UNIVERSIDADE ESTADUAL DE PONTA GROSSA
DEPARTAMENTO DE INFORMÁTICA
ENGENHARIA DE COMPUTAÇÃO
2ª PARTE DO TRABALHO DA DISCIPLINA DE LFC – 2019 – Parte 2 de 3

ALUNOS: ANDRÉ VIEIRA BERNARDO
DERICSON PABLO CALARI NUNES
GABRIEL HENRIQUE SIMIONI
RAILAN DAL COL NASCIMENTO

1) **Descreva a estrutura da implementação dos códigos do Scanner e do Parser gerados pelo programa Coco/R**.

Scanner:

- O scanner é especificado por uma lista de declarações de token. Literais (por exemplo, "se" ou "while") não precisa ser declarado como tokens, mas pode ser usado diretamente no produções da gramática.
- O scanner é implementado como um autômato finito determinístico (DFA). Assim sendo os símbolos do terminal (ou tokens) devem ser descritos por uma gramática regular.
- Os comentários podem estar aninhados. Pode-se especificar vários tipos de comentários para uma linguagem.
- O scanner suporta caracteres Unicode codificados em UTF-8.
- O scanner pode ser sensível a maiúsculas ou minúsculas.
- O scanner pode reconhecer tokens dependendo do seu contexto no fluxo de entrada.
- O scanner pode lidar com os chamados pragmas, que são tokens que não fazem parte da sintaxe, mas pode ocorrer em qualquer lugar do fluxo de entrada (por exemplo, diretivas do compilador ou caracteres de fim de linha).
- O usuário pode suprimir a geração de um scanner e fornecer um documento manuscrito de scanner em vez disso.

Parser:

- O parser é especificado por um conjunto de produções EBNF com atributos e ações semânticas. As produções permitem alternativas, repetição e partes opcionais. O Coco / R converte as produções em um parser recursivo eficiente. O parser permite várias instâncias dele estarem ativas ao mesmo tempo.
- Os símbolos não terminais podem ter qualquer número de atributos de entrada e saída. Os símbolos do terminal não possuem atributos explícitos, mas os tokens retornados pelo scanner contêm informações que podem ser visualizadas como atributos. Todos os atributos são avaliados durante a análise.
- Ações semânticas podem ser colocadas em qualquer lugar da gramática (não apenas no final do produções). Elas podem conter declarações ou declarações arbitrárias escritas na linguagem do parser gerado.
- O símbolo especial ANY pode ser usado para indicar um conjunto de tokens complementares.

- Toda produção pode ter suas próprias variáveis locais. Além desses, pode-se declarar variáveis ou métodos globais, traduzidos em campos e métodos de um parser. As ações semânticas também podem acessar outros objetos ou métodos das classes escritas pelo usuário ou das classes da biblioteca.
- As mensagens de erro impressas pelo analisador gerado podem ser configuradas de acordo com um formato específico do usuário.
- O parser e o scanner gerados podem ser especificados para pertencer a um determinado namespace (ou pacote).

2) O que é erro léxico? Descreva uma estratégia para tratar erros léxicos

Um erro léxico é um trecho do código que não poderá ser identificado pelo analisador léxico. Por exemplo em C, o trecho, int x = 8d, é um erro léxico, pois o analisador não conseguirá transformar 8d em um inteiro válido.

A análise léxica é muito prematura para identificar alguns erros de compilação, como mostrado neste exemplo abaixo:

fi (a == “123”)

O analisador léxico não identifica o erro desta instrução, pois ele não consegue identificar que em determinada posição deve ser declarado a palavra reservada if ao invés de fi. Essa verificação só é possível ser feita na análise sintática.

Porém é importante ressaltar que o compilador deve continuar o processo de compilação a fim de encontrar o maior número de erros possível.

Uma situação comum de erro léxico é a presença de caracteres que não pertencem a nenhum padrão conhecido da linguagem, como por exemplo o caractere ¢. Nesse caso o analisador léxico deve sinalizar um erro informando a posição desse caractere.

Um possível tratamento para esse tipo de erro é o chamado de Modo de Pânico onde é removido os caracteres seguintes até encontrar um token bem formado.

3) O que é erro sintático? Descreva um erro sintático e uma estratégia para tratá-lo.

