Segundo Trabalho Compiladores

Luan Marques Batista
Otavio Augusto Teixeira
Instituto de Biociências,Letras e Ciências Exatas, Unesp -
Univ Estadual Paulista (São Paulo State University)
Rua Cristóvão Colombo 2265, Jd Nazareth, 15054-000, São José do Rio Preto - SP, Brazil
e-mail: luan.batista@unesp.br
e-mail: otavio.a.teixeira@unesp.br

November 1, 2024

Abstract

Neste relatório, abordaremos a construção de uma linguagem de programação que chamaremos de: KNOT. Também discutiremos o que é um Analisador Sintático, as ferramentas utilizadas e o que é uma análise sintática.

1 Introdução

Para a construção deste relatório, introduziremos alguns conceitos básicos para o entendimento do leitor.

1.1 Análise Sintática

A análise sintática, também conhecida como *parsing*, é uma das etapas fundamentais de um compilador, responsável por verificar a estrutura gramatical do código-fonte. Após a análise léxica, que converte o código em uma sequência de tokens, a análise sintática organiza esses tokens em uma estrutura hierárquica, geralmente representada por uma árvore de derivação ou árvore sintática. Essa árvore reflete a estrutura sintática do código de acordo com uma gramática formal previamente definida, chamada gramática livre de contexto. Nessa etapa, o compilador valida a conformidade do código com as regras sintáticas da linguagem, detectando erros como o uso incorreto de operadores ou uma estrutura de comandos que não corresponde ao esperado.

Além da validação, a análise sintática também prepara o terreno para as próximas fases do compilador, como a análise semântica e a geração de código. Uma vez que o código foi estruturado em uma árvore sintática, o compilador consegue identificar de forma organizada as relações entre as diversas partes do código, como expressões, variáveis e blocos de controle. Dessa forma, a análise sintática é essencial para garantir que o código seja interpretado corretamente pelo compilador, facilitando a geração de código eficiente e confiável na fase final da compilação.

1.2 Bison

O GNU Bison é uma ferramenta de geração de analisadores sintáticos que ajuda na construção de compiladores e interpretadores, convertendo uma gramática formal em código C ou C++. Ele recebe como entrada uma descrição de uma gramática livre de contexto, geralmente definida em formato BNF (Backus-Naur Form), e gera um código capaz de analisar e validar cadeias de entrada de acordo com essa gramática. O Bison automatiza a criação de um parser (analisador sintático), o que reduz o esforço de programadores ao criar a parte responsável pela análise sintática de uma linguagem de programação.

Essencialmente, o Bison lê a gramática definida pelo usuário e cria um programa que analisa expressões e estruturas de acordo com as regras especificadas, detectando erros e organizando a entrada em uma forma que possa ser processada em etapas subsequentes de um compilador ou interpretador. Ele é comumente utilizado em conjunto com o Flex, um gerador de analisador léxico, para fornecer uma solução completa de análise e compilação de código.

A seguir, mostraremos um exemplo de como ficam os tokens e como eles são usados pelo Bison:

```
# Aqui definimos os caracteres terminais
%token SE SENAO WHILE FOR NOT PRINTF
%%
# Parte onde definimos as regras
Programa_principal:
  MAIN ABRE_PARENTESES FECHA_PARENTESES ABRE_CHAVE Comandos FECHA_CHAVE
  | error { yyerror("Erro sintático: função main mal estruturada"); };
```

Dessa forma, o analisador será capaz de identificar um número e um identificador quando lidos.

2 KNOT

Knot é uma linguagem de programação baseada na linguagem C. Ela foi pensada e desenvolvida durante uma das aulas de Compiladores. A seguir, podemos visualizar a definição formal da linguagem.

