UNIVERSIDADE DE SÃO PAULO
INSTITUTO DE CIÊNCIAS MATEMÁTICAS E DE COMPUTAÇÃO
http://www.icmc.usp.br/

Curso: Bacharelado em Ciência da Computação

Turma: SCC0201 – Introdução à Ciência da Computação II

Professores: Moacir Ponti (moacir@icmc.usp.br)
Rodrigo Fernandes de Mello (mello@icmc.usp.br)

Alunos PAE: Lucas Pagliosa (lucas.pagliosa@usp.br)
Ricardo Fuzeto (ricardofuzeto@gmail.com)
Yule Vaz (yule.vaz@ups.br)

Trabalho 03: JSON Parser

1 Regras e Especificações

O trabalho descrito a seguir é individual e não será tolerado qualquer tipo de plágio ou cópia em partes ou totalidade do código. Caso seja detectada alguma irregularidade, os envolvidos serão chamados para conversar com algum dos professores responsáveis pela disciplina.

A entrega deverá ser feita única e exclusivamente por meio do sistema run.codes no endereço eletrônico `https://run.codes`, até a data estipulada pelos professores. Apesar da data de entrega estar suscetível à variações, fique atento com o prazo de entrega previamente estabelecido pelo site. O run.codes está programado para não aceitar submissões após este horário e não serão aceitas entregas fora do sistema.

Leia a descrição do trabalho com atenção e várias vezes, anotando os pontos principais e as possíveis formas de resolver o problema. Comece a trabalhar o quanto antes para que você não fique com dúvidas e que consiga entregar o trabalho a tempo.

2 Descrição do Problema

Uma linguagem fonte é um texto escrito em ASCII ou UNICODE (portanto, uma coleção de palavras ou símbolos) que definem instruções ou informações a serem processadas por um computador. Tal linguagem é definida por regras específicas de acordo com uma gramática G, normalmente definida por comandos em alto nível, de modo que humanos possam entende-las e replicá-las mais facilmente do que linguagens objeto, sequência de instruções em binário pronta para ser interpretada pela máquina.

Uma gramática pode ser descrita na forma de um texto, onde cada linha define produções (regras) entre símbolos não terminais e terminais (ou *tokens*), da seguinte maneira:

```
1 Não terminal : TOKENS
```

tal que um não terminal, identificado por palavras cuja primeira letra é maiúscula, é uma "estrutura" que irá aparecer ao lado esquerdo da gramática, enquanto um token, identificado por palavras com letras maiúsculas, é uma sequência de caracteres que aparece a direita das produções.

Neste contexto, um compilador tem como propósito transformar uma linguagem fonte em linguagem objeto. Para tanto, um compilador é composto por diversas etapas de validação, a fim de verificar se o texto fornecido está de acordo ou não com as regras da gramática G.

Além disso, a complexidade do compilador se dá de acordo com o propósito da linguagem fonte utilizada. Neste cenário, um compilador de C++ por exemplo, pode ser dividido, resumidamente, na seguinte forma:

1. Análise Léxica: Leitura dos caracteres, também chamados de símbolos léxicos, e produção de tokens que serão manipulados por um *parser*.
2. Análise Sintática: O parser verifica se a ordem fornecida de tokens está de acordo com a regra gramatical G. → **Você deve fazer até aqui.**
3. Análise Semântica: Verifica se a sequência de tokens tem lógica e sentido.
4. Gerador de código intermediário: Gera um código em binário, mas ainda não o final.
5. Otimizações específicas da arquitetura: Otimiza o código intermediário.
6. Código objeto ou de máquina: Geração de um código final a ser interpretado pelo computador.

Note, no entanto, que um compilador de uma linguagem mais simples não necessariamente otimiza um código intermediário ou gera um código objeto. No caso de um compilador de **J**ava**S**cript **O**bject **N**otation (JSON), por exemplo, a função do compilador se resumo a uma verificação léxica, sintática e semântica. No entanto, neste trabalho você deve implementar, **SOMENTE** um analisar léxico e sintático para a gramática G (baseada em JSON), descrita a seguir:

```
G: Value EOF
Value: Object | Array | STRING | INTEGER | REAL | TRUE | FALSE | NULL
Object: { Members ? }
Members: Pair ( , Pair ) *
Pair: STRING : Value
Array: [ Elements ? ]
Elements: ( , Value ? ) *
```

