COMS W4115:
Programming Languages and Translators

GPA: General Purpose to Assembly Language
12/18/2006

mbp2103 : Michael Pierorazio, Project Manager
dg2267 : Dmitry Gimzelberg, Systems Architect
vaz2001 : Vincenzo Zarrillo, Systems Integrator
jmg2105: Juan Gutierrez, Quality Assurance

GPA:
General Purpose to Assembly Language

Introduction

GPA is a light weight, general purpose language whose design incorporates the low-level understanding of assembly language while allowing the use of high level constructs such as functions and loops. The premise behind GPA comes from programmers who are used to working close to hardware but require some high level constructs, and as a result, GPA falls somewhere between assembly code and the C language in terms of abstraction. GPA is simple to learn for those who know another language, but due to its high level syntax, it is also easy for beginners to understand. The goal of GPA is to be simple, powerful, and fast.

Features

Since GPA is designed to be simple for beginners, it has no explicit types. Those with no experience in programming languages are often confused by data types so the simple construct for variable declaration is ‘**var** x’. GPA is also capable of recursion.

Goals

Since this language is more low-level than C, it can translate almost one-to-one with assembly code blocks. This will enable GPA to be compiled easily to x86 assembly code. This may lead to GPA running faster than other higher level languages. If there is time, basic library support will be added to GPA.

Sample Code

```
proc gcd(a, b, c)
{
  c = 0;

  loop(1)
  {
    // no mod, use divison and subtraction
    c = a/b;
    c = a - (c*b);

    if(c == 0)
      ret b;
    else {}
    a = b;
    b = c;
  }
```

```
}

proc getNum(name, b, c)
{
  var temp;

  temp=0;

  print 'What is the value of ' ;
  print name ;
  print '?\n';

  input temp;
  ret temp;
}

proc go(a,b,c)
{
  a = 0;
  b = 0;
  c = 0;

  a = getNum(65,0,0);
  b = getNum(66,0,0);

  if(a>b)
    c = gcd(a,b,0);
  else
    c = gcd(b,a,0);

  print 'The gcd of ';
  print a;
  print ' and ';
  print b;
  print ' is:\n';
  print c;
  print '\n';

  ret 1;
}
```

1. Brief Tutorial

1.1 Structure of a Program

```
proc go(a,b,c)
{
  print ‘hello world!’;
 }
```

This is the simplest program in gpa. The “hello world!” program simply prints hello world!. How does it do this? Let’s go through it line by line.

The first line, “proc go(a,b,c)” is required. The function “go” is the point where all gpa programs begin execution. The word proc simply tells us that go is a procedure. The letters and commas inside the parentheses are required since all functions in gpa take three arguments. The argument names can be anything, they just need to be there. The line “print ‘hello world!’” is a statement in gpa. The keyword print simply tells gpa to output the thing after print to the screen. In this case, it will output the string “‘hello world!’.” We know it is a string because of the single quotes (‘) surrounding the words hello world!.
The braces are required for function definitions as well.

1.2 Variables

```
var a;
```

The above statement will declare a variable called “a.” All variables must be declared before they are used in gpa and they must be declared at the top of a function.

1.3 Operators

```
a = b + c;
```

The gcd program shows of some of the operators in gpa. You may notice “/” for division, “-“ for subtraction, and “*” for multiplication. There are many other operators but we will not cover them in this quick tutorial.

1.4 Basic I/O

```
var x;
input x;
print x;
```

In gpa, there are two kinds of I/O statements: “input” and “print.” The input statement takes a variable and assigns input from a prompt to that variable (only integers may be input). The print statement can take a variable name or a string as input and print out either the value of the variable name or the string.

1.5 Control Structures

```
var x;
x = 0;
loop(x < 7) {
    if (x >= 5)
        print x;
    else
```

```
        print 'X is less than 5';
    x = x + 1;
}
```

The gpa language has two control structures, "if-else" and "loop."

loop <condition> <statement>

The loop statement simply repeats the code after it until the condition is no longer true. It acts like the while loop in many programming languages.

if <condition> <statement> else <statement>

The if statement in gpa executes the code after it only if the condition is true. If the condition is false, the if will skip the statement. The else is optional and executed if the condition in the if statement is not met.

1.6 Functions

```
proc foo (a,b,c) {
    …
}
```

In gpa, functions always start with the word proc followed by the function name and the list of arguments. All functions in gpa must return something. If no return statement is put it into a procedure, the compiler will make the procedure return 0 at the end.

2. Lexical Conventions

2.1 Comments

GPA accepts two different types of comments:

2.1.1 Single Line Comments

The characters `//` begin a single line comment. Everything to the right side of these characters on the same line is ignored by the compiler.

2.1.2 Multiple Line Comments

The characters `/*` begin a multiple line comment and the characters `*/` will end the comment.

2.2 Identifiers

Identifiers are made of at least one letter and then may be followed by any number or combination of letters, numbers or underscores.

2.3 Keywords

The following identifiers are reserved for use as keywords, and may not be used otherwise (note: if the language has a shorthand equivalent defined for a keyword it is noted in the second column):

<u>Keyword</u>
`var`
`proc`
`ret`
`loop`
`if`
`else`
`print`

<u>2.4 Variables</u>
There are is one type of object stored by using the `var` identifier, the integer type. Strings cannot be stored as variables, but are only used by the print statement.

<u>2.4.1 Integers</u>
An integer is a sequence of digits.

<u>2.4.2 Strings</u>
A string is a sequence of characters started and terminated by the single quote character (‘). Strings refer to an area of storage initialized with the given characters. Within a string a single quote must be preceded another single-quote (‘). Certain non-graphic characters may be escaped according to the following table:

‘	‘’
\	\\
BS	\b
NL	\n
CR	\r
HT	\t

<u>3. Syntax Notation</u>
The syntax notation used in this manual, syntactic categories are indicated by *italic* type, and literal and characters in `Courier.` Alternatives are listed on separate lines. An optional terminal or non-terminal symbol is indicated by the subscript “opt,” so that

{ $\textit{expression}_{\text{opt}}$ }
would indicate an optional expression in braces.

<u>4. lvalues & procedures</u>
There is only one type of lvalue in GPA which is an identifier.

Procedures take three arguments of type integer and return an integer type only.

They are defined outside of the `go` procedure like this:

```
proc foo(a, b, c)
{
 …
}
```

And are used within a procedure like this:

{ x =$_{opt}$ } foo(a, b, c)

5. Conversions

No conversions can be done because operations can only be done on either a single constant expression or a set of homogenous variables therefore there is no need to do any conversions.

6. Expressions

The precedence of expression operators (see chart)

6.1 Primary expressions

Primary expressions involving function calls group left to right.

6.1.1 *identifier*

An identifier is a primary expression once it has been declared.

6.1.2 variable

An integer is a primary expression.

6.1.3 (*expression*)

A parenthesized expression is a primary expression whose type and value are identical to the same expression without parentheses.

6.1.4 *primary-expression* (*expression-list$_{opt}$*)

A function call is a primary expression followed by an optional listing of comma-separated identifiers or integers which serve as the arguments to the function.

6.2 Unary operators

Expressions with unary operators group right-to-left (these operators only apply to variables of integer type).

6.2.1 ! *expression*

The result of the '!' operator is the one's complement of the *expression*.

6.3 Multiplicative Expressions
The multiplicative operators * and / group left-to-right (these operators only apply to variables of integer type).

6.3.1 *expression * expression*
The '*' operator yields the product of the two expressions.

6.3.2 *expression / expression*
The '/' operator yields the (integer) quotient of the two expressions.

6.4 Additive Operators
The additive operators + and – group left-to-right (these operators only apply to variables of integer type).

6.4.1 *expression + expression*
The '+' operator yields the sum of the two expressions.

6.4.2 *expression – expression*
The '-' operator yields the difference of the two expressions.

6.5 Relational Operators
The relational operators group left-to-right, but this is not useful because if "a<b" is true then "a<b<c" is breaks down to "1<c" which is probably not what a programmer wanted to write (these operators only apply to variables of integer type).

6.5.1 *expression* < *expression*
6.5.2 *expression* > *expression*
For each of these operators, the result is 0 if the relation between each *expression* is false, 1 if true.

6.6 Equality operators
These operators only apply to variables of integer type.

6.6.1 *expression == expression*
6.6.2 *expression != expression*
The == (equal to) and the != (not equal to) operators are exactly analogous to the relational operators except that they have a lower precedence.

6.7 *expression* & *expression*
These operators only apply to variables of integer type.
The '&' operator groups from left-to-right and gives the bitwise 'and' of the two expressions.

6.8 *expression | expression*
These operators only apply to variables of integer type.
The ‘|’ operator groups from left-to-right and gives the bitwise ‘or’ of the two expressions.

6.9 *expression* && *expression*
These operators only apply to variables of integer type.
The ‘&&’ operator groups from left-to-right and yields 1 if both its operands are 1, otherwise, 0.

6.10 *expression* || *expression*
These operators only apply to variables of integer type.
The ‘||’ operator groups from left-to-right and yields 1 if one or both of its operands are 1 and 0 otherwise.

6.11 *identifier = expression*
The value of the expression replaces the value of the identifier (if it has already been defined).

7. Declarations

Declarations have the form:

7.1 variable *identifier*
A single identifier is created and is assumed to be defined later on. The variable is implicitly defined as 0 by the compiler.

8. Statements

Except as indicated, statements are executed in sequence.

8.1 expression statement
    expression

8.2 compound statement
compound-statement:
    { statement-list }

statement-list:
    statement
    statement \n *statement-list*

8.3 Conditional statement
There are multiple forms of the conditional statement:

if (*expression*) *statement*
if (*expression*) *statement* else *statement*
'if' must have its *expressions* evaluate to 1 in order for their respective *statements* (or *{ statement-list }*) to be executed, otherwise they jump down to the next 'else', or continue. The 'else' ambiguity is resovled by connecting it to the last elseless 'if.'

8.4 loop statement

The loop statement has the form:

`loop` *expression* (or 1)
    statement

The loop statement can take a conditional *expression.*

8.7 return statement

The return statement has one form:
`return` *expression*
This exits out of a function back to the calling procedure. It is used to return a value computed by the function that the calling procedure is using in an *expression*.

9. Scope Rules

Variables are local to the procedure in which they are declared unless defined in the go function, which makes them global.

10. Constant Expressions

All expressions evaluate to constants.

11. Examples

```
proc foo(a, b, m) {
  var c;
  c = 0;
  if(a > b)
     return 0;
  loop (a <= b) {
     c = c + 1;
     a = a + 1;
  }
  ret c;
}

proc loopDemo(a, b, m) {
  var z;
  var d;
  var i;
  d = 10;
```

```
  z = a + b - d;
  loop (z != b) {
    z--;
    if ( z <= 0 )
        return 0;
    else
      i++;
  }
}

proc gcd(a, b, m) {
  if (a == 0 && b == 0)
    b = 1;
  if (b == 0)
    b = a;
  if (a != 0)
    loop (a != b) {
      if (a < b)
        b -= a;
      else
        a -= b;
    }
   ret b;
}

go() {
   var a;
   var b;
   var c;
   var x;
   var y;
   var z;
   a = 4;
   b = 9;
   c = foo(a, b, 0);
   if (c == 0)
    print 'b is less than or greater than a.\n';
   else {
    print 'The difference of a and b is ';
    print c;
    print '.\n';
   }

   loopDemo(c, a, 0);

  y = 12;
  x = 18;
```

```
  z = gcd(x, y, 0);
  print ‘the gcd of ‘;
  print x;
  print ‘and ‘;
  print y;
  print ‘is ‘;
  print z;
  print ‘\n’;
}
```

1. Project Plan

Process for planning, specification, development and testing

When we began to plan for this project, we decided on a weekly meeting schedule. When we began to write the specification for the language, we used hints from C and assembly code along with some stylistic personal additions. Development was initially done by every member of the group to allow everyone's participation in constructing the grammar. This ensured that everyone understood the fundamental pieces of the language, namely the grammar. Tests were run everytime a change occurred in the grammar as well as in any part of the parse, walker, and compiler routines.

