گزارش کار پروژه میان‌ترم نظریه زبان ها و ماشین ها

پارسا اسدی - ۴۰۲۳۱۰۷۲

مرحله اول ساختار کلی و کلاس‌های ساخته شده:

ابتدای کلاس های Transition و Rule را تعریف و می‌کنیم:

```
class Rule
{
public:
    // Attributes
    char from;
    string to;

    // Constructor
    Rule(char a, string b)
    {
        from = a;
        to = b;
    }
};

class Transition
{
public:
    // Attributes
    string from;
    string by;
    string to;

    // Constructor
    Transition(string a, string b, string c)
    {
        from = a;
        by = b;
        to = c;
    }
};
```

حالا کلاس‌های Grammar و NFA را با توجه به کلاس های فوق تعریف می‌کنیم:

```
class Grammar
{
public:
    // Attributes
    const vector<Rule> rules;
    const vector<char> alphabets;
    const vector<char> variables;
    char start;
    // Constructor
    Grammar(char s, vector<char> a, vector<char> v, vector<Rule> r) :
alphabets(a), variables(v), rules(r)
    {
        start = s;
    }
};
class NFA
{
public:
    vector<string> states;
    vector<Transition> transitions;
    vector<char> alphabets;
    string start;
    vector<string> finalStates;

    NFA() = default;
    NFA(vector<char> a, vector<string> s, vector<Transition> t, string start,
vector<string> f)
        : alphabets(a), states(s), transitions(t), start(start), finalStates(f)
    {
    }
};
```

نکته: با توجه به اینکه NFA فوق تمامی خواص DFA ها را نیز در بر می‌گیرد نیازی به تعریف کلاس DFA نیست.

هدف ما این است که ابتدا هنگام ورودی گرفتن یک شی از کلاس Grammar ساخته سپس با استفاده از تابع RGtoNFA گرامر فوق را به NFA با حرکات لاندا تبدیل کنیم. NFA فوق را با استفاده از تابع noLanda به یک NFA بدون حرکات لاندا تبدیل می‌کنیم و سپس با

استفاده از تابع NFAtoDFA این NFA را به DFA تبدیل کنیم و بعد هر یک از این DFA ها را در vector<NFA> DFAs ذخیره کرده و روی آن‌ها عملیات انجام دهیم.

نکته: در هنگام ورودی گرفتن با کاراکتر ε به مشکلاتی برخواهیم خورد که از همان ابتدای کار برای کار کردن راحت‌تر با حرکات لاندا این کاراکتر را تبدیل به رشته "epsilon" می‌کنیم.

پیاده سازی تابع‌ها:

تابع isLeftLinear بررسی می‌کند که گرامر موجود خطی از سمت چپ هست یا خیر. اگر نبود با توجه به فرض پروژه که ورودی ما یک گرامر منظم است نتیجه می‌گیریم که خطی از سمت راست است.

تابع RGtoNFA:

ابتدا به ازای هر variable در گرامر یک state به state های NFA اضافه می‌کنیم. همچنین F را به state های NFA اضافه می‌کنیم و هرگاه در سمت چپ Rule هیچ متغیری وجود نداشت سمت راست Rule به ازای رشته سمت چپ Rule به state نهایی یا همان F می‌رود.
اگر سمت راست Rule حاوی متغیر بود سمت چپ Rule با الفبای سمت راست Rule به state متناظر با متغیر فوق می‌رود.
نکته: اگر تعداد الفبای موجود در سمت راست قاعده بیش از یکی بود state های میانی را اضافه می‌کنیم.

تابع getLandaClosure:

همان طور که از اسم آن مشخص است بستار لاندای هر state مد نظر را به ما می‌دهد.

تابع noLanda:

این تابع با استفاده از تابع getLandaClosure تبدیل‌های لاندا را حذف کرده و NFA مورد نظر را به NFA بدون حرکات لاندا تبدیل می‌کند.

تابع NFAtoDFA:

NFA بدون حرکات لاندا را دریافت کرده و به DFA تبدیل می‌کند. (هر استیت در این ماشین یکی از اعضای مجوعه توانی استیت‌های NFA ورودی است و در انتهای تابع استیت‌های غیر قابل دسترس از استیت ابتدایی حذف می‌شوند.)

تابع operationHandling:
مجموعه از DFA ها را گرفته و با توجه به عملیات درخواست شده تابع مورد نظر را صدا می‌کند.
تابع‌های complementOP و unionOP و intersectionOP با توجه به تئوری‌های درس پیاده سازی شده اند.

نکته: در نهایت در DFA به دست آمده از operationHandling با استفاده از تابع minimize استیت‌هایی که به استیت‌های نهایی دسترسی ندارند یا به اصطلاح dead states با استفاده از الگوریتم BFS حذف شده اند.

و در نهایت تابع renameStates برای اسم گذاری بهتر استیت‌ها استفاده می‌شود.