Homework 1 Report

Taylor Berger

November 30, 2014

1 Synopsis

I decided to continue with the work my team did for project 2 and use the AST generator that was already in place. Continuing with the tradition, all work was done in Python 2.7. I implemented a simple lexer using regular expressions to feed into the parser. Most of my time was spent reducing the AST into a concrete syntax tree. The conversion looks incredibly messy and I am not satisfied with the succinctness of the conversion method, but it works and I ran out of time to make it abide any sort of sanity standards.

2 The AST

The AST constructor was finished in project 2 with on minor adjustments as I kept coding the homework. The tokens are now suppposed to be pairs of token identifiers and the value of the token. Since variables and numbers don't have a specific value that can be assigned and identified, they token identifiers serve as the additional information that is needed for the parser to identify literals as the same class of tokens. The code is all located in the appendix

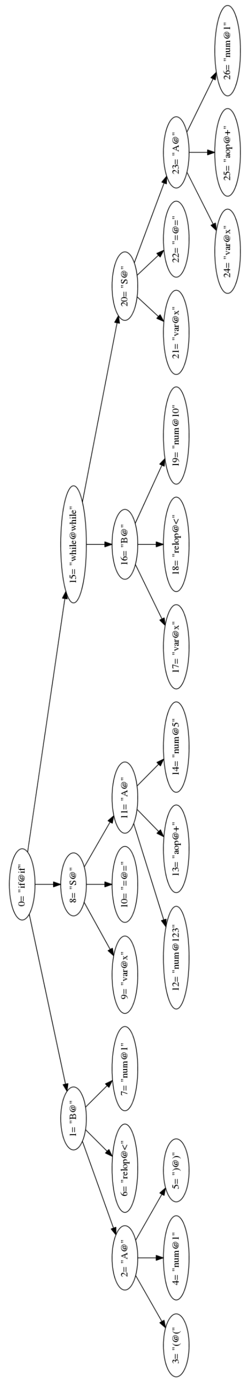
3 Diplaying the AST

The AST can be displayed with some utility functions in the file `ast_reductions.py`. After parsing any ll1 grammar into an AST you can do the following to come up with a nice looking AST in a specified png file:

```python
root, _ = parser.ll1_parse(token_stream)
root = reduce_ast(root)
graph = pydot.Dot('Parse Tree', graph_type='digraph')
g, _ = root.pydot_append(graph, 0)
g.write_png(filename)
```

Here is an example parse of a small program I used to test parsing was happening correctly. This is after the reductions were done to help improve the quality of the tree.

A better picture can be found here.


0= "if@if"
1= "B@"
2= "A@"
3= "(@("
4= "num@1"
5= ")@)"
6= "relop@<"
7= "num@1"
8= "S@"
9= "var@x"
10= "=@="
11= "A@"
12= "num@123"
13= "aop@+"
14= "num@5"
15= "while@while"
16= "B@"
17= "var@x"
18= "relop@<"
19= "num@10"
20= "S@"
21= "var@x"
22= "=@="
23= "A@"
24= "var@x"
25= "aop@+"
26= "num@1"

4 Converting to LLVM

All conversion from a Concrete Syntax tree occurs in the file `ast_to_llvm.py` and should be viewed in the code appendix immediately following. Once in a Concrete Syntax Tree, the LLVM conversion was very simple – it follows the CST directly and only two special cases were needed (while and if statements).

5 Tieing it All Together

A testing suite that compiles all pieces together into native code is provided in `homework1_suite.py`. The file can be run directly on another file that contains the arithmetic language and will output native code.

6 Problems and Limitations

There is a small problem with the Parse Table and grammar representation. For some reason, when constructing the parse table, there are multiple entries in some of the table cells. However using other external tools, the grammar is an LL1 grammar. Upon closer inspection, the parse table tends to have an extra production of the form X -¿ EPSILON in some of the table's cells. Instead of failing, we try to parse the using the "left-most" production (which is the non-epsilon production). Every parsed instance shows that this is perfectly fine and there has been no cases in which the parse fails on a valid grammar when it finds a table's cell with two productions.

Code Appendicies

I'm only including the new code in here that differed from project 2. The other code can be found at my github. The minor changes to the other files were excluded since they were minor bug fixes and did not affect the rest of the code.

