Análise Sintática Descendente

João Marcelo Uchôa de Alencar
joao.marcelo@ufc.br
UFC-Quixadá

Análise Sintática Descendente Recursiva

Análise Sintática LL(1)

Conjuntos Primeiros e de Sequência

Um Analisador Sintático para TINY

Recuperação de Erros

Análise Sintática Descendente

- Analisa a cadeia de marcas através da **derivação à esquerda**;
- a árvore de análise é percorrida em **pré-ordem**;
- **analisador com retrocesso**: testa diferentes possibilidades;
- **analisador preditivo**: baseado na leitura de uma ou mais marcas;
  - descendentes recursivos;
  - LL(1).
- vamos estudar os preditivos.

O Método Descendente Recursivo Básico

Metodologia

1. A regra gramatical de um não terminal é usada para definir um procedimento;
2. sequências de terminais correspondem a casamento da entrada;
3. sequências de não terminais correspodem a ativações de outros procedimentos;
4. a alternância corresponde a declarações de *if* ou *case*.

O Método Descendente Recursivo Básico

$exp \rightarrow exp\ soma\ termo | termo$
$soma \rightarrow + | -$
$termo \rightarrow termo\ mult\ fator | fator$
$mult \rightarrow *$
$fator \rightarrow (exp) | \textbf{número}$

```
procedure fator;
begin
    case marca of
    ( : casamento(();
        exp;
        casamento());
    numero:
        casamento(numero);
    else: erro;
    end case;
end fator;
```

O Método Descendente Recurso Básico

```
procedure casamento
          (marcaEsperada);
begin
   if marca = marcaEsperada then
      capturaMarca;
   else
      erro;
   end if;
end casamento;
```

- Por enquanto, não definimos *erro*;
- assumimos *exp* e *termo* definidos;
- como esses não terminais não tem terminais na sua definição, não é tão fácil construir seus procedimentos;
- vamos usar a EBNF.

Repetição e Escolha: o Uso da EBNF

if-decl → **if** (*exp*) *declaração*
| **if** (*exp*) *declaração* **else** *declaração*

```
procedure declIf;
begin
   casamento(if);
   casamento(();
   exp;
   casamento());
   declaracao;
   if marca = else then
      casamento(else);
      declaracao;
   end if;
end declIf;
```

if-decl → **if** (*exp*) *declaracao* [
else *declaracao*]

- Ambas opções começa com **if**;
- por isso, usamos a EBNF;
- os colchetes são traduzidos em testes;
- a gramática é ambígua, mas o procedimento trata esse problema.

Repetição e Escolha: o Uso da EBNF

$$exp \rightarrow exp\ soma\ termo \mid termo$$

- Usando a metodologia, *exp* invocaria *exp*!
- Um teste para escolher entre *exp* e *termo* é problemático;
- ambos pode resultar em (ou **número**.

$$exp \rightarrow termo\ \{\ soma\ termo\ \}$$

```
procedure exp;
begin
   termo;
   while marcar = + or marca = - do
      casamento(marca);
      termo;
   end while;
end exp;
```

Repetição e Escolha: o Uso da EBNF

$$termo \rightarrow fator \ \{ \ mult \ fator \ \}$$

```
procedure termo;
begin
   fator;
   while marca = * do
      casamento(marca);
      fator;
   end while;
end termo;
```

Gramática de Aritmética de Inteiros

```
function exp:inteiro;
var temp:inteiro;
begin
   temp := termo;
   while marca = + or marca = - do
      case marca of
      + : casamento(+);
          temp := temp + termo;
      - : casamento(-);
          tempo := temp - termo;
      end case;
   end while;
end exp;
```

Gramática de Aritmética de Inteiros

```c
#include <stdio.h>
#include <stdlib.h>

char token; /* variável de marca global */
/* protótipos de funções para ativações recursivas */
int exp(void);
int term(void);
int fator(void);
void error(void) {
    fprintf(stderr, "Erro\n");
    exit(1);
}
void match(char expectedToken) {
    if (token == expectedToken) token = getchar();
    else error();
}
```

