Arithmetic Calculation Language

Nathan Corvino

May 12, 2011

1 Introduction

The Arithmetic Calculation Language (ACL) focuses on arithmetic operations, combined with common procedural control structures. It's primary motivation is to explore assembly code generation; as such it is primarily focused on remaining simple to keep this task tractable. At the same time, it does strives to explore as many different code constructs as feasible. It includes variables, conditionals, loops, and function calls.

The syntax is c-like. Statements are terminated with a semicolon, and a list of statements can be enclosed in brackets.

1.1 Data Types

ACL supports ints and arrays of ints.

1.2 Operators

Addition, subtraction, multiplication, and division of integers is supported. Comparison operators <, >, =, <=, >=, and != compare int values, and return 1 if the comparison is true, 0 if false.

1.3 Control Structures

ACL supports while loops and if-else structures.

1.4 Functions

Function calls are be supported, and the number of parameters in the definition and call has to match. There is no limit on the number of parameters for a function. Functions return ints.

The program entry point is the main method, which returns an int. Passing command line arguments to it is not be supported.

1.5 print

ACL has a built-in print function, that prints an int with a newline to standard output.

1.5 Example

An example program, that illustrates these constructs in action:

```
abs(int number)
{
	if (number < 0) {
		return -1 * number;
```

```
        } else {
                return number;
        }
}

power(int base, int exponent)
{
        int total;

        total = 1;

        while (exponent > 0) {
                total = base * total; exponent = exponent - 1;
        }

        return total;
}

int main()
{
        int x;
        int y;

        x = 4;
        y = -3

        print(power(x, abs(y));
}
```

This sample program returns 64 when compiled by gcc, as expected; it does the same when compiled by the ACL compiler.

This example demonstrates comparisons, looping, branching, and function calls.

2 Tutorial

The Arithmetic Calculation Language (ACL) is a simple procedural language for manipulating integers. Its is syntax is c-like, although greatly simplified.

Programs start execution in a main function, and output results using the built in print function, performing integer calculations with standard operators.

2.1 An First Example

```
int main()
{
	int x;
	int y;

	x = 4;
	y = -3

	print(x + y);
	print(x * y);
	print(x - 1);
	print(x / 2);
}
```

The above program proceeds by assigning values to the two variables defined, x and y, and then performing simple math with them. It outputs:

```
1
-12
3
2
```

2.2 Compiling and Running

ACL generates assembly code for the GNU Linux assembler. Furthermore, it depends on the standard c libraries, so the linker must be told where to load them. To compile and run a program, test.acl, the following commands would be executed:

```
./acl -c < test.acl > test.s
as -o test.o test.s
ld -dynamic-linker /lib/ld-linux.so.2 -o test -lc test.o
./test
```

The first command uses ACL to generate the assembly code; the -c flag indicates compile. ACL prints its output to standard output, which is piped to test.s

Next, the assembler is run on test.s, production and object file, test.o. This is linked using the linker, specifying the location of the dynamic linker, and output the test executable. Finally, that is run.

2.3 Another Example

A more complicated example, which illustrates a lot of the features of ACL, calculates whether a number is prime:

```
prime(int a) {
	int i;
	int multiplier;
	int result;

	i = a / 2;

	while(i > 2) {
		multiplier = a / i;
		result = i * multiplier;

		if (result == a) {
			return 0;
		}

		i = i - 1;
	}

	return 1;
}

main()
{
	print(prime(7));
	print(prime(37));
	print(prime(42));
	print(prime(45));
}
```

ACL does not include remainder division, so a search for divisors proceeds by trying to divide, then multiply the division result, by all the integers between 3 and half of the argument to prime. If the result of dividing and re-multiplying produces the starting number, then a divisor is found, and 0 is returned. Otherwise, 1 is returned.

A while loop is used to try each number, subtracting one each time through the loop.

3 Language Reference Manual

3.1 Introduction

This reference manual describes the Arithmetic Calculation Language (ACL), a small subset of C that allows numeric calculations to be done in a procedural fashion. The language only allows integer values to be manipulated, hence the name Arithmetic Calculation Language. It does, however, allow for arrays of integers to be declared, and provides a language features that are illustrative of the core procedural constructs: variable declarations, function definitions, selection, and iteration.

Ints in ACL are 32 bits; it generates 32-bit Linux assembly code, and the data values are handled by the machine. No checks are done for overflow or underflow; the behavior of programs which produce such results is undefined.

3.2 Lexical Conventions

A program consists of a single file, containing global variable declarations and procedure definitions. Execution begins with the function named main, which is required.

3.2.1 Comments

Comments are introduced by /* or //. Comments introduced by /* are terminated with */, while comments introduced by // terminate at the next newline.

3.2.2 Identifiers

Identifiers consist of letters, digits, and underscores, and must begin with a letter or underscore; case is significant.

3.2.3 Keywords

The following identifiers are used as keywords, and may not be used as identifiers for functions or variables:

- if
- else
- while
- int
- return

3.2.4 Constants

Integer constants consist of a series of digits, and are treated as decimal numbers.

3.3 Variable Declarations

Variables are introduced by their type–the only allowed type is int–folowed by an identifier. If the identifier is followed by an expression in square brackets— [] —then the variable is treated as an array–a block of contiguous storage ints represented by the expression value. Array sizes can only be specified using positive literal values; expression, zero, or negative integers are errors. If the identifier appears without brackets, it stores a single integer value.

variable_declaration:
int *ID*;
int *ID*[*LITERAL*];

Assignment to identifiers is done an expression in the form E1 = E2, where E1 must be a non-array identifier, or an array identifier with an index expression in brackets. The expression E1 = E2 also has the value of E2.

Variable declarations can appear outside of all function declarations, in which case they are available throughout the program; or they appear at the beginning of a block–that is, in a list of statements enclosed in curly- braces, before the first statement.

3.4 Function Definitions

Functions are defined by an identifier that names the function, a list of parameters separated by commas, enclosed in parentheses, and a block of code enclosed in braces. Parameters are optional, but the parentheses are mandatory. Only integers can be passed as parameters, not arrays.

function_definition:
ID(*parameter_opt*) {*declaration_opt statement_opt*}

parameter_opt:
parameter_list$_{opt}$

parameter_list:
parameter_declaration
parameter_list, parameter_declaration

parameter_declaration:
int *ID*

The parameters named are available as identifiers within the block of code; when the function is called, these parameters are matched with expressions in the calling code–the function call arguments–to supply their values.

