Construção de Compiladores
Período Especial
Aula 3: Analisador Sintático

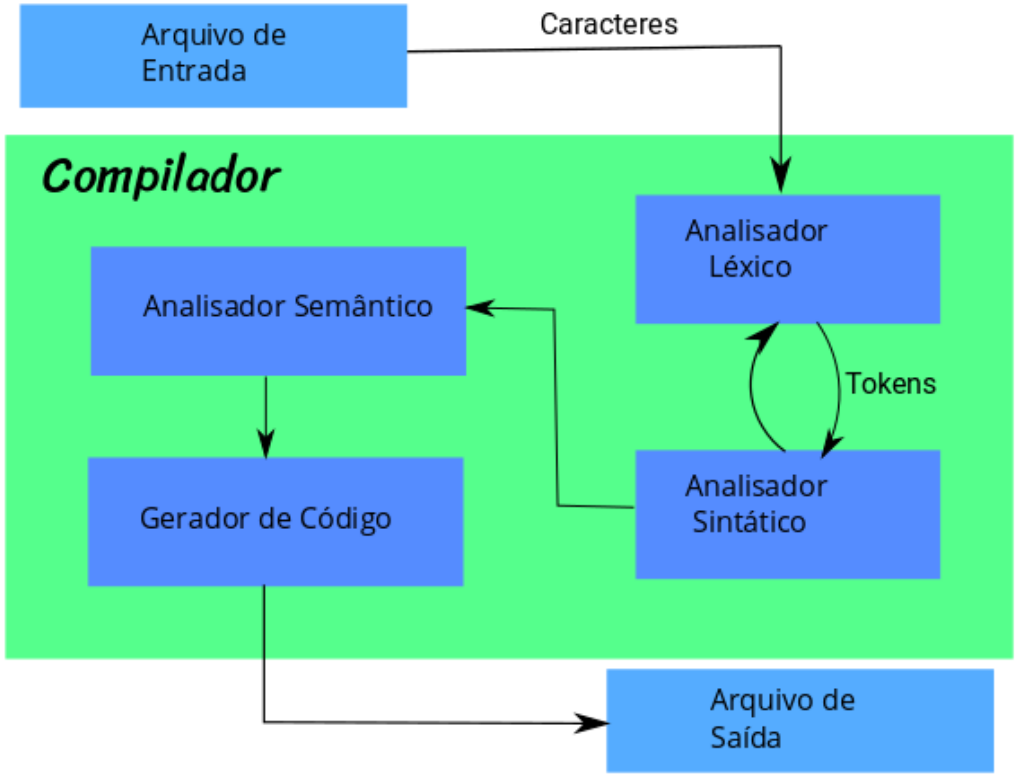
Bruno Müller Junior

Departamento de Informática
UFPR

2020

Compilador

- Partes que Compõe um Compilador



Análise Sintática

- A análise sintática (*parsing*) é um processo que verifica se uma determinada entrada (sentença) corresponde ao de uma gramática.
  - Seja G1 uma gramática;
  - Seja L(G1) a linguagem definida por G1;
  - Seja α uma sentença de entrada.
  - Então, formalmente, um analisador sintático é uma ferramenta capaz de dizer se:

$$\alpha \in L(G1)$$

Gramáticas

- Uma gramática é a formalização de uma determinada linguagem.

```
G2 = {    Sentence    ::= Noun Verb Article Noun
        Conjunction ::= "and" "or" "but"
        Noun        ::= "birds" "fish" "C++" "sky" "sea" "computer
        Verb        ::= "rules" "fly" "swim"
        Article     ::= "the" }
```

- A gramática G2 abaixo permite:
  - Gerar o conjunto de todas as sentenças válidas (ou seja, a linguagem, $L(G2)$).
  - Verificar se uma dada sentença segue corretamente a regra gramatical ($\alpha \in L(G)$).

Hierarquia de Chomsky

- é uma classificação de gramáticas formais descrita em 1959 pelo linguista Noam Chomsky.

Reconhecedores

- Cada linguagem da hierarquia tem um tipo de autômato que é capaz de reconhecê-la.
- Exemplos:
  - Autômato Finito reconhece Ling. Regulares.
  - Autômato a Pilha reconhece LLC.
- A hierarquia de Chomsky não apresenta uma classe importante de linguagems: As linguagens livres de contexto determinísticas (um subconjunto das LLC onde as linguagens não são ambíguas).
- A teoria (e prática) de compiladores trata desta classe.
- Todas as linguagens de programação pertencem a esta classe.