Style Guide

Using the example code provided above as a style guide, one can see the following trends:
1. Statements in the same set of braces are indented to the same point.
2. There are spaces between the commas and the next variable name in function calls and function definitions.
3. In parenthesized expressions, there are spaces between operaters and variable names or numbers.
4. Since gpa does not care about whitespace, there is liberal use of tabs and spaces to make the code easier to read.

Project timeline

We met weekly, to determine what sort of language we wanted to write.
September 26: Formalized a whitepaper which showed what features we wanted to have in our language.
October 19: Created a Language Reference Manual (LRM) which fleshed out the constructs of our language and how the grammar would look. Though this was not the final version of the language, it gave us a formal idea of what features to implement in our language. We also began to study ANTLR and how to write the Lexer and Parser.
November 20: The Lexer and Parser were formalized and we began to study how to write the Tree Walker.
December 1: Began to write an interpreter as a starting point to understanding how to interpret the Tree that ANTLR produces. Also, we began to write an Intermediate Representation (IR) to compile our code into. We then split the group around the IR, where 2 members would write the gpa-to-IR modules and the other 2 would write the IR-to-x86 component.
December 10: Decided we were ready to begin writing the compiler after we could see how the java routines interfaced with the Parse Tree.
December 17: Ran into a major problem where we felt we could not come up with a complete compiler. After rewriting the Intermediate Representation and determining how to solve a problem with recursive stack routines, we successfully completed our compiler.
December 19: Complete project is due along with the presentation.

Roles and responsibilities of each team member

Michael Pierorazio, Project Manager
- Lead the team and setup meeting times.
- Made sure all group members were present at meetings.
- Worked on the IR to x86 team.
- Responsible for the final IR.

Dmitry Gimzelberg, Systems Architect
- Created the initial design for the grammar and parse tree.
- Responsible for the initial IR.
- Worked on the gpa to IR team.
- Responsible for code generation routines.

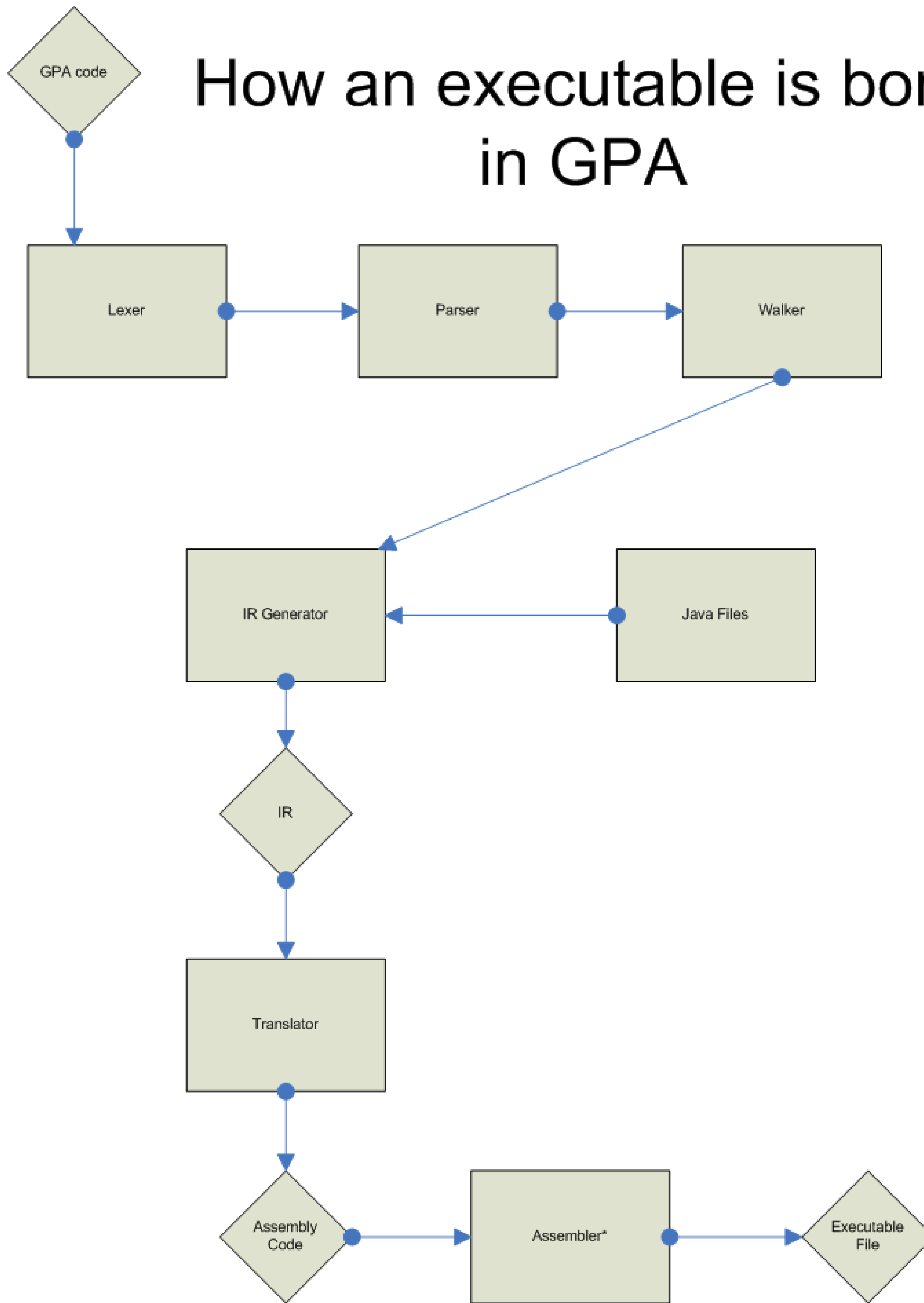
Juan Gutierrez, Quality Assurance
- Worked on the gpa to IR team.
- Responsible for the semantic checking routines for the gpa code.
- Responsible for writing several programs in gpa to debug the compiler.

Vincenzo Zarrillo, Systems Integrator
- Worked on the IR to x86 team.
- Responsible for writing several programs in gpa to debug the compiler.

Software development environment used (tools and languages)

The tools we used were linux and emacs. We did not use any sort of CVS software for this project, but created frequent backups.
ANTLR was the primary coding environment along with Java to write the tree parser and walker.
Perl was used to write the IR to x86 assembly translator.
To compile the java code and run it, we use a bash script.
Finally, the whole compilation from gpa to executable is controlled by a bash script.

2. Architectural Design



Interfaces between components

The gpa.g and walker.g files are parsed by ANTLR to create the lexer, parser, and walker java files. The walker calls the routines in the gpaGen java file which looks at the genTable and gpaSymTab java files to check for compile-time errors. It then prints out the code. gpaMain is the java file that serves as the entry point for this system. After this stage, the output code is passed to the translator perl script which generates x86 assembly code. This is then piped to a file that is assembled in gcc to create an executable file.

State who implemented each component

Michael Pierorazio and Vincenzo Zarrillo
- Implemented translator.pl

Dmitry Gimzelberg
- Implemented gpaMain.java
- Implemented gpaGen.java

Juan Gutierrez
- Implemented gpaSymTab.java
- Implemented genTable.java

Grammar and Parser implemented by everyone.
Walker implemented by Dmitry and Juan.
Bash Script to run everything implemented by Juan.

(report continues on page 36 after test code is shown)

3. Test Plan

bigOps.gpa

```
proc poop(a,b,c) {
 var e;
 var f;

 c = b+a;
 ret c;
}

proc bar(a,b,c) {
 var d;
 var g;

 d = poop(14,a,0);
 g = d + a;
 ret g;
}

proc go(a,b,c) {
 var g;
 var k;

 a = 7;
 b = 5;
 b = poop(a,a,0);
 k = bar(2,0,0);
 g = 12 * b + a * k + bar(3,3,3);
 print 'First g=: ';
 print g;
 print '\n';
 if(a == 7 && b == 5)
   c=8;

 loop(a < 100 && b < 200)
 {
   g=12 * b + a * k + bar(3,3,3) + poop(a,a,0) / bar(k,0,0) - 30;
   print g;
   print '\n';

   a=a+10;
   b=b+10;
 }

 print g;
 print '\n';
 //g = (a*3)+12 || b-c*12 < 4/a;

 ret 0;
}
```

