SBVCONC: Construção de Compiladores

Aula 06: Análise Sintática (*Parsing*)

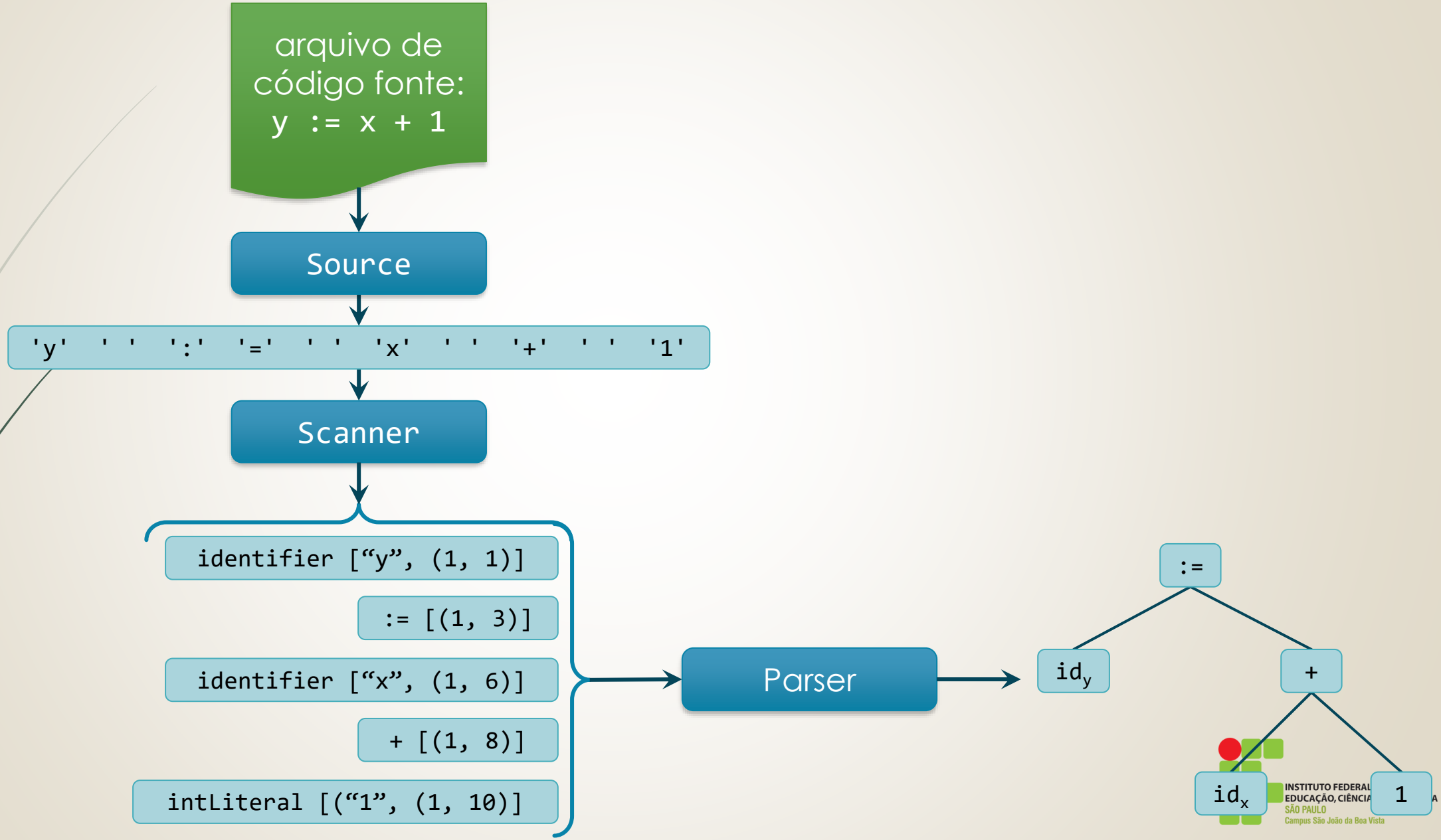
Bacharelado em Ciência da Computação
Prof. Dr. David Buzatto

INSTITUTO FEDERAL DE EDUCAÇÃO, CIÊNCIA E TECNOLOGIA
SÃO PAULO
Campus São João da Boa Vista

Analisador Sintático (*Parser*)

- Verifica se as regras gramaticais da linguagem são satisfeitas;
- A estrutura geral de um *parser* é baseada em gramáticas livres de contexto, representadas usando BNF/EBNF;
- **Entrada:** um fluxo de tokens do analisador léxico (*scanner*). Da perspectiva do *parser*, cada token é tratado como um símbolo terminal;
- **Saída:** uma representação intermediária do código fonte, por exemplo, árvores sintáticas abstratas.

Analisador Sintático (*Parser*)



Funções do *Parser*

- Reconhecimento da linguagem baseada na sintaxe, assim como definida na gramática livre de contexto;
- Tratamento/Recuperação de Erros;
- Geração de representação intermediária, no nosso caso, geração de árvores sintáticas abstratas;
- Nesta aula nosso foco será no reconhecimento da linguagem, nas próximas duas aulas lidaremos com o tratamento/recuperação de erros e a geração das árvores sintáticas abstratas.

Análise Descendente Recursiva

- Técnica de análise usada nessa disciplina: análise descendente recursiva com um único símbolo de *lookahead*. Vamos discutir rapidamente outras técnicas existentes no final dessa aula;
- Usa métodos recursivos para "descer" através da árvore de análise (análise *top-down*) enquanto o programa é analisado;
- O *parser* é construído sistematicamente a partir da gramática, aplicando uma série de refinamentos.

Transformações Iniciais da Gramática

- Iniciar com uma gramática não ambígua;
- Na gramática, separar as regras léxicas das regras estruturais:
  - O *scanner* tratará regras simples (operadores, identificadores etc);
  - Símbolos reconhecidos pelo *scanner* tornam-se símbolos terminais na gramática do *parser*;
- Eliminar recursões à esquerda;
- Realizar a fatoração à esquerda das regras sempre que possível;
- Algumas restrições gramaticais serão discutidas à seguir.

Análise Descendente Recursiva
Refinamento 1

- Para cada regra da gramática:

  ```
  N = … .
  ```

  definimos um método do *parser* com o nome:

  ```
  parseN()
  ```

- Exemplo, para a regra:

  ```
  assignmentStmt = variable ":=" expression ";" .
  ```

  definimos um método do *parser* com o nome:

  ```
  parseAssignmentStmt()
  ```

- As transformações na gramática podem ser usadas para simplificar a gramática antes de aplicar esse refinamento, por exemplo, a substituição de não-terminais.

Os Métodos da Análise Descendente Recursiva

- Os métodos na forma `parseN()` do *parser* funcionam da seguinte forma:
  - O método `getSymbol()` do *scanner* provê um símbolo de *lookahead* para os métodos de análise;
  - No início (entrada) da execução do método `parseN()`, o símbolo retornado pelo *scanner* **deve ser um símbolo que está no início do lado direito** da regra `N = … .`
  - No fim (saída) da execução do método `parseN()`, o símbolo retornado pelo *scanner* **deve ser o primeiro símbolo que segue uma frase sintática** que corresponde à `N`
  - Se as regras de produção possuem referências recursivas, os métodos de análise também terão chamadas recursivas.

