Compiladores e Intérpretes
Generación de Código Máquina e Intérpretes

Sebastian Gottifredi

Universidad Nacional del Sur

Departamento de Ciencias e Ingeniería de la Computación

2018

Repaso

Repaso

- Para entender y controlar la **estructura** de un programa fuente hay que analizar si sigue las reglas de **sintaxis** del lenguaje
- Estas reglas están expresadas en términos de **tokens**, mientras que el fuente es una cadena de caracteres
- El **Analizador Léxico** es el encargado armar los **tokens**
- Para expresar las reglas de sintaxis del lenguaje utilizamos **gramáticas libres de contexto**

Repaso

- El **analizador sintáctico** es el encargado de **reconocer** si un programa sigue esas expresadas por la gramática, para eso:
  - La **gramática** tiene que ser **no ambigua**
  - **Simula** el proceso de **derivación** usando una estrategia:
    - **Descendentes:** arrancando del no terminal Inicial reconstruir la derivación a izquierda hasta llegar a la cadena
    - **Ascendentes:** aplican las producciones en orden inverso hasta llegar al símbolo inicial de la gramática

Repaso

- El análisis semántico es el encargado validar y entender el **significado** del programa
- Para esto el analizador semántico debe:
  - Recolectar, entender y controlar todas las entidades declaradas

    Chequeo de Declaraciones
  - Recolectar, entender y controlar todas las sentencias asociadas a las entidades recolectadas

    Chequeo de Sentencias

Repaso

- Para las tareas del análisis semántico y optimización suele ser necesaria una **representación alternativa del fuente** estructuralmente adecuada para trabajar con toda la **información que recolecto** el compilador
- Estas formas son conocidas como

Representaciones Intermedias (RI)

  - Son **representaciones alternativas** del fuente que aun **no son código maquina**

Repaso

- En un compilador es posible **Optimizaciones** que afectan la performance en ejecución del programa resultante
  - Optimización sobre **código maquina** o de bajo nivel
  - Optimización sobre **código intermedio** o de alto nivel
    - **Nivel Local:** estudian como mejorar bloques de sentencias sin cambios de flujo de control interno (bloques básicos)
    - **Nivel Globales:** estudian como mejorar el cuerpo completo de una unidad de manera aislada
    - **Nivel Interproceso:** estudia como mejorar el código considerando la interacción entre las unidades del programa

Generación de Código Maquina

Generación de Código Maquina

- Es la fase encargada de traducir el fuente (usualmente una RI del fuente) a **código ensamblador** de la **maquina destino**.
- Para hacer la traducción tener en claro de **donde partimos (RI)** y las **características de la maquina destino**
  - **Memoria y Registros**
    - Mapear entidades y etiquetas a lugares en la arquitectura
  - **Instrucciones**
    - Como utilizan sus operandos, acceso a registros, funcionamiento de Jumps
  - **Performance**

Generación de Código Maquina

- Como ejemplo vamos a estudiar como **traducir de RI basada en pila** (CeIASM) **a código SPIM** (simulador MISP)
- SPIM **no** es una maquia pila, sino que **trabaja con registros rápidos** (como las **arquitecturas RISC**)
- Vamos a tener **traducir el comportamiento tipo "pila"** de las instrucciones CeIASM en instrucciones que usan registros rápidos

Generación de Código Maquina

- **SPIM – MIPS**
  - **Arquitectura RISC** (reduced instruction set)
  - Operaciones aritméticas **solo sobre registros**
    - Para usar/guardar valores de memoria hay que cargarlos/almacenarlos en registros!!!
  - Tiene **32 registros** de propósito general
    - Para los ejemplos vamos a usar los registros rápidos especiales $sp y $fp para indicar el tope de la pila y el registro de activación actual
    - $t1, $t2,… para guardar valores temporales a la hora de computar expresiones

