پروژه‌ی درس کامپایلر

دانشگاه صنعتی اصفهان

استاد درس: زینب زالی

تابستان ۹۹

فهرست مطالب

مقدمه:

در این بخش، به فاز سوم و نهایی از پروژه می‌پردازیم؛ لازم به یادآوری است که در فازهای قبلی پروژه، ما توانستیم که برنامه‌ی ورودی را از لحاظ ایرادهای لغوی و تا حدودی از لحاظ ایرادهای معنایی بررسی کنیم و درخت معنایی آن را رسم کنیم اما همان طور که می‌دانید وظیفه‌ی یک کامپایلر، صرفا تولید درخت معنایی نیست بلکه خروجی یک کامپایلر باید یک برنامه به زبان اسمبلی باشد[1] و ما هم در این فاز قصد داریم که به این هدف برسیم.

برای تولید کد اسمبلی، کافی است که **semantic action** های فاز قبل را به گونه‌ای تغییر دهید که منجر به تولید کد اسمبلی شود بنابراین نیاز نیست که شما همه چیز را از صفر بنویسید بلکه همان برنامه‌ای را که برای فاز اول و دوم پروژه نوشتید را در این جا تکمیل کنید.

زبان اسمبلی‌ای که برای کد خروجی در نظر گرفته‌ایم، زبان **MIPS** است[2]. البته قصد نداریم که از دستورات پیچیده‌ی آن استفاده کنید و صرفا یک آشنایی مقدماتی با آن می‌تواند برای انجام این فاز کافی باشد.

تمام قوانینی که در فازهای قبلی گفته شد در این جا نیز پا برجا است اما به منظور راحتی شما و ساده‌تر شدن پروژه، محدودیت‌های جدیدی را در زبان ورودی اعمال می‌کنیم که در ادامه به آن‌ها خواهیم پرداخت.

هدف:

شما باید با استفاده از flex و yacc یک برنامه بنویسید و آن را در سامانه آپلود کنید.

برنامه‌ی شما مشابه یک کامپایلر واقعی عمل می‌کند؛ به این صورت که پس از دریافت فایل ورودی، یک فایل به زبان MIPS تولید کند به گونه‌ای که قابلیت اجرا در محیط SPIM[3] را داشته باشد.

اگر برای اجرا در فضای SPIM، به تعریف برخی از ماکروها نیاز باشد، بهتر است که کامپایلر بتواند خودش آنها را تولید کند اما در صورتی که به هیچ عنوان امکان اعمال آن‌ها وجود نداشت با تی‌ای هماهنگ کنید.

از آنجا که ما در این جا انتظار یک کامپایلر را داریم، باید در صورت وجود هرگونه خطا در فایل ورودی، بدون تولید فایل خروجی و با نمایش خطای مربوطه خارج شود.

---

[1] در این جا اسمبلر را جزوی از فرآیند کامپایل در نظر نمی گیریم

[2] فرض بر این است که شما این زبان را در درس معماری کامپیوتر آموخته‌اید. چنانچه این گونه نیست به استاد اطلاع دهید.

[3] یک برنامه‌ی شبیه سازی برای اجرای کدهای MIPS است.

تغییرات زبان ورودی:

قواعد زیر به زبان ورودی اضافه می‌شوند که هر برنامه‌ی نوشته شده‌ای به این زبان باید آن‌ها را رعایت کند:

۱. در این فاز، همه‌ی متغیرها از نوع int هستند و از هیچ تایپ دیگری استفاده نمی‌شود، از طرفی تضمین می‌شود که کلمات string و char و ... به عنوان id نیز در فایل‌های ورودی وجود ندارد. بنابراین شما نیاز نیست که قسمت‌های مربوطه در کد خود را تغییر دهید بلکه کافی است semantic action آن را خالی بگذارید.
۲. خروجی تابع تنها از نوع int و void می‌تواند باشد.
۳. خروجی توابع، آرایه نخواهد بود.

نکات پروژه:

هنگام نوشتن پروژه، نکات زیر را در نظر داشته باشید:

۱. باید هر تابع تنها به متغیرهای گلوبال و متغیرهای محلی مجاز خود دسترسی داشته باشد(بدیهی است که شرط if و حلقه‌ی foreach، همانند زبان C، دسترسی‌های خاص خود را دارند)
۲. نگران پرش‌های با طول زیاد نباشید، فرض می‌شود که آدرس همه‌ی توابع و jumpها با هر دستور پرشی قابل دسترسی هستند.
۳. برای نگهداری متغیرها و آرایه‌ها می‌توانید از حافظه‌ی استک استفاده کنید و اجباری در استفاده از حافظه‌ی heap نیست.
۴. متغیرهای گلوبال و محلی را کنترل کنید
۵. تقسیم بر صفر را چه به صورت صریح و چه به صورت غیر صریح کنترل کنید و در صورت وجود تقسیم بر صفر، یک هشدار چاپ کنید(دقت کنید که خطایی چاپ نمی‌شود! یعنی شما کد اسمبلی را تولید می‌کنید و تنها یک هشدار به کاربر نمایش می‌دهید)
۶. برای نوشتن کد MIPS کافی است صرفا از دستورات تدریس شده در درس معماری استفاده کنید.

