Trabalho 2 - Análise Sintática

Arthur Correia, Matheus Queiroz, Pedro Marcos Estrela Dezembro, 2025

Introdução

Este projeto tem como objetivo o desenvolvimento de um analisador sintático para a linguagem de programação imperativa criada pelo grupo no Trabalho 1, utilizando as ferramentas **Flex** e **Bison**, no contexto da disciplina MATA61 – Compiladores.

Enquanto o analisador léxico (Trabalho 1) preocupava-se apenas em identificar os tokens individualmente, o analisador sintático aqui apresentado define a estrutura gramatical da linguagem. O objetivo final é receber o código-fonte como entrada e gerar uma **Árvore Sintática** que represente a hierarquia dos comandos, declarações e expressões.

A linguagem suporta tipagem estática, vetores, condicionais, laços e funções.

Descrição do Código e Implementação

O analisador sintático foi desenvolvido utilizando a ferramenta Bison, que trabalha em conjunto com o analisador léxico (Flex) produzido na etapa anterior. A estrutura do código é reconhecida e construída a partir dos tokens individuais (folhas) até formar a estrutura completa do programa (raiz).

A principal função deste analisador é validar a gramática da linguagem e construir a Árvore Sintática, que representa visualmente a hierarquia e a lógica do código fonte fornecido.

Estrutura de Dados da Árvore

Para permitir a visualização hierárquica exigida, foi definida uma estrutura de dados em C chamada `ASTNode`. Esta estrutura é responsável por armazenar as informações de cada nó da árvore:

- **Tipo:** A categoria do nó (ex: "declaracao", "if_statement", "operacao").
- **Valor:** O conteúdo textual, caso exista (ex: nome da variável "x", valor numérico "10").
- **Filhos:** Uma lista de ponteiros para outros nós, permitindo o aninhamento de comandos.

No Bison, a união de tipos (**%union**) foi configurada para manipular esses nós, permitindo que cada regra gramatical retorne um ponteiro para um nó da árvore.

```
%union {
    int intval;
    double floatval;
    char *strval;
```

```
    struct ASTNode *node;
}
```

Dessa forma, o Flex identifica os tokens e envia os valores brutos (strings ou números), enquanto o Bison organiza esses valores criando e conectando os nós da árvore.

Regras de Produção e Justificativas

Estrutura do Programa

A regra inicial da gramática é o **programa**. Ela define que um código válido é composto por uma lista sequencial de comandos.

```
programa:
    lista_comandos {
        ASTNode *root = create_node("programa", NULL);
        add_child(root, $1);
        print_tree(root, 0);
        free_tree(root);
    }
    ;
```

Lista de Comandos e Blocos

A construção do programa ocorre de forma recursiva através da regra **lista_comandos**, que permite o encadeamento infinito de instruções. Além disso, a regra **bloco** define o escopo de execução, fundamental para estruturas condicionais e laços.

```
lista_comandos:
    lista_comandos comando {
        $$ = $1;
        add_child($$, $2); }
    | comando {
        $$ = create_node("lista_comandos", NULL);
        add_child($$, $1); }
    ;
bloco:
    T_LCHAVE lista_comandos T_RCHAVE { $$ = $2; }
    | T_LCHAVE T_RCHAVE { $$ = create_node("bloco_vazio", NULL); }
    ;
```

Gerenciamento de Variáveis e Vetores

Além das variáveis escalares simples, a linguagem suporta vetores através da palavra-chave `serpente`.

```
declaracao_serpente:
    T_SERPENTE_KW T_ID T_LPAREN tipo T_RPAREN T_ATRIBUICAO T_LPAREN
    elementos_serpente T_RPAREN {
        $$ = create_node("serpente_declaracao", $2);
        add_child($$, $4);
        add_child($$, $8);
    }
    ;

elementos_serpente:
    elementos_serpente T_VIRGULA expressao { ... }
    | expressao { ... }
    ;
```

A gramática exige inicialização explícita para vetores, garantindo que a estrutura `serpente` seja sempre instanciada com seus elementos, facilitando a alocação de memória na geração da AST.