Gramática de Aritmética de Inteiros

```
int main(int argc, char *argv[]) {
  int result;
  token = getchar(); /* carrega a marca com o primeiro
                        caractere para verificação
                        à frente */

  result = exp();
  if (token == '\n') /* teste final de linha */
     printf("Resultado = %d\n", result);
  else error(); /* caracteres indevidos na linha */
  return 0;
}
```

Gramática de Aritmética de Inteiros

```
int exp(void) {
  int temp = term();
  while ((token == '+') || (token == '-'))
     switch(token) {
        case '+' : match('+');
                   temp += term();
                   break;
        case '-' : match('-');
                   temp -= term();
                   break;
     }
  return temp;
}
```

Gramática de Aritmética de Inteiros

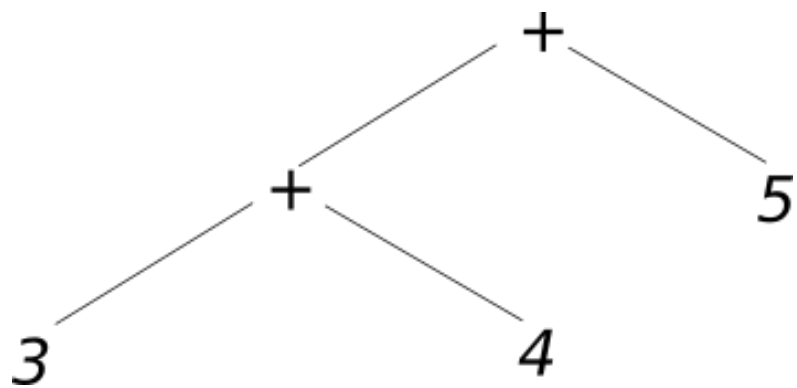
```
int term(void) {
  int temp = factor();
  while (token == '*') {
     match('*');
     temp *= factor();
  }
  return temp;
}
```

Gramática de Aritmética de Inteiros

```
int factor(void) {
 int temp;
 if (token == '(') {
    match('(');
    temp = exp();
    match(')');
 } else if (isdigit(token)) {
    ungetc(token, stdin);
    scanf("%d", &temp);
    token = getchar();
 }
 else error();
 return temp;
}
```

E as Árvores?

- A metodologia adotada mostra como reconhecer uma cadeia da gramática, mas não trata da definição das árvores;
- a associatividade foi mantida porque os cálculos foram feitos durante o reconhecimento;
- em um compilador, precisamos construir a árvore para garantir a associatividade, já que a execução ocorre em outro momento.



Para a expressão $3 + 4 + 5$, o nó que representa $3 + 4$ deve ser criado antes do nó que representa a adição de 5.

Construção de Árvores Sintáticas

```
function exp: arvoreSintatica;
var temp, novatemp, arvoreSintatica;
begin
   temp := termo;
   while marca = + or marca = - do
      case marca of
         + : casamento(+);
             novatemp := criaNoOp(+);
             filhoEsq(novatemp) := temp;
             filhoDir(novatemp) := termo;
             temp := novatemp;
         - : casamento(-);
             novatempo := criaNoOp(-);
             filhoEsq(novatemp) := temp;
             filhoDir(novatemp) := termo;
             temp := novatemp;
      end case;
   end while;
end exp;
```

Construção de Árvores Sintáticas - If Then Else

```
function declaracaoIf: arvoreSintatica;
var temp : arvoreSintatica;
begin
   casamento(if);
   casamento(();
   temp := criaNoDecl(if);
   testeFilho(temp) := exp;
   casamento());
   thenFilho(temp) := declaracao;
   if marca = else then
      casamento(else);
      elseFilho(temp) := declaracao;
   else
      elseFilho(temp) := nil;
   end if;
end declaracaoIf;
```

Limitações

- Nem sempre é fácil converte de BNF para EBNF;
- na regra $A \rightarrow \alpha|\beta$, se α e β iniciarem com os mesmos não terminais, como decidir?
- **conjuntos primeiros** α e β: conjuntos de marcas que podem iniciar legamente cada cadeia de caracteres;
- **conjunto sequência** de A: quais marcas podem suceder legalmente o não terminal A.

