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 – 3ª PARTE

Projete e implemente o analisador semântico para a linguagem C-- e o gerador de código intermediário:

1) Use a solução da parte II do projeto como ponto inicial. **Corrija todos os erros detectados na análise sintática.** Eles podem se propagar para esta fase e criar ainda mais problemas.
2) Crie **uma ou mais classes *visitantes*** da árvore de sintaxe abstrata que realizem a verificação de: registros, tipos, comandos, chamadas de funções, seletores, alternativas, laços, atribuições, expressões e uso de identificadores.
3) Amplie o gerenciador de erros para emitir mensagens e tratar os erros semânticos.
4) Crie **uma classe visitante** para traduzir a árvore de sintaxe abstrata em uma representação intermediária **para cada procedimento**. Essa representação é na verdade o código a ser executado por uma máquina abstrata.
5) Atenda a **todas** as especificações mínimas a seguir. O resultado esperado desta etapa é um compilador capaz de aceitar um programa fonte C-- correto e traduzi-lo em uma representação intermediária.

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 sucinta sobre o desenvolvimento do trabalho. Explique as decisões de implementação tomadas.
   - Descrição dos módulos e suas interdependências. Apresente uma breve descrição de cada módulo bem como um diagrama, por exemplo, mostrando suas inter-relações;
3) Analisador semântico:
   - Liste as regras semânticas para a avaliação sensível ao contexto sobre a GLC C--;
   - Apresente as estruturas de dados, os métodos e os algoritmos utilizados;
   - Tabela de símbolos:
     i. Descreva e justifique a forma com a qual vocês modificaram a tabela de símbolos.
     ii. Apresente as estruturas de dados, os métodos e os algoritmos utilizados;
   - Gerenciador de erros semânticos:
     i. Apresente claramente o método de gerenciamento de erros escolhido;
     ii. Apresente as estruturas de dados, os métodos e os algoritmos utilizados;
4) Gerador da representação intermediária
   - Comente a estrutura e o funcionamento da classe visitante que traduz uma árvore de sintaxe abstrata em uma representação intermediária hierárquica;
5) 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 a ser implementada. Prove que seu compilador é capaz de realizar as tarefas especificadas abaixo.
6) 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.
7) 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.
8) 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 gerais mínimas para o compilador:

- Este trabalho é realizável somente através da leitura dos livros. Não reinvente a roda. Seja objetivo e desenvolva como está na literatura;
- Crie um módulo separado para quaisquer rotinas especiais que forem criadas para auxiliar a análise semântica;
- Evitem problemas, **COMECEM A TRABALHAR HOJE MESMO E EM GRUPO**. O risco de se deixar para o final do prazo (ou não dividir o trabalho) é muito grande;
- Pode ser que você tenha de modificar partes do seu compilador. Não se esqueça de comentar as suas decisões;

- Não deixem de consultar o livro "*Modern compiler implementation in Java*", capítulos 8 e 9, para fazer este trabalho.
- Crie um módulo exclusivo para todas as rotinas usadas na geração de código;
- Abuse ostensivamente de comentários esclarecedores sobre e em todo o código do compilador;
- As mensagens de erro **devem** ser enviadas **exclusivamente para `stderr`**. Não crie nenhum arquivo de mensagens de erro;
- Concentre-se primeiro na gramática mínima fornecida. Somente depois disso programe as extensões propostas nos projetos anteriores;
- Abuse ostensivamente de comentários esclarecedores sobre e em todo o código do compilador.

Especificações mínimas para o analisador semântico:

