فاز نهایی پروژه کامپایلر

این فاز متشکل از دو فاز دوم و سوم می باشد که هر دو در یک فایل فراهم گردیده و مهلت و تحویل این دو فاز در یک تاریخ می باشد.

فازدوم : پارسر

فاز دوم پروژه شامل پیاده سازی واحد تحلیلگر معنایی است که در اینجا پارسر نامیده میشود. وظیفه اصلی واحد پارسر تعیین صحت کد ورودی به زبان شبه C میباشد. این برنامه با دریافت یک فایل ورودی به زبان شبه C باید تعیین نماید که آیا کد وارد شده صحیح می باشد یا دارای خطاست. در پیاده سازی این فاز شما از Bison استفاده می کنید که ساختاری شبیه Flex داشته و ورودی آن یک گرامر می باشد. نوشتن گامر به عهده ی خودتان است.

گرامر توصیف کننده زبان علاوه بر امکان تولید تابع main() باید شامل دستورالعملهای زیر باشد:

- دستورالعمل تعریف متغیر با امکان تعریف چند متغیر به صورت ساده، آرایه.
- دستورالعملهای if، for، while، do while، Assignment، Switch case
- امکان تعریف تابع و فراخوانی تابع(امتیازی)

توجه 1: دقت کنید که گرامر عبارات ریاضی (E) باید شامل عملگر ضرب، تقسیم، جمع، ++، - -، تفریق و منهای تک عملوندی باشد. علاوه بر این باید شامل اعداد (num) به صورت صحیح و اعشاری باشد.

توجه 2: دقت کنید که انواع تایپ های موجود شامل int، char و bool میباشد. Int های خود را 16 بیت فرض کنید که توانایی ریختن آنها را در register های 16 بیتی خود داشته باشیم. (درباره ی register ها صحبت خواهد شد)(تعریف کردن int 32 بیت امتیازی)

توجه 3: دقت کنید که گرامر مربوط به عبارات بولین شامل عمگر های کوچکتر(مساوی)، بزرگتر(مساوی) ، برابر و نقیض (!) می باشد.

توجه 4: فرض کنید که دو دستور cout و cin جزء خود زبان C بود و نیازی به کتابخانه نمی باشد. در این فاز پروژه (در گرامر ، پارسر و code generator) این دو دستور را پیاده سازی کنید.

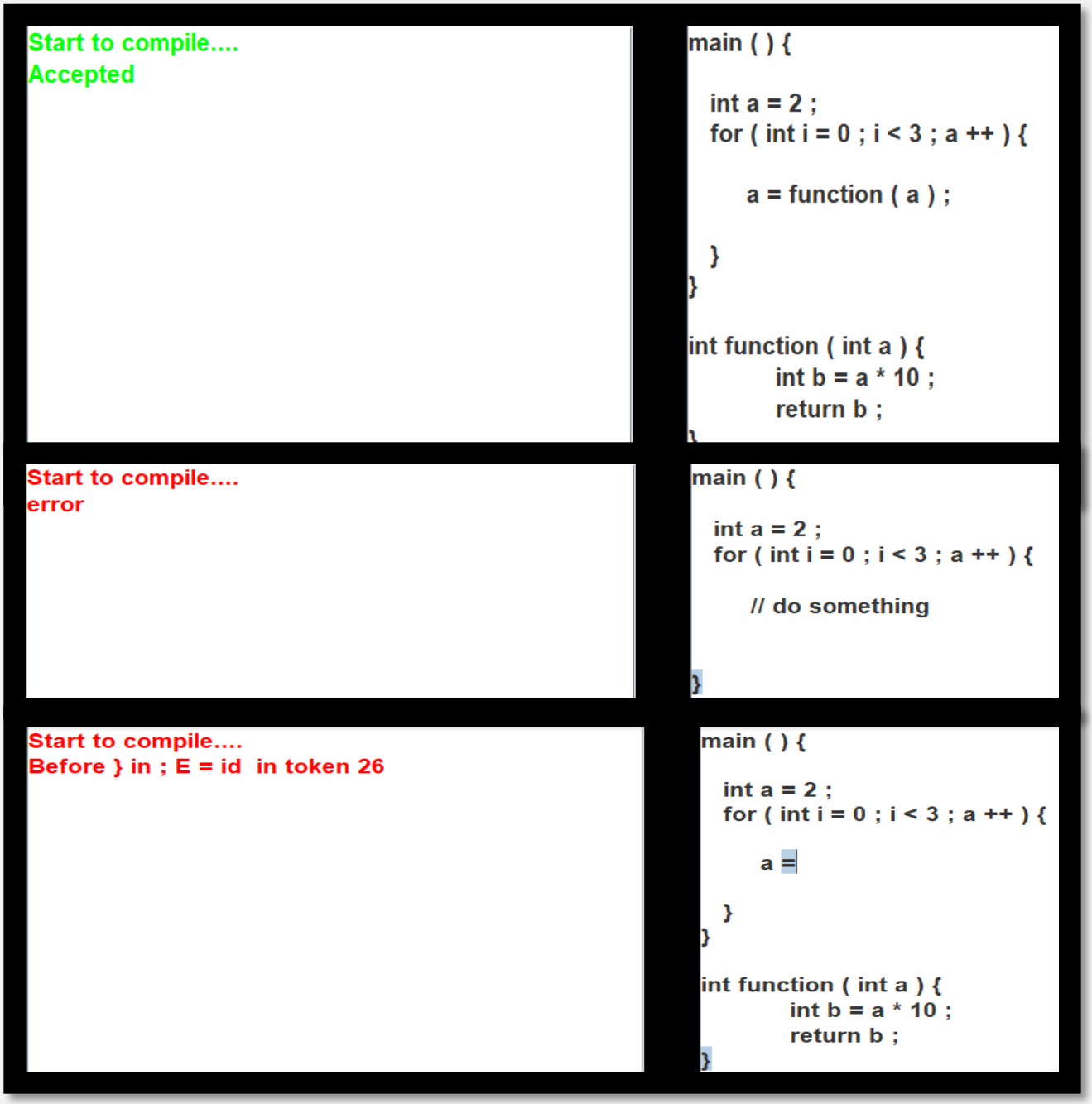
برای آشنایی و یادگیری Bison :