6.1 ast_to_llvm.py

```
from cfg import Grammar
from ast_parser import Parser
import pydot
import ast_reductions
import subprocess
import re

#bad practice, but oh well.
from llvm import *
from llvm.core import *

tp_int = Type.int()
tp_bool = Type.int(1)
tp_main = Type.function(tp_int, [])
gv_results = {}
```

```python
def reduce_cst_to_llvm(cst):
    llvm_module = Module.new('Arithmetic Code')
    gv_results = {}

    f_main = llvm_module.add_function(tp_main, 'main')
    bb = f_main.append_basic_block('entry')
    builder = Builder.new(bb)

    _, post_builder = cst.to_llvm(builder, f_main)

    exit_bb = build_exit(llvm_module, f_main)
    post_builder.branch(exit_bb)

    return llvm_module

def build_exit(llvm_module, f_main):
    exit_bb = f_main.append_basic_block('exit')
    exit_builder = Builder.new(exit_bb)

    #copied from the example code...
    printstring = Constant.stringz('Global variable %s = %d\n')
    gv_printstring = llvm_module.add_global_variable(printstring.type, 'print_fmt')
    gv_printstring.initializer = printstring
    pt_printstring = exit_builder.gep(gv_printstring,
                                  [Constant.int(tp_int, 0),
                                   Constant.int(tp_int, 0)],
                                  inbounds=True)

    #creating a prototype for printf
    tp_string = Type.pointer(Type.int(8))
    tp_print = Type.function(Type.void(), [tp_string], var_arg=True)
    f_printf = llvm_module.add_function(tp_print, 'printf')

    sorted_keys = gv_results.keys()[0:]
    sorted_keys.sort()
    for gv in sorted_keys:
        value = exit_builder.load(gv_results[gv])
        string_val = Constant.stringz(gv)
        string_val_global = llvm_module.add_global_variable(string_val.type, gv+'_var')
        string_val_global.initializer = string_val
        pt_string = exit_builder.gep(string_val_global,
                                  [Constant.int(tp_int, 0),
                                   Constant.int(tp_int, 0)],
                                  inbounds=True)

        exit_builder.call(f_printf, [pt_printstring, pt_string, value])
    exit_builder.ret(Constant.int(tp_int, 0))
    return exit_bb

def ast_to_cst(ast):
    if ast.symbol == 'if':
        boolean_expression = ast_to_cst(ast.children[0])
        then_statement = ast_to_cst(ast.children[1])
        else_statement = ast_to_cst(ast.children[2])

        node = StatementIf(boolean_expression,
                           then_statement,
                           else_statement)

        boolean_expression.parent = node
        then_statement.parent = node
        else_statement.parent = node
        return node
    elif ast.symbol == 'while':
        boolean_expression = ast_to_cst(ast.children[0])
        do_statement = ast_to_cst(ast.children[1])
        node = StatementWhile(boolean_expression, do_statement)

        boolean_expression.parent = node
        do_statement.parent = node
        return node

    elif ast.symbol == 'skip':
        return StatementSkip()
    #we have sequential statements and will need to parse them
    #separeately
    elif ';' in [c.symbol for c in ast.children]:
        idx = [c.symbol for c in ast.children].index(';')
        c1 = ast.children[0:idx]
        c2 = ast.children[idx+1:]

        ast.children = c1
        s1 = ast_to_cst(ast)
        ast.children = c2
        s2 = ast_to_cst(ast)

        node = StatementSequential(s1, s2)
        s1.parent = node
        s2.parent = node

        return node
    elif ast.symbol == 'S':
```

```
    symbols = [c.symbol for c in ast.children]

    if ':=' in symbols:
        var = ast.children[0].value
        rhs = ast_to_cst(ast.children[2])
        node = StatementAssignment(var, rhs)
        rhs.parent = node
        return node
    if 'S' in symbols:
        #skip this node,
        return ast_to_cst(ast.children[symbols.index('S')])
    elif len(symbols) == 1:
        return ast_to_cst(ast.children[0])
    else:
        raise ValueError('should never get here... Symbol:' + str(ast.symbol))
elif ast.symbol in ['A', 'num', 'var']:
    if ast.symbol == 'A':
        symbols = [c.symbol for c in ast.children]

        if '(' in symbols:
            expr = ast_to_cst(ast.children[1])
            return ArithmeticParenthesized(expr)
        elif 'aop' in symbols:
            lhs = ast_to_cst(ast.children[0])
            op = ast.children[1].value
            rhs = ast_to_cst(ast.children[2])

            node = ArithmeticOperation(lhs, op, rhs)
