialization. Then, as long as condition is true, it repeatedly executes action block and then increment.

The for in loop, for iterating over all the key,value pairs of a spark key value RDD:

for (k,v) in RDD action statement for RDD

3.15.Functions

3.16.Built-in or pass through to python

print or print(item1,item2,..)

The print function does output with simple, standardized formatting. You specify only the strings or numbers to be printed, in a list separated by commas. They are output, separated by single spaces, followed by a newline. The statement looks like this:

print item1, item2, ...

The entire list of items may optionally be enclosed in parentheses. The items to be printed can be constant strings or numbers, fields of the current record (such as $1), variables, or any expressions. Numeric values are converted to strings, and then printed.

index(s, t)

Return the position, in characters, numbering from 1, in string s where string t first occurs, or zero if it does not occur at all.

length[([s])]

Return the length, in characters, of its argument taken as a string, or of the whole record, $0, if there is no argument.

split(s, a[, fs])

Split the string s into array elements a[1], a[2], ..., a[n], and return n. All elements of the array shall be deleted before the split is performed. The separation shall be done with fs or with the field separator FS if fs is not given. Each array element shall have a string value when created and, if appropriate, the array element shall be considered a numeric string.

mapReduceByKey(mapper,reducer)

Merge the values for each key, using an associative mapper and reducer functions . This is a call to python spark API and returns an RDD of collected key /value pairs.

3.17.User-defined

functions can be defined as:

function name([parameter, ...]) { statements }

A function can be referred to anywhere in an SPAWK program; in particular, its use can precede its definition. The scope of a function is global.

Function parameters, if present, can be either scalars or arrays; the behavior is undefined if an array name is passed as a parameter that the function uses as a scalar, or if a scalar expression is passed as a parameter that the function uses as an array. Function parameters shall be passed by value if scalar and by reference if array name.

3.18.Scope

3.19.User-defined Functions

In SPAWK, there is no way to make a variable local to a { … } block/compound statement. However, one can make a variable local to a user-defined function.

To make a variable local to a function, simply declare the variable as an argument after the actual function arguments (see Definition Syntax). Look at the following example, where variable i is a global variable used by both functions foo() and bar():

```
function bar()
{
    for (i = 0; i < 3; i++)
        print "bar's i=" i
}
```

3.20.GAMMAR

SPAWK programs take the form:

pattern { action }

structure with:

where pattern is /regular expression/ or “understood empty condition for match every line”.

SPAWK supports semantic type checking for mathematical operations and the for in loop for processing key/value pairs in a spark RDD.

The grammar is given in the ocamlyacc syntax style, starting with the program definition as either end of file or a series of pattern action statements.

```
program:
EOF { [] }
| padecl program {  $1 :: $2 }
```

```
padecl:
LEFTBRACE stmt_list RIGHTBRACE { { pattern = "emptyPattern" ; actions = List.rev $2 } }
| DIVIDE IDENTIFIER DIVIDE LEFTBRACE stmt_list RIGHTBRACE { { pattern = $2 ; actions = List.rev $5 } }
```

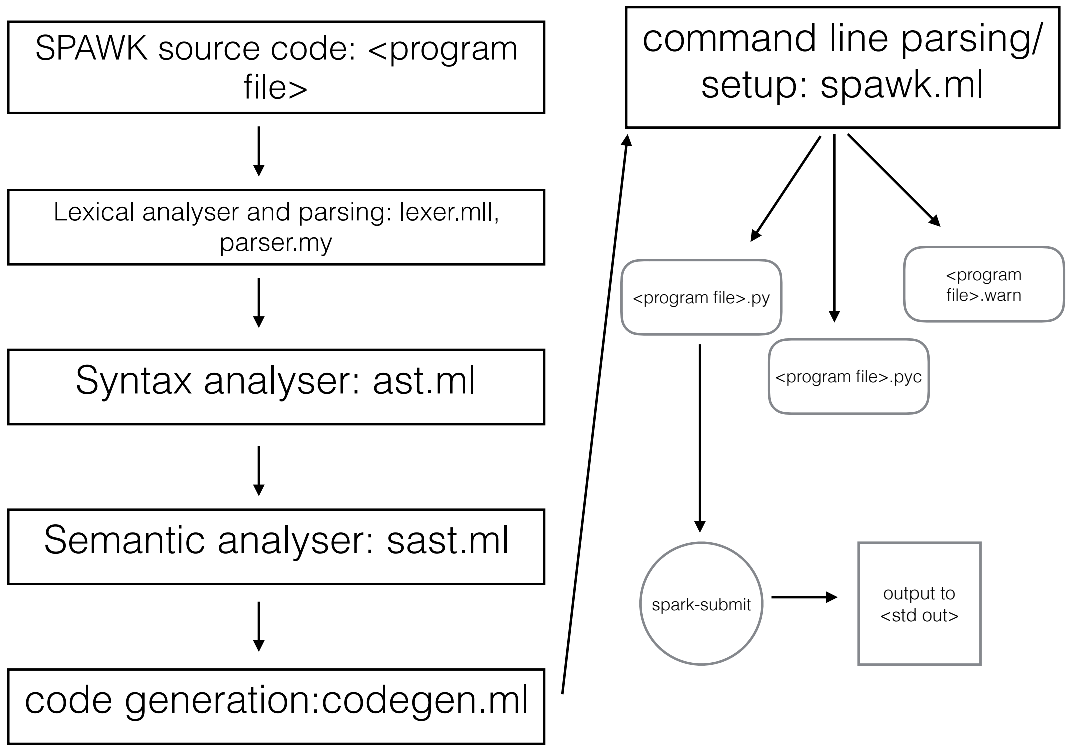
```
formals_opt:
    /* nothing */ { [] }
  | formal_list   { List.rev $1 }

formal_list:
    IDENTIFIER                  { [$1] }
  | formal_list COMMA IDENTIFIER { $3 :: $1 }



stmt_list:
    /* nothing */  { [] }
  | stmt_list stmt { $2 :: $1 }

stmt:
    expr  { Expr($1) }
  | RETURN expr  { Return($2) }
  | LEFTBRACE stmt_list RIGHTBRACE { Block(List.rev $2) }
  | IF LEFTPAREN expr RIGHTPAREN stmt %prec NOELSE { If($3, $5, Block([])) }
  | IF LEFTPAREN expr RIGHTPAREN stmt ELSE stmt    { If($3, $5, $7) }
  | FOR LEFTPAREN expr_opt SEMICOLON expr_opt SEMICOLON expr_opt
RIGHTPAREN stmt
     { For($3, $5, $7, $9) }
  | FOR LEFTPAREN IDENTIFIER COMMA IDENTIFIER RIGHTPAREN IN
IDENTIFIER stmt { ForIn($3,$5,$8,$9) }
  | WHILE LEFTPAREN expr RIGHTPAREN stmt { While($3, $5) }
  | FUNCTION IDENTIFIER LEFTPAREN actuals_opt RIGHTPAREN LEFTBRACE
stmt_list RIGHTBRACE  { Def($2,$4,List.rev $7)}
  | IDENTIFIER ASSIGN MRBK LEFTPAREN actuals_opt RIGHTPAREN { Mrbk($1,$5) }



expr_opt:
   /* nothing */ { Noexpr }
  | expr         { $1 }

expr:
   NUMBER          { Number($1) }
  | STRING         { String($1) }
  | IDENTIFIER              { Id($1) }
  | expr PLUS   expr { Binop($1, Add,   $3) }
  | expr MINUS  expr { Binop($1, Sub,   $3) }
  | expr TIMES  expr { Binop($1, Mult,  $3) }
  | expr DIVIDE expr { Binop($1, Div,   $3) }
  | expr EQ     expr { Binop($1, Equal, $3) }
  | expr NEQ    expr { Binop($1, Neq,   $3) }
  | expr LT     expr { Binop($1, Less,  $3) }
  | expr LEQ    expr { Binop($1, Leq,   $3) }
  | expr GT     expr { Binop($1, Greater,  $3) }
  | expr GEQ    expr { Binop($1, Geq,   $3) }
  | expr MOD    expr { Binop($1, Mod,   $3) }
  | IDENTIFIER ASSIGN expr   { Assign($1, $3) }
  | IDENTIFIER LEFTBRACK expr RIGHTBRACK { Array($1,$3) }
  | IDENTIFIER LEFTPAREN actuals_opt RIGHTPAREN { Call($1, $3) }
  | LEFTPAREN expr COMMA expr RIGHTPAREN { KeyValue($2,$4) }
  | LEFTPAREN expr RIGHTPAREN { $2 }

actuals_opt:
   /* nothing */ { [] }
  | actuals_list  { List.rev $1 }

actuals_list:
   expr                  { [$1] }
  | actuals_list COMMA expr { $3 :: $1 }
```

