Inatel

S201 – Paradigmas de Programação

DESCREVENDO SINTAXE E SEMÂNTICA

Marcelo Vinícius Cysneiros Aragão

marcelovca90@inatel.br

Introdução

- A **sintaxe** de uma linguagem de programação é a forma de suas expressões, sentenças e unidades de programas.
- Sua **semântica** é o significado dessas expressões, sentenças e unidades de programas.
- Em uma linguagem de programação bem projetada, a semântica deve seguir diretamente a partir da sintaxe; ou seja, a aparência de uma sentença deve sugerir o que a sentença realiza.

Introdução

- Exemplo: a sintaxe de uma sentença **while** em Java é

```
while (expressão_booleana) sentença
```

- A semântica desse formato de sentença é:

*"Quando o valor atual da expressão booleana for verdadeiro, a sentença dentro da estrutura é executada. Caso contrário, o controle continua após a construção **while**. O controle retorna implicitamente para a expressão booleana para repetir o processo."*

Introdução

- Descrever a sintaxe é mais fácil do que descrever a semântica, especialmente porque uma notação aceita universalmente está disponível para a descrição de sintaxe, mas nenhuma ainda foi desenvolvida para descrever semântica.

Introdução

- Terminologia:
  - **Sentenças:** conjunto de cadeias de caracteres de um alfabeto.
  - **Linguagem** : conjunto de sentenças.
  - **Lexemas:** unidades sintáticas de mais baixo nível de uma linguagem (exemplo: `*`, `soma`, `início`).
  - **Token:** uma categoria de lexemas (exemplo: identificador).

Introdução

- Considere a seguinte sentença Java:

```
index = 2 * count + 17;
```

- Os lexemas e *tokens* dessa sentença são:

Lexemas	*Tokens*
`index`	identificador
`=`	sinal_de_igualdade
`2`	literal_inteiro
`*`	operador_de_multiplicação
`count`	identificador
`+`	operador_de_adição
`17`	literal_inteiro
`;`	ponto e vírgula

DESCREVENDO A SINTAXE:

GLC – GRAMÁTICA LIVRE DE CONTEXTO,
BNF – FORMALISMO DE BACKUS-NAUR &
EBNF – FORMALISMO DE BACKUS-NAUR ESTENDIDO

GLC, BNF & EBNF

- No meio dos anos 1950, Noam Chomsky e John Backus, em esforços de pesquisa não relacionados, desenvolveram o mesmo formalismo de descrição de sintaxe.

- Tal formalismo se tornou o método mais usado para descrever a sintaxe de linguagens de programação.

GLC, BNF & EBNF

- Na segunda metade da década de 1950, Chomsky descreveu classes de gramáticas e de linguagens, que acabaram sendo úteis para descrever a sintaxe de linguagens de programação, dentre as quais:
  - Gramáticas livres de contexto: sintaxe de uma linguagens de programação completa, com pequenas exceções.
  - Expressões regulares: forma dos tokens das linguagens de programação.

GLC, BNF & EBNF

- Logo após o trabalho de Chomsky em classes de linguagens, o grupo ACM-GAMM iniciou o projeto do ALGOL 58.
- Um artigo descritivo foi apresentado por John Backus em uma conferência internacional em 1959.

- Era uma nova notação formal para especificar a sintaxe de linguagens de programação, modificada por Peter Naur para descrever o ALGOL 60.
- O método revisado de descrição de sintaxe ficou conhecido como **Forma de Backus-Naur**, ou simplesmente **BNF**.

GLC, BNF & EBNF

- BNF é uma notação natural para descrever sintaxe.
- Algo similar à BNF era usada por Panini para descrever a sintaxe de sânscrito séculos A.C. .
- A BNF não foi imediatamente aceita pelos usuários de computadores,
- Rapidamente se tornou o método mais popular para descrever a sintaxe de linguagens.
- É notável que a BNF seja praticamente idêntica às **gramáticas livres de contexto**.

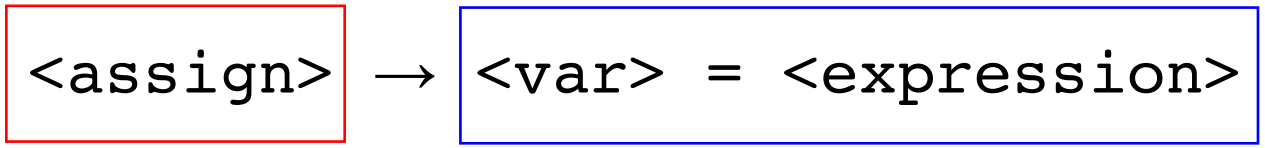
GLC, BNF & EBNF

- Uma **metalinguagem** é uma linguagem usada para descrever outra.
- BNF é uma metalinguagem para linguagens de programação.
- BNF usa abstrações para estruturas sintáticas.

```
<assign> → <var> = <expression>
```

