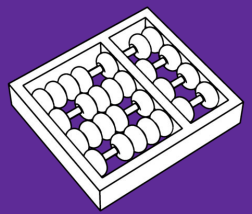
Análise Sintática

O chamado *Parser*

Hervé Yviquel
herve@ic.unicamp.br

Universidade Estadual de Campinas (Unicamp)
Instituto de Computação (IC)
Laboratório de Sistemas de Computação (LSC)

MC921 • Projeto e Construção de Compiladores • 2022 S2



UNICAMP

Aula Anterior

Resumo

- Visão Geral do Front-End
- Analisador Léxico
- Especificação de Tokens
- Gerador de Lexer

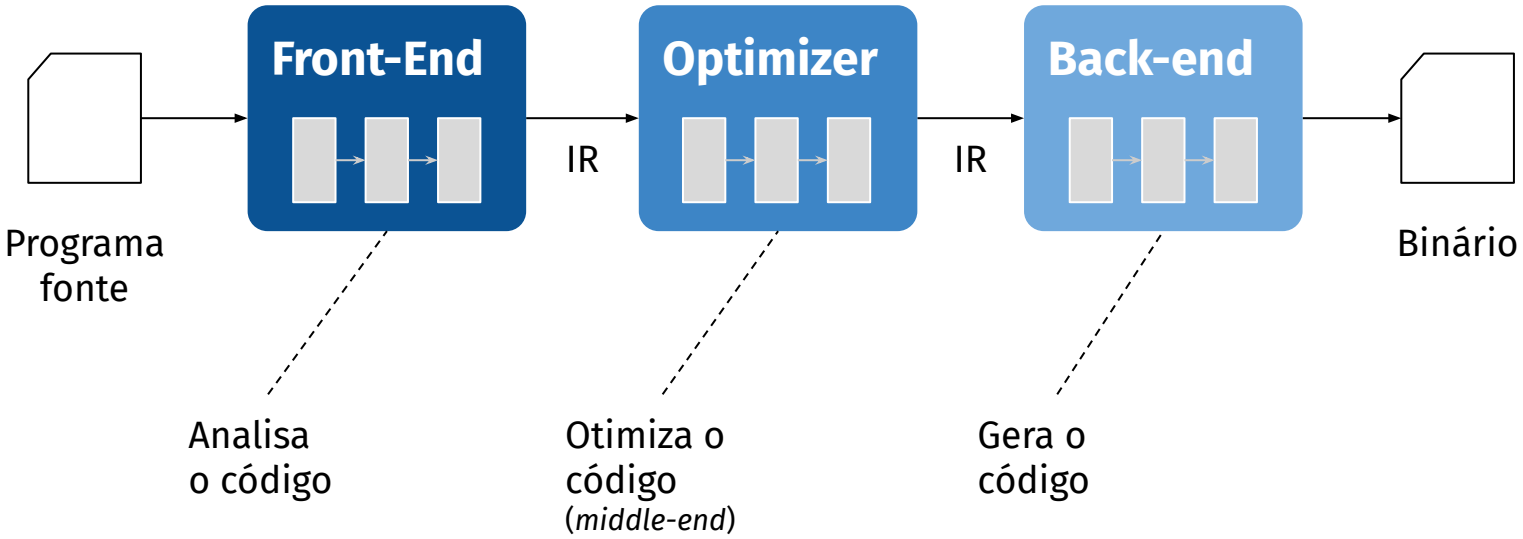
Aula de Hoje

Plano

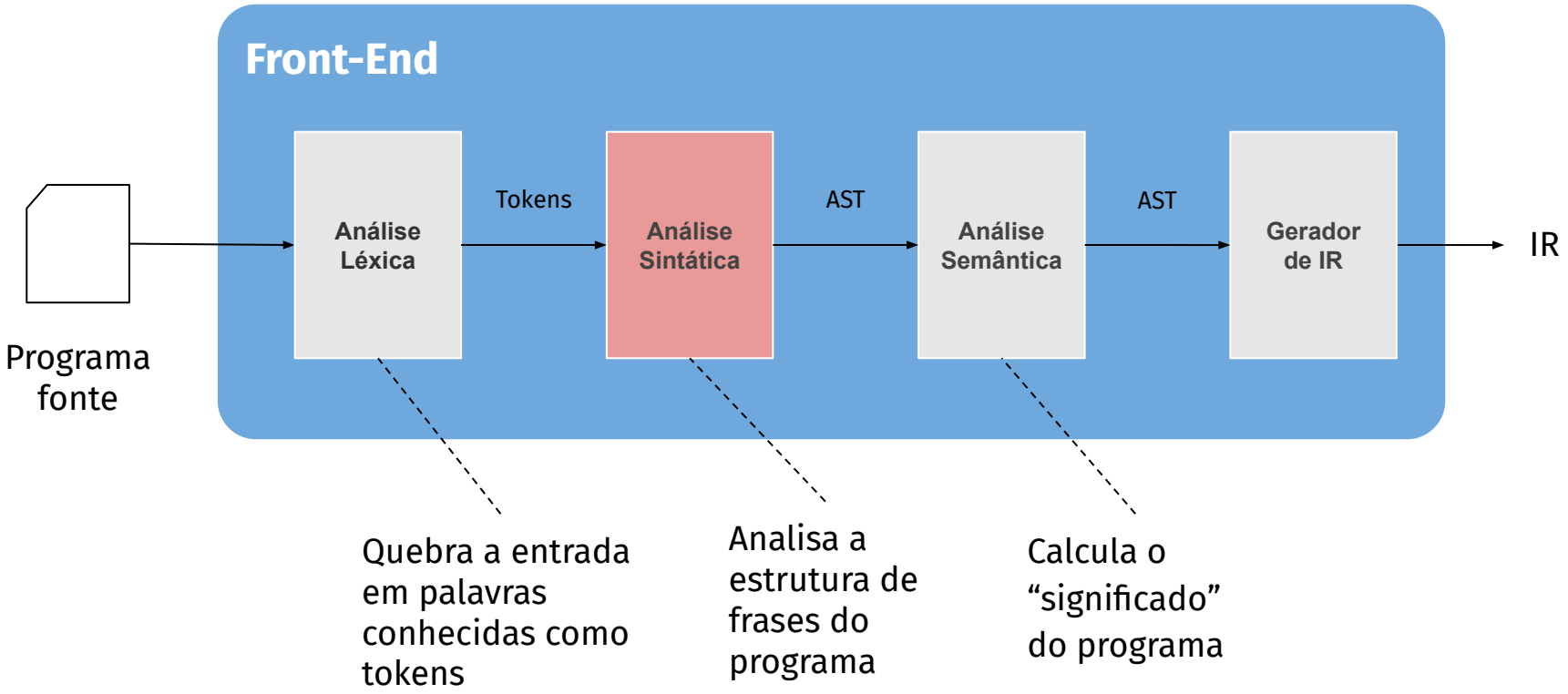
- Visão Geral do Front-end
- Analisador Sintático
- Gramáticas Livre de Contexto
- Gramáticas Ambíguas
- Especificação de Gramática
- Gerador de Parser