3.4.1 print

There is a built in function, print, that takes a single parameter. It prints that int, plus a newline, to standard output.

3.5 Expressions

Expression precedence is the same as the order of the following subsec- tions, highest order first; the associativity of each operator is specified within the subsection.

expression:
LITERAL
variable_reference
expression + *expression*
expression - *expression*
expression * *expression*
expression / *expression*
expression == *expression*
expression != *expression*
expression < *expression*
expression > *expression*
expression < *expression*
expression >= *expression*
variable_reference = *expression*
ID(*argument_opt*)
(*expression*)

3.5.1 Function Calls

Function calls consist of an identifier followed by parentheses which contain a comma separated list of arguments to the function call. The identifier and number of arguments must match with a declared function.

When calling a function, a copy of each argument is made, and assigned to corresponding parameter in the function definition; the number of arguments must match with the previous declaration. The block of the function declaration is then executed with the copied values, with the expression evaluating to the value returned.

No check is done for whether a function actually returns a value; using the value of a function that does not explicitly return a value is undefined.

3.5.2 Variable Access

Variables can be used in expressions. An array value must be used with an index, which is any expression. No check is done to see whether the result of this expression

falls within the array bounds; accessing array values outside of the array bounds is undefined.

3.5.3 Multiplicative Operators

Multiplication and division are binary operators specified by * and /, respectively. They associate left-to-right.

3.5.4 Additive Operators

Addition and subtraction are binary operators specified by + and -, respectively. They associate left-to-right.

One quirk is that when subtracting a positive literal, a space is needed between the - and the literal; otherwise it is interpreted as a negative literal, and the operator goes missing.

3.5.5 Relational Operators

The relational operators perform numeric comparisons: <, >, <=, >=. They associate left-to-right, and return one of the specified comparison is true, 0 if it is false.

3.5.6 Equality Operators

Equality comparisons test for numeric equality, == test for equality, and != tests for inequality. They associate left-to-right, and return 1 if the relation holds, 0 if it does not.

3.5.7 Assignment

The assignment operation, =, groups right-to-left. It requires a declared identifier as its left operator (lvalue), and that identifier must not have been declared an array, or must have an index expression contained in brackets. In the expressions E1 = E2, E2 is stored into the lvaue E1, and the value of the expression is E2.

3.6 Statements

In addition to expressions, there are statements for selection—if; iteration— while; and the statement to return a value form a function. In addition, curly-braces can contain a list of statements, and be treated as a statement.

statement:
expression;
return *expression*;
{*statement_opt*}
if (*expression*) *statement*

if (*expression*) *statement* else *statement*
while (*expression*) *statement*

3.6.1 if Statement

The if statement performs selection, and takes the form:

if (*expression*) *statement*
if (*expression*) *statement* else *statement*

The else statement binds with the closest if statement. If the expression evaluates to 0, the else statement is evaluated if present; if the expression evaluates to anything else, then the if statement is executed.

3.6.2 while Statement

The while statement performs iteraion, and takes the form:

while (*expression*) *statement*

The while statement evaluates the specified expression, and if it evaluates to 0, skips its specified statement and proceeds to the subsequent statement. If the specified statement evaluates to anything but 0 then the specified statement is executed and this process is repeated; that is, the expression is evaluated again, and either the while statement is finished, or the process continues.

3.6.3 return Statement

From within a function, the return statement ends execution of the function block, returning to the calling code; the specified statement provides the return value. The return statement is required in a function that declares itself to return an int.

4 Project Plan

This project is a solo project, so all has been planned and done by Nathan Corvino. Microc was relied on a fair bit--the grammar didn't stray far from Microc, and a lot of the constructs used for compilation were gleaned from Micorc's comile.ml.

4.1 Planning, Specification, and Development, and Testing process.

The project proceeded in an iterative fashion. Once the initial grammar was reached, free of conflicts, the basic outline of the program was laid down. The starting point was acl outputting the AST. Closely thereafter, a basic assembly template was laid down to generate a file that would assemble, link and run. The final bootstrapping step was to generate function definitions, and get call implemented to the point where it could both invoke main on startup, and print could be implemented.

With print in hand, the Microc test harness was adapted to assemble, link, and run acl programs, and the hello world test was written. With a functional test in hand, individual constructs were selected and implemented in a way that seemed logical. As each construct was implemented, a specific test was added addressing that construct. Binary operators were done next, followed by local scalar variables, local array variables, and finally global variables. Next, statements were done--if and while statements. Function arguments and return values where the last feature done; it was useful to have done variables--and arrays--to better understand how the stack and addressing worked. With the major features implemented, additional checking and tests were added.

4.2 Coding Style

There were not a lot of stylistic conventions to work out, as a lot of the style was just natural in Ocaml.

A Caml mode in Emacs was used to handle indenting--with a few exceptions made when it was clearly confused by the program structure.

Line length was limitied--not to a strict 80 characters, but close. Newlines were also used to add clarity to the program structure. For instance, in cases where a list was being constructed that spanned multiple lines, elements were place on individual lines.

Names for small functions where kept short, usually one character. Other names of functions and variables were lowercase with underscores for word separators--although in a few cases, singe character prefixes where used without an underscore.

Comments were used in cases where the intent or code needed clarification; in general, function and variable names were chosen to be as self-documenting as possible.

4.3 Development Tools

Linux was chosen as the target platform for assembly language, in part due to the availability of references for it, and also as it was sure to be available for testing. Some investigation was done on the Mac, but the assembly was found to be different enough--OS X has some specific alignment requirements, for instance--that this was not pursued fully.

Ubuntu Linux was used as the target OS, and Git was used for source control. OCaml was used as the implementation language, with Ocamllex and Ocamlyacc handling scanning and parsing.

Make was used for building the compiler, although running unit tests were not added to the Makefile. In part, this was done because the Ocaml source was moved between a host Mac and a Ubuntu virtual machine for testing.

