IS744 - Proyecto 4 : Generación de Código

Fecha de Entrega: Jueves, 9 de Junio de 2016, 11:59 a.m.

Bien, si han llegado hasta aquí, su compilador esta listo hacia la producción de código. En este proyecto, se extenderá su compilador para que produzca código ensamblador ejecutable para ser usado en una estación de trabajo Sun SPARC. Al final de este proyecto, su compilador finalmente será capaz de crear programas MiniPascal ejecutables.

NO ESPERE HASTA EL ÚLTIMO MINUTO PARA INICIAR ESTE PROYECTO!

Corrigiendo el Proyecto 3

Si su implementación de revisión de tipos del proyecto 3 está mal, es absolutamente crítico que la haga trabajar. Por el momento, la parte más difícil del proyecto 3 es la inferencia de tipo en funciones. Si se sienten agotados o frustados con esa parte del proyecto, aquí hay una solución simple que les permitirá continuar con el proyecto 4: basta con establecer todos los tipos de retorno de la función a un "int". Obviamente, hacer esto no les permitirá pasar cualquier prueba final. Sin embargo, ustedes todavía son capaces de pasar un gran número de pruebas.

Si su compilador aún no analiza correctamente los programas, sería una muy buena idea que empiecen a trabajar tan pronto como sea posible. Por favor, consulten.

Introducción

Su tarea principal en este proyecto es tomar el árbol de sintaxis creado por su parser y producir un archivo de código ensamblador SPARC que pueda ser ensamblado en un programa ejecutable.

Por ejemplo, si un usuario proporciona un programa MiniPascal `hello.pas` como esto:

```
fun main()
   begin
       print("Hola Mundo\n")
   end
```

La salida de su compilador debe ser un archivo `hello.s` que se vea algo como esto:

```
! Creado por hoc.py

    .section     ".text"

! function: main

    .global main
    .type main,2

main:

    save %sp, 64, %sp
    sethi %hi(.L1), %o0
    or    %o0, %lo(.L1), %o0
```

```
        call  hlprint
        nop

        call _exit
        nop

        .section     ".rodata"

    .L1:   .asciz  "Hola Mundo\n"
```

Este archivo será ensamblado con el ensamblador de Sun y enlazado con una pequeña biblioteca MiniPascal para producir un ejecutable. Por ejemplo:

```
% python hoc.py hello.hoc
% as hello.s
% ld hello.o hoc.o -lc
% a.out
Hola Mundo
%
```

Este proyecto no asume ninguna experiencia previa con el lenguaje ensamblador SPARC (detalles a continuación). Sin embargo, deberán hacer una cantidad considerable de experimentación para asegurarsen que las cosas trabajan correctamente.

Parte 0 - Detalles Administrativos

Para completar esta parte del proyecto se requiere tener acceso a una máquina Sun Solaris. En el peor caso, contar con un emulador (QEMU). La otra posibilidad es tener acceso a una de las máquinas de SIRIUS. Pensaremos en eso más adelante.

Parte 1 - Limpieza de la Interfaz

Hasta ahora, se han desarrollado un par de módulos de diferente análisis como `hoclex.py` y `hocparse.py`. Antes de iniciar con la generación de código, escriba un nuevo módulo que proporcione un nuevo programa principal para su compilador.

1. Crear un archivo 'hoc.py'.

2. Escriba un script o programa principal que comience por obtener el nombre del archivo desde la línea de comandos y establezca el nombre del archivo de salida. Por ejemplo:

   ```
   import sys
   import os.path

   filename = sys.argv[1]
   outname = os.path.splitext(filename)[0] + ".s"
   ```

3. Leer el archivo de entrada e invocar el analizador sintático. Por ejemplo:

   ```
   f = open(filename)
   data = f.read()
   f.close()

   import hocparse
   ```

```
top = hocparse.parse(data)
```

La salida de su analizador debe ser el nodo raíz del árbol de sintaxis bien formado representando el contenido del achivo de entrada. Si un error de sintaxis o de tipo ha ocurrido durante el análisis, es posible que su analizador retorne None para indicar un error.

4. Crear un archivo nuevo `hocgen.py` y colocar las siguientes funciones en él:

```
def generate(file,top):
    print >>file, "! Creado por hoc.py"
    print >>file, "! Sunombre, IS744 (2015-2)"
```

5. Modifique el programa principal para llamar a `generate()` como sigue:

```
top = hocparse.parse(data)
if top:
     import hocgen
     outf = open(outfile,"w")
     hocgen.generate(outf,top)
     outf.close()
```

6. Probar la nueva interfaz ejecutándola con un programa sencillo. Su compilador deberá producir un archivo .s como esto:

```
% python hoc.py simple.pas
% more simple.s
! Creado por hoc.py
! Sunombre, IS744 (2015-2)
%
```

Felicitaciones, su compilador acaba de producir su primer archivo de salida.

7. Para simplificar la depuración puede que quiera añadir algo de comprobación de errores o comandos especiales de línea a su programa principal. Por ejemplo, adicionar soporte para una opción "-t" que imprima el árbol de sintaxis. No se revisará nada de esto--esto es solamente usado para ayudar con la depuración del compilador.

Parte 2 - Recorriendo el Árbol de Sintaxis

A continuación su tarea es escribir código que recorra su árbol de sintaxis y busque tipos de nodo específicos. Piense en como su programa está estructurado. Por ejemplo, un programa es una lista de funciones, una función es una lista de declaraciones, una declaración puede ser un read(), write(), print(), asignación, while, if-else, break, return y así sucesivamente.