Em muitos compiladores, ao encontrar uma construção mal formada o erro é reportado e a tarefa da Análise Sintática é dada como concluída mas na prática o compilador pode e até deve reportar o erro e tentar continuar a Análise Sintática para detectar outros erros, se houver, e assim diminuir a necessidade de recomeçar a compilação a cada relato de erro. A realização efetiva do tratamento de erros pode ser uma tarefa difícil

O tratamento inadequado de erros pode introduzir uma avalanche de erros espúrios, que não foram cometidos pelo programador, mas pelo tratamento de erros realizado. Três processos, geralmente, são realizados nesse ponto:

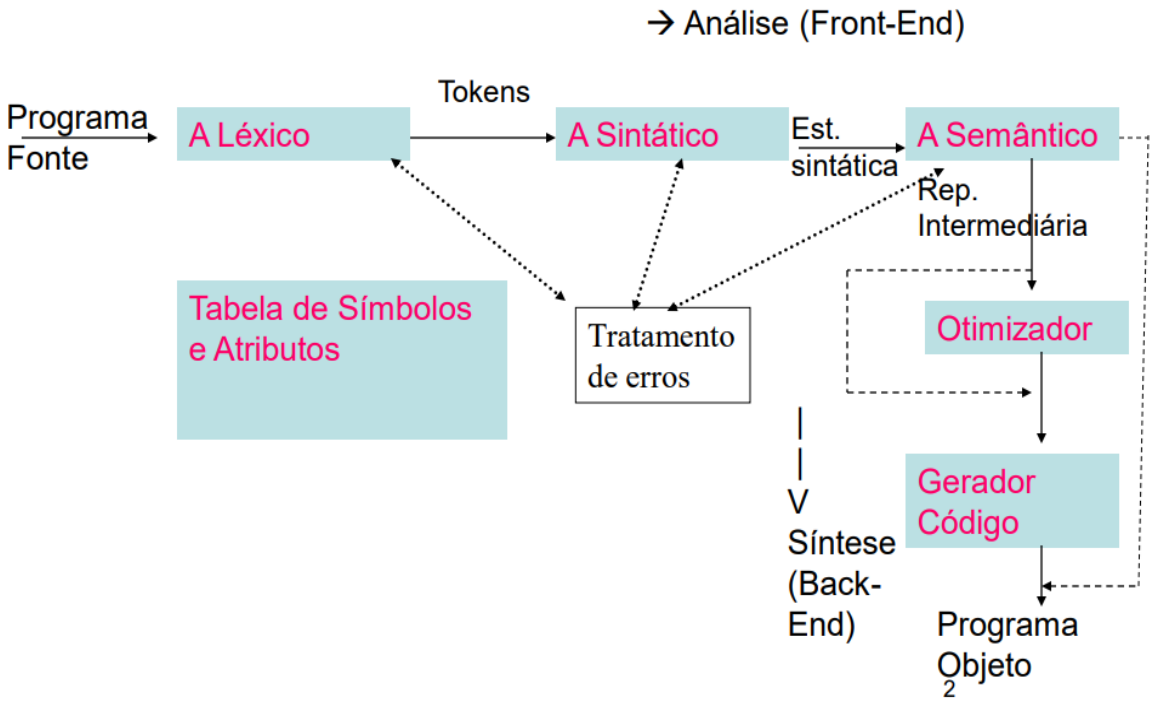
1. Notificação
2. Recuperação (modo de pânico): pula-se parte subseqüente da entrada até encontrar um token de sincronização (porto seguro para continuar a análise)
3. Reparo (recuperação local): faz-se algumas suposições sobre a natureza do erro e a intenção do escritor do programa.

Altera-se 1 símbolo apenas: despreza o símbolo, ou substitui ele por outro ou ainda insere um novo token. A opção mais comum é inserir 1 símbolo (comum para ; faltantes).

1. A localização de um erro sintático é notificada
2. Se possível, os tokens que seriam uma continuação válida do programa são impressos
3. Os tokens que podem servir para continuar a análise são computados. Uma seqüência mínima de tokens é pulada até que um destes tokens de continuação seja encontrado
4. A localização da recuperação (ponto de recomeço) é notificada
5. a análise pode ser chaveada para o “modo reparo” também. Neste ponto, o analisador se comporta como usual, exceto que nenhum token de entrada é lido. Ao invés, uma seqüência mínima (geralmente um símbolo) é sintetizada para reparar o erro. Os tokens sintetizados são notificados como símbolos inseridos. Depois de sair do modo reparo, a A.S. continua como usual.

4) Quais as tarefas executadas pelo analisador semântico?