Abstração (também chamada **regra** ou **produção)**

GLC, BNF & EBNF



LHS (*left-hand side*)
abstração que está sendo definida

RHS (*right-hand side*)
misto de *tokens*, lexemas e referências a outras abstrações

- Essa regra em particular especifica que a abstração <assign> é definida como uma instância da abstração <var>, seguida pelo lexema =, seguido por uma instância da abstração <expression>
- Uma sentença de exemplo cuja estrutura sintática é descrita por essa regra é

```
total = subtotal1 + subtotal2
```

GLC, BNF & EBNF

- As abstrações em uma descrição de uma BNF, ou gramática, são chamadas de **símbolos não terminais**, ou simplesmente **não terminais.**
- Lexemas e *tokens* das regras são chamados de **símbolos terminais**, ou simplesmente **terminais**.
- Uma descrição BNF, ou **gramática**, é uma coleção de regras.
- Símbolos não terminais podem ter duas ou mais definições, representando duas ou mais formas sintáticas possíveis na linguagem .
- Múltiplas definições podem ser escritas como uma única regra, separadas pelo símbolo |, que significa um OU lógico.

GLC, BNF & EBNF

- Por exemplo, uma sentença **if** em Java pode ser descrita com as regras

```
<if_stmt> → if (<logic_expr>) <stmt>
<if_stmt> → if (<logic_expr>) <stmt> else <stmt>
```

- ou com a regra

```
<if_stmt> → if (<logic_expr>) <stmt>
          | if (<logic_expr>) <stmt> else <stmt>
```

- Nessas regras, <stmt> representa ou uma sentença única ou uma sentença composta.

GLC, BNF & EBNF

- Apesar de a BNF ser simples, é poderosa o suficiente para descrever praticamente toda a sintaxe das linguagens de programação.
- É capaz de expressar:
  - listas de construções similares;
  - ordem pela qual as diferentes construções devem aparecer,
  - estruturas aninhadas a qualquer profundidade;
  - precedência e associatividade de operadores.

GLC, BNF & EBNF

- Listas de tamanho variável são geralmente escritas usando uma elipse (...);
- BNF não inclui elipses, então a alternativa é a recursão.
- Uma regra é recursiva se seu lado esquerdo aparecer em seu lado direito.
- A seguinte regra ilustra como a recursão é usada para descrever listas:

```
<ident_list> → identifier
             | identifier , <ident_list>
```

- Essa regra define uma lista de identificadores (<ident_list>) como um *token* isolado (identificador) ou um identificador seguido por uma vírgula e outra instância de <ident_list>.

GLC, BNF & EBNF

- Uma gramática é um dispositivo de geração para definir linguagens.
  - As sentenças da linguagem são geradas por meio de uma sequência de aplicação de regras, começando com um não terminal especial chamado de **símbolo inicial**.
  - A geração de uma sentença é chamada de **derivação**.
- Em uma gramática para uma linguagem completa, o símbolo inicial representa um programa completo e é chamado de <program>.

GLC, BNF & EBNF

- Ao lado, um exemplo de uma gramática para uma pequena linguagem.
- A linguagem escrita por esta gramática tem apenas uma forma sentencial: atribuição.

<program> → **begin** <stmt_list> **end**
<stmt_list> → <stmt>
| <stmt> ; <stmt_list>
<stmt> → <var> = <expression>
<var> → A | B | C
<expression> → <var> + <var>
| <var> – <var>
| <var>

GLC, BNF & EBNF

- Um programa consiste na palavra especial **begin**, seguida por uma lista de sentenças separadas por ponto e vírgula, seguida da palavra especial **end**.
- Uma expressão é uma única variável ou duas variáveis separadas por um operador + ou -.

<program> → **begin** <stmt_list> **end**
<stmt_list> → <stmt>
| <stmt> ; <stmt_list>
<stmt> → <var> = <expression>
<var> → A | B | C
<expression> → <var> + <var>
| <var> – <var>
| <var>

- Os únicos nomes de variáveis nessa linguagem são A, B e C.