Para recorrer el árbol, cree una colección de funciones que trate de imprimir información acerca de cada declaración en su program fuente. Por ejemplo (nota: deberá modificar este código para que coincida con su representación del árbol de sintaxis):

```
def emit_program(out,top):
    print >>out,"\n! program"
    funclist = (get list of functions)
    for f in funclist:
          emit_function(out,f)
```

```
def emit_function(out,func):
    print >>out,"\n! function: %s (start) " % func.name
    ...
    statements = (get list of statements)
    emit_statements(out,statements)
    ...
    print >>out,"! function: %s (end) " % func.name

def emit_statements(out,statements):
    for s in statements:
         emit_statement(out,s)

def emit_statement(out,s):
    if s.name == 'print':
         emit_print(out,s)
    elif s.name == 'read':
         emit_read(out,s)
    elif s.name == 'write':
         emit_write(out,s):
    elif s.name == 'while':
         emit_while(out,s):
    ...

def emit_print(out,s):
    print >>out, "\n! print (start)"
    ...
    print >>out, "! print (end)"

def emit_while(out,s):
    print >>out, "\n! while (start)"
    ...
    statement = (get body of while)
    emit_statement(out,statement)
    ...
    print >>out, "! while (end)"
```

Para cada declaración en su programa, cree la función emit correspondiente que imprima un marcador de inicio (start) y final (end) como se muestra en el ejemplo. Nota: líneas iniciando con "!" son comentariones en ensamblador.

Ahora, modifique su función `generate()` para recorrer a través de un programa e imprimir información acerca de cada función y declaraciones individuales. Por ejemplo, si se facilitó el siguiente programa:

```
/* Calcula GCD */
fun main()
    x: int;
    y: int;
    g: int;
    begin
       read(x);
       read(y);
       g := y;
       while x > 0 do
           begin
              g := x;
              x := y - (y/x)*x;
              y := g
           end;
       write(g)
```

```
    end
```

La salida de su compilador deberá ser más o menos como esto:

```
! Creado por hoc.py
! Sunombre, IS744 (2015-2)

! program

! function: main (start)

! read (start)
! read (end)

! read (start)
! read (end)

! assign (start)
! assign (end)

! while (start)

! assign (start)
! assign (end)

! assign (start)
! assign (end)

! assign (start)
! assign (end)

! while(end)

! write (start)
! write (end)

! function: main (end)
```

Notas:

- Su salida solo deberá incluir información acerca de declaraciones y la estructura general del programa. Detalles sobre expresiones, lista de parámetros o declaraciones no serán impresas.
- Ignore funciones anidadas por ahora.
- Por favor incluya los mensajes de inicio (start) y fin (end) para cada tipo de declaración. Esto probará ser útil mas tarde (doy mi palabra).
- Si se hizo bien el trabajo en el analizador sintático, no tendrá que ejecutar ninguna clase de chequeos del árbol de sintaxis---revisiones de tipo deben haber capturado todos los problemas potenciales antes de iniciar esta fase del compilador.

Parte 3 - Expresión Pseudocódigo

Uno de los métodos más sencillos para evaluar expresiones aritméticas es utilizar una pila de evaluación de expresión (si alguna vez a utilizado una calculadora HP con RPN, ha visto esta técnica en acción). En

pocas palabras, aquí están las reglas:

- Mantenga una pila de valores
- Cuando aparecen números e identificadores, ellos son puestos en la pila.
- Un operador binario (por ejemplo, "+") extrae dos valores de la pila, calcula el resultado y lo coloca en la pila.
- Un operador unario extrae el valor superior de la pila, cambia el valor e inserta el resultado.
- Para obtener el valor de una expresión, simplemente extraígalo de la pila.

Por ejemplo, si se tiene la expresión 2+3*4-9, puede ser evaluada como sigue:

```
push 2          # Pila : 2
push 3          # Pila : 3 2
push 4          # Pila : 4 3 2
mutiply         # Pila : 12 2
add             # Pila : 14
push 9          # Pila : 9 14
sub             # Pila : 5
val := pop      # Pila :          # val obtiene el valor 5
```

Escriba una función `eval_expression()` que produce pseudocódigo para la evaluación de expresión usando esta ténica. Por ejemplo:

```
def eval_expression(out,expr):
    if expr.name == 'number':
          print >>out, "!   push", expr.value
    elif expr.name == 'id':
          print >>out, "!   push", expr.value
    elif expr.name == '+':
          left = expr.children[0]
          right = expr.children[1]
          eval_expression(out,left)
          eval_expression(out,right)
          print >>out, "!   add"
    elif expr.name == '-':
          left = expr.children[0]
          right = expr.children[1]
          eval_expression(out,left)
          eval_expression(out,right)
          print >>out, "!   sub"
    ...
```

Además, producir pseudocódigo para las operaciones de relación y operadores booleanos (and, or, not).

Ahora, modifique las funciones emit para las declaraciones que involucren expresiones para producir pseudocódigo de expresión. Por ejemplo:

```
def emit_write(out,s):
    print >>out, "\n! write (start)"
    expr = (get expr from s)
    eval_expression(out,expr)
    print >>out, "!   expr := pop"
    print >>out, "!   write(expr)"
    print >>out, "! write (end)"
```

Con estas modificaciones, la salida de su compilador se verá algo como esto:

```
! Creado por hoc.py
! Sunombre, IS744 (2015-2)

! program

! function: main (start)

! read (start)
! read (end)

! read (start)
! read (end)

! assign (start)
!   push y
!   g := pop
! assign (end)

! while (start)
!   push x
!   push 0
!   gt
!   relop := pop

! assign (start)
!   push x
!   g := pop
! assign (end)

! assign (start)
!   push y
!   push y
!   push x
!   div
!   push x
!   mul
!   sub
!   x := pop
! assign (end)

! assign (start)
!   push g
!   y := pop
! assign (end)

! while(end)

! write (start)
!   push g
!   expr := pop
!   write(expr)
! write (end)

! function: main (end)
```

Notas:

- Para llamados de función, simplemente produzca pseudocódigo tal como "! push blah()"
- Los arreglos involucrarán multiples niveles de evaluación de expresión (para calcular el índice).

