Universidade do Estado da Bahia – UNEB
Colegiado de Sistemas de Informação
Fundamentos de Compiladores – 2024.2
Prof. Antonio Atta

Especificação da Linguagem Proc – v1.52

Notação BNF Estendida

Nas regras léxicas e sintáticas descritas abaixo os caracteres da notação BNF estão grifados em verde.

- Alternativas são separadas por barras verticais, ou seja, '*a* | *b*' significa "*a* ***ou*** *b*".
- Colchetes indicam ocorrência opcional: '[*a*]' significa um *a* opcional, ou seja, equivale a "*a* | *ε*" (*ε* refere-se à cadeia vazia).
- Chaves indicam repetição: '{ *a* }' significa " *ε* | *a* | *aa* | *aaa* | ..."
- Parênteses indicam ocorrência de uma das alternativas: '(*a* | *b* | *c*)' significa obrigatoriedade de escolha de *a* ou *b* ou *c*.

---

1. Regras Léxicas

letra	::=	`a` | `b` | ... | `z` | `A` | `B` | ... | `Z`
digito	::=	`0` | `1` | ... | `9`
`id`	::=	{`_`} *letra* { *letra* | *digito* | `_` }
`intcon`	::=	*digito* { *digito* }
`realcon`	::=	`intcon.intcon`
`charcon`	::=	`'`*ch*`'` | `'\n'` | `'\0'`, onde *ch* denota qualquer caractere imprimível da tabela ASCII, como especificado pela função **isprint()** da lingugem C, diferente de \ (barra invertida) e ' (apóstrofo).
`stringcon`	::=	`"`{*ch*}`"`, onde *ch* denota qualquer caractere imprimível da tabela ASCII, como especificado pela função **isprint()** da linguagem C, diferente de " (aspas) e do caractere *newline*.
`idconst`	::=	indica a ocorrência de uma constante inteira declarada e inicializada previamente com "**const int**"
`idproc`	::=	indica a ocorrência de um nome de procedimento, exceto **init**

comentário	Comentários iniciam sempre com o sufixo // (dupla barra comum) e se estendem até a ocorrência do caracter de fim de linha.

---

2. Regras Sintáticas

Símbolos *não-terminais* são apresentados em itálico; símbolos **terminais** são apresentados em negrito e, eventualmente, entre aspas por questões de clareza.

Declarações e comandos da linguagem têm estrutura própria, que indica como inicia e quando termina um comando. Sendo assim, declarações e comandos devem estar estruturalmente completos em uma única linha.

2.1 Regras de produção da gramática

prog	::=	{*decl_list_var*} {*decl_def_proc*}
decl_list_var	::=	[**const**] *tipo decl_var* { **,** *decl_var*}
decl_def_proc	::=	**prot** `idproc` **(** [[**&**] *tipo* { [ ] } { **,** [**&**] *tipo* { [ ] } }]] **)** |
		def (**init** | `idproc` **(** [[**&**] *tipo* **id**$_1$ { [(`intcon`$_2$ | `idconst`$_2$)] } { **,** [**&**] *tipo* **id**$_2$ { [(`intcon`$_2$ | `idconst`$_2$)] } }] **)**) {*decl_list_var*} { *cmd* } **endp**
decl_var	::=	`id` [= (`intcon` | `realcon` | `charcon`)] | `id` {[`intcon` | `idconst`]} [= **{** (`intcon` | `realcon` | `charcon`) {**,** (`intcon` | `realcon` | `charcon`) } **}**]
tipo	::=	**char**
	|	**int**
	|	**real**
	|	**bool**
cmd	::=	**do** `idproc` **(** [*expr*$_1$ { **,** *expr*$_2$ }] **)**
	|	**while (***expr***)** { *cmd* } **endw**
	|	**var** `id` **from** *expr*$_1$ (**to** | **dt**) *expr*$_2$ [**by** (`intcon` | `idconst`)] { *cmd* } **endv**
	|	**if (** *expr* **)** { *cmd* }{ **elif** **(** *expr* **)** { *cmd* } } [**else** { *cmd* }] **endi**