bigOps.s

```
.LC0:
```

```
   .string "%d"
.LC1:
   .string "%s"
   .text
.LC2:
   .string "First g=: "
   .text
.LC3:
   .string "\n"
   .text
.LC4:
   .string "\n"
   .text
.LC5:
   .string "\n"
   .text
gpaIR:


.globl poop
    .type poop, @function
poop:
    pushl    %ebp
    movl     %esp, %ebp
    subl     $56, %esp
    movl     $0, -12(%ebp)
    movl     $0, -16(%ebp)
    movl     12(%ebp), %eax
    movl     %eax, -8(%ebp)
    movl     -8(%ebp), %eax
    movl     %eax, -20(%ebp)
    movl     8(%ebp), %eax
    movl     %eax, -8(%ebp)
    movl     -8(%ebp), %eax
    movl     %eax, -24(%ebp)
    movl     -20(%ebp), %eax
    addl     -24(%ebp), %eax
    movl     %eax, -8(%ebp)
    movl     -8(%ebp), %eax
    movl     %eax, 16(%ebp)
    movl     16(%ebp), %eax
    movl     %eax, -8(%ebp)
    movl     -8(%ebp), %eax
    jmp   .RE1
.RE1:
    leave
    ret
    .size poop, .-poop
    .section .rodata
.globl bar
    .type bar, @function
bar:
    pushl    %ebp
    movl     %esp, %ebp
    subl     $56, %esp
    movl     $0, -12(%ebp)
    movl     $0, -16(%ebp)
```

```
    movl     $0, %eax
    movl     %eax, 8(%esp)
    movl     8(%ebp), %eax
    movl     %eax, 4(%esp)
    movl     $14, %eax
    movl     %eax, (%esp)
    call     poop
    movl     %eax, -8(%ebp)
    movl     -8(%ebp), %eax
    movl     %eax, -12(%ebp)
    movl     -12(%ebp), %eax
    movl     %eax, -8(%ebp)
    movl     -8(%ebp), %eax
    movl     %eax, -20(%ebp)
    movl     8(%ebp), %eax
    movl     %eax, -8(%ebp)
    movl     -8(%ebp), %eax
    movl     %eax, -24(%ebp)
    movl     -20(%ebp), %eax
    addl     -24(%ebp), %eax
    movl     %eax, -8(%ebp)
    movl     -8(%ebp), %eax
    movl     %eax, -16(%ebp)
    movl     -16(%ebp), %eax
    movl     %eax, -8(%ebp)
    movl     -8(%ebp), %eax
    jmp   .RE2
.RE2:
    leave
    ret
    .size bar, .-bar
    .section .rodata
.globl main
    .type main, @function
main:
    leal     4(%esp), %ecx
    andl     $-16, %esp
    pushl    -4(%ecx)
    pushl    %ebp
    movl     %esp, %ebp
    pushl    %ecx
    subl     $344, %esp
    movl     $0, -12(%ebp)
    movl     $0, -16(%ebp)
    movl     $7, %eax
    movl     %eax, -8(%ebp)
    movl     -8(%ebp), %eax
    movl     %eax, 8(%ebp)
    movl     $5, %eax
    movl     %eax, -8(%ebp)
    movl     -8(%ebp), %eax
    movl     %eax, 12(%ebp)
    movl     $0, %eax
    movl     %eax, 8(%esp)
    movl     8(%ebp), %eax
    movl     %eax, 4(%esp)
    movl     8(%ebp), %eax
```

```
movl    %eax, (%esp)
call    poop
movl    %eax, -8(%ebp)
movl    -8(%ebp), %eax
movl    %eax, 12(%ebp)
movl    $0, %eax
movl    %eax, 8(%esp)
movl    $0, %eax
movl    %eax, 4(%esp)
movl    $2, %eax
movl    %eax, (%esp)
call    bar
movl    %eax, -8(%ebp)
movl    -8(%ebp), %eax
movl    %eax, -16(%ebp)
movl    $12, %eax
movl    %eax, -8(%ebp)
movl    -8(%ebp), %eax
movl    %eax, -20(%ebp)
movl    12(%ebp), %eax
movl    %eax, -8(%ebp)
movl    -8(%ebp), %eax
movl    %eax, -24(%ebp)
movl    -20(%ebp), %eax
imull   -24(%ebp), %eax
movl    %eax, -8(%ebp)
movl    -8(%ebp), %eax
movl    %eax, -28(%ebp)
movl    8(%ebp), %eax
movl    %eax, -8(%ebp)
movl    -8(%ebp), %eax
movl    %eax, -32(%ebp)
movl    -16(%ebp), %eax
movl    %eax, -8(%ebp)
movl    -8(%ebp), %eax
movl    %eax, -36(%ebp)
movl    -32(%ebp), %eax
imull   -36(%ebp), %eax
movl    %eax, -8(%ebp)
movl    -8(%ebp), %eax
movl    %eax, -40(%ebp)
movl    -28(%ebp), %eax
addl    -40(%ebp), %eax
movl    %eax, -8(%ebp)
movl    -8(%ebp), %eax
movl    %eax, -44(%ebp)
movl    $3, %eax
movl    %eax, 8(%esp)
movl    $3, %eax
movl    %eax, 4(%esp)
movl    $3, %eax
movl    %eax, (%esp)
call    bar
movl    %eax, -8(%ebp)
movl    -8(%ebp), %eax
movl    %eax, -48(%ebp)
movl    -44(%ebp), %eax
```

```
    addl     -48(%ebp), %eax
    movl     %eax, -8(%ebp)
    movl     -8(%ebp), %eax
    movl     %eax, -12(%ebp)
    movl     $.LC2, (%esp)
    call     printf
    movl     -12(%ebp), %eax
    movl     %eax, 4(%esp)
    movl     $.LC0, (%esp)
    call     printf
    movl     $.LC3, (%esp)
    call     printf
    movl     8(%ebp), %eax
    movl     %eax, -8(%ebp)
    movl     -8(%ebp), %eax
    movl     %eax, -52(%ebp)
    movl     $7, %eax
    movl     %eax, -8(%ebp)
    movl     -8(%ebp), %eax
    movl     %eax, -56(%ebp)
    movl     -52(%ebp), %eax
    cmpl     -56(%ebp), %eax
    sete     %al
    movzbl   %al, %eax
    movl     %eax, -8(%ebp)
    movl     -8(%ebp), %eax
    movl     %eax, -60(%ebp)
    movl     12(%ebp), %eax
    movl     %eax, -8(%ebp)
    movl     -8(%ebp), %eax
    movl     %eax, -64(%ebp)
    movl     $5, %eax
    movl     %eax, -8(%ebp)
    movl     -8(%ebp), %eax
    movl     %eax, -68(%ebp)
    movl     -64(%ebp), %eax
    cmpl     -68(%ebp), %eax
    sete     %al
    movzbl   %al, %eax
    movl     %eax, -8(%ebp)
    movl     -8(%ebp), %eax
    movl     %eax, -72(%ebp)
    movl     -60(%ebp), %eax
    andl     -72(%ebp), %eax
    movl     %eax, -8(%ebp)
    movl     -8(%ebp), %eax
    cmpl     $1, %eax
    jne   .L1
    movl     $8, %eax
    movl     %eax, -8(%ebp)
    movl     -8(%ebp), %eax
    movl     %eax, 16(%ebp)
    jmp   .E1
.L1:
.E1:
.L2:
    movl     8(%ebp), %eax
```

```
movl     %eax, -8(%ebp)
movl     -8(%ebp), %eax
movl     %eax, -76(%ebp)
movl     $100, %eax
movl     %eax, -8(%ebp)
movl     -8(%ebp), %eax
movl     %eax, -80(%ebp)
movl     -76(%ebp), %eax
cmpl     -80(%ebp), %eax
setl     %al
movzbl   %al, %eax
movl     %eax, -8(%ebp)
movl     -8(%ebp), %eax
movl     %eax, -84(%ebp)
movl     12(%ebp), %eax
movl     %eax, -8(%ebp)
movl     -8(%ebp), %eax
movl     %eax, -88(%ebp)
movl     $200, %eax
movl     %eax, -8(%ebp)
movl     -8(%ebp), %eax
movl     %eax, -92(%ebp)
movl     -88(%ebp), %eax
cmpl     -92(%ebp), %eax
setl     %al
movzbl   %al, %eax
movl     %eax, -8(%ebp)
movl     -8(%ebp), %eax
movl     %eax, -96(%ebp)
movl     -84(%ebp), %eax
andl     -96(%ebp), %eax
movl     %eax, -8(%ebp)
movl     -8(%ebp), %eax
cmpl     $1, %eax
jne   .E2
movl     $12, %eax
movl     %eax, -8(%ebp)
movl     -8(%ebp), %eax
movl     %eax, -100(%ebp)
movl     12(%ebp), %eax
movl     %eax, -8(%ebp)
movl     -8(%ebp), %eax
movl     %eax, -104(%ebp)
movl     -100(%ebp), %eax
imull    -104(%ebp), %eax
movl     %eax, -8(%ebp)
movl     -8(%ebp), %eax
movl     %eax, -108(%ebp)
movl     8(%ebp), %eax
movl     %eax, -8(%ebp)
movl     -8(%ebp), %eax
movl     %eax, -112(%ebp)
movl     -16(%ebp), %eax
movl     %eax, -8(%ebp)
movl     -8(%ebp), %eax
movl     %eax, -116(%ebp)
movl     -112(%ebp), %eax
```

```
imull    -116(%ebp), %eax
movl     %eax, -8(%ebp)
movl     -8(%ebp), %eax
movl     %eax, -120(%ebp)
movl     -108(%ebp), %eax
addl     -120(%ebp), %eax
movl     %eax, -8(%ebp)
movl     -8(%ebp), %eax
movl     %eax, -124(%ebp)
movl     $3, %eax
movl     %eax, 8(%esp)
movl     $3, %eax
movl     %eax, 4(%esp)
movl     $3, %eax
movl     %eax, (%esp)
call     bar
movl     %eax, -8(%ebp)
movl     -8(%ebp), %eax
movl     %eax, -128(%ebp)
movl     -124(%ebp), %eax
addl     -128(%ebp), %eax
movl     %eax, -8(%ebp)
movl     -8(%ebp), %eax
movl     %eax, -132(%ebp)
movl     $0, %eax
movl     %eax, 8(%esp)
movl     8(%ebp), %eax
movl     %eax, 4(%esp)
movl     8(%ebp), %eax
movl     %eax, (%esp)
call     poop
movl     %eax, -8(%ebp)
movl     -8(%ebp), %eax
movl     %eax, -136(%ebp)
movl     $0, %eax
movl     %eax, 8(%esp)
movl     $0, %eax
movl     %eax, 4(%esp)
movl     -16(%ebp), %eax
movl     %eax, (%esp)
call     bar
movl     %eax, -8(%ebp)
movl     -8(%ebp), %eax
movl     %eax, -140(%ebp)
movl     -136(%ebp), %eax
cltd
idivl    -140(%ebp)
movl     %eax, -8(%ebp)
movl     -8(%ebp), %eax
movl     %eax, -144(%ebp)
movl     -132(%ebp), %eax
addl     -144(%ebp), %eax
movl     %eax, -8(%ebp)
movl     -8(%ebp), %eax
movl     %eax, -148(%ebp)
movl     $30, %eax
movl     %eax, -8(%ebp)
```

```
        movl     -8(%ebp), %eax
        movl     %eax, -152(%ebp)
        movl     -148(%ebp), %eax
        subl     -152(%ebp), %eax
        movl     %eax, -8(%ebp)
        movl     -8(%ebp), %eax
        movl     %eax, -12(%ebp)
        movl     -12(%ebp), %eax
        movl     %eax, 4(%esp)
        movl     $.LC0, (%esp)
        call     printf
        movl     $.LC4, (%esp)
        call     printf
        movl     8(%ebp), %eax
        movl     %eax, -8(%ebp)
        movl     -8(%ebp), %eax
        movl     %eax, -156(%ebp)
        movl     $10, %eax
        movl     %eax, -8(%ebp)
        movl     -8(%ebp), %eax
        movl     %eax, -160(%ebp)
        movl     -156(%ebp), %eax
        addl     -160(%ebp), %eax
        movl     %eax, -8(%ebp)
        movl     -8(%ebp), %eax
        movl     %eax, 8(%ebp)
        movl     12(%ebp), %eax
        movl     %eax, -8(%ebp)
        movl     -8(%ebp), %eax
        movl     %eax, -164(%ebp)
        movl     $10, %eax
        movl     %eax, -8(%ebp)
        movl     -8(%ebp), %eax
        movl     %eax, -168(%ebp)
        movl     -164(%ebp), %eax
        addl     -168(%ebp), %eax
        movl     %eax, -8(%ebp)
        movl     -8(%ebp), %eax
        movl     %eax, 12(%ebp)
        jmp   .L2
    .E2:
        movl     -12(%ebp), %eax
        movl     %eax, 4(%esp)
        movl     $.LC0, (%esp)
        call     printf
        movl     $.LC5, (%esp)
        call     printf
        movl     $0, %eax
        movl     %eax, -8(%ebp)
        movl     -8(%ebp), %eax
        jmp   .RE3
    .RE3:
        addl     $344, %esp
        popl     %ecx
        popl     %ebp
        leal     -4(%ecx), %esp
        ret
```

```
    .size main, .-main
    .section .note.GNU-stack,"",@progbits
```