1) http://alumni.cs.ucr.edu/~lgao/teaching/bison.html

2) https://www.youtube.com/watch?v=yTXCPGAD3SQ

3) http://dinosaur.compilertools.net/bison/bison_5.html

نمونه هایی از ورودی و خروجی برای پارسر : اگر فایل ورودی با گرامر شما تطابق داشت، Accept کرده و به مرحله ی Code Generator برود. اگر فایل ورودی با گرامر شما تطابق نداشت و یا دارای خطا بود، برنامه شما می بایست Error دهد. شما می توانید شماره ی توکن یا خط دارای خطا را چاپ کنید که پوئن مثبت حساب می شود.



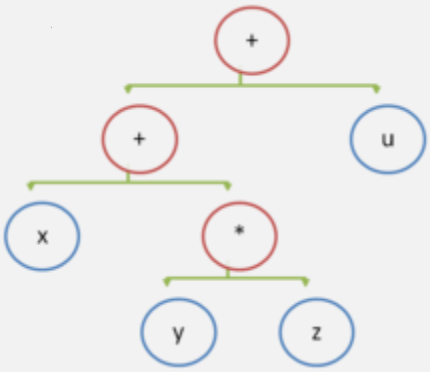
فازسوم : Code Generator

در این بخش از شما می خواهیم کد ورودی را در صورت عبور از مرحله ی تحلیلگر معنایی به کد هدف که در اینجا هدف ماشین های پشته(Stack Based Machine) ها می باشند تبدیل کنید.

در مهندسی کامپیوتر و پیاده‌سازی زبان‌های برنامه‌نویسی، ماشین پشته‌ای یک کامپیوتر واقعی یا شبیه‌سازی شده‌است که به جای استفاده از ثبات‌های تکی، از یک پشته برای ارزیابی زیردستورها در برنامه استفاده می‌کند. کامپیوتر پشته‌ای با مجموعه دستورالعمل‌هایی که به روش نشانه‌گذاری لهستانی معکوس (نشانه‌گذاری پسوندی) نوشته‌شده‌اند، برنامه‌نویسی شده‌است. اشین پشته‌ای، ثبات‌ها را با یک پشته پیاده‌سازی می‌کند. عملوندهای واحد محاسبه و منطق همواره دو ثبات بالایی موجود در پشته هستند و نتیجه واحد محاسبه و منطق در ثبات بالایی پشته ذخیره می‌شود. مجموعه دستورالعمل تقریباً تمام عملیات واحد محاسبه و منطق را با نشانه‌گذاری پسوندی (روش لهستانی معکوس)، که فقط در پشته به کار می‌آید نه در رجیسترها و سلول‌های حافظه، پیش می‌برد.