Ambiguidades

- G2 abaixo é uma gramática ambígua pois permite duas árvores de derivação para uma mesma sentença ($\alpha =$ "aaa")

$$G2 \quad = \quad \{A \rightarrow Aa|aA|a\}$$

- G3 não é ambígua.

$$G3 \quad = \quad \{A \rightarrow Aa|a\}$$

- $L(G2) = L(G3)$
- Esta linguagem não é ambígua, mas uma gramática mal escrita pode levar a pensar assim.
- Por esta razão, muitas vezes é possível reescrever gramáticas e retirar a ambiguidade.

Analisadores Sintáticos

- Métodos para se construir a árvore sintática de um α:
  1. "cima para baixo" (*top-down*)
  2. "baixo para cima" (*bottom-up*)
- Existem ferramentas que recebem como entrada uma gramática e geram como saída o A.S. desta gramática. Destaco duas:
  1. *top-down*: javacc;
  2. *bottom-up*: bison;
- É importante frisar que bison e javacc exigem gramáticas em formatos incompatíveis entre si.
- Estas ferramentas incluem mecanismo de executar código do usuário em pontos determinados da árvore sintática. A isto se dá o nome de "Tradução Dirigida pela Sintaxe" (TDS)

Conceituação
TDS

- Considere o problema de transformar uma entrada que está na notação infixa para a notação posfixa. Exemplo:
  $\boxed{"a_1 + a_2 * a_3 - a_4" \Rightarrow "A_1 A_2 A_3 * + A_4 - "}$.
- A entrada obedece a uma gramática (G4 abaixo).

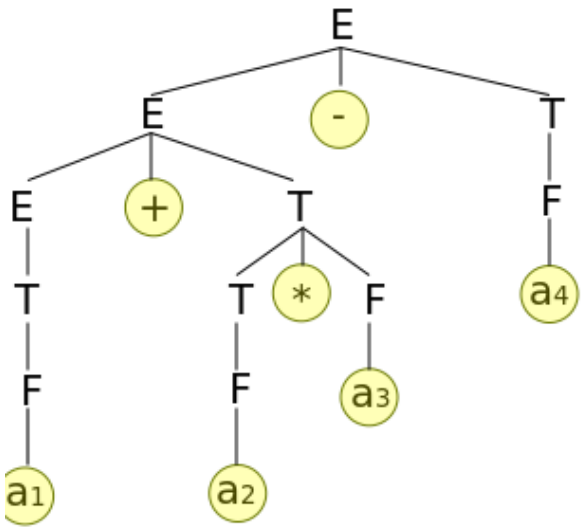
```
G4 = { E ::= E+T |  E-T | T
       T ::= T*F | F
       F ::= a
     }
```

- Desenhe a árvore sintática para a entrada e indique os nós percorridos no caminhamento inorder.

Árvore Sintática

```
G4 = { E ::= E+T |  E-T | T
       T ::= T*F | F
       F ::= a
     }
```

$a_1 + a_2 * a_3 - a_4$



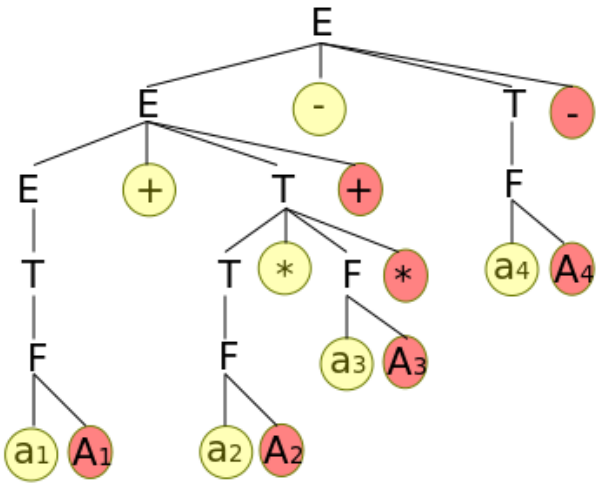
Árvore Sintática

- A idéia é acrescentar nós "executáveis" à árvore. Estes nós são trechos de programa, no nosso caso, C.
- Toda vez que um trecho destes for encontrado, deve-se pendurá-lo na árvore.
- Ao concluir a construção da árvore, faça o caminhamento inorder, executando os nós "executáveis".
- Construa a árvore abaixo para a mesma entrada e caminhe inorder. Ao encontrar um nó executável, execute-o e veja o resultado gerado na saída.

```
G4 = { E ::= E+T {printf ("+");} | T
       T ::= T*F {printf ("*");} | F
       F ::= a { printf ("A"); }
     }
```

Árvore Sintática

```
G4 = { E ::= E+T {printf ("+");} | T
       T ::= T*F {printf ("*");} | F
       F ::= a { printf ("A"); }
     }
```