Visão Geral do Front-end

Estrutura Moderna do Compilador

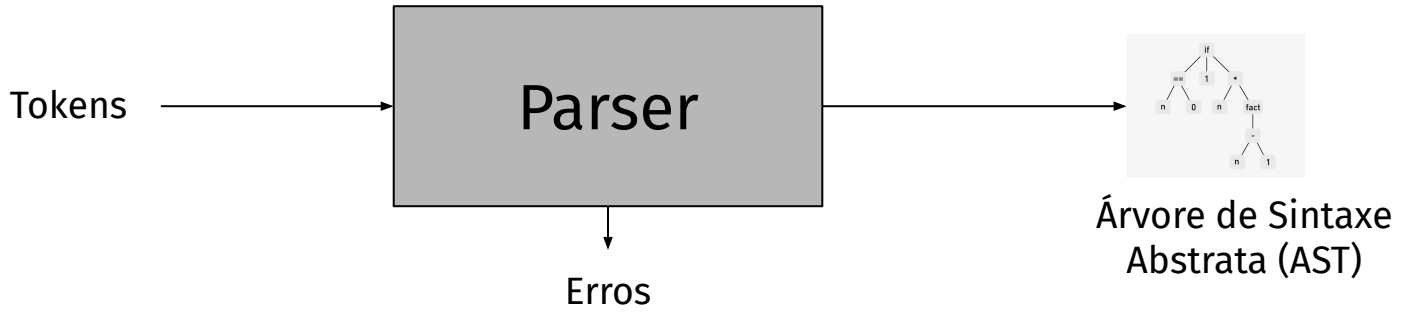


Front-end do Compilador



Analisador Sintático

Analisador Sintático



- Recebe uma sequência de tokens e determina se pode ser gerada através da gramática da linguagem fonte
  - É esperado ainda que ele reporte os erros de uma maneira inteligível
  - Seja capaz de se recuperar de erros comuns, continuando a processar a entrada

Que notação usar para representar “palavras” dentro de uma frase?

Gramáticas Livre de Contexto

Gramáticas Livre de Contexto

- Descreve uma linguagem através de um conjunto de regras de produção da forma:
  - *símbolos -> símbolos símbolos símbolos ... símbolos*
- Símbolos:
  - **terminais**: uma string do alfabeto da linguagem
  - **não-terminais**: aparecem do lado esquerdo
  - nenhum token aparece do lado esquerdo
  - existe um não-terminal definido como símbolo inicial

Gramáticas Livre de Contexto

1. S → S; S
2. S → id := E
3. S → print(L)
4. E → id
5. E → num
6. E → E + E
7. E → (S, E)
8. L → E
9. L → L, E

```
id := num; id := id + (id := num + num, id)
```

Possível código fonte:

```
a := 7;
b := c + (d := 5 + 6, d)
```

Derivações

`a := 7; b := c + (d := 5 + 6, d)`

- *S*
- *S* ; *S*
- *S* ; id := *E*
- id := *E*; id := *E*
- id := num ; id := *E*
- id := num ; id := *E* + *E*
- id := num ; id := *E* + (*S*, *E*)
- id := num ; id := id + (*S*, *E*)
- id := num ; id := id + (id := *E*, *E*)
- id := num ; id := id + (id := *E* + *E*, *E*)
- id := num ; id := id + (id := *E* + *E*, id)
- id := num ; id := id + (id := num + *E*, id)
- id := num ; id := id + (id := num + num, id)

1. S → S; S
2. S → id := E
3. S → print (L)
4. E → id
5. E → num
6. E → E + E
7. E → (S, E)
8. L → E
9. L → L, E

Definições

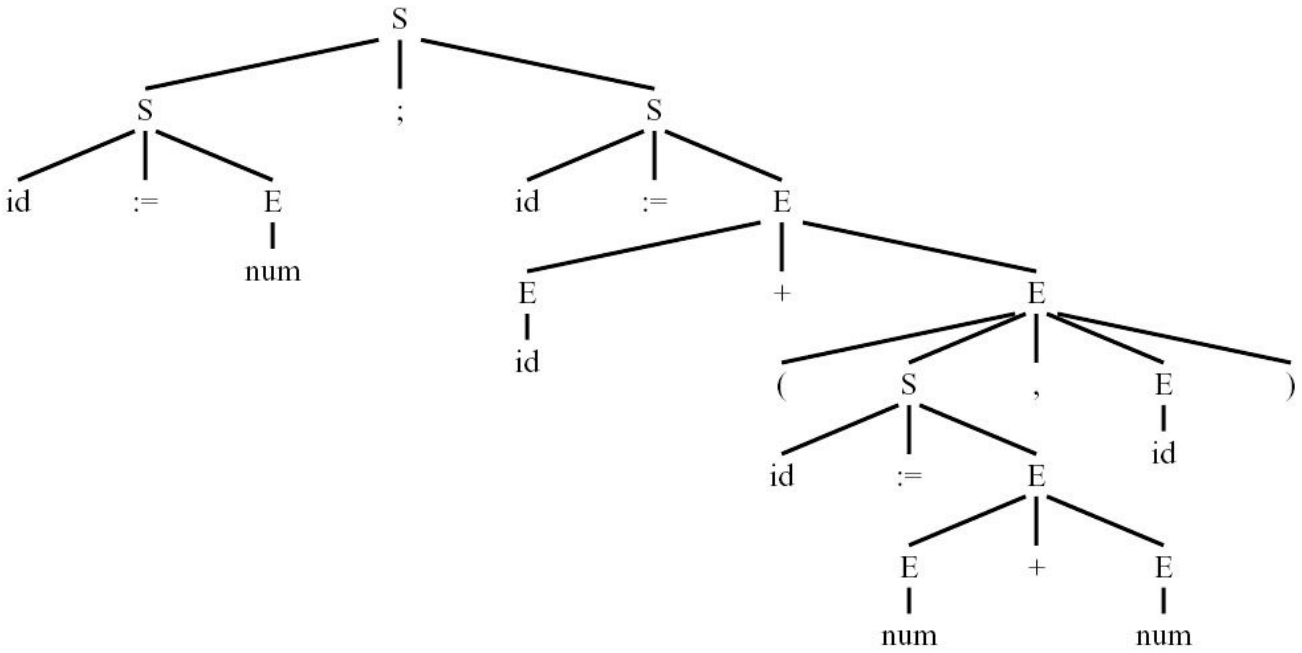
- *left-most*
  - o não terminal mais à esquerda é sempre o expandido
- *right-most*
  - idem para o mais à direita
- Qual é o caso do exemplo anterior?

