Universidade Católica de Pelotas

Análise Sintática

André Rauber Du Bois
dubois@ucpel.tche.br

Análise Sintática

- Cada linguagem possui regras que descrevem a estrutura sintática dos programas bem formados
- A sintaxe das linguagens de programação pode ser descrita através de gramáticas livres de contexto ou pela notação BNF (Backus-Naur)
- Gramáticas fornecem uma especificação sintática precisa e fácil de entender

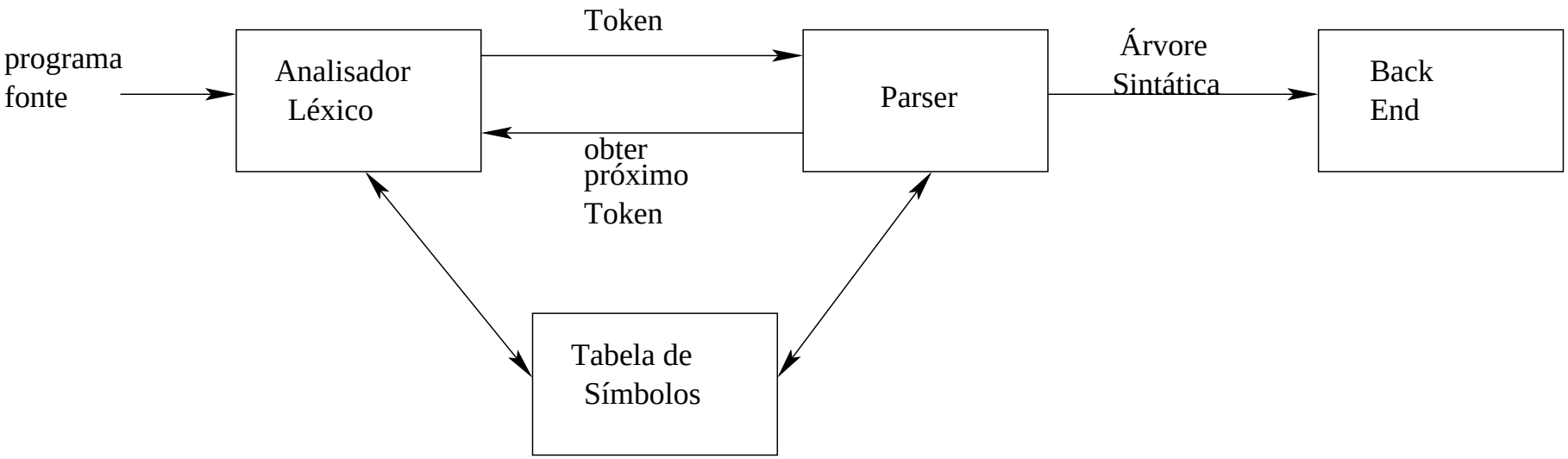
Análise Sintática

- Para certas gramáticas, pode-se construir automáticamente um Analisador Sintático que determine se o programa está sintaticamente bem formado
- O processo de construção do analisador pode detectar ambiguidades sintáticas
- Existem programas que geram parser partindo de gramáticas. Ex: Yacc, javacc, etc.

Análise Sintática

- O analisador sintático (*parser*) obtém uma cadeia de *tokens* do analisador léxico e verifica se o programa é reconhecido pela gramática da linguagem fonte
- Os métodos mais usados para a implementação de analisadores sintáticos são o *bottom-up* e o *top-down*
- O *bottom-up*, constrói a árvore sintática de baixo para cima, ou seja, das folhas para a raiz
- O *top-down* contrói a árvore de cima para baixo, ou seja, da raiz para as folhas

O PARSER


programa
fonte
Analisador
Léxico
Token
obter
próximo
Token
Parser
Árvore
Sintática
Back
End
Tabela de
Símbolos

Erros de Sintaxe

- Um bom compilador deve assistir o programador na identificação e correção de erros
- Os programas podem conter erros em muitos níveis
  - Léxicos: errar a grafia de um identificador ou palavra chave
  - Sintáticos: expressões aritméticas com parênteses não balanceados
  - Semânticos: operador aplicado a um operando incompatível
  - Lógicos: chamada infinitamente recursiva