            lhs.parent = node
            rhs.parent = node
            return node

    elif ast.symbol == 'num':
        return ArithmeticNumber(ast.value)
    else:
        return ArithmeticVariable(ast.value)
elif ast.symbol in ['B', 'true', 'false']:
    if ast.symbol == 'B':
        symbols = [c.symbol for c in ast.children]
        if 'bop' in symbols:
            idx = symbols.index('bop')
            op = ast.children[idx].value

            c1 = ast.children[0:idx]
            c2 = ast.children[idx+1:]
```

```
                ast.children = c1
                b1 = ast_to_cst(ast)

                ast.children = c2
                b2 = ast_to_cst(ast)

                node = BooleanBinary(b1, op, b2)
                b1.parent = node
                b2.parent = node
                return node
            elif 'not' in symbols:
                b = ast_to_cst(ast.children[1])
                node = BooleanNot(b)
                b.parent = node
                return node
            elif 'relop' in symbols:
                lhs = ast_to_cst(ast.children[0])
                op  = ast.children[1].value
                rhs = ast_to_cst(ast.children[2])
                node = BooleanRelation(lhs, op, rhs)
                lhs.parent = node
                rhs.parent = node
                return node
            else: #true or false value here...
                return BooleanValue(ast.children[0].value)

        else:
            return BooleanValue(ast.value)


class ProgramNode:
    def __init__(self):
        pass
    def to_llvm(self, incoming_builder, f_main):
        pass

class StatementWhile(ProgramNode):
    def __init__(self, boolean_expression, do_statement, parent=None):
        self.parent = parent
        self.bool_expr = boolean_expression
        self.do_statement = do_statement

    def to_llvm(self, incoming_builder, f_main):
        exit_block = f_main.append_basic_block('exit')
        exit_builder = Builder.new(exit_block)
        do_block = f_main.append_basic_block('do_block')
        do_builder = Builder.new(do_block)

        b, builder = self.bool_expr.to_llvm(incoming_builder,
                                            f_main)

        #true if b is true, false if b is false
        tmp = builder.icmp(ICMP_NE, b, Constant.int(tp_bool, 0))
        builder.cbranch(tmp, do_block, exit_block)

        self.do_statement.to_llvm(do_builder, f_main)

        do_builder.branch(exit_block)

        return None, exit_builder

class StatementIf(ProgramNode):
    def __init__(self, bool_expr, then_statement, else_statement, parent=None):
        self.parent = parent
        self.boolean_expression = bool_expr
        self.then_statement = then_statement
        self.else_statement = else_statement

    def to_llvm(self, incoming_builder, f_main):
        exit_block = f_main.append_basic_block('exit')
        exit_builder = Builder.new(exit_block)

        then_block = f_main.append_basic_block('then')
        then_builder = Builder.new(then_block)

        else_block = f_main.append_basic_block('else')
        else_builder = Builder.new(else_block)

        b, builder = self.boolean_expression.to_llvm(incoming_builder,
                                                     f_main)

        tmp = builder.icmp(ICMP_NE, b, Constant.int(tp_bool, 0))
        builder.cbranch(tmp, then_block, else_block)

        _, then_builder = self.then_statement.to_llvm(then_builder,
                                                      f_main)
        then_builder.branch(exit_block)

        _, else_builder = self.else_statement.to_llvm(else_builder,
                                                      f_main)
        else_builder.branch(exit_block)
        return None, exit_builder

class StatementSequential(ProgramNode):
    def __init__(self, s1, s2, parent=None):
        self.parent = parent
        self.s1 = s1
        self.s2 = s2

    def to_llvm(self, incoming_builder, f_main):
        _, builder = self.s1.to_llvm(incoming_builder, f_main)
        ret, builder = self.s2.to_llvm(builder, f_main)
        return ret, builder

class StatementAssignment(ProgramNode):
    def __init__(self, var, value, parent=None):
        self.parent = parent
        self.var = var
        self.value = value

    def to_llvm(self, incoming_builder, f_main):
        #calculate the rhs
        v, builder = self.value.to_llvm(incoming_builder, f_main)

        #Is this something we already have a variable for?
