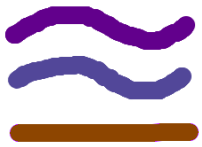
**Disciplina Compiladores - UFS**
**Turma T01 2023.2**
**Profa. Beatriz Trinchão Andrade**



# Especificação da Linguagem  Cida

## 1. Introdução

Já pensaram em como seriam as linguagens de programação se certas escolhas fossem feitas de forma diferente? Se, por exemplo, ao especificar o que seria o primeiro comando condicional, ao invés de usar a palavra *if* alguém pensasse em outra palavra no momento e isso se tornasse o padrão?

Tendo esse pensamento em mente, vamos implementar uma linguagem cujas expressões e palavras reservadas são levemente diferentes das que costumamos usar na maioria das linguagens de programação. A linguagem **Cida** é uma linguagem imperativa, e apresenta as características descritas neste documento.

**Cida**, obviamente, é uma linguagem experimental, então esta especificação é passível de adaptações. Em caso de modificações na especificação, versões atualizadas serão postadas via SIGAA (no tópico de aula “Definições do Projeto”) e notificações serão enviadas aos alunos e alunas.

As Seções 2, 3, 4 e 5 deste documento apresentam a especificação da linguagem. A Seção 6 contém informações sobre a avaliação e entregas.

## 2. Características principais e léxico

Regras para identificadores:

- Pode-se utilizar apenas letras maiúsculas,  letras minúsculas.
- Não são permitidos números, acentos, espaços em branco e outros caracteres especiais (ex.: _ , @, $, +, -, ^, % etc.).
- Identificadores não podem ser iguais às palavras reservadas ou operadores da linguagem.
- Identificadores que representam constantes com tipos primitivos possuem modificador **`unalterable`**; os que representam varáveis possuem modificador **`alterable`**

Tipos primitivos:

- A linguagem aceita os tipos symbol, number, e answer.
- **symbol**: tipo que representa um elemento da tabela ASCII, sendo escrito com aspas simples. Exemplo: 'a', '\n'.
- **number**: números inteiros ou reais. Sendo real, a parte decimal do número é separada por um ponto.
- **answer**: podem assumir dois valores: **yes** e **no**.

Vetores:

- Um vetor é composto de uma ou mais variáveis do mesmo tipo primitivo.
- Não existem vetores constantes, ou seja, que possuem modificador **`unalterable.`**
- O tamanho dos vetores é definido durante sua criação.
- Os índices dos vetores vão de 1 ao seu tamanho.
- Existem vetores multidimensionais. Exemplo: **`number vector [2][3] nome;`**
- Vetores unidimensionais do tipo **`symbol`** podem ter seus valores definidos por cadeias entre aspas duplas (“ e ”).

Blocos

- Delimitados pelas palavras **`start`** e **`finish`**.

Comentários:

- A linguagem aceita comentários de linha, indicados pelo símbolo - - no início da linha.
- A linguagem aceita comentários de bloco (possivelmente de múltiplas linhas) delimitados por {- e -}.
- O funcionamento dos comentários de bloco em **Cida** são similares aos da linguagem C. Exemplo: o que acontece se você compilar /**/*/ em C? Estudem como um compilador C reconhece o fim de um comentário de bloco.

Estruturas de controle (mais detalhes na Seção Semântico):

- **`in case that:`** estrutura similar ao ‘se/senão’
- **`as long as:`** estrutura similar ao ‘enquanto’
- **`considering:`** estrutura similar ao ‘para’

Operadores:

- Possui operadores aritméticos, relacionais e booleanos (tipo answer).
- Comandos são terminados com ‘.’ (ponto-final).

