Análise Sintática

A sintaxe das construções de uma linguagem de programação pode ser descrita por gramáticas livres de contexto ou pela notação BNF ou EBNF.

Vantagens oferecidas pelas gramáticas:

- Oferece uma especificação sintática precisa de fácil compreensão.
- Podemos construir automaticamente um analisador sintático para algumas classes de gramática.
- Uma gramática pode ser uma estrutura útil à tradução correta de programas-fonte em código objeto e também à detecção de erros.
- As novas construções de uma linguagem são facilmente incluídas quando existe uma implementação baseada em uma descrição gramatical da linguagem.

O papel do analisador sintático

- O analisador sintático obtém uma cadeia de tokens provenientes do analisador léxico e verifica se a mesma pode ser gerada pela gramática da linguagem-fonte.
- O analisador sintático deve relatar quaisquer erros de sintaxe que ocorram e deve recuperar-se dos erros mais comuns para continuar a verificar o restante da entrada.
- Existem 3 tipos gerais de analisadores sintáticos:

  Os métodos universais de análise sintática (algoritmos de Cocker-Younger-Kasami e de Earley) – tratam qualquer gramática e são muito ineficiente para se usar em um compilador de produção.

  Top-Down: constrói a árvore gramatical do topo para as folhas;

  Bottom-up: constrói a árvore gramatical das folhas para a raiz.

Os analisadores implementados manualmente trabalham freqüentemente com gramáticas LL.

Os da classe mais ampla das gramáticas LR são normalmente construídos através de ferramentas automatizadas.

A saída do analisador sintático é uma árvore de análise sintática para o fluxo de tokens produzido pelo analisador léxico.

Outras tarefas que podem ser conduzidas durante a análise sintática:

- Coletar informações sobre os tokens;
- Realizar a verificação de tipos;
- Gerar o código intermediário.

Tratamento dos erros de sintaxe

A tarefa de descrever como o compilador deve responder aos erros é deixada para o projetista do compilador.

Grande parte da detecção e recuperação de erros num compilador gira em torno da fase de análise sintática.

O tratador de erros num analisador sintático possui metas simples a serem estabelecidas:

- Relatar a presença de erros de forma clara.
- Recuperar-se de erros de forma rápida.
- Não deve retardar o processamento de programas corretos.

Os métodos de análise LL e LR detectam o erro tão cedo quanto possível (prefixo viável).

Um tratador de erros deve no mínimo informar o local no programa-fonte onde o erro foi detectado. Uma estratégia comum é imprimir a linha ilegal com um apontador para a posição na qual o erro foi detectado.

Existem formas de recuperação de erros onde o analisador tentar restaurar a si mesmo para um estado onde o processamento da entrada possa continuar.

Um trabalho inadequado de recuperação pode levar a uma série de erros espúrios.

Estratégias de recuperação de erros

- Método do pânico: ao descobrir um erro, o analisador sintático descarta símbolos de entrada, um de cada vez, até que seja encontrado um token pertencente a um conjunto designado

de tokens de sincronização, que normalmente são delimitadores.

- Recuperação de frases: ao descobrir um erro o analisador sintático pode realizar uma correção local na entrada restante. Correções locais típicas – substituir uma vírgula por um ponto e vírgula, remover um ponto e vírgula estranho ou inserir um ausente.
- Produções de erro: se tivéssemos uma boa idéia dos erros comuns que podem ser encontrados, poderíamos aumentar a gramática com produções que gerassem construções ilegais.

- Correção global: Neste caso usaríamos algoritmos para detectar uma seqüência mínima de mudanças a fim de obter uma correção global de menor custo.

Gramáticas livres de contexto

Consiste de símbolos terminais, não-terminais, um símbolo de partida e produções.

1. Os terminais são símbolos básicos que formam as cadeias. Aqui chamamos estes símbolos de tokens.
2. Os não-terminais são variáveis sintáticas que denotam cadeias de tokens.
3. Um não-terminal é um símbolo de partida, e o conjunto de cadeias que o mesmo denota é a linguagem definida pela gramática.
4. As produções especificam a forma pela qual os terminais e não-terminais podem ser combinados a fim de formar cadeias. Cada produção consiste de um não-terminal, seguido de uma seta (ou ::=), seguido por uma cadeia de não-terminais e terminais.

Convenções da notação

1. Símbolos terminais:
   i) Letras minúsculas do início do alfabeto, tais como a, b, c.
   ii) Símbolos de operadores, tais como +, -.
   iii) Símbolos de pontuação, tais como parênteses, vírgula.
   iv) Os dígitos 0, 1, ..., 9.
   v) Cadeias em negrito como **id** ou **if**.

2. Símbolos não-terminais
   i) Letras maiúsculas do início do alfabeto, tais como A, B, C.
   ii) A letra S é normalmente usada como símbolo de partida.
   iii) Os nomes em itálico formados por letras minúsculas, como *expr* ou *cmd*.
3. As letras maiúsculas do final do alfabeto, tais como X, Y, Z, representam símbolos gramaticais.

4. Letras minúsculas, ao fim do alfabeto, representam cadeias terminais.
5. Letras gregas minúsculas, α, β, γ, representam cadeias de símbolos gramaticais.
6. Se $A \rightarrow \alpha_1$, $A \rightarrow \alpha_2$, ..., $A \rightarrow \alpha_n$, podemos escrever $A \rightarrow \alpha_1 \mid \alpha_2 \mid ... \mid A \rightarrow \alpha_n$.
7. A menos que seja explicitamente estabelecido, o lado esquerdo da primeira produção é o símbolo de partida.

Derivações

É uma descrição precisa do processo Top-Down da construção da árvore gramatical.

O símbolo não-terminal à esquerda é substituído pela cadeia do lado direito da produção.

Exemplo: E → E + E | E * E | (E) | -E | **id**

$E \Rightarrow -E \Rightarrow -(E) \Rightarrow -(\ \mathbf{id}\ )$

$\overset{*}{\Rightarrow}$ - deriva em zero ou mais passos

$\overset{+}{\Rightarrow}$ - deriva em um ou mais passos

Dizemos que uma cadeia de terminais w está em L(G) se, e somente se, $S \overset{+}{\Rightarrow} w$.

Se $S \overset{*}{\Rightarrow} \alpha$, onde α pode conter não terminais, dizemos que α é uma forma sentencial de G. Uma sentença é uma forma sentencial sem não-terminais.

Exemplo: a cadeia –(**id** + **id**)

A cada passo em uma derivação, temos:

1. escolher qual não terminal substituir.
2. escolher qual alternativa usar na substituição daquele não-terminal.

Derivações mais à esquerda: derivações nas quais somente o não-terminal mais a

esquerda em qualquer forma sentencial é substituída a cada passo.

Ex.: Produção mais à esquerda de –(**id+id**)

Derivações mais à direita : derivações nas quais somente o não-terminal mais a direita em qualquer forma sentencial é substituída a cada passo.