Para manejar el acceso a un arreglo tal como x:= 2 + a[4*i+10], su pseudocódigo se verá más o menos como esto:

```
!   push 2
!   push 4
!   push i
!   mul
!   push 10
!   add
!   index := pop
!   push a[index]
!   add
!   x := pop
```

Parte 4 - Control de Flujo Pseudocódigo

Ahora, modifique las funciones emit para las declaraciones de control de flujo a fin de incluir un poco más de información acerca del control de flujo del programa. Por ejemplo:

```
def emit_while(out,s):
    print >>out, "\n! while (start)"
    print >>out, "! test:"
    relop = (get relation expression from s)
    eval_expression(out,relop)
    print >>out, "!   relop := pop"
    print >>out, "!   if not relop: goto done"

    statement = (get statement from s)
    emit_statement(out,statement)

    print >>out, "! goto test"
    print >>out, "! done:"
    print >>out, "! while (end)"
```

Con estos cambios, su programa deberá tener este aspecto:

```
! Creado por hoc.py
! Sunombre, IS744 (2015-2)

! program

! function: main (start)

! read (start)
! read (end)

! read (start)
! read (end)

! assign (start)
!   push y
!   g := pop
! assign (end)

! while (start)
! test:
!   push x
!   push 0
```

```
!   gt
!   relop := pop
!   if not relop: goto done

! assign (start)
!   push x
!   g := pop
! assign (end)

! assign (start)
!   push y
!   push y
!   push x
!   div
!   push x
!   mul
!   sub
!   x := pop
! assign (end)

! assign (start)
!   push g
!   y := pop
! assign (end)

! goto test
! done:
! while(end)

! write (start)
! push g
! expr := pop
! write(expr)
! write (end)

! function: main (end)
```

Notas:

- La mayoría de las construcciones de flujo tienen puntos de entrada y salida. Por ejemplo, la estructura de un bucle while puede tener este aspecto:

```
test:
    calcula expresion relacional
    if false: goto done

    statements
    goto test           # Vuelve para otra prueba

done:
    ... continua con programa ...
```

- Una declaración if es muy similar. Por ejemplo:

```
calcula expresion relacional
if false: goto else

statements for true
goto next
```

```
        else:

        statements for false

        next:
        ... proxima declaracion ...
```

Parte 5 - Llamado a Funciones

A continuación, modifique la evaluación de expresión y el código de declaración para un llamado a función. Esto es realmente muy fácil---sólo evalue cada argumento de la función separadamente (como una expresión) y "guarde" el resultado. Por ejemplo, si se tiene un llamado a función como esto:

```
x := foo(3,2*x+y,b) + 2
```

El pseudocódigo sería así:

```
!  push 3
!  arg1 := pop
!  push 2
!  push x
!  mul
!  push y
!  add
!  arg2 := pop
!  push b
!  arg3 := pop
!  push foo(arg1,arg2,arg3)
!  push 2
!  add
!  x := pop
```

Pausa

Por ahora, la idea general de lo que está pasando debe de comenzar a ser un click. Todos los comentarios del pseudocódigo están realmente describiendo, en cierto sentido, la operación del programa a bajo nivel. De hecho, lo que realmente han hecho es crear una representación informal intermedia del programa. Si estuviera escribiendo un compilador real, debieron haber decidido generar alguna clase de representación intermedia que más adelante podrían apuntar a muchos tipos diferentes de máquinas. Sin embargo, para nuestro proyecto, ahora empezaremos a producir código ensamblador para SPARC (usando el pseudocódigo como guía).

Parte 6 - Estructura del Programa

El archivo .s producido por su compilador necesita dividir datos entre un segemento de texto y otro de datos. El segmento de texto contiene todas las instrucciones de máquina para el programa. El segmento de datos contiene las constantes y variables globales. Para MiniPascal, lo único en el segmento de datos son las literales de cadena y constantes numéricas (si es necesario).

La estructura general de su salida con estos dos segmentos es especificada en el ensamblador Sun de la siguiente manera:

```
! Creado por hoc.py
```

```
! Sunombre, IS744 (2015-2)

    .section     ".text"

Instructions

    .section     ".rodata"

Data
```

Puesto que estas dos secciones son generalmente creadas al mismo tiempo, la forma más fácil para hacer frente al segmento de datos es crear un archivo de cadena en-memoria de esta manera:

```
import StringIO
data = StringIO.StringIO()

...
def emit_print(out,n):
    value = (get string literal value)
    label = new_label()
    # Drop a literal in the data segment
    print >>data, '%s:   .asciz  "%s"' % (label, value)
    ...
    # emit rest of print statement

def generate(out,top):
    ...
    print >>out,  '     .section  ".text"'
    print >>data, '     .section  ".rodata"'
    emit all of the code
    ...
    # Append the data segment at the end of the output
    print >>out, data.getvalue()
```

Dentro del código ensamblador creado por su compilador, tendrá que darle un nombre a varias cosas. Estos nombres se conocen a menudo como etiquetas. Las etiquetas se indican siempre por un nombre y dos puntos de esta manera:

```
     .section ".text"

label1:
     statements

label2:
     statements

     .section ".rodata"

label3:
     data

label4:
     data
```

Como regla general, las únicas etiquetas explícitas definidas por un programa MiniPascal son los nombres de funciones. Para las otras etiquetas, se deberá escribir una función de utilidad `new_label()` que fabrique un nuevo nombre de etiqueta cada vez que es llamada. Por lo general, estos nombres fabricados corresponden a partes de la construcción del flujo de control tales como if y while. También pueden

referir a datos globales en el segmento de datos. Normalmente, estos tipos de etiquetas especiales reciben nombres como ".L12" o ".L23" para que no entren en conflicto con nombres válidos de funciones de programa (los cuales nunca comienzan con un ".").