Análise Sintática LL(1)

Vamos evitar a recursão com o uso de uma **pilha**!

$$S \to (S)S \mid \varepsilon$$

Processando a cadeia *()*:

	Pilha	Entrada	Ação
1	$\$S$	$()\$$	$S \to (S)S$
2	$\$S)S($	$()\$$	casamento
3	$\$S)S$	$)\$$	$S \to \varepsilon$
4	$\$S)$	$)\$$	casamento
5	$\$S$	$\$$	$S \to \varepsilon$
6	$\$$	$\$$	aceita

Aceitação garantida se a pilha e a entrada ficarem vazias.

Análise Sintática LL(1)

Um analisador sintático descendente substitui um não terminal no topo da pilha por uma de suas escolhas na regra gramatical (em BNF).

Ações:

1. **gera**: substituir um não terminal A no topo da pilha por uma cadeia α com base na escolha da regra $A \rightarrow \alpha$;
2. **casamento**: casar uma marca no topo da pilha com a marca de entrada seguinte.

- As inserções na pilha equivalem aos passos de uma derivação;
- no lugar as ações, podemos ter procedimentos que constroem a árvore sintática.

A Tabela e o Algoritmo LL(1)

- Com um não terminal A no topo da pilha, precisamos escolher qual regra de A para colocar na pilha com base na marca corrente da entrada;
- se marca na pilha for igual a marca da entrada, temos o *casamento*, se forem diferentes, erro;
- vamos listar as escolhas em uma tabela chamada $M[N, T]$:
  - N são os não terminais;
  - T são os terminais mais o símbolo \$.

A Tabela e o Algoritmo LL(1)

Regras de Construção da Tabela

1. Se $A \to \alpha$ for uma escolha de produção, e houver derivação $\alpha \Rightarrow^* a\beta$, na qual a é uma marca, acrescente $A \to \alpha$ à célula da tabela $M[A, a]$;
2. se $A \to \alpha$ for uma escolha de produção, e houver derivações $\alpha \Rightarrow^* \varepsilon$ e $S\$ \Rightarrow^* \beta A a \gamma$, nas quais S é símbolo de começo e a é uma marca, acrescente $A \to \alpha$ à célula $M[A, a]$.

Motivação das Regras

1. Dada uma marca a na entrada, queremos selecionar uma regra $A \to \alpha$, se α puder produzir a para casamento;
2. se A derivar a cadeia vazia (via $A \to \alpha$), e se a for uma marca que possa suceder legalmente A em uma derivação, então selecionamos $A \to \alpha$ para se livrarmos do A.

A Tabela e o Algoritmo LL(1)

$$S \to (S)S \mid \varepsilon$$

- Pela regra 1, $M[S, (]$ aponta para $S \to (S)S$;
- Pela regra 2, considerando $S \Rightarrow (S)S$, $\alpha = \varepsilon$, $\beta = (, A = S$, $a =)$, e $\gamma = S\$$, podemos incluir $S \to \varepsilon$ em $M[S,)]$;
- como $S\$ \Rightarrow^* S\$$, podemos adicionar $S \to \varepsilon$ à $M[S, \$]$.