        if self.var in gv_results.keys():
            #yes, then just load the value into the ptr
            builder.store(v, gv_results[self.var])
        else:
            #allocate memory for that value
            value_pt = builder.malloc(tp_int, self.var)
            builder.store(v, value_pt)
            gv_results[self.var] = value_pt

        return v, incoming_builder

class StatementSkip(ProgramNode):
    def __init__(self, parent=None):
        self.parent = parent

    def to_llvm(self, incoming_builder, f_main):
        return None, incoming_builder

class ArithmeticParenthesized(ProgramNode):
    def __init__(self, a_statement, parent=None):
        self.parent = parent
        self.expr = a_statement

    def to_llvm(self, incoming_builder, f_main):
        return self.expr.to_llvm(incoming_builder, f_main)

class ArithmeticOperation(ProgramNode):
    def __init__(self, lhs, op, rhs, parent=None):
        self.parent = parent
        self.lhs = lhs
        self.op  = op
        self.rhs = rhs

    def to_llvm(self, incoming_builder, f_main):
        x, builder = self.lhs.to_llvm(incoming_builder, f_main)
        y, builder = self.rhs.to_llvm(builder, f_main)

        if self.op == '+':   ret = builder.add(x, y)
        elif self.op == '-': ret = builder.sub(x, y)
        elif self.op == '*': ret = builder.imul(x, y)
        else: raise ValueError("Error in Arithmetic Operation. Operator was found to be: " + self.op)

        return ret, builder

class ArithmeticNumber(ProgramNode):
    def __init__(self, num, parent=None):
        self.parent = parent
        self.num = num

    def to_llvm(self, incoming_builder, f_main):
        val = Constant.int(tp_int, self.num)
        return val, incoming_builder

class ArithmeticVariable(ProgramNode):
    def __init__(self, var, parent=None):
        self.parent = parent
        self.var = var

    def to_llvm(self, incoming_builder, f_main):
        if self.var in gv_results.keys():
            value = incoming_builder.load(gv_results[self.var])
            return value, incoming_builder
        else:
            print gv_results
            raise ValueError(self.var + " is not in memory yet!")

class BooleanValue(ProgramNode):
    def __init__(self, value, parent=None):
        self.parent = parent
        self.value = value

    def to_llvm(self, incoming_builder, f_main):
        if self.value == 'true':    ret = Constant.int(tp_bool, 1)
        elif self.value == 'false': ret = Constant.int(tp_bool, 0)
        else: raise ValueError("Boolean value is not true of false: " + self.value)

        return ret, incoming_builder

class BooleanNot(ProgramNode):
    def __init__(self, boolean_expression, parent=None):
        self.parent = parent
        self.expr = boolean_expression

    def to_llvm(self, incoming_builder, f_main):
        ret, builder = self.expr.to_llvm(incoming_builder, f_main)
        #one's compliment of the return value
        builder.not_(ret)
        return ret, builder

class BooleanBinary(ProgramNode):
    def __init__(self, lhs, boolean_op, rhs, parent=None):
        self.parent = parent
        self.lhs = lhs
        self.op  = boolean_op
        self.rhs = rhs

    def to_llvm(self, incoming_builder, f_main):
        lhs,builder = self.lhs.to_llvm(incoming_builder, f_main)
        rhs,builder = self.rhs.to_llvm(builder, f_main)

        if self.op == '&&':   ret = builder.and_(lhs, rhs)
        elif self.op == '||': ret = builder.or_(lhs, rhs)
        else:raise ValueError('Not a valid boolean self.operator:' + self.op)

        return ret, builder

class BooleanRelation(ProgramNode):
    def __init__(self, lhs, comparison_operator, rhs, parent=None):
        self.parent = parent
        self.lhs = lhs
        self.op  = comparison_operator
        self.rhs = rhs
```

```
def to_llvm(self, incoming_builder, f_main):
    lhs,builder = self.lhs.to_llvm(incoming_builder, f_main)
    rhs,builder = self.rhs.to_llvm(builder, f_main)

    if self.op == '<':   ret = builder.icmp(ICMP_SLT, lhs, rhs)
    elif self.op == '<=':ret = builder.icmp(ICMP_SLE, lhs, rhs)
    elif self.op == '=': ret = builder.icmp(ICMP_EQ, lhs, rhs)
    elif self.op == '>=':ret = builder.icmp(ICMP_SGE, lhs, rhs)
    elif self.op == '>': ret = builder.icmp(ICMP_SGT, lhs, rhs)
    elif self.op == '!=':ret = builder.icmp(ICMP_NE, lhs, rhs)
    else: raise ValueError('Not a valid relational self.self.operator:' + self.op)

    return ret, builder
```

