Colegiado de Engenharia da Computação
Universidade Federal do Vale do São Francisco

DOCUMENTAÇÃO – PROJETO DE COMPILADORES

Prof. Marcus Vinicius Midena Ramos

Elpidio Hans dos Santos Leao Junior
Manoel Ferreira de Santana Filho

Resumo

Neste documento são apresentados diversos aspectos relacionados a uma linguagem de programação que se pretende compilar. Inicia-se com a descrição geral da linguagem, incluindo sua gramática, análise sintática e análise léxica. A gramática é detalhada, abrangendo desde as produções das regras até os símbolos terminais e não-terminais. Discute-se a arquitetura de um compilador, construído com Java, destacando a análise léxica, sintática, de contexto e a geração de código intermediário. Cada etapa do compilador requer estruturas de dados específicas e algoritmos adequados para analisar, verificar e traduzir o código-fonte em uma forma executável. Tem-se atenção especial às formas de análise, detalhando a gramática sintática, as manipulações realizadas na gramática original, a verificação da condição LL(1) e as técnicas empregadas para tornar a análise eficiente e determinística. É exemplificada a construção de árvores sintáticas abstratas a partir de um programa-fonte para mostrar como elas representam a estrutura hierárquica do código. Na análise de contexto é abordada a fase de identificação e verificação de tipos. É descrita a utilização da tabela de símbolos e algoritmos para reconhecer e registrar os identificadores, bem como a verificação de consistência dos tipos de dados em expressões e comandos. Quanto ao ambiente de execução, é elucidado o tipo de máquina utilizada, a avaliação de expressões, a representação de dados, as formas de alocação de memória e a passagem de parâmetros e retorno de valor de função. Por fim, é citada a linguagem-objeto gerada pelo compilador. É explicada sua relação de instruções, sintaxe e semântica. A linguagem-objeto é uma representação intermediária que pode ser traduzida para código de máquina específico ou interpretada por uma máquina virtual.

1. Linguagem-fonte

O conjunto a seguir oferece uma especificação formal ou gramática de uma linguagem de programação, descrevendo suas regras de sintaxe. Cada regra define uma produção na gramática da língua. A notação usada é semelhante a Backus-Naur Form (BNF) ou Extended Backus-Naur Form (EBNF), que é comumente usada para descrever a sintaxe de linguagens de programação. Nesta gramática, várias construções de linguagem e seus relacionamentos são definidos.

<atribuição> ::=
<variável> := <expressão>
<bool-lit> ::=
true
| false
<comando> ::=
<atribuição>
| <condicional>
| <iterativo>
| <comando-composto>
<comando-composto> ::=
begin <lista-de-comandos> end
<condicional> ::=
if <expressão> then <comando> (else <comando> | <vazio>)
<corpo> ::=
<declarações> <comando-composto>
<declaração> ::=
<declaração-de-variável>
<declaração-de-variável> ::=
var <lista-de-ids> : <tipo>
<declarações> ::=
<declaração> ;
| <declarações> <declaração> ;
| <vazio>
<digito> ::=
0
| 1
| 2
| ...
| 9

<expressão> ::=
<expressão-simples>
| <expressão-simples> <op-rel> <expressão-simples>
<expressão-simples> ::=
<expressão-simples> <op-ad> <termo>
| <termo>
<fator> ::=
<variável>
| <literal>
| "(" <expressão> ")"
<float-lit> ::=
<int-lit> . <int-lit>
| <int-lit> .
| . <int-lit>
<id> ::=
<letra>
| <id> <letra>
| <id> <digito>
<int-lit> ::=
<digito>
| <int-lit> <digito>
<iterativo> ::=
while <expressão> do <comando>
<letra> ::=
a
| b
| c
| ...
| z
<lista-de-comandos> ::=
<comando> ;
| <lista-de-comandos> <comando> ;
| <vazio>
<lista-de-ids> ::=
<id>
| <lista-de-ids> , <id>
<literal> ::=
<bool-lit>

| <int-lit>
| <float-lit>
<op-ad> ::=
+
| -
| or
<op-mul> ::=
*
| /
| and
<op-rel> ::=
<
| >
| <=
| >=
| =
| <>
<outros> ::=
!
| @
| #
| ...
<programa> ::=
program <id> ; <corpo> .
<termo> ::=
<termo> <op-mul> <fator>
| <fator>
<tipo> ::=
| <tipo-simples>
<tipo-simples> ::=
integer
| real
| boolean
<variável> ::=
<id>
<vazio> ::=

A seguir, temos uma explicação mais detalhada da gramática, cada regra de produção com sua finalidade e componentes.

1. `<programa> ::= programa <id> ; <corpo> .`

Esta regra define a estrutura de um programa. Começa com a palavra-chave "programa" seguida de um identificador (nome do programa). Em seguida, há um ponto e vírgula, o corpo do programa é definido pela regra `<corpo>` e, finalmente, o programa termina com um ponto.

2. `<corpo> ::= <declarações> <comando-composto>`

A regra `<corpo>` representa o corpo de um programa. Consiste em duas partes: declarações (`<declarações>`) e um comando composto (`<comando-composto>`). As declarações são opcionais, mas o comando composto é obrigatório.

3. `<declarações> :: = <declaração> ; | <declarações> <declaração> ; | <vazio>`

Esta regra define uma sequência de declarações, separadas por ponto e vírgula. Ele permite várias declarações em um programa. É expresso usando recursão, o que permite que várias declarações sejam encadeadas. O símbolo `<vazio>` representa uma produção vazia, indicando que pode não haver declarações.

4. `<declaração> ::= <declaração-de-variável>`

Esta regra representa uma única declaração, que, neste caso, é uma declaração de variável (`<declaração-de-variável>`).

5. `<declaração-de-variável> ::= var <lista-de-ids> : <tipo>`

A regra `<declaração-de-variável>` define uma declaração de variável. Começa com a palavra-chave "var, seguida de uma lista de identificadores (`<lista-de-ids>`), dois pontos e o tipo de variável (`<tipo>`).