Árvore de derivação

- Constrói-se uma árvore de derivação (parse tree) conectando-se cada símbolo em uma derivação da qual ele foi derivado
  - cada nó corresponde a um não terminal
  - as folhas correspondem aos terminais (tokens)
- Duas derivações diferentes podem levar a uma mesma parse tree
  - Ambiguidade

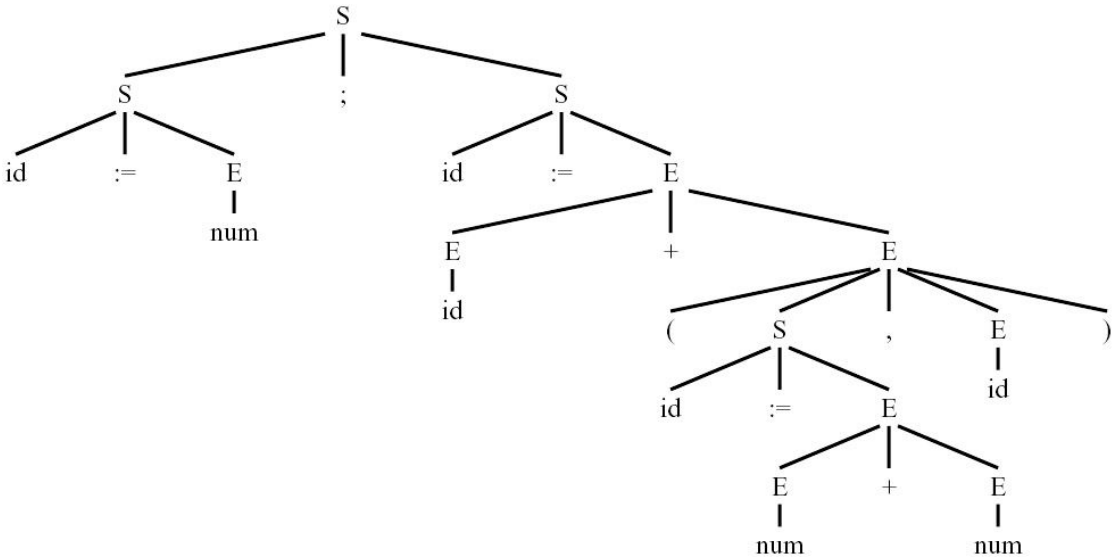
Exemplo de árvore de derivação


```
a := 7; b := c + (d := 5 + 6, d)
```



Análise Sintática

- O objetivo do parser é construir a árvore de derivação
  - no caso que não consegue, tem um erro sintático na entrada
- O árvore de derivação já contém suficiente informação para interpretar o programa



Exercícios

Escreva uma gramática usando os não-terminais A e B que reconhece

1. Cadeias que casam a expressão seguinte com n ≥ 1 e m ≥ 1
   $a^m b^n a^n b^m$
2. Cadeias no alfabeto {a,b} com no mínimo uma ocorrência de 'a' e uma ocorrência de 'b'

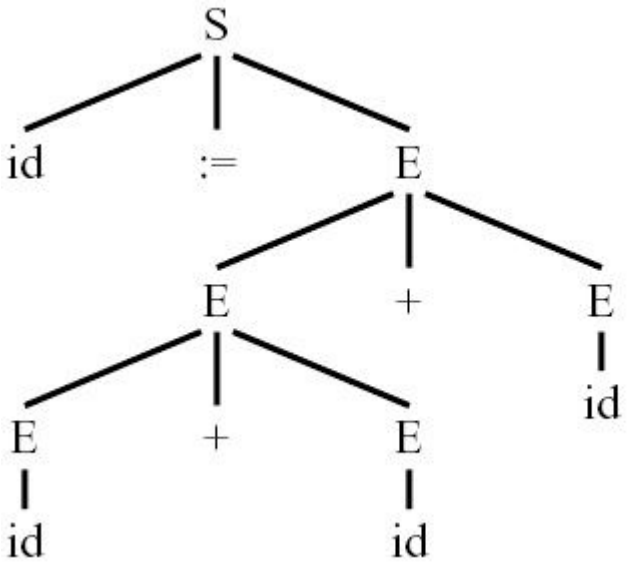
Soluções

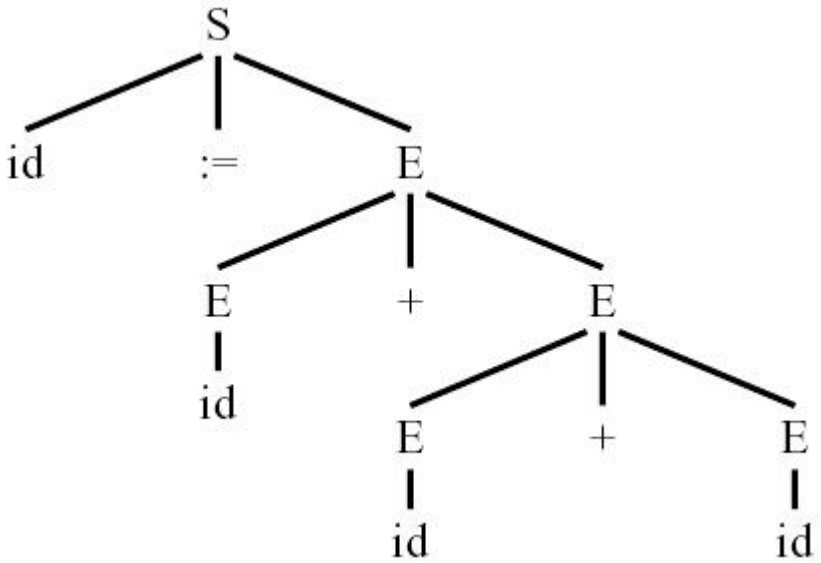
1. A->aAb
   A->aBb
   B->bBa
   B->ba
2. A->aBbB
   A->bBaB
   B->aB
   B->bB
   B->ε

Gramáticas Ambíguas

Gramáticas Ambíguas

- Uma gramática é ambígua quando, para alguma sentença da linguagem gerada, existe mais de uma árvore de derivação
  - id := id+id+id





É Ambígua?