As duas principais tarefas do analisador semântico são a análise de contexto com geração de código e verificação de erros em frases sintaticamente corretas.
A saída da fase de análise semântica é anotada na árvore do analisador gramatical.
As gramáticas de atributo são usadas para descrever a semântica de estática de um programa.
A fase de geração de código intermediário permite a geração de instruções para uma máquina abstrata, normalmente em código de três endereços, mais adequadas à fase de otimização. Esta forma intermediária não é executada diretamente pela máquina alvo.
A fase de otimização analisa o código no formato intermediário e tenta melhorá-lo de tal forma que venha a resultar um código de máquina mais rápido em tempo de execução, usando as réguas que denotam a semântica da linguagem-fonte. Uma das tarefas executadas pelo otimizador é a detecção e a eliminação de movimento de dados redundantes e a repetição de operações dentro de um mesmo bloco de programa.
E por fim, a fase de geração de código tem como objetivo analisar o código já otimizado e gerar um código objeto definitivo para uma máquina alvo. Normalmente este código objeto é um código de máquina relocável ou um código de montagem. Nesta etapa as localizações de memória são selecionadas para cada uma das variáveis usadas pelo programa. Então, as instruções intermediárias são, cada uma, traduzidas numa seqüência de instruções de máquina que realizam a mesma tarefa.



5) Gere um arquivo ATG para a linguagem CLC:

P = *begin* L *end*.

L = C ";" L | "showSymbolTable" "*(" ")*" " ;" | "showSymbolTree" "*(" ")*" " ;"

C = **id** ":= "E ";"| "*input*" "(" **id** ")" ";" | "*print*" "(" **id** ")" " ;"

E = T { ("+"|"-") T }.

T = F { ("*"|"/") F }.

F = (**id** |**num** | "(" E ")").

Obs. id e literal são categorias de TOKENS.

- Id : é uma sequência de letras minúsculas
- literal é um número inteiro

6) **Escreva um programa CLC para:**
   • calcular um polinômio de grau 2;

```
begin

input(a);
input(b);
input(c);
input(x);
y := a*x*x + b*x + c;
print(y);
showSymbolTable();
end.
```

   • calcular um polinômio de grau 3;

```
begin
input(a);
input(b);
input(c);
input(d);
input(x);
y := a*x*x*x + b*x*x + c*x + d;
print(y);
showSymbolTable();
end.
```

   • mostrar o resultado dos dois polinômios

   Usando print(y) para mostrar o resultado dos polinômios.

7) **Implemente e teste rotinas para criar e mostrar a tabela de símbolos da linguagem CLC no arquivo ATG.**

TABELA DE SÍMBOLOS DA LINGUAGEM CLC NO ARQUIVO ATG

```
Symbol Table:
------------

 nr name           typ  hasAt graph  del   line tokenKind
  0 EOF            t    false                0 fixedToken
  1 id             t    false               12 classLitToken
  2 num            t    false               13 classToken
  3 "begin"        t    false               18 litToken
  4 "end."         t    false               18 fixedToken
  5 ";"            t    false               19 fixedToken
  6 "showSymbolT   t    false               19 fixedToken
  7 "("            t    false               19 fixedToken
  8 ")"            t    false               19 fixedToken
  9 "showSymbolT   t    false               19 fixedToken
 10 ":="           t    false               20 fixedToken
 11 "input"        t    false               20 litToken
 12 "print"        t    false               20 litToken
 13 "+"            t    false               21 fixedToken
 14 "-"            t    false               21 fixedToken
 15 "*"            t    false               22 fixedToken
 16 "/"            t    false               22 fixedToken
 17 ???            t    false                0 fixedToken
  0 CLC            nt   false     1 false    18 fixedToken
  1 L              nt   false    13 false    19 fixedToken
  2 C              nt   false    30 false    20 fixedToken
  3 E              nt   false    38 false    21 fixedToken
  4 T              nt   false    46 false    22 fixedToken
  5 F              nt   false    56 false    23 fixedToken

Literal Tokens:
--------------
```

TABELA DE SÍMBOLOS DO CÓDIGO DO POLINÔMIO DE 2º GRAU

```
run:
Token lido: begin
Token lido: input
Token lido: input
Token lido: input
Token lido: input
Token lido: a
Token lido: x
Token lido: x
Token lido: x
Token lido: b
Token lido: x
Token lido: x
Token lido: c
Token lido: c
Token lido: c
Token lido: c
Token lido: print
Token lido: end.
0 errors detected
*************
CONSTRUÍDO COM SUCESSO (tempo total: 0 segundos)
```

TABELA DE SÍMBOLOS DO CÓDIGO DO POLINÔMIO DE 3º GRAU

```
run:
Token lido: begin
Token lido: input
Token lido: input
Token lido: input
Token lido: input
Token lido: input
Token lido: a
Token lido: x
Token lido: x
Token lido: x
Token lido: x
Token lido: b
Token lido: x
Token lido: x
Token lido: x
Token lido: c
Token lido: x
Token lido: x
Token lido: d
Token lido: d
Token lido: d
Token lido: d
Token lido: print
Token lido: end.
0 errors detected
************
CONSTRUÍDO COM SUCESSO (tempo total: 0 segundos)
```