6. `<lista-de-ids> ::= <id> | <lista-de-ids> , <id>`

Esta regra define uma sequência de um ou uma lista de identificadores.

Descrição da linguagem-fonte

Relação e Estrutura de Separadores

Na linguagem definida pela gramática, os separadores são símbolos que são utilizados para delimitar ou separar elementos do código. Esses separadores incluem:

- **Ponto e vírgula (;):** Utilizado para separar declarações e comandos individuais.
- **Dois pontos (:) e vírgula (,):** Utilizados para delimitar listas de identificadores e separar itens em uma lista.
- **Parênteses (( )):** Utilizados para agrupar expressões ou argumentos de funções.
- **Palavras-chave:** São palavras reservadas pela linguagem para definir comandos, declarações e outras estruturas. Alguns exemplos incluem "program," "var," "begin," "end," "if," "else," "while," "integer," "real," "boolean," etc.

Sintaxe Livre de Contexto

A gramática fornecida é uma sintaxe livre de contexto, que é uma classificação de gramática formal em que as produções podem ser aplicadas independentemente do contexto. Nesse tipo de gramática, cada regra de produção é aplicada independentemente das outras regras. Ela permite a construção de sequências de símbolos terminais e não terminais de acordo com as regras definidas na gramática.

As regras de produção na gramática seguem a notação BNF/EBNF, representando a estrutura gramatical em um formato facilmente compreensível.

Sintaxe Dependente de Contexto

A gramática fornecida não inclui especificamente aspectos de sintaxe dependente de contexto. Sintaxe dependente de contexto é uma extensão da sintaxe livre de contexto em que a interpretação de uma regra de produção pode depender do contexto ou dos símbolos ao redor da construção sendo analisada. A gramática fornecida é puramente sintaxe livre de contexto, mas pode ser estendida para incluir aspectos dependentes de contexto, se necessário.

Semântica

A gramática fornecida define apenas a sintaxe da linguagem, ou seja, as regras para a estruturação correta do código. No entanto, não inclui a especificação da semântica da linguagem, ou seja, o significado das construções do programa e como elas afetam a execução do código.

A semântica de uma linguagem de programação envolve as regras e o comportamento associados a construções específicas, como declaração de variáveis, atribuições, comandos condicionais, loops, etc. Além disso, inclui as regras para a avaliação de expressões, tratamento de tipos de dados, escopo de variáveis, e outros aspectos que afetam o comportamento do programa em tempo de execução.

A semântica é definida em um nível mais alto de abstração e geralmente é descrita usando uma mistura de texto, pseudocódigo e exemplos. Além disso, a semântica pode variar de uma implementação de compilador/interpretador para outra, desde que a execução final seja consistente com as especificações da linguagem.

2. Descrição geral da arquitetura do compilador

Uma descrição geral da arquitetura do compilador para esta linguagem envolve várias etapas e componentes. Temos uma arquitetura simplificada, que abrange as etapas essenciais do processo de compilação. Essa arquitetura pode ser expandida e adaptada conforme necessário para lidar com recursos adicionais da linguagem. A linguagem utilizada para a construção do compilador é Java, por se tratar de uma linguagem de programação orientada a objetos, o que permite melhorias da flexibilidade e reusabilidade do código.

A arquitetura geral do compilador inclui as seguintes etapas:

Análise Léxica (Scanner)

- Essa é a primeira etapa do compilador. O analisador léxico, também conhecido como scanner, lê o código-fonte do programa e divide-o em uma sequência de tokens (símbolos léxicos).
- Tokens são unidades léxicas significativas, como palavras-chave, identificadores, literais, operadores e símbolos especiais.
- Os tokens são usados como entrada para a próxima etapa do compilador, a análise sintática.

Análise Sintática (Parser)

- Nesta etapa, o analisador sintático recebe a sequência de tokens gerada pelo analisador léxico e verifica se o código segue a gramática livre de contexto da linguagem.
- O analisador sintático constrói uma árvore sintática abstrata (AST) que representa a estrutura hierárquica do código.
- Se houver algum erro de sintaxe, o compilador gera uma mensagem de erro apontando o local no código onde o erro ocorreu.

Análise Semântica

- Após a construção da AST, a análise semântica é realizada para verificar aspectos semânticos do programa, como a verificação de tipos, escopo de variáveis e outras regras da linguagem.
- A análise semântica garante que o programa é semanticamente correto antes de continuar com a geração de código intermediário.

Geração de Código Intermediário

- Nesta etapa, o compilador traduz a AST em uma representação intermediária, que é uma forma de código de mais baixo nível do que o código-fonte, mas ainda independente da arquitetura do computador.
- A representação intermediária permite otimizações e é mais fácil de converter em código de máquina.

Otimização de Código Intermediário

- O compilador pode aplicar várias otimizações ao código intermediário para melhorar o desempenho do programa final.
- Essas otimizações visam reduzir o uso de recursos, como memória e tempo de execução, e melhorar a eficiência do programa gerado.

Geração de Código de Máquina

- Nesta etapa, o compilador traduz o código intermediário em código de máquina, específico para a arquitetura do computador.
- O código de máquina é a sequência de instruções binárias que o processador do computador é capaz de executar diretamente.

Montagem e Vinculação

- O código de máquina é então montado para criar um arquivo executável. O montador converte as instruções de máquina em um formato executável que pode ser carregado e executado pelo sistema operacional.
- Se o programa depender de bibliotecas externas, a etapa de vinculação é realizada para conectar o código do programa com as bibliotecas necessárias.

Execução do Programa

O programa compilado é agora um arquivo executável que pode ser executado no computador. O sistema operacional carrega o programa em memória e o executa, produzindo a saída conforme especificado pelo código-fonte.