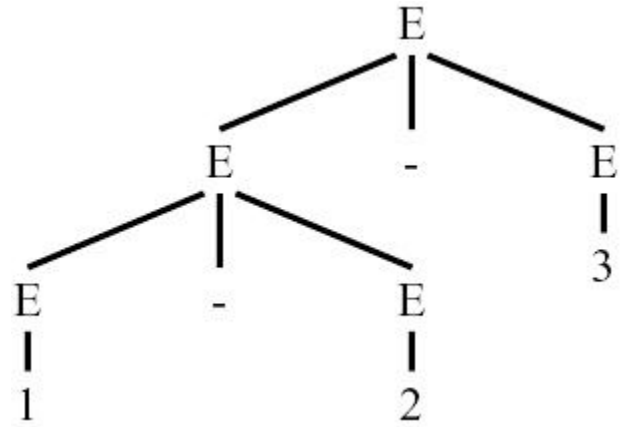
1. E → id
2. E → num
3. E → E * E
4. E → E / E
5. E → E + E
6. E → E - E
7. E → (E)

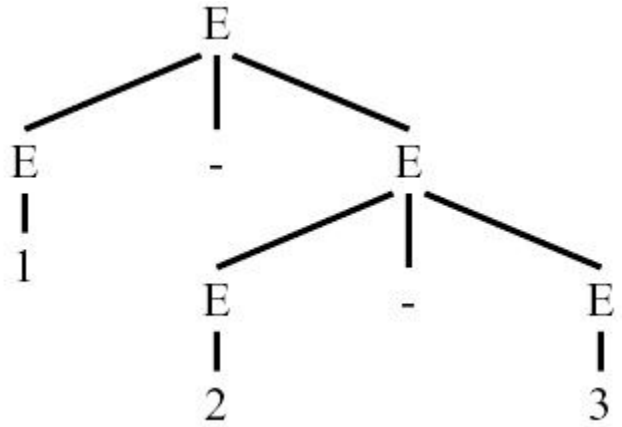
Construa Parse Trees para as seguintes expressões:

1-2-3

1+2*3

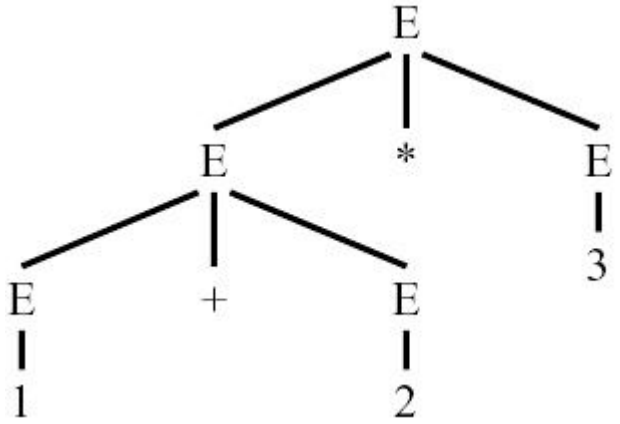
Exemplo: 1-2-3

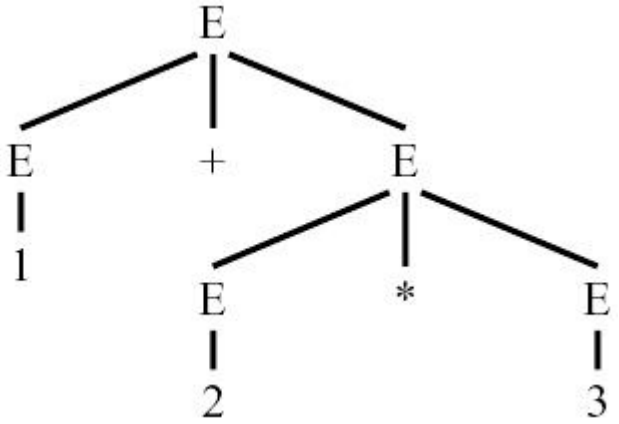




- Ambígua!
  - (1-2)-3 = -4 e 1-(2-3) = 2

Exemplo: 1+2*3





- Ambígua!
  - (1+2)*3 = 9 e 1+(2*3) = 7
- Mas qual o problema com isto?

Gramáticas Ambíguas

- Os compiladores usam as parse trees para extrair o significado das expressões
- A ambiguidade se torna um problema
- Podemos, geralmente, mudar a gramática de maneira a retirar a ambiguidade

Gramáticas Ambíguas

- Alterando o exemplo anterior:
  - Queremos colocar uma precedência maior para * em relação a + e –
  - Também queremos que cada operador seja associativo à esquerda: (1-2)-3 e não 1-(2-3)
- Conseguimos isso introduzindo novos não-terminais

Gramática para Expressões

1. E → E + T
2. E → E – T
3. E → T
4. T → T * F
5. T → T / F
6. T → F
7. F → id
8. F → num
9. F → (E)

Construa as derivações e Parse Trees para as seguintes expressões:

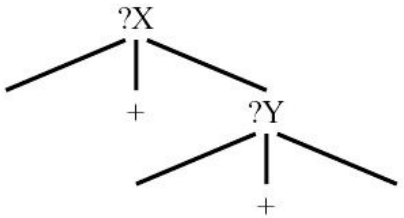
1-2-3

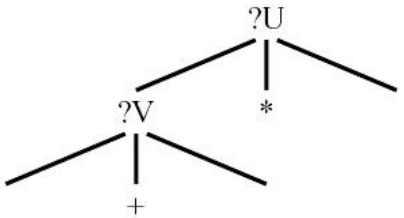
1+2*3

Gramática para Expressões

1. E → E + T
2. E → E − T
3. E → T
4. T → T * F
5. T → T / F
6. T → F
7. F → id
8. F → num
9. F → (E)

Essa gramática pode gerar as árvores abaixo?





Gramáticas Ambíguas

- Geralmente podemos transformar uma gramática para retirar a ambiguidade
- Algumas linguagens não possuem gramáticas não ambíguas
  - As ambiguidades são tratada na mão
- No caso que não for possível tratar as ambiguidades
  - elas não seriam apropriadas como linguagens de programação

Símbolo inicial


```
 1.  S → E $
 2.  E → E + T
 3.  E → E - T
 4.  E → T
 5.  T → T * F
 6.  T → T / F
 7.  T → F
 8.  F → id
 9.  F → num
10.  F → (E)
```

- Definir não terminal como símbolo inicial
  - “$” como fim do arquivo
- Sentença correta
  - Se consegue fazer um derivação a partir do símbolo inicial que consegue consumir o fim do arquivo

Como as gramáticas são especificadas?

BNF and EBNF

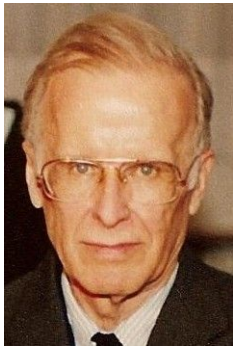
O que é BNF?

- Significa “Backus-Naur Form”
- É uma maneira formal e matemática de especificar gramáticas livres de contexto
- É preciso e inequívoco
- Antes da BNF, pessoas especificadas linguagens de programação de forma ambígua, ou seja, com inglês

Como surgiu o BNF?

- John Backus apresentou uma nova notação contendo a maioria dos elementos de BNF em uma conferência da UNESCO
  - Sua apresentação foi sobre Algol 58
- Peter Naur leu este relatório e descobriu que ele e Backus interpretaram Algol de forma diferente
  - Ele queria ainda mais precisão
- Então ele criou o que hoje conhecemos como BNF para Algol 60
  - Assim, BNF foi publicado pela primeira vez no *Algol 60 Report*

Quem?!?