Estruturas de Repetição

A linguagem implementa dois tipos de laços: `enquanto` (baseado em condição) e `ate` (iterativo).

```
comando_enquanto:
    T_ENQUANTO T_LPAREN expressao T_RPAREN bloco {
        $$ = create_node("enquanto_loop", NULL);
        ...
    }
    ;

comando_ate:
    T_ATE T_LPAREN expressao T_RPAREN bloco {
        $$ = create_node("ate_loop", NULL);
        ...
    }
    ;
```

Ambas as estruturas segregam claramente a *condição* de parada do *bloco* de execução, permitindo que a AST represente o fluxo de controle de maneira estruturada.

Funções e Entrada/Saída

As funções são cidadãos de primeira classe na estrutura sintática, possuindo tipo de retorno e lista de parâmetros.

```
definicao_funcao:
    tipo T_ID T_LPAREN lista_parametros T_RPAREN bloco {
        $$ = create_node("funcao_definicao", $2);
        add_child($$, $1);
        add_child($$, $4);
        add_child($$, $6);
    }
    ;

comando_devolva:
    T_DEVOLVA expressao { ... } ;

comando_receba:
    T_RECEBA T_LPAREN T_ID T_RPAREN { ... } ;

comando_imprima:
    T_IMPRIMA T_LPAREN expressao T_RPAREN { ... } ;
```

A regra `definicao_funcao` captura toda a assinatura do método antes de processar seu corpo. Os comandos de I/O (`receba`, `imprima`) e retorno (`devolva`) são tratados como instruções atômicas.

Expressões e Hierarquia de Operadores

Para garantir a precedência correta das operações matemáticas e lógicas sem o uso de parênteses excessivos, a gramática foi estratificada em três níveis: `expressao`, `termo` e `fator`.

```
expressao:
    expressao_simples
    | expressao_simples T_IGUAL expressao_simples { ... }
    /* ... outros operadores relacionais ... */
    ;

expressao_simples:
    termo
    | expressao_simples T_SOMA termo { ... }
    | expressao_simples T_SUB termo { ... }
    | expressao_simples T_OU termo { ... }
    ;

termo:
    fator
    | termo T_MULT fator { ... }
```

```
17      | termo T_DIV fator { ... }
18      | termo T_E fator { ... }
19      ;
20
21  fator:
22      T_LPAREN expressao T_RPAREN { $$ = $2; }
23      | T_ID { ... }
24      | T_INTEIRO_LIT { ... }
25      /* ... outros literais ... */
26      ;
```

Justificativa:

- **Fator:** É a unidade indivisível (números, variáveis) ou expressões priorizadas por parênteses.
- **Termo:** Processa multiplicações, divisões e o operador lógico 'E', que possuem precedência sobre soma/subtração.
- **Expressão Simples:** Processa somas, subtrações e o operador lógico 'OU'.
- **Expressão:** O nível mais alto, onde ocorrem as comparações relacionais (igualdade, maior que, etc.), que têm a menor precedência.

Essa cascata garante que a expressão `2 + 3 * 4` seja parseada corretamente como `2 + (3 * 4)` na árvore, resultando em 14, e não 20.

Declarações e Tipos

A linguagem suporta a declaração de variáveis com ou sem inicialização imediata. A gramática trata esses dois casos de forma distinta para maior clareza na árvore gerada.

```
1  declaracao:
2      tipo T_ID
3      | tipo T_ID T_ATRIBUICAO expressao
4      ;
```

Ao separar as regras, o analisador consegue diferenciar semanticamente uma simples reserva de memória (apenas declaração) de uma atribuição inicial, gerando nós específicos (`declaracao` ou `declaracao_init`) que facilitam o entendimento do código.

Expressões Matemáticas e Precedência

Para garantir que operações como multiplicação ocorram antes da soma (precedência matemática), a gramática das expressões foi estratificada em três níveis hierárquicos:

- **Fator (Nível mais alto):** Reconhece as unidades básicas, como números, variáveis e expressões entre parênteses.
- **Termo (Nível intermediário):** Processa operações de multiplicação ($*$) e divisão ($/$), que têm prioridade sobre a soma.
- **Expressão Simples (Nível base):** Processa somas ($+$) e subtrações ($-$).