Analisando o Lado Direito da uma Regra

- Agora, nos atentaremos ao refinamento do método `parseN()` associado à regra de produção "`N = … .`" examinando a forma da expressão gramatical que está do lado direito da regra;
- Como exemplo, para a regra:

```
assignmentStmt = variable ":=" expression ";" .
```

definimos um método de análise com o nome de:

```
parseAssignmentStmt()
```

Focaremos na implementação sistemática desse método examinando, para isso, o lado direito da regra.

Análise Descendente Recursiva
Refinamento 2

- Uma sequência de fatores sintáticos $F_1 F_2 F_3$ ... é reconhecida ao se analisar cada um dos fatores, um de cada vez e em ordem;
- Em outras palavras, o algoritmo para analisar $F_1 F_2 F_3$ ... é, simplesmente:
  - O algoritmo usado para analisar F_1, seguido de
  - O algoritmo usado para analisar F_2, seguido de
  - O algoritmo usado para analisar F_3, seguido de
  - ...

Análise Descendente Recursiva
Refinamento 2 (exemplo)

- O algoritmo usado para analisar:

  `variable ":=" expression ";"`

  é simplesmente:
  - O algoritmo usado para analisar `variable`, seguido de
  - O algoritmo usado para analisar `":="`, seguido de
  - O algoritmo usado para analisar `expression`, seguido de
  - O algoritmo usado para analisar `";"`

Análise Descendente Recursiva
Refinamento 3

- Um único símbolo terminal `t` do lado direito de uma regra é reconhecido ao se invocar o método auxiliar de análise `match(t)`, definido como:

```
private void match( Symbol expectedSymbol )
            throws IOException, ParserException {
    if ( scanner.getSymbol() == expectedSymbol )
        scanner.advance();
    else
        ...   // throw ParserException
}
```

- **Exemplo:** O algoritmo para reconhecer o operador de atribuição ":=" é simplesmente a invocação do método:

```
match( Symbol.assign );
```

Análise Descendente Recursiva
Refinamento 4

- Um símbolo não-terminal `N` do lado direito de uma regra é reconhecido ao se invocar o método correspondente à regra `N`, ou seja, o algoritmo que reconhece o não-terminal `N` é simplesmente a invocação do método `parseN()`;
- **Exemplo:** O algoritmo para reconhecer o símbolo não-terminal `expression` do lado direito de uma regra é simplesmente a invocação do método `parseExpression()`.

Aplicação dos Refinamentos para a Análise Descendente Recursiva

- Considere a regra da instrução de atribuição:

```
assignmentStmt = variable ":=" expression ";" .
```

- O método de análise completo que reconhece a instrução de atribuição é o seguinte:

```
public void parseAssignmentStmt() {
    parseVariable();
    match( Symbol.assign );
    parseExpression();
    match( Symbol.semicolon );
}
```

Conjuntos $First$ e $Follow$

- A construção de um parser preditivo é auxiliada por duas funções associadas a uma gramática G;
- As funções $First$ e $Follow$ nos permitem preencher as entradas de uma tabela de análise preditiva para G, sempre que possível;
- Os conjuntos de tokens gerados pela função $Follow$ também podem ser usados como tokens de sincronização durante a recuperação de erros.

Conjuntos $First$ $(Primeiros)$

- O conjunto de todos os símbolos terminais que podem aparecer no início de uma expressão sintática $\boldsymbol{\alpha}$ é denotado por $First(\boldsymbol{\alpha})$;
- A expressão sintática $\boldsymbol{\alpha}$ é uma cadeia de símbolos gramaticais (terminais e não-terminais);
- Os conjuntos $First$ provêm informações importantes que podem ser usadas para guiar as decisões durante o desenvolvimento do *parser* (tanto descendente quanto ascendente).

Regras para Computar os Conjuntos $First$

- Se t é um símbolo terminal, então:

  $First(\,\mathtt{t}\,) = \{\,\mathtt{t}\,\}$

- Se $\mathtt{E} \rightarrow \varepsilon$ é uma produção, então adicione ε ao $First(\mathtt{E})$:

- Se todas as strings derivadas de E não são vazias, então:

  $First(\,\mathtt{EF}\,) = First\{\,\mathtt{E}\,\}$

- Se algumas strings derivadas de E podem ser vazias, então:

  $First(\,\mathtt{EF}\,) = First\{\,\mathtt{E}\,\} \cup First\{\,\mathtt{F}\,\}$

- $First(\,\mathtt{E}|\mathtt{F}\,) = First\{\,\mathtt{E}\,\} \cup First\{\,\mathtt{F}\,\}$

Casos Especiais para a Computação dos Conjuntos $First$

- As regras abaixo podem ser derivadas como casos especiais das regras anteriores:
  - $First(\ (\mathrm{E})*\ ) = First(\mathrm{E})$
  - $First(\ (\mathrm{E})+\ ) = First(\mathrm{E})$
  - $First(\ (\mathrm{E})?\ ) = First(\mathrm{E})$
  - $First(\ (\mathrm{E})*\ \mathrm{F}\ ) = First(\mathrm{E}) \cup First(\mathrm{F})$
  - $First(\ (\mathrm{E})+\ \mathrm{F}\ ) = First(\mathrm{E})$ se todas as strings derivadas de E não são vazias
  - $First(\ (\mathrm{E})+\ \mathrm{F}\ ) = First(\mathrm{E}) \cup First(\mathrm{F})$ se algumas strings derivadas de E são vazias
  - $First(\ (\mathrm{E})?\ \mathrm{F}\ ) = First(\mathrm{E}) \cup First(\mathrm{F})$

Estratégia para Computar os Conjuntos $First$

- Usar uma abordagem de baixo para cima, ou seja, iniciar com regras mais simples e trabalhar em direção às regras mais complicadas (compostas).

Um Exemplo Usando Notação de CFGs

Material Prof. Daniel Lucrédio (UFSCar)

- Gramática

$S \rightarrow A\ B$

$A \rightarrow a\ A \mid a \mid d$

$B \rightarrow b\ B \mid c \mid A \mid Cd$

$C \rightarrow x \mid y \mid \varepsilon$

$D \rightarrow \varepsilon$

$First(abcd) = \{ a \}$

$First(ABC) = \{ a, d \}$

$First(AxCd) = \{ a, d \}$

$First(BzSB) = \{ b, c, a, d, x, y \}$

$First(CzSB) = \{ x, y, z \}$

$First(SABC) = \{ a, d \}$

$First(C) = \{ x, y, \varepsilon \}$

$First(DAB) = \{ a, d \}$

$First(DCe) = \{ x, y, e \}$

$First(DC) = \{ x, y, \varepsilon \}$

Um Exemplo Usando Notação de CFGs

Material Prof. Daniel Lucrédio (UFSCar)

- Gramática

$S \rightarrow A\ B$

$A \rightarrow a\ A \mid a \mid d$

$B \rightarrow b\ B \mid c \mid A \mid Cd$

$C \rightarrow x \mid y \mid \varepsilon$

$D \rightarrow \varepsilon$

a é terminal

$First(abcd) = \{ a \}$

$First(ABC) = \{ a, d \}$

$First(AxCd) = \{ a, d \}$

$First(BzSB) = \{ b, c, a, d, x, y \}$

$First(CzSB) = \{ x, y, z \}$

$First(SABC) = \{ a, d \}$

$First(C) = \{ x, y, \varepsilon \}$

$First(DAB) = \{ a, d \}$

$First(DCe) = \{ x, y, e \}$

$First(DC) = \{ x, y, \varepsilon \}$

Um Exemplo Usando Notação de CFGs

Material Prof. Daniel Lucrédio (UFSCar)