fibonacci.gpa

```
proc fib(n,m,o)
{
  var a;
  if(n < 2 || n == 2)
    ret 1;
  else
  {
    o=n-1;
    a=n-2;
    m=fib(o,0,0)+fib(a,0,0);
    ret m;
  }
}

proc go(a,b,c)
{
   print 'Calculate the nth fibonacci number!\n';
   print 'Enter n=: ';
   input a;
   b=fib(a,0,0);

   print b;
   print ' is the nth fibonacci number.\n';
}
```

fibonacci.s

```
.LC0:
   .string "%d"
.LC1:
   .string "%s"
   .text
.LC2:
   .string "Calculate the nth fibonacci number!\n"
   .text
.LC3:
   .string "Enter n=: "
   .text
.LC4:
   .string " is the nth fibonacci number.\n"
   .text
gpaIR:


.globl fib
    .type fib, @function
fib:
    pushl    %ebp
    movl     %esp, %ebp
    subl     $128, %esp
    movl     $0, -12(%ebp)
```

```
    movl     8(%ebp), %eax
    movl     %eax, -8(%ebp)
    movl     -8(%ebp), %eax
    movl     %eax, -16(%ebp)
    movl     $2, %eax
    movl     %eax, -8(%ebp)
    movl     -8(%ebp), %eax
    movl     %eax, -20(%ebp)
    movl     -16(%ebp), %eax
    cmpl     -20(%ebp), %eax
    setl     %al
    movzbl   %al, %eax
    movl     %eax, -8(%ebp)
    movl     -8(%ebp), %eax
    movl     %eax, -24(%ebp)
    movl     8(%ebp), %eax
    movl     %eax, -8(%ebp)
    movl     -8(%ebp), %eax
    movl     %eax, -28(%ebp)
    movl     $2, %eax
    movl     %eax, -8(%ebp)
    movl     -8(%ebp), %eax
    movl     %eax, -32(%ebp)
    movl     -28(%ebp), %eax
    cmpl     -32(%ebp), %eax
    sete     %al
    movzbl   %al, %eax
    movl     %eax, -8(%ebp)
    movl     -8(%ebp), %eax
    movl     %eax, -36(%ebp)
    movl     -24(%ebp), %eax
    orl   -36(%ebp), %eax
    movl     %eax, -8(%ebp)
    movl     -8(%ebp), %eax
    cmpl     $1, %eax
    jne   .L1
    movl     $1, %eax
    movl     %eax, -8(%ebp)
    movl     -8(%ebp), %eax
    jmp   .RE1
    jmp   .E1
.L1:
    movl     8(%ebp), %eax
    movl     %eax, -8(%ebp)
    movl     -8(%ebp), %eax
    movl     %eax, -40(%ebp)
    movl     $1, %eax
    movl     %eax, -8(%ebp)
    movl     -8(%ebp), %eax
    movl     %eax, -44(%ebp)
    movl     -40(%ebp), %eax
    subl     -44(%ebp), %eax
    movl     %eax, -8(%ebp)
    movl     -8(%ebp), %eax
    movl     %eax, 16(%ebp)
    movl     8(%ebp), %eax
    movl     %eax, -8(%ebp)
```

```
    movl     -8(%ebp), %eax
    movl     %eax, -48(%ebp)
    movl     $2, %eax
    movl     %eax, -8(%ebp)
    movl     -8(%ebp), %eax
    movl     %eax, -52(%ebp)
    movl     -48(%ebp), %eax
    subl     -52(%ebp), %eax
    movl     %eax, -8(%ebp)
    movl     -8(%ebp), %eax
    movl     %eax, -12(%ebp)
    movl     $0, %eax
    movl     %eax, 8(%esp)
    movl     $0, %eax
    movl     %eax, 4(%esp)
    movl     16(%ebp), %eax
    movl     %eax, (%esp)
    call     fib
    movl     %eax, -8(%ebp)
    movl     -8(%ebp), %eax
    movl     %eax, -56(%ebp)
    movl     $0, %eax
    movl     %eax, 8(%esp)
    movl     $0, %eax
    movl     %eax, 4(%esp)
    movl     -12(%ebp), %eax
    movl     %eax, (%esp)
    call     fib
    movl     %eax, -8(%ebp)
    movl     -8(%ebp), %eax
    movl     %eax, -60(%ebp)
    movl     -56(%ebp), %eax
    addl     -60(%ebp), %eax
    movl     %eax, -8(%ebp)
    movl     -8(%ebp), %eax
    movl     %eax, 12(%ebp)
    movl     12(%ebp), %eax
    movl     %eax, -8(%ebp)
    movl     -8(%ebp), %eax
    jmp   .RE1
.E1:
.RE1:
    leave
    ret
    .size fib, .-fib
    .section .rodata
.globl main
    .type main, @function
main:
    leal     4(%esp), %ecx
    andl     $-16, %esp
    pushl    -4(%ecx)
    pushl    %ebp
    movl     %esp, %ebp
    pushl    %ecx
    subl     $24, %esp
    movl     $.LC2, (%esp)
```

```
    call     printf
    movl     $.LC3, (%esp)
    call     printf
    leal     -8(%ebp), %eax
    movl     %eax, 4(%esp)
    movl     $.LC1, (%esp)
    call     scanf
    leal     -8(%ebp), %eax
    movl     %eax, (%esp)
    call     atoi
    movl     %eax, 8(%ebp)
    movl     $0, %eax
    movl     %eax, 8(%esp)
    movl     $0, %eax
    movl     %eax, 4(%esp)
    movl     8(%ebp), %eax
    movl     %eax, (%esp)
    call     fib
    movl     %eax, -8(%ebp)
    movl     -8(%ebp), %eax
    movl     %eax, 12(%ebp)
    movl     12(%ebp), %eax
    movl     %eax, 4(%esp)
    movl     $.LC0, (%esp)
    call     printf
    movl     $.LC4, (%esp)
    call     printf
.RE2:
    addl     $24, %esp
    popl     %ecx
    popl     %ebp
    leal     -4(%ecx), %esp
    ret
    .size main, .-main
       .section .note.GNU-stack,"",@progbits
```

gcd.gpa

```
proc gcd(a, b, c)
{
```

```
  c = 0;

  loop(1)
  {
    // no mod, use divison and subtraction
    c = a/b;
    c = a - (c*b);

    if(c == 0)
      ret b;
    else {}
    a = b;
    b = c;
  }

}

proc getNum(name, b, c)
{
  var temp;

  temp=0;

  print 'What is the value of ' ;
  print name ;
  print '?\n';

  input temp;
  ret temp;
}

proc go(a,b,c)
{
  a = 0;
  b = 0;
  c = 0;

  a = getNum(65,0,0);
  b = getNum(66,0,0);

  if(a>b)
    c = gcd(a,b,0);
  else
    c = gcd(b,a,0);

  print 'The gcd of ';
  print a;
  print ' and ';
  print b;
  print ' is:\n';
  print c;
  print '\n';

  ret 1;
}
```

gcd.s

```
.LC0:
   .string "%d"
.LC1:
   .string "%s"
   .text
.LC2:
   .string "What is the value of "
   .text
.LC3:
   .string "?\n"
   .text
.LC4:
   .string "The gcd of "
   .text
.LC5:
   .string " and "
   .text
.LC6:
   .string " is:\n"
   .text
.LC7:
   .string "\n"
   .text
gpaIR:


.globl gcd
    .type gcd, @function
gcd:
    pushl    %ebp
    movl     %esp, %ebp
    subl     $88, %esp
    movl     $0, %eax
    movl     %eax, -8(%ebp)
    movl     -8(%ebp), %eax
    movl     %eax, 16(%ebp)
.L1:
    movl     $1, %eax
    movl     %eax, -8(%ebp)
    movl     -8(%ebp), %eax
    cmpl     $1, %eax
    jne   .E1
    movl     8(%ebp), %eax
    movl     %eax, -8(%ebp)
    movl     -8(%ebp), %eax
    movl     %eax, -12(%ebp)
    movl     12(%ebp), %eax
    movl     %eax, -8(%ebp)
    movl     -8(%ebp), %eax
    movl     %eax, -16(%ebp)
    movl     -12(%ebp), %eax
    cltd
    idivl    -16(%ebp)
    movl     %eax, -8(%ebp)
    movl     -8(%ebp), %eax
```

```
    movl     %eax, 16(%ebp)
    movl     8(%ebp), %eax
    movl     %eax, -8(%ebp)
    movl     -8(%ebp), %eax
    movl     %eax, -20(%ebp)
    movl     16(%ebp), %eax
    movl     %eax, -8(%ebp)
    movl     -8(%ebp), %eax
    movl     %eax, -24(%ebp)
    movl     12(%ebp), %eax
    movl     %eax, -8(%ebp)
    movl     -8(%ebp), %eax
    movl     %eax, -28(%ebp)
    movl     -24(%ebp), %eax
    imull    -28(%ebp), %eax
    movl     %eax, -8(%ebp)
    movl     -8(%ebp), %eax
    movl     %eax, -32(%ebp)
    movl     -20(%ebp), %eax
    subl     -32(%ebp), %eax
    movl     %eax, -8(%ebp)
    movl     -8(%ebp), %eax
    movl     %eax, 16(%ebp)
    movl     16(%ebp), %eax
    movl     %eax, -8(%ebp)
    movl     -8(%ebp), %eax
    movl     %eax, -36(%ebp)
    movl     $0, %eax
    movl     %eax, -8(%ebp)
    movl     -8(%ebp), %eax
    movl     %eax, -40(%ebp)
    movl     -36(%ebp), %eax
    cmpl     -40(%ebp), %eax
    sete     %al
    movzbl   %al, %eax
    movl     %eax, -8(%ebp)
    movl     -8(%ebp), %eax
    cmpl     $1, %eax
    jne   .L2
    movl     12(%ebp), %eax
    movl     %eax, -8(%ebp)
    movl     -8(%ebp), %eax
    jmp   .RE1
    jmp   .E2
.L2:
.E2:
    movl     12(%ebp), %eax
    movl     %eax, -8(%ebp)
    movl     -8(%ebp), %eax
    movl     %eax, 8(%ebp)
    movl     16(%ebp), %eax
    movl     %eax, -8(%ebp)
    movl     -8(%ebp), %eax
    movl     %eax, 12(%ebp)
    jmp   .L1
.E1:
.RE1:
```

```
	leave
	ret
	.size gcd, .-gcd
	.section .rodata
.globl getNum
	.type getNum, @function
getNum:
	pushl	%ebp
	movl	%esp, %ebp
	subl	$32, %esp
	movl	$0, -12(%ebp)
	movl	$0, %eax
	movl	%eax, -8(%ebp)
	movl	-8(%ebp), %eax
	movl	%eax, -12(%ebp)
	movl	$.LC2, (%esp)
	call	printf
	movl	8(%ebp), %eax
	movl	%eax, 4(%esp)
	movl	$.LC0, (%esp)
	call	printf
	movl	$.LC3, (%esp)
	call	printf
	leal	-8(%ebp), %eax
	movl	%eax, 4(%esp)
	movl	$.LC1, (%esp)
	call	scanf
	leal	-8(%ebp), %eax
	movl	%eax, (%esp)
	call	atoi
	movl	%eax, -12(%ebp)
	movl	-12(%ebp), %eax
	movl	%eax, -8(%ebp)
	movl	-8(%ebp), %eax
	jmp	.RE2
.RE2:
	leave
	ret
	.size getNum, .-getNum
	.section .rodata
.globl main
	.type main, @function
main:
	leal	4(%esp), %ecx
	andl	$-16, %esp
	pushl	-4(%ecx)
	pushl	%ebp
	movl	%esp, %ebp
	pushl	%ecx
	subl	$40, %esp
	movl	$0, %eax
	movl	%eax, -8(%ebp)
	movl	-8(%ebp), %eax
	movl	%eax, 8(%ebp)
	movl	$0, %eax
	movl	%eax, -8(%ebp)
	movl	-8(%ebp), %eax
```

```
    movl     %eax, 12(%ebp)
    movl     $0, %eax
    movl     %eax, -8(%ebp)
    movl     -8(%ebp), %eax
    movl     %eax, 16(%ebp)
    movl     $0, %eax
    movl     %eax, 8(%esp)
    movl     $0, %eax
    movl     %eax, 4(%esp)
    movl     $65, %eax
    movl     %eax, (%esp)
    call     getNum
    movl     %eax, -8(%ebp)
    movl     -8(%ebp), %eax
    movl     %eax, 8(%ebp)
    movl     $0, %eax
    movl     %eax, 8(%esp)
    movl     $0, %eax
    movl     %eax, 4(%esp)
    movl     $66, %eax
    movl     %eax, (%esp)
    call     getNum
    movl     %eax, -8(%ebp)
    movl     -8(%ebp), %eax
    movl     %eax, 12(%ebp)
    movl     8(%ebp), %eax
    movl     %eax, -8(%ebp)
    movl     -8(%ebp), %eax
    movl     %eax, -12(%ebp)
    movl     12(%ebp), %eax
    movl     %eax, -8(%ebp)
    movl     -8(%ebp), %eax
    movl     %eax, -16(%ebp)
    movl     -12(%ebp), %eax
    cmpl     -16(%ebp), %eax
    setg     %al
    movzbl   %al, %eax
    movl     %eax, -8(%ebp)
    movl     -8(%ebp), %eax
    cmpl     $1, %eax
    jne   .L3
    movl     $0, %eax
    movl     %eax, 8(%esp)
    movl     12(%ebp), %eax
    movl     %eax, 4(%esp)
    movl     8(%ebp), %eax
    movl     %eax, (%esp)
    call     gcd
    movl     %eax, -8(%ebp)
    movl     -8(%ebp), %eax
    movl     %eax, 16(%ebp)
    jmp   .E3
.L3:
    movl     $0, %eax
    movl     %eax, 8(%esp)
    movl     8(%ebp), %eax
    movl     %eax, 4(%esp)
```

```
    movl     12(%ebp), %eax
    movl     %eax, (%esp)
    call     gcd
    movl     %eax, -8(%ebp)
    movl     -8(%ebp), %eax
    movl     %eax, 16(%ebp)
.E3:
    movl     $.LC4, (%esp)
    call     printf
    movl     8(%ebp), %eax
    movl     %eax, 4(%esp)
    movl     $.LC0, (%esp)
    call     printf
    movl     $.LC5, (%esp)
    call     printf
    movl     12(%ebp), %eax
    movl     %eax, 4(%esp)
    movl     $.LC0, (%esp)
    call     printf
    movl     $.LC6, (%esp)
    call     printf
    movl     16(%ebp), %eax
    movl     %eax, 4(%esp)
    movl     $.LC0, (%esp)
    call     printf
    movl     $.LC7, (%esp)
    call     printf
    movl     $1, %eax
    movl     %eax, -8(%ebp)
    movl     -8(%ebp), %eax
    jmp   .RE3
.RE3:
    addl     $40, %esp
    popl     %ecx
    popl     %ebp
    leal     -4(%ecx), %esp
    ret
    .size main, .-main
    .section .note.GNU-stack,"",@progbits
```