Además, algunas etiqueta pueden ser declaradas como globales para que puedan ser accesadas externamente. La directiva `.global` es usada para definir una etiqueta global. Todas sus funciones deberán ser declaradas de esta manera. Por ejemplo:

```
      .global main
main:
      statements
```

Para ilustrar el uso de etiquetas, modifique sus funciones emit para insertar etiquetas en un lugar apropiado. Por ejemplo,

```
def emit_while(out,s):
    print >>out, "\n! while (start)"


    test_label = new_label()
    done_label = new_label()
    print >>out, "%s:" % test_label

    relop = (get relation expression from s)
    eval_expression(out,relop)
    print >>out, "!   relop := pop"
    print >>out, "!   if not relop: goto %s" % done_label

    statement = (get statement from s)
    emit_statement(out,statement)

    print >>out, "! goto %s" % test_label
    print >>out, "%s:" % done_label

    print >>out, "! while (end)"
```

Con esto cambios, su salida deberá parecerse a esto (observe el uso de .L1, .L2 y main):

```
! Creado por hoc.py
! Sunombre, IS744 (2015-2)

    .section     ".text"

! program

! function: main (start)

        .global main
main:

! read (start)
! read (end)

! read (start)
! read (end)

! assign (start)
!   push y
```

```
!   g := pop
! assign (end)

! while (start)

.L1:

!   push x
!   push 0
!   gt
!   relop := pop
!   if not relop: goto .L2

! assign (start)
!   push x
!   g := pop
! assign (end)

! assign (start)
!   push y
!   push y
!   push x
!   div
!   push x
!   mul
!   sub
!   x := pop
! assign (end)

! assign (start)
!   push g
!   y := pop
! assign (end)

! goto .L1

.L2:

! while(end)

! write (start)
! push g
! expr := pop
! write(expr)
! write (end)

! function: main (end)

     .section ".rodata"
```

Parte 7 - Llamados a Función, marcos de pila y registros

La forma general de una subrutina en SPARC es la siguiente:

```
funcname:
        save %sp, -96, %sp
        ...
        instructions
```

```
        ...
        ret
        restore
```

La primera instrucción (save) se utiliza para asignar un nuevo marco de pila. El número 96 es el tamaño del marco de pila en bytes. El número real que se debe suministrar aquí dependerá del número de variables locales, temporales y convenciones de llamado a función usados por su compilador. **El tamaño del marco de pila debe ser siempre mayor a 64 bytes y ser un múltiplo de 8**. La instrucción `ret` es usada para retornar desde la subrutina. La instrucción `restore` restablece el puntero de pila a su valor anterior (deshacer los efectos de la instrucción `save`). (Nota: SPARC tiene retardo de ramas por lo que la instrucción `restore` aparece inmediatamente después de `ret` ejecutándose siempre antes de la instrucción `ret` para que tome efecto).

Dentro del cuerpo de una función, dos registros son usados para mantener información acerca de la ubicación del marco de pila. El registro `%fp` contiene la dirección de la parte superior del marco de pila (el valor de `%sp` antes de ejecutar la instrucción `save`). El registro `%sp` contiene la dirección de la parte inferior del marco de pila (después de la instrucción `save`). Tenga en cuenta que la pila crece hacia abajo ya que el valor de `%fp` es *siempre* más grande que el valor de `%sp`.

Los primeros 64 bytes inmediatamente por encima del puntero de pila `%sp` siempre están reservados por el sistema para almacenamiento de registro. **Su compilador no debe acceder o almacenar datos en el rango de direcciones de memoria [%sp, %sp+63]!**.

Las variables locales y temporales están almacenadas debajo del puntero del marco. Por ejemplo, si se tiene una función MiniPascal como esta:

```
fun foo()
   x : int;
   y : float;
   a : int;
   b : int[20];
   begin
        ...
   end
```

Tendría que destinar al menos 4+4+4+4*20=92 bytes de memoria para las variables locales. Así, el inicio de su procedimiento deberá tener este aspecto:

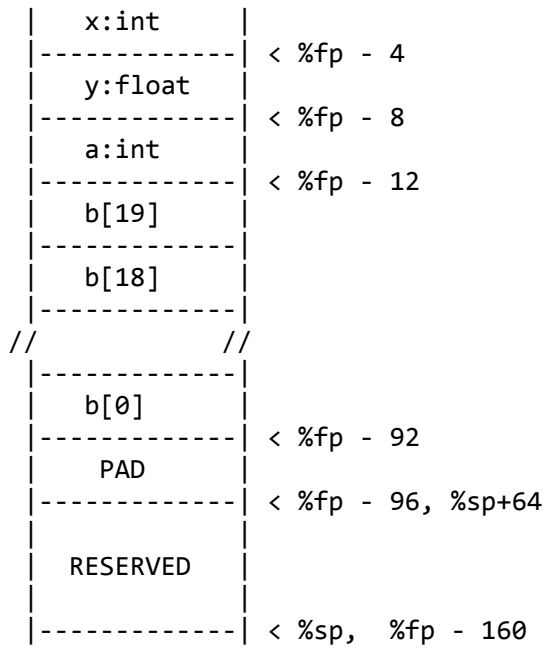
```
foo:
     save %sp, -160, %sp
     ...
```

El valor de 160 se formado tomando los 92 bytes de memoria necesaria para las variables locales, adicionando 64 bytes para el almacenamiento de registro y adicionar 4 bytes de relleno para hacer el valor un múltiplo de 8. Por supuesto, si su procedimiento tenía variables temporales adicionales, se tendría que asignar espacio extra para ellas.

Las variables locales se pueden almacenar en las siguientes posiciones de memoria:

```
x:int          -> [%fp - 4]
y:float        -> [%fp - 8]
a:int          -> [%fp - 12]
b:int[20]      -> [%fp - 92]

                  _____________  < %fp
```



Dentro de cada procedimiento, los siguientes registros de CPU se podrán utilizar para cálculos temporales.

- `%l0 - %l7`. Locales.
- `%i0 - %i5`. Entradas.
- `%o0 - %o5`. Salidas.
- `%g0 - %g7`. Globales.

Todo procedimiento automáticamente obtiene su propio conjunto privado de registros locales `%l0-%l7` (`save` se encarga de ello). Estos registros son solamente visibles para el procedimiento actual y nunca se destruyen o modifican por otras subrutinas. Su compilador deberá usar estos registros para ejecutar evaluación de expresiones y otros cálculos intermedios.

Los registros de entrada `%i0-%i5` comúnmente se utilizan para mantener los primeros 6 argumentos de entrada de una función.