نکته‌ی بسیار مهم: چنانچه نوشتن بخشی از کامپایلر در توانتان نبود، لطفا پروژه را رها نکنید و بقیه‌ی قسمت‌ها را انجام دهید و در یک فایل به اسم notImplemented.txt توضیح دهید که چه بخش‌هایی را نتوانستید بنویسید تا کد شما متناسب با آنچه که نوشته‌اید تصحیح شود(تست کیس های مناسب با چیزی که تحویل داده‌اید به کامپایلر شما داده خواهد شد) و تمام نمره را از دست ندهید.

سودوکدهای نمونه از نحوه‌ی کد خروجی:

در این بخش قصد داریم که چند نمونه از تبدیل کدهای ورودی به کد اسمبلی خروجی را با هم تجربه کنید اما دقت کنید که کد اسمبلی استفاده شده در این‌جا، به زبان MIPS نیست و صرفا جنبه‌ی سودوکد دارد تا شما با بعضی از مسیرهای تولید کد آشنا شوید(البته نمونه‌های آن را در کلاس هم دیده‌اید).

بدیهی است که بین کدهای ورودی و کدهای اسمبلی، هیچ تناظر یک به یکی وجود ندارد و از این رو هر کدی ممکن است چندین کد اسمبلی معادل داشته باشد در نتیجه کدهای زیر، صرفا جهت راهنمایی قرار داده شده است

۱. عملیات های unary

ابتدا عبارت مربوط به عملگر یونری را حساب می کنیم و سپس عملگر یونی را پردازش می‌کنیم:

```
Input code: -3

pseudo code :    movl    $3, %eax;      //EAX register
                 contains 3

                 neg     %eax;          //now EAX
                 register contains -3
```

۲. عملیات‌های باینری

ابتدا باید کد مربوط به e1 را تولید کنیم و مقدار آن را در استک ذخیره کنیم سپس کد مربوط به e2 را تولید می کنیم و مقدار آن را محاسبه می‌کنیم. مقدار e1 را از استک برمی‌داریم و عملیات جمع را انجام می‌دهیم.

```
Input code: e1 + e2

pseudo code:     <CODE FOR e1 GOES HERE>

                 push %eax ; save value of e1 on the
                 stack

                 <CODE FOR e2 GOES HERE>

                 pop %ecx ; pop e1 from the stack into
                 ecx
```

```
addl %ecx, %eax ; add e1 to e2, save
results in eax
```

۳. عملیات‌های باینری که می‌توان اتصال کوتاه(نیازی به اجرای کل دستور نباشد مانند || و &&) را در آن‌ها پیاده کرد.

همان مراحل قبل را طی می‌کنیم با این تفاوت که اگر پس از محاسبه‌ی e1 نتیجه‌ی محاسبات به صورت قطعی تعیین گردید پس دیگر نیازی به محاسبه‌ی e2 نیست و به سراغ کامپایل خط بعدی در برنامه می‌رویم.

```
Input code: e1 || e2

pseudo code:    <CODE FOR e1 GOES HERE>

                cmpl $0, %eax           ; check if e1
                is true

                je _clause2             ; e1 is 0, so
                we need to evaluate clause 2

                movl $1, %eax           ; we didn't
                jump, so e1 is true and therefore
                result is 1

                jmp _end
                _clause2:

                <CODE FOR e2 GOES HERE>

                cmpl $0, %eax           ; check if e2
                is true

                movl $0, %eax           ; zero out EAX
                without changing ZF

                setne %al               ; set AL
                register (the low byte of EAX) to 1 iff
                e2 != 0
```

```
_end:
```

۴. عبارت‌های شرطی

مقدار e1 را محاسبه کرده و آن را با ۰ مقایسه می‌کنیم چنانچه برابر با صفر بود آنگاه کد مربوط به قسمت else را باید اجرا گردد و در غیر این صورت، تنها کد مربوط به e2 باید انجام شود.

```
Input code:          if(e1) e2 else e3

Pseudo code:     <CODE FOR e1 GOES HERE>

                 cmpl $0, %eax

                 je   _e3                  ; if e1 == 0,
                 e1 is false so execute e3

                 <CODE FOR e2 GOES HERE>  ; we're still
                 here so e1 must be true. execute e2.

                 jmp  _post_conditional    ; jump over
                 e3

                 _e3:

                 <CODE FOR e3 GOES HERE>  ; we jumped
                 here because e1 was false. execute e3.

                 _post_conditional:            ; we need
                 this label to jump over e3
```