4. Project Plan (CVN)

Software development environment

- OS X 10.11.2
- git for version control, find files here: https://github.com/twk123/SPAWK.git
- dropbox for backup
- sublime text editor, supports OCaml syntax highlighting.
- python 2.7
- ocaml 4.02.1
- spark 1.3

5. Architectural Design


SPAWK source code: <program file>
Lexical analyser and parsing: lexer.mll, parser.my
Syntax analyser: ast.ml
Semantic analyser: sast.ml
code generation:codegen.ml
command line parsing/ setup: spawk.ml
<program file>.py
<program file>.pyc
<program file>.warn
spark-submit
output to <std out>

6. Test Plan

1. Test of word count with simple pattern match for each line, tests case of program with pattern then action of spark map reduce processing.

Input source program: wordcount.spawk

```
——————————————————————————
/RDD/ {


function mapper(x) { return (x,1) }
function reducer(x,y) { return (x+y) }


A = mapReduceByKey(mapper,reducer)


for (k,v) in A print(k,v)


 }
——————————————————————————-
```

Output target program: wordcount.py

```
————————————————————————-
from __future__ import print_function
import sys
import re
from operator import add
from pyspark import SparkConf, SparkContext
dataFile = "../wordcount_test.txt"  # Should be some file on your system
conf = (SparkConf()
         .setMaster("local")
         .setAppName("spawk")
         .set("spark.executor.memory", "1g"))

sc = SparkContext(conf = conf)
```

```
spawkTempData = sc.textFile(dataFile).cache()
def parse_rrd_line(line):
    match = re.search('RDD',line)
    if match is None:
        return ''
    return line

spawkData = (spawkTempData.map(parse_rrd_line).cache())



def mapper(x):
    return (x,1)

def reducer(x, y):
    return x + y

flatMapStep = spawkData.flatMap(lambda x: x.split(' '))
mapStep = flatMapStep.map(mapper)
reduceStep = mapStep.reduceByKey(reducer)
A = reduceStep.collect()
for (k,v) in A:
   print(k, v)



sc.stop()
```

---

2. Test of word count program with no pattern, match all line by default, tests default no pattern case of language with spark execution for map reduce processing.

Input source program: wordcount_nopat.spawk

———————————————————————————-

```
 {


function mapper(x) { return (x,1) }
function reducer(x,y) { return (x+y) }

A = mapReduceByKey(mapper,reducer)

for (k,v) in A print(k,v)

 }
```

———————————————————————————-

source target output: wordcount_nopat.py

———————————————————————————

```
from __future__ import print_function
import sys
import re
from operator import add
from pyspark import SparkConf, SparkContext
dataFile = "../wordcount_test.txt"  # Should be some file on your system
conf = (SparkConf()
         .setMaster("local")
         .setAppName("spawk")
         .set("spark.executor.memory", "1g"))

sc = SparkContext(conf = conf)




spawkData = sc.textFile(dataFile).cache()
```

```
def mapper(x):
    return (x,1)

def reducer(x, y):
    return x + y

flatMapStep = spawkData.flatMap(lambda x: x.split(' '))
mapStep = flatMapStep.map(mapper)
reduceStep = mapStep.reduceByKey(reducer)
A = reduceStep.collect()
for (k,v) in A:
   print(k, v)



sc.stop()
```

---

3. Test of gcd algorithm, note current SPAWK system requires a data file, but is not used in this test, this is a test of algorithm code only.