- Gramática

$S \rightarrow A\ B$

$A \rightarrow a\ A \mid a \mid d$

$B \rightarrow b\ B \mid c \mid A \mid Cd$

$C \rightarrow x \mid y \mid \varepsilon$

$D \rightarrow \varepsilon$

$First$ [A não tem ε à direita]

$First(ABC) = \{ a, d \}$

$First(AxCd) = \{ a, d \}$

$First(BzSB) = \{ b, c, a, d, x, y \}$

$First(CzSB) = \{ x, y, z \}$

$First(SABC) = \{ a, d \}$

$First(C) = \{ x, y, \varepsilon \}$

$First(DAB) = \{ a, d \}$

$First(DCe) = \{ x, y, e \}$

$First(DC) = \{ x, y, \varepsilon \}$

Um Exemplo Usando Notação de CFGs

Material Prof. Daniel Lucrédio (UFSCar)

- Gramática

$S \rightarrow A\ B$

$A \rightarrow a\ A \mid a \mid d$

$B \rightarrow b\ B \mid c \mid A \mid Cd$

$C \rightarrow x \mid y \mid \varepsilon$

$D \rightarrow \varepsilon$

$First(abcd) = \{ a \}$

$First$ [illegible]

A não tem ε à direita

$First(AxCd) = \{ a, d \}$

$First(BzSB) = \{ b, c, a, d, x, y \}$

$First(CzSB) = \{ x, y, z \}$

$First(SABC) = \{ a, d \}$

$First(C) = \{ x, y, \varepsilon \}$

$First(DAB) = \{ a, d \}$

$First(DCe) = \{ x, y, e \}$

$First(DC) = \{ x, y, \varepsilon \}$

Um Exemplo Usando Notação de CFGs

Material Prof. Daniel Lucrédio (UFSCar)

- Gramática

$S \rightarrow A\ B$

$A \rightarrow a\ A \mid a \mid d$

$B \rightarrow b\ B \mid c \mid A \mid Cd$

$C \rightarrow x \mid y \mid \varepsilon$

$D \rightarrow \varepsilon$

$First(abcd) = \{ a \}$

$First(ABC) = \{ a, d \}$

$First$ [illegible]

B não tem ε à direita e tem A e C

$First(BzSB) = \{ b, c, a, d, x, y \}$

$First(CzSB) = \{ x, y, z \}$

$First(SABC) = \{ a, d \}$

$First(C) = \{ x, y, \varepsilon \}$

$First(DAB) = \{ a, d \}$

$First(DCe) = \{ x, y, e \}$

$First(DC) = \{ x, y, \varepsilon \}$

Um Exemplo Usando Notação de CFGs

Material Prof. Daniel Lucrédio (UFSCar)

- Gramática

$S \rightarrow A\ B$

$A \rightarrow a\ A \mid a \mid d$

$B \rightarrow b\ B \mid c \mid A \mid Cd$

$C \rightarrow x \mid y \mid \varepsilon$

$D \rightarrow \varepsilon$

$First(abcd) = \{ a \}$

$First(ABC) = \{ a, d \}$

$First(AxCd) = \{ a, d \}$

F

C tem ε à direita, mas é seguido de z

$First(CzSB) = \{ x, y, z \}$

$First(SABC) = \{ a, d \}$

$First(C) = \{ x, y, \varepsilon \}$

$First(DAB) = \{ a, d \}$

$First(DCe) = \{ x, y, e \}$

$First(DC) = \{ x, y, \varepsilon \}$

Um Exemplo Usando Notação de CFGs

Material Prof. Daniel Lucrédio (UFSCar)

- Gramática

$S \rightarrow A\ B$

$A \rightarrow a\ A \mid a \mid d$

$B \rightarrow b\ B \mid c \mid A \mid Cd$

$C \rightarrow x \mid y \mid \varepsilon$

$D \rightarrow \varepsilon$

$First(abcd) = \{ a \}$

$First(ABC) = \{ a, d \}$

$First(AxCd) = \{ a, d \}$

$First(BzSB) = \{ b, c, a, d, x, y \}$

$First$ [illegible]

A não tem ε à direita

$First(SABC) = \{ a, d \}$

$First(C) = \{ x, y, \varepsilon \}$

$First(DAB) = \{ a, d \}$

$First(DCe) = \{ x, y, e \}$

$First(DC) = \{ x, y, \varepsilon \}$

Um Exemplo Usando Notação de CFGs

Material Prof. Daniel Lucrédio (UFSCar)

- Gramática

$S \rightarrow A\ B$

$A \rightarrow a\ A \mid a \mid d$

$B \rightarrow b\ B \mid c \mid A \mid Cd$

$C \rightarrow x \mid y \mid \varepsilon$

$D \rightarrow \varepsilon$

$First(abcd) = \{ a \}$

$First(ABC) = \{ a, d \}$

$First(AxCd) = \{ a, d \}$

$First(BzSB) = \{ b, c, a, d, x, y \}$

$First(CzSB) = \{ x, y, z \}$

$First$

C tem ε à direita e está sozinho

$First(C) = \{ x, y, \varepsilon \}$

$First(DAB) = \{ a, d \}$

$First(DCe) = \{ x, y, e \}$

$First(DC) = \{ x, y, \varepsilon \}$

Um Exemplo Usando Notação de CFGs

Material Prof. Daniel Lucrédio (UFSCar)

- Gramática

$S \rightarrow A\ B$

$A \rightarrow a\ A \mid a \mid d$

$B \rightarrow b\ B \mid c \mid A \mid Cd$

$C \rightarrow x \mid y \mid \varepsilon$

$D \rightarrow \varepsilon$

$First(abcd) = \{ a \}$

$First(ABC) = \{ a, d \}$

$First(AxCd) = \{ a, d \}$

$First(BzSB) = \{ b, c, a, d, x, y \}$

$First(CzSB) = \{ x, y, z \}$

$First(SABC) = \{ a, d \}$

F

D tem ε à direita, mas é seguido de A

$First(DAB) = \{ a, d \}$

$First(DCe) = \{ x, y, e \}$

$First(DC) = \{ x, y, \varepsilon \}$

Um Exemplo Usando Notação de CFGs

Material Prof. Daniel Lucrédio (UFSCar)

- Gramática

$S \rightarrow A\ B$

$A \rightarrow a\ A \mid a \mid d$

$B \rightarrow b\ B \mid c \mid A \mid Cd$

$C \rightarrow x \mid y \mid \varepsilon$

$D \rightarrow \varepsilon$

$First(abcd) = \{ a \}$

$First(ABC) = \{ a, d \}$

$First(AxCd) = \{ a, d \}$

$First(BzSB) = \{ b, c, a, d, x, y \}$

$First(CzSB) = \{ x, y, z \}$

$First(SABC) = \{ a, d \}$

Fi

Fi

D tem ε à direita, mas é seguido de C, que tem ε, e que é seguido de e

$First(DCe) = \{ x, y, e \}$

$First(DC) = \{ x, y, \varepsilon \}$

Um Exemplo Usando Notação de CFGs

Material Prof. Daniel Lucrédio (UFSCar)