- Crie regras sensíveis ao contexto em classes visitantes para verificar a semântica do programa fonte C-- na forma de uma árvore de sintaxe abstrata. Adicione novos campos à tabela de símbolos sempre que necessário;
- Use exatamente as mesmas regras da linguagem C para gerenciar e verificar:
    - **O escopo de identificadores**: implemente o aninhamento de escopos. Múltiplas declarações não são permitidas no mesmo escopo (novos identificadores serão inseridos na tabela somente no momento da declaração). Porém, um mesmo identificador pode ser usado em escopos mais internos. No momento do uso de identificadores ao longo de um programa, o compilador deve identificar a declaração mais interna ou, se não houver nenhuma, gerar um erro.
    - **Compatibilidade de tipos em expressões**: os operadores aritméticos e relacionais devem aceitar somente inteiros e reais. Os operadores lógicos AND, OR, e NOT devem somente ser usados com operandos do tipo lógico. Todos os operadores binários (aritméticos, relacionais ou lógicos) devem ser usados com dois operandos de mesmo tipo.
    - **Acesso correto a campos de classes/registros:** os campos de registro devem ser corretamente acessados. Combinações arbitrárias e hierárquicas de registros e arranjos devem ser adequadamente encadeadas.
    - **Acesso correto a atributos e métodos de classe:** os atributos de classe e de métodos devem ser corretamente acessados. Classes estendidas devem ser adequadamente encadeadas de forma a permitir acessos de atributos herdados. Ponteiros do tipo *this* devem ser corretamente utilizados.
    - **Compatibilidade de tipos em atribuições**: a expressão do lado direito deve ter o mesmo tipo do identificador do lado esquerdo.
    - **Compatibilidade de tipos na passagem de parâmetros**: o tipo das expressões usadas na chamada de funções e procedimentos deve casar com a declaração. É necessário criar regras para a passagem de classes herdadas;
    - **Compatibilidade de tipos no retorno de funções**;
    - **Expressões inteiras no acesso a elementos de arranjos**: além disso deve-se verificar se índices inteiros constantes pertencem ao intervalo declarado do *vetor*;
    - **Uso indevido de identificadores de *vetor* em expressões e passagem de parâmetros**: deve-se permitir a passagem de arranjos como parâmetro (escolha as restrições para isso), mas sem cópia de valores. O compilador não deve aceitar arranjos (sem índice) como operandos em expressões nem em atribuições;
    - **Chamadas de funções**;
    - **Controle e uso correto de *L-values* e *R-values* em atribuições e passagem de parâmetros**;
    - **Uso incorreto de literais em expressões numéricas**.
    - Etc.
- Na análise de expressões, operações cujos operandos são de tipos diferentes devem resultar em erro semântico por incompatibilidade.
- O compilador deve ser capaz de manipular expressões como:

```
x = x + z * 6;
arranjo[i].c = arranjo[j].c + f(arranjo, x, y + 8);
```

- Ao longo das verificações, você deve coletar e calcular as informações necessárias para montar o registro de ativação (*frame*) de cada procedimento e função. Podem-se guardar essas informações em tabelas de símbolos ou na árvore de sintaxe abstrata.

Especificações mínimas do tradutor para o código intermediário:

- Com o objetivo de distinguir os dois elementos básicos que devem ter seu espaço alocado pelo compilador, crie uma classe abstrata **fragment** (com um campo ponteiro para o próximo fragmento) a ser estendida pelas classes:
    - `procedure(Frame *frame, Stm *body)`: determina o *frame* e o corpo de um procedimento (qualquer procedimento ou função declarado **e o programa principal**).
        - `frame`: objeto com as informações necessárias para a **ativação** do procedimento. Deve conter: a localização de cada parâmetro (no frame ou em um registrador), número de variáveis locais alocados até o momento, o rótulo no qual o código de máquina do procedimento começa, e as instruções necessárias para realizar o *deslocamento do frame*.
        - `body`: contém o código intermediário relativo ao procedimento (sem prólogo e epílogo para a ativação).
    - `literal(char*)`: uma *string* literal deve ter seu espaço alocado pelo compilador. O acesso a ela deverá ser feito através de um rótulo que será gerado automaticamente na construção desse objeto. Constantes inteiras e reais serão representadas diretamente no código alvo.
    - Se a linguagem permite variáveis não-locais alocadas estaticamente
      `variable(Type t, int nbytes)`: O parâmetro `nbytes` indica o tamanho em bytes do tipo que pode ser primitivo ou composto. O acesso a dado deverá ser feito através de um rótulo cujo nome é exatamente o da variável correspondente no código fonte.