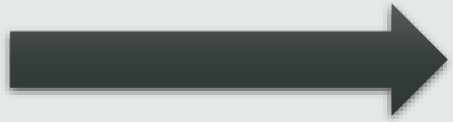
Generación de Código Maquina

- Algunas Instrucciones SPIM, si M es la memoria
  - lw $R1, offset($R2)
    - Carga Word: Guardo en el registro $R1 el valor de M[$R2+offset]
  - add $R1, $R2, $R3
    - Suma: Guardo en $R1 el resultado de sumar los contenidos de $R2 y $R3
  - addi $R1, $R2, x
    - Suma Entero: Guardo $R1 el resultado de sumar el contenido $R2 con x
  - sw $R1, offset($R2)
    - Guarda Word: Guardo en M[$R2+offset] el valor de $R1
  - li $R1, x
    - Carga Integer: Guardo x en $R1

Generación de Código Maquina

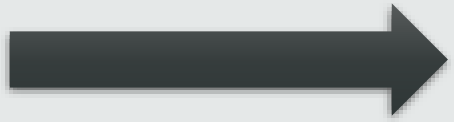
- En general

`PUSH x`



```
addi $sp $sp -4
li   $t1 x
sw   $t1 0($sp)
```

`ADD`

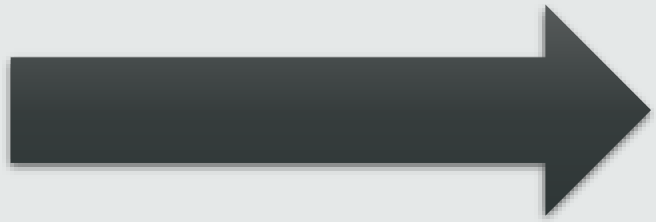


```
lw   $t1, 0($sp)
lw   $t2, 4($sp)
add  $t1, $t2, $t1
sw   $t1, 4($sp)
addi $sp, $sp, 4
```

Generación de Código Maquina

- La memoria en SPIM no es una pila entonces tenemos que simular su comportamiento…
- 12+3 en CelASM

```
PUSH 12
PUSH 3
ADD
```



```
addi $sp $sp -4
li   $t1 12
sw   $t1 0($sp)
```

PUSH 12

```
addi $sp $sp -4
li   $t1 3
sw   $t1 0($sp)
```

PUSH 3

```
lw   $t1, 0($sp)
lw   $t2, 4($sp)
add  $t1, $t2, $t1
sw   $t1, 4($sp)
addi $sp, $sp, 4
```

ADD

Generación de Código Maquina

- Las **variables locales** y los **parámetros** tienen su lugar en los **registros de activación (RA)**
- Como los **RA** operan sobre la **pila** también **tenemos que simularlos!**
  - La clave para usar los RA es el **frame pointer (fp)** que indica el **registro de activación actual**
- Para esto nos vamos a valer del **registro rápido $fp** que lo usamos para simular el comportamiento del puntero fp

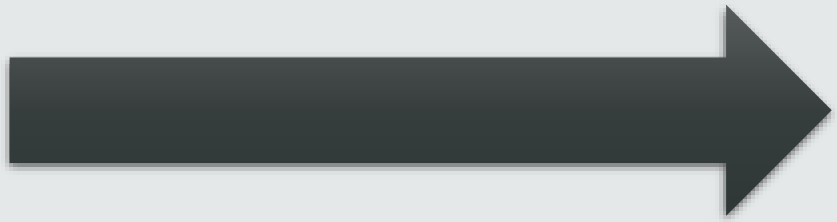
Generación de Código Maquina

- Los **offsets** de las **variables** locales/parámetros del código **CeIASM** generado es **relativo al fp**
  - Podemos **reusar los offset** de las instrucciones CeIASM!
  - En SPIM los **offsets** de esas entidades **será relativo al registo $fp**
  - Aun así, tenemos que tener en cuenta el **tamaño de las locaciones** de memoria…