Erros de Sintaxe

- Muitos dos erros são detectados durante a Análise sintática
- Erros geralmente ocorrem quando o programa desobedece a gramática da linguagem, ou seja, quando o fluxo de tokens vindo da analise léxica não respeita a gramática
- O tratador de erros do analisador sintático possui metas simples:
  - Deve relatar erros claramente
  - Deve se recuperar dos erros e continuar a análise para detectar erros subsequentes
  - Não deve retardar significativamente o processamento de programas corretos

Erros de Sintaxe

- Um erro é detectado quando a construção da árvore sintática falha
- Surgem então duas perguntas: que mensagem de erro deve ser fornecida ao usuário, e como prosseguir depois de um erro
- A posição em que o erro foi detectado pode não ter conexão com a posição real do erro que o usuário cometeu. Ex:

```
x = (p + q( - func(r - s);
```

- Provavelmente o erro do programa é que o parêntese depois do q deveria ser de fechamento, mas a maioria dos analisadores irá reportar a falta de um parêntese no final
- É quase impossível de se detectar o erro já que o q poderia ser uma função, e o sinal `-` um menos monádico

Erros de Sintaxe

- Um sistema de detecção de erros simples poderia simplesmente parar quando um erro é encontrado
- Porém é frustante para o programador ter que compilar várias vezes o mesmo programa, cada vez esperando que seja a última :-)
- Por isso programadores gostam de ter uma idéia de quantos erros restam no programa
- Existem básicamente duas estratégias de recuperação de erros, uma delas chamada de **correção de erros** e a outra **recuperação de erros sem correção**

Correção e Recuperação sem Correção

- No sistema de **correção de erros** o fluxo de entrada é corrigido de alguma maneira para que a análise possa continuar
  - Problema: Uma correção no meio do programa pode ser uma correção errada, estragando a sintaxe de todo o resto do programa
- No sistema de **recuperação de erros sem correção** assim que um erro é encontrado, ele descarta toda a parte inicial do programa e recomeça a análise partindo do resto do programa
  - Problema: recuperação de erros confiável porém dificil de implementar já que a gramática deve ser modificada em tempo de execução
- A maioria dos geradores de analisadores sintáticos tem um

mecanismo interno de detecção e recuperação de erros, assim o projetista tem pouco a fazer sobre o assunto. Porém é importante saber como o tratamento de erros funcionada para tornar o compilador mais amigável

UM ANALISADOR SINTÁTICO TOP-DOWN

- Em uma gramática
  `N -> R | R1 | R2 ... RN`
- Dado um símbolo de entrada `t` um analisador top-down deve decidir qual alternativa `R` deve ser tomada para se reconhecer `t`
- Uma solução simples para o problema é usar-se funções recursivas booleanas que testem as alternativas de `N` em sequência
- Essa aboradagem resulta em um *analisador sintático descendente recursivo*

Um analisador Sintático Descendente Recursivo

- Gramática:

```
entrada    -> expressao EOF
expressao  -> termo expressao_restante
termo      -> IDENTIFICADOR | expressao_entre_parenteses
expressao_entre_parenteses -> '(' expressao ')'
expressao_restante -> '+' expressao | epsilon
```

- Gramática simples na qual o operador de + é associativo à direita
- Exemplo de expressão:
  `IDENTIFICADOR + (IDENTIFICADOR + IDENTIFICADOR)`

