Universidade Federal de Juiz de Fora
Instituto de Ciências Exatas
Departamento de Ciência da Computação
Disciplina: Teoria dos Compiladores
Professor: Marcelo Bernardes Vieira

PROJETO E IMPLEMENTAÇÃO DE UM COMPILADOR – 2ª PARTE

Projete e implemente um analisador sintático para a linguagem C--, definida pela gramática livre de contexto listada no fim deste documento:

1) Use a solução da parte I do projeto como ponto inicial para a implementação;
2) Construa um Parser para a linguagem C--. Converta a gramática C-- em uma gramática LL(1) ou LR(1);
3) Durante o *parsing,* crie toda a **árvore de sintaxe abstrata** em memória;
4) Adicione um esquema de **recuperação** de erros com sincronização. Implemente no mínimo a recuperação do tipo *phrase-level*.
5) **Atenda a todas as especificações mínimas a seguir**. Lembre-se que este trabalho faz parte de um projeto muito maior.

Documentação mínima:

A documentação, a ser entregue **em papel**, deve ser como um **pequeno artigo** (objetivo, impessoal e completo) que explica os detalhes do quê o programa faz, como faz, e ainda apresenta conclusões obtidas pelo trabalho. É um documento a parte que deve conter pelo menos as seguintes estruturas (seções):

1) Título:
   - Apresente o título do trabalho e/ou nome do compilador;
   - Nome dos integrantes do grupo seguidos dos pesos de cada um. **A soma dos pesos deve ser 1**.
2) Visão geral:
   - Dê uma visão geral do funcionamento do programa;
   - Descrição dos módulos e suas interdependências. Apresente uma breve descrição de cada módulo bem como um diagrama, por exemplo, mostrando suas inter-relações:
     i. Analisador sintático:
        1. Descreva e justifique a forma com a qual vocês implementaram o analisador sintático;
        2. Apresente a Gramática LL(1) ou LR(1) resultante:
        3. Apresente as estruturas de dados, os métodos e os algoritmos utilizados;
     ii. Gerenciador de erros sintáticos:
        1. Apresente claramente o método de gerenciamento de erros escolhido;
        2. Apresente as estruturas de dados, os métodos e os algoritmos utilizados;
     iii. Árvore de sintaxe abstrata:
        1. Mostre em detalhes como a **árvore de sintaxe abstrata** é construída.
3) Resultados experimentais
   - Listagem dos testes executados: deve conter os arquivos de entrada compilados pelo programa (arquivos texto .cmm com código C--) com os **respectivos resultados de saída**.
   - **Comente** os testes executados;
   - **ATENÇÃO**: procure fazer testes relevantes levando em conta as diferentes construções da linguagem. Prove que seu compilador é capaz de realizar as tarefas especificadas abaixo.
4) Conclusão
   - Esse é o lugar em que vocês podem me comunicar **tudo** o que vocês acham que é importante de ser dito;
   - Descreva todas as características especiais do seu compilador nessa seção.
   - Indique extensões e funcionalidades extras.
5) Referências bibliográficas
   - Mostre quais as fontes que vocês consultaram para realizar **cada etapa do trabalho**. Inclua quaisquer artigos, livros, citações, etc., consultados. No caso dos livros do curso, indique quais as seções lidas/usadas para cada tarefa.
6) Listagem do programa fonte
   - O código deve ser claro, comentado, indentado e **sem linhas quebradas**. Use uma fonte isométrica (todos os caracteres têm a mesma largura e altura). Ex. *Courier*. A impressão deve ser legível.
   - Forneçam uma listagem organizada e compacta. Evite longos trechos de código dentro de um mesmo bloco.
   - Os programas fonte devem ser entregues prontos para compilar, sem erros nem avisos, com *makefile*, projeto CodeBlocks, projeto CodeLite, e descompactados. Na apresentação, leve um backup em *pendrive*. Não conte com o uso da rede. **ATENÇÃO**: o código deve rodar em ambiente Linux e Windows (MinGW).