M[N,T]	(	)	$
S	$S \to (S)S$	$S \to \varepsilon$	$S \to \varepsilon$

A Tabela e o Algoritmo LL(1)

Uma gramática é uma **gramática LL(1)** se a tabela de análise sintática LL(1) associada tiver no máximo uma produção em cada célula.

```
(* assume que $ marca o fim da pilha e da entrada *)
coloca o símbolo de começo no topo da pilha;
while topo da pilha != $ and próxima marca for != $ do
   if topo da pilha for o terminal a
      and próxima marca de entrada for a
   then (* casamento *)
      retira da pilha;
      avança entrada;
   else if topo da pilha for um não terminal A
        and próxima marca de entrada for terminal a
        and célula da tabela M[A,a] contiver a produção
            A -> X1X2...Xn
        then (*gera*)
           retira da pilha;
           for i := n downto 1 do
              coloca Xi na pilha;
        else erro;
if topo da pilha for igual a $
   and marca seguinte na entrada for igual a $
then aceita
else erro;
```

A Tabela e o Algoritmo LL(1)

declaração $\rightarrow$ *if-decl* | **outra**
if-decl $\rightarrow$ **if** **(** *exp* **)** *declaração else-parte*
else-parte $\rightarrow$ **else** *declaração* $|\varepsilon$
exp $\rightarrow$ **0** | **1**

Questões

- Como ficaria a tabela?
- Como seria a execução para **if** (0) **if** (1) outra **else** outra?

A Tabela e o Algoritmo LL(1)

$M[N, T]$	**if**	outra	**else**	0	1	$
declaração	*declaração* → *if-decl*	*declaração* → *outra*				
if-decl	*if-decl* → **if** (**exp**) *declaração* *else-parte*					
else-parte			*else-parte* → **else** *declaração* *else-parte* → ε			*else-parte* → ε
exp				*exp* → 0	*exp* → 1	

A entrada M[*else-parte*, **else**] contém duas opções. A gramática é ambígua e não é LL (1).
Podemos manualmente alterar o algoritmo para escolher sempre a primeira produção.

Remoção de Recursão e Fatoração à Esquerda

- Não podemos resolver os problemas da repetição e escolha na LL(1) transformando a gramática para a notação EBNF;
- **remoção da recursão à esquerda**;
- **fatoração à esquerda**;
- não há garantias de transformação em LL(1), mas funciona na maioria dos casos para gramáticas de linguagem de programação.

Remoção de Recursão à Esquerda

Caso 1: recursão imediata à esquerda simples

$A \rightarrow A\ \alpha|\beta$ se transforma em:
$A \rightarrow \beta\ A^{'}$
$A^{'} \rightarrow \alpha A^{'}|\varepsilon$
Lembrando que α e β são cadeias de terminas ou não terminais e β não começa com A.

Exemplo

exp $\rightarrow$ *exp soma termo* | *termo*

Remoção de Recursão à Esquerda

Caso 2: recursão imediata à esquerda geral

$A \rightarrow A\alpha_1|A\alpha_2|...|A\alpha_n|\beta_1|\beta_2|...|\beta_m$, em que nenhum β começa com A, se transforma em:
$A \rightarrow \beta_1 A^{'}|\beta_2 A^{'}|...|\beta_m A^{'}$
$A^{'} \rightarrow \alpha_1 A^{'}|\alpha_2 A^{'}|...|\alpha_n A^{'}|\varepsilon$

Exemplo

exp $\rightarrow$ *exp* + *termo* | *exp* - *termo* | *termo*

Remoção de Recursão à Esquerda

Caso 3: recursão à esquerda geral

for i:= 1 **to** m **do**
    for j:=1 **to** $i-1$ **do**
        substituir cada escolha de regra gramatical da forma $A_i \rightarrow A_j\beta$ pela regra $A_i \rightarrow \alpha_1\beta|\alpha_2\beta|...|\alpha_k\beta$, onde $A_i \rightarrow \alpha_1|\alpha_2|\alpha_k$ é a regra corrente para A_j;
    end for
    remover a recursão imediata à esquerda de A_i;
end for

Exemplo

$A \rightarrow Ba|Aa|c$
$B \rightarrow Bb|Ab|d$

Remoção de Recursão à Esquerda

$exp \rightarrow exp^{'}$
$exp^{'} \rightarrow soma\ termo\ exp^{'}|\varepsilon$
$soma \rightarrow +|-$
$termo \rightarrow fator\ termo^{'}$
$termo^{'} \rightarrow mult\ fator\ termo^{'}|\varepsilon$
$mult \rightarrow *$
$fator \rightarrow (exp)|\textbf{número}$

- A gramática agora é LL(1), somente BNF;
- o problema é que alteramos a associatividade à esquerda da subtração ou adição;
- considere árvore de análise de 3 - 4 - 5;
- por isso, apesar de úteis, gramáticas LL(1) não resolvem todos os problemas.