Um analisador Sintático Descendente Recursivo

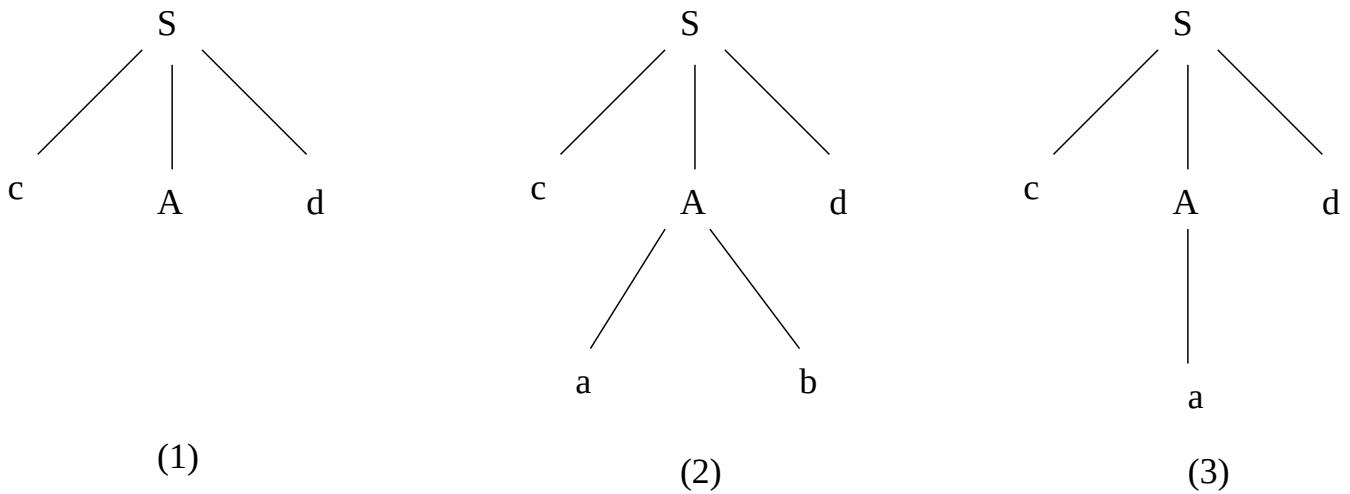
- Arquivo `parser.c`
- Cada regra `N` corresponde a uma rotina que retorna 1 (verdadeiro) se uma produção terminal de `N` foi encontrada. Quando encontrada a entrada é consumida
- Caso contrário a rotina retorna 0 (falso) e nenhuma entrada é consumida
- Os operadores lógicos (&&) e (||) são usados para requerer a presença de outros símbolos na entrada (&&), ou para representar uma alternativa (||)
- Se um membro de uma regra deve estar presente e não está, isso significa erro de sintaxe

PROBLEMAS DO AN. SINT. DESCENDENTE RECURSIVO

- O método geralmente exige retrocesso nas alternativas.
- Considere a seguinte gramática

```
S -> cAd
A -> ab| a
```

- e a cadeia de entrada *w* = *cad*



Problemas do An. Sint. Descendente Recursivo

- Primeiro, o apontador de entrada aponta para *c*, o primeiro caracter de entrada. A primeira regra S é escolhida e a árvore (1) é gerada
- O *c* é reconhecido pela folha à esquerda, e então a regra A é expandida gerando (2)
- O *a* é então reconhecido pela folha mais à esquerda, porém o próximo símbolo de entrada é *d*, e na árvore o próximo símbolo é o *b*
- Ocorre então o retrocesso, e a próxima alternativa de A é expandida gerando (3) e a entrada é reconhecida

Problemas do An. Sint. Descendente Recursivo

- Muitas vezes o método deixa de produzir um analisador correto
- Exemplo:

```
termo -> IDENTIFICADOR | elemento_indexado | ...
elemento_indexado -> IDENTIFICADOR '[' expressao ']'
```

  - A sequência `elemento_indexado` nunca será experimentada, já que `IDENTIFICADOR` será consumido deixando o resto `'[' expressao ']'` de fora

Problemas do An. Sint. Descendente Recursivo

- Exemplo 2: Gramática para reconhecer ab e aab

```
S -> A 'a' 'b'
A -> 'a' | epsilon

int S(void) {
  return (A() && require (token ('a')) &&
         require (token('b')));
}
int A(void){
   return token ('a') || 1;
}
```

  - ab não será reconhecido: `A()` consome `'a'` e o `require(token('a'))` falha.