Árvores Gramaticais e Derivações

Considere a derivação $\alpha_1 \Rightarrow \alpha_2 ... \Rightarrow \alpha_n$, onde α_1 é um não terminal único [a]

Para cada forma sentencial α_i na derivação construímos uma árvore gramatical.

Exemplo: - (**id+id**)

Ambigüidade

Uma gramática ambígua é aquela que produz mais do que uma derivação à esquerda, ou à direita, para a mesma sentença.

Escrevendo uma gramática

Uma parte limitada da análise sintática é realizada pelo analisador léxico.
Certas exigências, tais como, os identificadores devem ser declarados antes de serem usados, não podem ser descritas por uma gramática livre de contexto.
As fases subseqüentes utilizam a saída do analisador sintático para assegurar a concordância com as regras que não são verificadas pelo analisador sintático.

Expressões Regulares vs Gramática Livre de Contexto

Cada construção que pode ser descrita por uma expressão regular pode também ser descrita por uma gramática. Que pode ser gerada a partir de um AFN da expressão regular.
Ex: (a | b)*abb

Por que usar expressões regulares para definir a estrutura léxica da linguagem?

1. As regras léxicas são normalmente simples.
2. As expressões regulares normalmente providencial para os tokens da gramática, uma notação mais concisa e facilmente compreendida.
3. A partir de expressões regulares podem ser construídos

automaticamente analisadores léxicos mais eficientes do que a partir de gramáticas arbitrárias.

4. A separação da estrutura sintática da linguagem nas partes léxicas e não-léxicas fornecem uma forma conveniente de modularizar o front-end de um compilador em componentes facilmente dimensionados.

Eliminando a ambigüidade

cmd → **if** *expr* **then** *cmd*
| **if** *expr* **then** *cmd* **else** *cmd*
| **outro**

Exemplo:
if E_1 **then** S_1 **else if** E_2 **then** S_2 **else** S_3
if E_1 **then if** E_2 **then** S_1 **else** S_2

A regra geral é: associar cada **else** ao **then** anterior mais próximo ainda não associado.
Para torná-la não ambígua, a idéia está em que um enunciado figurando entre um **then** e um **else** precisa ser associado, isto é, não pode terminar com um **then** ainda não associado seguido por qualquer outro enunciado, pois o **else** seria forçado a se associar a esse **then** não associado.

Um enunciado associado ou é um enunciado **if-then-else** contendo somente enunciados associados ou é qualquer outro tipo de enunciado não condicional. Assim,

cmd → *cmd_associado*
| *cmd_não_associado*
cmd_associado → **if** *expr* **then** *cmd_associado* **else** *cmd_associado*
| **outro**
cmd_não_associado → **if** *expr* **then** *cmd*
| **if** *expr* **then** *cmd_associado* **else** *cmd_não_associado*
| **outro**

Associatividade dos operadores

9 + 5 + 2 é equivalente a (9 + 5)+ 2, associa à esquerda.
9 – 5 + 2 é equivalente a (9 – 5)+ 2, associa à esquerda.
9 * 5 * 2 é equivalente a (9 * 5)* 2, associa à esquerda.
9 / 5 * 2 é equivalente a (9 / 5)* 2, associa à esquerda.
a = b = c é equivalente a a = (b = c), associa à direita.
a ^ b ^ c é equivalente a a ^ (b ^ c), associa à direita.

Se a expressão é associada a esquerda a regra que a produz deve ter a recursão a esquerda.

Se a expressão é associada a direita a regra que a produz deve ter a recursão a direita.

Ex.:

Soma
E → E + T
| T
T → **id**
| **num**

Atribuição
A → L = **id**
| **id**

Prioridade dos operadores

9 + 5 * 2, a associatividade da soma e da multiplicação não resolve a prioridade.
Cada não terminal deve resolver um nível de prioridade.
Ex.:
E → E + F | E - F | E * F | E / F | F
F → (E) | **id** | **num**
Transforma-se em:
E → E + T | E - T | T
T → T * F | T / F | F
F → (E) | **id** | **num**

Exercício: Defina uma GLC, no formato BNF, para formar uma lista de identificadores separados por vírgula, associada à esquerda.

Analisa Sintática Top-Down

Tipos:

1. Analisador sintático top-down recursivo sem retrocesso;
2. Analisador sintático preditivo.

Análise sintática top-down recursiva

Pode envolver retrocesso.
Pode ser vista como uma tentativa de se encontrar a árvore de análise sintática, para a seqüência de tokens da entrada, a partir da raiz, cirando os nodos em pré-ordem.
Ex.: S → cAd
A → ab | a
Como reconhecer 'cad'?

Analisador sintático top-down recursivo sem retrocesso

Pode ser construído a partir de uma gramática cuidadosamente definida e sem recursão a esquerda e fatorada.

Ex.: tipo → tipo_simple

| ^ **id**
| **array [** tipo_simples **] of** tipo
Tipo_simples → **integer**
| **char**
| **num . . num**

O reconhecimento de uma seqüência de tokens pelas regras acima podem ser implementados da seguinte forma:

```
Seja lookahead o token corrente.
Seja o procedimento Reconhecer como:
    Procedimento Reconhecer( t : ClasseToken);
    Início
        Se lookahead.classe = t então
            lookahead = obterProximoToken();
        senão Erro;
    Fim;
```

Faça um procedimento para cada não terminal, onde cada procedimento:

1. Decide qual regra utilizar no reconhecimento. Verifica-se com quais símbolos as palavras geradas pela regra podem iniciar;
2. Usa uma regra imitando o lado direito. Um não terminal resulta em

uma chamada de procedimento. Um terminal (token) resulta na chamada ao procedimento Reconhecer, passando a classe a ser reconhecida.

Ex.:

```
Procedimento tipo;
Inicio
        Se lookahead.classe pertence a {integer,
char, num} então
            Tipo_simples
        Senão se lookahead.classe = '^' então
        Inicio
            Reconhecer( '^' );
            Reconhecer( id );
        Fim
        Senão se lookahead.classe = array então
        Início
            Reconhecer(array);
            Reconhecer( '[' );
            tipo_simples;
            Reconhecer( ']' );
            Reconhecer( of );
            Tipo;
        Fim
        Senão Erro;
Fim;
```



```
Procedimento tipo_simples;
Início
        Se lookahead.classe = integer então
            Reconhecer( integer )
        Senão Se lookahead.classe = char então
            Reconhecer( char )
        Se lookahead.classe = num então
        início
            Reconhecer( num );
            Reconhecer( '.' );
            Reconhecer( '.' );
            Reconhecer( num );
        Fim
        Senão Erro;
    Fim;
```

Exercício: Monte uma árvore de execução, usando os procedimentos acima, para reconhecer 'array [1 . . 10] of integer' e verifique a semelhança com a árvore de análise sintática.

