Compiladores

Análise Sintática

Cristiano Lehrer, M.Sc.

ybadoo

Introdução (1/3)

- A análise sintática constitui a segunda fase de um tradutor.
- Sua função é verificar se as construções usadas no programa estão gramaticalmente corretas.
- Normalmente, as estruturas sintáticas válidas são especificadas através de uma **gramática livre do contexto**.
- Dada uma gramática livre do contexto *G* e uma sentença *s*, o objetivo do analisador sintático é verificar se a sentença *s* pertence à linguagem gerada por *G*:
  - O analisador sintático, também chamado *parser*, recebe do analisador léxico a sequência de *tokens* que constitui a sentença *s* e produz como resultado uma árvore de derivação para *s*, se a sentença é válida, ou emite uma mensagem de erro, caso contrário.

Introdução (2/3)

- A árvore de derivação para *s* pode ser construída explicitamente (representada através de uma estrutura de dados) ou ficar implícita nas chamadas das rotinas que aplicam as regras de produção da gramática durante o reconhecimento.
- Os analisadores sintáticos devem ser projetados de modo que possam prosseguir na análise, até o fim do programa, mesmo que encontrem erros no texto fonte.
- Há duas estratégias básicas para a análise sintática:
  - TOP-DOWN ou DESCENDENTE.
  - BOTTOM-UP ou REDUTIVA.

Introdução (3/3)

- Os métodos de análise baseados na estratégia *top-down* constroem a árvore de derivação a partir do símbolo inicial da gramática (raiz da árvore), fazendo a árvore crescer até atingir suas folhas:
  - Em cada passo, um lado esquerdo de produção é substituído por um lado direito (expansão).
- A estratégia *bottom-up* realiza a análise no sentido inverso, isto é, a partir dos *tokens* do texto fonte (folhas da árvore de derivação) constrói a árvore até o símbolo inicial da gramática:
  - Em cada passo, um lado direito de produção é substituído por um símbolo não-terminal (redução).

Gramáticas Livres do Contexto

- Gramática livre do contexto é qualquer gramática G = (N, T, P, S) cujas produções são da forma $A \rightarrow \alpha$, onde A é um símbolo não-terminal e α é um elemento de $(N \cup T)^*$.
- A denominação livre do contexto deriva do fato de o não-terminal A pode ser substituído por α em qualquer contexto, isto é, sem depender de qualquer análise dos símbolos que sucedem ou antecedem A na forma sentencial em questão.
- Exemplo:
  - G = ({E}, {+, -, *, /, (,), x}, P, E), sendo
  - P = {E → E + E | E - E | E * E | E / E | (E) | x}

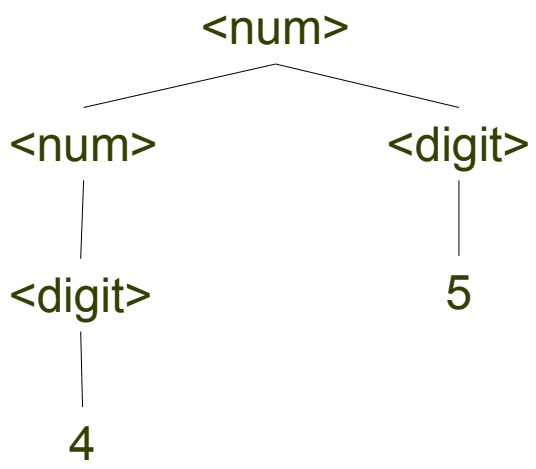
Árvore de Derivação (1/2)

- Árvore de derivação é a representação gráfica de uma derivação de sentença.
- Essa representação apresenta, de forma explícita, a estrutura hierárquica que originou a sentença.
- Dada uma gramática livre do contexto, a árvore de derivação para uma sentença é obtida como segue:
  - A raiz da árvore é o símbolo inicial da gramática.
  - Os vértices interiores, obrigatoriamente, são não-terminais. Se $A \rightarrow X_1X_2...X_n$ é uma produção da gramática, então A será um vértice interior, e X_1, X_2, ..., X_n serão os seus filhos (da esquerda para a direita).
  - Símbolos terminais e a palavra vazia são vértices folha.

ybadoo

Árvore de Derivação (2/2)

- Exemplo:
  - G = ({<num>, <digit>}, {0, 1, 2, 3, 4, 5, 6, 7, 8, 9}, P, <num>), sendo
  - P = {<num> → <num><digit> | <digit>, <digit> → 0 | 1 | 2 | ... | 7 | 8 | 9}
  - A árvore de derivação correspondente à sentença 45 é a mostrada a seguir.



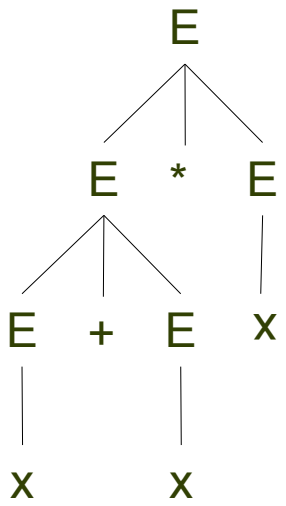
Derivações mais à Esquerda e mais à Direita