- Gramática

$S \rightarrow A\ B$

$A \rightarrow a\ A \mid a \mid d$

$B \rightarrow b\ B \mid c \mid A \mid Cd$

$C \rightarrow x \mid y \mid \varepsilon$

$D \rightarrow \varepsilon$

$First(abcd) = \{ a \}$

$First(ABC) = \{ a, d \}$

$First(AxCd) = \{ a, d \}$

$First(BzSB) = \{ b, c, a, d, x, y \}$

$First(CzSB) = \{ x, y, z \}$

$First(SABC) = \{ a, d \}$

$First(C) = \{ x, y, \varepsilon \}$

Fi

Fi

$First(DC) = \{ x, y, \varepsilon \}$

D tem ε à direita, mas é seguido de C, que tem ε

Outro Exemplo Usando Notação de CFGs

- Gramática

$$E \rightarrow T\ E'$$

$$E' \rightarrow +\ T\ E' \mid \varepsilon$$

$$T \rightarrow F\ T'$$

$$\mathrm{T}' \rightarrow *\ F\ T' \mid \varepsilon$$

$$\mathrm{F} \rightarrow (E) \mid id$$

$$First(E) = \{\ (,\ id\ \}$$

$$First(T) = \{\ (,\ id\ \}$$

$$First(F) = \{\ (,\ id\ \}$$

$$First(E') = \{\ +,\ \varepsilon\ \}$$

$$First(T') = \{\ *,\ \varepsilon\ \}$$

Exemplos de Conjuntos $First$ da Linguagem CPRL

- constDecl = "const" constId ":=" literal ";" .
  $First$(constDecl) = { "const" }

- varDecl = "var" identifiers ":" typeName ";" .
  $First$(varDecl) = { "var" }

- arrayTypeDecl = "type" typeId "=" "array" ... ";" .
  $First$(arrayTypeDecl) = { "type" }

Exemplos de Conjuntos $First$ da Linguagem CPRL

- `initialDecl = constDecl | varDecl | arrayTypeDecl .`
  $First$(`initialDecl`) = { `"const"`, `"var"`, `"type"` }

- `statementPart = "begin" statements "end" .`
  $First$(`statementPart`) = { `"begin"` }

- `loopStmt = ( "while" booleanExpr )? "loop" ... ";" .`
  $First$(`loopStmt`) = { `"while"`, `"loop"` }

Conjuntos $Follow$ $(Seguidores)$

- O conjunto de todos os símbolos terminais que seguem imediatamente após a expressão sintática $\boldsymbol{A}$, em alguma **forma sentencial**, é denotado por $Follow(\boldsymbol{A})$;
- Ou seja, é o conjunto de terminais $\boldsymbol{a}$, tal que existe uma derivação na forma $\boldsymbol{S} \overset{*}{\Rightarrow} \alpha \boldsymbol{A a} \beta$
  - $Follow(\boldsymbol{A}) = \{\, \boldsymbol{a} \mid \boldsymbol{S} \overset{*}{\Rightarrow} \alpha \boldsymbol{A a} \beta \,\}$
- A expressão sintática $\boldsymbol{A}$ representa obrigatoriamente um símbolo não-terminal;
- Se $\boldsymbol{A}$ é o símbolo mais à direita em alguma forma sentencial, então \$ (símbolo de marcação de fim da entrada, análogo ao EOF), está em $Follow(\boldsymbol{A})$;
- Entender os conjuntos $Follow$ é importante não somente para o desenvolvimento do *parser*, mas também para recuperação de erros, pois usaremos $Follow(A)$ durante a recuperação de erros quando houver a tentativa de análise de A. Para computar $Follow(A)$, devemos analisar todas as regras que referenciam A;
- A computação dos conjuntos $Follow$ pode ser mais complicada do que a computação dos conjuntos $First$ 😂😅😳

Regras para Computar os Conjuntos $Follow$

- Computando $Follow(A)$:
  - Insira \$ em $Follow(S)$, onde S é o símbolo inicial e \$ é o símbolo de marcação de fim de entrada;
  - Se há uma produção na forma $A \Rightarrow \alpha B \beta$, então todos os elementos em $First(\beta)$, com exceção de ε, são inseridos em $Follow(B)$;
  - Se há uma produção na forma $A \Rightarrow \alpha B$, ou uma produção na forma $A \Rightarrow \alpha B \beta$ onde $First(\beta)$ contém ε (ou seja, $\beta \Rightarrow \varepsilon$), então todos os elementos em $Follow(A)$ estarão em $Follow(B)$.

Regras para Computar os Conjuntos $Follow$

- Computando $Follow(T)$:
  - Considere todas as produções similares às abaixo:
    - $N = S\ T\ U$.
    - $N = S\ (\ T\ )*\ U$.
    - $N = S\ (\ T\ )?\ U$.
    - $Follow(T)$ inclui $First(U)$;
    - Se U puder ser vazio, então $Follow(T)$ também inclui $Follow(N)$;
  - Considere todas as produções similares às abaixo:
    - $N = S\ T$.
    - $N = S\ (\ T\ )*$.
    - $N = S\ (\ T\ )?$.
    - Em todos esses casos, $Follow(T)$ inclui $Follow(N)$;
    - Se T ocorre na forma $(\ T\ )*$ ou $(\ T\ )+$, então $Follow(T)$ inclui $First(T)$.

Estratégia para Computar os Conjuntos $Follow$

- Usar uma abordagem de cima para baixo, ou seja, iniciar com a primeira regra, a do símbolo inicial, e trabalhar em direção às regras mais simples.

Um Exemplo, Usando Notação de CFGs

Material Prof. Daniel Lucrédio (UFSCar)

- Gramática

$S \rightarrow A\ B$

$A \rightarrow a\ A \mid a \mid d$

$B \rightarrow b\ B \mid c \mid A \mid Cd$

$C \rightarrow x \mid y \mid \varepsilon$

$D \rightarrow \varepsilon$

$Follow(S) = \{\$\}$

$Follow(A) = \{b, c, a, d, x, y, \$\}$

$Follow(B) = \{\$\}$

$Follow(C) = \{d\}$

$Follow(D) = \{\}$

vazio, pois como D é inalcançável, não há seguidores de D

Outro Exemplo Usando Notação de CFGs

- Gramática

$E \rightarrow T\ E'$

$E' \rightarrow +\ T\ E' \mid \varepsilon$

$T \rightarrow F\ T'$

$\mathrm{T}' \rightarrow *\ F\ T' \mid \varepsilon$

$\mathrm{F} \rightarrow (E) \mid id$

$Follow(E) = \{\ ),\ \$\ \}$

$Follow(E') = \{\ ),\ \$\ \}$

$Follow(T) = \{\ +,\ ),\ \$\ \}$

$Follow(T') = \{\ +,\ ),\ \$\ \}$

$Follow(F) = \{\ +,\ *,\ ),\ \$\ \}$

Exemplos de Conjuntos $Follow$ da Linguagem CPRL

- O que pode seguir uma `initialDecl`?
  - A partir da regra:

    ```
    initialDecls = ( initialDecl )* .
    ```

    Sabemos que qualquer `initialDecl` pode seguir uma `initialDecl`, então o conjunto $Follow$ para `initialDecl` inclui o conjunto $First$ de `initialDecl`, ou seja, `"const"`, `"var"` e `"type"`;
  - Das regras:

    ```
    declarativePart = initialDecls subprogramDecls .

    subprogramDecls = ( subprogramDecl )* .

    subprogramDecl = procedureDecl | functionDecl .
    ```

    Sabemos que uma `procedureDecl` ou uma `functionDecl` pode seguir uma `initialDecl`, então o conjunto $Follow$ para `initialDecl` inclui `"procedure"` e `"function"`.