Input source code: gcd.spawk

```
{
 function gcd(x, y) {
    while (y != 0) {
    temp = x
            x = y
            y = temp % y
    }

    return (x)
    }

    print(gcd(40,56))
 }
```

---

target source program: gcd.py

```
from __future__ import print_function
import sys
import re
from operator import add
from pyspark import SparkConf, SparkContext
dataFile = "../wordcount_test.txt"  # Should be some file on your system
conf = (SparkConf()
         .setMaster("local")
         .setAppName("spawk")
         .set("spark.executor.memory", "1g"))

sc = SparkContext(conf = conf)



spawkData = sc.textFile(dataFile).cache()



def gcd(x, y):
    while (y != 0):
        temp = x ; x = y ; y = temp % y
    return x

print(gcd(40, 56))
```

```
sc.stop()
```

————————————————————————-

Shell scripting was used to automate the OCaml project make (using ocamlbuild: ocamlbuild -use-menhir -yaccflag --trace -use-ocamlfind spawk.native) and test compile/ spark execution with known results from previously authored python + spark API code.

test_suite.sh:

```
#!/bin/sh
# run all tests

sh test_gcd.sh
sh test_wordcount_nopat.sh
sh test_wordcount_pat.sh
```

————————————————-

test_gcd.sh:

```
# script to compile and test spawk with

TEST="gcd program"

SignalError() {
    if  [ $error -eq 0 ] ; then
            echo "FAILED"
            error=1
    fi
    echo "  $1"
}

# Compare <outfile> <reffile> <difffile>
# Compares the outfile with reffile.  Differences, if any, written to difffile
Compare() {
   #generatedfiles="$generatedfiles $3"
   echo diff -b $1 $2 ">" $3 1>&2
   diff -b "$1" "$2" > "$3" 2>&1 || {
         SignalError "$1 differs"
         echo "FAILED $1 differs from $2" 1>&2
         exit
   }
   echo "Test PASS for $TEST"
}

# run the control python script through spark , sort needed for random reducer order
from spark parallel system
python test_gcd.py 2> /dev/null | sort > gcd_test.out

# compile and run generated python through spark
rm -f spawk.native
ocamlbuild -use-menhir -yaccflag --trace -use-ocamlfind spawk.native
rm -f gcd.py
./spawk.native gcd.spawk ../wordcount_test.txt > /dev/null 2>&1
rm -f spawk_output.out
spark-submit gcd.py 2> /dev/null | sort > spawk_output.out

Compare spawk_output.out gcd_test.out diffFile
```

————————————————————-

```
test_wordcount_nopat.sh:
#!/bin/sh
# script to compile and test spawk with
```

```
TEST="no pattern word count program"

SignalError() {
  if [ $error -eq 0 ] ; then
        echo "FAILED"
        error=1
  fi
  echo "  $1"
}

# Compare <outfile> <reffile> <difffile>
# Compares the outfile with reffile.  Differences, if any, written to difffile
Compare() {
  #generatedfiles="$generatedfiles $3"
  echo diff -b $1 $2 ">" $3 1>&2
  diff -b "$1" "$2" > "$3" 2>&1 || {
        SignalError "$1 differs"
        echo "FAILED $1 differs from $2" 1>&2
        exit
  }
  echo "Test PASS for $TEST"
}

# run the control python script through spark , sort needed for random reducer order
from spark parallel system
spark-submit test_nopat_wc.py ../wordcount_test.txt 2> /dev/null | sort >
wordcount_nopat_test.out

# compile and run generated python through spark
rm -f spawk.native
rm -f wordcount_nopat.py
```

```
ocamlbuild -use-menhir -yaccflag --trace -use-ocamlfind spawk.native
./spawk.native wordcount_nopat.spawk ../wordcount_test.txt > /dev/null 2>&1
spark-submit wordcount_nopat.py 2> /dev/null | sort > spawk_output.out

Compare spawk_output.out wordcount_nopat_test.out diffFile
```

```
test_wordcount_pat.sh:
#!/bin/sh
# script to compile and test spawk with

TEST="pattern word count program"

SignalError() {
   if [ $error -eq 0 ] ; then
	echo "FAILED"
	error=1
   fi
   echo "  $1"
}

# Compare <outfile> <reffile> <difffile>
# Compares the outfile with reffile.  Differences, if any, written to difffile
Compare() {
   #generatedfiles="$generatedfiles $3"
   echo diff -b $1 $2 ">" $3 1>&2
   diff -b "$1" "$2" > "$3" 2>&1 || {
	SignalError "$1 differs"
	echo "FAILED $1 differs from $2" 1>&2
	exit
   }
   echo "Test PASS for $TEST"
}

# run the control python script through spark , sort needed for random reducer order from spark parallel system
spark-submit test_pat_wc.py ../wordcount_test.txt 2> /dev/null | sort > wordcount_pat_test.out

# compile and run generated python through spark
rm -f spawk.native
ocamlbuild -use-menhir -yaccflag --trace -use-ocamlfind spawk.native
rm -f spawk_output.out
rm -f wordcount.py
./spawk.native wordcount.spawk ../wordcount_test.txt > spawk_output.out 2> /dev/null
spark-submit wordcount.py 2> /dev/null | sort > spawk_output.out

Compare spawk_output.out wordcount_pat_test.out diffFile
——————————————————————————-
```

The above test automation will produce a spawk_output.out as the runtime execution output from the SPAWK generated python+spark API code. The comparison with previous known output is compared and reported as "Test PASS …" is successful.

```
diatomite:spawk_first tad$ sh test_suite.sh
Finished, 23 targets (23 cached) in 00:00:00.
diff -b spawk_output.out gcd_test.out > diffFile
Test PASS for gcd program
Finished, 23 targets (23 cached) in 00:00:00.
diff -b spawk_output.out wordcount_nopat_test.out > diffFile
Test PASS for no pattern word count program
Finished, 23 targets (23 cached) in 00:00:00.
diff -b spawk_output.out wordcount_pat_test.out > diffFile
Test PASS for pattern word count program
```

```
diatomite:spawk_first tad$
cat spawk_output.out
 13
(RDD) 1
+ 1
API 1
API. 1
AWK 1
Apache 1
Big 1
Data. 1
Hadoop 3
Inspiration 1
Java 1
Latin 1
Map/Reduce 1
Meos 1
One 1
Pig 2
Python 2
RDDs. 1
Scala, 1
Spark 7
Spark. 1
Spawk 1
The 3
a 8
```

7. Lessons Learned

New appreciation for compiler design and implementation. Not sure I have had enough of the OCaml cool-aid to be a believer. One major problem with implementing a translator to another language plus specialized system APIs is the resulting two sets of warnings and error messages, three if you count OCaml compiling. I have encountered this problem before with languages such as Hive and Pig that translate to java map/reduce code for Hadoop. I appreciate the python warnings and error messages over java. This type of translator system seems to be enjoying popularity at this time, translators to javascript even. Is this a specialized case of the computer science problem solving rule of just one more indirection?

