Compiladores

Análise Sintática (Parte 3)

Introdução

Após a introdução à análise sintática e a apresentação das estratégias de *parsers* ascendentes e descendentes, estas notas de aula seguem abordando a análise sintática com foco nas estratégias de recuperação de erros adotadas pelo compilador. Também está apresentada uma ferramenta geradora de analisadores sintáticos: o Bison.

Para ler os textos anteriores sobre análise sintática, basta clicar clicar nos links abaixo:

- Análise Sintática (Parte 1)
- Análise Sintática (Parte 2)

Erros e Detecção de Erros (baseado em Du Bois 2011)

Um programa pode apresentar erros léxicos, sintáticos, semânticos e de lógica de programação. A maioria dos erros detectados durante a compilação são erros sintáticos. Diz-se que ocorreu um erro sintático quando o *parser* falha na construção da árvore sintática.

Na análise sintática, o tratador de erros deve

- relatar os erros de forma clara;
- continuar mesmo após encontrar o primeiro erro, ou seja, a análise deve poder relatar também os próximos erros;
- atuar de forma a gerar o mínimo impacto possível, ou seja, não deve retardar significativamente o processo de programas corretos

Observação: algumas dessas características podem ser conflitantes!

Quando um erro é encontrado, o tratador de erros deve realizar duas ações: relatar o erro e continuar a análise. No entanto, oferecer mensagens de erro significativas não é uma tarefa simples. Por exemplo, qual é o erro do programa abaixo?

```
x = (a + b ( - foo(x - y);
```

Recuperação de Erros

Existem duas formas de fazer com que o *parser* se recupere de erros: com e sem correção.

Na estratégia **com correção**, o *parser* tenta corrigir o erro para que possa continuar com a análise. O problema é que em muitos casos é difícil descobrir qual erro aconteceu. Um erro mal corrigido pode acabar introduzindo outros erros sintáticos.

Exemplos de correções:

- a substituição de uma "," (vírgula) por ";” (ponto e vírgula);
- a exclusão ou a inserção de um ";" (ponto e vírgula).

A estratégia com correção é também chamada de *modo de declaração* (*statement mode*) e *recuperação em nível de frase* (*phrase level*).

Outra forma de implementar correção é elaborar gramáticas estendidas para as linguagens, prevendo também como válidos determinados erros comuns da linguagem. Esta estratégia é conhecida como *produção de erro*" e exige do projetista que ele tenha conhecimento sobre os erros que podem ocorrer.

Existe ainda, em teoria, um caso particular chamado de *recuperação de erro com correção global*. Adotando esta estratégia, o *parser* analisaria todo código e tentaria descobrir um código o mais próximo possível dele que não contenha erros. Na prática, este método não é implementado.

Na estratégia **sem correção**, a análise é feita até o ponto do erro e é abandonada; depois essa

análise começa novamente a partir do erro. Esse tipo de análise funciona melhor, mas é mais difícil de implementar, pois é necessário modificar a gramática em tempo de execução.

O ***modo pânico*** (***panic mode***) é uma estratégia de recuperação de erros na qual o *parser* remove/despreza entradas de dados, uma a uma, até que encontre um *token* sincronizante, que geralmente é um delimitador, mas também pode ser algum *token* representado por símbolos constantes no conjunto FOLLOW.

- Essa forma de recuperação de erros evita que o analisador caia em *loops* infinitos.
- É fácil de compreender e implementar.
- Traz como desvantagem o fato de que as entradas desprezadas podem apresentar outros erros, que acabam sendo ignorados.

Exemplo: `a = b + cd (6) * 30;`

Supondo que falte um operador entre `c` e `d`, após o símbolo do terminal para a variável `c` ter sido colocado na pilha, um erro é detectado quando a variável `d` é lida. Esse método de recuperação de erro descarta os *tokens* de origem até que um *token* que permita uma movimentação seja lido. Portanto, os tokens `d`, `(`, `6` e `)` são descartados e a análise é retomada com o operador de multiplicação que segue o caractere `)`, ou seja, parte para o caractere `*`.

PERGUNTA

Mas por que o símbolo `*`, se ele não é um delimitador?

Porque um *token* sincronizante também pode ser um símbolo constante no conjunto FOLLOW. Como se sabe, após um nome de variável o operador `*` é sintaticamente aceito, então este símbolo terminal estaria no conjunto FOLLOW de um identificador.

Um resumo dos tipos de erros e as estratégias possíveis pode ser visto na Tabela 1.