- Quem foi John Backus?
  - Inventou o FORTRAN
  - Grande influência na invenção de programação funcional na década de 1970
  - Ganhou o Prêmio Turing de 1977 para BNF e FORTRAN

- Quem foi Peter Naur?
  - Astrônomo dinamarquês virou cientista de Ciência da Computação
  - Contribuiu ao ALGOL 60

Um pouco mais de história...

- BNF originalmente significava “Backus Normal Form”
- Em 1964, Donald Knuth escreveu uma carta publicada em *Communications of the ACM* em que sugere que seja a forma Backus-Naur
  - Reconhecer a contribuição de Naur
  - A BNF não é tecnicamente uma “forma normal”
  - Não há apenas uma maneira correta de escrever uma gramática

Como é o BNF?


```
<number> ::= <digit> | <number> <digit>
<digit> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

- “::=” significa “é definido como”
  - algumas variantes usam “:=” ou “:”
- “|” significa "ou"
- Colchetes angulares “<>” significam um não terminal
- Símbolos sem colchetes são terminais

Outros exemplos de BNF


```
<while loop> ::= while ( <condition> ) <statement>

<assignment statement> ::= <variable> = <expression>

<statement list> ::= <statement>
                   | <statement list> <statement>

<unsigned integer> ::= <digit>
                     | <unsigned integer><digit>
```

BNF para Expressões


```
<expression> ::= <expression> + <term>
               | <expression> - <term>
               | <term>

<term> ::= <term> * <factor>
         | <term> / <factor>
         | <factor>

<factor> ::= <primary> ^ <factor>
           | <primary>
```

```
<primary> ::= <primary>
            | <element>

<element> ::= ( <expression> )
            | <variable>
            | <number>
```

EBNF

- Significa “Extended Backus-Naur Form”
- Depois que a BNF apareceu com o Algol 60, muitos pessoas adicionaram suas próprias extensões
  - Niklaus Wirth queria ver uma única forma, então publicou “What Can We Do About the Unnecessary Diversity of Notation for Syntactic Definitions” em *Communications of the ACM* em 1977
  - Ele sugeriu o uso de “[..]” para opcional símbolos (0 ou 1 ocorrências), “{ .. }” para 0 ou mais ocorrências.
- Não mencionou “EBNF” ou Kleene cross

Quais são as Extensões?

- Eles variam, mas geralmente são derivados da sintaxe das expressões regulares
  - “*” (The Kleene Star): significa 0 ou mais ocorrências
  - “+” (The Kleene Cross): significa 1 ou mais ocorrências
  - “?”: significa 0 ou 1 ocorrência (às vezes “[...]” usado no lugar)
  - Uso de parênteses para agrupamento

BNF vs EBNF

BNF

```
<expr> ::= '-' <num> | <num>

<num> ::= <digits>
        | <digits> '.' <digits>

<digits> ::= <digit>
           | <digit> <digits>

<digit> ::= '0' | '1' | '2' | '3'
          | '4' | '5' | '6' | '7'
          | '8' | '9'
```

EBNF

```
<expr> ::= '-'? <digit>+ ('.' <digit>+)?

<digit> ::= '0' | '1' | '2' | '3'
          | '4' | '5' | '6' | '7'
          | '8' | '9'
```

- Muito mais conciso!
- Um '-' opcional, um ou mais dígitos, um ponto decimal seguido por um ou mais dígitos opcional

Conversões simples

- Se tiver uma regra como:

```
<expr> ::= <digits>

<digits> ::= <digit>
           | <digit> <digits>
```

- Pode substituí-la por:

```
<expr> ::= <digit>+
```

Conversões simples (2)

- Se tiver uma regra como:

```
<expr> ::= <digits> | empty
```

- Pode substituí-la por:

```
<expr> ::= <digit>*
```

Conversões simples (3)

- Se tiver uma regra como:

```
<id> ::= <letter>
       | <id><letter>
       | <id><digit>
```

- Pode substituí-la por:

```
<id> ::= <letter> (<letter> | <digit>)*
```

Gerador de Parser

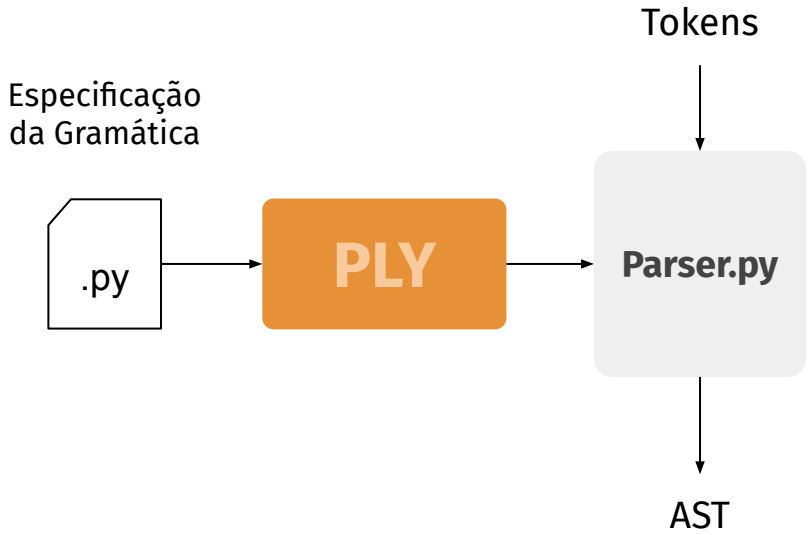
Especificação do Parser

- As regras gramaticais são listadas
  - Eventualmente associada com ações (construção do AST)
- Esse arquivo serve como entrada do gerador de lexer