۵. فراخوانی تابع:

```
input code:      foo(1, 2, 3)

pseudo Code:
```

ابتدا پارامترها را در ریجسترهای مناسب و یا در استک ذخیره می‌کنیم:

```
push $3
push $2
```

```
push $1
```

تابع مورد نظر را صدا می‌زنیم:

```
call _foo
```

حذف آرگومان‌های ورودی تابع foo از استک:

```
add $0xc, %esp
```

```
_foo:
```

ذخیره‌ی آدرس شروع استک مربوط به تابع صدا زننده:

```
Push %ebp
```

مقداری دهی استک برای تابع foo :

```
Mov %esp, %ebp
```

انجام کارهای داخل تابع:

```
Do stuff
```

حذف همه‌ی متغیرهای گرفته شده از استک در طول اجرای تابع foo:

```
Mov %ebp, %esp
```

بازگرداندن اطلاعات استک مربوط به تابع صدازننده:

```
Pop %ebp
```

برگشت به تابع قبلی:

```
ret
```

چند نمونه تولید کد MIPS:

امید است که تا اینجای کار یک دید کلی از کاری که قرار است در این فاز انجام دهید و نحوه‌ی انجام آن داشته باشید. در این بخش سعی داریم که با جزییات بیشتری به فرآیند تولید کد MIPS بپردازیم.

در ادامه چند مثال آورده شده است که شامل کد ورودی به همراه کد خروجی متناظر تولید شده است(بخش اول کد زبان ورودی است و بخش کد دوم کد معادل زبان mips):

۱. تابع:

```
Void main(int x[], int a[], int andis){
    X = a;
    X[andis] = a[andis];
}
```

```
main:    //x in $a0, a in $a1, andis in $a2
Addi $sp, $sp, -4
Sw $s0, 0($sp)
Add $s0, $a2, $zero
Add $s2, $s2, $s2
Add $t0, $a0, $zero
Add $t1, $a1, $zero
Add $s2, $s2, $s2
Add $t0, $t0, $s2
Add $t1, $t1, $s2
Lw $t2, 0($t1)
Sw $t2, 0($t0)
Lw $s0, 0($s)
Addi $sp, $sp, 4
Jr $ra;
```

۲. مثال از عبارت‌های شرطی:

```
If (i < N)
    A[i] = 0;
```

```
//Assume that i in $sp, N in $sp + 4, A in $sp + 8
Lw $t0, 0($sp)
Lw $t1, 4($sp)
Lw $t2, 8($sp)
Slt $t1, $t0, $t1
Beq $t1, $zero, ENDIF
Sll $t0, $t0, 2
Add $t0, $t0, $sp
```

```
Sw $zero, 0($sp)
ENDIF:
```

۳. تعریف متغیر گلوبال:

```
Int c;

.data:
X:   .word        5
```

۴. تغییر مقدار متغیر گلوبال:

```
Int c = 3;

Lw $t0, x($gp)
Addi $t0, $t0, 3
Sw $t0, x($gp)
```

خطایابی:

در بخش "هدف" گفته شد که برنامه‌ی شما باید بتواند هر گونه خطایی را تشخیص دهد.

منظور از هر گونه خطا، تمام خطاهایی است که یک کامپایلر C آن‌ها را در نظر می‌گیرد: بنابراین کامپایلر شما دقیقا مشابه یک کامپایلر C رفتار خواهد کرد مگر در مواردی که در این فاز و یا فازهای قبلی، قانونی برای آن‌ها ذکر شده باشد.

برای مثال فرض کنید که یک متغیر به شکل int a[[5]] تعریف شده است: در فازهای قبلی ما هیچ قانونی در مورد تعریف کردن آرایه به این شکل نداشتیم. بنابراین معیار ما، رفتار کامپایلر C با این syntax خواهد بود و اگر به کامپایلر C مراجعه کنید خواهید دید که این نحوه‌ی تعریف آرایه در زبان C (و چه بسا در هر زبان دیگری) به عنوان خطا در نظر گرفته می‌شود پس کامپایلر شما هم نباید اجازه‌ی این شیوه‌ی تعریف را بدهد.

یا مثلا اگر متغیری تعریف نشده باشد، استفاده از آن غیرمجاز است و باید به عنوان خطا در نظر گرفته شود.

در صورت وجود هرگونه ابهام، می‌توانید به تی‌ای‌های درس مراجعه کنید.