Tabela 1. Erros e possíveis estratégias de recuperação.

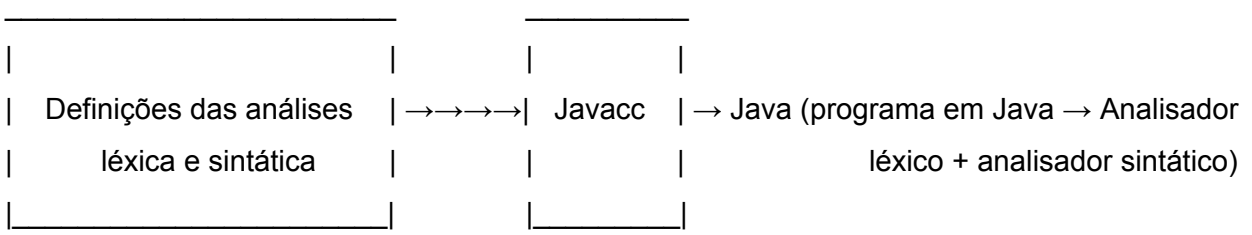
Tipo de Erro	**Estratégias Possíveis**
Léxico	• Pânico
Sintático	• Pânico • Declaração • Produção de Erros • Global
Semântico	• Tabela de Símbolos

Ferramentas Geradoras de Analisadores Sintáticos

Já citadas nas Notas de Aula sobre Análise Sintática (Parte 2), as principais ferramentas geradoras de *parsers* estão citadas e resumidas a seguir:

- `antlr`
  - *ANother Tool for Language Recognition*
  - Gerador de *parsers* que usa LL para análise.
- `pegen`
  - Gerador de *parser* usado na implementação de CPython.
  - Disponível em https://github.com/we-like-parsers/pegen.
- `happy`
  - Gerador de analisadores sintáticos LALR(1) em Haskell.
- `yacc`
  - *yet another compiler compiler.*
  - É um *parser* LALR (*look-ahead* LR).
  - Sua primeira versão surgiu na década de 1970.
  - Costuma ser usado em conjunto com o `lex`.
- `javacc`
  - Java *compiler compiler*.
  - Gerador automático de analisadores sintáticos.
  - Permite também a definição da análise léxica.

- Gera *parsers top-down* (descendentes) recursivos.
- Gera um *parser* implementado em Java.
- Possui sintaxe parecida com a do `lex` para definir as expressões regulares.
- Possui sintaxe parecida com a do *parser* descendente recursivo para definição do *parser*, isto é, cada não terminal vira uma função.
- O esquema abaixo ajuda a ilustrar o funcionamento do `javacc`:



- Vídeo demonstrando o uso do javacc.
- Arquivos usados no vídeo (arquivo ZIP, tamanho 654,0 kB).

- `bison`
  - Gerador de analisadores sintáticos utilizado em conjunto com o `flex`.
  - É o sucessor do `yacc`.
  - Gera analisadores *bottom-up* (ascendentes).
  - Bison gera, por padrão, *parsers* LALR(1), mas também pode gerar *parsers* LR, IELR(1) e GLR.
  - Assim como ocorre com o `flex`, as seções do arquivo `bison` são:

```
DEFINIÇÕES
%%
REGRAS
%%
CÓDIGOS
```

  - O arquivo `bison` é geralmente salvo com a extensão `.y`. Isso vem do `yacc`, já que o `bison` é compatível com esta ferramenta.
  - Gera uma função C chamada `yyparse()`, que recebe como entrada um fluxo de *tokens* (*string*) e retorna:
    - zero, se a entrada estiver em conformidade com gramática; ou
    - diferente de zero caso contrário.
  - Depois, chama `yylex()` para obter o próximo *token* (conforme já mostrado na

Figura 3 da Aula 3).

- Ele para quando `yylex()` retorna `0`.
- Definições úteis (DONNELY; STALLMAN, 2021): `%start`, `%token`, `%type`, `%left`, `%right` entre outras.
  - `%start`: Especifica o símbolo inicial da gramática.
  - `%token`: Declara um símbolo terminal, ou uma categoria de *token*.
  - `%type`: Usada para declarar símbolos não terminais.
  - `%left`: Declara um símbolo terminal que é associativo à esquerda.
  - `%right`: Declara um símbolo terminal que é associativo à direita.