- Defina um tradutor (ex.: classe formada por visitantes) que receba uma **árvore de sintaxe abstrata** (objeto `Program*` correto) e gere uma **lista de fragmentos** que formam sua representação intermediária. Ele deve retornar o primeiro fragmento desta lista encadeada de fragmentos.

- A geração de código **final** terá de escolher registradores para os parâmetros e para as variáveis locais. Deve também definir os endereços no código texto para os corpos dos procedimentos. **Mas ainda é muito cedo** para determinar exatamente quais registradores estão disponíveis ou onde exatamente o corpo de uma função estará localizado. O termo ***temporário*** (`Temp`) indica um valor que será temporariamente mantido em um registrador e o termo ***rótulo*** (`Label`) indica algum local em linguagem de máquina cujo endereço exato ainda deverá ser determinado.
  - `Temp` é um nome abstrato para variáveis locais. Exemplo em Java:

```
public class Temp {
       public String toString(); // Ex. temp$196, temp$197, …
       public Temp();
       public Temp(String s); //Ex. eax, fp, $21, … (registradores)
}
public class TempList { TempList(Temp *t, TempList *prox); }
```

  - `Label` é um nome abstrato para endereços estáticos de memória (dados ou código). Exemplo em Java:

```
public class Label {
       public String toString();
       public Label();              // Ex. Label$432, label$433, ...
       public Label(String s);      // Ex. calcula, quicksort, etc.
}
public class LabelList { LabelList(Label *l, LabelList *prox); }
```

  - Um `new Temp()` deve retornar um novo temporário de um conjunto infinito de temporários. Um `new Temp("r")` deve retornar um temporário associado ao registrador **r** específico da máquina. Um `new Label()` deve retornar um novo rótulo de um conjunto infinito de rótulos. Um `new Label(s)` deve retornar um novo rótulo cujo nome em linguagem *assembly* é `s`.

- A classe `Frame` deve ser abstrata porque seu *layout* e sua estrutura variam de máquina para máquina. A classe `acessoLocal` indica como um dado local é acessado. Ela deve gerar a seqüência de instruções necessárias para acessar o nome. Cada parâmetro e variável local deve ter seu objeto `acessoLocal` alocado no frame.

```
public abstract class acessoLocal {  // define como será o acesso a um dado local
     abstract public Stm codigoAcesso(); // retorna o código de máquina p/ acessar o nome
}

public class ListaAcesso { ListaAcesso(acessoLocal *l, ListaAcesso *prox); }

public abstract class Frame {
       // o frame decide se o dado local estará em um registrador ou em um temporário.
       abstract public acessoLocal adicionaParam(boolean escapa, int tambytes);
       abstract public acessoLocal adicionaLocal(boolean escapa, int tambytes);
}
```

- Nesta etapa, crie *frames* para o microprocessador MIPS. Um dado local poderá estar em um registrador ou no frame:

```
class inFrame extends acessoLocal {int offset; …} // p/ alocação no frame
class inReg   extends acessoLocal {Temp temp; …}  // p/ alocação no registrador

Temp FP("fp"); // Temp único que representa o registrador FP (ponteiro do frame)

class FrameMIPS extends Frame {
       Label       rotulo;     // Rótulo para o inicio do código do procedimento (nome)
       Temp        valoretorno;// Temporario que contém o valor de retorno da função
       ListaAcesso dadoslocais;// Lista de acessos locais (parâmetros e variáveis locais)

       // essas funções decidem se o dado local estará em um registrador ou no frame.
       // uma variável escapa, ela tem que ser colocada no frame.
       abstract public acessoLocal adicionaParam(boolean escapa, int tambytes);
       abstract public acessoLocal adicionaLocal(boolean escapa, int tambytes);
}
```

- O método abstrato `codigoAcesso()` da classe `acessaLocal` deve retornar o trecho de código (instruções da máquina abstrata) que permite acessar o dado local. **Exemplo**: um `new inFrame(o)` indica um local de memória no offset **o** a partir do FP (*frame pointer*). Um `new inReg(t098)` indica que o dado será mantido no registrador de máquina temp$098 (nome do temporário). O objeto inFrame(8) deve gerar as instruções de acesso MEM(BINOP(+, TEMP(FP), CONST(8))) e o objeto inReg(t098) deve gerar as instruções de acesso TEMP(t098).