Por sua vez, os tokens são definidos com o auxílio de expressões regulares intermediarias, identificadas por letras minúsculas, e caracteres da seguinte forma:

```
EOF: \0
STRING:  "  ~  ( \ ? " | \ ( b | f | n | r | t | u | \ ) ? ) * "
digit: [ 0 - 9 ]
sign: ( + | - )
int: sign ? ( 0 | [ 1 - 9 ] digit *)
frac: . digit +
exp: ( e | E ) sign ? digit +
NUMBER: int frac ? exp ?
TRUE: true
FALSE: false
NULL: null
```

Para reforçar, os símbolos de expressões regulares (*regex*) indicam:

- A | B → A ou B
- ? → Pode ocorrer zero ou um vez
- * → Pode ocorrer zero ou mais vezes
- + → Pode ocorrer uma ou mais vezes
- ( ) → Símbolos de controle para delimitar uma sequência de tokens
- ~ ( ) → Pode ocorrer tudo menos o que está dentro dos parênteses
- [A - Z] → Sequência de A até Z

3 Arquivos de Entrada e Saída

Cada linha de um arquivo de entrada será um objeto JSON. Você deverá ler cada linha e verificar por erros. No primeiro erro encontrado, aborde o programa e imprima:

```
ERROR line #\n
```

tal que `#` é a linha (JSON) do primeiro erro encontrado. Independente de erro ou não, a contagem de tokens e alguns não terminais encontrados até o momento do erro, de todos os JSON previamente processados e incluindo o atual, deverá ser informada da seguinte forma:

```
fprintf(file, "Number of Objects: %d\n", numberOfObjects);
fprintf(file, "Number of Arrays: %d\n", numberOfArrays);
fprintf(file, "Number of Pairs: %d\n", numberOfPairs);
fprintf(file, "Number of Strings: %d\n", numberOfStrings);
fprintf(file, "Number of Numbers: %d\n", numberOfNumbers);
fprintf(file, "Number of Trues: %d\n", numberOfTrues);
fprintf(file, "Number of Falses: %d\n", numberOfFalses);
fprintf(file, "Number of Nulls: %d\n", numberOfNulls);
```

A fim de reforçar e não gerar dúvidas, a contagem é do arquivo de entrada como um todo, e não de um único JSON.

3.1 Exemplo 1

Entrada:

```
{"adasd":2
```

Saída:

```
Error line 1\n
Number of Objects: 0\n
Number of Arrays: 0\n
Number of Pairs: 1\n
Number of Strings: 1\n
Number of Numbers: 1\n
Number of Trues: 0\n
Number of Falses: 0\n
Number of Nulls: 0\n
```

3.2 Exemplo 2

Entrada:

```
{"\raasd":}
```

Saída:

```
Error line 1
Number of Objects: 0
Number of Arrays: 0
Number of Pairs: 0
Number of Strings: 0
Number of Numbers: 0
Number of Trues: 0
Number of Falses: 0
Number of Nulls: 0
```

3.3 Exemplo 3

Entrada:

```
{"":{"":2}}\n
{"":    [,,,,,2, null, true]}\n
{"a":+}
```

Saída:

```
Error line 3\n
Number of Objects: 3\n
Number of Arrays: 1\n
Number of Pairs: 3\n
Number of Strings: 4\n
Number of Numbers: 2\n
Number of Trues: 1\n
Number of Falses: 0\n
Number of Nulls: 1\n
```

4 Dicas

Vejam os exemplos de arquivos de entrada e saída no run.codes. Lá vocês poderão ver outros exemplos e como é esperado o formato dos erros e saídas. Esse é um programa considerado difícil. Muitos alunos mesmo após passarem da disciplina de Compiladores têm dificuldades em implementar um parser. Dia 15/09 eu darei uma "aula" sobre o trabalho, em uma sala a ser definida. Presença será cobrada, pois é importante vocês irem para esclarecimentos sobre o trabalho. Além disso, estou a disposição para eventuais dúvidas no VICG, sala 100-7, Bloco 1, a qualquer horário.

5 Observações importantes

- Programe as impressões na tela EXATAMENTE como exemplificado no decorrer deste documento. Tome cuidado com pulos de linha, tabs, espaços, etc.
- Coloque dentro do zip todos os arquivos de código (`*.h *.c`), o makefile e um arquivo texto com o nome e número USP.