Los registros de salida `%o0-%o5` son usados para mantener los parámetros de salida cuando un procedimiento quiere llamar a otra subrutina. Hay una relación especial entre los registros %i y %o--- específicamente, cuando un llamado a procedimiento es hecho, los registros de salida en el llamante se convierten en los registros de entrada en el procedimiento llamado. Este cambio se debe realmente al comportamiento de las ventanas de registro de SPARC (descrito más adelante).

Los registros globales `%g0-%g7` son compartidos por todos los procedimientos y sólo deben utilizarse para cálculos provisionales como calcular direcciones de memoria. El registro `%g0` es cableado al valor de 0. Los registros `%g4-%g7` son reservados por el sistema operativo y generalmente deben ser evitados por su compilador.

El registro `%i6` es el mismo que el puntero de marco `%fp`. El registro `%o6` es el mismo que el puntero de pila `%sp`. El registro `%i7` contiene la dirección retornada de una subrutina (el valor del PC que se restaurará cuando un procedimiento retorna).

Instrucciones específicas:

- Modificar su generador de código para que produzca código apropiado para la entrada y salida para

cada función. Para empezar, cablee su compilador para que produzca el siguiente código:

```
funcname:
        save %sp, -128, %sp
        ...
.Ln:
        ret
        restore
```

La etiqueta especial `.Ln` al final se utiliza para etiquetar la localización del retorno de la función. Se va a utilizar esto un poco más adelante.

- Para la función main(), necesitará añadir algun procedimiento especial al terminar la ejecución del programa. En este caso, modifique el código de la salida de la siguiente manera:

```
.Ln:
         mov 0, %o0      ! solamente aparece en main
         call _exit      ! solamente aparece en main
         nop             ! solamente aparece en main
         ret
         restore
```

La biblioteca en tiempo de ejecución de MiniPascal

Una serie de declaraciones en la especificación de MiniPascal son usados para E/S. Específicamente `read()`, `write()` y `print()`. Para implementar estas declaraciones, una pequeña biblioteca en tiempo de ejecución es proporcionada en el archivo `hoc.o`. Las funciones en esta biblioteca son las siguientes (prototipos en C).

```
void  flwritei(int x);            /* Escribe un entero */
void  flwritef(float f);          /* Escribe un real */
int   flreadi();                  /* Lee un entero */
float flreadf();                  /* Lee un real */
void  hlprint(char *s);           /* Imprime una cadena */
```

Para llamar a estas funciones desde el lenguaje ensamblador, deberá usar la siguiente secuencia de código:

```
! call flwritei(int)
         mov  val, %o0
         call flwritei
         nop

! call flwritef(float)
         mov val, %o0
         call flwritef
         nop

! call flreadi()
         call flreadi
         nop
         st  %o0, result

! call flreadf()
         call flreadf
         nop
         st %o0, result
```

```
! call hlprint()
         sethi %hi(addr), %o0
         or    %o0, %lo(addr), %o0
         call hlprint
         nop
```

En estos ejemplos, *val* se supone es un registro y *result* se supone es una posición de memoria. *addr* es la dirección de la literal de cadena en el segmento de datos del programa. Consulte la siguiente sección.

Parte 8 - Hola Mundo

Uf, ahora es el momento para compilar el programa tradicional de hola mundo. Aquí está el programa en MiniPascal:

```
fun main()
    begin
        print("Hola Mundo\n")
    end
```

Para ello, implemente la función emit para la declaración `print` para que produzca el siguiente código:

- En el segmento de datos (ver Parte 6), escriba una declaración como esta:

```
.Ln:       .asciz    "Hola Mundo\n"
```

- Ahora, en el segmento de texto, escriba el siguiente código en ensamblador

```
sethi %hi(.Ln), %o0
or    %o0, %lo(.Ln), %o0
call  hlprint
nop
```

El nombre de etiqueta `.Ln` deberá ser fabricada por su compilador. No importa que pueda ser tan grande ya que no puede ser el mismo que otra etiqueta.

Ahora, vamos a probar su programa hola mundo:

```
% hoc.py hello.pas
% cat hello.s
! Creado por hoc.py
! Sunombre, IS744 (2015-2)

     .section ".text"

! program

! function: main (start)

        .global main
main:
       save %sp, -128, %sp
       sethi %hi(.L2), %o0
       or    %o0, %lo(.L2), %o0
       call  hlprint
       nop
```

```
.L1:
       mov  0, %o0
       call _exit
       nop
       ret
       restore

! function: main (end)

     .section ".rodata"

.L2:   .asciz  "Hello World\n"
% as hello.s
% ld hello.o hoc.o -lc
% a.out
Hola Mundo
%
```

Felicitaciones, acaba de compilar su primer programa ejecutable en MiniPascal.

Parte 9 - Asignación de almacenamiento para variables locales

Ahora, es el momento para que pueda hacerle frente al problema de gestión de almacenamiento para las variables locales.

Las variables locales de cada función deben de estar contenidas en algun tipo de lista o estructura de árbol de sintaxis creado en los proyectos 2/3. En esta parte del proyecto, se recorrerá a través de esta lista y se averiguará donde las variables se suponen deben ser almacenadas en el marco de pila de la función. Recordemos que las variables locales son usualmente almacenadas en la parte superior del marco de pila (realmente por debajo de %fp).

Primero, se necesita conocer el tamaño de cada tipo de datos:

```
int           4 bytes (32 bits)
float         4 bytes (32 bits)
int[n]        4*n bytes
float[n]      4*n bytes
```

Ahora, escriba una función `allocate_locals()` que itere sobre la lista de variables locales y asigne un desplazamiento de marco para cada variable. **Este desplazamiento debe ser guardado en la entrada de la tabla de símbolos para cada variable local.** Por ejemplo, si se tiene la función,

```
fun main()
   x : int;
   y : float;
   a : int;
   b : int[20];
   begin
        ...
   end
```

se debe asignar los siguientes valores de desplazamiento (ver parte 7):

```
x.symtab.frame_offset = -4      [%fp - 4]
y.symtab.frame_offset = -8      [%fp - 8]
a.symtab.frame_offset = -12     [%fp - 12]
```

```
b.symtab.frame_offset = -92     [%fp - 92]
```

La función `allocate_locals()` deberá retornar el número total de bytes necesarios para almacenar las variables locales.