GLC, BNF & EBNF

- Exemplo de uma derivação de um programa nessa linguagem:

<program> => **begin** <stmt_list> **end**
=> **begin** <stmt> ; <stmt_list> **end**
=> **begin** <var> = <expression> ; <stmt_list> **end**
=> **begin** A = <expression> ; <stmt_list> **end**
=> **begin** A = <var> + <var> ; <stmt_list> **end**
=> **begin** A = B + <var> ; <stmt_list> **end**
=> **begin** A = B + C ; <stmt_list> **end**
=> **begin** A = B + C ; <stmt> **end**
=> **begin** A = B + C ; <var> = <expression> **end**
=> **begin** A = B + C ; B = <expression> **end**
=> **begin** A = B + C ; B = <var> **end**
=> **begin** A = B + C ; B = C **end**

GLC, BNF & EBNF

- Essa derivação, como todas, começa com o símbolo inicial, aqui <program>.
- O símbolo => é lido como “deriva”.
- Cada cadeia sucessiva na sequência é derivada da cadeia anterior, substituindo um dos não terminais por uma das definições de não terminais.
- Cada uma das cadeias na derivação é chamada de **forma sentencial**.

```
<program> => begin <stmt_list> end
          => begin <stmt> ; <stmt_list> end
          => begin <var> = <expression> ; <stmt_list> end
          => begin A = <expression> ; <stmt_list> end
          => begin A = <var> + <var> ; <stmt_list> end
          => begin A = B + <var> ; <stmt_list> end
          => begin A = B + C ; <stmt_list> end
          => begin A = B + C ; <stmt> end
          => begin A = B + C ; <var> = <expression> end
          => begin A = B + C ; B = <expression> end
          => begin A = B + C ; B = <var> end
          => begin A = B + C ; B = C end
```

GLC, BNF & EBNF

- Nessa derivação, o não terminal substituído é sempre o mais à esquerda na forma sentencial anterior.
- É chamada de **derivação mais à esquerda**.
- Ela continua até que a forma sentencial não contenha mais nenhum não terminal.
- Essa forma sentencial, consistindo em apenas terminais, ou lexemas, é a sentença gerada.

```
<program> => begin <stmt_list> end
          => begin <stmt> ; <stmt_list> end
          => begin <var> = <expression> ; <stmt_list> end
          => begin A = <expression> ; <stmt_list> end
          => begin A = <var> + <var> ; <stmt_list> end
          => begin A = B + <var> ; <stmt_list> end
          => begin A = B + C ; <stmt_list> end
          => begin A = B + C ; <stmt> end
          => begin A = B + C ; <var> = <expression> end
          => begin A = B + C ; B = <expression> end
          => begin A = B + C ; B = <var> end
          => begin A = B + C ; B = C end
```

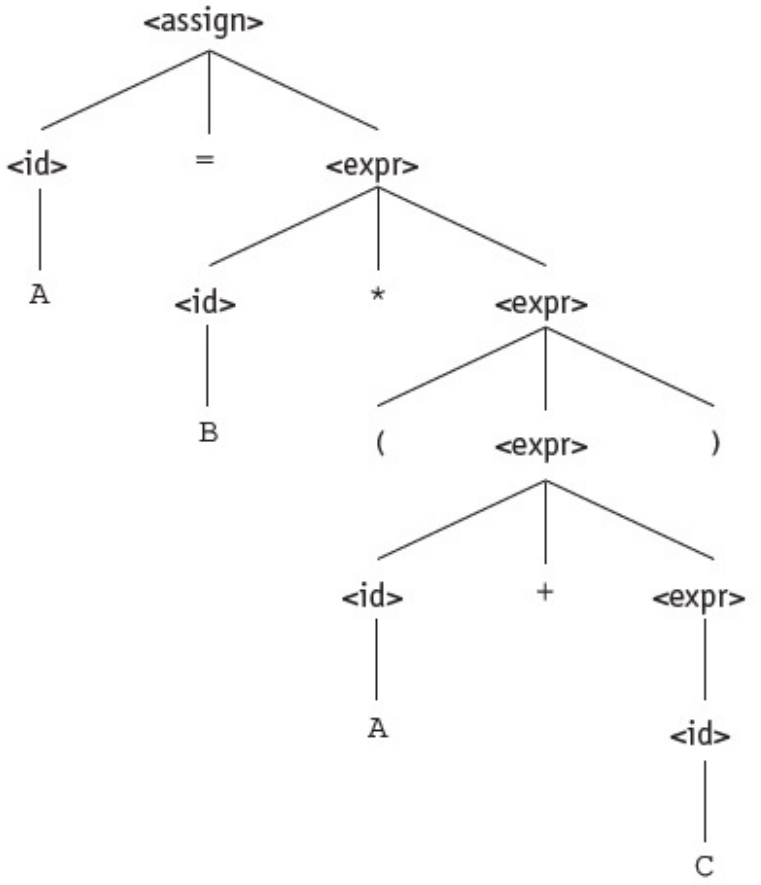
GLC, BNF & EBNF

<program> → **begin** <stmt_list> **end**
<stmt_list> → <stmt>
| <stmt> ; <stmt_list>
<stmt> → <var> = <expression>
<var> → A | B | C
<expression> → <var> + <var>
| <var> – <var>
| <var>