Test suites used to test translator

We ran a number of programs at each step of the way to make sure the components of the compiler evaluated everything correctly.
We ran checks to make sure the grammar parsed correctly and built the tree in the proper way. We fed programs like this to the parser:

```
proc ftest (a,b,c) {

    if (1) {
        print a;
    }

    loop (x < 5)
        x = x + 1;
}
```

We then proceeded to incrementally test the grammar until all aspects of it worked.

Why are these test cases?

These test programs and test case were each chosen for their complexity and ability to test building the code. Specifically, fibonacci is used to test the recursion while bigOps is used to make sure that the tree generated by the walker is transformed into gcd code correctly.

Automation used in testing

No automated testing was done since all tests were run by the individuals who made changes.

State who did what

Juan and Vincenzo were responsible for writing and running the final test programs. Everyone wrote test cases in their individual teams to test their portions of the code.

4. Lessons Learned

Each team member explains his or her most important lesson

Michael

As both a developer and the project manager I had to wear two different hats at the same time and thus saw a unique sets of problems. As project manager I faced the problem of coordinating common time for meeting between four different schedules. Because the PLT project was not due until the end of the semester it was difficult to motivate work on the project when other deadlines were imminent. I learned that constant reminders of meeting times kept the PLT project in the forefront. When working on various components of the compiler I ran into several different roadblocks. I learned that stepping back and rethinking obstacles is the best way to discover creative solutions. For example when redesigning the IR I thought about what information was absolutely essential to the translator and what could be calculated with an initial pass.

Dmitry

In working on the code generation portion of the project, I learned how to properly generate assembly-like code. The greatest problem I faced was how to generate assembly code for a long list of math operations.
For example, how to parse: "c = a * (b - 3/f) + bar(g);"
Since the parser generated a tree of operations, I determined I would parse it using Left-Right-Root order, or post-order traversal. Also, the result of each of these ops is stored in a temporary variable in the IR called _out. I learned how a compiler works and how to quickly generate some sort of assembly code.

Juan

Specifically, I learned the power of ANTLR and grew to like it because it is so compact. Generally, in helping construct these set of algorithms, I was immersed in the path of what my code has been doing my entire programming career. This has basically changed how I look at programming: its all the same thing, the variations are in the different levels of indirection.

Vincenzo

In general, I learned that adding a new layer of indirection is not necessarily a bad thing. Also, because I was on the sub-team that dealt with programming the IR to x86 assembly language (AT&T syntax), I became a pro at understanding assembly and being able to produce it. As far as logistical learning went, it is very difficult for a four person group with four very different

schedules to coordinate meeting times. So I would suggest to other groups to create a weekly meeting time early in the semester that works with everyone's schedule. This will save a lot of time.

Advice the team has for future teams

- Step back and rethink obstacles
- ANTLR is fairly complex, don’t underestimate it
- Make sure the abstraction is specific enough so implementation is easier
- Make a block of time per week dedicated to the project that fits into everyone’s schedule

5. Appendix

gpa.g

```
class gpaLexer extends Lexer;
options {
    testLiterals = false;
    k = 2;
    exportVocab = gpa;
}

{
    int nr_error=0;
    public void reportError(String s)
    {
        System.out.println(s);
        super.reportError(s);
        nr_error++;
    }
    public void reportError(RecognitionException e)
    {
        System.out.println(e);
        super.reportError(e);
        nr_error++;
    }
}

PLUS    : '+' ;
MINUS   : '-' ;
MUL     : '*' ;
DIV     : '/' ;
ASSIGN  : '=' ;
LPAREN  : '(' ;
RPAREN  : ')' ;
LBRACE  : '{' ;
RBRACE  : '}' ;
SEMI    : ';' ;
COLON   : ':' ;
NOT     : '!' ;
AND     : "&&" ;
OR      : "||" ;
IFEQ    : "==" ;
LT      : '<' ;
GT      : '>' ;
SQUOTE  : '\'' ;
LBRACK  : '[' ;
RBRACK  : ']' ;
NEQUAL  : "!=" ;
COMMA   : ',' ;
DOT     : '.' ;
TILDE   : '~' ;


protected LETTER : ('a'..'z' | 'A'..'Z') ;
protected DIGIT  : '0'..'9' ;

ID options { testLiterals = true; } : LETTER (LETTER | DIGIT | '_')* ;
NUMBER : (DIGIT)+;
```

```
STRING : '\'' ('\'' '\''! | ~('\''))* '\'' ;

COMMENT : ( "/*" (
                    options {greedy=false;} :
                    (NL)
                    | ~( '\n' | '\r' )
                 )* "*/"
              | "//" (~( '\n' | '\r' ))* (NL)
            )                       { $setType(Token.SKIP); }
        ;

NL      : ('\n' | ('\r' '\n') => '\r' '\n' | '\r')
              { $setType(Token.SKIP); newline(); }
        ;

WS : ( ' ' | '\t')+ { $setType(Token.SKIP); } ;


class gpaParser extends Parser;
options {
    buildAST = true;
    k = 2;
    exportVocab = gpa;
}

tokens {
    DECL_LST;
    PROC;
    PROC_STMT;
    BLK_STMT;
    LOOP;
    PRGRM;
    PROC_CALL;
    PRNT_STMT;
    INPUT;
    CALL_ID_LIST;
    RET_STMT;
}

{
    int nr_error=0;
    public void reportError(String s)
    {
        System.out.println(s);
        super.reportError(s);
        nr_error++;
    }
    public void reportError(RecognitionException e)
    {
        System.out.println(e);
        super.reportError(e);
        nr_error++;
    }
}

// Main program
```

```
prgrm : /*decl_lst*/ (proc)+ EOF!
        {#prgrm = #([BLK_STMT, "PRGRM"], #prgrm); } ;

decl_lst : "var"! ID SEMI!
           { #decl_lst = #([DECL_LST, "DECL_LST"], #decl_lst); } ;

// Procedures

proc : "proc"! ID LPAREN! ID COMMA! ID COMMA! ID RPAREN! proc_stmt
       { #proc = #([PROC, "PROC"], #proc); } ;

// Procedure calls

proc_call : ID LPAREN! call_id_list RPAREN!
            { #proc_call = #([PROC_CALL, "PROC_CALL"], proc_call); };

call_id_list : ((ID | NUMBER) COMMA! (ID | NUMBER) COMMA! (ID |
NUMBER))
               { #call_id_list = #([CALL_ID_LIST, "CALL_ID_LIST"],
call_id_list); };

// Statements

stmt : ID ASSIGN^ bool SEMI!
     | proc_call
     | "if"^ bool (stmt | blk_stmt) (options { greedy = true; }:
"else"! (stmt | blk_stmt))?
     | loop
     | ret_stmt
     | prnt_stmt
     | input
     | SEMI!
     ;

input : "input"! ID SEMI! { #input = #([INPUT, "INPUT"], input); } ;

prnt_stmt : "print"! (STRING | ID | NUM) SEMI! { #prnt_stmt =
#([PRNT_STMT, "PRNT_STMT"], prnt_stmt); } ;

ret_stmt : "ret"! bool SEMI! { #ret_stmt = #([RET_STMT, "RET_STMT"],
ret_stmt); } ;

proc_stmt : (LBRACE! (decl_lst)* (stmt)* RBRACE!) { #proc_stmt =
#([BLK_STMT, "PROC_STMT"], proc_stmt); } ;

blk_stmt : (LBRACE! (stmt)* RBRACE!) { #blk_stmt = #([BLK_STMT,
"BLK_STMT"], blk_stmt); } ;

loop : "loop"! LPAREN! bool RPAREN! (stmt | blk_stmt) { #loop =
#([LOOP, "LOOP"], loop); } ;

// Operations (order & associativity)

bool     : join (OR^ join)*;
join     : rel (AND^ rel)* ;
rel      : expr ((IFEQ^ | NEQUAL^ | LT^ | GT^) expr)* ;
```

```
expr     : term ((PLUS^ | MINUS^) term)* ;
term     : unary ((MUL^ | DIV^) unary)* ;
unary    : NOT^ unary | factor ;
factor   : (LPAREN! bool RPAREN!) | proc_call | NUMBER | ID | "true" |
"false" ;
```