| | *atrib*
| | **getout**
| | **getint** `id`
| | **getreal** `id`
| | **getchar** `id`
| | **getstr id**
| | **putint** (`id` | `intcon`)
| | **putreal** (`id` | `realcon`)
| | **putchar** (`id` | `charcon`)
| | **putstr** (`id` | `stringcon`)

atrib ::= `id` { **[** *expr* **]** } = *expr*

expr ::= *expr_simp* [*op_rel* *expr_simp*]

expr_simp ::= [+ | –] *termo* {(+ | – | ||) *termo*}

termo ::= *fator* {(* | / | &&) *fator*}

fator ::= `id` { [*expr*] } | `intcon` | `realcon` | `charcon` | (*expr*) | ! *fator*

op_rel ::= ==
| | !=
| | <=
| | <
| | >=
| | >

2.2. Associatividade e Precedência de Operadores

A tabela a seguir apresenta a associatividade de diversos operadores, assim como, as regras de precedência de cada um deles. A operação de um operador de maior precedência deve ser executada antes da operação associada a um operador de menor precedência. A precedência dos operadores diminui, à medida que avançamos da primeira linha para a última linha da tabela.

Operador	**Associatividade**
!, +, – (unário)	direita para esquerda
***, /**	esquerda para direita

+, – (binário)	esquerda para direita
<, <=, >, >=	esquerda para direita
==, !=	esquerda para direita
&&	esquerda para direita
||	esquerda para direita

---

3. Regras Semânticas

3.1. Declarações

As regras a seguir indicam como o processamento das declarações deve ser feito. Aqui a *definição* de uma função refere-se à especificação de seus parâmetros formais, variáveis locais e ao próprio corpo da função.

1. Um identificador pode ser declarado no máximo uma vez como global e no máximo uma vez como local em um procedimento particular qualquer. Contudo, um mesmo identificador pode aparecer como local em vários procedimentos distintos. Caso ocorra de um identificador local a um procedimento possuir o mesmo lexema de um identificador global já declarado, o escopo local tem precedência enquanto ele estiver em execução.
2. Declarações de identificadores como variáveis e nomes de procedimentos (protótipo ou definição) podem ocorrer globalmente (escopo mais externo) segundo a ordem estabelecida na gramática da linguagem, mas a referência a qualquer desses elementos só pode ser feita após a sua declaração.
3. A declaração de todos os identificadores locais (no corpo da definição de um procedimento) deve ocorrer sempre antes de qualquer comando (instrução de programa), conforme indicado na gramática.
4. No escopo global, um procedimento pode ter no máximo um nome e uma assinatura associada (protótipo); um procedimento pode ser definido no máximo uma vez. A linguagem não aceita procedimentos declarados e definidos localmente, ou seja, no escopo interno de outro procedimento.
5. A declaração do protótipo (ou assinatura) de um procedimento é opcional, mas caso ocorra deve anteceder à sua definição (com o respectivo corpo de comando).
6. Se um procedimento for usado (chamado) no corpo de comando de outro procedimento, a declaração do protótipo do procedimento chamado é obrigatória e deve ser ocorrer antes da definição do procedimento chamador.

7. Quando um procedimento não possuir um protótipo declarado, a sua assinatura (protótipo) corresponderá ao que é declarado na sua primeira linha de definição (antes das declarações das respectivas variáveis locais e corpo de comandos).
8. Os tipos dos parâmetros formais declarados na definição do procedimento devem ser iguais, em número, tipificação e ordem, com os parâmetros presentes no protótipo, se declarado.
9. O procedimento principal (**init**) é o único que não deve possuir protótipo (assinatura) associado.
10. Procedimentos não possuem valor de retorno; caso seja necessário retornar valores processados ou gerados no procedimento, deve-se usar parâmetros (um ou mais) passados por referência (tipo precedido por &).
11. Um identificador pode aparecer no máximo uma vez na lista dos parâmetros formais na definição de um procedimento.
12. Os parâmetros formais de um procedimento possuem escopo local a esse procedimento.
13. A execução de todo procedimento (a exceção do procedimento principal - **init**) deve obrigatoriamente ser feita mediante o uso do comando "**do**". O procedimento principal é o ponto de entrada de execução do programa.
14. Programas sem procedimento principal (**init**) podem ser compilados, mas não executados – funcionam como bibliotecas de procedimentos.
15. Nesta versão, a linguagem não suporta tipos apontadores.
16. Na opção de inicialização de identificadores em *decl_var*, o uso de chaves casadas do lado direito do sinal de '=' só é permitido (e deve ser sempre usado) na inicialização de *arrays;* para variáveis escalares, a inicialização com o uso de chaves casadas representa erro semântico.
17. Outras regras semânticas não tratadas neste documento podem existir e devem ser elucidadas à medida que questionamentos associados forem surgindo ao longo do desenvolvimento do compilador