For future students, if not an familiar with OCaml do not try to convert existing code to fit your project, I spent weeks of coding and then starting over ….

8. References

8.1. Kernighan, Brian W.; Ritchie, Dennis (1988-03-22). C Programming Language (p. 191). Pearson Education.

8.2. Karau, H., & Konwinski, A. (2015). Learning Spark. Databricks: O'Reilly Media, Inc.

8.3. Aho, A., & Kernighan, B. (1988). The AWK programming language. Reading, Mass.: Addison-Wesley Pub.

8.4. Apache Spark™ - Lightning-Fast Cluster Computing. (n.d.). Retrieved 2015, from http://spark.apache.org/

8.5. Yadav, R. (2015). Spark cookbook: Over 60 recipes on Spark, covering Spark Core, Spark SQL, Spark Streaming, MLib, and GraphX libraries. Packt Publishing Ltd.

8.6. Robbins, A. (2003). GAWK: Effective AWK programming : A user's guide for GNU Awk (Ed. 3. ed.). Boston, MA: Free Software Foundation.

8.7. Minsky, Y., & Madhavapeddy, A. (2013). Real world OCaml. Sebastopol, CA: O'Reilly Media.

9. Appendix

(CVN all source code by author) https://github.com/twk123/SPAWK.git

```
==> ast.ml <==
type op = Add | Sub | Mult | Div | Equal | Neq | Less | Leq | Greater | Geq | Mod
type expr =
    Number of string
  | String of string
  | Id of string
  | Binop of expr * op * expr
  | Assign of string * expr
  | Call of string * expr list
  | PTuple of expr * expr list
  | Array of string * expr
  | KeyValue of expr * expr
  | Noexpr


type stmt =
    Block of stmt list
  | Expr of expr
  | Return of expr
  | If of expr * stmt * stmt
  | For of expr * expr * expr * stmt
  | ForIn of string * string * string * stmt
  | While of expr * stmt
  | Def of string * expr list * stmt list
  | Mrbk of string * expr list

type pat_act = {
     pattern : string;
     actions : stmt list;
}


type func_decl = {
    fname : string;
    formals : string list;
    body : stmt list;
  }

type program = pat_act list

let rec last = function
    | [] -> None
    | [x] -> Some x
    | _ :: t -> last t

let rec at k = function
    | [] -> None
    | h :: t -> if k = 1 then Some h else at (k-1) t

let rec string_of_expr = function
    String(s) -> s
  | Number(n) -> n
  | Id(s) -> s
  | Binop(e1, o, e2) ->
      string_of_expr e1 ^ " " ^
      (match o with
            Add -> "+" | Sub -> "-" | Mult -> "*" | Div -> "/"
       | Equal -> "==" | Neq -> "!="
       | Less -> "<" | Leq -> "<=" | Greater -> ">" | Geq -> ">=" | Mod -> "%") ^ " " ^
      string_of_expr e2
  | Assign(v, e) -> v ^ " = " ^ string_of_expr e
  | Array(v,e) -> v ^ "[" ^ string_of_expr e ^ "]"
  | PTuple(v, e) -> "(" ^ string_of_expr v ^ ", " ^ String.concat ", " (List.map string_of_expr e) ^ ")"
  | Call(f, el) ->
```

```
      f ^ "(" ^ String.concat ", " (List.map string_of_expr el) ^ ")"
  | KeyValue(e1,e2) -> "(" ^ string_of_expr e1 ^ "," ^ string_of_expr e2 ^ ")"
  | Noexpr -> ""

let rec string_of_stmt = function
    Block(stmts) ->
      "{\n" ^ String.concat "" (List.map string_of_stmt stmts) ^ "}\n"
  | Expr(expr) -> string_of_expr expr ^ ";\n";
  | Return(expr) -> "return " ^ string_of_expr expr ^ ";\n";
  | If(e, s, Block([])) -> "if (" ^ string_of_expr e ^ ")\n" ^ string_of_stmt s
  | If(e, s1, s2) ->  "if (" ^ string_of_expr e ^ ")\n" ^
      string_of_stmt s1 ^ "else\n" ^ string_of_stmt s2
  | For(e1, e2, e3, s) ->
      "for (" ^ string_of_expr e1  ^ " ; " ^ string_of_expr e2 ^ " ; " ^
      string_of_expr e3  ^ ") " ^ string_of_stmt s
  | ForIn(f,g,h,s) -> "for (" ^ f  ^ "," ^ g ^ ")" ^ " in " ^ h ^ ":\n   " ^ string_of_stmt s
  | While(e, s) -> "while (" ^ string_of_expr e ^ ") " ^ string_of_stmt s
  | Def(f,el,sl) -> "def " ^ f ^ "(" ^ String.concat ", " (List.map string_of_expr el) ^ "):\n    "
                 ^ String.concat "" (List.map string_of_stmt sl) ^ "\n"
  | Mrbk(a,sl) -> "flatMapStep = spawkData.flatMap(lambda x: x.split(' '))\n" ^
                 "mapStep = flatMapStep.map(" ^
                  (match sl with
                    | hd :: tl -> string_of_expr hd) ^ ")\n" ^
                 "reduceStep = mapStep.reduceByKey(" ^
                  (match sl with
                    | hd :: tl -> string_of_expr (List.hd tl)) ^ ")\n" ^
                 a ^ " = reduceStep.collect()\n"

let string_of_vdecl id = "int " ^ id ^ ";\n"

let string_of_fdecl fdecl =
  fdecl.fname ^ "(" ^ String.concat ", " fdecl.formals ^ ")\n{\n" ^
  String.concat "" (List.map string_of_stmt fdecl.body) ^
  "}\n"

let string_of_padecl padecl =
  padecl.pattern ^ "\n{\n" ^
  String.concat "" (List.map string_of_stmt padecl.actions) ^
  "}\n"

let string_of_program (patacts) =
  String.concat "\n" (List.map string_of_padecl patacts)

==> codegen.ml <==
open Ast


module StringMap = Map.Make(String)

(* Symbol table: Information about all the names in scope *)
type env = {
    function_index : int StringMap.t; (* Index for each function *)
    global_index   : int StringMap.t; (* "Address" for global variables *)
    local_index    : int StringMap.t; (* FP offset for args, locals *)