Fatoração à Esquerda

$A \rightarrow \alpha\beta|\alpha\gamma$
α é o mesmo prefixo, impossibilitando a escolha da produção. A solução é transformar em:
$A \rightarrow \alpha A'$
$A' \rightarrow \beta|\gamma$

Exemplos

1. *decl-sequência* $\rightarrow$ *decl*; *decl-sequência*|*decl*
   decl $\rightarrow$ *s*
2. *if-decl* $\rightarrow$ *if* (*exp*)*declaração*|*if* (*exp*)*declaração else declaração*
3. *exp* $\rightarrow$ *termo* + *exp*|*termo*
4. *declaração* $\rightarrow$ *atribuição-decl*|*ativação-decl*|**outra**
   atribuição-decl $\rightarrow$ **identificador** := *exp*
   ativação-decl $\rightarrow$ **identificador**(*exp-list*)

Conjuntos Primeiros e de Sequência

- Até agora, a gramática sendo LL(1) e tendo a tabela, existe um algoritmo que determina a sequência de ações para o reconhecimento de cadeias;
- podemos definir a árvore sintática através de ações de criação de nós de acordo com as expansões na pilha do algoritmo LL(1);
- entretanto, não mostramos um algoritmo para construção da tabela $M[N, T]$, necessário para total automatização do processo;
- para tal, precisamos definir os **conjuntos primeiros e de sequência**.

Conjuntos Primeiros - Definição

Seja X um símbolo gramatical (terminal ou não terminal) ou ε. O conjunto **Primeiro(X)** é composto por terminais, e possivelmente ε, e definido da seguinte maneira:

1. Se X for um terminal ou ε, então Primeiro(X) = {X};
2. Se X for um não terminal, então para cada escolha de produção $X \rightarrow X_1X_2...X_n$, Primeiro(X) contém Primeiro(X_1) - $\{\varepsilon\}$. Adicionalmente, se para algum $i < n$ todos os conjuntos Primeiro(X_1), ... , Primeiro(X_n) contiverem ε, então Primeiro(X) conterá Primeiro(X_{i+1}) - $\{\varepsilon\}$. Se todos os conjuntos Primeiro(X_1), ... , Primeiro(X_n) contiverem ε, então Primeiro (X) também conterá ε.

Definimos **Primeiro(α)** para um cadeia qualquer $\alpha = X_1X_2...X_n$ da seguinte forma: Primeiro(α) contém Primeiro(X_1) - $\{\varepsilon\}$. Para cada $i = 2, ..., n$, se Primeiro(X_k) contiver ε para todo $k = 1, ..., i-1$, então Primeiro(α) conterá Primeiro(X_i) - $\{\varepsilon\}$. Finalmente, se para todo $i = 1, ..., n$, Primeiro (X_i) contiver ε, então Primeiro(α) conterá ε.

Conjuntos Primeiros - Algoritmo

```
for cada não terminal A do Primeiro(A) := {};
while houver alterações em algum Primeiro(A) do
   for cada escolha de produção A → X1X2...Xn do
      k := 1;
      continuar := true;
      while continuar = true and k <= n do
         acrescente Primeiro(Xk) - {ε} a Primeiro (A);
         if ε não pertencer a Primeiro (Xk) then
            continuar := false;
         k := k + 1;
      if continuar := true then
         acrescente ε a Primeiro(A);
```

Conjuntos Primeiros - Algoritmo

Podemos simplificar o algoritmo, caso não existam ε-produções:

```
for cada não terminal A do Primeiro := {};
while houver alterações em algum Primeiro(A) do
    for cada escolha de produção A → X1X2...Xn do
        acrescente Primeiro(X1) a Primeiro(A);
```

Definições

- Um não terminal A é **anulável** se houver uma derivação $A \Rightarrow^* \varepsilon$;
- um não terminal é anulável se e somente se Primeiro(A) contiver ε.