Procedimentos primitivos:

- A linguagem possui dois procedimentos primitivos: **`capture`** e **`show`**

**3. Sintático**

A gramática da linguagem foi escrita em uma versão de E-BNF seguindo as seguintes convenções:

- Variáveis da gramática são escritas em letras minúsculas sem aspas;
- Tokens são escritos entre aspas simples;
- Símbolos escritos em letras maiúsculas representam o lexema de um token do tipo especificado;
- O símbolo | indica produções diferentes de uma mesma variável;
- O operador [ ] indica uma estrutura sintática opcional;
- O operador { } indica uma estrutura sintática que é repetida zero ou mais vezes.

```
programa : 'code' ID bloco

bloco : 'start' { declaração} { comando } 'finish'

declaracao :   'alterable' tipo ID '.'
             | 'unalterable' tipo-base ID [ '<<' exp ] '.'

tipo-base :   'number'
            | 'answer'
            | 'symbol'

tipo :  tipo-base
      | tipo-base 'vector' '[' exp ']' {'[' exp ']'}

local :  ID
       | local '[' exp ']'

comando :
  local '<<' exp '.'
| 'capture' '(' {local ','} local ')' '.'
| 'show' '(' {exp ','} exp ')' '.'
| 'in' 'case' 'that' '(' exp ')' 'do' comando ['otherwise'
comando]
| 'as' 'long' 'as' '(' exp ')' 'do' comando
| 'considering' local 'from' exp 'to' exp 'by' exp 'do' comando
| bloco

exp :
  SYMBOL
| NUMBER
| ANSWER
| STRING
| local
| '(' exp ')'
| '-' exp
| exp '+' exp
| exp '-' exp
| exp '*' exp
| exp '/' exp
| exp '%' exp
| exp '==' exp
| exp '!=' exp
| exp '<=' exp
| exp '>=' exp
| exp '<' exp
| exp '>' exp
| 'not' exp
| exp 'and' exp
| exp 'or' exp
| exp 'xor' exp
```

**4. Semântico:**

Geral

- Nos casos omissos neste documento, a semântica da linguagem segue a semântica de C.
- A execução de um programa consiste na execução do código iniciado com a palavra **`code`**, seguida pelo nome do programa e por um bloco.
- A precedência dos operadores é igual à de C.

Blocos

- Contêm comandos e declarações
- Podem ser aninhados
- Escopo: o tempo de vida dos identificadores declarados em um bloco é igual ao tempo de vida deste bloco

Espaços de memória

- Variáveis são identificadas com o prefixo **`alterable`**, e podem ser inicializadas apenas em comandos posteriores à sua declaração, com uma atribuição (operador <<) ou através do comando capture. Variáveis só podem ser usadas se forem inicializadas. Identificadores que representam varáveis (**`alterable`**) não possuem limitação no número de inicializações.
- Constantes são identificadas com o prefixo **`unalterable`**. A inicialização durante a definição é opcional. Caso seja feita posteriormente, deve ser usado o operador de atribuição "<<" ou através do comando capture. Constantes representam apenas tipos primitivos e só podem ser inicializadas uma vez.

Estruturas de controle:

- **`in case that:`** equivale ao "if-else" do C.
- **`as long as:`** equivale ao "while" do C.
- **`considering`**: similar ao "for" do C, mas a terceira expressão representa o incremento (passo) que a variável sofrerá a cada iteração.

Operadores**:**

- A prioridade dos operadores é igual à de C, e pode ser alterada com o uso de parênteses.

Procedimentos primitivos:

- **`capture:`** procedimento fictício para entrada de dados a partir do teclado. Salva os valores lidos nos locais (espaços de memória) que foram passadas como argumentos.
- **`show:`** procedimento para exibição de um ou mais valores resultantes de expressões passadas como argumentos.

O que deve ser verificado na análise semântica:

- Se o nome do programa e entidades criadas pelo usuário (variáveis, vetores e constantes) são inseridos na tabela de símbolos - com os atributos necessários - quando são declarados;
- Se uma entidade foi declarada e está em um escopo válido no momento em que ela é utilizada (regras de escopo são iguais às de C);

- Se entidades foram definidas (inicializadas) quando isso se fizer necessário;
- Checar a compatibilidade dos tipos de dados envolvidos nos **comandos**, **expressões e atribuições**.

## 5. Geração de Código

- O código alvo do compilador é C.