Exemplos de Conjuntos $Follow$ da Linguagem CPRL

- Das regras:

```
program = declarativePart statementPart "." .

statementPart = "begin" statements "end" .
```

Sabemos que `statementPart` pode seguir uma `initialDecl`, então o conjunto $Follow$ de `initialDecl` inclui `"begin"`;

- Conclusão:

$Follow($`initialDecl`$) = \{$ `"const"`, `"var"`, `"type"`, `"procedure"`, `"function"`, `"begin"` $\}$

Análise Descendente Recursiva
Refinamento 5

- Um fator sintático na forma $(E) *$ é reconhecido pelo seguinte algoritmo:

```
enquanto o símbolo atual estiver em First(E):
    aplique o algoritmo que reconhece E
fim enquanto
```

- **Restrição Gramatical 1**: $First(E)$ e $Follow((E) *)$ têm que ser disjuntos nesse contexto, ou seja:

$$First(E) \cap Follow((E) *) = \emptyset$$

- Porque?

Análise Descendente Recursiva
Refinamento 5 (exemplo)

- Considere a regra para `initialDecls`:

```
initialDecls = ( initialDecl )* .
```

- O método da CPRL para analisar `initialDecls` é:

```
public void parseInitialDecls() {
    while ( scanner.getSymbol() == Symbol.constRW ||
            scanner.getSymbol() == Symbol.varRW   ||
            scanner.getSymbol() == Symbol.typeRW ) {
        parseInitialDecl();
    }
}
```

- Na CPRL, os símbolos `"const"`, `"var"`, e `"type"` não podem seguir `initialDecls`. Quais podem?

Métodos Auxiliares na Classe `Symbol`

- A classe Symbol fornece vários métodos auxiliares para testar as propriedades dos símbolos:

```
public boolean isReservedWord()
public boolean isInitialDeclStarter()
public boolean isSubprogramDeclStarter()
public boolean isStmtStarter()
public boolean isLogicalOperator()
public boolean isRelationalOperator()
public boolean isAddingOperator()
public boolean isMultiplyingOperator()
public boolean isLiteral()
public boolean isExprStarter()
```

Método isStmtStarter()

```
/**
 * Retorna true caso o símbolo possa iniciar uma instrução.
 */
public boolean isStmtStarter() {
    return this == Symbol.exitRW
        || this == Symbol.identifier
        || this == Symbol.ifRW
        || this == Symbol.loopRW
        || this == Symbol.whileRW
        || this == Symbol.readRW
        || this == Symbol.writeRW
        || this == Symbol.writelnRW
        || this == Symbol.returnRW;
}
```

Método isInitialDeclStarter()

```
/**
 * Retorna true caso o símbolo possa iniciar uma declaração inicial.
 */
public boolean isInitialDeclStarter() {
    return this == Symbol.constRW
        || this == Symbol.varRW
        || this == Symbol.typeRW;
}
```

Usando os Métodos Auxiliares da Classe `Symbol`

- Usando os métodos auxiliares da classe `Symbol`, podemos escrever o método `parseInitialDecls()` da seguinte forma:

```
public void parseInitialDecls() {
    while ( scanner.getSymbol().isInitialDeclStarter() ) {
        parseInitialDecl();
    }
}
```

Análise Descendente Recursiva
Refinamento 6

- Dado que um fator sintático na forma (E) + é equivalente a E (E) $*$, um fator sintático na forma (E) + é reconhecido pelo seguinte algoritmo:

```
aplique o algoritmo que reconhece E
enquanto o símbolo atual estiver em First(E):
    aplique o algoritmo que reconhece E
fim enquanto
```

- De forma equivalente, o algoritmo para reconhecer (E) + pode ser escrito usando um laço com teste no final:

```
faça
    aplique o algoritmo que reconhece E
    saia quando o símbolo atual não estiver em First(E)
fim faça
```

Análise Descendente Recursiva
Refinamento 6

- Em Java, como sabemos, a estrutura de repetição que realiza o teste no fim da execução do bloco de código é o `do-while`. Sendo assim, o algoritmo implementado em Java se parecerá com algo assim:

> *faça*
> *aplique o algoritmo que reconhece* E
> *enquanto o símbolo atual estiver em* $First(E)$

- **Restrição Gramatical 2**: Se E pode gerar uma string vazia, então $First(E)$ e $Follow((E)+)$ têm que ser disjuntos nesse contexto, ou seja:

$$First(E) \cap Follow((E)+) = \emptyset$$

- Porque?

Análise Descendente Recursiva
Refinamento 7

- Um fator sintático na forma (E)? é reconhecido pelo seguinte algoritmo:

> *se o símbolo atual estiver em* $First(E)$*, então*
> *aplique o algoritmo que reconhece* E
> *fim se*

- **Restrição Gramatical 3**: $First(E)$ e $Follow((E)?)$ têm que ser disjuntos nesse contexto, ou seja:

$$First(E) \cap Follow((E)?) = \emptyset$$

- Pela mesma razão anterior.

Método Auxiliar `matchCurrentSymbol()`

- O método `matchCurrentSymbol()` é similar ao método `match()`, com a exceção de que ele não possui parâmetros e não lança uma exceção. Ele simplesmente avança o *scanner*;
- O método `matchCurrentSymbol()` é usado quando já sabemos que o próximo símbolo do fluxo de entrada é o que queremos. Poderíamos usar o método `match()` para esse propósito, mas o método `matchCurrentSymbol()` é um pouco mais eficiente;
- Código do método `matchCurrentSymbol()`:

```
private void matchCurrentSymbol() throws IOException {
    scanner.advance();
}
```

Análise Descendente Recursiva
Refinamento 7 (exemplo)

- Considere a regra da instrução `exit`:

```
exitStmt = "exit" ( "when" booleanExpr )? ";" .
```

- O método de análise da instrução `exit` é:

```
public void parseExitStmt() {

    match( Symbol.exitRW );

    if ( scanner.getSymbol() == Symbol.whenRW ) {
        matchCurrentSymbol();
        parseExpression();
    }

    match( Symbol.semicolon );

}
```

verifica a cláusula `when`, que é opcional

um pouco mais eficiente que `match(Symbol.whenRW)`

Análise Descendente Recursiva
Refinamento 7 (exemplo)

- O conjunto $First$ da cláusula opcional `when` é simplesmente { `"when"` }, então usamos a palavra reservada `when` para nos dizer se devemos ou não processar a cláusula `when`;
- Para pensar:
  - Qual o conjunto $Follow$ para a cláusula opcional `when`?
  - Qual problema haveria se o mesmo contivesse a palavra reservada `"when"`?