3. Fundamentação teórica e técnicas empregadas na análise sintática

As técnicas empregadas na análise sintática da linguagem podem ser resumidas em dois métodos principais: Análise Descendente (Top-Down) e Análise Ascendente (Bottom-Up). Ambos os métodos têm o objetivo de verificar se o código-fonte segue as regras gramaticais definidas pela linguagem.

Análise Descendente (Top-Down)

- A análise descendente começa do símbolo inicial da gramática e tenta construir a árvore sintática, descendo da raiz para as folhas.
- Essa técnica geralmente utiliza recursão para chamar os procedimentos correspondentes às regras de produção e tenta fazer correspondências com as partes do código-fonte para cada produção.

- O compilador tenta aplicar as regras de produção da gramática, começando pela raiz até os símbolos terminais. Se o código-fonte for estruturado corretamente e seguir a gramática, a análise terá sucesso.
- As variantes da análise descendente incluem o Predictive Parsing e Recursive Descent Parsing.

Análise Ascendente (Bottom-Up)

- A análise ascendente começa a partir dos símbolos terminais do código-fonte e tenta construir a árvore sintática subindo em direção à raiz.
- Nessa técnica, o compilador tenta identificar trechos de código que correspondam a regras de produção da gramática, reduzindo-os a símbolos não terminais até alcançar o símbolo inicial da gramática.
- A análise ascendente é mais geral do que a análise descendente e pode analisar uma ampla variedade de gramáticas, incluindo gramáticas ambíguas.
- Algumas variantes da análise ascendente incluem o SLR (Simple LR), LALR (Look-Ahead LR), e LR(1) parsing.

Para uma linguagem como a descrita na gramática referida, é provável que seja utilizada uma análise descendente ou uma variante da análise ascendente. A escolha entre esses métodos depende da complexidade da gramática e dos requisitos específicos da linguagem.

Ambas as técnicas de análise sintática são fundamentais para garantir que o código-fonte seja analisado corretamente e para construir a árvore sintática que representa a estrutura hierárquica do programa. Isso permite que o compilador prossiga com as etapas de análise semântica, geração de código intermediário e, finalmente, a produção de código de máquina executável.

4. Análise léxica

A Análise Léxica é a primeira fase do processo de compilação, responsável por identificar e categorizar os elementos léxicos (tokens) presentes no código-fonte. Nesta etapa, o código-fonte é dividido em uma sequência de tokens que são utilizados como entrada para a Análise Sintática (Parser).

Relação de Tokens

A relação de tokens descreve as categorias léxicas em que as sequências de caracteres do código-fonte são classificadas. Cada categoria representa uma classe de elementos léxicos com significado específico na linguagem. Alguns exemplos de categorias léxicas comuns incluem:

- **Identificadores:** Nomes de variáveis, funções ou outras entidades definidas pelo programador.
- **Literais:** Números (inteiros e reais) e strings.
- **Palavras-chave:** Palavras reservadas pela linguagem que têm significado especial.
- **Operadores:** Símbolos usados para realizar operações, como adição, subtração, multiplicação, etc.
- **Delimitadores:** Símbolos que delimitam blocos de código, como parênteses, chaves, colchetes e ponto e vírgula.

Gramática Léxica

A gramática léxica define as regras para a formação de cada categoria léxica ou token na linguagem. Em outras palavras, ela descreve como os padrões léxicos são reconhecidos no código-fonte. As regras da gramática léxica geralmente são definidas usando expressões regulares.

Verificação da Condição LL(1) na Gramática

A condição LL(1) é uma propriedade importante para garantir que a gramática possa ser analisada de maneira determinística e sem ambiguidades usando o método de análise sintática LL(1). Essa condição indica que, dado um não-terminal, a escolha da regra de produção é única (1) com base no próximo token de entrada (LL).

Para verificar a condição LL(1) em uma gramática, é necessário analisar os conjuntos FIRST e FOLLOW dos não-terminais. O conjunto FIRST de um não-terminal é o conjunto de símbolos terminais que podem iniciar uma sequência derivada desse não-terminal. O conjunto FOLLOW de um não-terminal é o conjunto de símbolos terminais que podem aparecer imediatamente após uma sequência derivada desse não-terminal.

A condição LL(1) é satisfeita se, para cada par de regras de produção com o mesmo não-terminal à esquerda, a interseção dos conjuntos FIRST das produções for vazia ou disjunta e se a interseção dos conjuntos FOLLOW do não-terminal à esquerda e do próximo token de entrada também for vazia ou disjunta.

Técnicas Utilizadas e Visão Geral do Funcionamento

A Análise Léxica é geralmente implementada com a ajuda de ferramentas geradoras de analisadores léxicos, como Flex (para C/C++) ou Lex (para linguagens baseadas em UNIX). Essas ferramentas permitem definir padrões léxicos usando expressões regulares e associar ações para cada token reconhecido.

A seguir, uma visão geral do funcionamento da Análise Léxica:

- O analisador léxico lê o código-fonte caractere a caractere.
- Ele aplica as regras definidas na gramática léxica para reconhecer os tokens.

Quando um padrão é reconhecido, o analisador léxico gera um token correspondente, armazenando o valor do lexema (sequência de caracteres que representa o token) e sua categoria léxica.

Os tokens gerados são fornecidos à próxima etapa do compilador, que é a Análise Sintática.

Exemplos de Entradas e Saídas

Para comprovar a aceitação de cadeias bem formadas e a rejeição de cadeias mal formadas, temos alguns exemplos de teste:

Exemplo de cadeia bem formada:

```
program exemplo;
var x, y, z: integer;

begin
  x := 10;
  y := 5;
```

```
  z := x + y;
  if z > 15 then
    z := z - 10
  else
    z := z + 5;
...
```

```
> Análise Léxica - START
< Análise Léxica - END

> Análise Sintática - START
< Análise Sintática - END

> Impressão da Ast - START
---> Iniciando impressão da árvore
000 [PROGRAMA]
001 ├TOKEN exemplo
001 └[CORPO]
002  ├[DECL-VAR]
003  ||├[TIPO-SIMPLES]
004  |||└integer
003  ||├[LI]
004  |||└x
005  ||| └[LI]
006  |||  └y
```

Fonte: o autor

No caso da cadeia bem formada, a compilação segue para o passo seguinte.