Finalmente, realice las siguientes modificaciones a su generador de código:

- En el código de emisión de una función, llame `allocate_locals()` para averiguar cuanto espacio de variables locales es necesario y asignar desplazamiento de marco a las variables locales.

- Modifique el código que imprime la declaración "save" para producir un cálculo preciso del tamaño del marco. El tamaño del marco es igual a 64 mas el valor retornado por `allocate_locals()` redondeado al múltiplo de 8 más cercano. Para la función anterior se deberá producir `save %sp, -160, %sp` en vez del cableado `save %sp, -128, %sp` que se usó para el programa de hola mundo.

Parte 10 - Evaluación de Expresiones Simples

En la parte 3, se generó pseucódigo para evaluar expresiones usando una pila. Ahora es el momento para traducir esto al hardware real. Hacer esto no es tan difícil si se piensa acerca del problema en el lado derecho de la mente.

La forma más fácil de traducir el pseudocódigo de pila a una implementación real es tratar los registros %l0-%l7 como representando siempre los 8 elementos superiores de la pila de expresión. Así, si se desea evaluar la expresión 2+3*4-9, puede ser evaluada de la siguiente forma:

```
Ensamblado SPARC                  Pseudocódigo
-----------------------------------------------
mov   2, %l0                        push 2
mov   3, %l1                        push 3
mov   4, %l2                        push 4
mov   %l1,%o0
call  .mul                          multiply
mov   %l2,%o1
mov   %o0, %l1
add   %l0,%l1,%l0                   add
mov   9, %l1                        push 9
sub   %l0,%l1,%l0                   sub
st    %l0, value                    val := pop
```

Para implementar esto, cree un par de funciones que implementen la funcionalidad de una máquina de pila. Por ejemplo, su implementación podría tener las siguientes funciones o métodos:

- push(). Prepara para agregar un elemento a la pila. Retorna un nombre de registro donde el dato deberá ser colocado.

- pop(). Extrae un elemento de la pila. Retorna un nombre de registro donde el dato está guardado. Si el elemento requerido ha sido guardado en memoria, deberá ser recuperado y colocado en un registro.

Este es un breve ejemplo de como se utilizan estas funciones para implementar el ejemplo anterior:

```
print >>out,"mov 2, %s" % push()
print >>out,"mov 3, %s" % push()
```

```
print >>out,"mov 4, %s" % push()
r = pop()
l = pop()
print >>out,"mov %s, %%o0" % l
print >>out,"call .mul"
print >>out,"mov %s, %%o1" % r
print >>out,"mov %%o0, %s" % push()
r = pop()
l = pop()
print >>out,"add %s, %s, %s" % (l,r,push())
print >>out,"mov 9, %s" % push()
r = pop()
l = pop()
print >>out,"sub %s, %s, %s" % (l,r,push())
result = pop()
print >>out,"st %s, %s" % (result,val)
```

Un problema con la implementación de la pila es que si se tienen más de 8 valores en ella, los valores viejos deberán ser derramados en memoria para dar cabida a los nuevos valores. Para controlar el derrame en memoria, cree una variable temporal region que se encuentra debajo de todas las variables locales en el marco de pila de la función. Ahora, modifique la función push() y pop() como sigue:

- Si ho hay más registros, push() deberá producir código que guarde los registros más antiguos a memeoria e incremente una variable contador temporal. El nombre de los registros viejos deberá estar disponible.

- Si la operación pop() es ejecutada, pero el elemento requerido no está almacenado en un registro, se deberá emitir algo de código para cargar el valor antiguo desde memoria dentro de un registro.

El funcionamiento de derrame de registro puede ser visto en el siguiente ejemplo. Suponga que se tiene la siguiente expresión:

```
1+(2+(3+(4+(5+(6+(7+(8+(9+10))))))))
```

El código del ensamblador SPARC para la evaluación de esto en la pila puede ser como esto:

```
mov  1, %l0               ! push 1
mov  2, %l1               ! push 2
mov  3, %l2               ! push 3
mov  4, %l3               ! push 4
mov  5, %l4               ! push 5
mov  6, %l5               ! push 6
mov  7, %l6               ! push 7
mov  8, %l7               ! push 8
st   %l0, [%fp - 160]     ! spill %l0 (1)
mov  9, %l0               ! push 9
st   %l1, [%fp - 164]     ! spill %l1 (2)
mov  10, %l1              ! push 10
add  %l0,%l1,%l0          ! (9+10)
add  %l7,%l0,%l7          ! 8+(9+10)
add  %l6,%l7,%l6          ! 7+(8+(9+10))
add  %l5,%l6,%l5          ! 6+(7+(8+(9+10)))
add  %l4,%l5,%l4          ! 5+(6+(7+(8+(9+10))))
add  %l3,%l4,%l3          ! 4+(5+(6+(7+(8+(9+10)))))
add  %l2,%l3,%l2          ! 3+(4+(5+(6+(7+(8+(9+10))))))
ld   [%fp - 164], %l1     ! recover 2
add  %l1, %l2, %l1        ! 2+(3+(4+(5+(6+(7+(8+(9+10)))))))
ld   [%fp - 160], %l0     ! recover 1
```

```
add  %l0, %l1, %l0         ! 1+(2+(3+(4+(5+(6+(7+(8+(9+10))))))))
```

Su implementación deberá realizar un seguimiento del número máximo de variables derramadas durante la evaluación de expresión. Este número deberá ser adicionado la tamaño del marco de pila dado en la declaración `save`.