کامپایلرهای برای ماشین پشته‌ای ساده‌تر و سریع‌تر از کامپایلرها برای سایر ماشین‌ها

هستند. تولید کد بسیار جزیی و مستقل از کد اولیه یا کد بعدی است. برای مثال، برای دستور x+y*z+u، درخت و کد ترجمه شده معادل به صورت روبرو است:




```
push x
push y
push z
multiply
add
push u
add
```

معماری STACK MACHINE:

یک Stack machine شامل program memory و data stack میباشد. دستورالعمل های برنامه در program memory ذخیره میشوند و داده هایی که توسط دستورالعمل ها تغییر داده میشوند در stack میباشند. دستورالعمل ها داده ها را از پشته pop میکنند و پس از اعمال تغییرات دوباره آن هارا push میکنند.

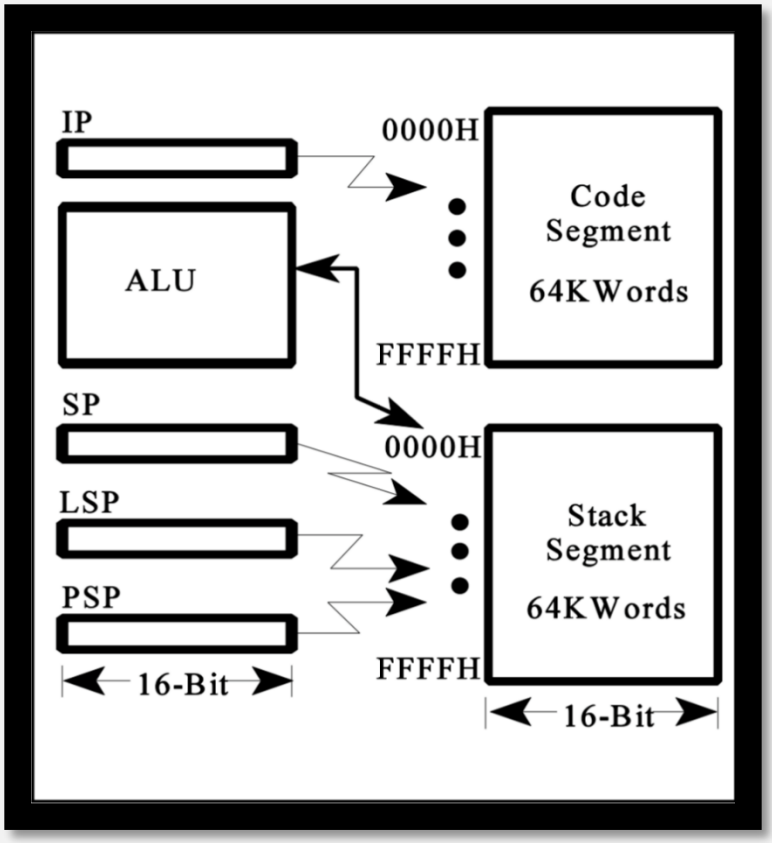
ماشین پشته ای مورد نظر ما شامل بخش های زیر میباشد:

1. Stack segment
2. ALU
3. Code segment

همچنین شامل چهار register به شرح زیر میباشد:

1. IP: اشاره میکند به آدرس دستورالعمل بعدی که باید اجرا شود.
2. Stack Pointer (SP) : اشاره میکند به آدرس top پشته.
3. Local Scope Pointer (LSP) : اشاره میکند به آدرس local data هایی که در طول تولید کد تعریف شده اند و در پشته ذخیره شده اند.