Generación de Código Maquina

- Por ejemplo si la variable local v tiene offset 5 y tenemos que
- v = 12 + 3 en CeIASM

```
PUSH 12
PUSH 3
ADD
STORE 5
```



```
addi   $sp $sp -4
li     $t1 12
sw     $t1 0($sp)
```
PUSH 12

```
addi   $sp $sp -4
li     $t1 3
sw     $t1 0($sp)
```
PUSH 3

```
lw     $t1, 0($sp)
lw     $t2, 4($sp)
add    $t1, $t2, $t1
sw     $t1, 4($sp)
addi   $sp, $sp, 4
```
ADD

```
lw     $t1, 0($sp)
sw     $t1, 20($fp)
addi   $sp, $sp, 4
```
STORE 5

Generación de Código Maquina

- Por ejemplo si la varia ne offset 5 y tenemo
- v = 12 + 3 en Cel

```
PUSH 12
PUSH 3
ADD
STORE 5
```

¿Se puede optimizar?

Si somos consistentes en como usamos los registros rápidos, fíjense que en t1 siempre queda el valor del tope de la pila

Además podemos ver si la instrucción inmediata siguiente consume o no lo recientemente apilado

```
addi   $sp $sp -4
li     $t1 12
sw     $t1 0($sp)
```
PUSH 12

```
addi   $sp $sp -4
li     $t1 3
sw     $t1 0($sp)
```
PUSH 3

```
lw     $t1, 0($sp)
lw     $t2, 4($sp)
add    $t1, $t2, $t1
sw     $t1, 4($sp)
addi   $sp, $sp, 4
```
ADD

```
lw     $t1, 0($sp)
sw     $t1, 20($fp)
addi   $sp, $sp, 4
```
STORE 5

Generación de Código Maquina

- Por ejemplo si la varia ne offset 5 y tenemos que
- v = 12 + 3 en CelASM

```
addi   $sp $sp -4
li     $t1 12
sw     $t1 0($sp)
li     $t1 3
lw     $t2, 0($sp)
add    $t1, $t2, $t1
addi   $sp, $sp, 4
sw     $t1, 20($fp)
```

PUSH 12

PUSH 3

ADD

STORE 5

¿Se puede optimizar?

```
addi   $sp $sp -4
li     $t1 12
sw     $t1 0($sp)
addi   $sp $sp -4
li     $t1 3
sw     $t1 0($sp)
lw     $t1, 0($sp)
lw     $t2, 4($sp)
add    $t1, $t2, $t1
sw     $t1, 4($sp)
addi   $sp, $sp, 4
lw     $t1, 0($sp)
sw     $t1, 20($fp)
addi   $sp, $sp, 4
```

PUSH 12

PUSH 3

ADD

STORE 5

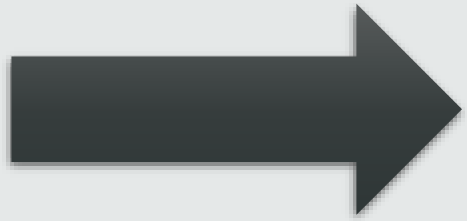
Generación de Código Maquina

- Otras Instrucciónes SPIM, si M es la memoria
  - beqz $R1, Label
    - Salto condicional: Salta a Label si $R1 == 0
  - j Label
    - Salto incondicional: Salta a Label

Generación de Código Maquina

- Por ejemplo, si v tiene offset 5 y w offset 8
- if(w) v = 15 else v = 30

```
        LOAD 8
        BF e1
        PUSH 15
        STORE 5
        JUMP e2
e1:     PUSH 30
        STORE 5
e2:     …
```



```
        addi   $sp, $sp, -4
        lw     $t1, 32($fp)
        sw     $t1, 0($sp)
        lw     $t1, 0($sp)
        addi   $sp, $sp, 4
        beqz   $t1, e1
        addi   $sp $sp -4
        li     $t1 15
        sw     $t1 0($sp)
        lw     $t1, 0($sp)
        sw     $t1, 20($fp)
        addi   $sp, $sp, 4
        j      e2
e1:     addi   $sp $sp -4
        li     $t1 30
        sw     $t1 0($sp)
        lw     $t1, 0($sp)
        sw     $t1, 20($fp)
        addi   $sp, $sp, 4
e2:     …
```

LOAD 8

BF e1

PUSH 15

STORE 5

JUMP e2

PUSH 30

STORE 5

Generación de Código Maquina

- Por ejemplo, si v tiene
  y w offset 8
- if(w) v = 15 else v

```
        LOAD 8
        BF e1
        PUSH 15
        STORE 5
        JUMP e2
e1:     PUSH 30
        STORE 5
e2:     …
```

¿Se puede optimizar?