Estas son alguna reglas generales para la generación de código que se deberán utilizar en la evaluación de expresiones:

- **Constantes enteras**. Si el valor está en el rango de -4095 a 4095, haga lo siguiente para guardar un valor en la pila:

  ```
  mov  constant, %ln
  ```

  Si el valor tiene un valor absoluto mayor que 4095, haga lo siguiente:

  ```
  sethi %hi(.Lm), %g1
  or    %g1, %lo(.Lm), %g1
  ld    [%g1], %ln
  ...
         .section rodata

  .Lm:    .word    constant
  ```

- **Constantes de punto flotante**. Siga el mismo procedimiento para las constantes entera mayores que 4095. Sin embargo, en lugar de declarar una .word, haga esto:

  ```
  sethi %hi(.Lm), %g1
  or    %g1, %lo(.Lm), %g1
  ld    [%g1], %ln
  ...
         .section rodata

  .Lm:    .float    floatconstant
  ```

- **Variables locales y argumentos**. Generar código como esto:

  ```
  ld  [%fp + offset], %ln
  ```

  donde *offset* es el desplazamiento del puntero del marco (guardado en la tabla de símbolos) para la variable local y `%ln` es el registro que corresponda a la parte superior de la pila de expresión.

- **Operadores binarios**. Deberá aplicar un operador a los dos elementos superiores de la pila y colocar el resultado en la parte superior de la pila. SPARC sólo tiene un número limitado de operaciones binarias. De hecho, ni siquiera soporta multiplicación y división. Aquí están las secuencia de instrucciones que son necesarias para implementar aritmética entera:

  - Suma (+), Resta (-)

    ```
    add %li, %lj, %lk     ! %li + %lj -> %lk
    sub %li, %lj, %lk     ! %li - %lj -> %lk
    ```

  - Multiplicación (*), División (/)

    ```
    mov %li, %o0          ! %li * %lj -> %lk
    call .mul
    ```

```
mov %lj, %o1          ! Nota: esto es en la ranura de rama de retardo
mov %o0, %lk

mov %li, %o0          ! %li / %lj -> %lk
call .div
mov %lj, %o1          ! Nota: esto es en la ranura de rama de retardo
mov %o0, %lk
```

- **Expresiones Relacionales**. Dependiendo de su implementación, puede que le resulte más fácil la implementación de operadores de relación como operadores enteros con un 0 representando false y 1 representando true. Las siguientes instrucciones pueden ser usadas para evaluar varios operadores de relación:

  - and, or bit a bit

```
and %li, %lj, %lk     ! %li and %lj -> %lk
or %li, %lj, %lk      ! %li or  %lj -> %lk
```

  - <, <=, > >=, ==, !=

```
! <
   cmp %li, %lj       ! if %li < %lj then %lk = 1 else %lk = 0
   bl .Lm             ! rama menor que
   mov 1, %lk         ! rama retardo (siempre se ejecuta)
   mov 0, %lk
.Lm:

! <=
   cmp %li, %lj       ! if %li <= %lj then %lk = 1 else %lk = 0
   ble .Lm            ! rama menor que o igual
   mov 1, %lk         ! rama retardo (siempre se ejecuta)
   mov 0, %lk
.Lm:

! >
   cmp %li, %lj       ! if %li > %lj then %lk = 1 else %lk = 0
   bg .Lm             ! rama mayor que
   mov 1, %lk         ! rama retardo (siempre se ejecuta)
   mov 0, %lk
.Lm:

! >=
   cmp %li, %lj       ! if %li >= %lj then %lk = 1 else %lk = 0
   bge .Lm            ! rama mayor que o igual
   mov 1, %lk         ! rama retardo (siempre se ejecuta)
   mov 0, %lk
.Lm:

! ==
   cmp %li, %lj       ! if %li == %lj then %lk = 1 else %lk = 0
   be .Lm             ! rama igual
   mov 1, %lk         ! rama retardo (siempre se ejecuta)
   mov 0, %lk
.Lm:

! !=
   cmp %li, %lj       ! if %li != %lj then %lk = 1 else %lk = 0
   bne .Lm            ! rama no igual
   mov 1, %lk         ! rama retardo (siempre se ejecuta)
```

```
        mov 0, %lk
    .Lm:
```

- **Operadores Unarios**. Simplemente aplique el operador a la parte superior de la pila.
  - (-) unario

```
neg %li, %lj    ! -%li -> %lk
```

  - No lógico (asumiento 1=true, 0=false)

```
xor %li, 1, %lj    ! not %li -> %lk
```

Parte 11 - Implementación de la asignación simple

Implemente asignación simple de variables para enteros. Específicamente, la declaración

```
a := expr
```

Se traduce en un código como esto,

```
! evalue expresión
    ...
! asignación
    st %lm, [%fp + offset]
```

donde `%lm` es el registro conteniendo el resultado de la expresión y *`offset`* es el desplazamiento del marco de la variable.

Parte 12 - Implementación del write

Implemente la declaración write(expr) para probar el evaluador de expresiones. Esta función requiere el uso de la biblioteca de MiniPascal. Específicamente, el código se verá así,

```
! evalue expresión
...
! write
          mov  %lm, %o0
          call flwritei
          nop
```

donde `%lm` es el registro conteniendo el resultado de la exresión dada a write.

Pruebe su declaración write() y funciones de asignación mediante la compilación de programa tan simple como el siguiente:

```
fun main()
   a : int;
   b : int;
   c : int;
   begin
         a := 3;
         b := a + 10;
         c := b * 2;
         write(c)
```

```
    end
```

Parte 13 - Control de flujo

Implemente las declaraciones básicas de control de flujo while e if-else. La forma general de una declaración while es como sigue:

```
! declaración while

.Lm:
! Evalue relop
      ...

      cmp  %li, %g0           ! %li contiene resultado de relop
      be   .Ln                ! == 0, false, exit
      nop

      ...
      statements
      ...
      ba  .Lm
      nop

.Ln:
```

La forma general de la declaración if-else es como sigue:

```
! declaración if-else

! Evalue relop
      ...
      cmp  %li, %g0           ! %li contiene resultado de relop
      be   .Lm                ! == 0, false, goto else
      nop

! parte then
      ...
      statements
      ...
      ba .Ln                  ! parte saltar else
      nop

! parte else
.Lm:
      ...
      statements
      ...

.Ln:
```

Parte 14 - Arreglos

Cuando los arreglos son localizados en la pila, el desplazamiento del marco se refiere a la localización del primer elemento del arreglo. Por ejemplo, un arreglo `int[100]` puede ocupar la región de memoria [%fp - 600, %fp - 200].