```
<expr> ::= <digits>

<digits> ::= <digit>
           | <digit> <digits>

...
```

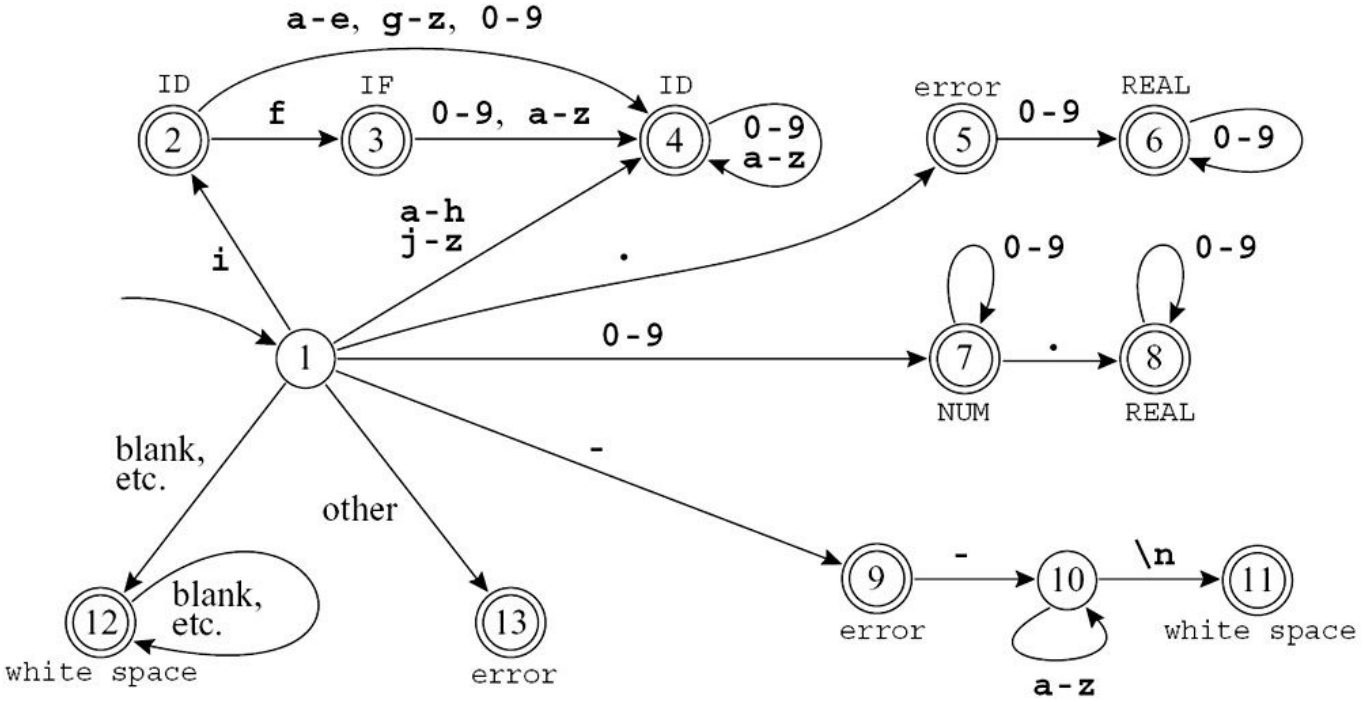
Fluxo do gerador de parser

- O gerador de parser gera o parser automaticamente
  - a partir da especificação
- Tem várias geradores de parser disponíveis
  - Yacc, Bison, ANTLR, PLY, ...
- Usarão PLY no projeto 2
  - Gerar um parser para a linguagem uC



Como funciona o gerador de parser?

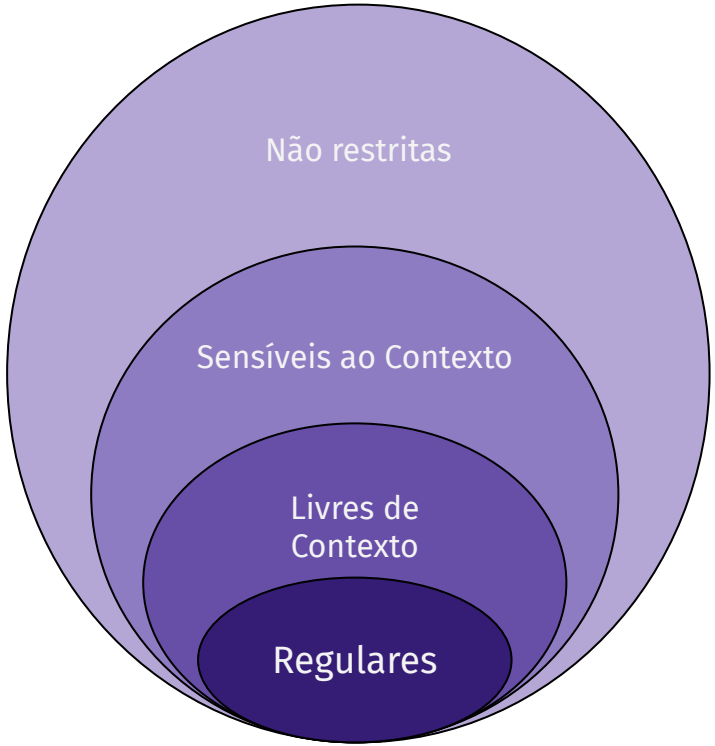
Como Autômatos Finitos?



Problema da Recursão

- É possível usar expressões regulares para definir uma linguagem para expressões que tenham parênteses balanceados?
  - (1+(245+2))
- ERs são boas para definir a estrutura léxica de maneira declarativa
  - Mas não são “poderosas” o suficiente para conseguir definir declarativamente a estrutura sintática de linguagens de programação
  - Vamos precisar de associar o autômato com uma pilha
- Que notação usar para representar “palavras” (cadeias) dentro de uma frase?
  - Gramáticas

Hierarquia de Chomsky*

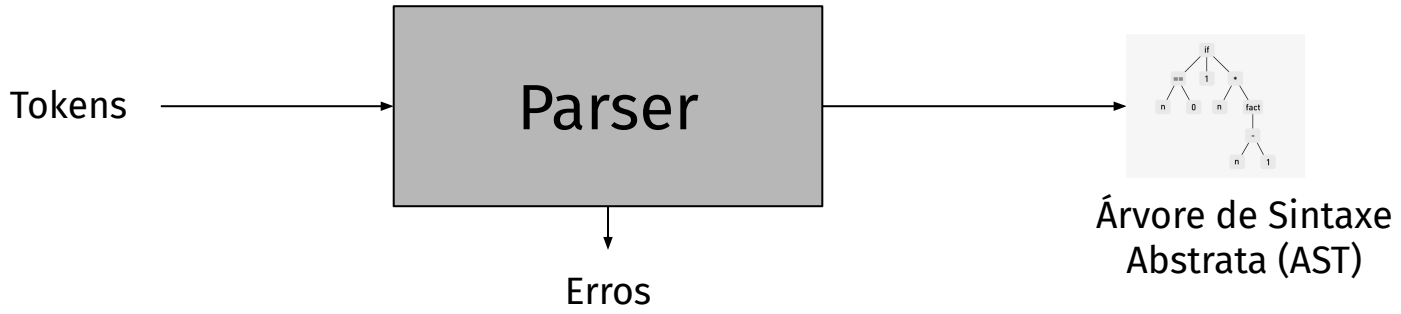


Class	Grammars	Languages	Automaton
Type-0	Unrestricted	Recursive Enumerable	Turing Machine
Type-1	Context sensitive	Context sensitive	Linear-Bound
Type-2	Context free	Context free	Pushdown
Type-3	Regular	Regular	Finite

*Noam Chomsky é um linguista americano considerado com "o pai da linguística moderna"

Gerar um Parser com PLY

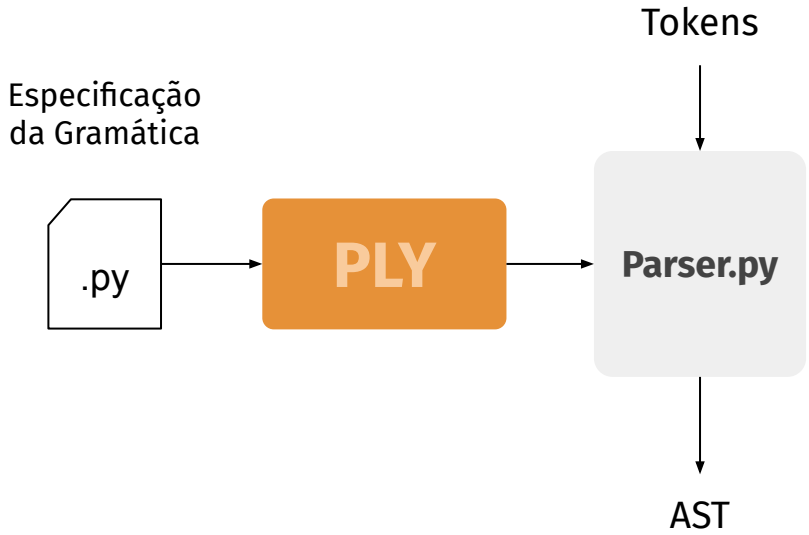
Analisador Sintático



- Recebe uma sequência de tokens e determina se pode ser gerada através da gramática da linguagem fonte
  - É esperado ainda que ele reporte os erros de uma maneira inteligível
  - Seja capaz de se recuperar de erros comuns, continuando a processar a entrada

Fluxo do gerador de parser

- O gerador de parser gera o parser automaticamente
  - a partir da especificação
- Tem várias geradores de parser disponíveis
  - Yacc, Bison, ANTLR, PLY, ...
- Usarão PLY no projeto 2
  - Gerar um parser para a linguagem uC



Exemplo