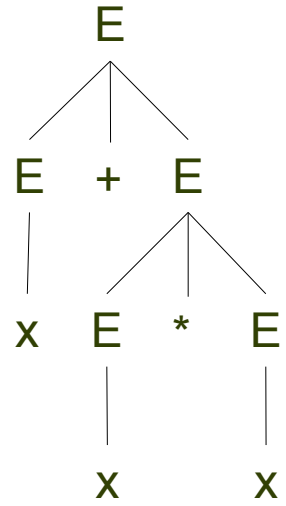
- Derivação mais à esquerda de uma sentença é a sequência de formas sentenciais que se obtém derivando sempre o símbolo não-terminal mais à esquerda.
- Uma derivação mais à direita aplica as produções sempre ao não-terminal mais à direita.
- Exemplo:
  - G = ({E}, {+, -, *, /, (,), x}, {E → E + E | E - E | E * E | E / E | (E) | x}, E)
  - Derivação mais à esquerda da sentença x + x * x
    - E ⇒ E + E ⇒ x + E ⇒ x + E * E ⇒ x + x * E ⇒ x + x * x
  - Derivação mais à direita da sentença x + x * x
    - E ⇒ E + E ⇒ E + E * E ⇒ E + E * x ⇒ E + x * x ⇒ x + x * x

ybadoo

Gramática Ambígua (1/2)

- Gramática ambígua é uma gramática que permite construir mais de uma árvore de derivação para uma mesma sentença.
- Exemplo:
  - G = ({E}, {+, -, *, /, (,), x}, {E → E + E | E - E | E * E | E / E | (E) | x}, E)
  - Apresentação de duas árvores de derivação para a sentença x + x * x





Gramática Ambígua (2/2)

- A existência de gramáticas ambíguas torna-se um problema quando os reconhecedores exigem derivações unívocas para obter um bom desempenho, ou mesmo para concluir a análise sintática.
- Não existe um procedimento geral para eliminar a ambiguidade de uma gramática.
- Existem gramáticas para as quais é impossível eliminar produções ambíguas.
- Uma linguagem é dita inerentemente ambígua se qualquer gramática livre do contexto que a descreve é ambígua.
- Por exemplo, a seguinte linguagem é inerentemente ambígua:
  - $\{w \mid w = a^n b^n c^m d^m$ ou $a^n b^m c^m d^n, n \geq 1, m \geq 1\}$

ybadoo

Gramática Recursiva à Esquerda

- Gramática recursiva à esquerda é uma gramática livre do contexto que permite a derivação $A \Rightarrow^{+} A\alpha$ para algum $A \in N$, ou seja, um não-terminal deriva ele mesmo, de forma direta ou indireta, como o símbolo mais à esquerda de uma sub-palavra gerada.
- Os reconhecedores *top-down* exigem que a gramática não apresente recursividade à esquerda.
- Quando a recursão é direta, a eliminação é simples.
- Quando a recursão apresenta-se de forma indireta, a eliminação requer que a gramática seja inicialmente simplificada.

Gramática Simplificada

- Gramática simplificada é uma gramática livre do contexto que não apresenta símbolos inúteis, produções vazias, nem produções do tipo A → B.
  - Sequência de simplificação:
    - Exclusão das produções vazias.
    - Exclusão das produções da forma A → B.
    - Exclusão dos símbolos inúteis.

Transformações de Gramática Livre do Contexto

- Uma vez que existem diferentes métodos de análise, cada qual exigindo gramáticas com características específicas, é importante que uma gramática possa ser transformada, porém, sem perder a qualidade de gerar a mesma linguagem.
- As gramáticas que, mesmo tendo conjuntos diferentes de produções, geram a mesma linguagem são ditas gramáticas equivalentes.

Eliminação de Recursividade à Esquerda (1/4)

- Eliminação de recursividade direta:
  - Substituir cada regra da forma:
    - $A \rightarrow A\alpha_1 \mid A\alpha_2 \mid \ldots \mid A\alpha_n \mid \beta_1 \mid \beta_2 \mid \ldots \mid \beta_m$
  - onde nenhum β_i começa por A, por:
    - $A \rightarrow \beta_1 X \mid \beta_2 X \mid \ldots \mid \beta_m X$
    - $X \rightarrow \alpha_1 X \mid \alpha_2 X \mid \ldots \mid \alpha_n X \mid \varepsilon$
  - Se produções vazias não são permitidas, a substituição deve ser:
    - $A \rightarrow \beta_1 \mid \beta_2 \mid \ldots \mid \beta_m \mid \beta_1 X \mid \beta_2 X \mid \ldots \mid \beta_m X$
    - $X \rightarrow \alpha_1 \mid \alpha_2 \mid \ldots \mid \alpha_n \mid \alpha_1 X \mid \alpha_2 X \mid \ldots \mid \alpha_n X$

Eliminação de Recursividade à Esquerda (2/4)

- Exemplo:
  - G = ({A}, {a, b}, {A → Aa | b}, A)
  - Eliminação da recursividade direta, com palavra vazia:
    - G = ({A, X}, {a, b}, {A → bX, X → aX | ε}, A)
  - Eliminação da recursividade direta, sem palavra vazia:
    - G = ({A, X}, {a, b}, {A → b | bX, X → a | aX}, A)

Eliminação de Recursividade à Esquerda (3/4)

- Eliminação de recursões indiretas:
  - A gramática livre do contexto precisa estar simplificada.
  - Renomeação dos não-terminais em uma ordem crescente qualquer:
    - Sendo n a cardinalidade de N, renomear os não-terminais para A_1, A_2, ..., A_n e fazer as correspondentes renomeações nas regras de P.
  - Transformação das produções para a forma $A_i \rightarrow A_j\gamma$, onde $i \leq j$:
    - Substituir cada produção da forma $A_i \rightarrow A_j\gamma$ pelas produções $A_i \rightarrow \delta_1\gamma \mid \delta_2\gamma \mid ... \mid \delta_k\gamma$, onde $A_j \rightarrow \delta_1 \mid \delta_2 \mid \ldots \mid \delta_k$ são produções de A_j.
  - Exclusão das recursões diretas.