walker.g

```
{
    import java.io.*;
    import java.util.*;
}

class gpaWalker extends TreeParser;
options {
    importVocab = gpa;
}

{
    gpaGen irgen = new gpaGen();
}

expr
{
    String l[] = null;
```

```
    String arg = null;
}

    : #(OR a1:. b1:.              { irgen.genOpCode(this,"or", #a1,
#b1); } )
    | #(AND a2:. b2:.             {
irgen.genOpCode(this,"and",#a2,#b2); } )
    | #(NOT a3:.                  {
irgen.genOpCode(this,"not",#a3,null); } )
    | #(IFEQ a4:. b3:.            {
irgen.genOpCode(this,"seq",#a4,#b3); } )
    | #(NEQ a5:. b4:.             {
irgen.genOpCode(this,"sne",#a5,#b4); } )
    | #(LT a6:. b5:.              {
irgen.genOpCode(this,"slt",#a6,#b5); } )
    | #(GT a7:. b6:.              {
irgen.genOpCode(this,"sgt",#a7,#b6); } )
    | #(PLUS a8:. b7:.            {
irgen.genOpCode(this,"add",#a8,#b7); } )
    | #(MINUS a9:. b8:.           {
irgen.genOpCode(this,"sub",#a9,#b8); } )
    | #(MUL a10:. b9:.            {
irgen.genOpCode(this,"mult",#a10,#b9); } )
    | #(DIV a11:. b10:.           {
irgen.genOpCode(this,"div",#a11,#b10); } )
    | #(ASSIGN id1:ID b11:.       {
irgen.genAsnCode(this,id1.getText(),#b11); } )
    | #(PROC_CALL a12:ID l=plst   { irgen.genCallCode(a12.getText(),l);
} )
    | num:NUMBER                  { irgen.getTerm("num",num.getText());
}
    | "true"                      { irgen.getTerm("num","1"); }
    | "false"                     { irgen.getTerm("num","0"); }
    | #(id2:ID                    { irgen.getTerm("id",id2.getText());
} )
    | #("if" a13:. thenp:. (elsep:.)?) {
            if (elsep == null)
                irgen.genIf(this,#a13,#thenp,null);
            else
                irgen.genIf(this,#a13,#thenp,#elsep);
        }
    | #(BLK_STMT (stmt:.                { expr(#stmt); } )* )
    | #(LOOP a14:. loopbody:.           {
irgen.genLoop(this,#a14,#loopbody); } )
    | #(RET_STMT retp:.                 { irgen.retCode(this,#retp); }
)
    | #(PROC pname:ID v1:ID v2:ID v3:ID pbody:. {
                irgen.procReg(this,
pname.getText(),v1.getText(),v2.getText(),v3.getText(),#pbody); }
        )
    | #(DECL_LST dcl:ID) {
            if (dcl == null)
                irgen.varDec(null);
            else
                irgen.varDec(dcl.getText());
        }
    | #(PRNT_STMT arg=idnum)           { irgen.printarg(arg); }
```

```
    | #(INPUT id3:ID                   { irgen.inputarg(id3.getText());
} )
    ;


plst returns [ String[] rv ]
{
    Vector v;
    String a;
    rv = null;
}

    : #(CALL_ID_LIST       { v = new Vector(); }
            (a=idnum      { v.add(a); }
            )*
        )                 { rv = irgen.convertVarList(v); }
     ;

idnum returns [ String r ]
{
    r = null;
}
    : #(id:ID             { r = id.getText(); } )
    | #(num:NUMBER        { r = num.getText(); } )
    | #(str:STRING        { r = str.getText(); } )
    ;

idlst returns [ String[] rv ]
{
    Vector v;
    rv = null;
}

    : #(DECL_LST            { v = new Vector(); }
            (a:ID           { v.add(a.getText()); }
            )*
        )                   { rv = irgen.convertVarList(v); }
    ;
```

genTable.java

```
import java.util.*;
import java.io.PrintWriter;

public class genTable extends HashMap
{
    final Object a = 1;

    public genTable()
    {}

    public boolean testValue(String name)
    {
   //System.out.println("Checking if " + name + " exists.");
   Object x = this.get(name);

   if(x != a)
```

```
        return false;
    else
        return true;
     }

}
```

gpaGen.java

```
import java.io.*;
import java.util.*;
import java.lang.*;
import java.util.regex.Pattern;
import antlr.CommonAST;
import antlr.collections.AST;
import antlr.RecognitionException;
import antlr.TokenStreamException;
import antlr.TokenStreamIOException;

public class gpaGen
{
    // globals
    // Make zero in constructor

    int lpifcnt;
    int rcnt;
    Object a = 1;
    genTable gt;
    gpaSymTab current;

    public gpaGen()
    {
   lpifcnt = rcnt = 0;
   gt = new genTable();
   current = new gpaSymTab();
    }

    public void genOpCode(gpaWalker walker, String op, AST left, AST
right) throws RecognitionException {

   rcnt++;
   String in1 = new String(" _in" + rcnt);
   int temp = ++rcnt;
   String in2 = new String(" _in" + temp);

   walker.expr(left);
   // Generate Code that makes in1 = out
   System.out.println("asn _out" + in1);

   if (right != null) {
       walker.expr(right);
       // Generate Code that makes in2 = out
       System.out.println("asn _out" + in2);
   }

   // Generate Code that makes out = op(in1,in2)
   if (op != "not") {
```

```
        System.out.println(op + in1 + in2  + " _out");
    }
    else
        System.out.println(op + in1 + " _out");
     }

     public void getTerm(String type, String name) {

    if (type == "num")
        System.out.println("asn " + "$" + name + " _out");
    else if (type == "id") {
        // Look in the symbol table to see if the variable exists
        // If found in the table, print the code
        if(current.testValue(name))
      System.out.println("asn " + name + " _out");
        else
        {
      System.out.println("Error: variable " + name + " has not been
declared.");
      System.exit(0);
        }
    }
     }

     public void genAsnCode(gpaWalker walker, String name, AST right)
throws RecognitionException {

    if(current.testValue(name)) {
        // Look in the symbol table to see if the variable exists
        // If found in the table, print the code
        walker.expr(right);
        System.out.println("asn _out " + name);
    }
    else {
        System.out.println("Error: variable " + name + " has not been
declared.");
        System.exit(0);
    }
     }

     public void procReg(gpaWalker walker, String pname, String v1,
String v2, String v3, AST pbody) throws RecognitionException {

    // Check if pname is already defined
    if(gt.testValue(pname)) {
        System.out.println("Error: Function + " + pname + " already
defined.");
        System.exit(0);
    }
    // If not, print code to define
    gt.put(pname,a);
    System.out.println("def " + pname + " " + v1 + " " + v2 + " " + v3);
    // Add v1,v2,v3 to the symbol table for procedure pname
    current.put(v1,a);
    current.put(v2,a);
    current.put(v3,a);
    // Generate body code
```

```
    walker.expr(pbody);
    System.out.println("end " + pname);

    current = new gpaSymTab();
     }

     public void genIf(gpaWalker walker, AST a, AST thenp, AST elsep)
throws RecognitionException {

    lpifcnt++;
    int temp = lpifcnt;
    System.out.println("if " + temp);
    walker.expr(a);
    System.out.println("then " + temp);
    walker.expr(thenp);
    System.out.println("neht " + temp);
    System.out.println("else " + temp);
    if (elsep != null)
        walker.expr(elsep);
    System.out.println("fi " + temp);
     }

     public String[] convertVarList(Vector v) {
    String[] sv = new String[ v.size() ];
    for(int i=0; i < sv.length; i++)
        sv[i] = (String)v.elementAt(i);
    return sv;
     }

     public void genLoop(gpaWalker walker, AST a, AST loopbody) throws
RecognitionException {

    lpifcnt++;
    int temp = lpifcnt;
    System.out.println("loop " + temp);
    walker.expr(a);
    System.out.println("run loop " + temp);
    walker.expr(loopbody);
    System.out.println("pool " + temp);
     }

     public void retCode(gpaWalker walker, AST retp) throws
RecognitionException {

    walker.expr(retp);
    System.out.println("ret _out");
     }

     public void genCallCode(String pname, String[] l) {

    // Make sure procedure is defined
    if(!gt.testValue(pname)) {
        System.out.println("Error: Function " + pname + " is
undefined.");
        System.exit(0);
    }
    // If it is defined
```

```
    System.out.print("call " + pname);
    boolean wasId;
    for (int i = 0; i < l.length; i++) {

        wasId = false;
        try {
       int t = (Integer.parseInt(l[i]));
        }
        catch (NumberFormatException e) {
       // This means t was an ID
       wasId = true;
        }
        if (wasId) {
       if(!current.testValue(l[i])) {
           System.out.println("Error: variable " + l[i] + " is
undefined.");
           System.exit(0);
       }
       System.out.print(" " + l[i]);
        }
        else System.out.print(" $" + l[i]);
    }
    System.out.println();
     }

     public void varDec(String l) {

    if (l == null)
        return;
    //for (int i = 0; i < l.length; i++) {
        // Check if variable previously defined
        // If not, generate code
    if(current.testValue(l)) {
        System.out.println("Error: variable " + l + " is already
defined.");
        System.exit(0);
    }
    System.out.println("decl $0 " + l);
    //System.out.println("PUTTING IN " + l + " INTO SYMBOL TABLE");
    current.put(l,a);
     }


     public void printarg(String l) {

    int t;
    if (l.startsWith("'")) {
        System.out.print("prst ");
        String stmt = l.substring(1,l.length()-1);
        stmt.replaceAll("\" ","c");
        System.out.print("\"" + stmt + "\"");
    }

    else {
        System.out.print("prnm");
        boolean wasId;
        wasId = false;
```

```
        try {
       t = Integer.parseInt(l);
        }
        catch (NumberFormatException e) {
       // This means t was an ID
       wasId = true;
        }
        if (wasId) {
       if(!current.testValue(l)) {
           System.out.println("Error: variable " + l + " is
undefined.");
           System.exit(0);
       }
       System.out.print(" " + l);
        }
        else System.out.print(" $" + l);
   }
   System.out.println();
    }

    public void inputarg(String i) {

   //Check if argument in symbol table
   //If it is, generate code
   if(!current.testValue(i)) {
       System.out.println("Error: variable " + i + " is undefined.");
       System.exit(0);
   }
   System.out.println("input " + i);
    }
}
```

gpaSymTab.java

```
import java.util.*;
import java.io.PrintWriter;

public class gpaSymTab extends HashMap
{
    final Object a = 1;

    public gpaSymTab()
    {}

    public boolean testValue(String name)
    {
   //System.out.println("Checking if " + name + " exists.");
   Object x = this.get(name);

   if(x != a)
       return false;
   else
       return true;
    }
}
```