<program> => **begin** <stmt_list> **end**
=> **begin** <stmt> ; <stmt_list> **end**
=> **begin** <var> = <expression> ; <stmt_list> **end**
=> **begin** A = <expression> ; <stmt_list> **end**
=> **begin** A = <var> + <var> ; <stmt_list> **end**
=> **begin** A = B + <var> ; <stmt_list> **end**
=> **begin** A = B + C ; <stmt_list> **end**
=> **begin** A = B + C ; <stmt> **end**
=> **begin** A = B + C ; <var> = <expression> **end**
=> **begin** A = B + C ; B = <expression> **end**
=> **begin** A = B + C ; B = <var> **end**
=> **begin** A = B + C ; B = C **end**

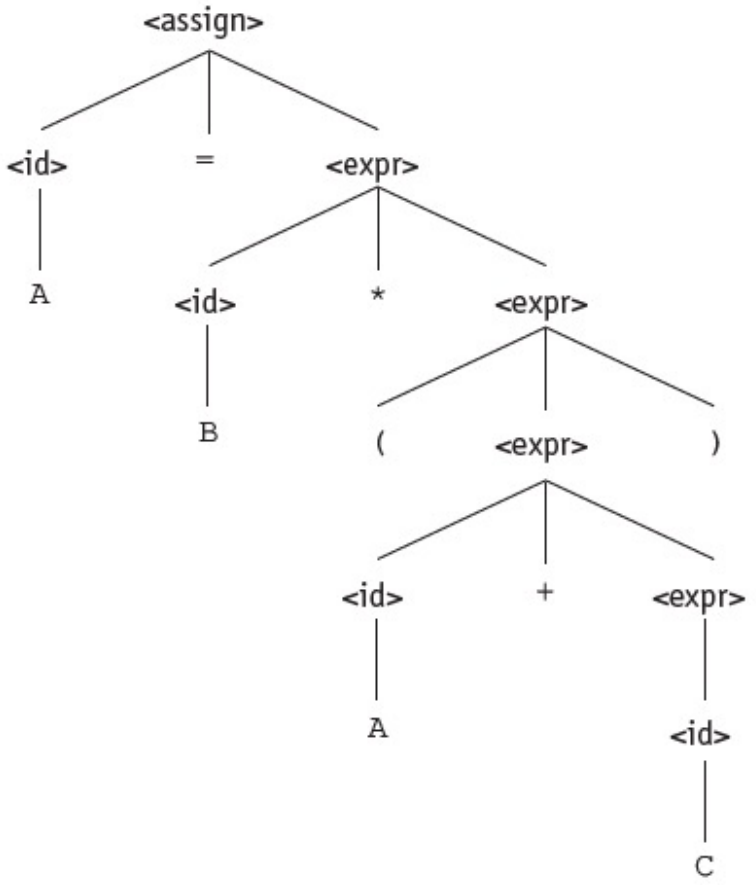
GLC, BNF & EBNF

- Um recurso relevante das gramáticas é que elas naturalmente descrevem a estrutura hierárquica das sentenças das linguagens que definem.
- Essas estruturas são chamadas de árvores de análise sintática (*parse trees*).
- À direita é exibida uma árvore de análise sintática para a sentença simples A = B * (A + C).



GLC, BNF & EBNF

- Cada nó interno de uma árvore de análise sintática é rotulado com um símbolo não terminal; cada folha é rotulada com um símbolo terminal.
- Cada subárvore descreve uma instância de uma abstração da sentença.



GLC, BNF & EBNF

- Por causa de algumas pequenas inconveniências na BNF, ela tem sido estendida de diversas maneiras.
  - ABNF: https://tools.ietf.org/pdf/rfc5234.pdf
  - EBNF: http://standards.iso.org/ittf/PubliclyAvailableStandards/s026153_ISO_IEC_14977_1996(E).zip

- Três extensões são comumente incluídas na EBNF:
  - Parte opcional à direita (entre colchetes);
  - Parte repetível/opcional à direita (entre chaves);
  - Opções de múltipla escolha.

GLC, BNF & EBNF

- A primeira denota uma parte opcional de um lado direito, entre colchetes.
- Por exemplo, uma sentença **if-else** em C pode ser descrita como:

```
<if_stmt> → if (<expression>) <statement> [else <statement>]
```

- Sem o uso dos colchetes, a descrição sintática dessa sentença necessitaria das duas regras a seguir:

```
<if_stmt> → if (<expression>) <statement>
          | if (<expression>) <statement> else <statement>
```

GLC, BNF & EBNF

- A segunda denota o uso de chaves em um lado direito para indicar que a parte envolta pode ser repetida indefinidamente ou deixada de fora.
- Listas podem ser construídas com uma regra, em vez de duas regras e recursão.
- Exemplo: uma lista de identificadores separados por vírgulas pode ser descrita pela seguinte regra:

<ident_list> → <identifier> {, <identifier>}

- Essa é uma substituição ao uso de recursão por meio de uma iteração implícita; a parte envolta em chaves pode ser iterada qualquer número de vezes.