Exemplo de cadeia mal formada:

```
program exemplo;
var x, y, z: integer;

begin
  x := 10;
  y := 5;
  z := x + y;
  if z > 15 then
    z := z - 10
  else
    z := (z + 5;
...
```

```
> Análise Léxica - START
< Análise Léxica - END

> Análise Sintática - START
        !ERRO - ANÁLISE SINTÁTICA
          * Linha: 13, Posição: 20
          └ Token lido inesperado: [;] (token do tipo 29) enquanto era experado um ")" (token do tipo 26).
        A ANÁLISE SINTÁTICA FOI INTERROMPIDA DEVIDO A ERROS OCORRIDOS
PS E:\Documentos\UNIVASF_2022_2\Compiladores\ProjetoCompilador\Projeto\ProjetoCompilador\src\src>
```

Fonte: o autor

No caso da cadeia mal formada, a análise sintática é interrompida devido a erros ocorridos.

Nesse exemplo, o analisador léxico reconhece corretamente cada token na cadeia bem formada, mas relata um erro léxico na cadeia mal formada devido à falta do parêntese de fechamento.

5. Análise sintática

A Análise Sintática é a segunda fase do processo de compilação, responsável por verificar se o código-fonte segue as regras da gramática sintática da linguagem. Essa etapa utiliza a gramática formal da linguagem para construir a árvore sintática abstrata (AST), representando a estrutura hierárquica do programa.

Gramática Sintática

A gramática sintática define a estrutura e a sintaxe da linguagem de programação. Ela consiste em regras de produção que especificam como as construções sintáticas da linguagem são formadas. Cada regra de produção consiste em um não-terminal (um símbolo não terminal) e uma sequência de símbolos terminais e/ou não terminais que podem substituir o não-terminal. A gramática sintática é geralmente especificada usando a notação BNF (Forma Normal de Backus-Naur) ou EBNF (Forma Normal Estendida de Backus-Naur).

Manipulações Efetuadas na Gramática Original

Antes de realizar a Análise Sintática, algumas manipulações podem ser efetuadas na gramática original para torná-la mais adequada para a análise. Essas manipulações incluem:

1. **Eliminação de Ambiguidades**: Se a gramática original for ambígua (ou seja, uma sequência de símbolos pode ter mais de uma árvore sintática válida), a ambiguidade deve ser eliminada para facilitar a análise. Isso pode ser feito por meio de fatoração à esquerda e eliminação de recursão à esquerda.

2. **Eliminação de Recursão à Esquerda**: Se a gramática contiver recursão à esquerda, ela pode ser eliminada por meio de técnicas como substituição de regras recursivas ou uso de produções aninhadas.

3. **Fatoração à Esquerda**: Se a gramática contiver ambiguidade devido a fatoração à esquerda, é possível aplicar a técnica de fatoração à esquerda para evitar conflitos.

Verificação da Condição LL(1) na Gramática

A condição LL(1) é uma propriedade importante para garantir que a gramática possa ser analisada de maneira determinística e sem ambiguidades usando o método de análise sintática LL(1). Essa condição é verificada para cada não-terminal da gramática e baseia-se nos conjuntos FIRST e FOLLOW.

A condição LL(1) é satisfeita se, para cada par de regras de produção com o mesmo não-terminal à esquerda, a interseção dos conjuntos FIRST das produções for vazia ou disjunta e se a interseção dos conjuntos FOLLOW do não-terminal à esquerda e do próximo símbolo de entrada também for vazia ou disjunta.

Técnicas Utilizadas

Existem várias técnicas de análise sintática que podem ser usadas para implementar a Análise Sintática:

1. **Análise Descendente (Top-Down):** Essa técnica começa com o símbolo inicial da gramática e tenta construir a árvore sintática descendo da raiz para as folhas. Pode ser implementada de forma recursiva (Recursive Descent Parsing) ou usando tabelas de análise (Predictive Parsing).

2. **Análise Ascendente (Bottom-Up):** Essa técnica começa com os símbolos terminais do código-fonte e tenta construir a árvore sintática subindo em direção à raiz. Existem várias variantes dessa técnica, como SLR (Simple LR), LALR (Look-Ahead LR) e LR(1) parsing.

Exemplos de Entradas e Saídas:

Vamos considerar o exemplo de uma soma simples:

Exemplo de cadeia bem formada:

```
! SOMA SIMPLES -
program exemplo;
var x, y, z: integer;

begin
   x := 10;
   y := 5;
   z := x + y;
end.
```

output: Árvore Sintática Abstrata (AST).

```
> Análise Sintática - START
< Análise Sintática - END

> Impressão da Ast - START
---> Iniciando impressão da árvore
000 [PROGRAMA]
001 ├TOKEN exemplo
001 └[CORPO]
002  ├[DECL-VAR]
003  │├[TIPO-SIMPLES]
004  ││└integer
003  ├├[LI]
004  ├│└x
005  ├│ └[LI]
006  ├│  └y
007  ├│   └[LI]
008  ├│    └z
002  └[LC]
003   ├[COM-ATRIBUIÇÃO]
004   │└x
005   │ ├=
005   │ └10
003   └[LC]
004    ├[COM-ATRIBUIÇÃO]
005    │└y
006    │ ├=
006    │ └5
004    └[LC]
005     └[COM-ATRIBUIÇÃO]
006      └z
007       ├=
007       └x
008        ├+
008        └y
< Impressão da Ast - END
```