Eliminação de Recursividade à Esquerda (4/4)

- Exemplo:
  - $G = (\{S, A\}, \{a, b\}, \{S \rightarrow AA \mid a, A \rightarrow SS \mid b\}, S)$
  - Renomeação dos não-terminais em ordem crescente:
    - $G = (\{A_1, A_2\}, \{a, b\}, \{A_1 \rightarrow A_2A_2 \mid a, A_2 \rightarrow A_1A_1 \mid b\}, A_1)$
  - Transformação das produções para a forma $A_i \rightarrow A_j\gamma$, onde $i \leq j$:
    - $G = (\{A_1, A_2\}, \{a, b\}, \{A_1 \rightarrow A_2A_2 \mid a, A_2 \rightarrow A_2A_2A_1 \mid aA_1 \mid b\}, A_1)$
  - Exclusão das recursões diretas:
    - $G = (\{A_1, A_2, X\}, \{a, b\}, P, A_1)$, sendo
    - $P = \{A_1 \rightarrow A_2A_2 \mid a, A_2 \rightarrow aA_1 \mid b \mid aA_1X \mid bX, X \rightarrow A_2A_1 \mid A_2A_1X\}$

Fatoração à Esquerda

- Fatorar à esquerda a produção $A \rightarrow \alpha_1\alpha_2...\alpha_n$ é introduzir um novo não-terminal X e, para algum i, substituí-la por $A \rightarrow \alpha_1\alpha_2...\alpha_iX$ e $X \rightarrow \alpha_{i+1}...\alpha_n$.
- A fatoração à esquerda permite eliminar a indecisão sobre qual produção aplicar quando duas ou mais produções iniciam com a mesma forma sentencial.
- Por exemplo, $\alpha\beta_1 \mid \alpha\beta_2$ seria eliminada fatorando as mesmas para $A \rightarrow \alpha X$ e $X \rightarrow \beta_1 \mid \beta_2$.
- Para a análise descendente preditiva, é necessário que a gramática esteja fatorada à esquerda.

Análise Descendente

- A **análise descendente** (*top-down*) de uma sentença (ou programa) pode ser vista como uma tentativa de construir uma árvore de derivação em pré-ordem (da esquerda para a direita) para a sentença em questão:
  - Cria a raiz e, a seguir, cria as subárvores filhas, da esquerda para a direita.
  - Esse processo produz uma derivação mais à esquerda da sentença em análise.
- Três tipos de analisadores sintáticos descendentes:
  - Recursivo com retrocesso (*backtracking*).
  - Recursivo preditivo.
  - Tabular preditivo.