GLC, BNF & EBNF

- A terceira extensão comum trata de opções de múltipla escolha.
- Quando um elemento deve ser escolhido de um grupo, as opções são colocadas em parênteses e separadas pelo operador OU, |.
- Exemplo em BNF:

```
<term> → <term> * <factor>
       | <term> / <factor>
       | <term> % <factor>
```

- Exemplo em EBNF:

```
<term> → <term> (*|/|%) <factor>
```

GLC, BNF & EBNF

- Os colchetes, chaves e parênteses nas extensões EBNF são **metassímbolos**, ou seja, são ferramentas notacionais e não símbolos terminais nas entidades sintáticas que eles ajudam a descrever.
- Quando esses metassímbolos também são símbolos terminais na linguagem sendo descrita, as instâncias que são símbolos terminais devem ser destacadas (sublinhadas ou colocadas entre aspas).

DESCREVENDO A SEMÂNTICA:

SEMÂNTICA ESTÁTICA
SEMÂNTICA DINÂMICA

Semântica Estática

- Há características da estrutura das linguagens de programação que são difíceis de descrever com BNF – e algumas impossíveis.
- Como um exemplo de uma regra sintática difícil de especificar com uma BNF, considere as regras de compatibilidade de tipos.
- Neste caso, serão utilizados trechos de código na linguagem Java.

Semântica Estática

- Exemplo: um valor de ponto flutuante não pode ser atribuído a uma variável do tipo inteiro, apesar de o contrário ser permitido.

```
class Example1 {
    public static void main(String[] args) {
        int value = 42.0;
        System.out.println("Bye bye!");
    }
}
```

```
$ javac Example1.java
Example.java:3: error: incompatible types: possible lossy conversion from double to int
int value = 42.0;
            ^
1 error
```

Semântica Estática

- Exemplo: um valor de ponto flutuante não pode ser atribuído a uma variável do tipo inteiro, apesar de o contrário ser permitido.

```
class Example1 {
    public static void main(String[] args) {
        double value = 42;
        System.out.println("Bye bye!");
    }
}

$ javac Example1.java
<ok>

$ java Example1
Bye bye!
```

Semântica Estática

- Apesar de essa restrição poder ser especificada em BNF, ela requer símbolos não terminais e regras adicionais.
- Se todas as regras de tipos em Java fossem especificadas em BNF, a gramática se tornaria muito grande para ser útil, porque seu tamanho determina o tamanho do analisador sintático.
  - https://docs.oracle.com/javase/specs/jls/se7/html/jls-18.html

Semântica Estática

- Um exemplo mais radical é a regra de que todas as variáveis devem ser declaradas antes de serem usadas.
- Foi provado que essa regra não pode ser especificada em BNF.
- Tais problemas exemplificam as categorias de regras de linguagem chamadas de **regras semânticas**.

Semântica Estática

- Exemplo em linguagem Java de uso de variável sem declaração prévia.

```
class Example2 {
    public static void main(String[] args) {
        value = 42;
    }
}

$ javac Example2.java
Example2.java:3: error: cannot find symbol
System.out.println(value);
                   ^
  symbol:   variable value
  location: class Example2
1 error
```

Semântica Estática

- Exemplo em linguagem Java de uso de variável com declaração prévia.

```
class Example2 {
    public static void main(String[] args) {
        int value;
        value = 42;
        System.out.println(”Value = ” + value);
    }
}

$ javac Example2.java
<ok>

$ java Example2
42
```

Semântica Estática

- A **semântica estática** é apenas indiretamente relacionada ao significado dos programas durante a execução; ela tem mais a ver com as formas permitidas dos programas (sintaxe em vez da semântica).
- Muitas regras de semântica estática definem restrições de tipo.
- É dita estática porque a análise necessária para verificar essas especificações pode ser feita em tempo de compilação.

Semântica Estática

- Por causa dos problemas da descrição de semântica estática com BNF, mecanismos mais poderosos foram criados para essa tarefa.
- As **gramáticas de atributos** foram projetadas por Donald Knuth em 1968 para descrever tanto a sintaxe quanto a semântica estática de programas.
  - http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.455.1434
  - São uma abordagem formal, usada tanto para descrever quanto para verificar a corretude das regras de semântica estática de um programa.
  - Seus conceitos básicos são amplamente usados em todos os compiladores.

Semântica Estática

- Exemplo 1: em um procedimento em Ada, o nome no ***end*** deve ser igual ao nome do procedimento.

Regra sintática: <proc_def> → **`procedure`** <proc_name>[1]
<proc_body> **`end`** <proc_name>[2];
Predicado: <proc_name>[1].string == <proc_name>[2].string

- Esta regra não pode ser expressa com BNF.