Problemas do An. Sint. Descendente Recursivo

- Exemplo 3: Recursão à esquerda

```
expressao -> expressao '-' termo | termo

int expressao(void) {
  return (expressao () && require(token('-'))
      && require (termo()) || require (termo()));
}
```

  - Loop infinito em `expressao()`

ELIMINAÇÃO DA RECURSÃO À ESQUERDA

- Uma gramática é recursiva à esquerda se existe um não terminal A e exista, direta ou indiretamente uma derivação `A -> Aa`
- Análises *top-down* não conseguem lidar com recursão à esquerda, por isso é necessário um método que a elimine
- Um par de produções recursivas á esquerda do tipo

  ```
  A -> Aa | B
  ```

  poderia ser substituído pelas produções não recursivas

  ```
  A -> BA'
  A'-> aA' | epsilon
  ```

Eliminação da Recursão à esquerda

- Exemplo: Considere a seguinte gramática para expressões aritméticas

```
E -> E + T | T
T  -> T * F | F
F -> (E) | identificador
```

- Eliminado a recursão à esquerda, temos:

```
E  -> TE'
E' -> +TE' | epsilon
T  -> FT'
T' -> *FT' | epsilon
F  -> (E)  | identificcador
```

ELIMINAÇÃO DA RECURSÃO À ESQUERDA

- Exemplo: Expressões aritméticas apresentada anteriormente

```
expressao -> expressao '-' termo | termo
```

- Eliminado a recursão à esquerda, temos:

```
expressao       -> termo expressao_final
expressao_final -> '-' termo expressao_final
                 | epsilon
```

ELIMINAÇÃO DA RECURSÃO À ESQUERDA

- Cuidado: Recursão à esquerda indireta:

```
S -> AA | a
A -> SS | b
```

- Existem algorítmos para eliminar a recursão indireta

FATORAÇÃO À ESQUERDA

- É uma transformação que pode ser aplicada quando duas regras começam exatamente com o mesmo símbolo gramatical
- A idéia básica é postergar a decisão até quando tenhamos visto o suficiente da entrada para que possamos realizar a escolha certa
- Exemplo:

```
A -> aB1 | aB2
```

- Fatorando à esquerda obtemos:

```
A  -> aA'
A' -> B1 | B2
```

FATORAÇÃO À ESQUERDA

- Exemplo:

```
cmd -> IF expr THEN cmd ELSE cmd
     |  IF expr THEN cmd
```

- Fatorando-se à esquerda:

```
cmd  -> IF expr THEN cmd cmd'
cmd' -> ELSE cmd | epsilon
```

FATORAÇÃO À ESQUERDA

- Exemplo do compilador Descendente Recursivo:

```
termo -> IDENTIFICADOR
         | IDENTIFICADOR '[' expressao ']' | ...
```

- Fatorando-se à esquerda:

```
termo     -> IDENTIFICADOR id_depois | ...
id_depois -> '[' expressao ']' | epsilon
```

SUBSTITUIÇÃO

- A substituição envolve substituir um não terminal `N` no lado direito de uma regra de produção pelas alternativas de N
- Ela é usada quando as entidades em conflito não são diretamente visiveis
- Exemplo

```
S -> p A q
A -> a | B c | epsilon
```

- Por substituição:

```
S -> p a q | p B c q | p q
```

SUBSTITUIÇÃO

- Exemplo

```
termo -> IDENTIFICADOR | elem_indexado | ..
elem_indexado -> IDENTIFICADOR '[' expressao ']'
```

- Por substituição:

```
termo -> IDENTIFICADOR
        | IDENTIFICADOR '[' expressao ']' | ...
```

- Agora temos um conflito direto que pode ser resolvido por fatoração à esquerda

SUBSTITUIÇÃO

- Problema apresentado anteriormente

```
S -> A 'a' 'b'
A -> 'a' | epsilon
```

- Por substituição:

```
S -> A 'a' 'b' | 'a' 'b'
```

Análise Recursiva Preditiva

- É possível implementar analisadores recursivos sem retrocesso
- A produção a ser aplicada é escolhida baseando-se no símbolo de entrada (análise preditiva)
- A gramática não pode ter recursividade à esquerda, deve estar fatorada à esquerda e para os não-terminais com mais de uma regra de produção, os primeiros terminais deriváveis devem ser capazes de identificar, sem equivocos, qual produção deve ser aplicada

Análise Recursiva Preditiva

- Dada uma entrada **a** e o não terminal A, deve-se saber qual das produções alternativas `A -> B1|B2|B3|...` deriva a sequência que inicia por **a**
- Temos então que calcular os conjuntos de primeiros símbolos produzidos por todas as alternativas na gramática, ou seja, os conjuntos FIRST
- O conjunto FIRST, de uma alternativa α, FIRST(α), contém todos os terminais com que α pode começar

