Aula: Parser Preditivo Tabular LL(1)

Objetivo da Aula

- Compreender-se o funcionamento do Parser Preditivo Tabular (LL(1)).
- Aprender-se a construção da tabela de análise a partir de uma gramática.
- Aplicar-se o parser em um exemplo prático.
- Analisar-se implementações em pseudocódigo e Python.

---

1. Introdução

“Será abordada uma técnica de parsing top-down conhecida como **Parser Preditivo Tabular**, ou LL(1). Diferentemente do Parser Preditivo Recursivo, que utiliza funções recursivas, o LL(1) emprega uma tabela para orientar a análise sintática. O termo ‘LL’ indica que a análise é realizada da esquerda para a direita (Left-to-right) e constrói uma derivação mais à esquerda (Leftmost derivation), enquanto ‘(1)’ refere-se ao uso de apenas um símbolo de lookahead para decisões. Trata-se de uma abordagem eficiente e amplamente aplicada em compiladores reais.”

Razões para seu estudo:

- É caracterizada por sistematicidade e evita recursão excessiva.
- Prepara para a compreensão de ferramentas automáticas de geração de parsers.

---

2. Conceitos Fundamentais

O que é um Parser LL(1)?

- Define-se como um parser top-down que utiliza uma tabela para determinar a produção a ser aplicada.
- Exige-se uma gramática LL(1), isto é, livre de ambiguidades ou conflitos para o mesmo símbolo de lookahead.

Componentes do Parser

1. **Pilha**: Armazena os símbolos a serem processados.
2. **Tabela de Análise**: Relaciona não-terminais e terminais às produções correspondentes.
3. **Entrada**: Consiste na sequência de tokens a ser analisada.
4. **Símbolo de lookahead**: Representa o próximo token da entrada.

Funcionamento

- Inicia-se com o símbolo inicial na pilha.
- Para cada não-terminal no topo da pilha, consulta-se a tabela com base no lookahead e aplica-se a produção indicada.
- Caso o topo da pilha seja um terminal, compara-se com o lookahead e avança-se se houver correspondência.

Construção da Tabela: FIRST e FOLLOW

A tabela de análise LL(1) depende de dois conceitos essenciais: os conjuntos **FIRST** e **FOLLOW**. Esses conjuntos auxiliam na decisão sobre qual produção aplicar, com base no símbolo de entrada atual (lookahead). Será explicada sua definição e construção de forma detalhada.

FIRST(A): Definição e Finalidade

- **Definição**: O conjunto `FIRST(A)` inclui todos os **terminais** que podem ser o primeiro símbolo em uma derivação a partir do não-terminal `A`. Se `A` pode derivar o vazio (`ϵ`), `ϵ` também é incluído em `FIRST(A)`.
- **Finalidade**: Indica os símbolos de entrada que podem iniciar uma produção específica de `A`, sendo fundamental para o preenchimento da tabela LL(1), pois o lookahead orienta a escolha da produção.

FOLLOW(A): Definição e Finalidade

- **Definição**: O conjunto `FOLLOW(A)` abrange todos os **terminais** que podem aparecer imediatamente após o não-terminal `A` em uma derivação válida. O símbolo de fim de entrada (`$`) é incluído se `A` pode estar no final da derivação.
- **Finalidade**: Auxilia na decisão sobre produções que geram `ϵ`, pois, nesse caso, deve-se considerar os símbolos que seguem `A` (isto é, `FOLLOW(A)`) para preencher a tabela.