  }

exception Codegen_error of string

let codegen_error msg = raise (Codegen_error msg)

(* code generation *)
let chan = ref stdout
let dataFile = ref "string"
(* let var_hash = Hashtbl.create 123456 *)

let set_chan new_chan = chan := new_chan

let set_input new_chan = dataFile := new_chan

let gen v = output_string !chan v; output_string !chan "\n"



let generate_begin () = gen
("from __future__ import print_function
import sys
import re
from operator import add
from pyspark import SparkConf, SparkContext
dataFile = \"" ^ !dataFile ^ "\"  # Should be some file on your system
conf = (SparkConf()
        .setMaster(\"local\")
        .setAppName(\"spawk\")
        .set(\"spark.executor.memory\", \"1g\"))

sc = SparkContext(conf = conf)

")

let generate_end () = gen
"
sc.stop()
"
```

```
let generate_regexp i = gen
("
spawkTempData = sc.textFile(dataFile).cache()
def parse_rrd_line(line):
    match = re.search(\'" ^ i ^ "\',line)
    if match is None:
        return ''
    return line

spawkData = (spawkTempData.map(parse_rrd_line).cache())
"
)

let generate_noregexp () = gen
("
spawkData = sc.textFile(dataFile).cache()
"
)


let rec generate_of_expr = function
    String(s) -> s
  | Number(n) -> n
  | Id(s) -> s
  | Binop(e1, o, e2) ->
      generate_of_expr e1 ^ " " ^
      (match o with
    Add -> "+" | Sub -> "-" | Mult -> "*" | Div -> "/"
      | Equal -> "==" | Neq -> "!="
      | Less -> "<" | Leq -> "<=" | Greater -> ">" | Geq -> ">=" | Mod -> "%") ^ " " ^
```

```
     generate_of_expr e2
  | Assign(v, e) -> v ^ " = " ^ generate_of_expr e
  | Array(v, e) -> v ^ "[" ^ generate_of_expr e ^ "]"
  | PTuple(v, e) -> "(" ^ generate_of_expr v ^ ", " ^ String.concat ", " (List.map generate_of_expr
e) ^ ")"
  | Call(f, el) -> f ^ "(" ^ String.concat ", " (List.map generate_of_expr el) ^ ")"
  | KeyValue(e1,e2) -> "(" ^ generate_of_expr e1 ^ "," ^ generate_of_expr e2 ^ ")"
  | Noexpr -> ""

let rec generate_of_stmt = function
    Block(stmts) ->
     "    " ^ String.concat " ; " (List.map generate_of_stmt stmts) ^ "\n"
  | Expr(expr) -> generate_of_expr expr
  | Return(expr) -> "return " ^ generate_of_expr expr ^ "\n";
  | If(e, s, Block([])) -> "if (" ^ generate_of_expr e ^ ")\n" ^ generate_of_stmt s
  | If(e, s1, s2) ->  "if (" ^ generate_of_expr e ^ ")\n" ^
     generate_of_stmt s1 ^ "else\n" ^ generate_of_stmt s2
  | For(e1, e2, e3, s) ->
     "for (" ^ generate_of_expr e1  ^ " ; " ^ generate_of_expr e2 ^ " ; " ^
     generate_of_expr e3  ^ ") " ^ generate_of_stmt s
  | ForIn(f,g,h,s) -> "for (" ^ f  ^ "," ^ g ^ ")" ^ " in " ^ h ^ ":\n    " ^ generate_of_stmt s
  | While(e, s) -> "while (" ^ generate_of_expr e ^ "):\n     " ^ generate_of_stmt s
  | Def(f,el,sl) -> "def " ^ f ^ "(" ^ String.concat ", " (List.map generate_of_expr el) ^ "):\n     "
               ^ String.concat "     " (List.map generate_of_stmt sl) ^ "\n"
  | Mrbk(a,sl) -> "flatMapStep = spawkData.flatMap(lambda x: x.split(' '))\n" ^
               "mapStep = flatMapStep.map(" ^
                (match sl with
                  | hd :: tl -> generate_of_expr hd) ^ ")\n" ^
               "reduceStep = mapStep.reduceByKey(" ^
                (match sl with
                  | hd :: tl -> generate_of_expr (List.hd tl)) ^ ")\n" ^
                a ^ " = reduceStep.collect()\n"

let generate_of_vdecl id = "int " ^ id ^ ";\n"

let generate_of_fdecl fdecl =
  fdecl.fname ^ "(" ^ String.concat ", " fdecl.formals ^ ")\n{\n" ^
  String.concat "" (List.map generate_of_stmt fdecl.body) ^
  "}\n"

let generate_actions actions = "\n" ^ String.concat "" (List.map generate_of_stmt actions) ^ "\n"

let generate_of_padecl padecl =
  match padecl.pattern with
    | "emptyPattern" -> generate_noregexp () ; generate_actions padecl.actions
    | _ ->  generate_regexp padecl.pattern ; generate_actions padecl.actions



let generate_of_program (patacts) =
  gen(String.concat "\n" (List.map generate_of_padecl patacts))




(* val enum : int -> 'a list -> (int * 'a) list *)
let rec enum stride n = function
    [] -> []
  | hd::tl -> (n, hd) :: enum stride (n+stride) tl

(* val string_map_pairs StringMap 'a -> (int * 'a) list -> StringMap 'a *)
let string_map_pairs map pairs =
  List.fold_left (fun m (i, n) -> StringMap.add n i m) map pairs
```

```
(** Translate a program in AST form into a bytecode program.  Throw an
    exception if something is wrong, e.g., a reference to an unknown
    variable or function *)



let codegen p =
        generate_begin (); generate_of_program p; generate_end ()




==> lexer.mll <==
{
open Lexing
open Parser

(* current token line number *)
let line_num = ref 1

(* keyword -> token translation table *)
let keywords = [
    "begin", BEGIN; "end", END; "read", READ; "write", WRITE; "function", FUNCTION
]

exception Syntax_error of string

let syntax_error msg = raise (Syntax_error (msg ^ " on line " ^ (string_of_int !line_num)))

}

let blank = [' ' '\r' '\t']