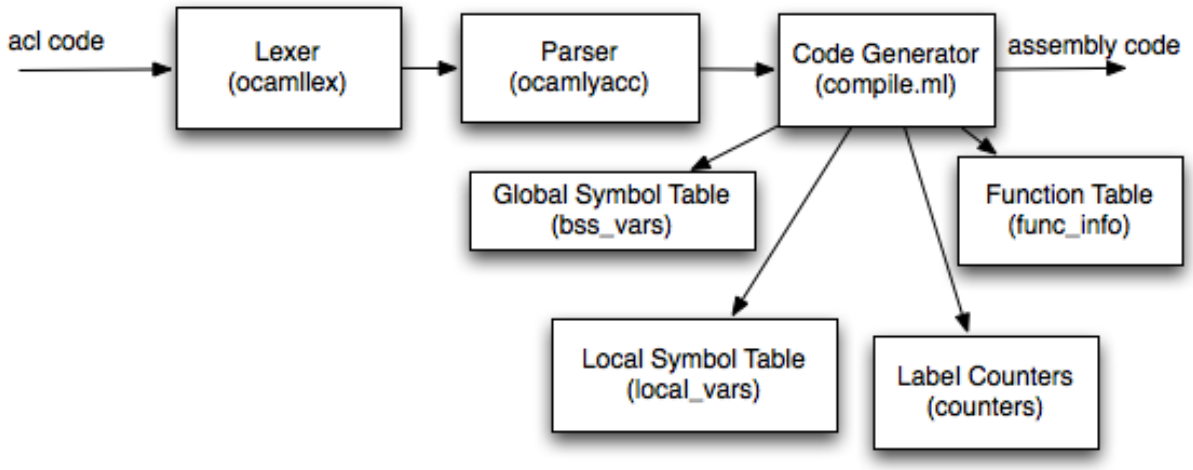
4.4 Project Log

3-17-2011	Work on lexer and parser began, in preparation for language reference manual.
4-8-2011	Assembly language investigation completed. Use of Ubuntu settled, and a good portion of Professional Assembly Language, by Richard Blum, completed.
4-30-2011	Basic assembly shell completed.
5-9-2011	Tests, functions, and variables completed.
5-10-2011	Expressions and statements completed.
5-11-2011	Function arguments, additional checking completed.
5-12-2011	Additional tests and polish.

5 Architectural Design

5.1 Components

There were three major pieces to the compiler--the lexer, the parser, and the code generator. The lexer and parser were handled with OCamllex and OCamlyacc, respectively; the most work was done on the code generator. The code generator used several structures to aid in the checking and generating code: symbol tables for local and global data; a table of functions for checking that parameters align; and a table containing mutable counters to aid in generating unique labels for control structure code.



The HashTable, counters, contains two integer values, which code for generating if and while statements append to labels to generate unique labels for the code generate by each control structure.

The global variables are folded into a Map keyed by strings, bss_vars, which records their offsets from the label for the bss section. The same function folds the local_vars of each function in the Map keyed by strings, allowing their offsets to be reached; but first, the function parameters are folded into the map, using the same logic, but with a different starting point and stride to the offset, pointing them up the stack where the calling function places them.

The function table stores the number of parameters for each function, allowing easy checking when generating code for a call that the number of parameters is the same. There is also a index assigned to the functions, which could be used for generating a label to jump to exit code, even though return was not implemented this way.

6 Test Plan

The test suite was an important part of this project. For one thing, the build-and-test process involved several steps--each sample program had to be compiled, assembled, linked and run. Further, the test suite allowed regressions to be caught immediately, saving time tracking them down.

6.1 Examples

The prime number checker from the tutorial is a good sample program:

```
prime(int a) {
      int i;
      int multiplier;
      int result;

      i = a / 2;

      while(i > 2) {
            multiplier = a / i;
            result = i * multiplier;

            if (result == a) {
                  return 0;
            }

            i = i - 1;
      }

      return 1;
}

main()
{
      print(prime(-7));
      print(prime(37));
      print(prime(42));
      print(prime(45));
}
```

It generates:

```
.section .data
output:
    .asciz "%d\n"
.section .text
.globl _start
_start:
    call main
    pushl $0
    call exit

.type main, @function
main:
```

```
    pushl %ebp
    movl %esp, %ebp
    addl $0, %esp
    movl $-7, %eax
    pushl %eax
    call prime
    addl $4, %esp
    pushl %eax
    movl $output, %eax
    pushl %eax
    call printf
    addl $8, %esp
    movl $37, %eax
    pushl %eax
    call prime
    addl $4, %esp
    pushl %eax
    movl $output, %eax
    pushl %eax
    call printf
    addl $8, %esp
    movl $42, %eax
    pushl %eax
    call prime
    addl $4, %esp
    pushl %eax
    movl $output, %eax
    pushl %eax
    call printf
    addl $8, %esp
    movl $45, %eax
    pushl %eax
    call prime
    addl $4, %esp
    pushl %eax
    movl $output, %eax
    pushl %eax
    call printf
    addl $8, %esp
    movl %ebp, %esp
    popl %ebp
    ret
.type prime, @function
prime:
    pushl %ebp
    movl %esp, %ebp
    addl $-12, %esp
    movl $2, %eax
    pushl %eax
    movl 8(%ebp), %eax
    popl %ebx
    movl %eax, %edx
    sarl $31, %edx
    idivl %ebx
    movl %eax, -4(%ebp)
loop_label_0:
    movl $2, %eax
    pushl %eax
```

```
    movl -4(%ebp), %eax
    popl %ebx
    cmp %ebx, %eax
    movl $0, %eax
    movl $1, %ebx
    cmovg %ebx, %eax
    cmp $0, %eax
    je loop_label_1
    movl -4(%ebp), %eax
    pushl %eax
    movl 8(%ebp), %eax
    popl %ebx
    movl %eax, %edx
    sarl $31, %edx
    idivl %ebx
    movl %eax, -8(%ebp)
    movl -8(%ebp), %eax
    pushl %eax
    movl -4(%ebp), %eax
    popl %ebx
    imul %ebx, %eax
    movl %eax, -12(%ebp)
    movl 8(%ebp), %eax
    pushl %eax
    movl -12(%ebp), %eax
    popl %ebx
    cmp %ebx, %eax
    movl $0, %eax
    movl $1, %ebx
    cmove %ebx, %eax
    cmp $0, %eax
    je cond_label_0
    movl $0, %eax
    movl %ebp, %esp
    popl %ebp
    ret
    jmp cond_label_1
cond_label_0:
cond_label_1:
    movl $1, %eax
    pushl %eax
    movl -4(%ebp), %eax
    popl %ebx
    subl %ebx, %eax
    movl %eax, -4(%ebp)
    jmp loop_label_0
loop_label_1:
    movl $1, %eax
    movl %ebp, %esp
    popl %ebp
    ret
    movl %ebp, %esp
    popl %ebp
    ret
```