Construção dos Conjuntos FIRST e FOLLOW

Serão apresentados os passos para a construção desses conjuntos, com regras claras e exemplos baseados na gramática do exemplo prático:

```
E → T E'
E' → + T E' | ϵ
T → F T'
T' → * F T' | ϵ
F → ( E ) | id
```

Construção do Conjunto FIRST

Para cada não-terminal e produção, aplicam-se as seguintes regras:

1. **Se A é terminal**: `FIRST(A) = {A}`. (Não aplicável aqui, pois o cálculo é feito apenas para não-terminais.)
2. **Para uma produção $A \rightarrow X_1 X_2 \ldots X_n$**:

- Incluem-se todos os terminais de $FIRST(X_1)$ em $FIRST(A)$.
- Se X_1 pode derivar ϵ, adicionam-se os terminais de $FIRST(X_2)$ a $FIRST(A)$, prosseguindo-se até que um X_i não derive ϵ ou até o fim da produção.
- Se todos os X_i podem derivar ϵ, adiciona-se ϵ a $FIRST(A)$.

Passos para a gramática:

- **FIRST(F)**:
  - Produções: $F \rightarrow (E)$ e $F \rightarrow id$.
  - Para $F \rightarrow (E)$, o primeiro símbolo é $($ → Inclui-se $\{ (\}$ em $FIRST(F)$.
  - Para $F \rightarrow id$, o primeiro símbolo é id → Inclui-se $\{ id \}$ em $FIRST(F)$.
  - Resultado: $FIRST(F) = \{ (, id \}$.
- **FIRST(T')**:
  - Produções: $T' \rightarrow * F T'$ e $T' \rightarrow \epsilon$.
  - Para $T' \rightarrow * F T'$, o primeiro símbolo é $*$ → Inclui-se $\{ * \}$ em $FIRST(T')$.
  - Para $T' \rightarrow \epsilon$, adiciona-se ϵ → Inclui-se $\{ \epsilon \}$ em $FIRST(T')$.
  - Resultado: $FIRST(T') = \{ *, \epsilon \}$.
- **FIRST(T)**:
  - Produção: $T \rightarrow F T'$.
  - O primeiro símbolo é F, então inclui-se $FIRST(F) = \{ (, id \}$ em $FIRST(T)$.
  - T' segue, mas como F não deriva ϵ, encerra-se em F.
  - Resultado: $FIRST(T) = \{ (, id \}$.
- **FIRST(E')**:

- Produções: `E' → + T E'` e `E' → ϵ`.
- Para `E' → + T E'`, o primeiro símbolo é `+` → Inclui-se `{ + }` em `FIRST(E')`.
- Para `E' → ϵ`, adiciona-se `ϵ` → Inclui-se `{ ϵ }` em `FIRST(E')`.
- Resultado: `FIRST(E') = { +, ϵ }`.

- **FIRST(E)**:

  - Produção: `E → T E'`.
  - O primeiro símbolo é `T`, então inclui-se `FIRST(T) = { (, id }` em `FIRST(E)`.
  - `E'` segue, mas `T` não deriva `ϵ`, encerra-se em `T`.
  - Resultado: `FIRST(E) = { (, id }`.

Resumo dos FIRST:

- `FIRST(E) = { (, id }`
- `FIRST(E') = { +, ϵ }`
- `FIRST(T) = { (, id }`
- `FIRST(T') = { *, ϵ }`
- `FIRST(F) = { (, id }`

Construção do Conjunto FOLLOW

Para o cálculo de `FOLLOW`, aplicam-se estas regras:

1. **Para o símbolo inicial S**: Adiciona-se `$` a `FOLLOW(S)` (representa o fim da entrada).
2. **Para uma produção A → α B β**:
   - Incluem-se os terminais de `FIRST(β)` (exceto `ϵ`) em `FOLLOW(B)`.
3. **Para uma produção A → α B** ou **A → α B β** onde `β` pode derivar `ϵ` **:
   - Adiciona-se `FOLLOW(A)` a `FOLLOW(B)`.

