UNIVERSIDADE FEDERAL DE MATO GROSSO DO SUL
FACOM - FACULDADE DE COMPUTAÇÃO

COMPILADORES I - 2022/1

PROFA. BIANCA DE ALMEIDA DANTAS

Trabalho Prático
Descrição da 1^a Etapa – Análise Léxica

1 DESCRIÇÃO

A primeira parte do trabalho prático de nossa disciplina consiste na implementação, usando a linguagem C++, do analisador léxico para a linguagem hipotética *X++* descrita no livro "Como Construir um Compilador Utilizando Ferramentas Java" de Márcio Eduardo Delamaro. O analisador léxico deve ser capaz de reconhecer os *tokens* da linguagem e **ignorar espaços em branco e comentários**.

Tokens:

1. Identificadores (ID): Uma sequência de letras, dígitos e *underscores*, começando sempre com uma letra ou com *underscore*. A linguagem distingue letras maiúsculas e minúsculas;
2. Números inteiros (INTEGER_LITERAL): uma sequência de dígitos decimais, que denotam o número inteiro correspondente;
3. Operadores (OP): $<$, $>$, $<=$, $>=$, $+$, $-$, $*$, $/$, $\%$, $=$, $==$ e $!=$;
4. Separadores (SEP): (,), [,], {, }, ;, . e ,;
5. Literais do tipo *string* são delimitados por **"** e **"**.
6. Comentários: os comentários devem ser ignorados. Existem dois tipos: comentário de linha (iniciado com $//$ e indo até o final da linha) e comentário de bloco (iniciado com $/*$ e encerrado em $*/$ sem aninhamentos);
7. Espaços em branco: devem ser ignorados.

Observe que as palavras reservadas da linguagem, nesta etapa, serão reconhecidas como identificadores. Assim que estudarmos a estrutura da tabela de símbolos, iremos usá-la como auxiliar para diferenciar as palavras reservadas dos identificadores. As palavras reservadas são as seguintes: **class**, **extends**, **int**, **string**, **break**, **print**, **read**, **return**, **super**, **if**, **else**, **for**, **new**, **constructor**.

2 ESPECIFICAÇÕES

- O trabalho prático poderá ser realizado em grupos de, no máximo, 3 alunos **sem exceções**.
- A linguagem C++ deverá ser utilizada na implementação do trabalho.
- A entrega de todas as etapas deve ser realizada até o dia: **28/06/2022**.

UNIVERSIDADE FEDERAL DE MATO GROSSO DO SUL
FACOM - FACULDADE DE COMPUTAÇÃO

COMPILADORES I - 2022/1

PROFA. BIANCA DE ALMEIDA DANTAS

Trabalho Prático
Descrição da 2^a Etapa – Análise Sintática

1 DESCRIÇÃO

A segunda parte do trabalho prático de nossa disciplina consiste na implementação do analisador sintático descendente preditivo para a linguagem *X++*, descrita no livro "Como Construir um Compilador Utilizando Ferramentas Java" de Márcio Eduardo Delamaro, com algumas alterações, cujos tokens já foram descritos na primeira etapa do trabalho e cuja gramática encontra-se no final deste arquivo.

O analisador sintático deve ser capaz de percorrer o programa fonte, detectar e reportar erros. **Não é necessário implementar nenhuma técnica de recuperação de erros sintáticos**.

Os códigos fontes do programa desenvolvido serão compilados usando o compilador g++ no UNIX.

2 EXECUÇÃO E ENTRADA

O seu programa deve ser capaz de realizar a compilação de um arquivo de texto com a extensão **.xpp**, cujo nome será fornecido na linha de comando do terminal logo após o nome do executável de seu compilador. Por exemplo, se seu executável possuir o nome **xpp_compiler** e o arquivo de entrada for **teste1.xpp**, a seguinte instrução será digitada no terminal:

```
./xpp_compiler teste1.xpp
```

3 SAÍDA

O compilador deve emitir mensagens de erros, caso encontre algum, informando claramente o erro e a linha de ocorrência. Caso não sejam encontrados erros, o compilador deve imprimir que a compilação foi encerrada com sucesso. Todas as mensagens devem ser mostradas no terminal. Sugere-se usar como inspiração mensagens geradas por compiladores reais (como o próprio g++).

4 AVALIAÇÃO

O seu programa será compilado usando o comando:

```
g++ *.cpp -o xpp\_compiler
```

Caso a compilação gere erros e o executável não seja gerado, o trabalho receberá nota **zero**.

O programa será executado com n arquivos fontes, podendo conter erros ou não, e a nota

atribuída será proporcional ao número de testes cuja execução de seu compilador conseguir detectar os erros (ou a falta deles) corretamente. Testes em que a execução não gerar o resultado esperado serão zerados.

O programa deve receber a entrada e gerar a saída **exatamente** como especificado nas descrições das etapas, caso isso não ocorra, a nota será penalizada.