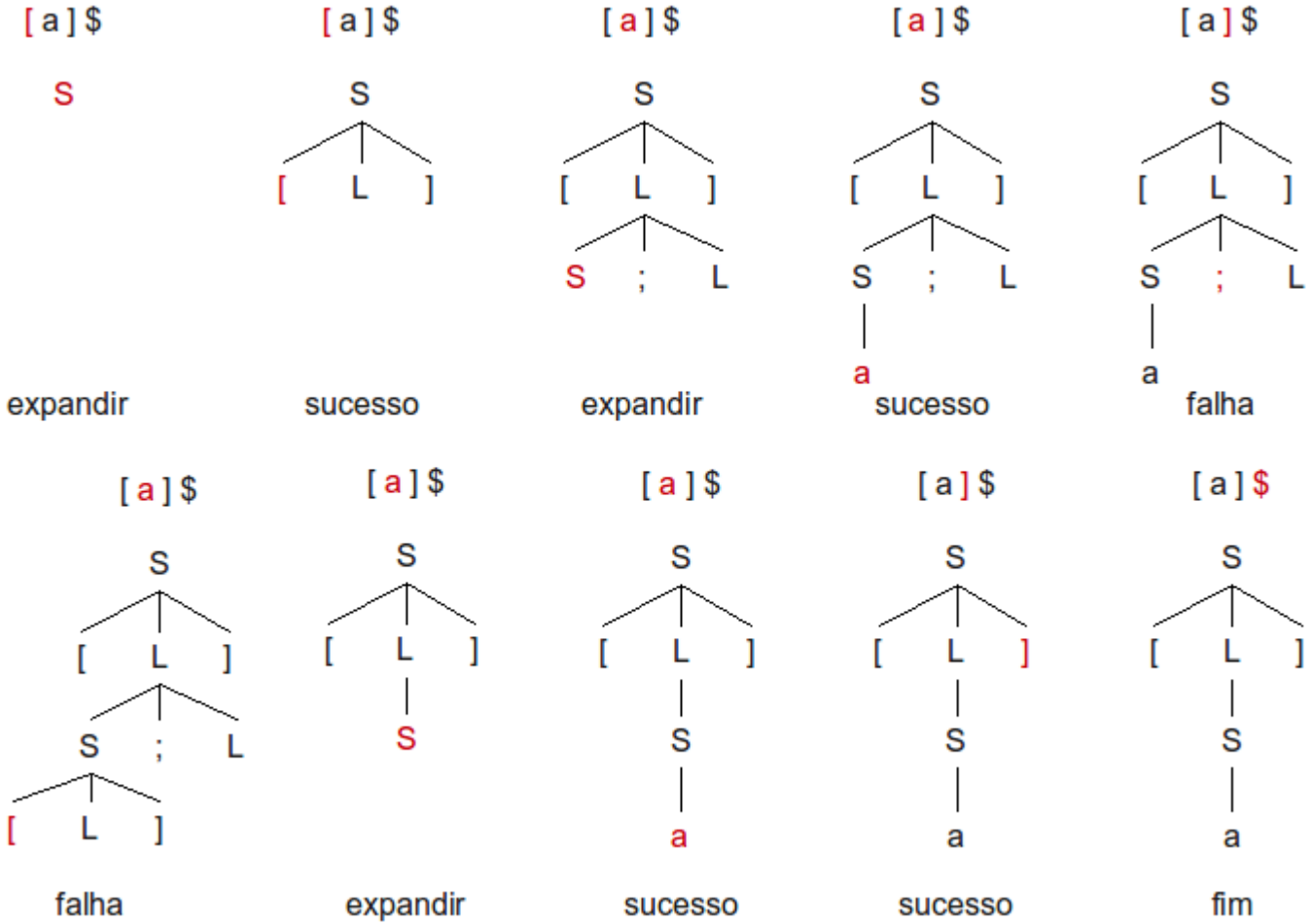
ybadoo

Análise Recursiva com Retrocesso (1/3)

- Faz a expansão da árvore de derivação a partir da raiz, expandindo sempre o não-terminal mais à esquerda.
- Quando existe mais de uma regra de produção para o não-terminal a ser expandido, a opção escolhida é função do símbolo corrente na fita de entrada (*token* sob o cabeçote de leitura).
- Se o *token* de entrada não define univocamente a produção a ser usada, então todas as alternativas vão ser tentadas até que se obtenha sucesso (ou até que a análise falhe irremediavelmente).
- É bom relembrar que a presença de recursividade à esquerda em uma gramática ocasiona problemas para analisadores descendentes:
  - Como a expansão é sempre feita para o não-terminal mais à esquerda, o analisador irá entrar num ciclo infinito se houver esse tipo de recursividade.

Análise Recursiva com Retrocesso (2/3)

- Exemplo da sentença [a] sobre a gramática:
  - G = ({S, L}, {a, ;, [,]}, {S → a | [L], L → S ; L | S}, S)



Análise Recursiva com Retrocesso (3/3)

- Ao processo de voltar atrás no reconhecimento e tentar produções alternativas dá-se o nome de retrocesso ou *backtracking*.
- Tal processo é ineficiente, pois leva à repetição da leitura de partes da sentença de entrada e, por isso, em geral, não é usado no reconhecimento de linguagens de programação:
  - Como o reconhecimento é, geralmente, acompanhado da execução de ações semânticas (por exemplo, armazenamento de identificadores na Tabela de Símbolos), a ocorrência de retrocesso pode levar o analisador sintático a ter que desfazer essas ações.
  - Outra desvantagem dessa classe de analisadores é que, quando ocorre um erro, fica difícil indicar com precisão o local do erro, devido à tentativa de aplicação de produções alternativas.

First(α) (1/2)

- Se α é uma forma sentencial (sequência de símbolos da gramática), então FIRST(α) é o conjunto de terminais que iniciam formas sentenciais derivadas a partir de α. Se $\alpha \Rightarrow^* \varepsilon$ então a palavra vazia também faz parte do conjunto:
  - Se a é terminal, então FIRST(a) = {a}.
  - Se $X \rightarrow \varepsilon$ é uma produção, então adicione ε a FIRST(X).
  - Se $X \rightarrow Y_1Y_2...Y_k$ é uma produção e, para algum i, todos Y_1, Y_2, ..., Y_{i-1} derivam ε, então FIRST(Y_i) está em FIRST(X), juntamente com todos os símbolos não-ε de FIRST(Y_1), FIRST(Y_2), ..., FIRST(Y_{i-1}). O símbolo ε será adicionado a FIRST(X) apenas se todo Y_j(j=1, 2, ..., k) derivar ε.

ybadoo

First(α) (2/2)

- Exemplo:
  - G = ({E, E', T, T', F}, {∨, &, ¬, id}, P, S), onde
  - P = {E → TE', E' → ∨TE' | ε, T → FT', T' → &FT' | ε, F → ¬F | id}

  - FIRST(E) = {¬, id}
  - FIRST(E') = {∨, ε}
  - FIRST(T) = {¬, id}
  - FIRST(T') = {&, ε}
  - FIRST(F) = {¬, id}

ybadoo

Follow(A) (1/2)

- A função FOLLOW é definida para símbolos não-terminais.
- Sendo A um não-terminal, FOLLOW(A) é o conjunto de terminais a que podem aparecer imediatamente à direita de A em alguma forma sentencial. Isto é, o conjunto de terminais a, tal que existe uma derivação da forma $S \Rightarrow^* \alpha A a \beta$ para α e β quaisquer.
  - Se S é o símbolo inicial da gramática e $ é o marcador de fim da sentença, então $ está em FOLLOW(S).
  - Se existe produção do tipo $A \rightarrow \alpha X \beta$, então todos os símbolos de FIRST(β), exceto ε, fazem parte de FOLLOW(X).
  - Se existe produção do tipo $A \rightarrow \alpha X$, ou $A \rightarrow \alpha X \beta$, sendo que $\beta \Rightarrow^* \varepsilon$, então todos os símbolos que estiverem em FOLLOW(A) fazem parte de FOLLOW(X).