Fonte: o autor

Exemplo de cadeia mal formada:

```
! SOMA SIMPLES -
program exemplo;
```

```
var x, y, z: integer;

begin
   x := 10;
   y := 5;
   z := x + * y;
end.
```

output: Erro de Sintaxe: Token lido inesperado "*".

```
> Análise Léxica - START
< Análise Léxica - END

> Análise Sintática - START
        !ERRO - ANÁLISE SINTÁTICA
          * Linha: 8, Posição: 14
          └ Token lido inesperado: [*] (token do tipo 32) enquanto era esperado um "id" (token do
 do tipo 1), "false" (token do tipo 2), "int-lit" (token do tipo 8), "float-lit" (token do tipo 9)
 25).
        A ANÁLISE SINTÁTICA FOI INTERROMPIDA DEVIDO A ERROS OCORRIDOS
PS E:\Documentos\UNIVASF_2022_2\Compiladores\ProjetoCompilador\Projeto\ProjetoCompilador\src\src>
```

Fonte: o autor

Nesse exemplo, a análise sintática aceita corretamente a cadeia bem formada, construindo uma árvore sintática abstrata que representa a expressão aritmética. Na cadeia mal formada, a análise sintática encontra um erro de sintaxe devido à falta de um fator após o operador de adição.

6. Descrição das principais estruturas de dados utilizadas

Para implementar o compilador para a linguagem apresentada, diversas estruturas de dados são utilizadas para armazenar informações relevantes durante as diferentes etapas do processo de compilação. Algumas das principais estruturas de dados que podem ser usadas são:

Tabela de Símbolos (Symbol Table)

- A tabela de símbolos é uma estrutura de dados que mantém informações sobre os identificadores (variáveis, funções, etc.) encontrados no código-fonte.
- Cada entrada na tabela de símbolos armazena o nome do identificador, o tipo de dado associado, seu escopo (se aplicável) e outras informações relevantes.

- A tabela de símbolos é usada durante a análise léxica e sintática para verificar a validade dos identificadores e durante a análise semântica para resolver referências e garantir a consistência do código.

Árvore Sintática Abstrata (Abstract Syntax Tree - AST)

- A AST é uma estrutura de dados hierárquica que representa a estrutura sintática do programa em formato de árvore.
- Cada nó na árvore representa um elemento sintático do código, como expressões, comandos, operadores, literais, etc.
- A AST é construída durante a análise sintática e é usada como base para a análise semântica e a geração de código intermediário.

Pilha (Stack)

- A pilha é uma estrutura de dados que é amplamente utilizada durante a análise sintática.
- Na análise descendente (Recursive Descent Parsing), uma pilha é usada para rastrear os símbolos não terminais sendo processados e suas regras de produção correspondentes.
- Na análise ascendente (Bottom-Up Parsing), a pilha é usada para realizar reduções das sequências de tokens para os não terminais correspondentes.

Tabela de Análise (Parsing Table)

- Em algumas técnicas de análise sintática, como Predictive Parsing, é comum usar uma tabela de análise que mapeia o estado atual do analisador (estado da pilha) e o próximo token de entrada para a ação a ser tomada (deslocar, reduzir ou aceitar).
- Essa tabela é construída a partir da gramática e pode ser usada para tornar a análise mais eficiente e determinística.

Árvore de Derivação (Derivation Tree)

- A árvore de derivação é outra estrutura de dados hierárquica que representa como uma sequência de regras de produção é aplicada para derivar a cadeia de tokens do código-fonte.
- Ela não é tão comum como a AST, mas pode ser útil para depurar e entender como a análise sintática está funcionando.

Estruturas para Otimização e Geração de Código

- Durante as etapas de otimização de código intermediário e geração de código de máquina, diversas outras estruturas de dados podem ser usadas, como tabelas para otimizações locais, listas para controle de fluxo e representações intermediárias, entre outras.

Essas são algumas das principais estruturas de dados que podem ser utilizadas para implementar o compilador para a linguagem apresentada. A escolha das estruturas específicas dependerá da abordagem de implementação e das técnicas utilizadas em cada etapa do processo de compilação.

7. Montagem e impressão da árvore de sintaxe abstrata

A montagem e impressão da Árvore de Sintaxe Abstrata (AST) envolve a construção da estrutura de dados da árvore e um algoritmo de percurso para imprimir a árvore de forma organizada. Vamos descrever os principais componentes envolvidos e apresentar um exemplo de programa-fonte e sua respectiva AST.

Estrutura de Dados para a AST

A AST é uma estrutura de dados hierárquica, onde cada nó representa um elemento sintático do código-fonte.

Algoritmo de Montagem da AST

O algoritmo de montagem da AST é implementado durante a análise sintática do compilador. À medida que a análise sintática percorre o código-fonte e reconhece as construções sintáticas, ela constrói a árvore sintática abstrata.

Algoritmo de Impressão da AST

Após a montagem da AST, é possível imprimir a árvore de forma organizada para visualização e depuração. Um algoritmo de impressão pode ser implementado utilizando um percurso em profundidade (depth-first traversal) na árvore.

Exemplo de Programa-Fonte e Respetiva AST

Considere o seguinte programa-fonte em código semelhante à linguagem apresentada:

```
! SOMA SIMPLES -
program exemplo;
var x, y, z: integer;

begin
   x := 10;
   y := 5;
   z := x + y;
end.
```

A AST gerada para esse programa possui a seguinte estrutura:

```
> Impressão da Ast - START
---> Iniciando impressão da árvore
000 [PROGRAMA]
001 ├TOKEN exemplo
001 └[CORPO]
002  ├[DECL-VAR]
003  │├[TIPO-SIMPLES]
004  ││└integer
003  │├[LI]
004  ││└x
005  ||  └[LI]
006  ||   └y
007  ||    └[LI]
008  ||     └z
002  └[LC]
003   ├[COM-ATRIBUIÇÃO]
004   │└x
005   │ ├=
005   │ └10
003   └[LC]
004    ├[COM-ATRIBUIÇÃO]
005    │└y
006    │ ├=
006    │ └5
004    └[LC]
005     └[COM-ATRIBUIÇÃO]
006      └z
007       ├=
007       └x
008        ├+
008        └y
< Impressão da Ast - END
```

Fonte: o autor

Nesse exemplo, a AST mostra a estrutura hierárquica do programa, onde cada nó representa uma construção sintática do código-fonte, como declarações e comandos de atribuição. A estrutura da AST facilita a análise semântica e a geração de código intermediário, tornando essas etapas mais organizadas e eficientes.