Exercício: Tente construir uma analisador sintático top-down recursivo

sem retorcesso para a gramática a seguir:

E → E **+** T | E **–** T | T
T → T ***** F | T **/** F | F
F → **(** E **)** | **id** | **num**

Quais os problemas encontrados?
Como resolvê-los?

Eliminação da recursão à esquerda

Os métodos de análise top-down não podem processar gramáticas recursivas à esquerda.

$A \rightarrow A\alpha \mid \beta$ pode ser substituído por

$A \rightarrow \beta A'$

$A' \rightarrow \alpha A' \mid \varepsilon$

Exemplo: $E \rightarrow T+E \mid T$

$T \rightarrow T*F \mid F$

$F \rightarrow (E) \mid$ **id**

Generalizando,

$A \rightarrow A\alpha_1 \mid A\alpha_2 \mid \ldots \mid A\alpha_m \mid \beta_1 \mid \beta_2 \mid \ldots \mid \beta_n$

é substituído por

$A \rightarrow \beta_1 A' \mid \beta_2 A' \mid \ldots \mid \beta_n A'$

$A' \rightarrow \alpha_1 A' \mid \alpha_2 A' \mid \ldots \mid \alpha_m A' \mid \varepsilon$

O procedimento anterior não elimina recursão em 1 ou mais passos.

Ex.: $S \rightarrow Aa \mid b$ $A \rightarrow Ac \mid Sd \mid \varepsilon$

Algoritmo – Eliminação da recursão à esquerda.

Entrada: Uma gramática G sem ciclos ou produções-ε.

Saída: Uma gramática equivalente sem recursão à esquerda.

Método: Aplicar o algoritmo:

1. Colocar os não terminais em alguma ordem $A_1, A_2, \ldots, A_n$
2. **Para** i := 1 **até** n **faça início**
   Para j := 1 **até** i−1 **faça início**
   Substituir cada produção da forma $A_i \rightarrow A_j \gamma$ pelas produções $A_i \rightarrow \delta_1 \gamma \mid \delta_2 \gamma \mid \ldots \mid \delta_k \gamma$, onde $A_j \rightarrow \delta_1 \mid \delta_2 \mid \ldots \mid \delta_k$ são todas as produções A_j correntes;
   fim
   eliminar a recursão imediata à esquerda entre as produções-A_i
   fim

Exemplo: $S \rightarrow Aa \mid b$

$A \rightarrow Ac \mid Sd \mid \varepsilon$

Exercícios: Elimine a recursão à esquerda de: $S \rightarrow (L) \mid a$

$L \rightarrow L, S \mid S$

Fatoração à esquerda

É útil para a análise sintática presciente (preditiva).

Idéia: quando tiver duas produções possíveis a seguir a partir de um não terminal A, reescrevem-se as produções A e adia a decisão até que tenhamos visto o suficiente da entrada para realizar a escolha certa.

Ex.: *cmd* → **if** *expr* **then** *cmd* **else** *cmd*

| **if** *expr* **then** *cmd*

$A \rightarrow \alpha\beta_1 \mid \alpha\beta_2 \mid \gamma$

$A \rightarrow \alpha A' \mid \gamma$

$A \rightarrow \beta_1 \mid \beta_2$

Ex.: Fatore à esquerda as gramáticas a seguir.

a) $S \rightarrow iEtS \mid iEtSeS \mid a$

$E \rightarrow b$

b) $S \rightarrow abA \mid abB$

$A \rightarrow aA \mid \varepsilon$

$B \rightarrow bB \mid b$

Análise Sintática Top-Down
Análise Sintática de Descendência Recursiva

Pode envolver retrocesso.

Pode ser vista como uma tentativa de se encontrar a árvore gramatical para a cadeia de entrada, a partir da raiz, criando nós da árvore gramatical em pré-ordem.

Ex.: $S \rightarrow cAd$

$A \rightarrow ab \mid a$

w = cad

Analisadores Sintáticos Prescientes (Preditivos)

É um analisador de descendência recursiva sem retrocesso.
Pode-se obter uma nova gramática processável por um analisador presciente, escrevendo cuidadosamente a gramática, eliminando a recursão à esquerda e fatorando-se à esquerda.
Ex.: *cmd* → **if** *expr* **then** *cmd* **else** *cmd*
| **while** *expr* **do** *cmd*
| **begin** *lista_de_comandos* **end**

Diagrama de transição para analisadores sintáticos prescientes

Existe um diagrama para cada não terminal.
Os rótulos das arestas são tokens e não terminais.
Uma transição em um token significa que devemos realizá-la se aquele token for o próximo símbolo da entrada.
Uma transição em um não terminal *A* é uma chamada a um procedimento A.
Depois de eliminar a recursão à esquerda e fatorar, para cada não terminal A, faça:

1. Cria-se um estado inicial e final.
2. Para cada produção $A \rightarrow X_1 X_2 \ldots X_n$ cria-se um percurso a partir do estado inicial até o final, com as arestas rotuladas $X_1, X_2, \ldots, X_n$.

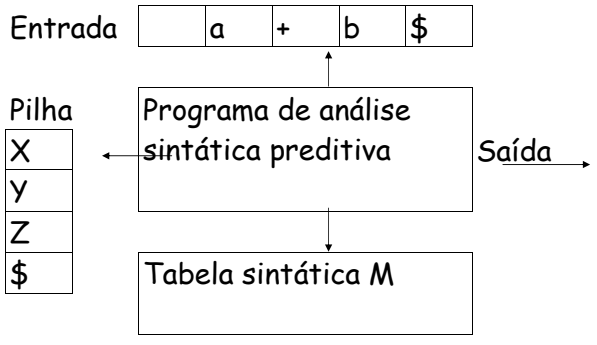
Começa no símbolo inicial para o símbolo de aceitação.
Se está no estado *s* e possui uma aresta a para t e o próximo símbolo da entrada é a, mova o cursor da entrada e vai para t.
Se for rotulado pelo não terminal A, vai para o estado de partida de A sem mover o cursor.
Se existir um lado de *s* para t rotulado ε, vai, a partir de *s*, imediatamente para t sem avançar o cursor na entrada.
Ex.: E→ TE'
E'→ +TE'|ε
T → FT'
T'→ *FT'|ε
F → (E) | **id**
Fazer o diagrama de transição.

Análise sintática preditica não recursiva

Pode-se manter explicitamente uma pilha.

Problema chave: Qual produção dever ser aplicada a um não terminal.

O analisador não recursivo procura pela produção a ser aplicada em uma tabela sintática.



A tabela sintática é um array bidimensional M[A,a], onde A é um não terminal e a é um símbolo terminal ou $.

Inicialmente, a pilha contém acima do $ o símbolo de partida da gramática. Considerando X o símbolo do topo da pilha e a o símbolo corrente, há 3 possibilidades:

1. Se X=a=$, o analisador para e anuncia o término com sucesso da análise sintática.
2. S X=a ≠ $, o analisador sintático remove X da pilha e avança o apontador da entrada para o próximo símbolo.
3. S X é um não terminal, o programa consulta a entrada M[X,a] da tabela sintática M. Essa entrada será uma produção X da entrada ou um erro.