O processo é iterativo, pois os conjuntos `FOLLOW` são interdependentes. Seguem os cálculos:

Passos para a gramática:

- **FOLLOW(E)**:
  - `E` é o símbolo inicial → Adiciona-se `$`.
  - Produção `F → ( E )`: Após `E`, aparece `)`, então inclui-se `{ ) }` em `FOLLOW(E)`.
  - Resultado: `FOLLOW(E) = { ), $ }`.
- **FOLLOW(E')**:
  - Produção `E → T E'`: Após `E'`, segue-se o que está em `FOLLOW(E)`, então inclui-se `FOLLOW(E) = { ), $ }` em `FOLLOW(E')`.
  - Resultado: `FOLLOW(E') = { ), $ }`.
- **FOLLOW(T)**:
  - Produção `E → T E'`: Após `T`, vem `E'`. Como `FIRST(E') = { +, ϵ }`, inclui-se `{ + }` em `FOLLOW(T)`. Como `ϵ` está em `FIRST(E')`, adiciona-se `FOLLOW(E) = { ), $ }`.
  - Produção `E' → + T E'`: Após `T`, vem `E'`, aplicando-se a mesma lógica: inclui-se `{ + }` e `FOLLOW(E') = { ), $ }`.
  - Resultado: `FOLLOW(T) = { +, ), $ }`.
- **FOLLOW(T')**:
  - Produção `T → F T'`: Após `T'`, segue-se o que está em `FOLLOW(T)`, então inclui-se `FOLLOW(T) = { +, ), $ }` em `FOLLOW(T')`.
  - Resultado: `FOLLOW(T') = { +, ), $ }`.

- **FOLLOW(F)**:
    - Produção `T → F T'`: Após `F`, vem `T'`. Como `FIRST(T') = { *, ϵ }`, inclui-se `{ * }`. Como `ϵ` está em `FIRST(T')`, adiciona-se `FOLLOW(T) = { +, ), $ }`.
    - Produção `T' → * F T'`: Após `F`, vem `T'`, aplicando-se a mesma lógica: inclui-se `{ * }` e `FOLLOW(T') = { +, ), $ }`.
    - Resultado: `FOLLOW(F) = { +, *, ), $ }`.

Resumo dos FOLLOW:

- `FOLLOW(E) = { ), $ }`
- `FOLLOW(E') = { ), $ }`
- `FOLLOW(T) = { +, ), $ }`
- `FOLLOW(T') = { +, ), $ }`
- `FOLLOW(F) = { +, *, ), $ }`

Uso de FIRST e FOLLOW na Tabela

Com os conjuntos definidos, preenche-se a tabela LL(1):

- Para cada produção `A → α`:
    - Se `t ∈ FIRST(α)` (e `t ≠ ϵ`), insere-se `A → α` em `[A, t]`.
    - Se `ϵ ∈ FIRST(α)`, insere-se `A → α` em `[A, t]` para todo `t ∈ FOLLOW(A)`.

---

3. Exemplo Prático

“Será construído um parser LL(1) para uma gramática simples e analisada uma entrada.”

Gramática

Considera-se a gramática para expressões aritméticas básicas:

```
E → T E'
E' → + T E' | ϵ
T → F T'
T' → * F T' | ϵ
F → ( E ) | id
```

Onde:

- `E` é o símbolo inicial (expressão).
- Terminais: `+`, `*`, `(`, `)`, `id`, `$` (fim da entrada).
- Não-terminais: `E`, `E'`, `T`, `T'`, `F`.

Passo 1: Calcular FIRST

(Resultados já obtidos: `FIRST(E) = { (, id }`, `FIRST(E') = { +, ϵ }`, etc.)

Passo 2: Calcular FOLLOW

(Resultados já obtidos: `FOLLOW(E) = { ), $ }`, `FOLLOW(T) = { +, ), $ }`, etc.)