8. Análise de contexto

A Análise de Contexto é a fase do processo de compilação que ocorre após a Análise Sintática. Nessa etapa, o compilador realiza a verificação das informações de identificação, como declaração de variáveis e funções, e também verifica a consistência dos tipos de dados usados nas expressões e comandos.

Descrição das Estruturas de Dados e Algoritmos na Fase de Identificação

Tabela de Símbolos (Symbol Table)

- A tabela de símbolos é uma estrutura de dados usada durante a fase de identificação para armazenar informações sobre os identificadores encontrados no código-fonte (variáveis, funções, etc.).
- Cada entrada na tabela de símbolos contém informações como o nome do identificador, seu tipo, escopo e outras propriedades relevantes.
- O compilador constrói a tabela de símbolos enquanto analisa as declarações no código-fonte e realiza a verificação para evitar a declaração duplicada de identificadores no mesmo escopo.

Algoritmo de Identificação

- O algoritmo de identificação é responsável por percorrer o código-fonte e reconhecer e registrar os identificadores declarados no programa.
- Ao encontrar uma declaração de variável ou função, o algoritmo adiciona uma entrada correspondente à tabela de símbolos.
- Durante esse processo, o compilador também realiza a verificação de escopos, garantindo que os identificadores estejam sendo declarados e utilizados corretamente em seus respectivos escopos.

Descrição das Estruturas de Dados e Algoritmos na Fase de Verificação de Tipos

Verificação de Tipos

- Durante a fase de verificação de tipos, o compilador analisa as expressões e comandos do programa para garantir a consistência dos tipos de dados usados em operações e atribuições.
- O compilador deve verificar se os operandos de uma operação são compatíveis em termos de tipo, por exemplo, se é possível somar dois números inteiros ou se uma variável está sendo atribuída a um tipo compatível.

Algoritmo de Verificação de Tipos

- O algoritmo de verificação de tipos é implementado por meio de um percurso na árvore sintática abstrata (AST) do programa.
- O algoritmo analisa cada nó da AST e verifica se as operações e atribuições são consistentes em termos de tipo.
- O compilador também pode usar a tabela de símbolos para obter informações sobre o tipo declarado de variáveis e funções.

Exemplo de Entradas e Saídas

Considere o seguinte programa-fonte em código semelhante à linguagem apresentada:

Vamos considerar um exemplo bem formado:

```
program exemplo;
var x, y: integer;

begin
   x := 10;
   y := x + 5;

   if y > 15 then
      y := y - 10
   else
      y := y + 5;

end.
```

```
---> Iniciando impressão da árvore
000 [PROGRAMA]
001 ├TOKEN exemplo
001 └[CORPO]
002  ├[DECL-VAR]
003  │├[TIPO-SIMPLES]
004  ││└integer
003  │├[LI]
004  ││└x
005  ││ └[LI]
006  ||  └y
002  └[LC]
003   ├[COM-ATRIBUIÇÃO]
004   │└x
005   │ ├=
005   │ └10
003   └[LC]
004    ├[COM-ATRIBUIÇÃO]
005    │└y
006    │ ├=
006    │ └x
007    │  ├+
007    │  └5
004    └[LC]
005     └[COM-CONDICIONAL]
006      ├[EXPRESSÃO]
007      │└y
007      │└>
007      │└15
006      ├[C1]
007      │└[COM-ATRIBUIÇÃO]
008      │ └y
009      │  ├=
009      │  └y
010      │   ├-
010      │   └10
006      └[C2]
007       └[COM-ATRIBUIÇÃO]
008        └y
009         ├=
009         └y
010          ├+
010          └5
< Impressão da Ast - END

> Análise de Contexto - START
---> Iniciando identificação de nomes
< Análise de Contexto - END
```

Fonte: o autor

Neste exemplo, o programa está bem formado e possui a identificação correta dos identificadores x e y. Além disso, as operações realizadas estão usando tipos compatíveis. Portanto, o compilador passa pela Análise de Contexto sem gerar erros.

Vamos considerar um exemplo mal formado:

```
program erro;
var x, y: integer;

begin
```

```
   x := 10;
   y := "hello";
end.
```

```
> Análise Léxica - START
        !ERRO - ANÁLISE LÉXICA
          * Linha: 6, Posição: 10
          └ O caractere lido ["] (código de caractere 34) não pode ser u
        !ERRO - ANÁLISE LÉXICA
          * Linha: 6, Posição: 16
          └ O caractere lido ["] (código de caractere 34) não pode ser u
< Análise Léxica - END

> Análise Sintática - START
        !ERRO - ANÁLISE LÉXICA
          * Linha: 6, Posição: 10
          └ O caractere lido ["] (código de caractere 34) não pode ser u
        !ERRO - ANÁLISE SINTÁTICA
          * Linha: 6, Posição: 10
          └ Token lido inesperado: ["] (token do tipo 42) enquanto era e
 do tipo 1), "false" (token do tipo 2), "int-lit" (token do tipo 8), "f
 25).
        A ANÁLISE SINTÁTICA FOI INTERROMPIDA DEVIDO A ERROS OCORRIDOS
PS E:\Documentos\UNIVASF_2022_2\Compiladores\ProjetoCompilador\Projeto\
```

Fonte: o autor

Neste exemplo, o programa está mal formado porque está tentando atribuir uma string à variável y, que foi declarada como inteiro. Nesse caso, durante a Análise de Contexto, o compilador detecta o erro de tipos e gera uma mensagem de erro informando que a operação é inválida.