2.1 Definição Formal

Para a realização de forma correta do parser, foi preciso alterar algumas regras da gramática feitas no projeto passado. Comentarei o que foi mudado e o que não mudou deixaremos como estava antes:

2.1.1 Alfabeto

O alfabeto Σ da linguagem não se alterou e continua:

- Letras: $a - z, A - Z$
- Dígitos: $0 - 9$
- Símbolos: +, −, ∗, /, =, <, >, !, &, |, ”, {, }, (,), [,], ,, ;

- Espaços em branco: \t, \r, \n, " " (espaço)
- Caracteres de escape: \

2.1.2 Tokens

Aqui foi preciso alterar alguns tokens para que o parser conseguisse entender.Em resumo, retiramos o comentário para facilitar o parser que já estava muito complexo. Separamos os Delimitadores e as palavras reservadas para que o parser conseguisse identificar os tokens de uma forma mais otimizada.
Dessa forma os tokens da linguagem ficaram:

- Número Inteiro: NUMBER → [0-9]+
- Identificador: ID → [A-Za-z][A-Za-z0-9]*
- Espaço: ESPACO → [\t\r" "]+
- Operadores Relacionais: OP_REL → "=="|"!="|"<"|">"|"<="|">="
- Operadores Aritméticos: OP_ARIT → +|-|*|/
- Operadores Lógicos: OP_LOG → "&&" | "||" | "!"
- Operador de Atribuição: ATRIBUICAO → "="
- String: STRING → \" ([^\\"]|\\.)* \"
- ABRE PARENTESES: ABRE_PARENTESES → "("
- FECHA PARENTESES: FECHA_PARENTESES → ")"
- ABRE CHAVE: ABRE_CHAVE → "{"
- FECHA CHAVE: FECHA_CHAVE → "}"
- ABRE_COLCHETES: ABRE_COLCHETES → "["
- FECHA_COLCHETES: FECHA_COLCHETES → "]"
- VIRGULA: VIRGULA → ","
- PONTO_E_VIRGULA: PONTO_E_VIRGULA → ";"
- FLOAT: FLOAT → "float"
- INT: INT → "float"
- BOOL: BOOL → "boolean"
- STRING_TIPO: STRING_TIPO → "string"
- SE: SE → "if"
- SENAO: SENAO → "else"

- MAIN: `MAIN` → `"main"`
- PRINTF: `PRINTF` → `"printf"`
- NOT: `NOT` → `"!"`
- VERDADEIRO: `VERDADEIRO` → `"true"`
- FALSO: `FALSO` → `"false"`
- FOR: `FOR` → `"for"`

2.1.3 Regras Léxicas

As regras léxicas não foram alteradas em sua execencia, apenas mudamos como elas retornas os tokens, ao invés de usar um print, agora retornamos o token direto para o parser:

- `"("` → `return (ABRE_PARENTESES);`
- `")"` → `return (FECHA_PARENTESES);`
- `"{"` → `return (ABRE_CHAVE);`
- `"}"` → `return (FECHA_CHAVE);`
- `"["` → `return (ABRE_COLCHETES);`
- `"]"` → `return (FECHA_COLCHETES);`
- `","` → `return (VIRGULA);`
- `";"` → `return (PONTO_E_VIRGULA);`
- `"float"` → `return (FLOAT);`
- `"int"` → `return (INT);`
- `"boolean"` → `return (BOOL);`
- `"string"` → `return (STRING_TIPO);`
- `"if"` → `return (SE);`
- `"else"` → `return (SENAO);`
- `"while"` → `return (WHILE);`
- `"for"` → `return (FOR);`
- `"main"` → `return (MAIN);`
- `"printf"` → `return (PRINTF);`
- `"!"` → `return (NOT);`

- "false" → return (FALSO);
- "true" → return (VERDADEIRO);
- "{ATRIBUICAO}" → return (IGUAL);
- "{ID}" → return (IDENTIFICADOR);
- "{NUMBER}+" → return (INTEIRO);
- "{NUMBER}+"."NUMBER+" → return (REAL);
- "{STRING}" → return (STRING);
- "{OP_REL}" → return (OP_RELACIONAL);
- "{OP_ARIT}" → return (OP_ARIT);
- "{OP_LOG}" → return (OP_LOGICO);
- \n → linhas++;
- {ESPACO}* → /**/
- {NUMBER}+"."{NUMBER}*{ID}+ →
  printf("Número não suportado: %s (Utilize . como separador) - ",
  yytext);
  erros++;
  printf("Linha: %d \n", linhas);
- {NUMBER}+","{NUMBER}+ →
  printf("Número mal formatado: %s - ", yytext);
  erros++;
  printf("Linha: %d \n", linhas);
- {NUMBER}+{ID} →
  printf("Identificador inválido: %s - ", yytext);
  erros++;
  printf("Linha: %d \n", linhas);
- . →
  printf("ERRO: %c \n", yytext[0]);
  erros++;
  printf("Linha: %d \n", linhas);