Passo 3: Construir a Tabela LL(1)

	id	(	)	+	*	$
E	E→T E’	E→T E’				
E’			E’→ε	E’→+TE’		E’→ε
T	T→F T’	T→F T’				
T’			T’→ε	T’→ε	T’→*FT’	T’→ε
F	F→id	F→(E)				

Passo 4: Analisar uma Entrada

Entrada: `id + id * id $`

Pilha	Entrada	Ação
$ E	id + id * id $	[E, id] → E→T E’
$ E’ T	id + id * id $	[T, id] → T→F T’
$ E’ T’ F	id + id * id $	[F, id] → F→id
$ E’ T’ id	id + id * id $	Match `id`
$ E’ T’	+ id * id $	[T’, +] → T’→ε
$ E’	+ id * id $	[E’, +] → E’→+ T E’

Pilha	Entrada	Ação
$ E’ T +	+ id * id $	Match `+`
$ E’ T	id * id $	[T, id] → T→F T’
$ E’ T’ F	id * id $	[F, id] → F→id
$ E’ T’ id	id * id $	Match `id`
$ E’ T’	* id $	[T’, *] → T’→ F T’
$ E’ T’ F *	* id $	Match `*`
$ E’ T’ F	id $	[F, id] → F→id
$ E’ T’ id	id $	Match `id`
$ E’ T’	$	[T’, $] → T’→ε
$ E’	$	[E’, $] → E’→ε
$	$	Aceita!

Resultado

A entrada `id + id * id` é considerada válida conforme a gramática.

---

4. Implementação em Pseudocódigo

“Será apresentado um exemplo de implementação do Parser Preditivo Tabular LL(1) em pseudocódigo, baseado na gramática e tabela construídas.”

Pseudocódigo

A implementação utiliza uma pilha e a tabela de análise para processar a entrada. Assume-se que:

- A tabela `Tabela[A, t]` contém as produções conforme construída.
- A entrada é um vetor de tokens terminado por `$`.
- Os símbolos são representados como strings (não-terminais em maiúsculas, terminais em minúsculas ou símbolos).

```
Função ParserLL1(entrada):
    Iniciar pilha com ['$', 'E']  // Símbolo inicial 'E' e fim '$'
    Definir lookahead como entrada[0]  // Primeiro token da entrada
    Definir i como 0  // Índice da entrada

    Enquanto pilha não estiver vazia:
        Definir topo como pilha.topo()

        Se topo for terminal ou '$':
            Se topo = lookahead:
                Remover topo da pilha  // Match
                Incrementar i
                Definir lookahead como entrada[i]
            Senão:
                Retornar "Erro: token inesperado"
```

```
        Senão se topo for não-terminal:
            Se Tabela[topo, lookahead] existir:
                Definir produção como Tabela[topo, lookahead]
                Remover topo da pilha
                Se produção ≠ ϵ:
                    Inserir símbolos da produção na pilha (em ordem inversa)
            Senão:
                Retornar "Erro: nenhuma produção aplicável"

    Se lookahead = '$' e pilha vazia:
        Retornar "Entrada aceita"
    Senão:
        Retornar "Erro: entrada incompleta ou inválida"

Fim Função
```

Notas sobre a Implementação

- A tabela deve ser pré-construída e armazenada (ex.: como matriz ou dicionário).
- Erros são reportados se o lookahead não corresponde ao esperado ou se não há produção aplicável.

---

5. Implementação em Python

“Será demonstrada uma implementação funcional do Parser LL(1) em Python, utilizando a mesma gramática e tabela do exemplo prático.”