Análise Descendente Recursiva
Refinamento 8

- Um fator sintático na forma $E \mid F$ é reconhecido pelo seguinte algoritmo:

```
se o símbolo atual estiver em First(E), então
    aplique o algoritmo que reconhece E
senão se o símbolo atual estiver em First(F), então
    aplique o algoritmo que reconhece F
senão
    erro de análise
fim se
```

- **Restrição Gramatical 4**: $First(E)$ e $First(F)$ têm que ser disjuntos nesse contexto, ou seja:

$$First(E) \cap First(F) = \emptyset$$

Análise Descendente Recursiva
Refinamento 8 (exemplo)

- Considere a regra da CPRL para `initialDecl`:

```
initialDecl = constDecl | varDecl | arrayTypeDecl .
```

- O método da CPRL de análise para `initialDecl` é:

```
public void parseInitialDecl() {
    if ( scanner.getSymbol() == Symbol.constRW ) {
        parseConstDecl();
    } else if ( scanner.getSymbol() == Symbol.varRW ) {
        parseVarDecl();
    } else if ( scanner.getSymbol() == Symbol.typeRW ) {
        parseArrayTypeDecl();
    } else {
        ...   // lançar uma InternalErrorException
    }
}
```

essa lógica também pode ser implementada usando um `switch`

Gramáticas $LL(1)$

- Se uma gramática satisfaz as restrições impostas pelas regras de análise apresentadas anteriormente, então essa gramática é chamada de **Gramática $LL(1)$**;
- A análise sintática descendente recursiva com um símbolo de *lookahead* só pode ser aplicada se a gramática for $LL(1)$;
  - *Left to Right with Leftmost Derivation;*
  - L: a leitura do arquivo fonte é feito da esquerda para a direita;
  - L: a descida na árvore de análise é feita da esquerda para a direita;
  - 1: um token de *lookahead*;
- Nem todas as gramáticas são $LL(1)$;
  - Por exemplo, qualquer gramática que tem recursão à esquerda não é $LL(1)$.

Gramáticas $LL(1)$

- Na prática, a sintaxe da maioria das linguagens de programação podem ser definidas, ou pelo menos ser bem aproximadas, por uma gramática $LL(1)$;
  - Por exemplo, realizar transformações na gramática, como eliminar recursão à esquerda;
- A expressão "descendente recursiva" refere-se ao fato que descemos (*top-down*) a árvore de análise usando chamadas à funções/métodos recursivos.

Análise Descendente Recursiva

- A parte "recursiva" da expressão "descendente recursiva", vem do uso das chamadas recursivas no *parser*, por exemplo, para analisar instruções de laço aninhadas:

```
parseLoop()          // chamado na análise do laço mais externo
   ...
   parseStatements()
      ...
      parseLoop()    // chamado na análise do laço mais interno
```

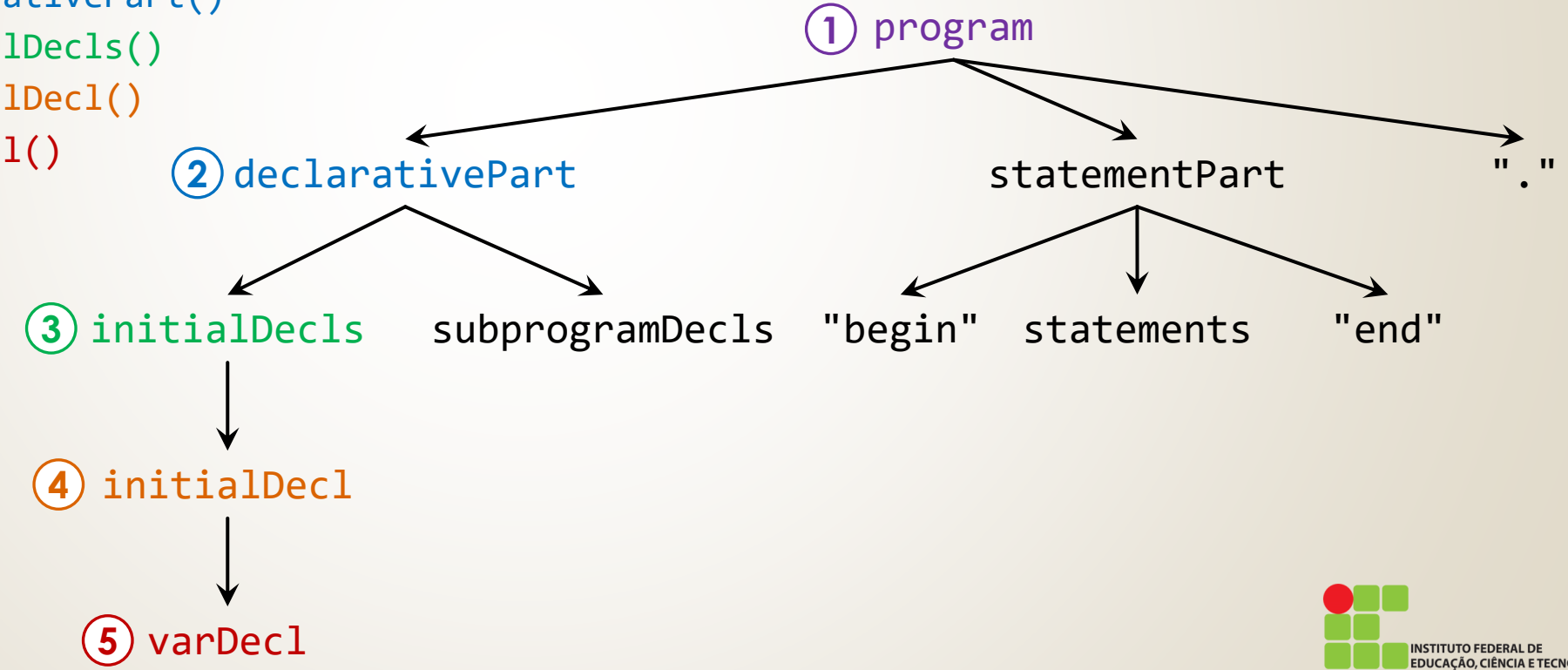
- Para a parte "descendente" da expressão "descendente recursiva", considere uma porção da árvore de análise para um programa simples escrito em CPRL:

```
var x : Integer;
begin
   ...
end.
```

Análise Descendente Recursiva

- Na árvore de análise abaixo, os números à esquerda dos nós correspondem à ordem de invocação dos métodos, ou seja, esses são os cinco primeiros métodos de análise durante a análise do programa.

```
parseProgram()
parseDeclarativePart()
parseInitialDecls()
parseInitialDecl()
parseVarDecl()
```



Desenvolvimento do *Parser* da CPRL

- Haverá três grandes versões do *parser* para o nosso projeto do compilador:
  - **Versão 1 (Projeto 2):** reconhecimento da linguagem, baseada em uma gramática livre de contexto, com verificações mínimas em relação às restrições da linguagem;
  - **Versão 2 (Projeto 3):** inserção da recuperação de erros;
  - **Versão 3 (Projeto 4):** inserção da geração das árvores sintáticas abstratas.

Desenvolvimento do *Parser* da CPRL

Versão 1: Reconhecimento da Linguagem (**Projeto 2**)

- Usar os refinamentos discutidos anteriormente;
- Verificar se as restrições gramaticais, em termos dos conjuntos $First$ e $Follow$, são satisfeitas pela gramática da CPRL;
- Usar a gramática para desenvolver a versão 1 do *parser*:
  - Requererá a análise da gramática;
  - Computação dos conjuntos $First$ e $Follow$.

Variáveis versus Valores Nomeados

- Da perspectiva da gramática, não há distinção real entre uma variável e um valor nomeado:

```
variable = ( varId | paramId ) ( "[" expression "]" )* .
namedValue = variable .
```

- Ambos são analisados de forma similar, mas faremos uma distinção baseada no contexto onde os identificadores aparecem;
- Por exemplo, considere a instrução de atribuição:

```
x := y;
```

O identificador "x" representa a variável e o identificador "y" representa o valor nomeado.