Dessa forma, o lexer conseguirá realizar a quebra do código em diversos *tokens*.

3 Parser

Mostrarei como o parser ficou em relação aos tokens e as regras de produção:

```
# Aqui declaramos todas as cadeias terminais que usaremos no código
%token MAIN ABRE_PARENTESES FECHA_PARENTESES ABRE_CHAVE FECHA_CHAVE ABRE_COLCHETES F
%token IDENTIFICADOR INT FLOAT BOOL STRING_TIPO INTEIRO REAL STRING VERDADEIRO FALSO
%token SE SENAO WHILE FOR NOT PRINTF
%token OP_ARIT OP_LOGICO OP_RELACIONAL

# Aqui foi declarado algumas regras para usarem essas cadeias
%left OP_LOGICO
%left OP_RELACIONAL
%left OP_ARIT
%right NOT

# A partir daqui, veremos as regras de produção
%%
Programa_principal:
  MAIN ABRE_PARENTESES FECHA_PARENTESES ABRE_CHAVE Comandos FECHA_CHAVE
  | error { yyerror("Erro sintático: função main mal estruturada"); };

Comandos:
  Comando Comandos
  | Comando
  ;

Comando:
  Declaracao
  | Atribuicao
  | Condicional
  | Whileloop
  | Forloop
  | Print
  | error { yyerror("Erro sintático: Comando não existente"); };

Declaracao:
  Tipo Lista_var PONTO_E_VIRGULA
  ;

Tipo:
  INT
  | FLOAT
  | STRING_TIPO
  | BOOL
  ;

Lista_var:
```

```
  IDENTIFICADOR
  | IDENTIFICADOR VIRGULA Lista_var
  | IDENTIFICADOR ABRE_COLCHETES INTEIRO FECHA_COLCHETES
  ;

Atribuicao:
  IDENTIFICADOR IGUAL Expressao PONTO_E_VIRGULA
  ;

Expressao:
  Valor
  | IDENTIFICADOR
  | Expressao OP_ARIT Expressao
  | ABRE_PARENTESES Expressao FECHA_PARENTESES
  ;

Valor:
  INTEIRO
  | REAL
  | VERDADEIRO
  | FALSO
  | STRING
  | error { yyerror("Erro sintático: Atribuição mal formatada"); };

Print:
  PRINTF ABRE_PARENTESES STRING FECHA_PARENTESES PONTO_E_VIRGULA
  ;

Condicional:
  SE ABRE_PARENTESES Exp_logica FECHA_PARENTESES ABRE_CHAVE Comandos FECHA_CHAVE
  | SE ABRE_PARENTESES Exp_logica FECHA_PARENTESES ABRE_CHAVE Comandos FECHA_CHAVE S
  ;

Exp_logica:
  Exp_logica OP_LOGICO Exp_logica
  | Exp_relacional
  | NOT Exp_logica
  | ABRE_PARENTESES Exp_logica FECHA_PARENTESES
  | VERDADEIRO
  | FALSO
  | IDENTIFICADOR
  ;

Exp_relacional:
  Exp_aritmetica OP_RELACIONAL Exp_aritmetica
  | IDENTIFICADOR OP_RELACIONAL Exp_aritmetica
  | Exp_aritmetica OP_RELACIONAL IDENTIFICADOR
  | IDENTIFICADOR OP_RELACIONAL IDENTIFICADOR
```

```
  ;

Exp_aritmetica:
  Exp_aritmetica OP_ARIT Exp_aritmetica
  | Num
  | ABRE_PARENTESES Exp_aritmetica FECHA_PARENTESES
  ;

Num:
  INTEIRO
  | REAL
  ;

Whileloop:
  WHILE ABRE_PARENTESES Exp_logica FECHA_PARENTESES ABRE_CHAVE Comandos FECHA_CHAVE
  ;

Forloop:
  FOR ABRE_PARENTESES Atribuicao Exp_logica PONTO_E_VIRGULA Atribuicao FECHA_PARENTI
  ;
```