Si somos consistentes en como usamos los registros rápidos, fíjense que en t1 siempre queda el valor del tope de la pila

Además podemos ver si la instrucción inmediata siguiente consume o no lo recientemente apilado

```
        addi   $sp, $sp, -4
        lw     $t1, 32($fp)
        sw     $t1, 0($sp)
        lw     $t1, 0($sp)
        addi   $sp, $sp, 4
        beqz   $t1, e1
        addi   $sp $sp -4
        li     $t1 15
        sw     $t1 0($sp)
        lw     $t1, 0($sp)
        sw     $t1, 20($fp)
        addi   $sp, $sp, 4
        j      e2
e1:     addi   $sp $sp -4
        li     $t1 30
        sw     $t1 0($sp)
        lw     $t1, 0($sp)
        sw     $t1, 20($fp)
        addi   $sp, $sp, 4
e2:     …
```

LOAD 8

BF e1

PUSH 15

STORE 5

JUMP e2

PUSH 30

STORE 5

Generación de Código Maquina

- Por ejemplo, si v tiene y w offset 8
- if(w) v = 15 else v = 30

```
        lw      $t1, 32($fp)
        beqz    $t1, e1
        li      $t1, 15
        sw      $t1, 20($fp)
        j       e2
e1:     li      $t1, 30
        sw      $t1, 20($fp)
e2:     …
```

¿Se puede optimizar?

```
        addi    $sp, $sp, -4
        lw      $t1, 32($fp)
        sw      $t1, 0($sp)
        lw      $t1, 0($sp)
        addi    $sp, $sp, 4
        beqz    $t1, e1
        addi    $sp $sp -4
        li      $t1 15
        sw      $t1 0($sp)
        lw      $t1, 0($sp)
        sw      $t1, 20($fp)
        addi    $sp, $sp, 4
        j       e2
e1:     addi    $sp $sp -4
        li      $t1 30
        sw      $t1 0($sp)
        lw      $t1, 0($sp)
        sw      $t1, 20($fp)
        addi    $sp, $sp, 4
e2:     …
```

LOAD 8

BF e1

PUSH 15

STORE 5

JUMP e2

PUSH 30

STORE 5

Interpretes Conceptos Generales

Intérpretes

- Es un programa que toma las instrucciones de un programa fuente y las ejecuta directamente sobre una maquina destino

```
Intérprete(Programa Fuente P):
    Loop
        Tomar siguiente instrucción i de P
        Si i es no valida
                Reportar ERROR!
        Sino
                traducir i al código maquina m
                ejecutar m
                Si se produce error al ejecutar m
                        Reportar ERROR!
                Sino si m tiene “Halt”
                        Finalizar el Loop
```

Intérpretes

- Básicamente hay **3 tipos** de interpretes:
  - Los que a partir del **programa fuente**, van tomando de a una sentencia, la **analizan, la traducen y la ejecutan** (**puros**)
    - Lisp, Prolog, etc.
  - Los que **primero traducen** el fuente a una **RI** conveniente (generalmente de alto nivel) y despues van ejecutando a partir de esa representación
    - Python, Ruby, Perl, etc.
  - Los que toman **archivos resultantes** de un proceso de **precompilacion**
    - Java, C#, etc.

Intérpretes Puros

- En los intérpretes puro

```
Intérprete(Programa Fuente P):
    Loop
        Tomar siguiente instrucción i de P
        Si i es no valida
                Reportar ERROR!
        Sino
                traducir i al código maquina m
                ejecutar m
                Si se produce error al ejecutar m
                        Reportar ERROR!
                Sino si m tiene “Halt”
                        Finalizar el Loop
```

¿Cuál es la “siguiente” instrucción?