5 ESPECIFICAÇÕES

- O trabalho prático poderá ser realizado em grupos de, no máximo, 3 alunos **sem exceções**.
- A linguagem C++ deverá ser utilizada na implementação do trabalho.
- Se qualquer parte do trabalho for copiada ou "fortemente inspirada" nos trabalhos de outros grupos, o trabalho receberá nota **zero**.
- Entrevistas podem ser realizadas com todos os grupos.
- A entrega de todas as etapas deve ser realizada até o dia: **28/06/2022**.

6 GRAMÁTICA DA LINGUAGEM X++

1. *Program* → *ClassList*
   | ϵ
2. *ClassList* → *ClassDecl* *ClassList*
   | *ClassDecl*
3. *ClassDecl* → **class ID** *ClassBody*
   | **class ID extends ID** *ClassBody*
4. *ClassBody* → **{** *VarDeclListOpt ConstructDeclListOpt MethodDeclListOpt* **}**
5. *VarDeclListOpt* → *VarDeclList*
   | ϵ
6. *VarDeclList* → *VarDeclList VarDecl*
   | *VarDecl*
7. *VarDecl* → *Type* **ID** `VarDeclOpt` **;**
   | *Type* **[] ID** `VarDeclOpt` **;**
8. *VarDeclOpt* → **, ID** *VarDeclOpt*
   | ϵ
9. *Type* → **int**
   | **string**
   | **ID**
10. *ConstructDeclListOpt* → *ConstructDeclList*
    | ϵ
11. *ConstructDeclList* → *ConstructDeclList ConstructDecl*
    | *ConstructDecl*
12. *ConstructDecl* → **constructor** *MethodBody*
13. *MethodDeclListOpt* → *MethodDeclList*
    | ϵ
14. *MethodDeclList* → *MethodDeclList MethodDecl*
    | *MethodDecl*
15. *MethodDecl* → *Type* **ID** *MethodBody*
    | *Type* **[] id** *MethodBody*
16. *MethodBody* → **(** *ParamListOpt* **) {** *StatementsOpt* **}**
17. *ParamListOpt* → *ParamList*
    | ϵ
18. *ParamList* → *ParamList* **,** *Param*
    | *Param*
19. *Param* → *Type* **ID**
    | *Type* **[] ID**
20. *StatementsOpt* → *Statements*
    | ϵ
21. *Statements* → *Statements* *Statement*
    | *Statement*
22. *Statement* → *VarDeclList*
    | *AtribStat* **;**
    | *PrintStat* **;**
    | *ReadStat* **;**
    | *ReturnStat* **;**
    | *SuperStat* **;**

| *IfStat*
| *ForStat*
| **break ;**
| **;**

23. *AtribStat* → *LValue* **=** *Expression*
| *LValue* **=** *AllocExpression*
24. *PrintStat* → **print** *Expression*
25. *ReadStat* → **read** *LValue*
26. *ReturnStat* → **return** *Expression*
27. *SuperStat* → **super (** *ArgListOpt* **)**
28. *IfStat* → **if (** *Expression* **) {** *Statements* **}**
| **if (** *Expression* **) {** *Statements* **} else {** *Statements* **}**
29. *ForStat* → **for (** *AtribStatOpt* **;** *ExpressionOpt* **;** *AtribStatOpt* **) {** *Statements* **}**
30. *AtribStatOpt* → *AtribStat*
| ϵ
31. *ExpressionOpt* → *Expression*
| ϵ
32. *LValue* → **ID** *LValueComp*
| **ID [** *Expression* **]** *LValueComp*
33. *LValueComp* → **. ID** *LValueComp*
| **. ID [** *Expression* **]** *LValueComp*
| ϵ
34. *Expression* → *NumExpression*
| *NumExpression* **RelOp** *NumExpression*
35. *AllocExpression* → **new ID (** *ArgListOpt* **)**
| *Type* **[** *Expression* **]**
36. *NumExpression* → *Term* **+** *Term*
| *Term* **-** *Term*
| *Term*
37. *Term* → *UnaryExpression* ***** *UnaryExpression*
| *UnaryExpression* **/** *UnaryExpression*
| *UnaryExpression* **%** *UnaryExpression*
| *UnaryExpression*
38. *UnaryExpression* → **+** *Factor*
| **-** *Factor*
39. *Factor* → **INTEGER_LITERAL**
| **STRING_LITERAL**
| *LValue*
| **(** *Expression* **)**
40. *ArgListOpt* → *ArgList*
| ϵ
41. *ArgList* → *ArgList* **,** *Expression*
| *Expression*

UNIVERSIDADE FEDERAL DE MATO GROSSO DO SUL
FACOM - FACULDADE DE COMPUTAÇÃO

COMPILADORES I - 2022/1

PROFA. BIANCA DE ALMEIDA DANTAS

Trabalho Prático
Descrição da 3^a Etapa – Análise Semântica

1 DESCRIÇÃO

A terceira etapa do trabalho prático de nossa disciplina consiste na implementação de algumas funcionalidades para verificação semântica da linguagem *X++*, descrita no livro "Como Construir um Compilador Utilizando Ferramentas Java" de Márcio Eduardo Delamaro. As funcionalidades que devem ser implementadas deverão ser realizadas concomitantemente ao processo de análise sintática, sendo, assim, uma atividade guiada pela sintaxe.