3.2. Requisitos de Consistência de Tipos

Variáveis devem ser declaradas antes de serem usadas. Procedimentos devem ter os seus tipos de parâmetros declarados (via assinatura), antes deles serem executados via comando "**do**" no corpo de comandos de outros procedimentos. Se um identificador é declarado como possuindo escopo local a um procedimento (variável local), então todos os usos daquele identificador se referem a essa instância local do identificador. Se um identificador não é declarado localmente a um procedimento, mas é declarado globalmente, então qualquer uso daquele identificador, dentro dos procedimentos, se refere à instância com escopo global do identificador. As regras a seguir indicam como deve ser feita a checagem da consistência de tipos. A noção de compatibilidade de tipos é definida como se segue:

1. inteiro é compatível com inteiro, e caracter é compatível com caracter;
2. inteiro é compatível com caracter, e vice-versa;
3. O tipo implícito de uma expressão relacional (ex.: a>=b) é booleano, que é compatível com o tipo inteiro; em uma variável do tipo booleano, um valor igual a 0

(zero) indica falso lógico e um valor inteiro diferente de zero indica verdadeiro lógico;
4. Qualquer par de tipos não coberto por uma das regras acima não é compatível.

3.2.1. Declaração e definição de Procedimentos

1. Procedimentos só podem ter sua execução iniciada via comando "**do**"; portanto, procedimentos não podem ocorrem como parte de uma expressão, por exemplo. Para usar valores resultantes do processamento do procedimento deve-se fazer uso auxiliar de parâmetros passados por referência.
2. A presença do marcador especial "**&**" antecedendo o tipo de um parâmetro na declaração de um procedimento (em sua assinatura) indica que aquele parâmetro será transferido por referência para o procedimento, podendo, portanto, sofrer alteração no código do procedimento, que será refletida no escopo externo; caso esse marcador especial (**&**) não ocorra, a transferência da variável do escopo externo para o escopo interno do procedimento no respectivo parâmetro será por valor (cópia). O marcador especial "**&**", quando necessário só deve aparecer na assinatura ou protótipo do procedimento (ou seja, em sua declaração), caso exista, e em sua respectiva definição – sua ocorrência é inválida, portanto, na chamada do procedimento (via comando "**do**").
3. Procedimentos podem possuir variáveis declaradas internamente. Nesse caso, elas terão escopo local e os espaços de memória alocados para elas, assim como a liberação desses espaços, ocorrerá a cada execução do procedimento.
4. Um procedimento não pode ser declarado ou definido internamente a outro procedimento.

3.2.2. Expressões

O tipo de uma expressão *expr* é estabelecido como se segue:

1. Se *expr* é uma constante inteira, então seu tipo é inteiro.
2. Se *expr* é um identificador, então o tipo de *expr* é o tipo daquele identificador.
3. Se *expr* é uma expressão na forma e_1 **+** e_2, e_1 **-** e_2, e_1 ***** e_2, e_1 **/** e_2, ou **-**e_1, então o tipo de *expr* é compatível com os tipos dos elementos da expressão, restritos a inteiro, caracter e real (**real**), ou seja, em "e_1 **+** e_2" se e_1 for caracter e e_2 for inteiro, a operação é possível porque estes tipos possuem compatibilidade entre si e o tipo da expressão fica sendo inteiro; por outro lado, se e_1 for um inteiro e e_2 for um real um conflito de tipos é estabelecido e uma mensagem de erro deve ser emitida;
4. Se *expr* é uma expressão na forma e_1 **>=** e_2, e_1 **<=** e_2, e_1 **>** e_2, e_1 **<** e_2, e_1 **==** e_2, ou e_1 **!=** e_2 então o tipo de *expr* é booleano.
5. Se *expr* é uma expressão na forma e_1 && e_2, e_1 || e_2, ou **!**e_1, então o tipo de *expr* é booleano.