Another good example is gcd; it is perfect for this language, and was a frequent example from class.

```
gcd(int a, int b) {
  while (a != b) {
    if (a > b) a = a - b;
    else b = b - a;
  }

  return a;
}

main()
{
  print(gcd(14,49));
  print(gcd(9,30));
  print(gcd(289,561));
  print(gcd(37,1369));
}
```

Which generates:

```
.section .data
output:
    .asciz "%d\n"
.section .text
.globl _start
_start:
    call main
    pushl $0
    call exit

.type main, @function
main:
    pushl %ebp
    movl %esp, %ebp
    addl $0, %esp
    movl $49, %eax
    pushl %eax
    movl $14, %eax
    pushl %eax
    call gcd
    addl $8, %esp
    pushl %eax
    movl $output, %eax
    pushl %eax
    call printf
    addl $8, %esp
    movl $30, %eax
    pushl %eax
    movl $9, %eax
    pushl %eax
    call gcd
    addl $8, %esp
    pushl %eax
    movl $output, %eax
    pushl %eax
    call printf
    addl $8, %esp
```

```
    movl $561, %eax
    pushl %eax
    movl $289, %eax
    pushl %eax
    call gcd
    addl $8, %esp
    pushl %eax
    movl $output, %eax
    pushl %eax
    call printf
    addl $8, %esp
    movl $1369, %eax
    pushl %eax
    movl $37, %eax
    pushl %eax
    call gcd
    addl $8, %esp
    pushl %eax
    movl $output, %eax
    pushl %eax
    call printf
    addl $8, %esp
    movl %ebp, %esp
    popl %ebp
    ret
.type gcd, @function
gcd:
    pushl %ebp
    movl %esp, %ebp
    addl $0, %esp
loop_label_0:
    movl 12(%ebp), %eax
    pushl %eax
    movl 8(%ebp), %eax
    popl %ebx
    cmp %ebx, %eax
    movl $0, %eax
    movl $1, %ebx
    cmovne %ebx, %eax
    cmp $0, %eax
    je loop_label_1
    movl 12(%ebp), %eax
    pushl %eax
    movl 8(%ebp), %eax
    popl %ebx
    cmp %ebx, %eax
    movl $0, %eax
    movl $1, %ebx
    cmovg %ebx, %eax
    cmp $0, %eax
    je cond_label_0
    movl 12(%ebp), %eax
    pushl %eax
    movl 8(%ebp), %eax
    popl %ebx
    subl %ebx, %eax
    movl %eax, 8(%ebp)
    jmp cond_label_1
```

```
cond_label_0:
    movl 8(%ebp), %eax
    pushl %eax
    movl 12(%ebp), %eax
    popl %ebx
    subl %ebx, %eax
    movl %eax, 12(%ebp)
cond_label_1:
    jmp loop_label_0
loop_label_1:
    movl 8(%ebp), %eax
    movl %ebp, %esp
    popl %ebp
    ret
    movl %ebp, %esp
    popl %ebp
    ret
```

6.2 Automation

The automation was based closely on microc; the testall.sh shell script was adapted to assemble, link, and run the assembly programs produced by acl.

```
#!/bin/sh

ACL="./acl"

# Set time limit for all operations
ulimit -t 30

globallog=testall.log
rm -f $globallog
error=0
globalerror=0

keep=0

Usage() {
    echo "Usage: testall.sh [options] [.mc files]"
    echo "-k    Keep intermediate files"
    echo "-h    Print this help"
    exit 1
}

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 -U3 "$1" "$2" > "$3" 2>&1 || {
      SignalError "$1 differs"
      echo "FAILED $1 differs from $2" 1>&2
    }
}

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

Check() {
    error=0
    basename=`echo $1 | sed 's/.*\\///
                              s/.acl//'`
    reffile=`echo $1 | sed 's/.acl$//'`
    basedir="`echo $1 | sed 's/\/[^\/]*$//'`/."

    echo -n "$basename..."

    echo 1>&2
    echo "###### Testing $basename" 1>&2

    generatedfiles=""

    generatedfiles="$generatedfiles ${basename}.s ${basename}.o $
{basename} ${basename}.test.out" &&

    Run "$ACL" "-c" "<" $1 ">" ${basename}.s &&
    Run "as" "-o" ${basename}.o ${basename}.s &&
    Run "ld" "-dynamic-linker /lib/ld-linux.so.2" "-o" ${basename} -lc
${basename}.o
    Run "./${basename}" ">" ${basename}.test.out &&
    Compare ${basename}.test.out ${reffile}.out ${basename}.test.diff

    # Report the status and clean up the generated files

    if [ $error -eq 0 ] ; then
      if [ $keep -eq 0 ] ; then
          rm -f $generatedfiles
      fi
      echo "OK"
      echo "###### SUCCESS" 1>&2
    else
      echo "###### FAILED" 1>&2
      globalerror=$error
    fi
}

while getopts kdpsh c; do
    case $c in
      k) # Keep intermediate files
```

```
            keep=1
            ;;
        h) # Help
            Usage
            ;;
    esac
done

shift `expr $OPTIND - 1`

if [ $# -ge 1 ]
then
    files=$@
else
    files="tests/fail-*.acl tests/test-*.acl"
fi

for file in $files
do
    case $file in
      *test-*)
            Check $file 2>> $globallog
            ;;
      *fail-*)
            CheckFail $file 2>> $globallog
            ;;
      *)
            echo "unknown file type $file"
            globalerror=1
            ;;
    esac
done

exit $globalerror
```

This proved to work quite well.