gpaMain.java

```
import java.io.*;
import antlr.CommonAST;
import antlr.collections.AST;
import antlr.RecognitionException;
import antlr.TokenStreamException;
import antlr.TokenStreamIOException;

class gpaMain {
    public static void makeTree(String filename, boolean diagnostic,
boolean exec) {
        try {
            InputStream input;
      if ( null != filename )
          input = (InputStream) new FileInputStream(filename);
      else
          throw new IOException();

            gpaLexer lexer = new gpaLexer( input );
            gpaParser parser = new gpaParser( lexer );

            parser.prgrm();

            if ( lexer.nr_error > 0 || parser.nr_error > 0 ) {
                System.err.println( "gpaIR: Parsing errors found. Can't
Continue." );
      System.err.println("Expect assembler errors.");
                return;
            }

            CommonAST tree = (CommonAST)parser.getAST();

       if (diagnostic) {
      System.out.println("gpaIR: +++++++TREE++++++");
      System.out.println();
      System.out.println(tree.toStringList());
       }

       if (exec) {

      gpaWalker walker = new gpaWalker();

      if (diagnostic) {
          //System.out.println("gpaIR: +++CODE-GEN OUTPUT+++");
          walker.expr(tree);
      }

      /*
      if (diagnostic)
          System.out.println("gpaIR: +++PROG RETURN+++");

      if (r != null)
          r.print();
      else
```

```
            System.out.println("gpaIR: r is null");
        */
        System.exit(0);

         }

          } catch( IOException e ) {
              System.out.println( "Error: I/O: " + e );
         System.exit(0);
          } catch( RecognitionException e ) {
              System.out.println( "Error: RecognitionException: " + e );
         System.exit(0);
          } catch( TokenStreamException e ) {
              System.out.println( "Error: TokenStream: " + e );
         System.exit(0);
          } catch( Exception e ) {
              System.out.println( "Error: (what did you do?): " + e );
         System.exit(0);
          }
    }

    public static void main( String[] args ) {

   boolean diagnostic = false;
   boolean exec = false;
   if (args.length < 1) {
       System.out.println("Not Enough arguments!!");
       System.exit(0);
   }
   diagnostic = true;
   exec = true;
   makeTree(args[0],diagnostic,exec);
    }
}
```

