بسمه تعالی

طراحی کامپایلرها

نیم‌سال اول ۹۸ـ۹۷

مدرسان: دکتر قاسم‌ثانی و دکتر فیض بخش رانکوه

دانشکده مهندسی کامپیوتر

پروژه پایانی

پروژه عملی درس شامل پیاده‌سازی یک کامپایلر تک گذره برای نسخه ساده‌شده‌ی C است که قابلیت وجود تابع‌های تودرتو [1] به آن افزوده شده‌است. توضیحات کلی آن در ادامه آمده است. توجه کنید که استفاده از کدهای موجود در مرجع درس یا سایر کتب کامپایلر، در صورت تسلط بر آن کد و اعلام مأخذ در مستندات همراه پروژه اشکالی ندارد ولی استفاده از کدها و برنامه‌های موجود در سایت‌ها و کدهای سایر گروه‌ها (در همین نیم‌سال یا سال‌های گذشته) اکیدا ممنوع است و در اکثر موارد سبب مردودی در درس خواهد شد. در این مورد تفاوتی میان گروه دهنده یا گیرنده کد وجود ندارد.

[1] nested functions

مشخصات کامپایلر

- کامپایلر تک‌گذره است و از ۴ جزء تحلیل‌گر لغوی، تحلیل‌گر نحوی، تحلیل‌گر معنایی و مولد کد میانی تشکیل شده است. شما این اجزا را در ۳ فاز پیاده‌سازی می‌کنید:

فاز	اجزاء	نمره
۱	پیاده‌سازی تحلیل‌گر لغوی (اسکنر)	۰/۵
۲	پیاده‌سازی تحلیل‌گر نحوی (اسکنر و پارسر)	۰/۵
۳	پیاده‌سازی کامپایلر نهایی(اسکنر، پارسر، تحلیل‌گر معنایی و مولد کد میانی)	۳

- ورودی کامپایلر همواره یک فایل متنی حاوی برنامه‌ای است که کامپایلر شما باید آن را ترجمه کند.
- شما باید در هر فاز، بخشی از کامپایلر مورد نظر را پیاده‌سازی کنید و با روی هم سوار کردن تدریجی این بخش‌ها در نهایت به کامپایلر نهایی دست بیابید.
- نمره‌ی در نظر گرفته شده برای هر فاز مستقل از فازهای دیگر است.

گرامر NC-Minus

گرامری که در ادامه آمده است مربوط به بخشی از زبان C است که قابلیت وجود تابع‌های تودرتو به آن افزوده شده است. در این گرامر، پایانه ها پررنگ تر از غیرپایانه ها نمایش داده شده اند. این گرامر صرفاً برای اطلاع شما در اختیارتان قرار گرفته و شما در فاز اول نیازی به این اطلاعات ندارید.

1. program → declaration-list **EOF**
2. declaration-list → declaration-list declaration | ϵ
3. declaration → var-declaration | fun-declaration
4. var-declaration → type-specifier **ID ;** | type-specifier **ID** [**NUM**] **;**
5. type-specifier → **int** | **void**
6. fun-declaration → type-specifier **ID** (params) compound-stmt
7. params → param-list | **void**
8. param-list → param-list **,** param | param
9. param → type-specifier **ID** | type-specifier **ID** [ ]
10. compound-stmt → **{** declaration-list statement-list **}**
11. statement-list → statement-list statement | ϵ
12. statement → expression-stmt | compound-stmt | selection-stmt | iteration-stmt | return-stmt | switch-stmt
13. expression-stmt → expression **;** | **continue ;** | **break ;** | **;**
14. selection-stmt → **if** (expression) statement **else** statement
15. iteration-stmt → **while** (expression) statement
16. return-stmt → **return ;** | **return** expression **;**
17. switch-stmt → **switch** (expression) **{** case-stmts default-stmt **}**
18. case-stmts → case-stmts case-stmt | ϵ
19. case-stmt → **case NUM :** statement-list
20. default-stmt → **default :** statement-list | ϵ