Variáveis versus Valores Nomeados

- Simplificando, um identificador é uma variável se estiver situado do lado esquerdo da uma instrução de atribuição e será considerado um valor nomeado se for usado em uma expressão;
- A distinção entre uma variável e um valor nomeado se tornará importante quando formos lidar com os tópicos relativos à recuperação de erros e geração de código, pois serão tratados de forma diferente.

Lidando com as Limitações da Gramática

- Da forma que está, a gramática da CPRL "não é bem" do tipo $LL(1)$;
- Exemplo: Realizando a análise da regra `statement`:

```
statement = assignmentStmt | ifStmt | loopStmt | exitStmt
          | readStmt | writeStmt | writelnStmt
          | procedureCallStmt | returnStmt .
```

- Usamos o símbolo de *lookahead* para selecionar o método de análise:
  - `"if"` → analisar uma instrução "if"
  - `"while"` → analisar uma instrução while
  - `"loop"` → analisar uma instrução de laço
  - `identifier` → analisar uma instrução de atribuição ou uma instrução de chamada de função, mas qual delas?
- Um identificador está no conjunto $First$ tanto de uma instrução de atribuição quanto numa instrução de chamada de procedimento.

Lidando com as Limitações da Gramática

- Um problema similar existe quando se analisa uma regra `factor`:

```
factor = "not" factor | constValue | namedValue
       | functionCall | "(" expression ")" .
```

- Um identificador está no conjunto $First$ de `constValue`, `namedValue`, e `functionCall`.

Possíveis Soluções

1. Usar um token de *lookahead* adicional ($LL(2)$):
   - Se um símbolo que segue um identificador é um `"["` ou `":="`, analise uma instrução de atribuição;
   - Se é um `"("` ou `";"`, analise uma instrução de chamada de procedimento;
2. Reprojetar ou fatorar a gramática, por exemplo, trocar

```
s = i x | i y .
```

   por

```
s = i ( x | y ) .
```

3. Usar uma tabela de identificadores (tabela de símbolos) para armazenar informações sobre como o identificador foi declarado e então, mais tarde, usar essa informação para determinar se o identificador é uma constante, uma variável, o nome de um procedimento etc. Usaremos essa abordagem!

Classe `IdTable`
Versão 1

- Criaremos uma versão preliminar da classe `IdTable` para nos ajudar a acompanhar os identificadores que foram declarados e nos auxiliar com as regras básicas de escopo na análise de restrições;
- Tipos dos identificadores (enumeração `IdType`):

```
public enum IdType {
    constantId,
    variableId,
    arrayTypeId,
    procedureId,
    functionId;
}
```

- A classe `IdTable` será estendida nas próximas versões do projeto de modo a executar uma análise mais completa das regras de escopo da CPRL.

Exemplo de Procedimento (Escopo)

```
var x : Integer;
var y : Integer;

procedure p is
   var x : Integer;
   var n : Integer;
begin
   x := 5;      // qual x?
   n := y;      // qual y?
end p;

begin
   x := 8;     // qual x?
end.
```

Lidando com Escopos Dentro da Classe `IdTable`

- Variáveis e constantes podem ser declaradas no nível do programa ou no nível de um subprograma, introduzindo assim o conceito de escopo;
- A classe `IdTable` precisará buscar por nomes tanto dentro do escopo atual e possivelmente em escopos mais externos;
- A classe `IdTable` é implementada como uma pilha de mapas, associando a string dos identificadores com seu tipo (`IdType`);
  - Quando um novo escopo é aberto, um novo mapa é empilhado;
  - A busca por uma declaração envolve pesquisar dentro do nível atual, nas declarações contidas no mapa no topo da pilha, e então dentro de escopos mais externos, ou seja, em mapas abaixo do topo da pilha.

Métodos Importantes da Classe `IdTable`

```
 /**
  * Abre um novo escopo para identificadores.
  */
 public void openScope()

 /**
  * Fecha o escopo mais interno.
  */
 public void closeScope()

 /**
  * Insere um token e seu tipo no nível de escopo atual.
  * @throws ParserException se o identificador do token já estiver definido
  *                         dentro do escopo atual.
  */
 public void add( Token idToken, IdType idType ) throws ParserException

/**
 * Retorna o IdType associado com o texto do token do identificador. Retorna null
 * se o identificador não for encontrado. Esse método busca em escopos mais externos
 * caso necessário.
 */
public IdType get( Token idToken )
```

Adicionando Declarações na `IdTable`

- Quando um identificador é declarado, o *parser* tentará adicionar o token do identificador e seu tipo à tabela de identificadores dentro do escopo atual;
  - Lança uma exceção caso um identificador com o mesmo nome, ou seja, o mesmo texto do token, já tiver sido previamente declarado no escopo atual;
- Por exemplo, no método `parseConstDecl()`

```
idTable.add( constId, IdType.constantId );
```

Lança uma `ParserException` se o identificador já estiver definido no escopo atual

Usando uma `IdTable` para Verificar Ocorrências Aplicadas dos Identificadores

- Quando um identificador é encontrado em uma instrução de um programa ou subprograma, o *parser* vai:
  - Verificar se o identificador foi declarado;
  - Usar a informação sobre como o identificador foi declarado para facilitar sua análise correta, por exemplo, você não pode atribuir um valor à um identificador que foi declarado como uma constante.

Usando uma `IdTable` para Verificar Ocorrências Aplicadas dos Identificadores

Exemplo

```
// no método parseFactor()
} else if ( scanner.getSymbol() == Symbol.identifier ) {

    // lida com os identificadores baseando-se se eles foram
    // declarados como variáveis, constantes ou funções.
    Token idToken = scanner.getToken();
    IdType idType = idTable.get( idToken );

    if ( idType != null ) {
        if ( idType == IdType.constantId ) {
            parseConstValue();
        } else if ( idType == IdType.variableId ) {
            parseNamedValue();
        } else if ( idType == IdType.functionId ) {
            parseFunctionCall();
        } else {
            throw error( "Identifier \"" + scanner.getToken() +
                         "\" is not valid as an expression." );
        }
    } else {
        throw error( "Identifier \"" + scanner.getToken() +
                     "\" has not been declared." );
    }
}
```

Analisando uma Declaração de Procedimento

Exemplo

```
// procedureDecl = "procedure" procId ( formalParameters )?
//                 "is" initialDecls statementPart procId ";" .
match( Symbol.procedureRW );
Token procId = scanner.getToken();
match( Symbol.identifier );
idTable.add( procId, IdType.procedureId );
idTable.openScope();

if ( scanner.getSymbol() == Symbol.leftParen ) {
    parseFormalParameters();
}

match( Symbol.isRW );
parseInitialDecls();
parseStatementPart();
idTable.closeScope();
Token procId2 = scanner.getToken();
match( Symbol.identifier );

if ( !procId.getText().equals( procId2.getText() ) ) {
    throw error(procId2.getPosition(),
                "Procedure name mismatch.");
}
match( Symbol.semicolon );
```

Note que o nome do procedimento é definido no escopo mais externo (do programa), mas seus parâmetros e declarações iniciais são definidos dentro do escopo do procedimento.