Exemplo adaptado de Osório (s.d.): supõe-se a existência da gramática a seguir:
comando → PALAVRA = expr | expr
expr → expr + NUMERO | expr - NUMERO | NUMERO

Analisando a gramática, percebe-se que os *tokens* são palavras e números, e o símbolo inicial é `comando`. No arquivo `flex`, cria-se o analisador léxico (`tokens.l`) como segue:

```
%{
#include "y.tab.h"
extern int yylval;
int error_code;
%}

%%

[0-9]+          { yylval = atoi(yytext); return NUMERO; }
[a-zA-Z]+       { return PALAVRA; }
[ \t]           ; /* ignora */
\n              { return 0; } /* EOF */
.               { return yytext[0]; }

%%
```

Observação: `yytext` contém o texto correspondente ao *token* atual. Portanto, `yytext[0]` contém o primeiro caractere do texto correspondente ao *token* atual.

O arquivo `tokens.l` define um *scanner* que reconhece todos os números e todas as palavras, bem como ignora espaços e tabulações, e é encerrado quando uma quebra de linha é encontrada.

A inclusão de `y.tab.h` assume que o arquivo `bison` foi gerado na forma padrão, pois ele gerará o arquivo `y.tab.c`. O arquivo `bison` (parser.y) implementa a gramática como segue:

```
%{
#include <stdio.h>
int yylex();
int yyerror(char *s);
%}

%token PALAVRA NUMERO
%start comando

%%
comando: PALAVRA '=' expr        { $$ = $1; printf("(%d)\n", $3); }
      | expr                     { $$ = $1; printf("= %d\n", $1); }
      ;

expr: expr '+' NUMERO            { $$ = $1 + $3; }
      | expr '-' NUMERO          { $$ = $1 - $3; }
      | NUMERO                   { $$ = $1; }
      ;

%%

#include <stdio.h>
```

```
extern FILE *yyin;

int yyerror(s)
char *s;
{
      fprintf(stderr,"%s\n",s);
}

int main () {
      do { yyparse(); }
      while (!feof(yyin));
      return 0;
}
```

Observação: `yyparse()` faz a correspondência da *string* de entrada com as ações das produções usadas e as executa.

Explicações sobre o código do arquivo `parser.y`:

- O arquivo `parser.y` indica que os *tokens* são `PALAVRA` e `NUMERO`. Isso foi feito utilizando `%token` na seção de definições.
- Ainda na seção de definições, `%start` foi usado para indicar o símbolo inicial, que neste caso é `comando`, como pede a gramática.
- `$$` é usado para armazenar o resultado da regra.
- Cada símbolo *i* da direita é associado a uma variável `$i`. Por exemplo, o primeiro valor está em `$1`, o segundo está em `$2` e assim por diante.
- `$i` contém o valor do *token* (retornado por `yylex()`) se o símbolo *i* for terminal; caso contrário, contém o valor do `$$` do não terminal (RIBEIRO, [s.d.]).

O *scanner* é gerado com o mesmo comando já visto na aula sobre o `flex`:

```
flex tokens.l
```

O parser é gerado com o seguinte comando:

bison -y -d parser.y

O comando flex cria um arquivo `lex.yy.c`. O comando bison cria um arquivo `y.tab.c`. Eles podem ser compilados com o seguinte comando:

gcc y.tab.c lex.yy.c -lfl

A execução é feita com **./a.out**.

A partir da execução, pode-se começar a usar o *parser*. Exemplos de dados de entrada:

1 + 10

1 - 10

A = 10 + 1 - 9

8

A

A = -10

-10

A = (10 + 1 + 1)

Por que as quatro últimas entradas resultaram em erro de sintaxe?

Altere o código do *parser* para que as entradas abaixo sejam aceitas e as expressões sejam corretamente avaliadas:

A = -10

-99-1

-1-0

var = -1-1-1

Após a(s) modificação(ões), como é a gramática desta linguagem? Uma resposta possível:

comando → PALAVRA = expr | expr

expr → expr + NUMERO | expr - NUMERO | NUMERO | -NUMERO

Há outras soluções possíveis. O próprio símbolo "-"[0-9]+ poderia ser reconhecido como um número pelo analisador léxico. No entanto, as entradas "+"[0-9]+ também teriam que ser

consideradas válidas, concordam?

Outro exemplo utilizando `flex` e `bison`, baseado em Schnorr (2019), é o de uma calculadora simples. Os arquivos criados são `scanner-calculadora.l` (*scanner*) e `parser-calculadora.y` (*parser*).