Follow(A) (2/2)

- Exemplo:
  - G = ({E, E', T, T', F}, {∨, &, ¬, id}, P, S), onde
  - P = {E → TE', E' → ∨TE' | ε, T → FT', T' → &FT' | ε, F → ¬F | id}

• FIRST(E) = {¬, id}	FOLLOW(E) = {$}
• FIRST(E') = {∨, ε}	FOLLOW(E') = {$}
• FIRST(T) = {¬, id}	FOLLOW(T) = {∨, $}
• FIRST(T') = {&, ε}	FOLLOW(T') = {∨, $}
• FIRST(F) = {¬, id}	FOLLOW(F) = {∨, &, $}

Análise Recursiva Preditiva (1/3)

- É possível implementar analisadores recursivos sem retrocesso:
  - Esses analisadores são chamados recursivos preditivos e, para eles, o símbolo sob o cabeçote de leitura determina exatamente qual produção deve ser aplicada na expansão de cada não-terminal.
- Esses analisadores exigem:
  - Que a gramática não tenha recursividade à esquerda.
  - Que a gramática esteja fatorada à esquerda.
  - Que, para os não-terminais com mais de uma regra de produção, os primeiros terminais deriváveis sejam capazes de identificar, univocamente, a produção que deve ser aplicada a cada instante da análise.

Análise Recursiva Preditiva (2/3)

COMANDO	→	CONDICIONAL
	|	ITERATIVO
	|	ATRIBUIÇÃO
CONDICIONAL	→	**if** EXPR **then** COMANDO
ITERATIVO	→	**repeat** LISTA **until** EXPR
	|	**while** EXPR **do** COMANDO
ATRIBUIÇÃO	→	**id :=** EXPR

Análise Recursiva Preditiva (3/3)

```
function COMANDO;
  if token = 'if'
  then if CONDICIONAL
    then return true
    else return false
  else if token = 'while' or token = 'repeat'
    then if ITERATIVO
      then return true
      else return false
    else if token = 'id'
       then if ATRIBUICAO
         then return true
         else return false
      else return false
```


ybadoo

Análise Preditiva Tabular (1/5)

- É possível construir analisadores preditivos não recursivos que utilizam uma pilha explícita ao invés de chamadas recursivas.
- Esse tipo de analisador implementa um autômato de pilha controlado por uma tabela de análise.
- O princípio do reconhecimento preditivo é a determinação da produção a ser aplicada, cujo lado direito irá substituir o símbolo não-terminal que se encontra no topo da pilha.
- O analisador busca a produção a ser aplicada na tabela de análise, levando em conta o não-terminal no topo da pilha e o token sob o cabeçote de leitura.

Análise Preditiva Tabular (2/5)

- Algoritmo para construir uma tabela de análise preditiva:
  - Para cada produção A → α de G, execute os passos a seguir para criar a linha A da tabela M:
    - Para cada terminal a de FIRST(α), adicione a produção A → α a M[A, a].
    - Se FIRST(α) inclui a palavra vazia, então adicione A → α a M[A, a] para cada b em FOLLOW(A).

E	→	T E'	FIRST(E) = {¬, id}	FOLLOW(E) = {$}
E'	→	˅ T E' | ε	FIRST(E') = {˅, ε}	FOLLOW(E') = {$}
T	→	F T'	FIRST(T) = {¬, id}	FOLLOW(T) = {˅, $}
T'	→	& F T' | ε	FIRST(T') = {&, ε}	FOLLOW(T') = {˅, $}
F	→	¬ F | id	FIRST(F) = {¬, id}	FOLLOW(F) = {˅, &, $}

Análise Preditiva Tabular (3/5)

- Para E → T E' tem-se FIRST(T E') = {¬, id} M[E, ¬] = M[E, id] = E → T E'
- Para E' → ˅ T E' tem-se FIRST(˅ T E') = {˅} M[E', ˅] = E' → ˅ T E'
- Para E' → ε tem-se FOLLOW(E') = {$} M[E', $] = E' → ε
- Para T → F T' tem-se FIRST(F T') = {¬, id} M[T, ¬] = M[T, id] = T → F T'
- Para T' → & F T' tem-se FIRST(& F T') = {&} M[T', &] = T' → & F T'
- Para T' → ε tem-se FOLLOW(T') = {˅ ,$} M[T', ˅] = M[T', $] = T' → ε
- Para F → ¬ F tem-se FIRST(¬ F) = {¬} M[F, ¬] = F → ¬ F
- Para F → id tem-se FIRST(id) = {id} M[F, id] = F → id

	id	**˅**	**&**	**¬**	**$**
E	E → T E'			E → T E'	
E'		E' → ˅ T E'			E' → ε
T	T → F T'			T → F T'	
T'		T' → ε	T' → & F T'		T' → ε
F	F → id			F → ¬ F	

ybadoo

Análise Preditiva Tabular (4/5)

- Considerando X como símbolo no topo da pilha e a como terminal da fita de entrada sob o cabeçote de leitura, o analisador executa uma das três ações possíveis:
  - Se X = a = $, o analisador para, aceitando a sentença.
  - Se X = a $\neq$ $, o analisador desempilha a e avança o cabeçote de leitura para o próximo símbolo na fita de entrada.
  - Se X é um símbolo não-terminal, o analisador consulta a entrada M[X, a] da tabela de análise. Essa entrada poderá conter uma produção da gramática ou ser vazia.
    - Supondo M[X, a] = {X $\rightarrow$ UVW}, o analisador substitui X (que está no topo da pilha) por WVU (ficando U no topo) e retorna a produção aplicada.
    - Se M[X, a] é vazia, isso corresponde a uma situação de erro; nesse caso, o analisador chama uma rotina de tratamento de erro.