let digit = ['0'-'9']
let int = '-'? digit digit*
let digits = digit*
let frac = '.' digit*
let exp = ['e' 'E']['-' '+']? digit+
let float = digit* frac? exp?
let number = (int | float)
let alpha = ['a'-'z' 'A'-'Z']
let iden = alpha (alpha | digit | '_')*

rule micro = parse
    | '='     { ASSIGN }
    | '+'      { PLUS }
    | '-'     { MINUS }
    | '*'     { TIMES }
    | ','     { COMMA }
    | ';'     { SEMICOLON }
    | '('     { LEFTPAREN }
    | ')'     { RIGHTPAREN }
    | '{'     { LEFTBRACE }
    | '}'     { RIGHTBRACE }
    | '['     { LEFTBRACK }
    | ']'     { RIGHTBRACK }
    | '/'     { DIVIDE }
    | '%'      { MOD }
    | '$'     { DOLLAR }
    | "#"      { comment lexbuf }           (* Comments *)
    | '"'     { read_string (Buffer.create 17) lexbuf}
    | "=="     { EQ }
    | "!="     { NEQ }
    | '<'     { LT }
    | "<="     { LEQ }
    | ">"      { GT }
    | ">="     { GEQ }
    | "if"     { IF }
    | "else"   { ELSE }
    | "for"    { FOR }
    | "in"     { IN }
    | "while"  { WHILE }
    | "return" { RETURN }
    | "mapReduceByKey" { MRBK }
    | "emptyPattern" { EP }
    | "var"     { VAR }
    | iden as i {
       (* try keywords if not found then it's identifier *)
       let l = String.lowercase i in
       try List.assoc l keywords
       with Not_found -> IDENTIFIER i
    }
    | number as d {
       (* parse number *)
       NUMBER  d
    }
    | '\n'     { incr line_num; micro lexbuf } (* counting new line characters *)
    | blank    { micro lexbuf } (* skipping blank characters *)
    | _        { syntax_error "couldn't identify the token" }
    | eof      { EOF } (* no more tokens *)
    and read_string buf =
       parse
        | '"'      { STRING (Buffer.contents buf) }
        | '\\' '/'  { Buffer.add_char buf '/'; read_string buf lexbuf }
        | '\\' '\\' { Buffer.add_char buf '\\'; read_string buf lexbuf }
        | '\\' 'b'  { Buffer.add_char buf '\b'; read_string buf lexbuf }
```

```
        | '\\' 'f'  { Buffer.add_char buf '\012'; read_string buf lexbuf }
        | '\\' 'n'  { Buffer.add_char buf '\n'; read_string buf lexbuf }
        | '\\' 'r'  { Buffer.add_char buf '\r'; read_string buf lexbuf }
        | '\\' 't'  { Buffer.add_char buf '\t'; read_string buf lexbuf }
        | [^ '"' '\\']+
          { Buffer.add_string buf (Lexing.lexeme lexbuf);
              read_string buf lexbuf
           }
        | _ { raise (Syntax_error ("Illegal string character: " ^ Lexing.lexeme lexbuf)) }
        | eof { raise (Syntax_error ("String is not terminated")) }

    and comment = parse
    '\n' { micro lexbuf }
    | _    { comment lexbuf }


==> parser.mly <==
%{

open Ast

%}

%token BEGIN END
%token <string> IDENTIFIER
%token <string> NUMBER
%token <string> STRING
%token READ WRITE FUNCTION
%token ASSIGN
%token LEFTPAREN RIGHTPAREN
%token LEFTBRACE RIGHTBRACE
```

```
%token COMMA SEMICOLON DOLLAR EP
%token LEFTBRACK
%token RIGHTBRACK
%token COLON
%token NEWLINE
%token PLUS MINUS TIMES DIVIDE
%token EQ NEQ LT LEQ GT GEQ MOD
%token RETURN IF ELSE FOR WHILE VAR IN MRBK
%token EOF

%nonassoc NOELSE
%nonassoc ELSE
%right ASSIGN
%left EQ NEQ
%left LT GT LEQ GEQ
%left PLUS MINUS
%left TIMES DIVIDE

%start program
%type <Ast.program> program

%%


program:
   EOF { [] }
 | padecl program {  $1 :: $2 }
```

```
padecl:
  LEFTBRACE stmt_list RIGHTBRACE { { pattern = "emptyPattern" ; actions = List.rev
$2 } }
 | DIVIDE IDENTIFIER DIVIDE LEFTBRACE stmt_list RIGHTBRACE { { pattern = $2 ;
actions = List.rev $5 } }



/* fdecl:
   FUNCTION IDENTIFIER LEFTPAREN formals_opt RIGHTPAREN LEFTBRACE
stmt_list RIGHTBRACE
     { { fname = $2;
   formals = $4;
   body = List.rev $7 } } */

formals_opt:
    /* nothing */ { [] }
  | formal_list   { List.rev $1 }

formal_list:
    IDENTIFIER                  { [$1] }
  | formal_list COMMA IDENTIFIER { $3 :: $1 }



stmt_list:
    /* nothing */  { [] }
  | stmt_list stmt { $2 :: $1 }

stmt:
    expr  { Expr($1) }
  | RETURN expr  { Return($2) }
  | LEFTBRACE stmt_list RIGHTBRACE { Block(List.rev $2) }
  | IF LEFTPAREN expr RIGHTPAREN stmt %prec NOELSE { If($3, $5, Block([])) }
  | IF LEFTPAREN expr RIGHTPAREN stmt ELSE stmt    { If($3, $5, $7) }
  | FOR LEFTPAREN expr_opt SEMICOLON expr_opt SEMICOLON expr_opt
RIGHTPAREN stmt
     { For($3, $5, $7, $9) }
  | FOR LEFTPAREN IDENTIFIER COMMA IDENTIFIER RIGHTPAREN IN
IDENTIFIER stmt { ForIn($3,$5,$8,$9) }
  | WHILE LEFTPAREN expr RIGHTPAREN stmt { While($3, $5) }
  | FUNCTION IDENTIFIER LEFTPAREN actuals_opt RIGHTPAREN LEFTBRACE
stmt_list RIGHTBRACE  { Def($2,$4,List.rev $7)}
  | IDENTIFIER ASSIGN MRBK LEFTPAREN actuals_opt RIGHTPAREN { Mrbk($1,$5) }


expr_opt:
    /* nothing */ { Noexpr }
  | expr          { $1 }


expr:
    NUMBER          { Number($1) }
  | STRING          { String($1) }
  | IDENTIFIER               { Id($1) }
  | expr PLUS   expr { Binop($1, Add,   $3) }
  | expr MINUS  expr { Binop($1, Sub,   $3) }
  | expr TIMES  expr { Binop($1, Mult,  $3) }