Translator.pl

```
#!/usr/bin/perl

my %func_hash = ();
my %var_loc = ();

if(@ARGV == 2)
{
  open(INFILE, $ARGV[0]) or die "Can't open file: $!\n";
  open(OUTFILE, ">" . $ARGV[1]);

  print OUTFILE ".LC0:\n \t.string \"%d\"\n";
  print OUTFILE ".LC1:\n \t.string \"%s\"\n";
  print OUTFILE "\t.text\n";

  $st_count = 2;
  $def_flag = 0;
  $curr_func = "";
  $stack_loc = -12;
  $func_num = 0;
```

```
while($line = <INFILE>)
{
  chomp($line);
  @split_line = 0;
  @split_line = split(/ /, $line);

  for($i = 0;$i < @split_line;$i++)
  {

 if($split_line[$i] =~ /^_(in|out)/)
 {
     if(exists $var_loc{ $curr_func . "_" . $split_line[$i] } )
     {
    ;
     }
     else
     {
    $func_hash{ $curr_func } = $func_hash{ $curr_func } + 1;
    $var_loc{ $curr_func . "_" . $split_line[$i] } = $stack_loc;
    $stack_loc -= 4;
     }
 }
  }

  if ($split_line[0] =~ /^prst$/)
  {
    shift(@split_line);

    print OUTFILE ".LC$st_count:\n \t.string @split_line\n";
    print OUTFILE "\t.text\n";
    ++$st_count;
  }
  if ($split_line[0] =~ /^Error:$/){
 print $line;
 break;
  }
  if($def_flag == 1 && $split_line[0] =~ /^decl$/)
  {
    $func_hash{ $curr_func } = $func_hash{ $curr_func } + 1;
    $var_loc{ $curr_func . "_" . $split_line[2] } = $stack_loc;
    $stack_loc -= 4;
  }
  elsif($def_flag == 1 && $split_line[0] !~ /^decl$/)
  {
    $def_flag = 0;
  }

  if ($split_line[0] =~ /^def$/)
  {
    $stack_loc = -12;
    $def_flag = 1;
    $curr_func = $split_line[1];
    $func_hash{ $curr_func } = 0;

    if($split_line[2] !~ /^~$/)
    {
   $var_loc{ $curr_func . "_" . $split_line[2] } = 8;
```

```
    }
    if($split_line[3] !~ /^~$/)
    {
   $var_loc{ $curr_func . "_" . $split_line[3] } = 12;
    }
    if($split_line[4] !~ /^~$/)
    {
   $var_loc{ $curr_func . "_" . $split_line[4] } = 16;
    }

    $var_loc{ $curr_func . "__out" } = -8;
  }

}
close(INFILE);
open(INFILE, $ARGV[0]) or die "Can't open file: $!\n";

$st_count = 2;
while($line = <INFILE>)
{
  chomp($line);
  @split_line = 0;
  @split_line = split(/ /, $line);

  if($split_line[0] =~ /^add$/)
  {
 if($split_line[1] =~ /^\$/)
 {
     print OUTFILE "\t movl \t $split_line[1], %eax\n";
 }
 else
 {
     $vp = $var_loc{ $curr_func . "_" . $split_line[1] };
     print OUTFILE "\t movl \t $vp(%ebp), %eax\n";
 }

 if($split_line[2] =~ /^\$/)
 {
     print OUTFILE "\t addl \t $split_line[2], %eax\n";
 }
 else
 {
     $vp = $var_loc{ $curr_func . "_" . $split_line[2] };
     print OUTFILE "\t addl \t $vp(%ebp), %eax\n";
 }

 $vp = $var_loc{ $curr_func . "_" . $split_line[3] };
 print OUTFILE "\t movl \t %eax, $vp(%ebp)\n";

  }
  elsif($split_line[0] =~ /^sub$/)
  {
 if($split_line[1] =~ /^\$/)
 {
     print OUTFILE "\t movl \t $split_line[1], %eax\n";
 }
 else
```

```
{
    $vp = $var_loc{ $curr_func . "_" . $split_line[1] };
    print OUTFILE "\t movl \t $vp(%ebp), %eax\n";
}

if($split_line[2] =~ /^\$/)
{
    print OUTFILE "\t subl \t $split_line[2], %eax\n";
}
else
{
    $vp = $var_loc{ $curr_func . "_" . $split_line[2] };
    print OUTFILE "\t subl \t $vp(%ebp), %eax\n";
}

$vp = $var_loc{ $curr_func . "_" . $split_line[3] };
print OUTFILE "\t movl \t %eax, $vp(%ebp)\n";
 }
 elsif($split_line[0] =~ /^mult$/)
 {
      if($split_line[1] =~ /^\$/)
{
    print OUTFILE "\t movl \t $split_line[1], %eax\n";
}
else
{
    $vp = $var_loc{ $curr_func . "_" . $split_line[1] };
    print OUTFILE "\t movl \t $vp(%ebp), %eax\n";
}

if($split_line[2] =~ /^\$/)
{
    print OUTFILE "\t imull \t $split_line[2], %eax\n";
}
else
{
    $vp = $var_loc{ $curr_func . "_" . $split_line[2] };
    print OUTFILE "\t imull \t $vp(%ebp), %eax\n";
}

$vp = $var_loc{ $curr_func . "_" . $split_line[3] };
print OUTFILE "\t movl \t %eax, $vp(%ebp)\n";

 }
 elsif($split_line[0] =~ /^and$/)
 {
      if($split_line[1] =~ /^\$/)
{
    print OUTFILE "\t movl \t $split_line[1], %eax\n";
}
else
{
    $vp = $var_loc{ $curr_func . "_" . $split_line[1] };
    print OUTFILE "\t movl \t $vp(%ebp), %eax\n";
}

if($split_line[2] =~ /^\$/)
```

```
{
    print OUTFILE "\t andl \t $split_line[2], %eax\n";
}
else
{
    $vp = $var_loc{ $curr_func . "_" . $split_line[2] };
    print OUTFILE "\t andl \t $vp(%ebp), %eax\n";
}

$vp = $var_loc{ $curr_func . "_" . $split_line[3] };
print OUTFILE "\t movl \t %eax, $vp(%ebp)\n";

 }
 elsif($split_line[0] =~ /^or$/)
 {
      if($split_line[1] =~ /^\$/)
{
    print OUTFILE "\t movl \t $split_line[1], %eax\n";
}
else
{
    $vp = $var_loc{ $curr_func . "_" . $split_line[1] };
    print OUTFILE "\t movl \t $vp(%ebp), %eax\n";
}

if($split_line[2] =~ /^\$/)
{
    print OUTFILE "\t orl \t $split_line[2], %eax\n";
}
else
{
    $vp = $var_loc{ $curr_func . "_" . $split_line[2] };
    print OUTFILE "\t orl \t $vp(%ebp), %eax\n";
}

$vp = $var_loc{ $curr_func . "_" . $split_line[3] };
print OUTFILE "\t movl \t %eax, $vp(%ebp)\n";

 }
 elsif($split_line[0] =~ /^div$/)
 {
      if($split_line[1] =~ /^\$/)
{
    print OUTFILE "\t movl \t $split_line[1], %eax\n";
}
else
{
    $vp = $var_loc{ $curr_func . "_" . $split_line[1] };
    print OUTFILE "\t movl \t $vp(%ebp), %eax\n";
}

print OUTFILE "\t cltd\n";

if($split_line[2] =~ /^\$/)
{
    print OUTFILE "\t idivl \t $split_line[2]\n";
}
```

```
else
{
    $vp = $var_loc{ $curr_func . "_" . $split_line[2] };
    print OUTFILE "\t idivl \t $vp(%ebp)\n";
}

$vp = $var_loc{ $curr_func . "_" . $split_line[3] };
print OUTFILE "\t movl \t %eax, $vp(%ebp)\n";

 }
 elsif($split_line[0] =~ /^sgt$/)
 {
      if($split_line[1] =~ /^\$/)
{
    print OUTFILE "\t movl \t $split_line[1], %eax\n";
}
else
{
    $vp = $var_loc{ $curr_func . "_" . $split_line[1] };
    print OUTFILE "\t movl \t $vp(%ebp), %eax\n";
}

if($split_line[2] =~ /^\$/)
{
    print OUTFILE "\t cmpl \t $split_line[2], %eax\n";
}
else
{
    $vp = $var_loc{ $curr_func . "_" . $split_line[2] };
    print OUTFILE "\t cmpl \t $vp(%ebp), %eax\n";
}

$vp = $var_loc{ $curr_func . "_" . $split_line[3] };

print OUTFILE "\t setg \t %al\n";
print OUTFILE "\t movzbl \t %al, %eax\n";
print OUTFILE "\t movl \t %eax, $vp(%ebp)\n";
 }
 elsif($split_line[0] =~ /^seq$/)
 {
if($split_line[1] =~ /^\$/)
{
    print OUTFILE "\t movl \t $split_line[1], %eax\n";
}
else
{
    $vp = $var_loc{ $curr_func . "_" . $split_line[1] };
    print OUTFILE "\t movl \t $vp(%ebp), %eax\n";
}

if($split_line[2] =~ /^\$/)
{
    print OUTFILE "\t cmpl \t $split_line[2], %eax\n";
}
else
{
    $vp = $var_loc{ $curr_func . "_" . $split_line[2] };
```

```
    print OUTFILE "\t cmpl \t $vp(%ebp), %eax\n";
}

$vp = $var_loc{ $curr_func . "_" . $split_line[3] };

print OUTFILE "\t sete \t %al\n";
print OUTFILE "\t movzbl \t %al, %eax\n";
print OUTFILE "\t movl \t %eax, $vp(%ebp)\n";
 }
 elsif($split_line[0] =~ /^sne$/)
 {
if($split_line[1] =~ /^\$/)
{
    print OUTFILE "\t movl \t $split_line[1], %eax\n";
}
else
{
    $vp = $var_loc{ $curr_func . "_" . $split_line[1] };
    print OUTFILE "\t movl \t $vp(%ebp), %eax\n";
}

if($split_line[2] =~ /^\$/)
{
    print OUTFILE "\t cmpl \t $split_line[2], %eax\n";
}
else
{
    $vp = $var_loc{ $curr_func . "_" . $split_line[2] };
    print OUTFILE "\t cmpl \t $vp(%ebp), %eax\n";
}

$vp = $var_loc{ $curr_func . "_" . $split_line[3] };

print OUTFILE "\t setne \t %al\n";
print OUTFILE "\t movzbl \t %al, %eax\n";
print OUTFILE "\t movl \t %eax, $vp(%ebp)\n";
 }
 elsif($split_line[0] =~ /^slt$/)
 {
if($split_line[1] =~ /^\$/)
{
    print OUTFILE "\t movl \t $split_line[1], %eax\n";
}
else
{
    $vp = $var_loc{ $curr_func . "_" . $split_line[1] };
    print OUTFILE "\t movl \t $vp(%ebp), %eax\n";
}

if($split_line[2] =~ /^\$/)
{
    print OUTFILE "\t cmpl \t $split_line[2], %eax\n";
}
else
{
    $vp = $var_loc{ $curr_func . "_" . $split_line[2] };
    print OUTFILE "\t cmpl \t $vp(%ebp), %eax\n";
```

```
}

$vp = $var_loc{ $curr_func . "_" . $split_line[3] };

print OUTFILE "\t setl \t %al\n";
print OUTFILE "\t movzbl \t %al, %eax\n";
print OUTFILE "\t movl \t %eax, $vp(%ebp)\n";
 }
 elsif($split_line[0] =~ /^loop$/)
 {
      print OUTFILE ".L$split_line[1]:\n";
 }
 elsif($split_line[0] =~ /^run$/)
 {
      print OUTFILE "\t movl \t -8(%ebp), %eax\n";
      print OUTFILE "\t cmpl \t \$1, %eax\n";
      print OUTFILE "\t jne \t .E$split_line[2]\n";
 }
 elsif($split_line[0] =~ /^pool$/)
 {
print OUTFILE "\t jmp \t .L$split_line[1]\n";
print OUTFILE ".E$split_line[1]:\n";
 }
 elsif($split_line[0] =~ /^if$/)
 {
;
 }
 elsif($split_line[0] =~ /^then$/)
 {
print OUTFILE "\t movl \t -8(%ebp), %eax\n";
      print OUTFILE "\t cmpl \t \$1, %eax\n";
      print OUTFILE "\t jne \t .L$split_line[1]\n";
 }
 elsif($split_line[0] =~ /^neht$/)
 {
print OUTFILE "\t jmp \t .E$split_line[1]\n";
 }
 elsif($split_line[0] =~ /^else$/)
 {
      print OUTFILE ".L$split_line[1]:\n";
 }
 elsif($split_line[0] =~ /^fi$/)
 {
print OUTFILE ".E$split_line[1]:\n";
 }
 elsif($split_line[0] =~ /^end$/)
 {
   print OUTFILE ".RE$func_num:\n";

   if($curr_func !~ /^go$/)
   {
  print OUTFILE "\t leave\n";
  print OUTFILE "\t ret\n";
   }
   else
   {
  $size =  ($func_hash{ $curr_func } + 3 ) * 8;
```

```
  print OUTFILE "\t addl \t \$$size, %esp\n";
  print OUTFILE "\t popl \t %ecx\n";
  print OUTFILE "\t popl \t %ebp\n";
  print OUTFILE "\t leal \t -4(%ecx), %esp\n";
  print OUTFILE "\t ret\n";
   }

   if($split_line[1] !~ /^go$/)
   {

  print OUTFILE "\t .size $split_line[1], .-$split_line[1]\n";
  print OUTFILE "\t .section .rodata\n";
   }
   else
   {
  print OUTFILE "\t .size main, .-main\n";
  print OUTFILE "\t .section .note.GNU-stack,\"\",\@progbits\n";
   }

 }
 elsif($split_line[0] =~ /^def$/)
 {
   ++$func_num;
   if($split_line[1] !~ /^go$/)
   {
  $size =  ($func_hash{ $split_line[1] } + 3 ) * 8;
  print OUTFILE ".globl $split_line[1]\n";
  print OUTFILE "\t .type $split_line[1], \@function\n";
  print OUTFILE "$split_line[1]:\n";
  print OUTFILE "\t pushl \t %ebp\n";
  print OUTFILE "\t movl \t %esp, %ebp\n";
  print OUTFILE "\t subl \t \$$size, %esp\n";
   }
   else
   {
  $size =  ($func_hash{ $split_line[1] } + 3 ) * 8;
  print OUTFILE ".globl main\n";
  print OUTFILE "\t .type main, \@function\n";
  print OUTFILE "main:\n";
  print OUTFILE "\t leal \t 4(%esp), %ecx\n";
  print OUTFILE "\t andl \t \$-16, %esp\n";
  print OUTFILE "\t pushl \t -4(%ecx)\n";
  print OUTFILE "\t pushl \t %ebp\n";
  print OUTFILE "\t movl \t %esp, %ebp\n";
  print OUTFILE "\t pushl \t %ecx\n";
  print OUTFILE "\t subl \t \$$size, %esp\n";
   }

   $curr_func = $split_line[1];
 }
 elsif($split_line[0] =~ /^decl$/)
 {
$vp = $var_loc{ $curr_func . "_" . $split_line[2] };
print OUTFILE "\t movl \t $split_line[1], $vp(%ebp)\n";
 }
 elsif($split_line[0] =~ /^asn$/)
 {
```

```
      if($split_line[1] =~ /^\$/)
{
    print OUTFILE "\t movl \t $split_line[1], %eax\n";
}
else
{
    $vp = $var_loc{ $curr_func . "_" . $split_line[1] };
    print OUTFILE "\t movl \t $vp(%ebp), %eax\n";
}

$vp = $var_loc{ $curr_func . "_" . $split_line[2] };
print OUTFILE "\t movl \t %eax, $vp(%ebp)\n";

 }
 elsif($split_line[0] =~ /^not$/)
 {
      if($split_line[1] =~ /^\$/)
{
    print OUTFILE "\t movl \t $split_line[1], %eax\n";
}
else
{
    $vp = $var_loc{ $curr_func . "_" . $split_line[1] };
    print OUTFILE "\t movl \t $vp(%ebp), %eax\n";
}
print OUTFILE "\t notl \t %eax\n";

$vp = $var_loc{ $curr_func . "_" . $split_line[2] };
print OUTFILE "\t movl \t %eax, $vp(%ebp)\n";

 }
 elsif($split_line[0] =~ /^ret$/)
 {
if($split_line[1] =~ /^\$/)
{
    print OUTFILE "\t movl \t $split_line[1], %eax\n";
}
else
{
    $vp = $var_loc{ $curr_func . "_" . $split_line[1] };
    print OUTFILE "\t movl \t $vp(%ebp), %eax\n";
}
print OUTFILE "\t jmp \t .RE$func_num\n";
 }
 elsif($split_line[0] =~ /^prnm$/)
 {
      if($split_line[1] =~ /^\$/)
{
    print OUTFILE "\t movl \t $split_line[1], %eax\n";
}
else
{
    $vp = $var_loc{ $curr_func . "_" . $split_line[1] };
    print OUTFILE "\t movl \t $vp(%ebp), %eax\n";
}
print OUTFILE "\t movl \t %eax, 4(%esp)\n";
print OUTFILE "\t movl \t \$.LC0, (%esp)\n";
```

```
print OUTFILE "\t call \t printf\n";
 }
 elsif($split_line[0] =~ /^prst$/)
 {
      print OUTFILE "\t movl \t \$.LC$st_count, (%esp)\n";
print OUTFILE "\t call \t printf\n";
++$st_count;
 }
 elsif($split_line[0] =~ /^input$/)
 {
print OUTFILE "\t leal \t -8(%ebp), %eax\n";
print OUTFILE "\t movl \t %eax, 4(%esp)\n";
print OUTFILE "\t movl \t \$.LC1, (%esp)\n";
print OUTFILE "\t call \t scanf\n";
print OUTFILE "\t leal \t -8(%ebp), %eax\n";
print OUTFILE "\t movl \t %eax, (%esp)\n";
print OUTFILE "\t call \t atoi\n";
$vp = $var_loc{ $curr_func . "_" . $split_line[1] };
print OUTFILE "\t movl \t %eax, $vp(%ebp)\n";
 }
 elsif($split_line[0] =~ /^call$/)
 {
      if($split_line[4] =~ /^\$/)
{
    print OUTFILE "\t movl \t $split_line[4], %eax\n";
}
else
{
    $vp = $var_loc{ $curr_func . "_" . $split_line[4] };
    print OUTFILE "\t movl \t $vp(%ebp), %eax\n";
}
      print OUTFILE "\t movl \t %eax, 8(%esp)\n";

      if($split_line[3] =~ /^\$/)
{
    print OUTFILE "\t movl \t $split_line[3], %eax\n";
}
else
{
    $vp = $var_loc{ $curr_func . "_" . $split_line[3] };
    print OUTFILE "\t movl \t $vp(%ebp), %eax\n";
}
      print OUTFILE "\t movl \t %eax, 4(%esp)\n";

      if($split_line[2] =~ /^\$/)
{
    print OUTFILE "\t movl \t $split_line[2], %eax\n";
}
else
{
    $vp = $var_loc{ $curr_func . "_" . $split_line[2] };
    print OUTFILE "\t movl \t $vp(%ebp), %eax\n";
}
      print OUTFILE "\t movl \t %eax, (%esp)\n";
print OUTFILE "\t call \t $split_line[1]\n";
print OUTFILE "\t movl \t %eax, -8(%ebp)\n";
 }
```

```
    else{ print OUTFILE "$split_line[0] \n"; }
  }

}
else
{
    print "Incorrect usage! see README.txt\n";
}

foreach $key (sort keys %func_hash)
{
    $value =  $func_hash{ $key };
    print OUTFILE "#$key has $value variables.\n";
}

foreach $key (sort keys %var_loc)
{
    $value =  $var_loc{ $key };
    print OUTFILE "#$key is stored at $value.\n";
}
```

script_run (Processes the ANTLR files and compiles the java files)

```
#!/bin/bash

runantlr gpa.g;
runantlr walker.g;
javac gpa*.java;
```

gpa

```
#!/bin/bash

flag=`java gpaMain $1 > output.ir`
echo $flag

if  [ "$flag" != "" ]; then
    echo "Exiting with errors..."
    exit 1
else
    temp=`./translator.pl output.ir output.s`
    echo $temp
    if  [ "$temp" != "" ]; then
   echo "Exiting with errors..."
   exit 1
    else
   gcc -o output output.s
    fi
fi
```