Código em Python

A seguir, apresenta-se uma implementação que simula o parser com a tabela LL(1) construída. A tabela é representada como um dicionário, e a entrada é processada passo a passo.

```
# Tabela LL(1) como dicionário: (não-terminal, terminal) -> produção
tabela = {
    ('E', 'id'): ['T', "E'"],
    ('E', '('): ['T', "E'"],
    ("E'", '+'): ['+', 'T', "E'"],
    ("E'", ')'): ['ϵ'],
    ("E'", '$'): ['ϵ'],
    ('T', 'id'): ['F', "T'"],
    ('T', '('): ['F', "T'"],
    ("T'", '+'): ['ϵ'],
    ("T'", '*'): ['*', 'F', "T'"],
    ("T'", ')'): ['ϵ'],
    ("T'", '$'): ['ϵ'],
    ('F', 'id'): ['id'],
    ('F', '('): ['(', 'E', ')']
}

# Função para verificar se um símbolo é não-terminal
def eh_nao_terminal(simbolo):
    return simbolo in ['E', "E'", 'T', "T'", 'F']

# Função do parser LL(1)
def parser_ll1(entrada):
    pilha = ['$', 'E']  # Pilha inicial com símbolo inicial 'E' e fim '$'
    entrada = entrada + ['$']  # Adiciona '$' ao final da entrada
    i = 0  # Índice da entrada
    lookahead = entrada[i]
```

```
    print(f"{'Pilha':<20} {'Entrada':<20} {'Ação':<20}")
    print("-" * 60)

    while pilha:
        topo = pilha[-1]  # Topo da pilha
        estado = f"{' '.join(pilha):<20} {' '.join(entrada[i:]):<20}"

        if not eh_nao_terminal(topo):  # Terminal ou '$'
            if topo == lookahead:
                pilha.pop()  # Match
                i += 1
                lookahead = entrada[i]
                print(f"{estado}Match {topo}")
            else:
                return f"Erro: token inesperado '{lookahead}'"
        else:  # Não-terminal
            chave = (topo, lookahead)
            if chave in tabela:
                producao = tabela[chave]
                pilha.pop()  # Remove o não-terminal
                if producao != ['ϵ']:  # Se não for epsilon, empilha a produção
                    pilha.extend(reversed(producao))
                print(f"{estado}{topo} → {' '.join(producao)}")
            else:
                return f"Erro: nenhuma produção para {topo} com lookahead {lookahead}"

    if lookahead == '$' and not pilha:
        return "Entrada aceita"
    return "Erro: entrada incompleta ou inválida"

# Teste da implementação
```

```
entrada = ['id', '+', 'id', '*', 'id']
resultado = parser_ll1(entrada)
print("\nResultado:", resultado)
```

Saída Esperada

Ao executar o código com a entrada `['id', '+', 'id', '*', 'id']`, obtém-se uma saída semelhante a:

```
Pilha                Entrada              Ação
------------------------------------------------------------
$ E                  id + id * id $       E → T E'
$ E' T               id + id * id $       T → F T'
$ E' T' F            id + id * id $       F → id
$ E' T' id           id + id * id $       Match id
$ E' T'              + id * id $          T' → ϵ
$ E'                 + id * id $          E' → + T E'
$ E' T +             + id * id $          Match +
$ E' T               id * id $            T → F T'
$ E' T' F            id * id $            F → id
$ E' T' id           id * id $            Match id
$ E' T'              * id $               T' → * F T'
$ E' T' F *          * id $               Match *
$ E' T' F            id $                 F → id
$ E' T' id           id $                 Match id
$ E' T'              $                    T' → ϵ
$ E'                 $                    E' → ϵ
$                    $

Resultado: Entrada aceita
```

Explicação do Código

- **Tabela**: Representada como um dicionário onde as chaves são tuplas `(não-terminal, terminal)` e os valores são listas de símbolos da produção.
- **Pilha**: Implementada como uma lista Python, com operações de `pop()` e `extend()` para manipulação.
- **Entrada**: Lista de tokens com `$` adicionado ao final.
- **Execução**: O parser segue o mesmo algoritmo do pseudocódigo, exibindo cada passo para fins didáticos.
- **Saída**: Mostra a evolução da pilha, entrada e ações realizadas, facilitando a depuração e o entendimento.

Notas sobre a Implementação

- A função assume uma entrada válida (tokens compatíveis com a gramática).
- Pode-se expandir o código para tratar erros mais detalhadamente ou integrar com um lexer.