  | expr DIVIDE expr { Binop($1, Div,   $3) }
  | expr EQ     expr { Binop($1, Equal, $3) }
  | expr NEQ    expr { Binop($1, Neq,   $3) }
  | expr LT     expr { Binop($1, Less,  $3) }
  | expr LEQ    expr { Binop($1, Leq,   $3) }
  | expr GT     expr { Binop($1, Greater,  $3) }
  | expr GEQ    expr { Binop($1, Geq,   $3) }
  | expr MOD    expr { Binop($1, Mod,   $3) }
  | IDENTIFIER ASSIGN expr   { Assign($1, $3) }
  | IDENTIFIER LEFTBRACK expr RIGHTBRACK { Array($1,$3) }
  | IDENTIFIER LEFTPAREN actuals_opt RIGHTPAREN { Call($1, $3) }
  | LEFTPAREN expr COMMA expr RIGHTPAREN { KeyValue($2,$4) }
  | LEFTPAREN expr RIGHTPAREN { $2 }

actuals_opt:
    /* nothing */ { [] }
  | actuals_list  { List.rev $1 }

actuals_list:
    expr                    { [$1] }
  | actuals_list COMMA expr { $3 :: $1 }

%%
==> sast.ml <==
open Ast




let var_hash = Hashtbl.create 123456

let rec last = function
    | [] -> None
    | [x] -> Some x
    | _ :: t -> last t

let rec at k = function
    | [] -> None
    | h :: t -> if k = 1 then Some h else at (k-1) t
```

```
let rec convert_of_expr = function
    String(s) -> s
  | Number(n) -> n
  | Id(s) -> s


let rec check_of_expr = function
    String(s) -> "String"
  | Number(n) -> "Number"
  | Id(s) -> "Indentifer"
  | Binop(e1, o, e2) ->
      (match o with
                Add -> if (check_of_expr e1) == "Number" && (check_of_expr e2) == "Number"
then "Add ok\n" else "Add error\n"
       | Sub -> if (check_of_expr e1) == "Number" &&  (check_of_expr e2) == "Number" then
"Sub ok\n" else "Sub error\n"
       | Mult -> if (check_of_expr e1) == "Number" &&  (check_of_expr e2) == "Number" then
"Mult ok\n" else "Mult error\n"
       | Div -> if (check_of_expr e1) == "Number" &&  (check_of_expr e2) == "Number" then
"Div ok\n" else "Div error\n"
       | Equal -> if (check_of_expr e1) == "Number" &&  (check_of_expr e2) == "Number" then
"Equal ok\n" else "Equal error\n"
       | Neq -> if (check_of_expr e1) == "Number" &&  (check_of_expr e2) == "Number" then
"Neq ok\n" else "Neq error\n"
       | Less -> if (check_of_expr e1) == "Number" &&  (check_of_expr e2) == "Number" then
"Less ok\n" else "Less error\n"
       | Leq -> if (check_of_expr e1) == "Number" &&  (check_of_expr e2) == "Number" then
"Leq ok\n" else "Leq error\n"
       | Greater -> if (check_of_expr e1) == "Number" &&  (check_of_expr e2) == "Number"
then "Greater ok\n" else "Greater error\n"
       | Geq -> if (check_of_expr e1) == "Number" &&  (check_of_expr e2) == "Number" then
"Geq ok\n" else "Geq error\n"
       | Mod -> if (check_of_expr e1) == "Number" &&  (check_of_expr e2) == "Number" then
"Mod ok\n" else "Mod error\n")
```

```
  | Assign(v, e) ->  v ^ "no assign check"
  | Array(v,e) -> v ^ "no array check"
  | PTuple(v, e) -> "no check"
  | Call(f, el) -> f ^ "no call check"
  | KeyValue(e1,e2) -> "no kv check"
  | Noexpr -> ""

let rec check_of_stmt = function
    Block(stmts) -> "{\n" ^ String.concat "" (List.map check_of_stmt stmts) ^ "}\n"
  | Expr(expr) -> check_of_expr expr
  | Return(expr) -> check_of_expr expr
  | If(e, s, Block([])) -> check_of_expr e ; check_of_stmt s
  | If(e, s1, s2) ->  check_of_expr e ; check_of_stmt s1 ; check_of_stmt s2
  | For(e1, e2, e3, s) ->  check_of_expr e1 ; check_of_expr e2 ; check_of_expr e3 ; check_of_stmt
s
  | ForIn(f,g,h,s) -> Hashtbl.add var_hash f  "String" ; Hashtbl.add var_hash g "String" ;
      (let rddVal = Hashtbl.find var_hash h in
        match rddVal with
          "RDD" -> "ForIn RDD ok\n" ^ check_of_stmt s
          | _ -> "ForIn error with: " ^ h)
  | While(e, s) -> "\n" ^ check_of_expr e  ^ check_of_stmt s
  | Def(f,el,sl) ->  "\n" ^ String.concat "" (List.map check_of_expr el) ^ "\n" ^ String.concat
"" (List.map check_of_stmt sl)
  | Mrbk(a,sl) -> Hashtbl.add var_hash a "RDD" ; "check"



let check_of_padecl padecl =
  padecl.pattern ^ "\n{\n" ^
  String.concat "" (List.map check_of_stmt padecl.actions) ^
  "}\n"

let check_of_program (patacts) =
  String.concat "\n" (List.map check_of_padecl patacts)

==> spawk.ml <==
(* compiling *)
let compile f d =
        let out = (Filename.chop_extension f) in
        let out_chan = open_out (out ^ ".py")
        and out_warn = open_out (out ^ ".warn")
        and lexbuf = Lexing.from_channel (open_in f) in
        try
            let program = Parser.program Lexer.micro lexbuf in
            let listing = Ast.string_of_program program
                in output_string out_warn listing ; output_string out_warn "\n";
            let slisting = Sast.check_of_program program
                in output_string out_warn slisting ; output_string out_warn "\n";
            Codegen.set_chan out_chan;
            Codegen.set_input d;
            Codegen.codegen program;
            close_out out_chan;
            close_out out_warn;
            (* ignore(Sys.command ("spark-submit " ^ out ^ ".py 2> spark.log")); *)
            ignore(Sys.command ("python -m py_compile " ^ out ^ ".py >> " ^ out ^ ".warn"))