6.2 ast_reductions.py

```
from cfg import Grammar, EOF
from ast_parser import Parser, Rose_Tree
import pydot

def simplify_ast(ast):
    '''Reduces the AST to a concrete syntax tree (CST), which removes
    all the extraneous non-terminal symbols and creates a logical
    program structure.

    For example a _while_ loop should be represented as it's own tree
    node, with a single node with 2 children: the boolean expression
    to test and the contents of the while loop.

    while loops and if statements are condensed, and sequential
    statements (S ; S) are put into one serial block of
    code.

    :param ast: a RoseTree that represents the AST of the parse
                of the arithmetic language defined in the
                homework specification.
    :return: a ProgramTree with the correct code flow and
             precedence ordering.

    '''
    children_symbols = [c.symbol for c in ast.children if c.value != '']

    #while statements: S -> (while) (B) (do) (S) (od)
    if 'while' in children_symbols:
        return reduceWhile(ast)
    #if statements: S -> (if) (B) (then) (S) (else) (S) (fi)
    elif 'if' in children_symbols:
        return reduceIf(ast)
    #else, leave it as is, this contains valuable information
    #regarding precedence
    else:
        node = Rose_Tree(ast.symbol, ast.value)

        for c in ast.children:
            child_node = reduce_ast(c)
            child_node.parent = node
            node.children.append(child_node)

        return node
```

```
def find_child_with_symbol(ast, symbol, num=1):
    count = 0
    for c in ast.children:
        if c.symbol == symbol:
            count += 1
            if count == num:
                return c
    return None;

#We know that at this level, the root of the AST is S, we can abstract
#away the S-> ...  and replace it with a simplified rose_tree node
#that represents the appropriate structure
def reduceWhile(ast):
    new_root = Rose_Tree(symbol = 'while', node_value='while')
    new_root.parent = ast.parent

    boolean_child = find_child_with_symbol(ast, 'B')
    while_block = find_child_with_symbol(ast, 'S')

    new_root.children = [reduce_ast(boolean_child),
                         reduce_ast(while_block)]
    return new_root

#reduces an S -> if .... fi production into a single node with only
#the three children: boolean expression, then statement, and else
#statement
def reduceIf(ast):
    new_root = Rose_Tree(symbol='if', node_value='if')
    new_root.parent = ast.parent

    boolean_child = find_child_with_symbol(ast, 'B')
    then_statement = find_child_with_symbol(ast, 'S')
    else_statement = find_child_with_symbol(ast, 'S', 2)

    new_root.children = [reduce_ast(boolean_child),
                         reduce_ast(then_statement),
                         reduce_ast(else_statement)]
    return new_root

#removes all nodes that were created from using epsilon productions
def filter_epsilon(ast):
    if not ast.children and (ast.value == ''):
        return None
    else:
        children = [filter_epsilon(c) for c in ast.children]
        tree = Rose_Tree(ast.symbol, ast.value)
        tree.children = filter(lambda x: x is not None, children)
        return tree

#compresses nodes with only one children since they do not contribute
#to the actual structure of the language
def reduce_singleton_children(ast):
    #leaf nodes always stay the same
    if len(ast.children) == 0:
        return ast

    #else, we only have one child, remove ourselves from the equation
    elif len(ast.children) == 1:
        new_child = reduce_singleton_children(ast.children[0])
        return new_child
    else:
        children = ast.children[0:len(ast.children)]
        ast.children = []

        for c in children:
            new_child = reduce_singleton_children(c)
            ast.children.append(new_child)

        return ast

#removes all A -> A' edges and condenses them into a the A node like
#we originally wanted in our grammar
def remove_ll1_requirement_syntax(ast):
    symbol = ast.symbol

    children = ast.children[0:]

    reduce_node = None

    for c in children:
        if (symbol + "'") == c.symbol:
            reduce_node = c

    if reduce_node is not None:
        ast.children.remove(reduce_node)
        for c in reduce_node.children:
            ast.children.append(c)

    for c in ast.children:
        remove_ll1_requirement_syntax(c)
```

```
def reduce_ast(ast):
    '''The compilation of all reductions possible into the smallest AST
    that still maintains grammar structure. See filter_epsilon,
    simplify_ast, reduce_singleton_children, and
    remove_ll1_requirement_syntax.

    '''
    root = filter_epsilon(ast)
    root = simplify_ast(root)
    root = reduce_singleton_children(root)
    remove_ll1_requirement_syntax(root)
    return root

if __name__ == '__main__':
    g = Grammar('./testdata/homework1_grammar.txt')
    x = Parser(g)

    root, _ = x.ll1_parse([('if', 'if'),
                            ('(', '('),
                            ('num', '1'),
                            (')', ')'),
                            ('relop','<'),
                            ('num', '1'),
                            ('then','then'),
                            ('var', 'x'),
                            (':=', ':='),
                            ('num','123'),
                            ('aop', '+'),
                            ('num', '5'),
                            ('else', 'else'),
                            ('while', 'while'),
                            ('var', 'x'),
                            ('relop', '<'),
                            ('num', '10'),
                            ('do', 'do'),
                            ('var', 'x'),
                            (':=',':='),
                            ('var', 'x'),
                            ('aop', '+'),
                            ('num', '1'),
                            ('fi', 'fi'),
                            ('\0', '\0')
                        ])

    root = reduce_ast(root)
    graph = pydot.Dot('Parse Tree', graph_type='digraph')
```

```
g, _ = root.pydot_append(graph, 0)
g.write_png('./testdata/test.png')
```