- Para cada procedimento, gere o código intermediário (árvore) para a máquina abstrata com as seguintes instruções (nodos):
  - **Exp**: operadores de expressões para o cálculo de algum valor:
    - **CONST**(*i*) a constante inteira ***i***;
    - **CONSTF**(*j*) a constante real ***j***;
    - **NAME**(*n*) a constante ***n*** (objeto `Label`) corresponde a um rótulo na linguagem *assembly*;
    - **TEMP**(*t*) o temporário ***t***. Um temporário na máquina abstrata é similar a um registrador em uma máquina real. Entretanto, a máquina abstrata tem um número infinito de registradores;
    - **BINOP**(*o*, *e1*, *e2*) a aplicação do operador ***o*** nos operandos ***e1*** e ***e2***. A subexpressão ***e1*** é avaliada antes de ***e2***. Os operadores aritméticos são PLUS, MINUS, MUL, DIV; os operadores lógicos inteiros bit a bit são AND, OR e XOR; os operadores lógicos de deslocamento bit a bit são LSHIFT e RSHIFT; o operador aritmético de deslocamento à direita é ARSHIFT;
    - **MEM**(*e*) o conteúdo de *wordsize* bytes de memória começando no endereço ***e*** (o tamanho da palavra da máquina alvo *wordsize* é definido no módulo frame). Quando MEM é usado como filho esquerdo de um MOVE, ele significa "armazenar". Ao contrário, significa "carregar";
    - **CALL**(*f*, *l*) uma chamada de procedimento. A aplicação da função ***f*** à lista de argumentos ***l***. A subexpressão ***f*** é avaliada antes dos argumentos que são avaliados da esquerda para a direita;

    - **ESEQ**(*s*, *e*) a sentença ***s*** é avaliada para resultados secundários e então ***e*** é avaliada para um resultado.
  - **Stm**: sentenças para resultados secundários e controle do fluxo:
    - **MOVE**(TEMP *t, e*) avalie ***e*** e mova o resultado para o temporário ***t***;
    - **MOVE**(MEM(*e1*), *e2*) avalie ***e1*** para obter o endereço ***a***. Então avalie ***e2*** e armazene o valor nos *wordsize* bytes de memória a partir de ***a***;
    - **EXP**(*e*) avalie ***e*** e descarte o resultado;
    - **JUMP**(*e*, *labs*) transfira o controle para o endereço ***e***. O destino *e* pode ser um rótulo literal NAME(*n*) ou pode ser um endereço calculado por qualquer outro tipo de expressão. Por exemplo, uma sentença `switch(i)` em linguagem C pode ser implementada realizando alguma aritmética em *i*. A lista de rótulos ***labs***, especifica todas as possíveis localidades que a expressão *e* pode resultar. Isso é necessário para uma análise de fluxo de dados. O caso simples de pular para um rótulo conhecido é escrito como JUMP(NAME *n*, new LabelList(*n*, NULL)) mas isso pode ser abreviado com um construtor específico.
    - **CJUMP**(*o*, *e1*, *e2*, *t* ,*f*) avalie ***e1***, ***e2*** nessa ordem, obtendo os valores ***a*** e ***b***. Então compare ***a*** e ***b*** usado o operador relacional ***o***. Se o resultado é TRUE pule para ***t***, senão pule para ***f***. Os operadores relacionais são EQ e NE para igualdade com ou sem sinal; LT, GT, LE, GE para desigualdades *com sinal*; e ULT, ULE, UGT, UGE para desigualdades *inteiras sem sinal*.
    - **SEQ**(*s1, s2*) a sentença ***s1*** seguida por ***s2***;
    - **LABEL**(*n*) define o nome constante ***n*** (objeto **`Label`**) para representar o endereço de código nesse local. Especifica a posição do rótulo no código *assembly*. A expressão NAME(***n***) poderá ser o alvo de JUMP, CJUMP e CALL.