6.3 Test Suite

The full test suite consists of targets tests developed with each chunk of functionality, as well as several broader programs added at the end. The suite follows, in the order they were developed:

test-hello.acl:

```
main()
{
  print(37);
  print(1369);
  print(1);
}
```

test-arith-expressions.acl:

```
main()
{
  print(3 + 7);
  print(3 - 7);
  print(3 * 7);
  print(7 / 3);

  /* 33 */
  print(4 * 2 + 37 - 7 / 3 - (3 + 7));

  /* 9 */
  print(4 * 12 - 3 * 9 - 180 / 15);
}
```

test-cond-expressions.acl:

```
main()
{
  print(2 == 2);
  print(2 == 3);

  print(2 != 3);
  print(2 != 2);

  print(2 < 3);
  print(3 < 2);

  print(3 > 2);
  print(2 > 3);

  print(3 >= 3);
  print(3 > 3);

  print(3 <= 3);
  print(3 < 3);

  print(2 <= 3);
  print(3 <= 2);

  print(3 >= 2);
  print(2 >= 3);
}
```

test-local-variables.acl:

```
main()
{
        int a;
        int b;
        int c[4];
        int d;

        a = 37;
        b = 1369;
        c[0] = 10;
        c[1] = 11;
```

```
        c[2] = 12;
        c[3] = 13;
        d = 42;

        print(a);
        print(b);
        print(c[0]);
        print(c[1]);
        print(c[2]);
        print(c[3]);
        print(d);
}
```

test-global-variables.acl:

```
int a;
int b;
int c[4];
int d;

main()

{
        a = 37;
        b = 1369;
        c[0] = 10;
        c[1] = 11;
        c[2] = 12;
        c[3] = 13;
        d = 42;

        print(a);
        print(b);
        print(c[0]);
        print(c[1]);
        print(c[2]);
        print(c[3]);
        print(d);
}
```

test-if.acl:

```
main()
{
        if (1)
                print(1);
        else
                print(0);

        if (0)
                print(1);
        else
                print(0);

        if (1)
                print(1);
```

```
        if (0)
                print(0);
}
```

test-while.acl:

```
main()

{
        int a;

        a = 0;

        while (a < 10) {
                print(a);
                a = a + 1;
        }
}
```

test-functions.acl:

```
first(int a, int b) {
        return a + b;
}

second(int a, int b) {
        int c;

        c = a * b;
        print(c);
}

third(int a, int b, int c) {
        return a - b - c;
}

main()
{
        print(first(3, 7));
        second(3, 7);
        print(third(37, 3, 7));
}
```

test-gcd.acl:

```
gcd(int a, int b) {
  while (a != b) {
    if (a > b) a = a - b;
    else b = b - a;
  }

  return a;
}

main()
```

```
{
  print(gcd(14,49));
  print(gcd(9,30));
  print(gcd(289,561));
  print(gcd(37,1369));
}
```

test-prime.acl:

```
prime(int a) {
        int i;
        int multiplier;
        int result;

        i = a / 2;

        while(i > 2) {
                multiplier = a / i;
                result = i * multiplier;

                if (result == a) {
                        return 0;
                }

                i = i - 1;
        }

        return 1;
}

main()
{
        print(prime(-7));
        print(prime(37));
        print(prime(42));
        print(prime(45));
}
```

test-fib.acl:

```
fib(int x)
{
  if (x < 2)
          return 1;

  return fib(x - 1) + fib(x - 2);
}

main()
{
  print(fib(0));
  print(fib(1));
  print(fib(2));
  print(fib(3));
  print(fib(4));
  print(fib(5));
  print(fib(6));
```

```
  print(fib(7));
}
```

7 Lessons Learned

I feel like I learned a lot from this project, although time will tell. Most importantly, I have a better understanding of how the higher level programming languages I regularly work with generate specific of machine code.

I don't understand how that code gets optimized at all. Some of the most obvious optimizations--adding a literal without moving it into a register, something I did in the very early stages of my development--got sacrificed to get a generic program working. While I did generate a data structure, that then got turned into the string of the assembly program, it is not anything like a generic three-op code. I am interested spending some time thinking about this, and maybe extending my project a bit, after I've had more time with the Dragon Book.

One lesson I would potentially pass on is to not get to hung up laboring over every word of the Dragon Book. I spent a large amount of time with the book--I found it pushing me a lot, but also a lot of it to be very slow to read. I wish I had focused more on the core pieces--fundamentals of parsing, for instance--without spending as much time on recursive-descent parsing or LR(1) automatons. This would have allowed me to spend more time in the intermediate-code and code generation chapters.

I would be remiss if I did not mention start earlier. Its not really a lesson learned-I knew this at the start. Still, I wish I had time to investigate optimizations.

A Grammar

program:
    program variable_declaration
    program function_definition

function_definition:
  ID(*parameter_opt*) {*declaration_opt statement_opt*}

parameter_opt:
    parameter_list$_{opt}$

parameter_list:
    parameter_declaration
    parameter_list, parameter_declaration

parameter_declaration:
    int *ID*

variable_declaration:
    int *ID*;
    int *ID*[*LITERAL*];

variable_reference:
    ID
    ID[*expression*]

declaration_opt:
    declaration_list$_{opt}$

declaration_list:
    variable_declaration
    declaration_list variable_declaration

statement_opt:
    statement_list$_{opt}$

statement_list:
    statement
    statement_list statement

statement:
    expression;
    return *expression*;

{*statement_opt*}
if (*expression*) *statement*
if (*expression*) *statement* else *statement*
while (*expression*) *statement*

expression:
LITERAL
variable_reference
expression + *expression*
expression - *expression*
expression * *expression*
expression / *expression*
expression == *expression*
expression != *expression*
expression < *expression*
expression > *expression*
expression < *expression*
expression >= *expression*
variable_reference = *expression*
ID(*argument_opt*)
(*expression*)

argument_opt:
$argument\_list_{opt}$

argument_list:
expression
argument_list, expression

B Code Listing

B.1 scanner.ml

```
{ open Parser }

rule token = parse
  [' ' '\t' '\r' '\n'] { token lexbuf }
| "/*"                 { comment lexbuf }

| ";"                  { SEMICOLON }
| ","                  { COMMA }

| "("                  { LEFT_PAREN }
| ")"                  { RIGHT_PAREN }
| "{"                  { LEFT_BRACE }
| "}"                  { RIGHT_BRACE }
| "["                  { LEFT_BRACKET }
| "]"                  { RIGHT_BRACKET }

| "+"                  { PLUS }
| "-"                  { MINUS }
| "*"                  { TIMES }
| "/"                  { DIVIDE }
| "="                  { ASSIGN }

| "=="                 { EQUAL }
| "!="                 { NOT_EQUAL }
| "<"                  { LESS_THAN }
| ">"                  { GREATER_THAN }
| "<="                 { LESS_EQUAL }
| ">="                 { GREATER_EQUAL }

| "if"                 { IF }
| "else"               { ELSE }
| "while"              { WHILE }

| "int"                { INT }

| "return"             { RETURN }

| eof                  { EOF }

| '-'?['0'-'9']+ as lit    { LITERAL(int_of_string lit) }
| ['a'-'z' 'A'-'Z']['a'-'z' 'A'-'Z' '0'-'9' '_']* as lit  { ID(lit) }

| _ as char     { raise (Failure("illegal character " ^ Char.escaped char)) }

and comment = parse
  "*/"         { token lexbuf }
| _            { comment lexbuf }

and line_comment = parse
  ['\r' '\n']  { token lexbuf }
| _            { comment lexbuf }
```