Especificações mínimas para o analisador sintático:

- Esse trabalho é realizável somente através da leitura dos livros. Não reinvente a roda. Seja objetivo e desenvolva como está na literatura, observando rigorosamente as especificações a seguir;
- Abuse ostensivamente de comentários esclarecedores sobre e em **todo** o código do compilador;
- A saída do programa **deve** ser enviada **exclusivamente para stdout**. Não crie nenhum arquivo de saída;
- As mensagens de erro **devem** ser enviadas **exclusivamente para stderr**. Não crie nenhum arquivo de mensagens de erro.
- A precedência dos operadores C--, da menor para a maior e que deve ser resolvida na gramática LL(1), é:
  1) Atribuição: =
  2) Or booleano: ||
  3) And booleano: &&
  4) Igual e diferente: == !=
  5) Outros relacionais: < <= >= >

6)	Adição, subtração e Or bit a bit:	+	-	|	
7)	Multiplicação, divisão e And bit a bit:	*	/	%	&
8)	Unários:	!	+	-	
9)	Parênteses:	(	)		

- Ao converter a GLC C-- para LL(1) ou LR(1), estejam certos de que vocês **não estão mudando a linguagem que é aceita pela gramática**. Não se preocupem com nenhuma regra sensível ao contexto. **Outras restrições à linguagem serão impostas na próxima parte do projeto**. De fato, há uma parte da GLC C-- que não pode ser convertida para LL(1) nem LR(1). O problema ocorre em algumas formas de expressões envolvendo o operador de atribuição. A dificuldade está em determinar se a expressão no lado esquerdo é um ***L-value*** no momento da derivação. Por exemplo, as seguintes expressões são válidas em C--:

```
a[i];                   // a[i] não é um L-value aqui.
a[i] = x;               // a[i] é um L-value aqui.
x = 8 + (y = 6);        // x e y são L-value aqui e (y = 6) é uma
                        // expressão que resulta no R-value 6.
```

  mas as seguintes não são válidas em C--:

```
a[i] + b[i] = x;        // (a[i] + b[i]) não é um L-value válido.
x = 8 + y = 6;          // (8 + y) não é um L-value.
                        // + tem precedência sobre =
```

  A manipulação correta desses casos requer mais do que um *token* de *look-ahead* ou um ou mais *tokens* do tipo *look-behind*. Por agora, a solução é desprezar o problema e traduzir a gramática C-- em uma gramática LL(1) ou LR(1) **quase** equivalente. Ou seja, a gramática será equivalente à linguagem C, com exceção de que aceitará strings do tipo a[i] + b[i] = *expr*. Portanto, a árvore de sintaxe abstrata aceita a atribuição de uma expressão a outra expressão. Na próxima parte do projeto, ao realizar a análise semântica e a geração de código, o compilador deverá verificar se a expressão no lado esquerdo de uma atribuição é um *L-value*.

- A conversão da gramática C-- para LL(1) ou LR(1) será a tarefa mais importante dessa parte do projeto. Mesmo verificando os conjuntos FIRST e FOLLOW, é recomendável testar a sua gramática manualmente para verificar se suas produções derivam corretamente alguns programas C-- bem simples. Por exemplo:

```
int    a;
float  b;
int calcula(int k, int h) {
       println(k, " foram calculados");
}
int main() {
       x = y + (u*7);
       soma(calcula(k,7,5,a));
}
```

- Uma vez verificado que a sua gramática é LL(1) ou LR(1) e deriva a mesma linguagem da GLC C--, então implemente o parser. Se sua escolha é por um **parser recursivo descendente,** gere-o a partir de uma gramática LL(1). Se você escolher uma estratégia bottom-up com uma gramática LR(1), será necessário utilizar um gerador de parsers. Neste caso, as decisões de implementação terão restrições impostas pelo gerador. Tenha certeza de que o grupo consegue realizar todas as tarefas neste contexto.

Especificações mínimas para o gerenciador de erros sintáticos:

- É preciso garantir que o *parser* pode derivar todas as construções da linguagem C--, e suas combinações, **antes de adicionar o código de recuperação de erros**;
- Após ter certeza de que o parser funciona, você deve implementar uma recuperação de erros com um esquema de ***phrase-level*** simplificado (para um único *token* faltante ou incorreto). Não é necessário apresentar esquemas mais complexos de recuperação de erros. Encontre casos simples de recuperação a partir de um único *token* esperado, mas não encontrado. Use no mínimo o conjunto FOLLOW para sincronizar o *parser*. Quando o *parser* encontrar um erro, ele deve imprimir uma mensagem (em **stderr**), tentar se recuperar e continuar o *parsing.* Por exemplo, após detectar o seguinte erro:

  ID LPARENT ID SEMICOLON

  o *parser* deve imprimir uma mensagem como "*Linha 34: erro sintático. Falta fecha parênteses na chamada do procedimento.*" e **continuar o *parsing*** como se o parênteses estivesse no local esperado. Emita mensagens com significado para **stderr**, contendo o número da linha na qual ocorreu o erro. Erros léxicos devem gerar mensagens de erro, mas não devem terminar o *parsing*. Busque o próximo *token* válido e o retorne;

Especificações mínimas para o gerador de código intermediário:

- Somente adicione o código para a construção da árvore de sintaxe abstrata após: 1) implementar todo o *parser,* no caso de construir um parser recursivo descendente; 2) ou gerar e entender como funciona a interface do gerador de parsers para inserção de ações.
- A função para o primeiro não-terminal da gramática (Program) **deverá retornar um ponteiro para o nodo raiz da árvore de sintaxe abstrata PROGRAM** indicado na tabela a seguir. Essa é a única interface do analisador sintático com o compilador.
- Implemente as estruturas e a construção da **árvore de sintaxe abstrata** de forma independente do *parser*. Esse apenas deve conter o código necessário para montar a árvore. **ATENÇÃO: apenas construa a árvore. NÃO verifique nada da linguagem. Isso é tarefa para o analisador semântico.**
- Insira ações no *parser* para criar uma árvore com uma **estrutura equivalente** à dos nodos a seguir:

	Nodo	Formas possíveis
1	PROGRAM	- (FUNCTION_LIST, TYPE_LIST, VAR_LIST)
2	VAR_LIST	- (NAME_DECL, VAR_LIST) - NULL
3	NAME_DECL	- (TYPE, ID)
4	FUNCTION_LIST	- (TYPE, ID, VAR_LIST, VAR_LIST, STATEMENT_LIST, FUNCTION_LIST) - NULL
5	TYPE_LIST	- (VAR_LIST, ID, TYPE_LIST) - NULL
6	TYPE	- (ID, SIZE) - (PRIMITIVE, SIZE) - POINTER
7	POINTER	- TYPE
8	STATEMENT_LIST	- (STATEMENT, STATEMENT_LIST) - NULL
9	STATEMENT	- (IF) - (WHILE) - (SWITCH) - (BREAK) - (PRINTLN) - (READ) - (RETURN) - (THROW) - (STATEMENT_LIST) - (CALL) - (TRY) - (EXP)
10	IF	- (EXP, STATEMENT, STATEMENT)
11	WHILE	- (EXP, STATEMENT)
12	SWITCH	- (EXP, CASEBLOCK)
13	BREAK	- NULL
14	PRINTLN	- (EXP_LIST)
15	READ	- (EXP)
16	RETURN	- (EXP)
17	CASEBLOCK	- (NUM, STATEMENT_LIST, CASEBLOCK) - NULL
18	THROW	- NULL
19	EXP_LIST	- (EXP, EXP_LIST) - NULL
20	TRY	- (STATEMENT, STATEMENT)
21	EXP	- (ID) - (NUMBER) - (LITERAL) - (CHAR) - (CALL) - (NAME_EXP) - (POINTERVALUE_EXP) - (ADDRESSVALUE) - (POINTERVALUE) - (ARRAY) - (ASSIGN) - (RELATIONAL_OP) - (ADDITION_OP) - (MULTIPLICATION_OP) - (BOOLEAN_OP) - (BITWISE_OP) - (NOT) - (SIGN) - (TRUE) - (FALSE)
22	ASSIGN	- (EXP, EXP)
23	NAME_EXP	- (EXP, ID)
24	POINTERVALUE_EXP	- (EXP, ID)
25	ADDRESSVALUE	- (EXP)
26	POINTERVALUE	- (EXP)
27	ARRAY	- (EXP, EXP_LIST)
28	CALL	- (ID, EXP_LIST)
29	RELATIONAL_OP	- (OP_REL, EXP, EXP)
30	ADDITION_OP	- (OP_ADD, EXP, EXP)
31	MULTIPLICATION_OP	- (OP_MUL, EXP, EXP)
32	BOOLEAN_OP	- (OP_BOOL, EXP, EXP)
33	BITWISE_OP	- (OP_BIT, EXP, EXP)
34	TRUE	- NULL
35	FALSE	- NULL
36	NOT	- (EXP)
37	SIGN	- (EXP)