Algoritmo:

Entrada: Uma cadeia w uma tabela sintática M para a gramática G.

Saída: Se w estiver em L(G), uma derivação mais a esquerda de w; caso contrário uma indicação de erro.

Faça ip apontar para o primeiro símbolo de w$;
Repetir
Seja X o símbolo ao topo da pilha e a o símbolo apontado por ip;
Se X for um terminal ou $ **faça**
Se X = a **então**
Remove X da pilha e avança ip
Senão erro()
Senão Se M[X,a] = $X \rightarrow Y_1Y_2...Y_k$ **então**
início
Remover X da pilha;
Empilhar $Y_k, Y_{k-1}, ..., Y_1$;
Escrever a produção $X \rightarrow X \rightarrow Y_1Y_2...Y_k$
Fim
Senão erro()
Até X = $

Exemplo:

Não terminal	Símbolo de entrada					
	id	+	*	(	)	$
E	E → TE'			E→TE'		
E'		E → +TE'			E→ε	E→ε
T	T→ FT'			T→FT'		
T'		T' → ε	T'→*FT'		T'→ε	T'→ε
F	F→**id**			F→(E)		

id + id * id

First(Primeiro) e Follow(seguinte ou próximo)

Primeiro(α) – conjunto de terminais que iniciam as cadeias derivadas a partir de α. Se α deriva ε, então ε também está em Primeiro (α).

Seguinte(A) – o conjunto de terminais a que podem figurar à direita de A em alguma forma sentencial. Se A for o símbolo mais à direita em alguma forma sentencial, então $ está em Seguinte(A)

Primeiro (X)

1. Se X for um não terminal, então Primeiro(X) é {X}.
2. Se X→ ε for uma produção, adicionar ε a Primeiro(X).
3. Se X for um não terminal e $X \rightarrow Y_1 Y_2 ... Y_k$ uma produção, colocar a em Primeiro(X) se, para algum i, a estiver em Primeiro(Y_i) e ε estiver em todos os Primeiro(Y_1), ..., Primeiro(Y_{i-1}).

Seguinte(A) –Aplique as seguintes regras até que nada mais possa ser adicionado a qualquer conjunto Seguinte.

1. Colocar $ em Seguinte(S), onde S é o símbolo de partida e $ é o marcador de fim de entrada à direita.
2. Se existir uma produção $A \rightarrow \alpha B \beta$, então tudo em Primeiro(β), exceto ε, é colocado em Seguinte(B).

3. Se existir uma produção $A \rightarrow \alpha B$ ou uma produção $A \rightarrow \alpha B \beta$, onde Primeiro($\beta$) contém ε ($\beta \Rightarrow^{*} \varepsilon$), então tudo em Seguinte(A) também está em Seguinte(B).

Exemplo: Seja a gramática
E→TE'
E'→+TE' | ε
T→FT'
T'→*FT' | ε
F → (E) | **id**

Construção de tabelas sintáticas prescientes (preditivas)

Algoritmo:
Entrada: Gramática G.

Saída: Tabela sintática M.

Método:
1. Para cada podução $A \rightarrow \alpha$ da gramática, execute os passos 2 e 3.
2. Para cada terminal a em Primeiro(α), adicione $A \rightarrow \alpha$ a M[A,a].
3. Se ε estiver em Primeiro(α), adicione $A \rightarrow \alpha$ a M[A,b], para cada terminal b em Seguinte(A). Se ε estiver em Primeiro(α) e $ em Seguinte(A), adicione $A \rightarrow \alpha$ a M[A,$].
4. Faça cada entrada não definida de M ser **erro**.

Exemplo: Construa a tabela sintática para a gramática do exemplo anterior.

Gramática LL(1)

Se G for recursiva à esquerda ou ambígua, M terá pelo menos uma entrada multiplamente definida.

Exemplo:

Construa a tabela sintática para a gramática:

S→ iEtSS' | a
S'→eS | ε
E → b

Uma gramática cuja tabela sintática não possui entradas multiplamente definidas é dita LL(1). O primeiro L significa a varredura da entrada da esquerda para a direita; o segundo, a produção de uma derivação mais a esquerda; e o 1, o uso de um símbolo de entrada como *lookahead* a cada passo para tomar as decisões sintáticas.

Uma gramática G é LL(1) se, e somente se, sempre que $A \rightarrow \alpha \mid \beta$ forem duas produções distintas de G, vigorarem as seguintes condições:

1. α e β não tiverem, ao mesmo tempo, cadeias começando pelo mesmo terminal a, qualquer que seja a.
2. No máximo um dos dois, α ou β, deriva a cadeia vazia.
3. Se $\beta \Rightarrow \varepsilon$, então α não deriva qualquer cadeia começando por um terminal em Seguinte(A).

A dificuldade principal em se usar a análise preditiva está na escrita de uma gramática para a linguagem fonte, tal que um analisador sintático preditivo possa ser construído a partir da mesma. A eliminação da recursão à esquerda e a fatoração à esquerda tornam a gramática difícil de ler e de usar para traduzir.

Recuperação de erros na análise (presciente) preditiva

Um erro é detectado durante a análise presciente quando o terminal ao topo da pilha não reconhece o próximo símbolo de entrada ou quando o não terminal A está ao topo da Pilha, a é o próximo símbolo de entrada e a entrada da tabela sintática M[A,a] está vazia.

A recuperação na **modalidade do pânico** está baseada na idéia de se pular símbolos na entrada até que surja um token pertencente a um conjunto pré selecionado de tokens de sincronização.

1. Como ponto de partida pode-se colocar todos os símbolos de Seguinte(A) no conjunto de tokens de sincronização para o não terminal A.
2. Não é suficiente usar Seguinte(A) como o conjunto de sincronização para A. Freqüentemente, existe uma estrutura

hierárquica nas construções da linguagem: as expressões aparecem dentro de enunciados, que figuram dentro de blocos e assim por diante. Desta forma, pode-se adicionar palavras-chave que iniciam comandos aos conjuntos de sincronização para os não terminais que geram expressões.

3. Se adicionarmos os símbolos em Primeiro(A) ao conjunto de sincronização para o não terminal A, pode ser possível retornar a análise a partir de A, se um símbolo em Primeiro(A) figurar na entrada.
4. Se um não terminal puder gerar a cadeia vazia, então a produção que deriva ε pode ser usada como *default*.
5. Se um não terminal ao topo da pilha não puder ser reconhecido, uma idéia simples é de removê-lo, emitir uma mensagem informando da remoção e prosseguir a análise sintática.

Exemplo: Usando símbolos de Seguinte e de Primeiro como tokens de sincronização para o exemplo anterior construa a tabela sintática com os tokens de sincronização adicionados.