La siguiente según el flujo de ejecución

¿Qué implicancia tiene esto?

El interprete tiene que estar consiente del runtime: los valores que toman variables y expresiones

Intérpretes Puros

- En los intérpretes puros

```
Intérprete(Programa Fuente P):
    Loop
        Tomar siguiente instrucción i de P
        Si i es no valida
                Reportar ERROR!
        Sino
                traducir i al código maquina m
                ejecutar m
                Si se produce error al ejecutar m
                        Reportar ERROR!
                Sino si m tiene “Halt”
                        Finalizar el Loop
```

Chequea Léxica, Sintáctica y Semánticamente la instrucción. Actualiza la tabla de símbolos con toda la información de la instrucción

Intérpretes Puros

- En los intérpretes puros

```
Intérprete(Programa Fuente P):
    Loop
        Tomar siguiente instrucción i de P
        Si i es no valida
                Reportar ERROR!
        Sino
                traducir i al código maquina m
                ejecutar m
                Si se produce error al ejecutar m
                        Reportar ERROR!
                Sino si m tiene “Halt”
                        Finalizar el Loop
```

Usa la información de la Tabla de Símbolos para hacer la traducción

Intérpretes Puros

- En los intérpretes puros

```
Intérprete(Programa Fuente P):
    Loop
        Tomar siguiente instrucción i de
        Si i es no valida
                Reportar ERROR!
        Sino
                traducir i al código maquina m
                ejecutar m
                Si se produce error al ejecutar m
                        Reportar ERROR!
                Sino si m tiene “Halt”
                        Finalizar el Loop
```

Si el interprete no hace ninguna distinción sobre las instrucciones que traduce ¿Qué implicancia tiene esto?

Que podemos traducir mas de una vez la misma instrucción!!!

Intérpretes Puros

- En los intérpretes puros
  - Los lenguajes **puramente interpretados no** requieren que se **declaren tipos**
  - Las **instrucciones** deben ser **totalmente independientes** por que sino no se puede traducir y ejecutar
  - **Ineficiente**, por que la **validación y la traducción** se de cada instrucción se hace **cada vez** que es seleccionada

Intérpretes con traducción a RI

- Los Interpretes que primero traducen el fuente a una RI conveniente y ejecutan a partir de esa representación

```
Intérprete(Programa Fuente P):
    Si P es no valido
        Reportar ERROR!
    Traducir P en RI
    Loop
        Tomar siguiente instrucción i de RI
        Traducir i al código maquina m
        ejecutar m
        Si se produce error al ejecutar m
                Reportar ERROR!
        Sino si m tiene “Halt”
                Finalizar el Loop
```

Intérpretes con traducción a RI

- Los Interpretes que primero trad
  conveniente y ejecutan a partir de

```
Intérprete(Programa Fuente P)
    Si P es no valido
        Reportar ERROR!
    Traducir P en RI
    Loop
        Tomar siguiente instrucción i de RI
        Traducir i al código maquina m
        ejecutar m
        Si se produce error al ejecutar m
                Reportar ERROR!
        Sino si m tiene “Halt”
                Finalizar el Loop
```

Realizan todos los controles sintácticos y la mayoría de los semánticos.

A diferencia de los intérpretes puros, se chequea sintáctica y semánticamente **todo** P

¿A que otra diferencia lleva esto respecto a los intérpretes puros?

Cada instrucción de P se chequea una sola vez!

Intérpretes con traducción a RI

- Los Interpretes que primero traducen conveniente y ejecutan a partir de esa

```
Intérprete(Programa Fuente P):
    Si P es no valido
        Reportar ERROR!
    Traducir P en RI
    Loop
        Tomar siguiente instrucción i de RI
        Traducir i al código maquina m
        ejecutar m
        Si se produce error al ejecutar m
              Reportar ERROR!
        Sino si m tiene “Halt”
              Finalizar el Loop
```

Al igual que en los Intérpretes puros se usa la información de la Tabla de Símbolos para hacer la traducción

¿Hay alguna diferencia con los puros?