Lembrem-se que essa etapa vale dois pontos na avaliação do trabalho, mas que ela não é obrigatória para que ele seja considerado. As duas etapas anteriores (léxica e sintática) são obrigatórias.

As seguintes atividades devem ser realizadas:

- Verificação de declaração de classes: ao declarar uma classe, ela deve ser inserida na tabela de símbolos do escopo corrente;
- Verificação de declaração de variáveis e métodos: ao acessar uma variável ou um método, deve-se verificar se sua declaração existe e pode ser acessada a partir do escopo corrente;
- Verificação de tipos em comandos de atribuição e chamadas de métodos: quando uma atribuição for realizada deve-se verificar se a expressão à direita é compatível com o *lvalue*. Ao invocar um método, deve-se verificar a compatibilidade dos tipos de seus parâmetros reais com os parâmetros formais, bem como a compatibilidade do tipo de retorno com o *lvalue*, caso a invocação esteja em um comando de atribuição.

Observe que para a realização das atividades descritas, é necessário que a tabela de símbolos seja manipulada de maneira coerente, lidando com o gerenciamento dos ponteiros para a tabela de símbolos global e para a tabela de símbolos corrente. Isso envolve saber o momento correto para a criação de novas tabelas para novos escopos e, eventualmente, para abandonar um escopo e voltar para o escopo que o englobe.

Erros apenas semânticos não devem parar a execução do processo de compilação. O seu compilador pode continuar executando normalmente apenas apresentando a mensagem adequada.

Os códigos fontes do programa desenvolvido serão compilados usando o compilador g++.

2 EXECUÇÃO E ENTRADA

O seu programa deve ser capaz de realizar a compilação de um arquivo de texto com a extensão **.xpp**, cujo nome será fornecido na linha de comando do terminal logo após o nome do executável de seu

compilador. Por exemplo, se seu executável possuir o nome **xpp_compiler** e o arquivo de entrada for **teste1.xpp**, a seguinte instrução será digitada no terminal:

```
./xpp_compiler teste1.xpp
```

3 SAÍDA

O compilador deve emitir mensagens de erros, caso encontre algum, informando claramente o erro e a linha de ocorrência. Caso não sejam encontrados erros, o compilador deve imprimir que a compilação foi encerrada com sucesso. Todas as mensagens devem ser mostradas no terminal. Sugere-se usar como inspiração mensagens geradas por compiladores reais (como o próprio g++).

4 AVALIAÇÃO

O seu programa será compilado usando o comando:

```
g++ *.cpp -o xpp\_compiler
```

Caso a compilação gere erros e o executável não seja gerado, o trabalho receberá nota zero.

O programa será executado com n arquivos fontes, podendo conter erros ou não, e a nota atribuída será proporcional ao número de testes cuja execução de seu compilador conseguir detectar os erros (ou a falta deles) corretamente. Testes em que a execução não gerar o resultado esperado serão zerados.

O programa deve receber a entrada e gerar a saída **exatamente** como especificado nas descrições das etapas, caso isso não ocorra, a nota será penalizada.

5 ESPECIFICAÇÕES

- O trabalho prático poderá ser realizado em grupos de, no máximo, 3 alunos **sem exceções**.
- A linguagem C++ deverá ser utilizada na implementação do trabalho.
- Se qualquer parte do trabalho for copiada ou "fortemente inspirada" nos trabalhos de outros grupos, o trabalho receberá nota **zero**.
- Entrevistas podem ser realizadas com todos os grupos.
- A entrega de todas as etapas deve ser realizada até o dia: **28/06/2022**.

6 AVALIAÇÃO

O trabalho prático será avaliado utilizando os seguintes critérios:

- A etapa de análise semântica valerá 20% da nota total.
- O funcionamento correto do analisador sintático em conjunto com o léxico valerá 70% da nota total.
- Trabalhos que não compilarem receberão nota **zero**.
- Um conjunto contendo $numTestes$ programas de teste será executado, com possíveis erros, e para cada teste seu programa deve emitir mensagens de erro (se houver) e, ao final, informar o término do processo de compilação.

- A cada teste para o qual o seu compilador executar corretamente, o grupo receberá pontuação equivalente $\frac{1}{numTestes} * nota\ total$. Executar corretamente significa terminar o percurso do programa de entrada e emitir mensagens para todos os erros do programa, se houver.
- Execuções parcialmente corretas receberão uma porcentagem da nota total.
- Se o seu programa travar na execução de um teste, a pontuação para tal teste será zerada.