Conjuntos Primeiros - Exemplos

1. $exp \rightarrow exp\ soma\ termo | termo$
   $soma \rightarrow + | -$
   $termo \rightarrow termo\ mult\ fator | fator$
   $mult \rightarrow *$
   $fator \rightarrow (exp) | \textbf{número}$
2. $declaração \rightarrow if\text{-}decl \mid \textbf{outra}$
   $if\text{-}decl \rightarrow \textbf{if (}\ exp\ \textbf{)}\ declaração\ else\text{-}parte$
   $else\text{-}parte \rightarrow \textbf{else}\ declaração \mid \varepsilon$
   $exp \rightarrow \textbf{0} \mid \textbf{1}$
3. $decl\text{-}sequência \rightarrow decl\ decl\text{-}seq'$
   $decl\text{-}seq' \rightarrow \textbf{;}decl\text{-}sequência | \varepsilon$
   $decl \rightarrow \textbf{s}$

Conjuntos de Sequência - Definição

Dado um não terminal A, o conjunto **Sequência(A)**, composto por terminais e possivelmente \$, é definido como segue:

1. Se A for o símbolo inicial, \$ pertence a Sequência (A);
2. se houver uma produção $B \rightarrow \alpha A \gamma$, então Primeiro($\gamma$) - $\{\varepsilon\}$ pertence a Sequência (A);
3. se houver uma produção $B \rightarrow \alpha A \gamma$ tal que ε pertença a Primeiro(γ), então Sequência(A) contém Sequência(B).

Conjuntos de Sequências - Algoritmo

```
Sequência(símbolo-inicial) := {$};
for cada não terminal A != símbolo inicial do
   Sequência(A) := {};
while houver alterações em algum conjunto de Sequência do
   for cada produção A → X1X2...Xn do
      for cada Xi que for não terminal do
         adicione Primeiro(Xi+1Xi+2...Xn) a Sequência (Xi)
         (* Nota: se i=n, então Xi+1Xi+2...Xn = ε *)
         if ε estiver em Primeiro(Xi+1Xi+2...Xn) then
            adicione Sequência(A) a Sequência(Xi)
```

Conjuntos Sequências - Exemplos

1. $exp \rightarrow exp\ soma\ termo | termo$
   $soma \rightarrow + | -$
   $termo \rightarrow termo\ mult\ fator | fator$
   $mult \rightarrow *$
   $fator \rightarrow (exp) | \textbf{número}$
2. $declaração \rightarrow if\text{-}decl \mid \textbf{outra}$
   $if\text{-}decl \rightarrow \textbf{if (}\ exp\ \textbf{)}\ declaração\ else\text{-}parte$
   $else\text{-}parte \rightarrow \textbf{else}\ declaração \mid \varepsilon$
   $exp \rightarrow \textbf{0} \mid \textbf{1}$
3. $decl\text{-}sequência \rightarrow decl\ decl\text{-}seq'$
   $decl\text{-}seq' \rightarrow \textbf{;}decl\text{-}sequência | \varepsilon$
   $decl \rightarrow \textbf{s}$

Construção da Tabela de Análise Sintática LL(1) M[N,T]

Repita os dois passoas a seguir para cada não terminal A e escolha de produção $A \rightarrow \alpha$:

1. Para cada marca em Primeiro(α), adicione $A \rightarrow \alpha$ a $M[A, a]$;
2. se ε pertencer a Primeiro(α), para cada elemento a de Sequência(A) (uma marca ou \$), adicione $A \rightarrow \alpha$ a $M[A, a]$.