Usando essa tabela verifique a entrada)**id** * + **id**

Recuperação em nível de frases – é implementada preenchendo-se as entradas em branco da tabela de sintática presciente com apontadores para rotinas de erro. Essas rotinas podem modificar, inserir ou remover símbolos da entrada e emitir as mensagens de erro apropriadas. Podem também remover o topo da pilha.

Análise Sintática Bottom-up

A análise sintática bottom-up é também conhecida como análise de empilhar e reduzir.

Ex.: Considere a gramática

S → aABe A → Abc | b B → d

Passos para reduzir a sentença abbcde a S:

abbcde	aABe
aAbcde	S
aAde	

Que corresponde à derivação mais à direita:

S⇒ aABe⇒ aAde⇒ aAbcde⇒ abbcde

Handles - É uma subcadeia que reconhece o lado direito de uma produção e cuja redução ao não terminal do lado esquerdo da produção representa um passo ao longo do percurso de uma derivação mais à direita.

Exemplo: Considere a gramática

(1) E → E + E (3) E → (E)

(2) E → E * E (4) E → id

Derivação mais a direita:

E ⇒ E + E ⇒ E + E * id ⇒ E + id * id ⇒ id + id * id

A poda do handle

Uma derivação mais a direita pode ser obtida "podando-se os handles". Realizando assim uma redução da forma sentencial.

Exemplo: Para a gramática do exemplo anterior.

Forma sentencial à direita	Handle	Produção redutora
id_1 + id_2 * id_3	**id_1**	E → **id**
E + **id_2 * id_3**	**id_2**	E → **id**
E + E ***** **id_3**	**id_3**	E → **id**
E + E * E	E * E	E → E * E
E + E	E + E	E → E + E
E		

Implementação de Pilha da Análise Sintática de Empilhar e Reduzir

Problemas:

1. Localizar uma subcadeia a ser reduzida numa forma sentencial à direita.
2. Determinar que produção escolher no caso de existir mais de uma produção com aquela cadeia no lado direito.

Um analisador sintático de empilhar e reduzir é implementado usando-se uma pilha para guardar os símbolos gramaticais e um buffer de entrada para uma cadeia w a ser decomposta.

Usamos o $ para marcar o fundo da pilha e o final à direita da entrada.

Inicialmente → Pilha: $ Entrada: w$

O analisador sintático empilha zero ou símbolos até que um handle β ocorra no topo da pilha. Reduz então β ao lado esquerdo da produção apropriada. Isso é feito até que ocorra um erro ou que a pilha e a entrada sejam→ Pilha: $S

Entrada $

Ações possíveis:

Empilhar – o próximo símbolo de entrada é colocado no topo da pilha.

Reduzir – o final de um handle está no topo da pilha, deve-se encontrar o seu início e decidir qual não terminal irá substituir o handle.

Aceitar – término com sucesso.

Erro – um erro sintático ocorreu e chama-se o recuperador de erro.

Exemplo: Entrada **id_1 + id_2 * id_3**

	Pilha	Entrada	Ação
(1)	$	$id_1 + id_2 * id_3$	

Exercício:

1. Usando a gramática,
   S → (L) | a
   L → L, S | S
   a) Construa uma derivação mais a direita para (a, (a,a)) e mostre o handle de cada forma sentencial à direita.
   b) Mostre os passos de um analisador sintático de empilhar e reduzir correspondentes à derivação mais a direita de (a).

Conflitos durante a análise sintática de empilhar e reduzir

Exemplo: cmd → **if** expr **then** cmd
| if expr **then** cmd **else** cmd
| outro

Pilha	Entrada
... if **expr** then **cmd**	else ...**$**

Empilhar ou reduzir?

Exemplo: Seja a gramática

cmd → **id (**lista_de_parametros**)**
cmd → expr := expr
lista_de_parametros → lista_de_parametros, parametro
lista_de_parametros → parametro
parametro → **id**
expr → **id (**lista_expr**)**
expr → **id**
lista_expr → lista_expr, expr
lista_expr → expr

Para um enunciado A(I,J) após empilhar os 3 primeiros tokens, qual redução utilizar?

Análise sintática de precedência de operadores

Gramática de operadores: Não possuem nenhum lado direito com produção ε e com dois não terminais adjacentes.

Exemplo:

E → E+E | E-E | E*E | E/E | (E) | -E | **id**

Definimos 3 relações de precedência:

Relação	Significado
a <· b	a "confere precedência" a b
a ≐ b	a "possui a mesma precedência que" b
a ·> b	a "tem precedência sobre" b

Usando relações de precedência de operadores

Objetivo: delimitar o handle de uma forma sentencial à direita, com <· assinalando o limite à esquerda e ·> assinalando o limite à direita.

Seja uma forma sentencial à direita $\beta_0 a_1 \beta_1 ... a_n \beta_n$, onde cada β_i é um ε ou um único não terminal e cada a_i é um único terminal.

Entre a_i e a_{i+1} vigora no máximo uma relação de precedência.

$ assinala o final da cadeia e $ <· b e b ·> $

Exemplo: Seja a tabela de relações de precedência:

	id	**+**	*****	**$**
id		·>	·>	·>
+	<·	·>	<·	·>
*****	<·	·>	·>	·>
$	<·	<·	<·	

A cadeia **id** + **id** * **id** com as relações de precedência inseridas é $ <· **id** ·> + <· **id** ·> * <· **id** ·> $

Quando nenhuma relação de precedência ocorre entre um par de terminais, então um erro sintático é detectado.

Algoritmo para a análise sintática de precedência de operadores:

```
fazer ip apontar para o primeiro símbolo de w$
repetir para sempre
se $ estiver no topo da pilha e ip apontar para $ então
    retornar
senão inicio
    Seja a o símbolo terminal ao topo da pilha e seja b
       o símbolo apontado por ip na entrada;
    Se a <· b ou a =· b então inicio
        Empilha b;
        Avança ip para o próximo símbolo de entrada;
    Fim
    Senão se a ·> b então  /* reduzir */
        Repetir
            Remover o topo da pilha
        Até que o terminal ao topo da pilha esteja
            relacionado por <· ao terminal mais
            recentemente removido
    Senão erro()
fim
```

Relação de precedência de operadores a partir da associatividade e prioridade

Regras para selecionar handles que reflitam as regras de precedência e associatividade para operadores binários:

1. Se um operador θ_1 possui maior precedência do que o operador θ_2, fazer θ_1 ·> θ_2 e θ_2 <· θ_1.
2. Se θ_1 e θ_2 são operadores de igual precedência, fazer θ_1 ·> θ_2 e θ_2 ·> θ_1, se os operadores forem associativos à esquerda, ou fazer θ_1 <· θ_2 e θ_2 <· θ_1, se os operadores forem associativos à direita.
3. Fazer θ <· **id**, **id** ·> θ, θ <· (, (<· θ,) ·> θ, θ ·>), θ ·> $ e $ <· θ, para todos os operadores θ.
4. Fazer também:

(≐)	$ <· (	$ <· **id**
(<· (	**id** ·> $	) ·> $
(<· **id**	**id** ·>)	) ·>)

Exemplo: Construa uma tabela de precedência para +, -, *, /, ^, **id**, (,), $

Tratando operações unárias

Operadores unários que não são operadores binários – incorpora-o ao esquema acima.