21. expression → var = expression | simple-expression

22. var → **ID** | **ID** [expression]

23. simple-expression → additive-expression relop additive-expression | additive-expression

24. relop → < | ==

25. additive-expression → additive-expression addop term | term

26. addop → + | -

27. term → term * signed-factor | signed-factor

28. signed-factor → factor | + factor | - factor

29. factor → (expression) | var | call | **NUM**

30. call → **ID** (args)

31. args → arg-list | ϵ

32. arg-list → arg-list , expression | expression

توضیحات تحلیل‌گر لغوی

شما در این فاز باید با استفاده از روشی که در کلاس آموخته‌اید، فایل متنی که به شما داده می‌شود را tokenize کنید. مراحل پیاده‌سازی تحلیل‌گر لغوی تا رسیدن به DFA را می‌توانید به صورت دستی انجام دهید. برنامه‌ای که می‌نویسید باید یک تابع به نام get_next_token داشته باشد که با فراخوانی آن، تا زمانی که یک token جدید تشخیص داده شود، برنامه موجود در فایل ورودی را به صورت حرف به حرف بخواند. این روند تا تشخیص تمام token های فایل ورودی ادامه پیدا می‌کند.

توجه داشته باشید که انتظار داریم تمام token ها را در یک فایل خروجی به نام scanner.txt قرار دهید. در این فایل به ازای هر خط از فایل ورودی یک خط قرار دارد. در واقع ساختار این فایل بدین صورت است که در ابتدای هر خط عددی می‌آید که نشان‌دهنده شماره خط فایل ورودی است. سپس در ادامه‌ی این خط به ترتیب تمام token های موجود در خط متناظر از فایل ورودی آورده می‌شوند.

هر token نیز در قالب یک دوتایی به شکل (token type, token string) می‌باشد. دقت کنید انواع token هایی که باید تشخیص دهید در جدول پایین آمده است:

Token Type	**Description**
NUM	Any string matching: [**0-9**]$^+$
ID	Any string matching: [**A-Za-z**][**A-Za-z0-9**]*
KEYWORD	**if else void int while break continue switch default case return**
SYMBOL	; : , [ ] ( ) { } + - * = < ==
COMMENT	Any string between a /* and a */ OR any string after a // and before a \n
WHITESPACE	blank (ASCII 32), \n (ASCII 10), \r (ASCII 13), \t (ASCII 9), \v (ASCII 11), \f (ASCII 12)
EOF	End of file

ملاحظات تحلیل‌گر لغوی

- در فایل ورودی هنگامی شماره خط عوض می‌شود که \n آمده باشد.
- در صورتی که متن فعلی بخشی از یک کامنت باشد، هر گونه کاراکتری می‌تواند در آن ظاهر شود.
- در هنگام تشخیص token ها نیازی به ذخیره و گزارش WHITESPACE ها و COMMENT ها نیست و این token ها را در نظر نگیرید.

خطاپردازی در تحلیل لغوی

به منظور رفع خطاهای پیش‌آمده هنگام تحلیل لغوی باید از روش Panic Mode استفاده کنید. در این روش، خطاپردازی که همراه تحلیل‌گر لغوی طراحی می‌کنید، می‌بایست هنگام برخورد با یک خطا (کاراکتر غیر مجاز و یا خانه‌ای خالی در جدول) در میانه‌ی تشخیص یک token، تمامی کاراکترهای گرفته شده از ورودی را از ابتدای آن token کنار گذاشته و فرایند تشخیص token ها را از بعد از کاراکتر آخر، از سر گیرد.