Código do *scanner*:

```
%{
#define YYSTYPE double
#include "y.tab.h"
%}

ignorar          [ \t]+
digito           [0-9]
inteiro          {digito}+
real             {inteiro}("."{inteiro})?

%%

{ignorar}        { }
{real}           { yylval = atof(yytext); return NUMERO; }

"+"              return MAIS;
"-"              return MENOS;
"*"              return VEZES;
"/"              return DIVIDE;
"^"              return POTENCIA;
"("              return ESQ;
")"              return DIR;
"\n"             return FIM;
```

Código do *parser*:

```
%{
#include <math.h>
#include <stdio.h>
#define YYSTYPE double
int yyerror (char const *s);
extern int yylex (void);
%}

%token NUMERO
%token MAIS MENOS VEZES DIVIDE POTENCIA
%token ESQ DIR
%token FIM

%left MAIS MENOS
%left VEZES DIVIDE
%left NEG
%right POTENCIA

%define parse.error verbose

%start Inicio

%%

Inicio: ; /* linha vazia */
Inicio: Inicio Linha;

Linha: FIM
Linha: Expressao FIM { printf("Resultado: %f\n", $1); }

Expressao: NUMERO                       { $$=$1; };
Expressao: Expressao MAIS Expressao   {  $$  =$1  +  $3;  printf("%f  +
```

```
%f\n", $1, $3); };
Expressao: Expressao MENOS Expressao  { $$ = $1 - $3; printf("%f -
%f\n", $1, $3); };
Expressao: Expressao VEZES Expressao  { $$ = $1 * $3; printf("%f *
%f\n", $1, $3); };
Expressao: Expressao DIVIDE Expressao { $$ = $1 / $3; printf("%f /
%f\n", $1, $3); };
Expressao: MENOS Expressao %prec NEG  { $$ = -$2; printf("- %f\n",
$2); };
Expressao: Expressao POTENCIA Expressao    {   $$   =   pow($1,   $3);
printf("%f ^ %f\n", $1, $3); };
Expressao: ESQ Expressao DIR          { $$ = $2; };

%%

int yyerror(char const *s) {
      printf("%s\n", s);
}

int main() {
      int ret = yyparse();
      if (ret) {
            fprintf(stderr, "%d error found.\n",ret);
      }
      return 0;
}
```

Gerando o *scanner*: **`flex scanner-calculadora.l`**

Gerando o *parser*: **`bison -y -d parser-calculadora.y`**

Compilando os arquivos gerados: **`gcc lex.yy.c y.tab.c -lfl -lm`**

Executando: **`./a.out`**

Neste exemplo, é necessário ligar a biblioteca matemática (com `-lm`) na compilação devido ao uso da função `pow` para o cálculo da potência.

A partir da execução, pode-se começar a usar o *parser*. Exemplos de dados de entrada:

```
2*(1+(9/(4-2^0)))
2*(4/4)
2*(-1)
1 * /0
```

Outras formas de testar são sugeridas abaixo.

```
echo "2*(1+(9/(4-2^0)))" | ./a.out # Forma 1

echo "2*(1+(9/(4-2^0)))" > arquivo.txt # Forma 2
./a.out < arqin.txt # Forma 2
```

No uso do `bison`, a diretiva `-y` emula o `yacc`. No padrão do `yacc`, a saída é escrita em `y.tab.c` e no cabeçalho `y.tab.h`. Por esse motivo, os *scanners* vistos têm `#include "y.tab.h"` sempre.

É possível omitir a diretiva `-y`. Neste caso, quando o `bison` for executado passando como parâmetro um arquivo chamado, por exemplo, `arquivo.y`, os arquivos gerados serão, por padrão, `arquivo.tab.h` e `arquivo.tab.c`. No entanto, é necessário acrescentar corretamente o arquivo de cabeçalho no arquivo flex: `#include "arquivo.tab.h"`.

Outro exemplo interessante é o de um analisador sintático que identifica a definição de classes e funções em Python inspirado na solução de Gudulin (s.d.). Ele está disponível através dos arquivos `python-scanner.l`, representando a especificação do *scanner*, e `python-parser.y`, que representa a especificação do *parser*. Os códigos dos arquivos `python1.py`, `python2.py` e `python3.py` são exemplos de entradas que podem ser utilizados para teste.