Esses exemplos ilustram como a Análise de Contexto é essencial para garantir que o programa está bem formado e que as operações e atribuições são feitas de acordo com as regras de tipos definidas pela linguagem.

9. Ambiente de execução

Tipo de máquina usada

O tipo de máquina utilizada pelo ambiente de execução pode variar dependendo da arquitetura do sistema no qual o compilador é executado. Para a compilação e os testes de avaliação, a arquitetura do hardware usada foi com processador baseado em x64, com 8GB de memória RAM, Processador Intel Core i5 2.30 GHz.

Avaliação de expressões

A avaliação de expressões ocorre durante a execução do programa quando são encontradas expressões aritméticas, lógicas ou relacionais. A ordem de avaliação das operações é determinada pelas regras da linguagem de programação, seguem a ordem padrão da matemática (precedência de operadores).

Representação de dados

O ambiente de execução precisa representar os diferentes tipos de dados utilizados no programa, como inteiros, reais, booleanos, strings e outros tipos definidos pelo usuário. Cada tipo de dado requer uma representação adequada em memória. Assim, inteiros são representados usando 2 ou 4 bytes, reais usando o padrão IEEE 754, booleanos podem ser representados usando 1 byte, etc.

Formas de alocação de memória empregadas

Existem várias formas de alocação de memória utilizadas pelo ambiente de execução, dependendo da linguagem e da arquitetura do sistema:

Alocação Estática: Variáveis locais e globais são alocadas em tempo de compilação e têm um endereço fixo em tempo de execução.

Alocação Dinâmica: Variáveis criadas em tempo de execução usando comandos new para Java. Essas variáveis têm endereços que são determinados em tempo de execução.

Gestão de Pilha: As variáveis locais de uma função são alocadas e desalocadas automaticamente usando a pilha de execução.

Gestão de Heap: A memória alocada dinamicamente é gerenciada pelo Garbage Collector do Java.

Passagem de parâmetros e retorno de valor de função

A forma como os parâmetros são passados e os valores de retorno de funções são tratados depende da convenção de chamada utilizada pela linguagem e pelo ambiente

de execução. As convenções de chamada determinam a forma como os argumentos são passados para as funções e como os valores de retorno são tratados.

Passagem por Valor: Uma cópia do valor do argumento é passada para a função. Qualquer alteração no parâmetro dentro da função não afeta a variável original fora da função.

Passagem por Referência: O endereço de memória da variável original é passado para a função. Isso permite que a função modifique diretamente o valor da variável fora dela.

Passagem por Ponteiro: Um ponteiro (endereço de memória) para a variável original é passado para a função. Isso é semelhante à passagem por referência, mas requer que o programador use ponteiros explicitamente.

O tratamento do valor de retorno de uma função pode seguir uma lógica semelhante, onde a função pode retornar um valor diretamente ou através de um ponteiro ou referência passada como argumento.

Em resumo, o ambiente de execução é responsável por interpretar e executar o código compilado. Ele lida com a representação dos dados, alocação de memória, passagem de parâmetros, retorno de valores e outras operações necessárias para garantir que o programa funcione corretamente em tempo de execução.

10. Linguagem-objeto

A linguagem-objeto gerada pelo compilador para a linguagem apresentada pode variar dependendo do alvo da compilação. Geralmente, a linguagem-objeto é uma representação intermediária que pode ser posteriormente traduzida para código de máquina específico ou executada por uma máquina virtual. Vamos abordar os aspectos principais da linguagem-objeto em relação à relação de instruções, sintaxe e semântica:

Relação de Instruções

A linguagem-objeto terá uma relação de instruções definidas pelo compilador para representar as operações e comandos da linguagem original em um nível mais baixo, mais próximo do código de máquina. Essas instruções podem ser do tipo:

- **Instruções de atribuição:** Para atribuir valores a variáveis.
- **Instruções aritméticas:** Para operações matemáticas como adição, subtração, multiplicação, etc.
- **Instruções lógicas e relacionais:** Para operações booleanas como AND, OR, igualdade, etc.
- **Instruções de controle de fluxo:** Para comandos condicionais (if-else) e loops (while, for).
- **Instruções de acesso a memória:** Para acessar variáveis e valores armazenados na memória.
- **Instruções de chamada e retorno de função:** Para chamar e retornar de funções e procedimentos.

Sintaxe

A sintaxe da linguagem-objeto é definida pelo compilador e pode ser diferente da linguagem original. Ela será mais próxima do nível de máquina e, portanto, pode ser menos legível para os programadores. A sintaxe da linguagem-objeto é projetada de forma que o código gerado seja executado corretamente pela plataforma ou máquina virtual de destino.

Semântica

A semântica da linguagem-objeto deve ser consistente com a semântica da linguagem original. Isso significa que o comportamento do código gerado é o mesmo que o comportamento do código-fonte na linguagem original.

Por exemplo, se na linguagem original o operador de adição realiza a soma de dois números inteiros, a instrução gerada pela linguagem-objeto também deve realizar essa soma de forma equivalente.

Além disso, a semântica deve garantir que as regras da linguagem original, como o escopo de variáveis, o tratamento de tipos e as regras de fluxo de controle, sejam preservadas na linguagem-objeto.

Considerações Adicionais

A linguagem-objeto é geralmente uma representação intermediária e pode ser traduzida para código de máquina nativo ou executada por uma máquina virtual específica, como JVM ou CLR.

O código gerado pelo compilador deve ser eficiente e otimizado para garantir o melhor desempenho possível.

O compilador deve realizar a verificação semântica durante a geração do código objeto para garantir a consistência e a correção do código gerado.

O código objeto é uma etapa intermediária no processo de compilação, e ele ainda precisa ser transformado em código executável ou interpretado pela máquina virtual antes de ser executado pelo computador ou dispositivo.