Teorema

Uma gramática BNF é LL(1) se as seguintes condições forem satisfeitas:

1. Para cada produção $A \rightarrow \alpha_1|\alpha_2|...|\alpha_n$, Primeiro($\alpha_i$) $\cap$ Primeiro(α_j) está vazio para cada i e j, $1 \leq i, j \leq n, i \neq j$;
2. para cada não terminal A tal que Primeiro(A) contenha ε, Primeiro(A) $\cup$ Sequência(A) está vazio.

Construção da Tabela - Exemplo

decl-sequência $\rightarrow$ *decl* *decl-seq*$'$
decl-seq$'$ $\rightarrow$ **;***decl-sequência*$|\varepsilon$
decl $\rightarrow$ **s**

Vamos encarar o **desafio**?

1. Construir os Conjuntos Primeiro;
2. construir os Conjuntos Sequência;
3. construir a tabela.

Analisadores Sintáticos LL(k)

- Podemos estender os algoritmos para k símbolos de verificação à frente:
  - Primeiro$_k(\alpha) = \{w_k | a \Rightarrow^* w\}$, onde w é uma cadeia de marcas e w_k são as k primeiras marcas de w;
  - Sequência$_k = \{w_k | S\$ \Rightarrow^* aAw\}$
- o problema é que a tabela vai ficar enorme;
- e não ajuda muito, pois se uma gramática não for LL(1), na prática ela não deve ser LL(k) para nenhum k.

Um Analisador Sintático para a Linguagem TINY

programa $\rightarrow$ *decl-sequência*
decl-sequência $\rightarrow$ *declaração*$\{$; *declaração*$\}$
declaração $\rightarrow$ *if-decl*$|$*repeat-decl*$|$*atribuição-decl*$|$*read-decl*$|$*write-decl*
if-decl $\rightarrow$ **if** *exp* **then** *decl-sequência* [**else** *decl-sequência*] **end**
repeat-decl $\rightarrow$ **repeat** *decl-sequência* **until** *exp*
atribuição-decl $\rightarrow$ **identificador** $:=$ *exp*
read-decl $\rightarrow$ **read identificador**
write-decl $\rightarrow$ **write** *exp*
exp $\rightarrow$ *simples-exp*[*comparação-op simples-exp*]
comparação-op $\rightarrow <|=$
simples-exp $\rightarrow$ *termo*[*soma termo*]
soma $\rightarrow +|-$
termo $\rightarrow$ *fator*$\{$*mult fator*$\}$
mult $\rightarrow *|/$
fator $\rightarrow$ (exp)**número**$|$**identificador**

Um Analisador Sintático para TINY

parser.h

```
TreeNode * parse(void);
```

parser.c

- 11 procedimentos recursivos;
- não terminais de operadores são reconhecidos diretamente;
- variável **token** que armazena a marca à frente;
- procedimento *match* (casamento);

Um Analisador Sintático para TINY

util.c e util.c

- **newStmtNode**: cria um nó da árvore de acordo com o tipo da declaração;
- **newExpNode**: cria um nó da árvore de acordo com o tipo de expressão;
- **copyString**: faz uma cópia de uma *string* em novo endereço.

O procedimento **printTree** apresenta uma representação da árvore sintática.

Recuperação de Erros em Analisadores Sintáticos Descendentes

Comportamento Mínimo

Reconhecedor: determinar se um programa está ou não sintaticamente correto.

Comportamento Esperado

Exibir uma mensagem significativa de erro, pelo menos do primeiro erro encontrado.

Comportamento Idealizado

Apresentar alguma forma de correção de erros, inferindo o programa correto com base no programa incorreto.

Recuperação de Erros em Analisadores Sintáticos Descendentes

A maioria das técnicas é *ad hoc*, sem estrutura ou algoritmos pré-definidos, feita especialmente para uma determinada linguagem. Considerações importantes:

- Tentar determinar um erro o mais rápido possível;
- tentar analisar o máximo possível de código após a descoberta de um erro, visando o identificar o máximo possível;
- evitar a **cascata de erros**;
- evitar laços infinitos.