        with
        | Codegen.Codegen_error s ->
            print_string s;
            print_string "\n";
            exit 1
        | Lexer.Syntax_error s ->
```

```
            print_string s;
            print_string "\n";
            exit 1

let help () = print_string "spawk.native <program file> <data file>\n"

let () = if Array.length Sys.argv < 2 then help ()
         else
             let file = Array.get Sys.argv 1 in
                 let dataFile = Array.get Sys.argv 2 in
                 Format.printf "compiling %s\n" file;
                 Format.print_flush ();
                 compile file dataFiled

==> build.sh <==
ocamlbuild -use-menhir -yaccflag --trace -use-ocamlfind spawk.native

==> test_gcd.sh <==
#!/bin/sh
# script to compile and test spawk with

TEST="gcd program"

SignalError() {
    if [ $error -eq 0 ] ; then
            echo "FAILED"
            error=1
    fi
    echo "  $1"
}
```

```
# Compare <outfile> <reffile> <difffile>
# Compares the outfile with reffile.  Differences, if any, written to difffile
Compare() {
    #generatedfiles="$generatedfiles $3"
    echo diff -b $1 $2 ">" $3 1>&2
    diff -b "$1" "$2" > "$3" 2>&1 || {
            SignalError "$1 differs"
            echo "FAILED $1 differs from $2" 1>&2
            exit
    }
    echo "Test PASS for $TEST"
}

# Run <args>
# Report the command, run it, and report any errors
Run() {
    echo $* 1>&2
    eval $* || {
            SignalError "$1 failed on $*"
            return 1
    }
}



# run the control python script through spark , sort needed for random reducer order from spark parallel system
python test_gcd.py 2> /dev/null | sort > gcd_test.out

# compile and run generated python through spark
rm -f spawk.native
ocamlbuild -use-menhir -yaccflag --trace -use-ocamlfind spawk.native
```

```
rm -f gcd.py
./spawk.native gcd.spawk ../wordcount_test.txt > /dev/null 2>&1
rm -f spawk_output.out
spark-submit gcd.py 2> /dev/null | sort > spawk_output.out

Compare spawk_output.out gcd_test.out diffFile



==> test_suite.sh <==
#!/bin/sh
# run all tests

sh test_gcd.sh
sh test_wordcount_nopat.sh
sh test_wordcount_pat.sh

==> test_wordcount_nopat.sh <==
#!/bin/sh
# script to compile and test spawk with

TEST="no pattern word count program"

SignalError() {
   if [ $error -eq 0 ] ; then
        echo "FAILED"
        error=1
   fi
   echo "  $1"
}

# Compare <outfile> <reffile> <difffile>
# Compares the outfile with reffile.  Differences, if any, written to difffile
Compare() {
    #generatedfiles="$generatedfiles $3"
    echo diff -b $1 $2 ">" $3 1>&2
    diff -b "$1" "$2" > "$3" 2>&1 || {
            SignalError "$1 differs"
            echo "FAILED $1 differs from $2" 1>&2
            exit
    }
    echo "Test PASS for $TEST"
}

# Run <args>
# Report the command, run it, and report any errors
Run() {
    echo $* 1>&2
    eval $* || {
            SignalError "$1 failed on $*"
            return 1
    }
}



# run the control python script through spark , sort needed for random reducer order from spark parallel system
spark-submit test_nopat_wc.py ../wordcount_test.txt 2> /dev/null | sort > wordcount_nopat_test.out

# compile and run generated python through spark
rm -f spawk.native
rm -f wordcount_nopat.py
```

```
}

ocamlbuild -use-menhir -yaccflag --trace -use-ocamlfind spawk.native
./spawk.native wordcount_nopat.spawk ../wordcount_test.txt > /dev/null 2>&1
spark-submit wordcount_nopat.py 2> /dev/null | sort > spawk_output.out

Compare spawk_output.out wordcount_nopat_test.out diffFile



==> test_wordcount_pat.sh <==
#!/bin/sh
# script to compile and test spawk with

TEST="pattern word count program"

SignalError() {
   if [ $error -eq 0 ] ; then
           echo "FAILED"
           error=1
   fi
   echo "  $1"
}

# Compare <outfile> <reffile> <difffile>
# Compares the outfile with reffile.  Differences, if any, written to difffile
Compare() {
   #generatedfiles="$generatedfiles $3"
   echo diff -b $1 $2 ">" $3 1>&2
   diff -b "$1" "$2" > "$3" 2>&1 || {
           SignalError "$1 differs"
           echo "FAILED $1 differs from $2" 1>&2
           exit
   }
```

```
    echo "Test PASS for $TEST"
}

# Run <args>
# Report the command, run it, and report any errors
Run() {
    echo $* 1>&2
    eval $* || {
            SignalError "$1 failed on $*"
            return 1
    }
}


# run the control python script through spark , sort needed for random reducer order from spark
parallel system
spark-submit test_pat_wc.py ../wordcount_test.txt 2> /dev/null | sort > wordcount_pat_test.out

# compile and run generated python through spark
rm -f spawk.native
ocamlbuild -use-menhir -yaccflag --trace -use-ocamlfind spawk.native
rm -f spawk_output.out
rm -f wordcount.py
./spawk.native wordcount.spawk ../wordcount_test.txt > spawk_output.out 2> /dev/null
spark-submit wordcount.py 2> /dev/null | sort > spawk_output.out

Compare spawk_output.out wordcount_pat_test.out diffFile
```