Comandos necessários para a execução do código do exemplo:

```
bison -d -y python-parser.y && flex python-scanner.l
```

```
g++ lex.yy.c y.tab.c
./a.out < python1.py # Troque o arquivo se desejar testar outros códigos.
```

Conclusão

A análise sintática é a fase de análise responsável por detectar a maioria dos erros encontrados em um processo de tradução. As estratégias de recuperação de erros garantem que o processo de compilação continue mesmo após a detecção de um erro, permitindo a identificação de mais problemas, se houver. A escolha da estratégia de recuperação de erros, seja com ou sem correção, impacta a complexidade do próprio compilador, bem como na clareza e na utilidade das mensagens de erro geradas.

Ferramentas como `bison` e `yacc` tornam a tarefa de implementar analisadores sintáticos mais fácil, pois automatizam boa parte do processo e permitem aos desenvolvedores focar principalmente em especificar as regras gramaticais e as ações associadas. Os exemplos práticos, como a criação de uma calculadora ou a definição de classes e funções em Python, ilustram a flexibilidade e a aplicabilidade das ferramentas geradoras de *parsers*. Essas aplicações mostram como os conceitos teóricos se traduzem em soluções práticas, permitindo a análise e a compilação de códigos complexos de maneira simples, automatizada e precisa.

Exercícios

1) Altere o código de `parser-calculadora.y` para utilizar o operador ou (símbolo `|`) na gramática.

2) Ainda tomando como base a calculadora, altere *scanner* e *parser* para permitir a operação de raiz quadrada. O símbolo de raiz quadrada deve ser `V`.
   Exemplos de execução desejáveis após a alteração:

```
V 9
V 9.000000
Resultado: 3.000000
```

```
V (3^2)
3.000000 ^ 2.000000
V 9.000000
Resultado: 3.000000
V (2^3)+1
2.000000 ^ 3.000000
V 8.000000
2.828427 + 1.000000
Resultado: 3.828427
V (2^3+1)
2.000000 ^ 3.000000
8.000000 + 1.000000
V 9.000000
Resultado: 3.000000
```

3) (POSCOMP) QUESTÃO 68 – Durante a análise sintática, erros podem ser detectados na sintaxe do programa fonte. Nesse caso, alguns compiladores podem reportar o erro e interromper a análise. Outros reportam o erro, mas, também, realizam uma recuperação do erro e tentam continuar a fase de análise, entretanto, a fase de síntese é desativada. Nesse sentido, analise as assertivas abaixo:

   I. Um recuperador de erros para um analisador sintático deve informar a presença de erros de forma clara e recuperar-se de maneira que consiga continuar a fase de análise sem se preocupar com o custo de processamento para tal atividade.

   II. O modo pânico é uma forma de recuperação de erro na qual o analisador despreza símbolos da entrada até que um *token* sincronizante seja encontrado.

   III. Erros sintáticos incluem divergências de tipo entre operadores e operandos.

   IV. Na recuperação em nível de frase ou local, há a alteração sobre um símbolo que pode ser feita: pela substituição, inserção ou exclusão de *token* de entrada.

   Quais estão corretas?

   a) Apenas I.

   b) Apenas I e III.

   c) Apenas II e IV.

d) Apenas I, II e III.

e) I, II, III e IV.

Respostas

1) Resposta no arquivo `parser-com-guardas-calculadora.y`.
2) Resposta nos arquivos `scanner-calculadora-com-sqrt.l` e `parser-calculadora-com-sqrt.y`.
3) C

Referências

DONNELY, C.; STALLMAN, Richard. Gnu bison–the yacc-compatible parser generator. Free Software Foundation, Cambridge, 2021. Disponível em: http://www.gnu.org/software/bison/manual/html_node/Decl-Summary.html. Acesso em: 31 mai. 2024.

DU BOIS, André Rauber. Notas de Aula sobre Compiladores. 01 set. 2011.

GUDULIN, Alexander. Parsim Python code with Flex and Bison. Disponível em: https://sudonull.com/post/144796-Parsim-Python-code-with-Flex-and-Bison. Acesso em: 31 mai. 2024.

OSÓRIO, Fernando Santos. LEX & YACC / FLEX & BISON. Disponível em: http://osorio.wait4.org/oldsite/compil/lex-yacc-hands-on.pdf. Acesso em: 31 mai. 2024.

RIBEIRO, Sérgio F. s.d. Disponível em: https://docplayer.com.br/143628068-Compiladores-lex-e-yacc-flex-e-bison-ferramentas-flex-bison.html. Acesso em: 31 mai. 2024.

SCHNORR, Lucas M. Calculator. Disponível em: https://github.com/schnorr/calc. Acesso em: 01 jun. 2024.