- Procure adicionar ao *parser* um código claro e simples para a construção e encadeamento dos nodos acima. Escolha qualquer estrutura de dados para implementar a árvore. Mas lembre-se que a análise semântica deverá percorrê-la eficientemente, encontrando padrões sintáticos. Esse módulo poderá ser desenvolvido em C++ de forma orientada a objetos (use somente construções permitidas em Java). Os construtores a seguir **exemplificam** uma implementação em Java:

  package ArvoreSintaxe;

  Program((FunctionList fl, VarList att, TypeList tl);

VarList(NameDecl nd, VarList next);
NameDecl(Type t, Identifier id);

FunctionList(Type t, Identifier id, VarList formal, VarList localvar, StmtList sl, FunctionList next);
StmtList(Statement s, StmtList next);
ExpList(Exp e, Explist next);
...

abstract class Statement;

// extends Statement
If(Exp e, Statement s1, Statement s2);
While(Exp e, Statement s1);
...

abstract class Exp;

// extends Exp
Assign(Exp v, Exp e);
Call(Identifier id, ExpList el);
Identifier(String s);
Name(Exp base, Identifier id);
Number(float n);
Number(int n);
Literal (String literal);
Addition_Op(int op, Exp e1, Exp e2);
Relational_Op(int op, Exp e1, Exp e2);
Not(Exp e);

- EXEMPLO: o código Java para montar a árvore da sentença **if (a < b) m(); else c.g = k + 1;** seria algo como:

Exp l = new Relational_Op (LESSTHAN, new Identifier("a"), new Identifier("b"));
Exp a = new Addition_Op(PLUS, new Identifier("k"), new Number(1));
Exp m = new Identifier("m");
Exp g = new Name(new Name(NULL, Identifier("c")), new Identifier("g"));

Statement s1 = new Call(m, NULL);
Statement s2 = new Assign(g, a);

Statement s = new If(l, s1, s2)

Entrada e saída do compilador:

- Você deve imprimir **TODO** não-terminal de sua gramática, seja LL(1) ou LR(1), que estiver sendo processado e **TODO** terminal que for casado com o *token* de entrada. Por exemplo, dado o seguinte programa:

```
int main() {
   float i;
   x = 5 – h;
}
```

O *parser* deve mostrar como saída algo como o seguinte (usando a GLC C--):

Program
FunctionDecl
Type
MATCH: INT
Pointer
MATCH: ID.main
MATCH: OPENPAR
FomalList
MATCH: CLOSEPAR
MATCH: OPENBRA
VarDecl
Type
MATCH: FLOAT
IdList
Pointer
MATCH: ID.i
Array
MATCH: SEMICOLON
VarDecl
StmtList
Stmt
Expr
Expr
Primary
MATCH: ID.x
MATCH: ASSIGN

Expr
Expr
Primary
MATCH: NUMINT.5
BinOp
MATCH: MINUS
Expr
Primary
MATCH: ID.h
MATCH: SEMICOLON
MATCH: CLOSEBRA
MATCH: EOF

- Crie uma interface com o padrão de programação Visitante (*Visitor*) para gerar uma impressão **hierárquica** dos nodos da árvore. Por exemplo, a sentença **if (a < b) m(); else c.g = k + 1;** poderia resultar em:

```
-IF
    -LESSTHAN
      -ID.a
      -ID.b
    -CALL
      -ID.m
    -ASSIGN
      -NAME
          -NAME
                - ID.c
          - ID.g
      -ADD_OP
          -ID.k
         -NUMBER.1
```

Prazo e avaliação:

Lembre-se que este projeto é o principal elemento de avaliação do curso. Uma documentação de alta qualidade técnica, formal e de escrita é importante. O código fonte deve estar correto, muito bem documentado (mas muito mesmo) e deve ser eficiente.

Todos os grupos não têm permissão de possuir ou manter consigo o código ou os resultados de outro grupo. É aconselhável que os grupos mantenham sigilo sobre suas soluções. Qualquer indício de cópia será considerado fato grave contra todos os grupos envolvidos, e a pena será a anulação de TODAS AS NOTAS DE TRABALHO no semestre.