ALGORITMO PARA CALCULAR O CONJUNTO FIRST

Inicialização:

1. se a é um terminal então FIRST(a) = a
2. Se `X -> ` ϵ é uma produção, adicione ϵ em FIRST(X)

   Regras de Inferência:
3. Para cada regra `N -> ` α, FIRST(`N`) deve conter FIRST(α)
4. Para cada alternativa α de forma Aβ, FIRST(α) deve conter FIRST(A) menos o ϵ se este o contém
5. Para cada alternativa α de forma Aβ, e FIRST(A) que contém ϵ, FIRST(α) deve conter FIRST(β) inclusive ϵ se este o contém

EXEMPLO DE CONJUNTO FIRST

Dada a Gramátia:

```
COMANDO -> CONDICIONAL | ITERATIVO | ATRIBUICAO
CONDICIONAL -> if EXPR then COMANDO
ITERATIVO -> repeat LISTA until EXPR |
             while EXPR do COMANDO
ATRIBUICAO-> id := EXPR
```

- FIRST (CONDICIONAL) = if
- FIRST (ITERATIVO) = while,repeat
- FIRST (ATRIBUICAO) = id
- FIRST (COMANDO) = if, while, repeat, id

IMPLEMENTANDO A FUNCAO COMANDO

```
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
```

FUNÇÃO COMANDO

- Na função `COMANDO` agora não existe mais *backtracking*, ou *retrocesso*
- Cada alternativa é escolhida baseando-se no caracter de entrada
- Por isso este analisador é chamado de Analisador Recursivo Preditivo

PROBLEMAS COM O CONJUNTO FIRST

- Regras do tipo: `A -> ` ϵ, não podem ser reconhecidas
- Como a Regra `A` não inicia com nenhum símbolo, como podemos determinar se ela é a alternativa correta?
- Regra: Quando `N` produz um string não-vazio, vemos um token com o qual `N` pode se iniciar; quando `N` produz um string vazio, vemos o símbolo que pode *seguir* `N`
- Usa-se então o conjunto FOLLOW das alternativas

ALGORITMO PARA CALCULAR O CONJUNTO FOLLOW

1. Se S é o símbolo inicial e $ representa fim de arquivo (E0F), então $ está em FOLLOW(S)
2. Se existe produção do tipo A -> αXβ, então FIRST(β) está em FOLLOW(X)
3. Se existe produção do tipo A -> αX ou A -> αXβ, sendo que β gera ϵ, então todos os terminais que estiverem em FOLLOW(A) fazem parte de FOLLOW(X)

EXEMPLO: CALCULANDO FIRST

```
E  -> TE'
E' -> +TE' | epsilon
T  -> FT'
T' -> *FT'| epsilon
F  -> -F | id
```

- Começar sempre pelos não terminais mais fáceis. FIRST(F) = $\{$-,id$\}$
- FIRST(T') = $\{$*,$\epsilon\}$ e similarmente FIRST(E') = $\{$+,$\epsilon\}$
- Como T deriva FT' e F não deriva a palavra vazia, FIRST(T) = FIRST(F)= $\{$-,id$\}$
- Como E deriva TE' e T não deriva ϵ, FIRST(E) = FIRST(T) = $\{$-,id$\}$

EXEMPLO: CALCULANDO FOLLOW

- FOLLOW(E) = $\{\$\}$ pela regra 1, e FOLLOW(E') = FOLLOW(E) = $\{\$\}$, pois é o último símbolo do lado direito da regra E-$>$ TE'
- partindo da produção E' -$>$ +TE', FOLLOW(T) é obtido pela união de FIRST(E') pela regra 2, com FOLLOW(E') pela regra 3, já que E' -$>$ ϵ. Então FOLLOW(T) = $\{+,\$\}$
- FOLLOW(T') = FOLLOW(T)= $\{+,\$\}$, pois T' é o último na produção T-$>$FT'
- FOLLOW(F) é obtido pela união entre FIRST(T'), pela regra 2, com FOLLOW(T'), pela regra 3, então FOLLOW(F) = $\{+,*,\$\}$