توجه داشته باشید که همراه با فایل خروجی scanner.txt باید یک فایل دیگر با نام lexical_errors.txt نیز ارائه کنید که در هر خط از با شماره خطی که در آن خطایی رخ داده است می‌آید و در ادامه آن دوتایی هایی می‌آید که هر کدام به شکل (string, message) است که string رشته‌ای است که هنگام خطا کنار گذاشته شده و پس از آن برای تشخیص token بعدی بررسی می‌گردد. message نیز پیغام خطای لغوی موردنظر را نشان می‌دهد. توجه داشته باشید که در این فایل خطاها باید به ترتیب رخ دادن نوشته شده باشند.

نمونه‌ی ورودی

توجه کنید که صحت خروجی تحلیل‌گر لغوی شما و خطاپرداز مربوطه آن با ورودی دادن فایل‌های تست از پیش ساخته شده بررسی خواهد شد. این تست‌ها به گونه‌ای خواهند بود که نکات مختلفی را از منظر تحلیل لغوی و هم از منظر خطاپردازی در بر خواهند داشت. نمونه‌ای ساده از یک فایل ورودی به همراه خروجی‌های موردنظر در ادامه آورده شده است:

```
void main(void){
    int a = 0;
    // comment2
    a = 2 + +2;
    a = a + -3;
    cde = a;
    if (b /* comment1 */ == 3) {
        a = 3;
        cd!e = -7;
    }
    else
    {
        b = a < cde;
        {cde = @2;
    }}
    return;
}
```

نمونه فایل ورودی

```
1. (KEYWORD, void) (ID ,main), (SYMBOL, (), (KEYWORD, void) (SYMBOL,)) (SYMBOL,{)
2. (KEYWORD, int) (ID, a) (SYMBOL, =) (NUM, 0) (SYMBOL, ;)
4. (ID, a) (SYMBOL, =) (NUM, 2) (SYMBOL, +) (SYMBOL, +) (NUM, 2) (SYMBOL, ;)
5. (ID, a) (SYMBOL, =) (ID, a) (SYMBOL, +) (SYMBOL, -) (NUM, 3) (SYMBOL, ;)
6. (ID, cde) (SYMBOL, =) (ID, a) (SYMBOL, ;)
7. (KEYWORD, if) (SYMBOL, () (ID, b) (SYMBOL, ==) (NUM, 3) (SYMBOl, )) (SYMBOL, {)
8. (ID, a) (SYMBOL, =) (NUM, 3) (SYMBOL, ;)
9. (ID, e) (SYMBOL, =) (SYMBOL, -) (NUM, 7) (SYMBOL, ;)
10. (SYMBOL, })
11. (KEYWORD, else)
12. (SYMBOL, {)
13. (ID, b) (SYMBOL, =) (ID, a) (SYMBOL, <) (ID, cde) (SYMBOL, ;)
14. (SYMBOL, {) (ID, cde) (SYMBOL, =) (NUM, 2) (SYMBOL, ;)
15. (SYMBOL, }) (SYMBOL, })
16. (KEYWORD, return) (SYMBOL, ;)
17. (SYMBOL, })
```

نمونه فایل خروجی تحلیل‌گر لغوی

```
9. (cd!, invalid input)
14. (@, invalid input)
```

نمونه فایل خروجی خطاپرداز لغوی

انجام پروژه

- پروژه باید به صورت انفرادی و یا گروه‌های دو نفره انجام شود. در صورتی که میزان مشارکت اعضای گروه‌های دو نفره با یکدیگر برابر نباشد، فردی که مشارکت کمتری داشته، نمره‌ی کمتری نسبت به دیگری می‌گیرد.
- در صورتی که هر گونه سوالی در رابطه با تعریف پروژه دارید، آن را از طریق کوئرا مطرح نمایید.
- مهلت بارگذاری سورس کد پروژه برای فاز اول، ساعت ۲۳:۵۹ روز جمعه ۲۳ فروردین است.
- تحویل حضوری دو فاز اول و دوم همزمان بوده و زمان‌بندی دقیق آن متعاقبا اعلام خواهد شد. (توجه کنید که حضور هر دو عضو گروه‌های دو نفره در جلسه‌ی تحویل الزامی است.)

موفق باشید.