Análise Preditiva Tabular (5/5)

	id	˅	&	¬	$
E	E → T E'			E → T E'	
E'		E' → ˅ T E'			E' → ε
T	T → F T'			T → F T'	
T'		T' → ε	T' → & F T'		T' → ε
F	F → id			F → ¬ F	

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 $	Desempilha e lê símbolo
$ E' T'	˅ id & id $	T' → ε
$ E'	˅ id & id $	E' → ˅ T E'
$ E' T ˅	˅ id & id $	Desempilha e lê símbolo
$ E' T	id & id $	T → F T'
$ E' T' F	id & id $	F → id
$ E' T' id	id & id $	Desempilha e lê símbolo
$ E' T'	& id $	T' → & F T'
$ E' T' F &	& id $	Desempilha e lê símbolo
$ E' T' F	id $	F → id
$ E' T' id	id $	Desempilha e lê símbolo
$ E' T'	$	T' → ε
$ E'	$	E' → ε
$	$	Aceita

Recuperação de Erros na Análise LL

- Na tabela LL, as lacunas representam situações de erro e devem ser usadas para chamar rotinas de recuperação.
- Pode-se alterar a tabela de análise para recuperar erros segundo dois modos distintos:
  - Modo pânico – na ocorrência de um erro, o analisador despreza símbolos da entrada até encontrar um *token* de sincronização.
  - Recuperação local – o analisador tenta recuperar o erro, fazendo alterações sobre um símbolo apenas:
    - Desprezando o *token* da entrada, ou substituindo-o por outro, ou inserindo um novo *token*, ou ainda, removendo um símbolo da pilha.

Modo Pânico (1/3)

- O conjunto de *tokens* de sincronização para um símbolo não-terminal A é formado pelos terminais em FOLLOW(A).
- Ao encontrar um token inesperado na sentença em análise:
  - Emitir mensagem de erro;
  - Tomar uma das seguintes atitudes:
    - Se a entrada na tabela estiver vazia, ler o próximo *token* (significa descarte do *token* lido).
    - Se a entrada é *sinc*, desempilhar o não-terminal do topo.
    - Se o *token* do topo não é igual ao símbolo da entrada, desempilhar o *token*.

ybadoo

Modo Pânico (2/3)

E	→	T E'	FIRST(E) = {¬, id}	FOLLOW(E) = {$}
E'	→	˅ T E' | ε	FIRST(E') = {˅, ε}	FOLLOW(E') = {$}
T	→	F T'	FIRST(T) = {¬, id}	FOLLOW(T) = {˅, $}
T'	→	& F T' | ε	FIRST(T') = {&, ε}	FOLLOW(T') = {˅, $}
F	→	¬ F | id	FIRST(F) = {¬, id}	FOLLOW(F) = {˅, &, $}

	id	˅	&	¬	$
E	E → T E'			E → T E'	sinc
E'		E' → ˅ T E'			E' → ε
T	T → F T'	sinc		T → F T'	sinc
T'		T' → ε	T' → & F T'		T' → ε
F	F → id	sinc	sinc	F → ¬ F	sinc

Modo Pânico (3/3)

	id	˅	&	¬	$
E	E → T E'			E → T E'	sinc
E'		E' → ˅ T E'			E' → ε
T	T → F T'	sinc		T → F T'	sinc
T'		T' → ε	T' → & F T'		T' → ε
F	F → id	sinc	sinc	F → ¬ F	sinc

Pilha	Entrada	Ação
$ E	id ˅ & id $	E → T E'
$ E' T	id ˅ & id $	T → F T'
$ E' T' F	id ˅ & id $	F → id
$ E' T' id	id ˅ & id $	desempilha e lê símbolo
$ E' T'	˅ & id $	T' → ε
$ E'	˅ & id $	E' → ˅ T E'
$ E' T ˅	˅ & id $	desempilha e lê símbolo
$ E' T	& id $	descarta a entrada
$ E' T	id $	T → F T'
$ E' T' F	id $	F → id
$ E' T' id	id $	desempilha e lê símbolo
$ E' T'	$	T' → ε
$ E'	$	E' → ε
$	$	**aceita**

ybadoo

Recuperação Local (1/3)

- As rotinas de atendimento a erros fazem descarte, substituição ou inserção de apenas um símbolo a cada erro descoberto, tendo o cuidado de, no caso de inserção, não provocar um ciclo infinito no analisador.
- A tabela LL deve ser expandida para incluir as situações em que ocorre discrepância entre o *token* do topo da pilha e o da fita de entrada.

	id	**∨**	**&**	**¬**	**$**
E	E → T E'	erro1	erro1	E → T E'	erro1
E'	E' → ε	E' → ∨ T E'	E' → ε	E' → ε	E' → ε
T	T → F T'	erro1	erro1	T → F T'	erro1
T'	T' → ε	T' → ε	T' → & F T'	T' → ε	T' → ε
F	F → id	erro1	erro1	F → ¬ F	erro1
id	desempilha				
∨		desempilha			
&			desempilha		
¬				desempilha	
$	erro2	erro2	erro2	erro2	**aceita**

Recuperação Local (2/3)

- Nas linhas da tabela original, onde existem produções vazias, as lacunas foram preenchidas com essas produções.
- As lacunas restantes foram preenchidas com nomes de rotinas de tratamento de erro:
  - `erro 1` – insere o *token* `id` na entrada e emite:
    - `operando esperado`
  - `erro 2` – descarta o *token* da entrada e emite:
    - `fim do arquivo encontrado`
- As lacunas que permanecerem vazias representam situações que jamais ocorrerão durante a análise.

ybadoo

Recuperação Local (3/3)

	id	˅	&	¬	$
E	E → T E'	erro1	erro1	E → T E'	erro1
E'	E' → ε	E' → ˅ T E'	E' → ε	E' → ε	E' → ε
T	T → F T'	erro1	erro1	T → F T'	erro1
T'	T' → ε	T' → ε	T' → & F T'	T' → ε	T' → ε
F	F → id	erro1	erro1	F → ¬ F	erro1
id	desempilha				
˅		desempilha			
&			desempilha		
¬				desempilha	
$	erro2	erro2	erro2	erro2	**aceita**