Exemplo: ¬ (negação lógica), fazer ¬ ·> θ, se ¬ tiver a maior precedência, ou fazer ¬ <· θ, caso contrário.
Operadores unários que são usados também como binário – o analisador léxico deve diferenciar entre o operador unário e o binário.

Função de precedência

A tabela de precedência pode ser codificada por duas funções, f e g, que mapeam símbolos terminais em inteiros.

1. f(a) < g(b), sempre que a <· b,
2. f(a) = g(b), sempre que a =· b,
3. f(a) > g(b), sempre que a ·> b.

Exemplo:

	+	-	*	/	^	(	)	id	$
f	2	2	4	4	4	0	6	6	0
g	1	1	3	3	5	5	0	5	0

Construção de funções de precedência

1. Cria os símbolos f_a e g_a para cada a que seja um terminal ou $.
2. Particionar os símbolos criados em tantos grupos quanto sejam possíveis, de tal forma que a =· b, então f_a e g_b estão no mesmo grupo.
3. Cria-se um grafo dirigido cujos nodos são os grupos encontrados em (2). Para quaisquer a e b , se a <· b, colocar uma aresta a partir do

grupo g_b para o de f_a. Se a ·> b colocar uma aresta a partir do grupo f_a para g_b.

4. Se o grafo possuir um ciclo não há função de precedência. Se não existir ciclos fazer f(a) igual ao comprimento do mais longo percurso começando no grupo de f_a e fazer g(a) igual ao comprimento do mais longo percurso começando no grupo de g_a.

Exemplo: Defina a função de precedência para **id**, *, + e $.

Exercício:

1. Para a tabela a seguir, faça:

	A	**(**	**)**	**,**	**$**
a			·>	·>	·>
(	<·	<·	=·	<·	
)			·>	·>	·>
,	<·	<·	·>	·>	
$	<·	<·			

Encontre as funções de precedência de operadores.

Recuperação de erros na análise sintática de precedência de operadores

Descobre um erro se:

1. Nenhuma relação vigorar entre o terminal ao topo da pilha e a entrada corrente.
2. Um *handle* foi encontrado, mas não existe produção que tenha tal *handle* como lado direito.

Tratando erros durante as reduções

Erros Tipo (2)

Pode-se remover da pilha os símbolos de acordo com o algoritmo. Como não produção a reduzir, nenhuma ação é executada e é impresso uma mensagem de erro.

Para determinar o que o diagnóstico deveria informar é necessário decidir com que lado direito de produção se assemelha o handle da pilha.

1. Se +, -, *, / ou ^ for reduzido, verifica se os não terminais aparecem em ambos os lados. Senão, emite o diagnóstico: Operador ausente

2. Se **id** for reduzido, verifica se existe um não-terminal à direita ou à esquerda. Se existir, emite a mensagem: operador ausente.
3. Se ( ) for reduzido, verifica se existe um não-terminal entre os parênteses. Se não, emite a mensagem: nenhuma expressão entre os parênteses.

Tratando erros de Empilhar/Reduzir

Quando nenhuma relação vigora entre o topo da pilha o símbolo corrente da entrada.

Sejam a e b símbolos do topo da pilha (b está no topo), c e d próximos símbolos da entrada e não existe relação de precedência entre b e c.

1. Se $a \lessdot c$, pode-se desempilhar b.
2. Se $b \lessdot d$, pode-se remover c da entrada.
3. Pode-se tentar encontrar um símbolo *e* tal que $b \lessdot e \lessdot c$ e inserir *e* à frente de c na entrada.

Para cada entrada em branco na matriz de precedência é precisa especificar uma rotina de recuperação de erros.

Exemplo:

	id	(	)	$

Id	e3	e3	·>	·>
(	<·	<·	=·	e4
)	e3	e3	·>	·>
$	<·	<·	e2	e1

e1 - /* quando a expressão estiver ausente */
inserir **id** na entrada
emitir diagnóstico: "operando ausente"

e2 - /* quando toda a expressão começa com um parêntese à direita */
desempilha)
emitir diagnóstico: "parêntese à direita não balanceado"

e3 - /* chamado quando **id** ou) é seguido por **id** ou (*/
inserir + na entrada
emitir diagnóstico: "operador ausente"

e4 - /* quando a expressão termina por um parêntese à esquerda */
desempilha (
emitir diagnóstico: "parêntese à direita ausente"

Exemplo: **id** +)

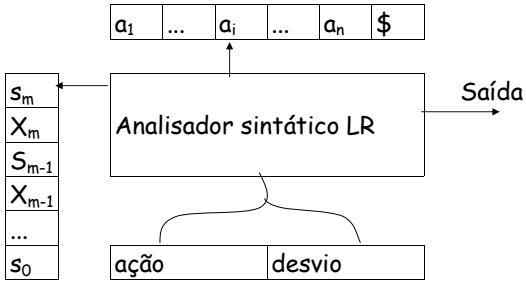
Analisadores Sintáticos LR(k)

L – varredura da entrada da esquerda para a direita .

R – constrói uma derivação mais a direita ao contrário.

K – o número de símbolos de entrada de lookahead que são usados para se tomar decisões na análise sintática.

O algoritmo de análise sintática LR



Lê símbolos terminais de um buffer de entrada.

Usa uma pilha para armazenar as cadeias compostas por X_i, símbolo gramatical, e s_i, estado.

A tabela sintática é composta por Ação e Desvio.

Ação[s_m,a_i]

1. empilha s, onde s é um estado.
2. Reduz através da produção gramatical $A \rightarrow \beta$.
3. Aceita.
4. Erro.

Desvio[s_{m-r},A] – produz um estado como saída.

Algoritmo

Entrada: Uma cadeia de entrada w e uma tabela sintática como as funções ação e desvio para uma gramática G.

Saída: Se w estiver em L(G), uma decomposição bottom-up para w; caso contrário, uma indicação de erro.

Método: Inicialmente, o analisador sintático possui s_0 na pilha e w$ no buffer de entrada.

Fazer *ip* apontar para o primeiro símbolo de w$;
Repetir para sempre início
Seja *s* o estado do topo da pilha e a o símbolo apontado por *ip*;
Se ação[s, a] = empilhar s' **então início**
Empilha a e em seguida s' no topo da pilha;
Avança *ip* para o próximo símbolo de entrada;
Fim
Senão se ação[s, a] = reduzir A→β **então início**
Desempilha 2*|β| símbolos para fora da pilha;
Seja s' o estado agora ao topo da pilha;
Empilha A e em seguida desvio[s', A];
Escrever a produção A→β
Fim
Senão se ação[s,a] = aceitar **então**
Retornar
Senão erro()
Fim

Exemplo: Sejam as produções
(1) E → E + T
(2) E → T
(3) T → T*F
(4) T → F
(5) F → (E)
(6) F →**id**

Estado	Ação						Desvio		
	id	+	*	(	)	$	E	T	F
0	s5			s4			1	2	3
1		s6				acc			
2		r2	s7		R2	r2			
3		r4	r4		R4	r4			
4	s5			s4			8	2	3
5		r6	r6		R6	r6			
6	s5			s4				9	3
7	s5			s4					10
8		s6			S11				
9		r1	s7		R1	r1			
10		r3	r3		R3	r3			
11		r5	r5		R5	r5			

Para a entrada **id * id + id**, mostre o comportamento do analisador sintático LR.