São objetivos **conflitantes**.

Recuperação de Erros em Analisadores Descendentes Recursivos

Modo Pânico

O manipulador de erros do analisador sintático consome um número grande de marcas, avançando a entrada o máximo possível, até encontrar um ponto adequado para finalizar a análise.

Recuperação de Erros em Analisadores Descendentes Recursivos

Marcas de Sincronização

A cada procedimento recursivo para um não terminal, passamos como parâmetro um conjunto de marcas de sincronização. Ao encontrar um erro, o modo pânico consome marcas até que uma marca de sincronização apareça.

- Conjuntos de sequência;
- conjuntos primeiros.

Recuperação de Erros em Analisadores Descendentes Recursivos

```
procedure varrepara( conjsincr );
   begin
      while not ( marca in conjsincr ∪ $ ) do
         capturaMarca;
   end varrepara;

procedure verificaentrada (conjprimeiro, conjsequencia);
   begin
     if not (marca in conjprimeiro) then
        error;
        varrepara (conjprimeiro ∪ conjsequencia );
        enf if;
   end;
```

Recuperação de Erros em Analisadores Descendentes Recursivos

```
procedure exp( conjsincr );
begin
   verificaentrada({(, numero}, conjsincr);
   if not (marca in conjsincr) then
      termo(conjsincr);
      while marca = + or marca = - do
         casamento(marca);
         termo(conjsincr);
      end while;
      verificaentrada(conjsincr, {(, numero});
    end if;
end exp;
```

Recuperação de Erros em Analisadores Descendentes Recursivos

```
procedure fator (conjsincr);
begin
   verificaentrada({(, numero}, conjsincr);
   if not (marca in conjsincr) then
      case marca of
      ( : casamento(();
          exp({)});
          casamento());
      numero:
          casamento(numero);
      else error;
      end case;
      verificaentrada(conjsincr, {(, numero});
   end if;
end fator;
```

Recuperação de Erros em Analisadores Descendentes Recursivos

- O conjunto Primeiro indica o início de uma produção. Os conjuntos Sequências indicam o final;
- então a técnica consiste em começar uma produção já verificando se algum dos elementos do conjunto Primeiro é antigível;
- se não for, entre em pânico, e consuma até outro elemento do conjunto Primeiro (uma nova produção) ou até um elemento do conjunto Sequências (o fim da produção defeituosa);
- caso nenhum seja encontrado, finalize o processo.

Recuperação de Erros em Analisadores Sintáticos LL(1)

- Podemos implementar o modo pânico utilizando uma nova pilha para o conjunto de sincronização;
- além do *casamento* e *gera*, uma nova ação, *verificaentrada* deve ser ativada antes de cada geração;
- o erro ocorre quando temos um não terminal A no topo da pilha e a marca na entrada está ausente em Primeiro(A) (ou Sequência(A), se ε pertencer a Primeiro(A)).

Recuperação de Erros em Analisadores Sintáticos LL(1)

Construção da Tabela com Marcas de Sincronização

Dado um não terminal A no topo da pilha e uma marca na entrada ausente de Primeiro (A) (ou Sequência(A), se ε pertencer a Primeiro(A)), há três alternativas:

1. Retirar A da pilha;
2. retirar marcas até encontrar uma que possa reiniciar a análise;
3. colocar um novo não terminal na pilha.

Recuperação de Erros no Analisador Sintático TINY

- Modo pânico, mas sem os conjuntos de sincronização;
- o procedimento **match** simplesmente declara o erro;
- os procedimentos **statement** e **factor** declaram um erro quando nenhuma escolha correta é encontrada;
- o procedimento **parse** declara um erro se uma marca diferente do final do arquivo for encontrada após o encerramento da análise sintática.

FIM

Dúvidas?