Pilha	Entrada	Ação
$ E	id ˅ & id $	E → T E'
$ E' T	id ˅ & id $	T → F T'
$ E' T' F	id ˅ & id $	F → id
$ E' T' id	id ˅ & id $	desempilha e lê símbolo
$ E' T'	˅ & id $	T' → ε
$ E'	˅ & id $	E' → ˅ T E'
$ E' T ˅	˅ & id $	desempilha e lê símbolo
$ E' T	& id $	erro1
$ E' T	id & id $	T → F T'
$ E' T' F	id & id $	F → id
$ E' T' id	id & id $	desempilha e lê símbolo
$ E' T'	& id $	T' → & F T'
$ E' T' F &	& id $	desempilha e lê símbolo
$ E' T' F	id $	F → id
$ E' T' id	id $	desempilha e lê símbolo
$ E' T'	$	T' → ε
$ E'	$	E' → ε
$	$	**aceita**

Análise Redutiva (1/2)

- A análise redutiva (*bottom-up*) de uma sentença pode ser vista como a tentativa de construir uma árvore de derivação a partir das folhas, produzindo uma derivação mais à direita ao reverso.
- A denominação redutiva refere-se ao processo que sofre a sentença de entrada, a qual é reduzida até ser atingido o símbolo inicial da gramática.
- Dá-se o nome de redução à operação de substituição do lado direito de uma produção pelo não-terminal correspondente.
- Duas classes de analisadores:
  - Analisadores de precedência de operadores.
  - Analisadores LR (*Left to right with Rightmost derivation*).

ybadoo

Análise Redutiva (2/2)

- As ações que podem ser realizadas por um reconhecedor *bottom-up* são as seguintes:
  - Empilha – coloca no topo da pilha o símbolo que está sendo lido e avança o cabeçote de leitura.
  - Reduz – substitui o *handle* do topo da pilha pelo não-terminal correspondente.
  - Aceita – reconhece que a sentença de entrada foi gerada pela gramática.
  - Erro – chama uma sub-rotina de atendimento a erros.

Analisadores de Precedência de Operadores (1/2)

- Esses analisadores operam sobre a classe das gramáticas de operadores.
- Nessas gramáticas os não-terminais aparecem sempre separados por símbolos terminais:
  - Nunca aparecem dois não-terminais adjacentes.
- Além disso, as produções não derivam a palavra vazia:
  - Nenhum lado direito de produção é ε.
- A análise de precedência de operadores é bastante eficiente e é aplicada principalmente, no reconhecimento de expressões.
- Dentre as desvantagens, estão a dificuldade em lidar com operadores iguais que tenham significado distintos e o fato de se aplicarem a uma classe restrita de gramáticas.

Analisadores de Precedência de Operadores (2/2)

- Para identificar o *handle*, os analisadores de precedência de operadores baseiam-se em relações de precedência existentes entre os *tokens* (operandos e operadores).
- São três as relações de precedência entre os terminais:
  - `a < b` significa que `a` tem precedência menor do que `b`.
  - `a = b` significa que `a` e `b` têm a mesma precedência.
  - `a > b` significa que `a` tem precedência maior do que `b`.
- A utilidade dessas relações na análise de um sentença é a identificação do *handle*:
  - < identifica o limite esquerdo do *handle*.
  - = indica que os terminais pertencem ao mesmo *handle*.
  - > identifica o limite direito do *handle*.

ybadoo

Construção da Tabela de Precedência (1/4)

- A tabela é uma matriz quadrada que relaciona todos os terminais da gramática mais o marcador `$`.
- Na tabela, os terminais nas linhas representam terminais no topo da pilha, e os terminais nas colunas representam terminais sob o cabeçote de leitura.

	id	v	&	(	)	$
id		>	>		>	>
v	<	>	<	<	>	>
&	<	>	>	<	>	>
(	<	<	<	<	=	
)		>	>		>	>
$	<	<	<	<		aceita

Construção da Tabela de Precedência (2/4)

- Considere dois operadores Θ_1 e Θ_2:
  - Regra 1 – se o operador Θ_1 tem maior precedência que o operador Θ_2, então Θ_1 (na pilha) > Θ_2 (na entrada) e Θ_2 (na pilha) < Θ_1 (na entrada):
    - Por exemplo, o operador de multiplicação (`*`) tem maior precedência que o operador de adição (`+`), logo `* > +` e `+ < *`
  - Regra 2 – se Θ_1 e Θ_2 têm igual precedência (ou são iguais) e são associativos à esquerda, então Θ_1 > Θ_2 e Θ_2 > Θ_1; se são associativos à direita, então Θ_1 < Θ_2 e Θ_2 < Θ_1:
    - Por exemplo, os operadores de multiplicação (`*`) e divisão (`/`) têm a mesma precedência e são associativos à esquerda, portanto, `* > /` e `/ > *`
    - Já o operador de exponenciação (`**`) é associativo à direita, logo `** < **`

Construção da Tabela de Precedência (3/4)

- Continuação:
  - As relações entre os operadores e os demais *tokens* (operandos e delimitadores) são fixas.
  - Regra 3 – para todos os operadores Θ, tem-se:

$$\Theta < \texttt{id} \qquad \Theta < \texttt{(} \qquad \Theta > \texttt{)} \qquad \Theta > \texttt{\$}$$

$$\texttt{id} > \Theta \qquad \texttt{(} < \Theta \qquad \texttt{)} > \Theta \qquad \texttt{\$} < \Theta$$