Essa divisão obriga o analisador a resolver primeiro o que está "mais fundo" na hierarquia (fatores e termos) antes de resolver as somas. Isso elimina a ambiguidade sem a necessidade de código complexo em C, utilizando a própria natureza da gramática para definir a ordem de avaliação.

Exemplos de Análise Sintática

A seguir, são apresentados exemplos de código fonte na linguagem desenvolvida e a respectiva Árvore Sintática gerada pelo analisador sintático.

Inicialização de Variáveis

Na linguagem idealizada, a inicialização de variáveis segue uma lógica similar àquela encontrada em linguagens de programação convencionais. Sob essa perspectiva, os tipos de dados disponíveis são definidos da seguinte maneira:

Inteiro

Equivalente a **int**. É utilizado para representar valores numéricos inteiros.

```
// Entrada:
inteiro a = 10;

// Saida:
programa
  lista_comandos
    declaracao_init
      tipo
        inteiro
      id
        a
      literal_inteiro
        10
```

Quebrado

Equivalente a **float**. É utilizado para representar valores numéricos com casas decimais.

```
// Entrada:
quebrado b = 20.5;

// Saida:
programa
  lista_comandos
    declaracao_init
      tipo
        quebrado
      id
        b
      literal_quebrado
        20.5
```

Texto

Equivalente a **string**. É utilizado para representar sequências de caracteres.

```
// Entrada:
texto c = "ola mundo";

// Saida:
programa
  lista_comandos
    declaracao_init
      tipo
        texto
      id
        c
      literal_texto
        "ola mundo"
```

Caractere

Equivalente a **char**. É utilizado para representar valores de caractere único.

```
// Entrada:
caractere d = "d";

// Saida:
programa
  lista_comandos
    declaracao_init
      tipo
        caractere
      id
```

```
        d
      literal_texto
        "d"
```

Fato

Equivalente a **bool**. É utilizado para representar valores lógicos. Na linguagem, pode assumir os valores **real** e **fake** que representam, respectivamente, **true** e **false**.

```
// Entrada:
fato x = real;
fato y = fake;

// Saida:
programa
  lista_comandos
    declaracao_init
      tipo
        fato
      id
        x
      literal_fato
        real
    declaracao_init
      tipo
        fato
      id
        y
      literal_fato
        fake
```

Serpente

Equivalente a **vetor**. É utilziado para armazenar sequências de elementos do mesmo tipo;

```
// Entrada:
serpente lista_fatos (fato) = (fake, real, fake);

// Saida:
programa
  lista_comandos
    serpente_declaracao
      lista_fatos
      tipo
        fato
      elementos
        literal_fato
```

```
          fake
        literal_fato
          real
        literal_fato
          fake
```

Estruturas de Controle e Repetição

Condicional Se-Senao

Equivalente a **if-else**. O comando **se** é utilizado para avaliar uma condição e executa o bloco correspondente caso a condição seja verdadeira. O comando **senao** define o bloco alternativo a ser executado quando a condição do **se** não é satisfeita.

```
// Entrada:
se (a > b) {
    b = b + 1;
}
senao {
    a = a + 1;
}

// Saida:
programa
  lista_comandos
    se_senao_statement
      condicao
        op_relacional
          >
          id
            a
          id
            b
      lista_comandos
        atribuicao
          id
            b
          op_aritmetica
            +
            id
              b
            literal_inteiro
              1
      bloco_senao
        lista_comandos
          atribuicao
            id
              a
            op_aritmetica
```

```
            +
            id
              a
            literal_inteiro
              1
```

Enquanto

Equivalente a **while**. O comando **enquanto** implementa laços de repetição, permitindo que um bloco de código seja executado repetidamente enquanto uma condição lógica for verdadeira.

```
// Entrada:
enquanto (c >= 15) {
  c = c - 1;
}

// Saida:
\begin{lstlisting}[style=MyLang]
programa
  lista_comandos
    enquanto_loop
      condicao
        op_relacional
          >=
          id
            c
          literal_inteiro
            15
      lista_comandos
        atribuicao
          id
            c
          op_aritmetica
            -
            id
              c
            literal_inteiro
              1
```