Gramática LR

Um analisador sintático não precisa ler toda a pilha para saber quando um handle aparece no topo da pilha.

Existe um autômato finito que através da leitura dos símbolos gramaticais na pilha, do topo para o fundo determinar qual é o handle.

A função desvio de uma tabela sintática LR representa tal autômato.

Construindo Tabelas Sintáticas SLR (LR Simples)

Um item LR(0), ou simplesmente item, para uma gramática G é uma produção de G com um ponto em alguma de suas posições no lado direito. Assim, A → XYZ produz 4 itens:

A → .XYZ
A → X.YZ
A → XY.Z
A → XYZ.

A produção A → ε gera somente um item, A → .

Um item pode ser representado por um par de inteiros. O primeiro fornece o número da produção e o segundo a posição do . .

A idéia central do método SLR é construir primeiro a partir da gramática um autômato finito determinístico que reconheça prefixos viáveis.

Uma coleção de conjuntos de itens LR(0), que chamamos uma coleção LR(0) canônica, providencia a base para a construção de analisadores sintáticos SLR.

Se G for uma gramática com símbolo de partida S, então G' é uma gramática aumentada para G com um novo símbolo de partida S', mais a produção S' → S. A aceitação ocorre se e somente se o analisador sintático estiver para reduzir de S'→S.

Operação de fechamento

Se *I* for um conjunto de itens para uma gramática G, então o fechamento(I) é o conjunto de itens construídos a partir de *I* por essas duas regras:

1. Inicialmente cada item de *I* é adicionado ao fechamento(I).
2. Se A→α . *B* β estiver em fechamento(I) e B→γ for uma produção, adicionar o item B→.γ a *I*, se não estiver lá.

Exemplo: Considere a gramática de expressões aumentada:

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

Função fechamento(*I*)
Início
J := *I*;
Repetir
Para cada item A → α . *B* β em *J* e cada produção B → γ de G tal que B → . γ não esteja em *J* **faça**
Incluir B → . γ a *J*
Até que não possa ser adicionados mais itens a *J*
Fim

A operação desvio

Desvio(I,X), onde I é um conjunto de itens e X um símbolo gramatical.

Se I for um conjunto de itens válidos para algum prefixo viável γ, então desvio(I,X) será o conjunto de itens válidos para o prefixo iável γX.

Exemplo: Se *I* for o conjunto de dois itens {[E'→E.], [E→E.+T]}, então desvio(I,+) consiste em

E → E + . T
T → .T * F
T → .F
F → .(E)
F → . **id**

A construção dos conjuntos de itens

Procedimento itens(*G'*);
Início
C := { fechamento ({[S'→S]}) };
Repetir
Para cada conjunto de itens *I* em *C* e cada símbolo gramatical X tal que desvio(I,X) não seja vazio e esteja em *C* **faça**
Incluir desvio(I,X) a *C*
Até que não haja mais conjuntos de itens a serem incluídos a *C*
Fim

Tabelas Sintáticas SLR

Dada uma gramática G, a aumentamos de forma a produzir G'.
A partir de G' construímos *C*, a coleção canônica de conjuntos de itens para G'.
Construímos ação e desvio a partir de *C* usando o seguinte algoritmo:
Entrada: Uma gramática aumentada G'.
Saída: As funções sintáticas SLR ação e desvio para G'.
Método:

1. Construir *C* = { I_0, I_1, ..., I_n}, a coleção de conjuntos de itens LR(0) para G'.
2. O estado i é construído a partir de I_i. As ações sintáticas para o estado i são determinadas como se segue:
   a) Se [A → α.Bβ] estiver em I_i e desvio (I_i,a)=I_j, então estabelecer ação[i,a] em "empilhar j". Aqui a deve ser um terminal.
   b) Se [A → α.] estiver em I_i , então estabelecer a ação[i,a] em "reduzir através de A→α", para todo a em Seguinte(A). A não pode ser S'.
   c) Se [S'→ S.] estiver em I_i, então fazer ação[i,$] igual a "aceitar".

   Se quaisquer ações conflitantes forem geradas pelas regras anteriores, dizemos que a gramática não é SLR(1).
3. As transições de desvio para o estado i são construídas para todos os não-terminais A usando-se a seguinte regra: se desvio(I_i,A) = I_j, então desvio[i,A]=j.
4. Todas as entradas não definidas pelas regras (2) e (3) são tomadas como erro.
5. O estado inicial é aquele atribuído a partir do conjunto de itens contendo [S'→.S].

Exercício: Construa a tabela SLR para a gramática:

bexpr → bexpr **or** btermo | btermo

btermo → btermo **and** bfator | bfator

bfator → (bexpr) | **not** bfator

| **true** | **false**

Exercícios: Construa a tabela sintática do analisador SLR para as grmáticas a seguir:

S → '(' S ')' S | ε

E → E + n | n

Análise Sintática Bottom-up - LR(1) e LALR

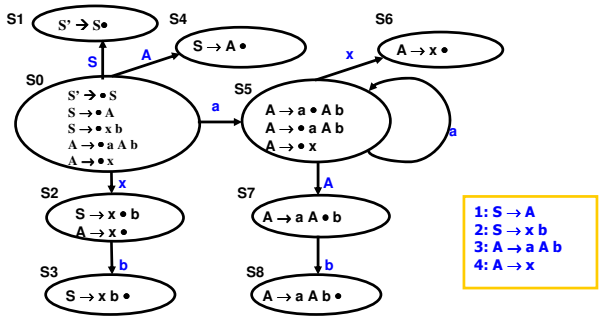
SLR(1) deixa muitos conflitos empilhar-reduzir sem resolver.

Problema: o conjunto Seguinte(N) é uma união de todos os contextos em que N pode ocorrer.

Exemplo

S → A | x b

A → a A b | x



Exercício:

Construa a tabela do analisador sintático SLR(1) (com os conflitos) para a seguinte gramática:

S → A | x b

A → a A b | x

Estado	Ação				Desvio	
	a	b	x	$	S	A
0	s5		s2		1	4
1				Ac.		
2		s3/r4		r4		
3		r2		r2		
4				r1		
5	s5		s6			7
6		r4		r4		
7		s8				
8		r3		r3		

Analisador sintático LR(1) - LR Canônico

Utiliza itens LR(1) que são uma extensão dos itens LR(0).

