پروژه‌ی درس کامپایلر

دانشگاه صنعتی اصفهان

استاد درس: مریم موزرانی

پاییز ۹۸

در این پروژه قصد داریم که یک کامپایلر برای زبان --c (که البته به منظور ساده سازی، اصلاح شده است) طراحی کنیم. پیاده سازی این کامپایلر باید با استفاده از ابزار بایسون و فلکس انجام شود.

کامپایلر هدف باید بتواند با دریافت کد ورودی که به زبان --c نوشته است را با در نظر گرفتن semantic action و دیگر مفایم لازم یک کد خروجی به زبان MIPSولید کند.

همچنین باید توانایی یافتن خطاهای احتمالی لازم همانند تقسیم بر صفر را داشته باشد.

توضیح زبان --c:

۱. برنامه حتما باید دارای یک تابع main باشد که اجرای برنامه از آن آغاز گردد.
۲. در زبان مورد نظر همه‌ی متغیرها از نوع int هستند و از هیچ تایپ دیگری استفاده نمی‌شود.
۳. خروجی تابع می‌تواند int و یا void باشد.
۴. هیچ حلقه‌ای مانند while و for در زبان مورد نظر وجود ندارد. اما پیاده سازی آنها به عنوان نمره‌ی اضافی در نظر گرفته می‌شود.
۵. شرط if ممکن است در کدها وجود داشته باشد.
۶. فراخوانی تابع نیز وجود دارد.
۷. متغیرهای از نوع int، ۳۲ بیتی هستند.
۸. تضمین می‌شود که پارامترهای ورودی تابع از ۴ عدد تجاوز نمی‌کند.

نکات پروژه:

۱. باید هر تابع تنها به متغیرهای محلی و یا متغیرهای گلوبال دسترسی داشته باشد.
۲. اگر کامپایلر با فلگ show-- اجرا شود، آنگاه همزمان با کامپایل کد، تا حد امکان کد را اجرا کند و اطلاعات نهایی را چاپ نماید: مقدار هر متغیر در انتهای اجرا چه مقداری است، هر متغیر در کدام خانه از استک و یا در کدام ریجستر قرار دارد، چند خانه از استک مورد استفاده قرار گرفته است.
۳. نگران پرش‌های با طول زیاد نباشید، فرض می‌شود که آدرس همه‌ی توابع و jumpها با هر دستور پرشی قابل دسترسی است.
۴. برای نگه‌داری متغیرها و آرایه‌ها از حافظه‌ی استک استفاده کنید و نیازی به استفاده از حافظه‌ی heap نیست.
۵. متغیرهای گلوبال و متغیرهای محلی را کنترل کنید.

۶. تقسیم بر صفر را کنترل کنید. در صورت احتمال وجود تقسیم بر صفر، یک هشدار چاپ کنید.
۷. برای کد MIPS از دستورات استفاده شده در درس معماری کامپیوتر استفاده کنید.
۸. توابع prototype ندارند و باید به صورت یکجا تعریف شوند.
۹. تمام برنامه در قالب یک فایل داده می‌شود.
۱۰. در صورت وجود خطا، تنها همان خطا را چاپ شود و کامپایلر بدون تولید هر گونه کدی خارج شود.

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.

برای نمونه یک گرامر ساده در زیر آماده است که می‌توانید از آن برای پیاده‌سازی کامپایلر خود استفاده کنید، این گرامر برخی از امکانات گفته شده را ندارد اما می‌توانید به راحتی آن را گسترش دهید:

```
PROGRAM → STMT_DECLARE PGM
PGM → TYPE ID '(' ')' '{' STMTS '}' PGM | epsilon
STMTS → STMT STMTS | epsilon
STMT → STMT_DECLARE | STMT_ASSIGN | STMT_RETURN | ';'
EXP → EXP '<' EXP
EXP → EXP '<=' EXP
EXP → EXP '>' EXP
EXP → EXP '>=' EXP
EXP → EXP '!-' EXP
EXP → EXP '==' EXP
EXP → EXP '+' EXP
EXP → EXP '-' EXP
EXP → EXP '*' EXP
```

```
EXP → EXP '/' EXP
EXP → EXP '&&' EXP
EXP → EXP '||' EXP
EXP → EXP '<' EXP
EXP → EXP '<=' EXP
EXP → EXP '|' EXP
EXP → EXP '&' EXP
EXP → EXP '^' EXP
EXP → '!' EXP
EXP → '~' EXP
EXP → '-' EXP
EXP → '(' EXP ')'
EXP → ID
EXP → NUM
STMT_DECLARE → TYPE ID IDS
IDS → ';' | ',' ID IDS
STMT_ASSIGN → ID '=' EXP ';'
STMT_RETURN → RETURN EXP ';'
TYPE → INT | VOID
```

در این پروژه برای هر عملیات باید semantic action متناسب با آن را انجام دهید به گونه‌ای که منجر به تولید کد اسمبلی صحیح شود؛ در زیر نمونه‌ای از این عملیات‌ها را با هم بررسی می‌کنیم(بدیهی است که بین کدهای اسمبلی و کدهای زبان c - - 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:

```
c-- 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.

```
c-- 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.

```
  c-- 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.

```
    C-- 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:

```
c-- 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 مثال از کامپایلر کردن کد --c به MIPS: بخش اول کد زبان --c 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:
```

قسمت‌های اختیاری پروژه:

امکان پیاده‌سازی آرایه و متغیرهای گلوبال را به شکل زیر فراهم کنید:

۱. خروجی توابع یک آرایه نخواهد بود.
۲. متغیرهای گلوبال حتما به صورت یک تک مقدار خواهند بود و نه یک آرایه.
۳. متغیرهای گلوبال حتما در ابتدا برنامه و قبل از هر تابعی آورده می‌شوند.
۴. پیاده سازی امکان فراخوانی توابع: به صورت پیش فرض، برای پروژه، شما تنها باید تابع main را کامپایل و کد آن را تولید کنید و در مورد توابع دیگر مجازید هر کاری انجام دهید(آنها را نادیده بگیرید و یا این که کد آنها را تولید کنید ولی فراخوانی نشوند.) اما در صورتی که کامپایلر بتواند امکان فراخوانی توابع را فراهم سازد، نمره‌ی اضافه در نظر گرفته می‌شود.

۵. یکی از روش‌های خطایابی را پیاده کنید: برای مثال همانند کامپایلر زبان سی، بعد از یافتن اولین خطا، به کامپایل ادامه می‌دهد و خطاهای دیگر را نیز پیدا می‌کند؛ البته این کار را به درستی انجام دهید و نه این که تمام توکن‌های بعدی را به عنوان خطا تشخیص دهید(توجه: ممکن است با دانشی که در کلاس بایسون کسب کردید قادر به پیاده‌سازی این بخش نباشید و لازم است در مورد آن تحقیق کنید.).

مثال:

۱. تعریف متغیر گلوبال:

```
Int c;

.data:
X:   .word        5
```

۲. تغییر مقدار متغیر گلوبال:

```
Int c = 3;

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

چنانچه هر سوال و یا ابهامی در مورد پروژه دارید، می‌توانید از طریق زیر، در میان بگذارید:

golgolniamilad@gmail.com

Telegram ID: @COMPULAR