11. Geração de código

Devido a erros no processo de programação do compilador, não foi possível implementar a geração de código de forma satisfatória.

12. Manual de instalação, compilação e utilização

Para o desenvolvimento do projeto foram utilizados os seguintes recursos:

IDE: Apache NetBeans 17, disponível em:
https://archive.apache.org/dist/netbeans/netbeans-installers/17/Apache-NetBeans-17-bin-windows-x64.exe

Sistema operacional: Windows 10 de 64 bits.
Terminal de comandos: PowerShell (Windows)
Pacote Java: Java versão "17.0.6" 2023-01-17 LTS;
Java(TM) SE Runtime Environment (build 17.0.6+9-LTS-190);
Java HotSpot(TM) 64-Bit Server VM (build 17.0.6+9-LTS-190), disponível em:

jdk-8u202-windows-x64
https://www.oracle.com/br/java/technologies/javase/javase8-archive-downloads.html#license-lightbox
jdk-17_windows-x64_bin
https://download.oracle.com/java/17/archive/jdk-17.0.6_windows-x64_bin.exe

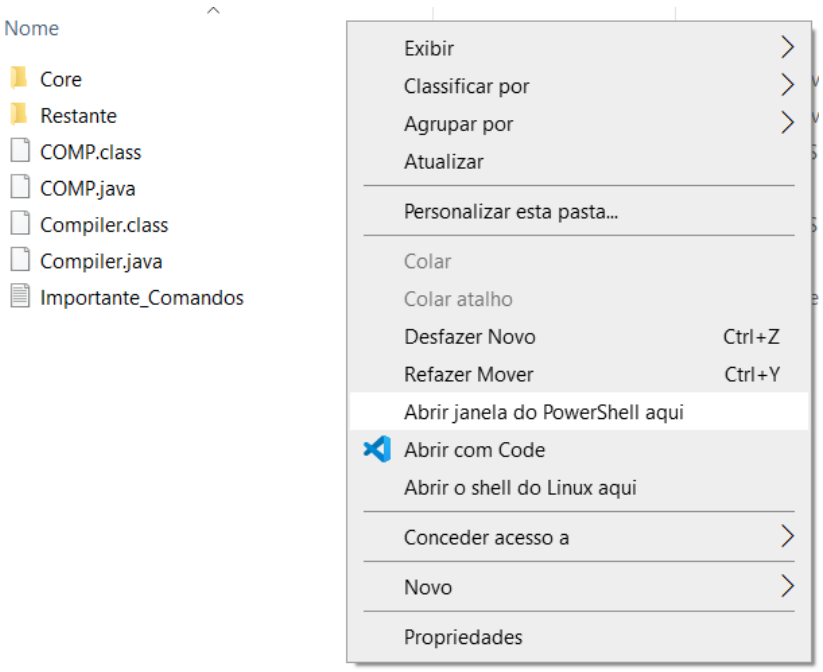
Abrir o projeto na IDE

1. Para acessar o Projeto, é necessário abri-lo no NetBeans.
2. Inicialmente, é preciso descompactar o arquivo chamado "Projeto.rar".
3. Na pasta "Projeto" é possível encontrar duas pasta nomeadas de "Exemplos_de_teste" e "ProjetoCompilador", a primeira contém exemplos de teste que foram feitos com o compilador e a segunda contém o projeto e todos os arquivos pertencentes a ele.
4. Na IDE, Abrir projeto → Procurar a pasta onde o projeto foi descompactado → Selecionar arquivo/pasta com o nome "ProjetoCompilador" → Abrir projeto.

Compilação

1. Com o arquivo "Projeto.rar" descompactado, é preciso localizar o arquivo chamado "COMP.java", que está na pasta "src".
   Exemplo: "...\Projeto\ProjetoCompilador\src\src\COMP.java"
2. Com o Terminal de Comandos aberto, digite, sem as aspas:
   "javac -encoding UTF8 COMP.java"
   Esse comando é responsável pela compilação.
3. Em seguida, digite, sem as aspas:
   "java COMP.java"
   Esse comando é responsável pela execução.
4. Agora é possível ver as opções disponíveis do compilador.

Dica: É possível abrir o Terminal de Comandos ou PowerShell, já com referência para a pasta atual, segurando na tecla Shift + um click do botão direito do mouse em um espaço em branco da pasta.



Fonte: o autor

Tela do Terminal de Comandos após compilação e execução do arquivo “COMP.java”:

```
----- BEM VINDO AO COMPILADOR -----


Opcoes de compilacao disponiveis:

         -l      Analise lexica.
         -s      Analise sintatica.
         -a      Impressao da AST.
         -c      Analise de contexto.

Para executar a compilacao e necessario digitar o comando "java COMP",
seguido da opcao de compilacao e o caminho completo
para o arquivo de codigo-fonte desejado.

O caminho para o arquivo deve ser escrito sem as aspas duplas.
Como exemplo da Analise lexica abaixo:

         java COMP -l "C:\Users\User\Documents\codigo-fonte.txt"
```

Fonte: o autor

Utilização

1. A pasta “Exemplos_de_teste” contém arquivos de teste que podem ser replicados.
2. Depois de compilar e executar o compilador basta digitar, no Terminal de Comandos, "java COMP", seguido da opção de compilação, e o caminho completo para o arquivo de código-fonte desejado. O caminho para o arquivo deve ser escrito sem as aspas duplas. Como no exemplo abaixo:

   java COMP –a E:\Documentos\...\Projeto\Exemplos_de_teste\Exemplo0_ok.txt

13. Conclusões finais

Diante das adversidades e desafios para se construir novos saberes e conhecimentos, a exposição a novos objetos de estudo é sempre uma oportunidade de crescimento.
Este projeto forneceu uma visão abrangente sobre a linguagem de programação, o processo de compilação e o ambiente de execução. Compreender esses conceitos é essencial para o desenvolvimento de programas eficientes e corretos, e para aprofundar-se no mundo da programação.