B.2 parser.mly

```
%{ open Ast %}

%token SEMICOLON COMMA
%token LEFT_PAREN RIGHT_PAREN LEFT_BRACE RIGHT_BRACE LEFT_BRACKET
RIGHT_BRACKET
%token PLUS MINUS TIMES DIVIDE ASSIGN
%token EQUAL NOT_EQUAL LESS_THAN GREATER_THAN LESS_EQUAL GREATER_EQUAL
%token RETURN IF ELSE WHILE INT VOID
%token <int> LITERAL
%token <string> ID
%token EOF

%nonassoc NOELSE
%nonassoc ELSE
%right ASSIGN
%left EQUAL NOT_EQUAL
%left LESS_THAN GREATER_THAN LESS_EQUAL GREATER_EQUAL
%left PLUS MINUS
%left TIMES DIVIDE

%start program
%type <Ast.program> program

%%

program:
    /* nothing */     { [], [] }
  | program variable_declaration  { ($2 :: fst $1), snd $1 }
  | program function_definition   { fst $1, ($2 :: snd $1) }


function_definition:
    ID LEFT_PAREN parameter_opt RIGHT_PAREN
      LEFT_BRACE declaration_opt statement_opt RIGHT_BRACE
      { {
        fname = $1;
        formals = $3;
        locals = $6;
        body = $7;
      } }

parameter_opt:
    /* nothing */    { [] }
  | parameter_list   { List.rev $1 }

parameter_list:
    parameter_declaration                          { [$1] }
  | parameter_list COMMA parameter_declaration   { $3 :: $1 }

parameter_declaration:
    INT ID    { Int($2) }

variable_declaration:
    INT ID SEMICOLON                                    { Int($2) }
  | INT ID LEFT_BRACKET LITERAL RIGHT_BRACKET SEMICOLON   { Array($2, $4) }
```

```
variable_reference:
    ID                                          { IntRef($1) }
  | ID LEFT_BRACKET expression RIGHT_BRACKET { ArrayRef($1, $3) }

declaration_opt:
    /* nothing */     { [] }
  | declaration_list  { List.rev $1 }

declaration_list:
    variable_declaration                    { [$1] }
  | declaration_list variable_declaration   { $2 :: $1 }

statement_opt:
    /* nothing */   { [] }
  | statement_list  { List.rev $1 }

statement_list:
    statement                { [$1] }
  | statement_list statement  { $2 :: $1 }

statement:
    expression SEMICOLON { Expression($1) }
  | RETURN expression SEMICOLON { Return($2) }
  | LEFT_BRACE statement_opt RIGHT_BRACE { Block($2) }
  | IF LEFT_PAREN expression RIGHT_PAREN statement %prec NOELSE { If($3, $5,
Block([])) }
  | IF LEFT_PAREN expression RIGHT_PAREN statement ELSE statement { If($3,
$5, $7) }
  | WHILE LEFT_PAREN expression RIGHT_PAREN statement { While($3, $5) }

expression:
    LITERAL                    { Literal($1) }
  | variable_reference         { VarRef($1) }
  | expression PLUS expression { Binop($1, Add, $3) }
  | expression MINUS expression { Binop($1, Sub, $3) }
  | expression TIMES expression { Binop($1, Mult, $3) }
  | expression DIVIDE expression { Binop($1, Div, $3) }
  | expression EQUAL expression { Binop($1, Equal, $3) }
  | expression NOT_EQUAL expression { Binop($1, NotEqual, $3) }
  | expression LESS_THAN expression { Binop($1, LessThan, $3) }
  | expression GREATER_THAN expression { Binop($1, GreaterThan, $3) }
  | expression LESS_EQUAL expression { Binop($1, LessEqual, $3) }
  | expression GREATER_EQUAL expression { Binop($1, GreaterEqual, $3) }
  | variable_reference ASSIGN expression { Assign($1, $3) }
  | ID LEFT_PAREN argument_opt RIGHT_PAREN
        { match $1 with
          "print" -> Call("printf", Data("$output") :: $3)
        | _ -> Call($1, $3) }
  | LEFT_PAREN expression RIGHT_PAREN { $2 }

argument_opt:
    /* nothing */  { [] }
  | argument_list  { List.rev $1 }

argument_list:
    expression                    { [$1] }
  | argument_list COMMA expression { $3 :: $1 }
```