```
program ::= assignment

assignment ::= ID EQUALS expr SEMI

expr ::= expr PLUS expr
       | expr TIMES expr
       | NUM
```

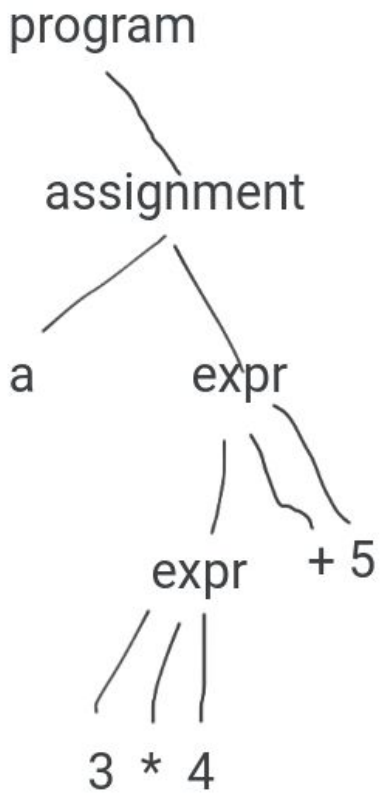
```python
def p_program(self, p):
    ''' program : assignment
    '''

def p_assign_statement(self, p):
    ''' assignment : ID EQUALS expression SEMI'''

def p_expr(self, p):
    ''' expression : expression PLUS expression
                   | expression TIMES expression
                   | expression : NUM
    '''
```

PLY gera uma árvore de derivação?

a = 3 * 4 + 5;

??

Não, tem que implementar usando **ações semânticas!**



Ações Semânticas


```
def p_program(self, p):
    ''' program : assignment
    '''
    p[0] = ('program', p[1])

def p_assign_statement(self, p):
    ''' assignment : ID EQUALS expression SEMI
    '''
    p[0] = ('assignment', p[1], p[3])

def p_expr(self, p):
    ''' expression : expression PLUS expression
                   | expression TIMES expression
                   | NUM
    '''
    p[0] = (p[1], p[2], p[3])
```

Cada **elemento da regra de produção** é um **elemento do p**

Qual é o problema?

Ações Semânticas

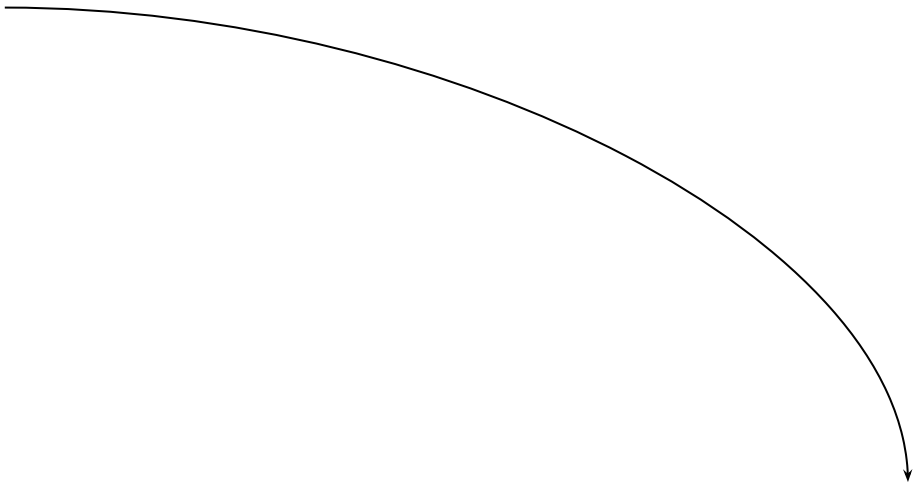

```
def p_program(self, p):
    ''' program : assignment
    '''
    p[0] = ('program', p[1])

def p_assign_statement(self, p):
    ''' assignment : ID EQUALS expression SEMI
    '''
    p[0] = ('assignment', p[1], p[3])

def p_binop_expr(self, p):
    ''' expression : expression PLUS expression
                   | expression TIMES expression
    '''
    p[0] = (p[1], p[2], p[3])
```

```
def p_num_expr(self, p):
    ''' expression : NUM
    '''
    p[0] = p[1]
```

Divide em 2 regras

Gerando o ParseTree como tupla

a = 3 * 4 + 5;



('program', ('assignment', 'a', ((3, '*', 4), '+', 5)))

E a ambiguidade?


```
program ::= assignment

assignment ::= ID EQUALS expr SEMI

expr ::= expr PLUS expr
       | expr TIMES expr
       | NUM
```

```
# Ensure TIMES has higher precedence than
PLUS.
   precedence = (
        ('left', 'PLUS'),
        ('left', 'TIMES'),
   )
```

O PLY sinaliza **ambiguidade** com “shift-reduce conflict”

Só precisa definir as prioridades

Árvore Sintática Abstrata

Parse Tree vs AST

- A árvore de derivação tem muita informação redundante
  - Separadores, terminadores, etc
  - não-terminais auxiliares introduzidos para contornar limitações das técnicas de análise sintática
- Também trata todos os nós de forma homogênea, dificultando processamento deles
- A AST joga fora a informação redundante
  - classifica os nós de acordo com o papel que eles têm na estrutura sintática da linguagem
  - fornece ao compilador uma representação compacta e fácil de trabalhar da estrutura dos programas

Representando ASTs

- Cada estrutura sintática da linguagem, normalmente dada pelas produções de sua gramática, dá um tipo de nó da AST
  - No nosso compilador uC, vamos usar uma classe para cada tipo de nó
- Não-terminais com várias produções podem usar varias tipo de classe
- Nem toda produção ganha sua própria classe
  - Algumas podem ser redundantes

AST para uC

- Árvore baseado em objeto
- 1 classe abstrata *Node*
- 1 classe abstrata *DeclType(Node)*
- 29 classes concretas implementando *Node* ou *DeclType*
  - *While*, *For*, *If*, *Compound*, *Assignment*, *Break*, *FuncCall*, *ID*, *VarDecl*, etc.
- Todas as operações são juntas em duas classes
  - *UnaryOp*, *BinaryOp*
  - A operação é só um campo
- Todos os tipos são junto em uma única classe
  - *Type*
  - O nome do tipo é só um campo

Exemplo de AST para uC