Analisando uma Declaração de Procedimento

Exemplo

```
// procedureDecl = "procedure" procId ( formalParameters )?
//                 "is" initialDecls statementPart procId ";" .
match( Symbol.procedureRW );
Token procId = scanner.getToken();
match( Symbol.identifier );
idTable.add( procId, IdType.procedureId );
idTable.openScope();

if ( scanner.getSymbol() == Symbol.leftParen ) {
    parseFormalParameters();
}

match( Symbol.isRW );
parseInitialDecls();
parseStatementPart();
idTable.closeScope();
Token procId2 = scanner.getToken();
match( Symbol.identifier );

if ( !procId.getText().equals( procId2.getText() ) ) {
    throw error(procId2.getPosition(),
                "Procedure name mismatch.");
}
match( Symbol.semicolon );
```

Note também a verificação se os nomes dos procedimentos (`procId` e `procId2`) combinam. Tecnicamente, a garantia que os nomes dos procedimentos combinam vai além da análise sintática e representa mais uma verificação de restrição. Do ponto de vista da gramática livre de contexto, eles são apenas identificadores.

Analisando uma Instrução
Exemplo

```
Symbol symbol = scanner.getSymbol();

if ( symbol == Symbol.identifier ) {

    IdType idType = idTable.get( scanner.getToken() );
    if ( idType != null ) {
        if ( idType == IdType.variableId ) {
            parseAssignmentStmt();
        } else if ( idType == IdType.procedureId ) {
            parseProcedureCallStmt();
        } else {
            throw error(...);
        }
    } else {
        throw error(...);
    }

} else if ( symbol == Symbol.ifRW ) {
    parseIfStmt();
} else if ( symbol == Symbol.loopRW || symbol == Symbol.whileRW ) {
    parseLoopStmt();
} else if ( symbol == Symbol.exitRW ) {
    parseExitStmt();
...
```

Classe `ErrorHandler`

- Usada para que haja consistência no relatório de erros;
- Implementa o padrão de projeto *Singleton*;
- Para obter uma instância de `ErrorHandler` deve-se invocar:

```
ErrorHandler.getInstance();
```

Métodos Chave da Classe `ErrorHandler`

```
/**
 * Reporta um erro. Para o processo de compilação se
 * uma quantidade máxima de erros for gerada.
 */
public void reportError( CompilerException e )

/**
 * Reporta o erro e termina a compilação.
 */
public void reportFatalError( Exception e )

/**
 * Reporta um aviso e continua a compilação.
 */
public void reportWarning( String warningMessage )
```

Usando a `ErrorHandler` para a **Versão 1 do *Parser* (Projeto 2)**

- A **Versão 1** (**Projeto 2**) do *parser* não implementará a recuperação de erros. Quando um erro é encontrado, o *parser* imprimirá a mensagem de erro e terminará a execução;
- De modo a suavizar a transição para a recuperação de erros para a próxima versão do *parser*, a maioria dos métodos de análise empacotarão a lógica básica em um bloco `try/catch`;
- Qualquer método de análise que invoque os métodos `match()` ou `add()` de `IdTable` precisarão ter um bloco `try/catch`;
- O relatório de erros será implementado dentro da cláusula `catch` do bloco `try/catch`.

Usando a `ErrorHandler` para a **Versão 1 do *Parser* (Projeto 2)**

```
public void parseAssignmentStmt() throws IOException {

    try {
        parseVariable();
        match( Symbol.assign );
        parseExpression();
        match( Symbol.semicolon );
    } catch ( ParserException e ) {
        ErrorHandler.getInstance().reportError( e );
        exit();
    }

}
```

As instruções de análise ficarão empacotadas dentro de um bloco `try/catch`

Essa abordagem nos fornecerá um alicerce que nós usaremos para a implementação da recuperação de erros

Implementando os Métodos `parseVariable()` e `parseNamedValue()`

- Para implementar os métodos `parseVariable()` e `parseNamedValue()` usaremos um método auxiliar que proverá uma lógica comum aos dois métodos;
- Esse método auxiliar não lidará com quaisquer exceções geradas pelo *parser* (`ParseException`), lançando-as ao método chamador para que possam ser manipuladas apropriadamente;
- Ume esboço desse método auxiliar, chamado de `parseVariableExpr()`, será mostrado a seguir;
- Ambos os métodos `parseVariable()` e `parseNamedValue()` invocam `parseVariableExpr()` com o intuito de auxiliar na análise da regra gramatical que trata de variáveis.

Método parseVariableExpr()

```
// analisa a regra gramatical abaixo:
// variable = ( varId | paramId ) ( "[" expression "]" )* .
public void parseVariableExpr() throws IOException, ParserException {
    Token idToken = scanner.getToken();
    match( Symbol.identifier );
    IdType idType = idTable.get( idToken );

    if ( idType == null ) {
        String errorMsg = "Identifier \"" + idToken +
                          "\" has not been declared.";
        throw error( idToken.getPosition(), errorMsg );
    } else if ( idType != IdType.variableId ) {
        String errorMsg = "Identifier \"" + idToken +
                          "\" is not a variable.";
        throw error( idToken.getPosition(), errorMsg );
    }

    while ( scanner.getSymbol() == Symbol.leftBracket ) {
        matchCurrentSymbol();
        parseExpression();
        match( Symbol.rightBracket );
    }
}
```

Método `parseVariable()`

- O método `parseVariable()` simplesmente invoca o método auxiliar para analisar sua regra gramatical.

```
public void parseVariable() throws IOException {
    try {
        parseVariableExpr();
    } catch ( ParserException e ) {
        ErrorHandler.getInstance().reportError(e);
        exit();
    }
}
```

- O método `parseNamedValue()` é implementado de forma similar.

Breve Discussão Sobre Outros Métodos/Técnicas para Análise Sintática

- Analisadores Descendentes:
  - Análise Sintática Preditiva:
    - Tabela preditiva;
    - Sem recursão;
    - Uso de pilhas;
  - $LL(1)$: facilidade de implementação, resolução de não-determinismos (ambiguidade, fatoração à esquerda);
  - $LL(k)$: mesma construção de $LL(1)$, mas com k símbolos de *lookahead*;
  - $LL(*)$: *lookahead* variável, sem fatoração à esquerda (gramáticas mais intuitivas);
  - $ALL(*)$ (*Adaptative* $LL(*)$): qualquer gramática não-recursiva à esquerda;

$$LL(1) \rightarrow LL(k) \rightarrow LL(*) \rightarrow ALL(*)$$

Breve Discussão Sobre Outros Métodos/Técnicas para Análise Sintática

- Analisadores Ascendentes:
  - Processo de inferência;
  - Construção da árvore de análise sintática de baixo para cima;
  - $\boldsymbol{LR}(k)$: *Left to Right with Rightmost Derivation*, leitura da esquerda para a direita, produzindo derivações mais à direita ao reverso (inferência recursiva);
  - Difícil de implementar (usar geradores).

Bibliografia

MOORE JR., J. I. **Introduction to Compiler Design: an Object Oriented Approach Using Java**. 2. ed. [s.l.]:SoftMoore Consulting, 2020. 284 p.

AHO, A. V.; LAM, M. S.; SETHI, R. ULLMAN, J. D. **Compiladores: Princípios, Técnicas e Ferramentas**. 2. ed. São Paulo: Pearson, 2008. 634 p.

COOPER, K. D.; TORCZON, L. **Construindo Compiladores**. 2. ed. Rio de Janeiro: Campus Elsevier, 2014. 656 p.

JOSÉ NETO, J. **Introdução à Compilação**. São Paulo: Elsevier, 2016. 307 p.

SANTOS, P. R.; LANGOLOIS, T. **Compiladores: da teoria à prática**. Rio de Janeiro: LTC, 2018. 341 p.