6.3 homework1_suite.py

```
import re
from cfg import Grammar
from ast_parser import Parser
import ast_to_llvm
import ast_reductions
import subprocess
import sys

var = ('var','[a-zA-Z]+')
num = ('num','[1-9]{1}[0-9]?')
eq = (':=',':=')
semi = (';', ';')
skip = ('skip', 'skip')
if_keyword = ('if', 'if')
fi_keyword = ('fi', 'fi')
then_keyword = ('then', 'then')
else_keyword = ('else', 'else')
do_keyword = ('do','do')
od_keyword = ('od', 'od')
while_keyword = ('while', 'while')
open_paren = ('(', '\(')
close_paren = (')', '\)')
relop = ('relop', '<=|>=|!=|=|<|>')
bop = ('bop', '&&|\|\|')
true = ('true', 'true')
false = ('false', 'false')

terminals = {c : re.compile(v) for c,v in [eq, semi, skip, if_keyword, fi_keyword, then_keyword, else_ke

def tokenize(string):
    string = re.sub('\s', ',', string)
    tokens = string.split(',')

    pairs = []
    for token in tokens:
        for term in terminals:
            #the first one that matches gets precedence
            m = re.search(terminals[term], token)
            if m:
                pairs.append((term, token))
                continue
    return pairs

def compile_to_llvm(file):
    print("Opening file....")
    string = open(file,'r').read()
    print('Done.')

    print('Lexing...')
    tokens = tokenize(string)
    print('Done.')

    print('Constructing parser...')
    parser = Parser(Grammar('./testdata/homework1_grammar.txt'))
    print('Done.')

    print('Parsing tokens...')
    root = parser.ll1_parse(tokens)
    print('Done.')

    print('Reducing AST...')
    root = ast_reductions.reduce_ast(root)
    print('Done.')

    print('Constructing CST...')
    cst = ast_to_llvm.ast_to_cst(root)
    print('Done.')

    print('Constructing LLVM code...')
    llvm_code = ast_to_llvm.reduce_cst_to_llvm(cst)
    llvm_code.verify()
    print('Done.')

    print
    print
    print('****LLVM Code:*****')
    print llvm_code
    print

    return llvm_code


def llvm_to_native(name, llvm):
    print('Constructing native code...')
    obj = name + '.o'
    dst = name

    print("* Compiling to '{}'.".format(dst))

    with open(obj, 'wb') as f:
        llvm.to_native_object(f)
    cmd = ['cc', '-o', dst, obj]
    r = subprocess.call(cmd)
    if r != 0:
        raise Exception("Failed to link with " + str(cmd))

    print('Done.')


if __name__ == '__main__':
    if len(sys.argv) != 2:
        print "Script file takes the file to parse, exiting..."
        exit(1)

    file_name = sys.argv[1]
    llvm_code = compile_to_llvm(file_name)
    llvm_to_native(file_name, llvm_code)
```