Que se cuenta con la tabla de símbolos completa, ya que P fue completamente analizado

Intérpretes con traducción a RI

- Los Interpretes que primero traducen [illegible] RI conveniente y ejecutan a partir de es[illegible]

```
Intérprete(Programa Fuente P):
    Si P es no valido
        Reportar ERROR!
    Traducir P en RI
    Loop
        Tomar siguiente instrucción i de RI
        Traducir i al código maquina m
        ejecutar m
        Si se produce error al ejecutar m
                Reportar ERROR!
        Sino si m tiene “Halt”
                Finalizar el Loop
```

Al igual que en los intérpretes puros…

tiene que ser consiente del runtime para elegir la próxima instrucción de forma adecuada

Si no hace distinción entre las instrucciones de RI las puede traducir mas de una vez

Intérpretes con traducción a RI

- Los Interpretes que primero traducen el fuente a una RI conveniente y ejecutan a partir de esa representación
  - Tiene un **overhead inicial mayor** que uno puro
  - En **ejecución es mas rápida** que uno puro
  - Tiene **mejor contexto y permite** estructuras **mas complejas** en el PF respecto a uno puro
  - Pueden **reportar errores antes** de ejecutar

Intérpretes de código Precompilado

- Los que toman archivos resultantes de un proceso de precompilación

```
Intérprete(Programa Precompilado BC):
    Loop
        Tomar siguiente instrucción i de BC
        traducir i al código maquina m
        ejecutar m
        Si se produce error al ejecutar m
                Reportar ERROR!
        Sino si m tiene “Halt”
                Finalizar el Loop
```

Intérpretes de código Precompilado

- Los que toman archivos resultantes d
  precompilación

```
Intérprete(Programa Precompilado BC):
    Loop
        Tomar siguiente instrucción i de BC
        traducir i al código maquina m
        ejecutar m
        Si se produce error al ejecutar m
                Reportar ERROR!
        Sino si m tiene “Halt”
                Finalizar el Loop
```

No debería ser necesario controlar la correctitud de BC -- ya fue controlado el fuente en proceso de precompilacion

O debería ser un control muy sencillo

Intérpretes de código Precompilado

- Los que toman archivos resultantes d
  precompilación

```
Intérprete(Programa Precompilado BC):
    Loop
        Tomar siguiente instrucción i de BC
        traducir i al código maquina m
        ejecutar m
        Si se produce error al ejecutar m
                Reportar ERROR!
        Sino si m tiene “Halt”
                Finalizar el Loop
```

¿Qué cosa era importante para poder hacer la traducción de manera adecuada en los otros tipos de intepretes?

La tabla de símbolos!

Dado que el precompilador no sobrevive…
¿Cómo hacemos?

Se crea una nueva pero solo para BC (es decir sin la información simbólica del fuente)

Intérpretes de código Precompilado

- Los que toman archivos resultantes de un proceso de precompilación
  - Son **mas eficientes** que los que transforman al fuente en una RI
  - **No cuentan con la TS** del fuente, por lo tanto **no pueden asociar** fácilmente **errores** de ejecución a **instrucciones del fuente**
  - Son **modulares**, pueden usarse para **ejecutar códigos de distintos lenguajes fuente** (mientras sean precompilados a la misma RI)
    - Por ejemplo la **maquina virtual de java** corre programas precompilados de Java, Kotlin, Scala, Groovy, Clojure

Intérpretes – Aplicaciones

- Aplicaciones Generales de un Intérprete
  - Lenguajes de comandos (Shell) o “glue”
  - Código automodificable
  - Virtualización
  - Sandboxing
  - Emulación

Intérpretes vs Compiladores

- Ventajas de los Intérpretes vs Compiladores
  - Usa menos memoria
  - Es mas fácil trabajar con tipado dinámico
  - Permiten mas interacción con el usuario
  - Son concientes del Runtime y permiten reportar mejor los errores en ejecución
  - Portabilidad
- Desventajas de los Intérpretes vs Compiladores
  - Eficiencia en tiempo de Ejecución
  - Capacidad de Optimización