## 6. Desenvolvimento do Trabalho

Trabalhos devem ser desenvolvidos em grupos de 4 alunos (preferencialmente), trios, duplas ou individualmente. Os grupos devem se cadastrar na planilha:

https://docs.google.com/spreadsheets/d/1JiUp0enzDKLp9bBWcOc-_sU232U9W_Jvtmg0ZCyRXjc/edit?usp=sharing

**Prazo de preenchimento da planilha: 12/12/2023.**

A submissão das tarefas do projeto deve ser feita via SIGAA, nas datas previstas na Seção 6.3. Foi aberto um fórum no SIGAA para a discussão sobre as etapas. Em caso de dúvida, verifique inicialmente no fórum se ela já foi resolvida. Se ela persiste, consulte a professora.

### 6.1. Ferramentas

- Implementação com SableCC, linguagem Java.
- IDE Java (recomendação: Eclipse).
- Submissão via SIGAA.

### 6.2. Avaliação

- A avaliação será feita com base nas etapas entregues e em arguições feitas com os grupos.
- O valor de cada etapa está definido no plano de curso da disciplina.
- O cumprimento das requisições de formato também será avaliado na nota de cada etapa.

Mais detalhes na especificação do projeto (vide aula do dia 06 de dezembro).

### 6.3. Entregas

**Etapa 1. Códigos em Cida**

- **Prazo: 18/12/2023**
- **Atividade:** escrever três códigos em **Cida** que, unidos, usem todas as alternativas gramaticais (ou seja, todos os recursos) da linguagem.
- **Formato de entrega:** arquivo comprimido contendo três códigos, onde cada código deve estar escrito em um arquivo de texto simples, com extensão "**.ci**".

**Etapa 2. Análise Léxica**

- **Prazo: 29/01/2024**
- **Atividade:** implementar analisador léxico em SableCC, fazendo a impressão dos lexemas e tokens reconhecidos ou imprimindo erro quando o token não for reconhecido.

- **Formato de entrega:** apenas o arquivo .sable deve ser enviado. O nome do arquivo deve ser `grupo_X.sable`, onde `X` é o número do grupo (vide planilha de cadastro de grupos). O nome do pacote a ser gerado pelo sablecc deve se chamar **`cida`** (em letras minúsculas e sem hífen).

**Etapa 3. Análise Sintática**

- **Prazo: 11/03/2024**
- **Atividade:** implementar analisador sintático em SableCC, fazendo impressão da árvore sintática em caso de sucesso ou impressão dos erros caso contrário.
- **Formato de entrega:** apenas o arquivo .sable deve ser enviado. O nome do arquivo deve ser `grupo_X.sable`, onde `X` é o número do grupo (vide planilha de cadastro de grupos). O nome do pacote a ser gerado pelo sablecc deve se chamar **`cida`**.

**Etapa 4. Sintaxe Abstrata**

- **Prazo: 27/03/2024**
- **Atividade:** implementar analisador sintático abstrato em SableCC, com impressão da árvore sintática resultante.
- **Formato de entrega:** apenas o arquivo .sable deve ser enviado. O nome do arquivo deve ser `grupo_X.sable`, onde `X` é o número do grupo (vide planilha de cadastro de grupos). O nome do pacote a ser gerado pelo sablecc deve se chamar **`cida`**.

**Etapa 5. Tabela de Símbolos e Análise Semântica**

- **Prazo: 10/04/2024**
- **Atividade:** validar escopo, declaração e definição de identificadores. Implementar verificação de tipos. Implementar e usar tabela de símbolos.
- **Formato de entrega:** projeto completo, incluindo obrigatoriamente: o arquivo .sable; todas as classes java escritas pelo grupo ou geradas automaticamente; e arquivos .**ci** que demonstrem o que foi feito nesta tarefa.
- Também é obrigatória a entrega de um pdf contendo uma breve explicação sobre o que foi implementado nesta etapa e como.

**Etapa 6. Geração de código (extra: 2.0 pontos):**

- **Prazo: o mesmo da Etapa 6**
- Compilação de código **Cida** com geração de código em linguagem alvo (C)

**Observações importantes:**

- Entregas após o prazo sofrem penalidade de metade da nota da etapa por dia de atraso.
- Trabalhos entregues com atraso devem ser submetidos na Tarefa ‘Entrega após prazo’, no SIGAA, que ficará aberta durante todo o período.
- Arquivos enviados por e-mail não serão considerados.

**Bom trabalho!**