Análise Preditiva Tabular

- É possível se construir analisadores preditivos *não recursivos* que utilizam uma pilha explícita, ao invés de chamadas recursivas
- Esse analisador implementa um autômato de pilha controlado por uma tabela de análise
- O analisador busca a produção a ser aplicada em uma tabela de análise, levando em conta o não-terminal no topo da pilha e o token sobre o cabeçote de leitura

ANÁLISE PREDITIVA TABULAR

- Exemplo, Gramática

```
E  -> TE'
E' -> +TE' | epsilon
T  -> FT'
T' -> *FT'| epsilon
F  -> - F | id
```

- E a tabela gerada usando os conjuntos FIRST and FOLLOW:

TABELA DE ANÁLISE

	id	+	*	-	$
E	E -> TE’			E -> TE’	
E’		E’->+TE’			E’-> ϵ
T	T->FT’			T->FT’	
T’		T’->ϵ	T’->*FT’		T’-> ϵ
F	F-> id			F-> -F	

Análise Preditiva Tabular

PILHA	ENTRADA	AÇÃO
$E	id+id*id$	E->TE’
$E’T	id+id*id$	T->FT’
$ET’F	id+id*id$	F->id
$ET’id	id+id*id$	desemplilha e lê símbolo
$E’T’	+id*id$	T’->ϵ
$E’	+id*id$	E’->+TE’
$E’T+	+id*id$	desemplilha e lê símbolo
$E’T	id*id$	T->FT’
$E’T’F	id*id$	F->id
$E’T’id	id*id$	desemplilha e lê símbolo
$E’T’	*id	T’->*FT’

Análise Preditiva Tabular (Continuação...)

PILHA	ENTRADA	AÇÃO
$E'T'F*	*id$	desemplilha e lê símbolo
$E'T'F	id$	F->id
$E'T'id	id$	desemplilha e lê símbolo
$E'T'	$	T'->ϵ
$E'	$	E'->ϵ
$	$	Aceita setença

Análise Sintática Bottom-Up

- A análise Bottom-up tenta construir uma árvore gramatical para uma cadeia de entrada começando pelas folhas, e trabalhando árvore acima em direção à raiz
- A cada passo de *redução*, uma subcadeia particular que reconheça o lado direito de uma produção, é substituída pelo símbolo á esquerda daquela produção
- Esse tipo de análise geralmente é implementada usando o método de empilhar e reduzir
- Vários geradores automáticos de parser são implementados dessa maneira, incluindo o famoso Yacc

ANÁLISE SINTÁTICA BOTTOM-UP

- Considerando a gramática

```
S -> aABe
A -> Abc | b
B -> d
```

- A sentença *abbcde* pode ser reduzida a S pelos seguintes passos:

```
abbcde -> aAbcde -> aAde -> aABe -> S
```

Implementação de Pilha da An. Empilhar e Reduzir

- Uma forma conveniente de implementar um analisador sintático de empilhar e reduzir é usar uma pilha para guardar os símbolos gramaticais e um buffer de entrada prar a cadeia *w* a ser decomposta

```
Pilha        Entrada
$                 w$
```

- O analisador sintático empilha zero ou mais símbolos na pilha até que um *handle* (lado direito de uma produção) surja no topo da pilha. Este então é substituido por seu lado esquerdo. Este ciclo é repetido até termos uma entrada vazia e o símbolo inicial no topo da pilha

```
Pilha        Entrada
$S                $
```

ANALISADOR SINTÁTICO DE EMPILHAR E REDUZIR

- Considerando a seguinte gramática

```
E -> E + E
E -> E * E
E -> (E)
E -> id
```

Analisador sintático de Empilhar e Reduzir

PILHA	ENTRADA	AÇÃO
$	id+id*id$	empilhar
$id	+id*id$	reduzir
$E	+id*id$	empilhar
$E+	id*id$	empilhar
$E+id	*id$	reduzir
$E+E	*id$	reduzi r
$E	*id$	empilhar
$E*	id$	empilhar
$E*id	$	reduzi r
$E*E	$	reduzir
$E	$	aceitar