As regras para checagem de tipos em uma expressão, além daquelas estabelecidas anteriormente, são as seguintes:

1. Cada argumento passado em uma execução de procedimento deve ser compatível com o parâmetro formal correspondente declarado na assinatura ou definição daquele procedimento.
2. As subexpressões associadas com os operadores +, -, *****, **/**, <=, >=, <, >, ==, e **!=** devem ser compatíveis com os tipos inteiro, caracter ou real. As subexpressões associadas com os operadores **&&**, ||, e **!** devem possuir tipos compatíveis com booleano.
3. Uma expressão usada como índice (entre colchetes) no acesso aos elementos de um *array* deve ser do tipo inteira.

3.3.3. Comandos

1. Apenas variáveis dos tipos básicos (inteiro, caracter, real e booleano) podem receber atribuições; o tipo associado ao lado direito de um comando de atribuição deve ser compatível com o tipo do lado esquerdo daquele comando de atribuição (o que está à direita e à esquerda do sinal de "=").
2. O tipo da expressão condicional em um comando "**while**", "**if**", e "**elif**" deve ser booleano. Em todos os casos, se a avaliação da expressão condicional for falsa, logo na primeira avaliação, nenhuma execução dos respectivos comandos internos deve ocorrer.
3. Em um comando "**var**", $expr_1$ deve ser menor que $expr_2$ caso a cláusula "**to**" seja usada; similarmente, $expr_1$ deve ser maior que $expr_2$ caso a cláusula "**dt**" seja usada. Em todos os outros casos, a execução do conjunto de comandos internos associados não deve ser iniciada.

---

4. Características Operacionais

A linguagem possui as mesmas características de execução de uma linguagem de programação estruturada em blocos como o C. A descrição abaixo trata de alguns pontos específicos que devem ser de interesse. Para outros aspectos não tratados explicitamente neste documento, deve-se discutir em sala o projeto da linguagem buscando encontrar uma solução de consenso entre a especificação e filosofia da linguagem e a sua implementação.

4.1. Dados

4.1.1. Escalares

Variáveis do tipo escalar (inteiro, real, caracter ou booleano) ocupam uma posição de memória na máquina virtual do compilador.

Valores do tipo caracter são considerados sinalizados e havendo necessidade de converter um caracter em um inteiro por questões de compatibilidade, essa conversão deverá estender o sinal.

4.1.2 Estruturados

A linguagem, nesta versão, só suporta o tipo estruturado *array* do tipo unidimensional (vetor) e bidimensional (matriz). *Arrays* são dados estruturados de um mesmo tipo para a linguagem e seus elementos escalares são acessíveis pelo uso do operador de indexação '[]'. Internamente, no código gerado, os *arrays* são armazenados em posições contíguas de memória; matrizes são armazenadas em posições contíguas por linha, ou seja, uma linha após a outra.

4.1.3. Constantes Cadeias

Uma cadeia de characters (*string*) "$a_1a_2a_3...a_n$" é um vetor de caracteres contendo n+1 elementos, cujos primeiros n elementos são os caracteres da respectiva cadeia de caracteres, e o último elemento é o character **NULL** ou '**0**'.

4.1.4. Inicialização

Identificadores constantes (declarados com **const**) devem ser *explicitamente* inicializados na sua declaração (e apenas neste ponto). Caso na linha de declaração de um identificador constante ocorra a declaração de outros identificadores daquele mesmo tipo, todos os identificadores dessa declaração são considerados constantes e devem ser individualmente inicializados (ex.: const int x = 10, y = 15). Identificadores não constantes podem ou não ser inicializados *explicitamente* na sua declaração – caso eles não sejam inicializados explicitamente pelo programador, seus valores devem *implicitamente* inicializados com 0, se o identificador for do tipo inteiro, 0.0, se for do tipo real, '\0' (caracter em branco), se for do tipo caracter e 0 (falso) se for do tipo booleano.