4. Parent Scope Pointer (PSP) :اشاره میکند به آدرس داده هایی خارج از scope فعلی که قابل دسترسی اند. (داده های global)

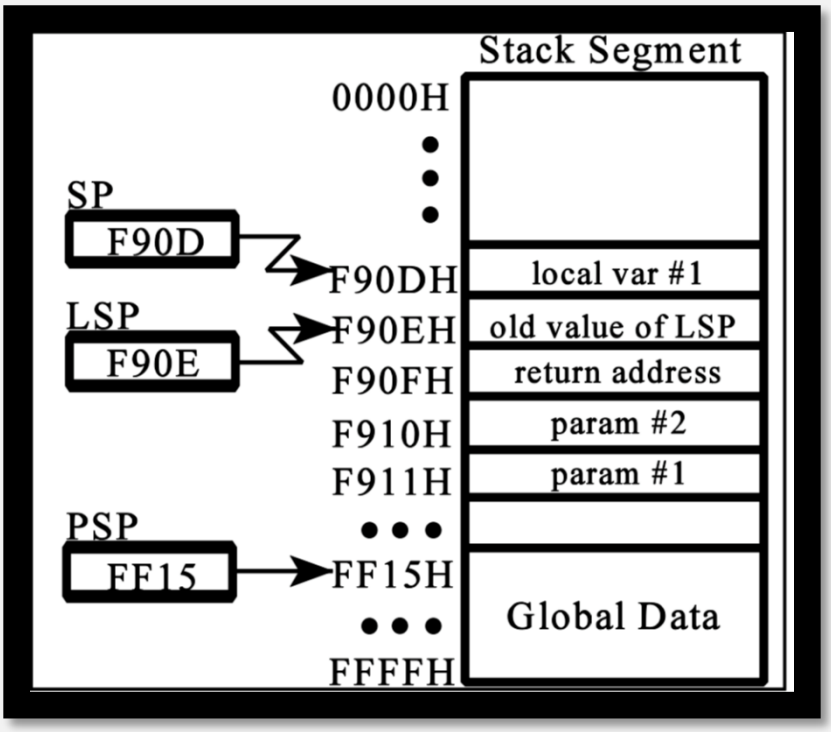


هر segment قابلیت ذخیره 64Kword را دارد که هر word دارای اندازه ی 16bit میباشد. هر register قابلیت ذخیره 16 bit را دارد که برای ذخیره ی آدرس بخش های مختلف segment ها استفاده میشود.. آدرس هر segment از 0000H شروع میشود و تا FFFFH ادامه میابد. هر داده ای که وارد Stack میشود در بزرگ ترین آدرس ممکن ذخیره میشود.

داده های Global در بزرگترین آدرس Stack دخیره میشوند که شروع آن از FFFFH میباشد. Normal Stack از جایی شروع میشود که داده های Global تمام شده باشند.

PSP به آدرس آخرین جایی که متغیر های Global در آن قرار دارند اشاره میکند بنابراین جدا کننده ی دو بخش normal stack و global data میباشد.

شکل زیر مثالی از stack میباشد که پس در حین اجرای تابع در حال استفاده میباشد. این تابع شامل دو پارامتر و یک متغیر محلی است.



نکته : به طور کلی شما هر طور که بخواهید میتوانید از این چهار register استفاده کنید و در آن چیز هایی که میخواهید را ذخیره کنید اما اهداف به کار بردن این register ها در بالا توضیح داده شد.

دستورالعمل های Stack machine:

دستورالعمل ها دو کلمه ای و یا یک کلمه ای میباشند و مطابق جدول زیر میتوانید از آن ها استفاده کنید.

Instruction	Description (all operands and results are on stack unless otherwise specified)
Directives:	
end	Physical end of the program (not a machine instruction)
exit	Halt the execution of the program and exit (not a machine instruction)
Arithmetic:	
add	Pop the top two locations, add, and push the result
div	Pop the dividend and divisor, divide, and push the quotient and then the remainder
mlt	Pop the multiplicand and multiplier, multiply, and push the result
neg	Pop the top location, negate, and push the result
sub	Pop subtrahend and minuend, subtract, and push the result
Logical:	
and	Pop the top two locations, perform bitwise AND, and push the result
not	Pop the top location, perform bitwise NOT, and push the result
or	Pop the top two locations, perform bitwise OR, and push the result
Input/Output:	
getc	Get a character byte from the standard input, pad it with zeros on the left, and push it
geti	Get a 16-bit integer from the standard input and push it
putc	Pop the top location and print the least significant byte as a character on the standard output
puti	Pop the top location and print it as a 16-bit integer on the standard output
Control Flow:	
call label	Push the return address and transfer the control to the instruction at address *label*
ret nloc	Pop the return address into IP and increment SP by nloc
cmp	Without popping stack, subtract the top location from the one below and push the result
jmp label	Unconditionally jump to the instruction at address *label*
jlt label	Pop the top location and jump to *label* if the popped location is less than 0
jgt label	Pop the top location and jump to *label* if the popped location is greater than 0
jz label	Pop the top location and jump to *label* if the popped location is zero
jnz label	Pop the top location and jump to *label* if the popped location is not zero
Stack Operations:	
push REG*	Push the value of REG
push int	Push a 16-bit integer value
push n[REG]	Push the contents of the location addressed by n+[REG]
pop REG	Pop the top location into REG
pop n[REG]	Pop the top of stack into the location addressed by n+[REG]
popi	Pop the top of stack, point to the location addressed by it, and pop the top into this location
rti	Pop the top of stack, point to the location addressed by it, and push the location's contents
xchg	Exchange the values of the top two locations
mov REG,SP	Set REG to the value of SP
isp n	Increment SP by n (a negative value of n will decrement it)

*REG is either LSP or PSP

برای مطالعه بیشتر میتوانید به مقاله زیر مراجعه کنید:

https://pdfs.semanticscholar.org/2338/2f3d3a72b91b191a8d0cc186eeec55a3a1a9.pdf