Semântica Estática

- Exemplo 2: verificação das regras de tipos de uma instrução de atribuição simples.

<assign> → <var> = <expr>
<expr> → <var> + <var>
| <var>
<var> → A | B | C

- Observações:
  - As variáveis podem ser de dois tipos: *int* ou *real*;
  - O tipo do lado esquerda expressão deve ser o mesmo do lado direito;
  - O tipo da expressão quando os tipos dos operandos são diferentes é *real.*

Semântica Estática: Exemplos

1. Regra sintática: <assign> → <var> = <expr>
   Regra semântica: <expr>.expected_type ← <var>.actual_type
2. Regra sintática: <expr> → <var>[2] + <var>[3]
   Regra semântica: <expr>.actual_type ←
   if (<var>[2].actual_type = int) and
   (<var>[3].actual_type = int)
   then int
   else real
   end if
   Predicado: <expr>.actual_type == <expr>.expected_type
3. Regra sintática: <expr> → <var>
   Regra semântica: <expr>.actual_type ← <var>.actual_type
   Predicado: <expr>.actual_type == <expr>.expected_type
4. Regra sintática: <var> → A | B | C
   Regra semântica: <var>.actual_type ← look-up(<var>.string)

A função `lookup` busca um dado nome de variável na tabela de símbolos e retorna o tipo de dessa variável.

Semântica Dinâmica

- A **semântica dinâmica** consiste no significado das expressões, sentenças e unidades de programa de uma linguagem de programação.
- Por causa do poder e da naturalidade da notação disponível, descrever a sintaxe é algo relativamente simples.
- Por outro lado, nenhuma notação ou abordagem universalmente aceita foi inventada para semântica dinâmica.

Semântica Dinâmica

- Por que criar uma metodologia e notação para descrever semântica?

- Os programadores obviamente precisam saber exatamente o que a sentenças de uma linguagem fazem antes de usá-la em seus programas.

- Desenvolvedores de compiladores devem saber o que as construções da linguagem significam para projetar implementações corretas para elas.

Semântica Dinâmica

- Se existisse uma especificação precisa de semântica de uma linguagem de programação, os programas escritos na linguagem poderiam, potencialmente, ser provados corretos sem a necessidade de testes.

- Além disso, os compiladores poderiam produzir programas que exibissem o comportamento dado na definição da linguagem; ou seja, sua corretude poderia ser verificada.

Semântica Dinâmica

- Uma especificação completa da sintaxe e da semântica da linguagem de programação poderia ser usada por uma ferramenta para gerar automaticamente um compilador para a linguagem.

- Por fim, os projetistas de linguagens, que desenvolveriam as descrições semânticas de suas linguagens, poderiam descobrir ambiguidades e inconsistências em seus projetos durante esse processo.

Semântica Dinâmica

- Os desenvolvedores de software e os projetistas de compiladores normalmente determinam a semântica das linguagens de programação pela leitura de explicações em linguagem natural disponíveis nos manuais da linguagem.
  - https://docs.oracle.com/javase/8/
  - https://docs.oracle.com/en/java/javase/16/
  - https://docs.python.org/2/
  - https://docs.python.org/3/
- Como são explicações normalmente imprecisas e incompletas, essa abordagem é claramente insatisfatória.

Semântica Dinâmica

- Em decorrência da falta de especificações semânticas completas de linguagens de programação:
  - Os programas são raramente provados corretos sem testes;
  - Os compiladores comerciais nunca são gerados automaticamente a partir de descrições de linguagem.
- Scheme é uma das poucas linguagens de programação cuja definição inclui uma descrição semântica formal.

Semântica Dinâmica

- Há três abordagens para descrição de semântica dinâmica:

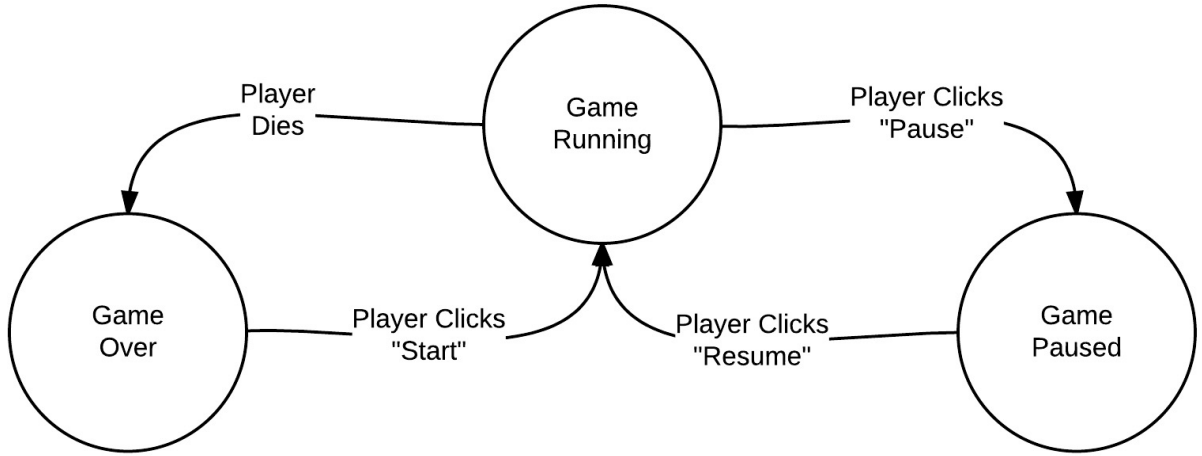
Operacional

Denotacional

Axiomática

Semântica Operacional

- Descreve o significado de um programa ao executar suas instruções em uma máquina, real ou simulada.
- As alterações que ocorrem no estado de uma máquina, quando determinada instrução é executada, define o significado desta.
- O estado de um computador é definido pelos valores de todos os seus registradores e de suas localizações de memória.



Semântica Operacional

- Processo básico:
  - Projetar uma linguagem intermediária cuja característica primária é a clareza;
  - Projetar uma máquina virtual (interpretador) para a linguagem intermediária ou inspecionar o código intermediário manualmente.

Sentença C

```
for (expr1; expr2; expr3) {
  . . .
}
```

Significado

```
        expr1;
loop:   if expr2 == 0 goto out
        . . .
        expr3;
        goto loop
out:    . . .
```

Semântica Denotacional

- Baseia-se na teoria de funções recursivas.
- Para cada entidade da linguagem, deve ser definido tanto um objeto matemático como uma função que relacione instâncias daquela entidade com as deste.
- Os objetos matemáticos denotam o significado de suas entidades sintáticas correspondentes.
- A dificuldade no uso deste método está em criar os objetos e as funções de correspondências.

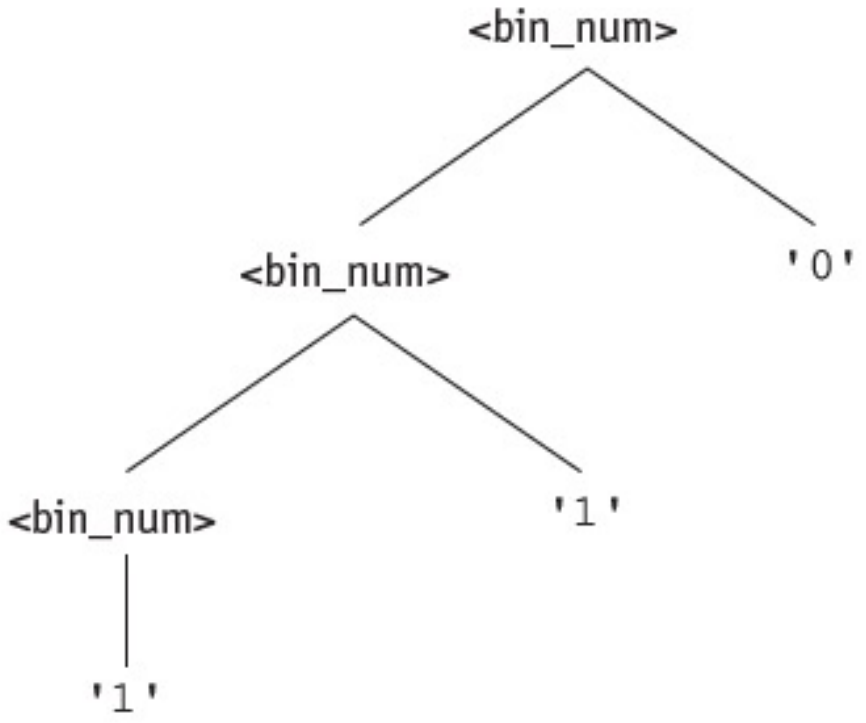
Semântica Denotacional

- Exemplo: considere uma construção de linguagem muito simples, a representação em cadeias de caracteres de números binários.
- A sintaxe desses números binários pode ser descrita pelas seguintes regras gramaticais:

```
<bin_num> → '0'
          | '1'
          | <bin_num> '0'
          | <bin_num> '1'
```

Semântica Denotacional

- Árvore de análise sintática para o número binário de exemplo 110



Semântica Denotacional

- Domínio sintático da função de mapeamento para números binários:
  - Conjunto de todas as representações em cadeias de caracteres dos números binários.
  - Exemplo: ‘0’, ‘1’, ‘010’, ‘111’, ‘101001’, ‘0101011101’, etc.
- Domínio semântico:
  - Conjunto dos números decimais não negativos N.
  - Exemplo: 0, 12, 305, 2064, 91857, 1027401, etc.