```
int a = 3 * 4 + 5, b[1][2];
```

Pretty-print

```
Program:
  GlobalDecl:
    Decl: ID(name=a)
      VarDecl:
        Type: int @ 1:1
      BinaryOp: + @ 1:9
        BinaryOp: * @ 1:9
          Constant: int, 3 @ 1:9
          Constant: int, 4 @ 1:13
        Constant: int, 5 @ 1:17
    Decl: ID(name=b)
      ArrayDecl:
        ArrayDecl:
          VarDecl:
            Type: int @ 1:1
          Constant: int, 2 @ 1:25
        Constant: int, 1 @ 1:22
```

Classe Program


```
class Program(Node):

    attr_names = ()

    def __init__(self, gdecls, coord=None):
        self.gdecls = gdecls  # program's global declarations
        self.coord = coord

    def children(self):
        nodelist = []
        for i, child in enumerate(self.gdecls or []):
            nodelist.append(("gdecls[%d]" % i, child))
        return tuple(nodelist)
```

Classe Constante


```
class Constant(Node):

    attr_names = ('type', 'value')

    def __init__(self, type, value, coord=None):
        self.type = type  # primitive type (int, char, etc.)
        self.value = value  # constant value
        self.coord = coord

    def children(self):
        return ()
```

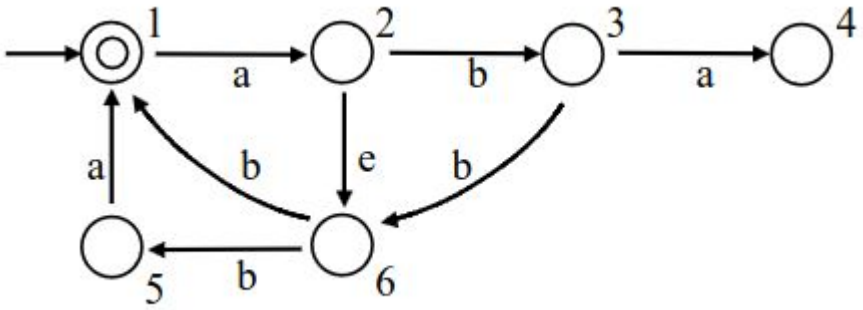
Exercícios

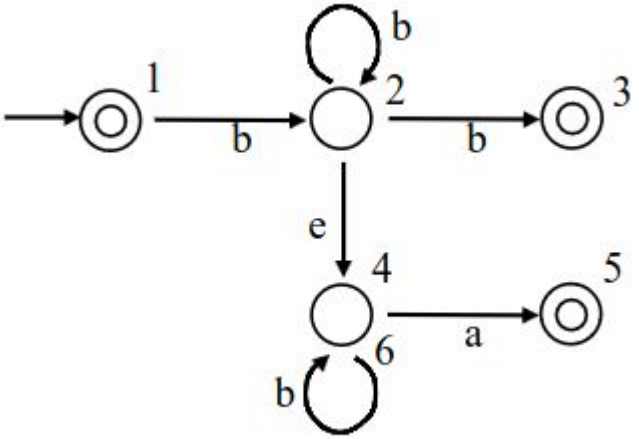
Exercícios

1. Escreva a menor gramática possível usando somente o não-terminal A que reconhece palíndromos no alfabeto {a,b}
2. Transforme essas expressões regulares em gramáticas:
   a. ((xy*x)|(yx*y))?
   b. ((0|1)+“.”(0|1)*)|((0|1)*“.”(0|1)+)
3. Escreva a gramática da BNF
4. Escreva a gramática das expressões regulares

Exercícios

Quais linguagens reconhecem esses autômatos?





Exercícios

***3.2** Write a grammar for English sentences using the words

```
time, arrow, banana, flies, like, a, an, the, fruit
```

and the semicolon. Be sure to include all the senses (noun, verb, etc.) of each word. Then show that this grammar is ambiguous by exhibiting more than one parse tree for "time flies like an arrow; fruit flies like a banana."

Resumo

- Visão Geral do Front-end
- Analisador Sintático
- Gramáticas Livre de Contexto
- Gramáticas Ambíguas
- Especificação de Gramática
- Gerador de Parser

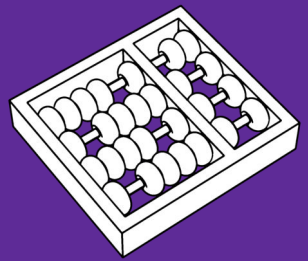
Leitura Recomendada

- Capítulo 3.1-3.2 do livro do Cooper.
- Capítulo 3-3.1 do livro do Appel.

Próxima Aula

- Front-end do Compilador (3)
  - Análise Semântica
  - Verificação de tipo
  - Geração da IR

Obrigado!
Merci!



UNICAMP

Pallete

Bubbles

BUBBLE
Lorem ipsum dolor sit amet, consectetur adipiscing elit.

BUBBLE
Lorem ipsum dolor sit amet, consectetur adipiscing elit.

BUBBLE
Lorem ipsum dolor sit amet, consectetur adipiscing elit.

BUBBLE
Lorem ipsum dolor sit amet, consectetur adipiscing elit.

BUBBLE
Lorem ipsum dolor sit amet, consectetur adipiscing elit.

BUBBLE
Lorem ipsum dolor sit amet, consectetur adipiscing elit.

BUBBLE
Lorem ipsum dolor sit amet, consectetur adipiscing elit.

BUBBLE
Lorem ipsum dolor sit amet, consectetur adipiscing elit.

BUBBLE
Lorem ipsum dolor sit amet, consectetur adipiscing elit.

BUBBLE
Lorem ipsum dolor sit amet, consectetur adipiscing elit.

BUBBLE
Lorem ipsum dolor sit amet, consectetur adipiscing elit.

DRACULA

Tables

Table Title	
Column 1	**Column 2**
One	Two
Three	Four

Table Title	
Column 1	**Column 2**
One	Two
Three	Four

Table Title	
Column 1	**Column 2**
One	Two
Three	Four

Table Title	
Column 1	**Column 2**
One	Two
Three	Four

Emojis