- Um exemplo de declaração em Java da linguagem intermediária é dado a seguir:

```
package IR;

abstract class Exp;

// extends Exp
CONST(int value);
CONSTF(float value);
NAME(Label l);
TEMP(Temp t);
BINOP(int binop, Exp left, Exp right);
MEM(Exp e);
CALL(Exp func, ExpList args);
ESEQ(Stm s, Exp e);

abstract class Stm;

// extends Stm
MOVE(Exp dst, Exp src);
EXP(Exp e);
JUMP(Exp e, LabelList targets);
CJUMP(int relop, Exp left, Exp right, Label iftrue, Label iffalse);
SEQ(Stm left, Stm right);
LABEL(Label l);

// Outras classes
ExpList(Exp prim, ExpList prox);
StmList(Stm prim, StmList prox);

// Constantes para operadores aritméticos e lógicos
final static int PLUS, MINUS, MUL, DIV, AND, OR, LSHIFT, RSHIFT, ARSHIFT, XOR;

// Constantes para operadores relacionais
final static int EQ, NE, LT, GT, LE, GE, ULT, ULE, UGT, UGE;
```

- Quando um campo *x* de uma classe/registro é acessado, deve-se calcular o seu endereço usando o ponteiro base do objeto. Dada a variável `a.b.c.d.e`, a forma geral (instruções) para o acesso é:
  - MEM(BINOP(+, CONST k_e, BINOP(+, CONST k_d, BINOP(+, CONST k_c, BINOP(+, CONST k_b, BINOP(+, CONST k_a, TEMP FP)))))), onde k_a é o offset do objeto **a** dentro no frame atual, o k_b é o *offset* do objeto **b** dentro do objeto **a**, k_c é o *offset* do objeto **c** dentro do objeto **b**, k_d é o *offset* do objeto **d** dentro do objeto **c**, e o k_e é o *offset* do objeto **e** dentro do objeto **d**.

- Se a linguagem permite aninhamento de funções:
  - Os *links* de acesso estático (*access links*) devem ser passados como um parâmetro implícito extra. Ex.: para chamar uma função `f(`a_1`,…,`a_n`)` , a instrução é CALL(NAME(Lf), [sl, e_1, e_2, ..., e_n]), onde **Lf** é o rótulo para *f* , **sl** é o *link* estático. Feito dessa forma, o endereço do link estático da função é sempre encontrado no endereço FP+0.
    - **sl** é calculado pela diferença entre níveis léxicos por acessos na forma MEM(MEM(… MEM(TEMP FP))…)))), assumindo que o *link* estático está no endereço FP+0. **O número de acessos MEM depende do nível léxico da função atual (*caller*) e da função chamada (*callee*).**
  - Quando uma variável *x* é declarada em um nível mais externo do escopo os *links* estáticos devem ser usados. A forma geral (instruções) para o acesso é:
    - Exemplo: se *x* é acessada em uma função de nível léxico 4, mas está declarada em uma função mais externa no nível 2, o acesso é dado por: MEM(BINOP(+, MEM(MEM(TEMP($FP_{nível4}$)) {endereço do *frame* do nível 3}) {endereço do *frame* do nível 2}), $offset_x$) {soma endereço do nível 2 com o offset de *x*}).

- Crie uma classe Visitante (*Visitor*) que transforme uma árvore de código intermediário em sua forma canônica. Transforme a árvore de código intermediário de cada procedimento do programa fonte.

- Para visualização do resultado, crie uma interface com o padrão de programação Visitante (*Visitor*) para gerar uma impressão **hierárquica** dos fragmentos
    - Procedimentos: *frame* (parâmetros, nomes locais, temporários e rótulos) e código intermediário,
    - Constantes literais
    - Se a linguagem permite variáveis não locais: inteiros, reais, booleanos, ponteiros, caracteres, registros e objetos

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.

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 20 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;
- 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 **14 de janeiro de 2023** até o início da aula, se houver.
Penalidade por dia de atraso (corridos): 3 pontos.

DICA: não deixem a apresentação para o último dia.

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`**;