B.3 compile.ml

```
open Ast

module StringMap = Map.Make(String)
module CounterHash = Hashtbl.Make(struct
  type t = string
  let equal x y = x = y
  let hash = Hashtbl.hash
end)

type function_info = {
      function_index : int;
      parameter_count : int;
  }

type condition = CGreaterThan | CLessThan | CGreaterEqual | CLessEqual |
CEqual | CNotEqual

type var_type = IntType | ArrayType

type instruction =
      Label of string
  |   Move of string * string
  | CondMove of condition * string * string
  | Jump of string
  | CondJump of string
  |   Push of string
  | Pop of string
  | Invoke of string
  | AddLiteral of int * string
  | AddValues of string * string
  | SubValues of string * string
  | MulValues of string * string
  | DivValues of string
  | Sarl of string * string
  | Cmp of string * string
  | FuncHeader of string
  | Ret
  | None

let string_of_asm = function
      Label(s) -> s ^ ":\n"
  | Move(s, t) -> "    movl " ^ s ^ ", " ^ t ^ "\n"
  | CondMove(c, s, t) -> "    " ^
      (match c with
        CGreaterThan -> "cmovg"
      | CLessThan -> "cmovl"
      | CGreaterEqual -> "cmovge"
      | CLessEqual -> "cmovle"
      | CEqual -> "cmove"
      | CNotEqual -> "cmovne")
      ^ " " ^ s ^ ", " ^ t ^ "\n"
  | Jump(s) -> "    jmp " ^ s ^ "\n"
  | CondJump(s) -> "    je " ^ s ^ "\n"
  |   Push(a) -> "    pushl " ^ a ^ "\n"
  | Pop(a) -> "    popl " ^ a ^ "\n"
  | Invoke(n) -> "    call " ^ n ^ "\n"
```

```
  | AddLiteral(l, a) -> "    addl $" ^ string_of_int l ^ ", %" ^ a ^ "\n"
  | AddValues(s, d) -> "    addl " ^ s ^ ", " ^ d ^ "\n"
  | SubValues(s, d) -> "    subl " ^ s ^ ", " ^ d ^ "\n"
  | MulValues(s, d) -> "    imul " ^ s ^ ", " ^ d ^ "\n"
  | DivValues(s) -> "    idivl " ^ s ^ "\n"
  | Sarl(b, t) -> "    sarl " ^ b ^ ", " ^ t ^ "\n"
  | Cmp(s, d) -> "    cmp " ^ s ^ ", " ^ d ^ "\n"
  | FuncHeader(n) -> ".type " ^ n ^ ", @function\n" ^ n ^ ":\n"
  | Ret -> "    ret\n"
  | None -> ""

let accumulate_statements a statement_list =
  a @ statement_list

let translate (globals, functions) =
  let counters = CounterHash.create 1 in
  CounterHash.add counters "loop_label" 0;
  CounterHash.add counters "cond_label" 0;
  let alloc_functions (functions, count) fdecl =
      if StringMap.mem fdecl.fname functions
      then raise (Failure (fdecl.fname ^ " used twice as a function name."))
      else (StringMap.add
                  fdecl.fname
                  { function_index = count; parameter_count = List.length
fdecl.formals }
                  functions,
              count + 1) in
  let (func_info, _) =
      List.fold_left alloc_functions
        (StringMap.add "printf" { function_index = -1; parameter_count = 2 }
             StringMap.empty, 0)
        functions in
  let alloc_vars start stride (vars, count) = function
        Int(n) -> if StringMap.mem n vars
              then raise (Failure (n ^ " used as a variable twice in the same
scope."))
              else (StringMap.add n (IntType, (count * stride) + start) vars,
count + 1)
      | Array(n, l) -> if StringMap.mem n vars
              then raise (Failure (n ^ " used as a variable twice in the same
scope."))
              else if l < 1 then raise (Failure ("Arrays must be of size
greater than 1."))
              else (StringMap.add n (ArrayType, (count * stride) + start)
vars, count + l) in
  let (bss_vars, bss_words) =
      List.fold_left (alloc_vars 1 1) (StringMap.empty, 0) globals in
  let translate_cond_expression cond =
      [ Cmp("%ebx", "%eax");
        Move("$0", "%eax");
        Move("$1", "%ebx");
        CondMove(cond, "%ebx", "%eax") ] in
  let rec translate_expr env = function
      Literal(l) -> [ Move("$" ^ string_of_int l, "%eax") ]
    | VarRef(v) ->
          (match v with
            IntRef(s) ->
                (try let (t, o) = (StringMap.find s env) in
```

```
                  if ArrayType = t then
                    raise (Failure ("attempt to use array variable " ^ s ^ "
as scalar"))
                  else [ Move(string_of_int (-4 * o) ^ "(%ebp)", "%eax") ]
                  with Not_found ->
                    (try let (t, o) = (StringMap.find s bss_vars) in
                    if ArrayType = t then
                        raise (Failure ("attempt to use array variable " ^ s
^ " as scalar"))
                    else [ Move("$global_buffer", "%edi");
                              Move(string_of_int (4 * o) ^ "(%edi)",
"%eax") ]
                    with Not_found -> raise (Failure ("undefined variable " ^
s))))
              | ArrayRef(s, e) ->
                  (try let (t, o) = (StringMap.find s env) in
                  if IntType = t then
                    raise (Failure ("attempt to use scalar variable " ^ s ^ "
as an array"))
                  else (translate_expr env e) @
                    [ MulValues("$-4", "%eax");
                        AddValues("%ebp", "%eax");
                        AddValues("$" ^ string_of_int (-4 * o), "%eax");
                        Move("%eax", "%ebx");
                        Move("0(%ebx)", "%eax"); ]
                  with Not_found ->
                    (try let (t, o) = (StringMap.find s bss_vars) in
                    if IntType = t then
                        raise (Failure ("attempt to use scalar variable " ^ s
^ " as an array"))
                    else
                        (translate_expr env e) @
                        [ MulValues("$4", "%eax");
                          Move("$global_buffer", "%ebx");
                          AddValues("%ebx", "%eax");
                          AddValues("$" ^ string_of_int (4 * o), "%eax");
                          Move("%eax", "%ebx");
                          Move("0(%ebx)", "%eax"); ]
                    with Not_found -> raise (Failure ("undefined variable " ^
s)))))
    | Binop(e1, o, e2) ->
        (translate_expr env e2) @
        Push("%eax") ::
        (translate_expr env e1) @
        Pop("%ebx") ::
        (match o with
          Add -> [ AddValues("%ebx", "%eax") ]
        | Sub -> [ SubValues("%ebx", "%eax") ]
        | Mult -> [ MulValues("%ebx", "%eax") ]
        | Div -> [ Move("%eax", "%edx"); Sarl("$31", "%edx"); DivValues
("%ebx") ]
        | Equal -> translate_cond_expression CEqual
        | NotEqual -> translate_cond_expression CNotEqual
        | LessThan -> translate_cond_expression CLessThan
        | GreaterThan -> translate_cond_expression CGreaterThan
        | LessEqual -> translate_cond_expression CLessEqual
        | GreaterEqual -> translate_cond_expression CGreaterEqual)
    | Assign(v, e) ->
```

```
            (translate_expr env e) @
            (match v with
              IntRef(s) ->
                  (try let (t, o) = (StringMap.find s env) in
                  if ArrayType = t then
                    raise (Failure ("attempt to use array variable " ^ s ^ "
as a scalar"))
                  else [ Move("%eax", string_of_int (-4 * o) ^ "(%ebp)") ]
                  with Not_found ->
                    (try let (t, o) = (StringMap.find s bss_vars) in
                    if ArrayType = t then
                        raise (Failure ("attempt to use array variable " ^ s
^ " as scalar"))
                    else [ Move("$global_buffer", "%edi");
                              Move("%eax", string_of_int (4 * o) ^
"(%edi)" ) ]
                    with Not_found -> raise (Failure ("undefined variable " ^
s))))
            | ArrayRef(s, e) ->
                  Push("%eax") :: (translate_expr env e) @
                  (try let (t, o) = (StringMap.find s env) in
                  if IntType = t then
                    raise (Failure ("attempt to use scalar variable " ^ s ^ "
as an array"))
                  else
                    [ MulValues("$-4", "%eax");
                        AddValues("%ebp", "%eax");
                        AddValues("$" ^ string_of_int (-4 * o), "%eax");
                        Move("%eax", "%ebx");
                        Pop("%eax");
                        Move("%eax", "0(%ebx)"); ]
                  with Not_found ->
                    (try let (t, o) = (StringMap.find s bss_vars) in
                    if IntType = t then
                        raise (Failure ("attempt to use scalar variable " ^ s
^ " as an array"))
                    else
                        [ MulValues("$4", "%eax");
                          Move("$global_buffer", "%ebx");
                          AddValues("%ebx", "%eax");
                          AddValues("$" ^ string_of_int (4 * o), "%eax");
                          Move("%eax", "%ebx");
                          Pop("%eax");
                          Move("%eax", "0(%ebx)"); ]
                    with Not_found -> raise (Failure ("undefined variable " ^
s)))))
    | Call(f, el) ->
            (try let finfo = StringMap.find f func_info in
            if finfo.parameter_count == List.length el then
              (* Evaluate and push the arguments, and do it in reverse order.
*)
              List.fold_left accumulate_statements []
                  (List.map (fun exp -> exp @ [Push("%eax")])
                     (List.map (translate_expr env) (List.rev el))) @
              (* Call the function. *)
              [ Invoke(f);
                  (* Pop the stack for the number of arguments pushed. *)
                  AddLiteral(4 * List.length el, "esp"); ]
```

```
            else raise (Failure ("call to function " ^ f ^ " which takes " ^
                                                 string_of_int
finfo.parameter_count ^ " args with " ^
                                                 string_of_int (List.length el) ^ "
args"))
            with Not_found -> raise (Failure ("call to undeclared function "
^ f)))
      | Data(s) -> [ Move(s, "%eax") ]
    | Noexpr -> [] in
  let rec translate_stmt env = function
      Block(stmts) -> translate_stmts env stmts
    | Expression(expr) -> translate_expr env expr
    | Return(expr) ->
            translate_expr env expr @
            [ Move("%ebp", "%esp"); Pop("%ebp"); Ret ]
    | If(e, s1, s2) ->
            let num = CounterHash.find counters "cond_label" in
            let false_label = "cond_label_" ^ string_of_int num in
            let true_label = "cond_label_" ^ string_of_int (num + 1) in
            CounterHash.replace counters "cond_label" (num + 2);
            translate_expr env e @
            [ Cmp("$0", "%eax") ;
              CondJump(false_label); ] @
            translate_stmt env s1 @
            Jump(true_label) ::
            Label(false_label) ::
            translate_stmt env s2 @
            [ Label(true_label) ]
    | While(e, s) ->
            let num = CounterHash.find counters "loop_label" in
            let start_label = "loop_label_" ^ string_of_int num in
            let done_label = "loop_label_" ^ string_of_int (num + 1) in
            CounterHash.replace counters "loop_label" (num + 2);
            Label(start_label) ::
            translate_expr env e @
            [ Cmp("$0", "%eax");
              CondJump(done_label) ] @
            translate_stmt env s @
            [ Jump(start_label);
              Label(done_label) ]
  and translate_stmts env stmts =
      List.fold_left (fun a l -> a @ l) [] (List.map (translate_stmt env)
stmts) in
  let translate_fdecl fdecl =
      let (parameter_vars, _) =
        List.fold_left (alloc_vars (-2) (-1)) (StringMap.empty, 0)
fdecl.formals in
      let (local_vars, local_words) =
        List.fold_left (alloc_vars 1 1) (parameter_vars, 0) fdecl.locals in
      FuncHeader(fdecl.fname) ::
      (* Save the stack pointer in %ebp, for accessing data, and preserve the
old
         value of %ebp *)
      Push("%ebp") :: Move("%esp", "%ebp") ::
      (* Add space for local vars. *)
      AddLiteral(-4 * local_words, "esp") ::
      translate_stmts local_vars fdecl.body @
    (* Restore the %ebp and stack pointer, then return. *)
```

```
  [ Move("%ebp", "%esp"); Pop("%ebp"); Ret ] in
let func_defs = List.map translate_fdecl functions in
let asm = List.fold_left accumulate_statements [] func_defs in
let text = List.map string_of_asm asm in
".section .data\n" ^
"output:\n" ^
"    .asciz \"%d\\n\"\n" ^
(if bss_words > 0 then
    ".section .bss\n" ^
    "    .lcomm global_buffer, " ^ string_of_int (4 * bss_words) ^ "\n"
else
    "") ^
".section .text\n" ^
".globl _start\n" ^
"_start:\n" ^
"    call main\n" ^
"    pushl $0\n" ^
"    call exit\n\n" ^
String.concat "" text
```