  - Regra 4 – as relações entre os *tokens* que não são operadores também são fixas:

$$\texttt{(} < \texttt{(} \qquad \texttt{)} > \texttt{)} \qquad \texttt{id} > \texttt{)} \qquad \texttt{\$} < \texttt{(}$$

$$\texttt{(} = \texttt{)} \qquad \texttt{)} > \texttt{\$} \qquad \texttt{id} > \texttt{\$} \qquad \texttt{\$} < \texttt{id}$$

$$\texttt{(} < \texttt{id}$$

Construção da Tabela de Precedência (4/4)

$$E \rightarrow E \ \mathbf{v} \ T \mid T$$

$$T \rightarrow T \ \mathbf{\&} \ F \mid F$$

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

	id	**v**	**&**	**(**	**)**	**$**
id		>	>		>	>
v	<	>	<	<	>	>
&	<	>	>	<	>	>
(	<	<	<	<	=	
)		>	>		>	>
$	<	<	<	<		**aceita**

Processo de Análise (1/2)

- Seja `a` o terminal mais ao topo da pilha e `b` o terminal sob o cabeçote de leitura:
  - Se `a < b` ou `a = b` então empilha.
  - Se `a > b` procura o *handle* na pilha (o qual deve estar delimitado pelas relações `<` e `>`) e o substitui pelo não-terminal correspondente.
- Os analisadores de precedência desconsideram os não-terminais da gramática, levando em conta apenas a presença dos mesmos (suas identidades não interessam).

Processo de Análise (2/2)

$E \rightarrow E\ \mathbf{v}\ T \mid T$

$T \rightarrow T\ \mathbf{\&}\ F \mid F$

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

	id	**v**	**&**	**(**	**)**	**$**
id		>	>		>	>
v	<	>	<	<	>	>
&	<	>	>	<	>	>
(	<	<	<	<	=	
)		>	>		>	>
$	<	<	<	<		**aceita**

Pilha	**Relação**	**Entrada**	**Ação**	***Handle***
$	<	id v id & id $	empilha id	
$ id	>	v id & id $	reduz	id
$ E	<	v id & id $	empilha v	
$ E v	<	id & id $	empilha id	
$ E v id	>	& id $	reduz	id
$ E v E	<	& id $	empilha &	
$ E v E &	<	id $	empilha id	
$ E v E & id	>	$	reduz	id
$ E v E & E	>	$	reduz	E & E
$ E v E	>	$	reduz	E v E
$ E		$	**aceita**	

Erros na Análise de Precedência (1/4)

- Na análise de precedência de operadores, existem dois momentos nos quais o analisador pode descobrir erros sintáticos:
  - Na consulta à matriz de precedência, quando não existe relação de precedência entre o terminal mais ao topo da pilha e o símbolo da entrada:
    - A entrada da matriz está vazia.
  - Quando o analisador supõe a existência de um *handle* no topo da pilha, mas não existe produção com o lado direito correspondente:
    - Não existe *handle* na pilha.

Erros na Análise de Precedência (2/4)

- As lacunas na tabela de precedência evidenciam condições de erro.
- Deve-se examinar caso a caso, para definir a mensagem e a recuperação apropriadas.
- Em geral, identificam-se classes de erros, tendo-se que implementar uma rotina para cada classe.
- Embora os símbolos não-terminais sejam transparentes na identificação do *handle*, a redução deve acontecer se realmente houver um *handle* (lado direito de produção) nas posições mais ao topo da pilha.

Erros na Análise de Precedência (3/4)

	id	v	&	(	)	$
id	erro 2	>	>	erro 2	>	>
v	<	>	<	<	>	>
&	<	>	>	<	>	>
(	<	<	<	<	=	erro 1
)	erro 2	>	>	erro 2	>	>
$	<	<	<	<	erro 3	**aceita**

- Erros na consulta a matriz:
  - `erro 1` – empilha `)` e emite: `falta parêntese à direita`
  - `erro 2` – insere **`v`** na entrada e emite: `operador esperado`
  - `erro 3` – descarta `)` da entrada e emite: `parêntese direito ilegal`
- Erros na redução do *handle*:
  - Se **`v`** ou **`&`** definem um *handle*, verificar se existem não-terminais em ambos os lados do operador:
    - Caso negativo, executar a redução e emitir: `falta expressão`
  - Se o par **`( )`** deve ser reduzido, verificar se existe um símbolo não-terminal entre os parênteses:
    - Caso negativo, executar a redução e emitir: `expressão nula entre parênteses`

ybadoo

Erros na Análise de Precedência (4/4)

E → E **v** T | T

T → T **&** F | F

F → **(** E **)** | **id**

	id	**v**	**&**	**(**	**)**	**$**
id	erro 2	>	>	erro 2	>	>
v	<	>	<	<	>	>
&	<	>	>	<	>	>
(	<	<	<	<	=	erro 1
)	erro 2	>	>	erro 2	>	>
$	<	<	<	<	erro 3	**aceita**

Pilha	**Relação**	**Entrada**	**Ação**	***Handle***
$	<	id v id id $	empilha id	
$ id	>	v id id $	reduz	id
$ E	<	v id id $	empilha v	
$ E v	<	id id $	empilha id	
$ E v id	erro 2	id $	insere v	
$ E v id	>	v id $	reduz	id
$ E v E	>	v id $	reduz	E v E
$ E	<	v id $	empilha v	
$ E v	<	id $	empilha id	
$ E v id	>	$	reduz	id
$ E v E	>	$	reduz	E v E
$ E		$	**aceita**	

ybadoo