A **avaliação do grupo** depende de:

- Atendimento ao que foi solicitado;
- Resultados práticos;
- Qualidade da documentação;
- Qualidade do código;
- Pontualidade na entrega;
- **Apresentação de no mínimo 30 minutos** demonstrando as funcionalidades do compilador;
- Criatividade na resolução dos problemas;
- Criatividade em ir além do mínimo (**mas somente após alcançar o mínimo**).

A **avaliação individual** depende de:

- Peso dado a cada um dos participantes como consenso do próprio grupo. A soma dos pesos deve ser um;
- **Presença e participação em todas as aulas e durante toda a aula**;
- Destaque no grupo. Ter peso maior do que os outros participantes;
- Destaque na apresentação do compilador.

A **avaliação da classe** depende de:

- Diferença entre a maior e a menor nota dos grupos;
- Diferença entre a maior e a menor nota individual;
- Apresentações dos grupos.

A apresentação deverá ser feita no momento da entrega da documentação.

PRAZOS:

O prazo máximo para a entrega e a apresentação é dia **22 de novembro de 2022.** Nesta semana, todos os alunos devem participar da aula. Além disso, se houver aula na semana da apresentação do grupo, todos os alunos do grupo deverão participar. Caso contrário a apresentação combinada será cancelada e novo horário deverá ser marcado, com a perda de **2 pontos** por apresentação marcada.

Penalidade por dia de atraso (corridos): 3 pontos.

Funcionalidades opcionais:

Não adicione funcionalidades opcionais antes que a solução básica esteja completa e correta. Uma implementação incompleta das funcionalidades obrigatórias mais adicionais resultará em uma nota **menor** do que uma implementação completa sem nenhuma característica extra. A seguir, há alguns exemplos de adicionais que vocês podem considerar: laços do tipo **for**;

Gramática C--:

```
Program      -->   FunctionDecl Program      |
                   TypeDecl Program          |
                   VarDecl Program           |
                   FunctionDecl

TypeDecl     -->   typedef struct { Type IdList ; VarDecl } ID ; TypeDecl | epsilon

VarDecl      -->   Type IdList ; VarDecl     |     epsilon

IdList       -->   Pointer ID Array          |
                   IdList , Pointer ID Array |

Pointer      -->   *      |     epsilon

Array        -->   [  NUM  ] Array   |    epsilon

FunctionDecl -->   Type Pointer ID ( FormalList ) { VarDecl StmtList }

FormalList   -->   Type Pointer ID Array FormalRest | epsilon

FormalRest   -->   , Type Pointer ID Array FormalRest | epsilon

Type         -->   long  |  int  |  float |  bool  |  ID  |  char  |  double

StmtList     -->   Stmt          |
                   Stmt StmtList

Stmt         -->   if ( Expr ) Stmt else Stmt       |
                   while ( Expr ) Stmt              |
                   switch( Expr ) { CaseBlock }     |
                   break ;                          |
                   print ( ExprList ) ;             |
                   readln ( Expr ) ;                |
                   return Expr ;                    |
                   throw ;                          |
                   { StmtList }                     |
                   ID ( ExprList ) ;                |
                   try Stmt catch ( "…" ) Stmt      |
                   Expr  ;                          |

CaseBlock    -->   case  NUM ":" StmtList CaseBlock|
                   case  NUM ":" CaseBlock

ExprList     -->   epsilon      |
                   ExprListTail

ExprListTail -->   Expr         |
                   Expr , ExprListTail

Expr         -->   Primary            |
                   UnaryOp Expr       |
                   Expr BinOp Expr    |
                   Expr = Expr


Primary      -->   ID | NUM | LITERAL  |
                   ´ ASCII ´          |
                   ( Expr )           |
                   Expr "."   ID      |
                   Expr "->"  ID      |
                   ID ( ExprList )    |
                   Expr [ Expr ]      |
                   "&"  Expr          |
                   "*"  Expr          |
                   true | false

UnaryOp      -->   "-"  |  "!" | "+"

BinOp        -->   "==" | "<" | "<=" | ">=" | ">" | "!=" | "+"  |
                   "-"  | "|" | "*"  | "/"  | "%" | "&"  | "&&" | "||"
```