Até

Equivalente a **for**. O comando **ate** permite definir um contador e executar um bloco de código repetidas vezes até que determinada condição seja satisfeita.

```
// Entrada:
```

```
ate (i > d) {
  i = i + 1;
}

// Saida:
programa
  lista_comandos
    ate_loop
      condicao
        op_relacional
          >
          id
            i
          id
            d
      lista_comandos
        atribuicao
          id
            i
          op_aritmetica
            +
            id
              i
            literal_inteiro
              1
```

Entrada e Saída de Dados

Receba

Equivalente a **input**. É utilizado para realizar a entrada de dados, permitindo que o programa capture informações fornecidas pelo usuário durante a execução.

```
// Entrada:
inteiro x;
receba(x);

// Saida:
programa
  lista_comandos
    declaracao
      tipo
        inteiro
      id
        x
    receba
      x
```

Imprima

Equivalente a **print**. É responsável pela saída de dados, permitindo exibir informações na tela durante a execução do programa.

```
// Entrada:
inteiro y = 10;
imprima(y);

// Saida:
programa
  lista_comandos
    declaracao_init
      tipo
        inteiro
      id
        y
      literal_inteiro
        10
    imprima
      id
        y
```

Funções

Na linguagem, as funções podem ser dos respectivos tipos: inteiro, quebrado, nada.

Devolva

Equivalente a **return**. É um comando de controle que interrompe a execução de uma função e retorna um valor (ou nenhum) ao ponto onde ela foi chamada.

```
// Entrada:
devolva a;

// Saida:
programa
  lista_comandos
    devolva
      id
        a
```

Funções do tipo Inteiro

São as funções que devolvem um valor do tipo inteiro.

```
// Entrada:
inteiro exemplo(inteiro a) {
    devolva a;
}

// Saida:
programa
  lista_comandos
    funcao_definicao
      exemplo
      tipo
        inteiro
      parametros
        parametro
          a
          tipo
            inteiro
      lista_comandos
        devolva
          id
            a
```

Funções do tipo Quebrado

São as funções que devolvem um valor do tipo quebrado.

```
// Entrada:
quebrado exemplo(quebrado b) {
  devolva b;
}

// Saida:
programa
  lista_comandos
    funcao_definicao
      exemplo
      tipo
        quebrado
      parametros
        parametro
          b
          tipo
            quebrado
      lista_comandos
        devolva
          id
```

```
21              b
```

Funções do tipo Nada

São as funções que não devolvem nada.

```
1  // Entrada:
2  nada exemplo() {
3  }
4
5  // Saida:
6  programa
7    lista_comandos
8      funcao_definicao
9        exemplo
10       tipo
11         nada
12       parametros
13         nenhum
14       bloco_vazio
```

Alterações em Relação ao Trabalho 1

Para viabilizar a análise sintática, foram feitas alterações na camada léxica (Flex) definida no Trabalho 1 :

1. **Remoção de Impressões Diretas:** O Flex deixou de imprimir os tokens (ex: ‘<inteiro>‘) diretamente na saída padrão.
2. **Retorno de Tokens:** O Flex agora retorna constantes inteiras (definidas pelo Bison) que representam o tipo do token.
3. **Passagem de Valores:** Foi implementada a lógica de ‘strdup(yytext)‘ para preencher a estrutura ‘yylval‘, permitindo que o Bison tenha acesso ao texto original dos identificadores e literais para a construção da árvore.

Considerações Finais

O desenvolvimento do analisador sintático com Bison permitiu estruturar formalmente a gramática da linguagem. A abordagem de construção da árvore baseada em manipulação de strings provou-se eficaz para atender aos requisitos de visualização hierárquica solicitados, mantendo o código modular e legível. A gramática livre de contexto elaborada cobre todas as especificações do projeto, incluindo expressões complexas, funções e estruturas de controle aninhadas.

GitHub

Link para acesso ao repositório do projeto: https://github.com/arthurscorreia/syntax-analysis.