Eles incluem uma marca única de verificação a frente em cada item.

Mantém o conjunto seguinte por item

Item LR(1): $N \rightarrow \alpha \bullet \beta \ \{\sigma\}$

O estado inicial é construído com o fecho do item da nova produção S´→ S, [S´→S,$], onde $ representa o marcador de final. Isso indica que começamos por reconhecer uma cadeia derivável de S, seguida do símbolo $.

Fecho para conjuntos de itens LR(1) :

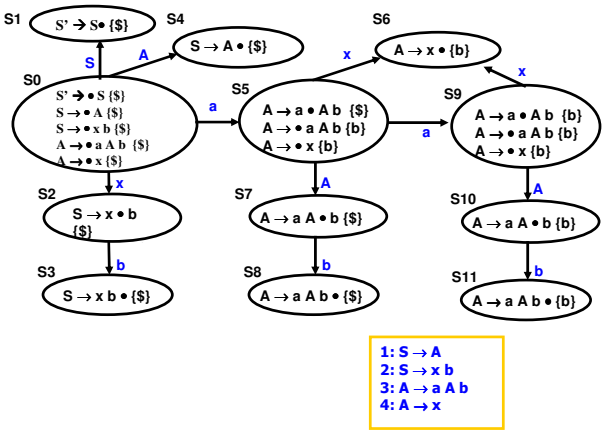
se conj S contém um item $P \rightarrow \alpha \bullet N \beta \ \{\sigma\}$

então

para cada regra de produção $N \rightarrow \gamma$

S deve conter o item $N \rightarrow \bullet \gamma \{\tau\}$

onde τ = Primeiro($\beta \{\sigma\}$)

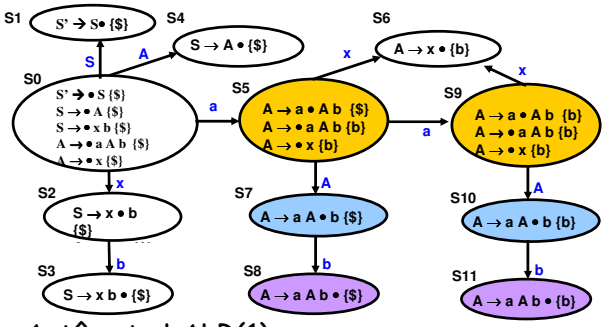


Analisador sintático LALR(1)

Tabelas LR Canônicas são bem grandes.

Opção: Combinar conjuntos de itens com núcleo "iguais" unindo os conjuntos de símbolos lookahead.

Autômato LR(1) – LR Canônico



Autômato LALR(1)

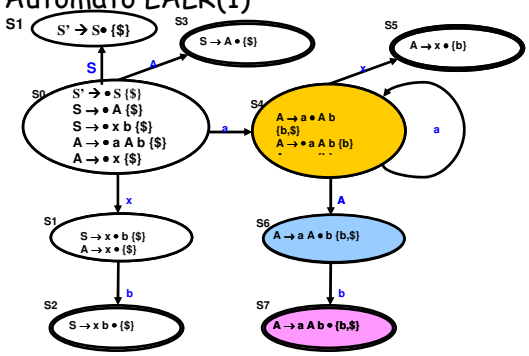


Tabela Ação/Desvio LALR(1)

Estado	Ação				Desvio	
	a	b	x	$	S	A
0	s5		s2		1	4
1				Ac.		
2		s3		r4		
3		r2		r2		
4				r1		
5	s5		s6			7
6		r4		r4		
7		s8				
8		r3		r3		

1: S → A
2: S → x b
3: A → a A b
4: A → x

Exercício:

Construa a tabela do analisador sintático SLR, LR(1) e LALR(1) para a gramática:

a) S → id | V := E
V → id
E → V | n

b) E → (L) | a
L → L , E | E

c) S → S (S) | ε

Árvore Sintática Abstrata

A árvore gramatical(árvore de análise) é uma estrutura de dados que mostra precisamente como os diversos segmentos do texto de pro-gramas serão vistos em termos da gramática.

A forma da árvore gramatical exigida pela gramática, normalmente não é a forma mais conveniente para processamento adicional.

Em geral usa-se uma forma modificada dessa árvore, chamada **árvore sintática abstrata ou AST (abstract sintaxe tree).**

Informações sobre a semântica podem ser adicionadas aos nós dessa árvore por meio de anotações em campos adicionais(atributos) nos nós.

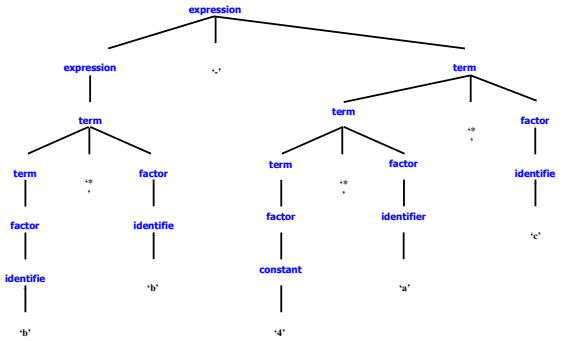
Exemplo de árvore gramatical para b*b-4*a*c, de acordo com as regras(fonte:Koen

Langendoen, "Projeto Moderno de Compiladores"):

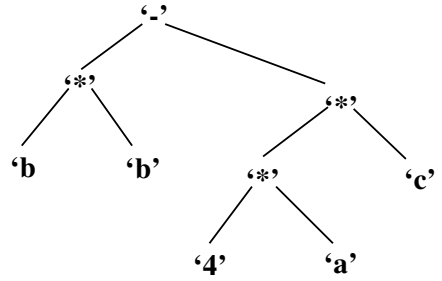
expression → expression '+' term | expression '-' term | term

term → term '*' factor | term '/' factor | factor

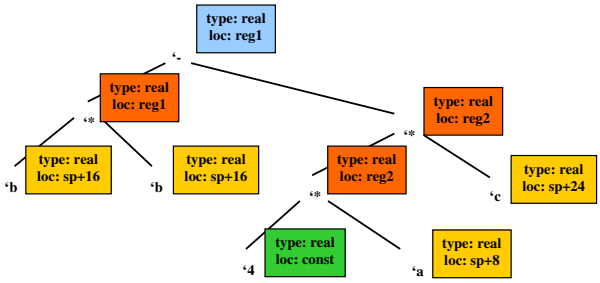
factor → identifier | constant | '(' expression ')'



Uma AST para o mesmo exemplo seria (fonte:Koen Langendoen, "Projeto Moderno de Compiladores") :



A mesma AST porém anotada (fonte:Koen Langendoen, "Projeto Moderno de Compiladores").



Exercícios:

Construa a árvore gramatical e a AST para:

a) regra: def_const → 'CONST' identificador '=' expressão ';'

a.1) CONST pi = 3,14159265;

a.2) CONST pi_quadrado = pi * pi;

b) instrução if do Pascal.