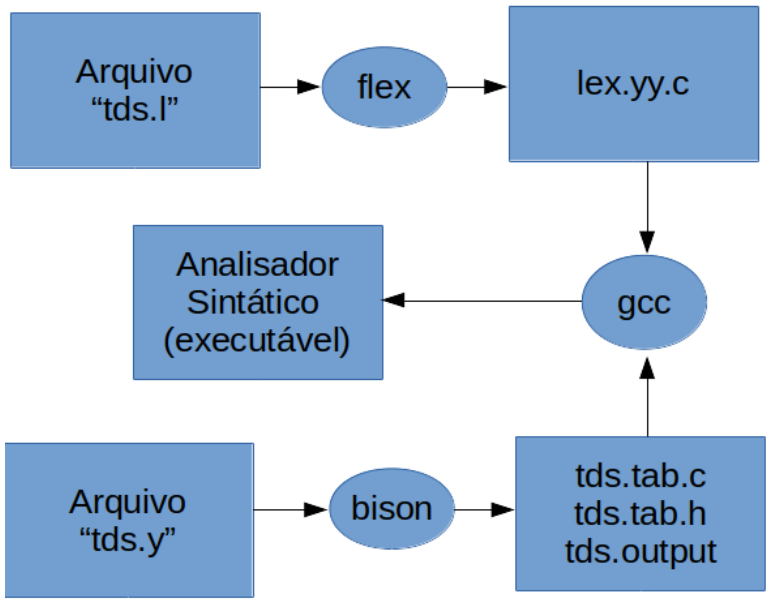
$$a_1 + a_2 * a_3 - a_4$$

$$\Rightarrow$$

$$A_1 A_2 A_3 * + A_4 -$$

Bison

- Bison: TDS para gramáticas *bottom-up*.
- Normalmente usado em conjunto com o flex.
- O arquivo ".tab.c" é um autômato a pilha.
- O arquivo ".output" é uma versão "legível" do autômato.



Estrutura do arquivo .y

- Um arquivo de entrada do bison é dividido em três partes.
- Organização semelhante ao flex.

```
...     } Definições
%%
...     } Regras
%%
...     } subrotinas
```

Definições

- O que for colocado entre `%{ ...%}` aqui será copiado no começo do arquivo `.tab.c`.
- O que vem em seguida são diretivas bison que geram código em `.tab.c` e `.tab.h`.
- A principal é `%token`, que é mapeado como um `#define` e pode ser usado no arquivo lex (`return IDENT`).

```
%{
#include <stdio.h>
%}
%token IDENT MAIS MENOS OR ASTERISCO DIV ABRE_PARENTESES FECHA_PARENTESES
```

Bison

- Formato de gramática para contruir árvores de derivação.
- Recebe os `tokens` (no caso, do flex) e os usa para "decorar" a árvore (colocar nos nós folha.
- O exemplo abaixo é de uma gramática não ambígua.

```
%%
expr        : expr MAIS termo {printf ("+"); } |
              expr MENOS termo {printf ("-"); } |
              termo
;
...
```

Bison

- Trechos de código que copiados para o arquivo `.tab.c`.
- Sempre incluir `main`, que executa `yyparse()` (a subrotina bison que faz o *parsing*).
- Por vezes precisa implementar `yyerror`.
- a função `yyparse` é quem dispara o analisador sintático.
-

```
%%
void yyerror(char *s) {
   fprintf(stderr, "%s\n", s);
   return 0;
}
main (int argc, char** argv) {
   yyparse();
   return 0;
}
```

Complemento

- Baixe o arquivo `Posfixo.tar.bz2`.
- Ele contém arquivos fonte flex e bison que converte uma entrada infixa e posfixa.
- Verifique como criar o executável (Makefile).
- Teste executável criado com várias entradas.
- Acrescente o identificador "b".
- Acrescente operadores "and" e "or".
- Assuma que "b" é sempre booleano e "a" é sempre inteiro. Agora, faça verificação de tipos (não é feito pelo bison, mas pelos nós "executáveis".
- Deve aceitar $\alpha =$ "$a + a * a$", $\beta =$ "b *and* b *or* b" mas deve dar erro com $\gamma =$ "$a + b$ e $\delta =$ "a *and* b.

Complemento

- Flex (*fast lex*) é uma alternativa de software livre e de código aberto ao lex;
- Bison X Yacc (*Yet Another Compiler Compiler)*

- Página para anotações

Licença

- Slides desenvolvidos somente com software livre:
  - LaTeX usando beamer;
  - Inkscape.
- Licença:
  - Creative Commons Atribuição-Uso Não-Comercial-Vedada a Criação de Obras Derivadas 2.5 Brasil License. `http://creativecommons.org/licenses/by-nc-nd/2.5/br/`