Identificadores do tipo *array* (vetores ou matrizes) podem ser inicializados na sua declaração com o uso dos caracteres '{' e '}' indicando o início e o fim dos valores de inicialização de cada elemento (separados por vírgula). Caso o número de valores de inicialização seja inferior ao número de elementos do vetor ou da matriz, o último valor especificado pelo programador deve ser usado para preencher o restante dos elementos da estrutura. Caso esse número de valores de inicialização seja maior que o número de elementos da estrutura, uma mensagem de erro deve ser emitida. No caso de *array* do tipo matriz a especificação do preenchimento na inicialização deve se dar por linha.

4.2. Expressões

4.2.1. Ordem de avaliação

- ***Expressões Aritméticas*** : os operandos de uma expressão (uma constante, um identificador, ou uma adição entre parênteses, por exemplo) devem ser calculados antes do cálculo da própria expressão. As regras sintáticas da linguagem descritas anteriormente garantem tanto a precedência quanto a associatividade dos operadores aritméticos e lógicos em uma expressão qualquer; operações de adição e subtração só são executadas após a execução das multiplicações e divisões a menos que as primeiras ocorram entre parênteses.

- ***Expressões booleanas*** : novamente, as regras sintáticas da linguagem estabelecem a ordem de avaliação dos operandos de uma operação de comparação envolvendo os operadores relacionais >=, >, <=, <, ==, **!**=; da mesma forma acontece para os operadores lógicos && (and) , || (or) e **!** (not); as expressões envolvendo esses conectores lógicos devem ser avaliadas segundo a técnica do "circuito mais curto".

4.2.2. Conversão de tipos

Se um objeto do tipo caracter é parte de uma expressão, seu valor deve ser convertido (estendendo o sinal) para um valor do tipo inteiro antes que a expressão possa ser avaliada.

4.3. Comandos de Atribuição

Ordem de Avaliação

A ordem em que a expressão do lado direito de um comando de atribuição (após o sinal de "=") será avaliada deverá respeitar as regras de precedência dos operadores estabelecidas na gramática.

4.4. Procedimentos

4.4.1. Execução de um procedimento

A execução de um procedimento é sempre iniciada pelo emprego do comando "**do**".

Um comando "**do**" executa o respectivo procedimento uma única vez.

4.4.2. Avaliação de argumentos de procedimentos

A ordem em que os argumentos de um procedimento serão avaliados antes do início da execução do procedimento obedecerá a ordem de ocorrência, ou seja, da esquerda para a direita.

4.4.3. Passagem de parâmetros

Argumentos são sempre passados por valor (cópia), a exceção dos declarados com o marcador especial "**&**" antecedendo o tipo na declaração de um procedimento (seu protótipo ou assinatura); nesse caso, a passagem do argumento se dá por referência – ou seja, alterar o valor do argumento dentro do bloco implica em perenizar essa alteração

quando o procedimento terminar sua execução – pela alteração do identificador referenciado.

Um argumento do tipo constante não pode ser associado a um parâmetro declarado com passagem por referência – apenas identificadores variáveis podem ser passados por referência.

Por definição, parâmetros do tipo *array* são sempre passados por referência (não demandam, nem devem, ser declarados com o marcador "**&**" antecedendo seu tipo).

Parâmetros do tipo *array* (vetores e matrizes) devem possuir suas respectivas dimensões máximas (limites) declarados no protótipo (assinatura), caso exista, e na definição do procedimento. Na execução do procedimento, via comando "**do**", apenas o nome do respectivo *array* (sem suas dimensões) deve ser especificado na lista de argumentos, posto que essas dimensões já foram declaradas no protótipo do procedimento e na sua definição.

4.4.4. Fim da execução de um procedimento

Um procedimento encerra sua execução e o fluxo de execução retorna para o ponto onde a execução do procedimento foi iniciada ao final da execução regular de um procedimento ou quando da ocorrência da execução do comando **getout** (inserido entre os comandos que compõem o procedimento). O comando **getout** é opcional no final de um procedimento – está implícito.

4.5. Execução do Programa

A execução do programa inicia no procedimento especial nomeado **init**, caso exista, que corresponde ao ponto de entrada de execução do programa.