B.4 acl.ml

```
type action = Ast | Compile

let _ =
  let action = if Array.length Sys.argv > 1 then
    List.assoc Sys.argv.(1) [ ("-a", Ast);
                              ("-c", Compile) ]

  else Compile in

  let lexbuf = Lexing.from_channel stdin in
  let program = Parser.program Scanner.token lexbuf in

  match action with
    Ast -> let listing = Ast.string_of_program program
        in print_string listing
  | Compile -> print_string (Compile.translate program)
```

B.5 Makefile

```
OBJS = ast.cmo parser.cmo scanner.cmo compile.cmo acl.cmo

all: acl

.PHONY: clean
clean:
      $(RM) acl parser.ml parser.mli scanner.ml *.cmo *.cmi


scanner.ml: scanner.mll
      ocamllex scanner.mll

parser.ml parser.mli: parser.mly
      ocamlyacc parser.mly

acl: $(OBJS)
      ocamlc -o acl $(OBJS)

%.cmo: %.ml
      ocamlc -c $<
%.cmi: %.mli
      ocamlc -c $<

# Generated by ocamldep *.ml *.mli
acl.cmo: scanner.cmo parser.cmi compile.cmo ast.cmo
acl.cmx: scanner.cmx parser.cmx compile.cmx ast.cmx
ast.cmo:
ast.cmx:
compile.cmo: ast.cmo
compile.cmx: ast.cmx
parser.cmo: ast.cmo parser.cmi
parser.cmx: ast.cmx parser.cmi
scanner.cmo: parser.cmi
scanner.cmx: parser.cmx
parser.cmi: ast.cmo
```