Semântica Denotacional

- No exemplo, os números decimais devem estar associados com as duas primeiras regras gramaticais.

<bin_num> → '0'
| '1'
| <bin_num> '0'
| <bin_num> '1'

Semântica Denotacional

- As outras duas são, em um sentido, regras computacionais, porque combinam um símbolo terminal (com o qual um objeto pode ser associado) com um não terminal (para o qual pode se esperar que represente alguma construção).

```
<bin_num> → '0'
          | '1'
          | <bin_num> '0'
          | <bin_num> '1'
```

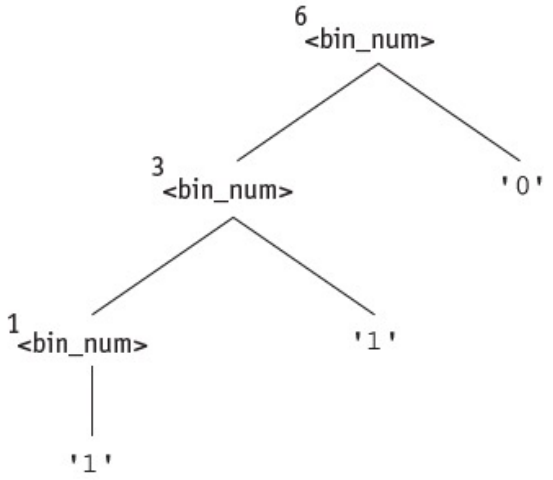
Semântica Denotacional

- A função semântica, chamada M_{bin}, mapeia os objetos sintáticos, conforme descrito nas regras gramaticais anteriores, para os objetos em N, o conjunto de números decimais não negativos.

$$M_{bin}(\texttt{'0'}) = 0$$
$$M_{bin}(\texttt{'1'}) = 1$$
$$M_{bin}(\texttt{<bin\_num> '0'}) = 2 * M_{bin}(\texttt{<bin\_num>})$$
$$M_{bin}(\texttt{<bin\_num> '1'}) = 2 * M_{bin}(\texttt{<bin\_num>}) + 1$$

Semântica Denotacional

- Os significados, ou objetos denotados (que, nesse caso, são números decimais), podem ser anexados aos nós da árvore de análise sintática, resultando em:



- Isso é uma semântica dirigida por sintaxe.
- Entidades sintáticas são mapeadas para objetos matemáticos com significado concreto.

Semântica Axiomática

- Método para provar a exatidão dos programas que mostra a computação descrita por sua especificação.
- Cada instrução de um programa é tanto precedida como seguida de uma expressão lógica que especifica restrições a variáveis.
- As expressões lógicas especificam o significado das instruções.
- As restrições são descritas pela notação da lógica de predicados.

Semântica Axiomática

Inatel

M020 – Matemática Discreta

1. LÓGICA FORMAL

1.6 Demonstração de Correção

Marcelo Vinícius Cysneiros Aragão

marcelovca90@inatel.br

RESUMO

Resumo

A Forma de Backus-Naur e as gramáticas livres de contexto são metalinguagens equivalentes bastante adequadas para a tarefa de descrever a sintaxe de linguagens de programação. Não apenas são métodos descritivos concisos, mas as árvores de análise sintática que podem ser associadas com suas ações de geração dão uma evidência gráfica das estruturas sintáticas subjacentes. Além disso, elas são naturalmente relacionadas aos dispositivos de reconhecimento para as linguagens que geram, o que leva à construção relativamente fácil de analisadores sintáticos para compiladores para essas linguagens.

Uma gramática de atributos é um formalismo descritivo que pode descrever tanto a sintaxe quanto a semântica estática de uma linguagem. Gramáticas de atributos são extensões de gramáticas livres de contexto. Uma gramática de atributos consiste em uma gramática, um conjunto de atributos, um conjunto de funções de computação de atributos e um conjunto de predicados, os quais descrevem as regras de semântica estática.

Existem três métodos principais de descrição semântica: operacional, denotacional e axiomática. A semântica operacional é um método para a descrição do significado das construções de uma linguagem em termos de seus efeitos em uma máquina ideal. Na semântica denotacional, objetos matemáticos são usados para representar o significado das construções da linguagem. Entidades de linguagem são convertidas para esses objetos matemáticos por meio de funções recursivas. Semântica axiomática, baseada em lógica formal, for criada como uma ferramenta para provar a corretude de programas.

Referência Bibliográfica

- SEBESTA, Robert W.; SANTOS, José Carlos Barbosa dos; TORTELLO, João Eduardo Nóbrega, Conceitos de linguagens de programação. 11 ed. Porto Alegre, RS: Editora Bookman, 2018, 758 p. ISBN 978-85-8260-468-7.