Para accesar un elemento del arreglo, se necesitarán tres cosas:

- Calcular el valor del índice del arreglo.
- Calcular la dirección del arreglo.
- Cargar el valor del arreglo.

Por ejemplo, para evaluar `a[expr]`, se deberá generar un código similar a este (nota: hay muchas formas de hacer esto):

```
! Evalue expr
        ...
! calcule offset
        sll  %lm, 2, %lm          ! Multiplique por 4 (desplazar a la izquierda 2 bits)
        add  %fp, %lm, %lm        ! Adicionar al puntero del marco
        ld   [%lm + offset], %ln
```

Además, se debería colocar chequeo de límites del arreglo en este código.

El almacenamiento de un arreglo es similar. Se necesitará primero calcular el índice del arreglo y la dirección de memoria antes de que un valor sea guardado.

Nota: Cuando se esté evaluando expresiones como `a + b[4*i-2] + c`, los cálculos del índice del arreglo se usará en la misma expresión de pila como la expresión externa. Sólo asegúrese que el localizador de pila permita evaluación de expresiones anidadas y todo debería funcionar.

Cuando los arreglos son pasados como parámetros a funciones, ellos deberán ser pasados por referencia (por ejemplo, un puntero al primer elemento del arreglo es pasado en vez de una copia del mismo arreglo).

Parte 15 - Hacer llamados a función

Para realizar un llamado a función, se necesitará hacer lo siguiente:

- Evalue la expresión para cada argumento de la función y guarde los resultados en la pila de expresión.
- Extraiga los argumentos de la pila de expresión y guardelos en los registros `%o`*n* apropiados (o pila).
- Haga el llamado a la subrutina usando la instrucción `call`.

Cuando construya los argumentos de la función, se necesitará guardar sus valores dentro de la pila de expresión hasta que se esté listo para hacer el llamado completo a la función. Así, si se tuviera la llamada:

```
foo(a1,a2,a3,a4)
```

El código que se generará aproximadamente corresponde a este prototipo:

```
evalue a1       pila de expresión: a1
evalue a2       pila de expresión: a2 a1
evalue a3       pila de expresión: a3 a2 a1
evalue a4       pila de expresión: a4 a3 a2 a1

! Haga llamado a función actual
t = pop()
```

```
store t, arg4
t = pop()
store t, arg3
t = pop()
store t, arg2
t = pop()
store t, arg1

call foo
```

Parte 16 - Punto flotante

Referirse al Proyecto 5.

Parte 17 - Funciones anidadas

Para implementar funciones anidadas se deberá usar la técnica de "displays" como se describe en el capítulo 7 del libro. En pocas palabras:

- Para cada identificador, almacene un nivel de anidamiento en la tabla de símbolos. Esto es un entero que define la cantidad de anidamiento (0 para la mayoría de funciones externas).
- Cree un arreglo global en el segmento .data para el display. Por ejemplo:

```
        .section ".data"

DISPLAY:   .word    0, 0, 0, 0, 0, 0, 0, 0, 0, 0
                  ! número de niveles de anidamiento aquí
```

- Dada una función con nivel de anidamiento i, generar código cuando los siguientes pasos son ejecutados:
  - El valor anterior de DISPLAY[i] es guardado en una variable temporal (en la pila)
  - El valor actual de %fp es escrito en DISPLAY[i].
  - El valor anterior de DISPLAY[i] es restaurado antes de la salida de la función.
- Para buscar una variable con nivel de anidamiento i, simplemente use el valor de DISPLAY[i] para obtener el puntero del marco correspondiente. Luego referencie a la variable usando el desplazamiento de marco guardado en la tabla de símbolos.

Sugerencias para la Implementación

- No es necesario ser un experto en lenguaje ensamblador para terminar esta parte del proyecto. Lo más que necesitará hacer es imprimir pedazos de fragmento de instrucciones en código ensamblador que son unidos para hacer un programa.
- Cuando el código de generación de funciones, puede ser difícil conocer el tamaño exacto del marco de pila hasta que se haya generado todo el código. Una manera de lidiar con esto es imprimir todo el código del cuerpo de la función a una cadena y adicionar la localización del marco de pila al final. Por ejemplo, se podría hacer esto:

```
import StringIO
f = StringIO.StringIO()       # Crea un objeto de archivo en-memoria
```

```
emit_function_body(f,node)

print >>outf,"! function (start)"
print >>outf,"%s:" % function_name
print >>outf,"     save %%sp, -%d, %%sp" % frame_size
print >>outf, f.getvalue()
```

Por supuesto, hay muchas formas de hacer esto.

- Para hacer frente a la asignación de registros, me resultó más fácil escribir una clase Register con dos métodos push() y pop(). Dentro de la clase, mantengo el estado relacionado con el puntero de pila de la expresión actual, cuales registros está en uso, cuales registros han sido derramados en memoria y así sucesivamente. Nota: la pila de expresión no tiene nada que ver con el llamado de pila.

Procedimiento de entrega

1. Su compilador deberá ejecutarse desde un archivo 'hoc.py'. Debemos ser capaces de correr este archivo como sigue:

   ```
   % python hoc.py testname.pas
   % a.out
   ```

2. Asegurarse de tener un archivo README que incluya sus nombres y cualquier cosa notable acerca de su implementación.

3. Cree un archivo tar de Unix para su proyecto. Este archivo tar deberá contener el directorio 'hoc' y debe de estar basado en el nombre de su usuario de correo. Por ejemplo:

   ```
   % tar -cf sulogin.tar hoc
   ```

4. Envien este archivo al correo a3udeloz@utp.edu.co. Colocar en asunto: Compiladores, proyecto 4 sulogin

5. Tareas de última hora no son aceptadas. No se recibirá ningún trabajo después de la fecha límite.

SI NO SE PUEDE DESEMPAQUETAR O CORRER SU IMPLEMENTACION USANDO SCRIPTS DE PRUEBAS ADICIONALES, NO RECIBIRAN UNA NOTA ADECUADA!