4 Modo de Usar

Para utilizar o analisador sintático, siga os passos abaixo:

1. Primeiramente, execute o comando `bash` para gerar o executvel do analisador sintático. O comando é:

   ```
   bash script.bash
   ```

2. Após gerar o executavel knot, use ele passando o arquivo de teste que deseja:

   ```
   ./knot Teste_Funcionando.txt
   ```

3. Como resultado, você verá a análise do conteúdo de `Teste_Funcionando.txt` exibida no terminal.

5 Manual da liguagem

Esta seção fornece uma visão geral da linguagem KNOT e como utilizá-la. A linguagem KNOT é inspirada em C, buscando simplicidade e facilidade de aprendizado.

5.1 Tipo de dados

int: representa números inteiros, como 10, -5, 0, 1000.
float: representa números de ponto flutuante (números com casas decimais), como 3.14, -2.5, 0.0, 1e6 (notação científica para 1 milhão). Use este tipo para representar valores que precisam de precisão decimal.
string: representa sequências de caracteres, como ”Olá, mundo!”, ”Knot”, ”123”. Strings são delimitadas por aspas duplas.
bool: representa valores booleanos, que podem ser true (verdadeiro) ou false (falso). Use este tipo para representar valores lógicos.

5.2 Operadores

KNOT suporta os operadores aritméticos comuns (+, -, *, /), relacionais (==, !=, ¡, ¿, ¡=, ¿=), lógicos(&& (E), ||(OU), ! (NÃO)), e de atribuição (=).

5.3 Estruturas de Controle de Fluxo

if-else (Condicional): Permite executar blocos de código diferentes dependendo de uma condição.
while (Laço Enquanto): Permite repetir um bloco de código enquanto uma condição for verdadeira.
for (Laço Para): Oferece uma maneira concisa de iterar sobre um intervalo de valores.

5.4 Função principal (main)

main(): A função main() é o ponto de entrada do programa KNOT. Todo código que deve ser executado precisa estar dentro da função main().

Alguns exemplos de uso será listado abaixo.

```
main() {
    int idade, quantidade, contador, i;
    float pi;
    pi = 3.14159;
    idade = 20;

    if (idade > 18) {
      printf("Maior de idade");
    } else {
      printf("Menor de idade");
    }

    contador = 0;
    while (contador < 10) {
      printf("Iteração do while");
      contador = contador + 1;
```

```
    }

    for (i = 0; i < 5; i = i + 1) {
      printf("Iteração do for");
    }
}
```

6 Conclusão

Concluímos este trabalho apresentando a construção da linguagem KNOT e sua integração com o analisador sintático utilizando a ferramenta GNU Bison. Abordamos o processo de análise sintática e sua importância no desenvolvimento de compiladores, onde o parser desempenha um papel fundamental na validação e organização da estrutura gramatical do código, preparando-o para etapas posteriores de compilação. A utilização de Bison permitiu uma implementação eficiente e simplificada do analisador sintático, integrando regras gramaticais que orientam a interpretação da linguagem conforme a estrutura formal planejada.

A construção do KNOT representa um estudo prático e detalhado dos elementos essenciais de uma linguagem de programação, como tokens, operadores e estruturas de controle de fluxo, que foram inspirados na linguagem C. Através da formalização desses componentes, conseguimos não apenas validar a funcionalidade da linguagem como também entender o processo de desenvolvimento de um compilador e sua complexidade. A experiência proporcionada por este projeto é fundamental para ampliar o entendimento dos desafios e das técnicas utilizadas na construção de sistemas de análise de código e compilação, essenciais para a área de Ciência da Computação.