Universidade do Estado da Bahia – UNEB
Colegiado de Sistemas de Informação
Fundamentos de Compiladores – 2022.2
Prof. Antonio Atta

Especificação da Linguagem[1] MinObjC – v1.0

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, "*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()** do C, diferente de \ (barra invertida) e **'** (apóstrofo).

stringcon ::= **"**{*ch*}**"**, onde *ch* denota qualquer caractere imprimível da tabela ASCII, como especificado pela função **isprint()** do C, diferente de **"** (aspas) e do caractere *newline*.

[1] Especificação adaptada parcialmente de http://www.cs.arizona.edu/classes/cs553/spring09/SPECS/c--.spec.html

comentario Comentários são como no C, i.e. uma sequência de caracteres precedida por /* e seguida por */, que não contém nenhuma ocorrência de */.

---

2. Regras Sintáticas

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

2.1 Regras de produção da gramática

prog ::= { *obj_def* | *decl* **';'** | *func* }

obj_def ::= **class id** **'{'** *data_sec* *meth_sec* **'}'**

data_sec ::= **data:** *decl_list_var* **';'** { *decl_list_var* **';'** }

meth_sec ::= **code:** { *func_prot* } **intern:** { *func_prot* }

decl_list_var ::= (*tipo* | **void**) [^]*decl_var* { **','** [^]*decl_var*}

decl ::= *decl_list_var*
| *tipo* **id** **'('** *tipos_param***')'** { **','** **id** **'('** *tipos_param***')'** }
| **void id** **'('** *tipos_param***')'** { **','** **id** **'('** *tipos_param***')'** }

decl_var ::= **id** ['[' intcon ']']

tipo ::= **char**
| **int**
| **float**
| **bool**
| *tipo_obj* [corresponde a um **id** de uma classe de objeto já declarada]

tipos_param ::= *ε*
| (*tipo* | **void**) ([^]**id** | **&**[^]**id** | [^]**id** **'['** **']'**) { ',' (*tipo* | **void**) ([^]**id** | **&**[^]**id** | [^]**id** **'['** **']'**) }

func ::= (*tipo* | **void**) [^] [*escopo***'::'**]**id** **'('** *tipos_param***')'** **'{'** { (*tipo* | **void**) [^] *decl_var*{ **','** [^]*decl_var*} **';'** } { *cmd* } **'}'**

func_prot ::= (*tipo* | **void**) [^] **id** **'('** *tipos_param***')**';

escopo ::= **id**

cmd ::= **if** **'('** *expr* **')'** *cmd* [**else** *cmd*]

```
                |   while '(' expr ')' cmd
                |   for  '(' [ atrib ] ';' [ expr ] ';' [ atrib ] ')' cmd
                |   return [ expr ] ';'
                |   atrib ';'
                |   [[^] id.]id '(' [expr { ',' expr } ] ')' ';'
                |   '{' { cmd } '}'
                |   delete id ';'
                |   ';'
atrib           ::= id [ '[' expr ']' ] = (expr  |  new tipo)
expr            ::= expr_simp [ op_rel  expr_simp ]
expr_simp       ::= [+ | – ] termo {(+ | – | ||) termo}
termo           ::= fator {(* | / | &&)  fator}
fator           ::= [^] id [ '[' expr ']' ]  | intcon | realcon | charcon |
                    [[^] id.]id '(' [expr { ',' expr } ] ')'  |  '(' expr ')'  | '!' fator
                |   [^] id.id [ '[' expr ']' ]
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 de cima para baixo na 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 uma função particular qualquer. Contudo, um identificador pode aparecer como local em várias funções distintas.
2. Declarações de identificadores como variáveis, classes de objetos e de funções (protótipo ou definição) podem ocorrer globalmente (escopo mais externo) em qualquer ordem, mas a referência a qualquer desses elementos só pode ser feita após a sua declaração.
3. A declaração de todos os identificadores localmente (no corpo da definição de uma função) devem ocorrer sempre antes de qualquer comando (instrução de programa).
4. No escopo global, uma função pode ter no máximo um nome e uma assinatura (protótipo); uma função global pode ser definida no máximo uma vez.
5. Nesta versão da linguagem, MinObjC não suporta sobrecarga de funções métodos (mesmo nome de identificador para funções métodos distintas) em um mesmo escopo de objeto; objetos distintos, no entanto, podem ter métodos com nomes de identificador similares.
6. Se uma função possuir uma assinatura declarada, então os tipos dos parâmetros formais declarados na definição da função devem ser iguais, em número e ordem, com os tipos presentes na assinatura. Da mesma forma, o tipo do valor de retorno na definição da função deve ser igual tipo do valor de retorno da assinatura da função. A assinatura, se presente, deve anteceder a definição da função.
7. Um identificador pode aparecer no máximo uma vez na lista dos parâmetros formais na definição da função.
8. O tipo **void** só pode ser usado na declaração de variáveis se a variável declarada for do tipo apontador. Da mesma forma, um parâmetro de função só pode ser declarado como **void** se ele for um apontador.

9. A identificação de escopo na linguagem está associada às classes de objetos e suas funções métodos. Nesses casos o identificador do escopo tem sempre o mesmo nome da classe do objeto associado.
10. Os atributos declarados em uma classe são sempre privados (não acessíveis por entidades fora da classe. Devem ser fornecidas funções métodos específicas para manipulação indireta desses atributos por entidades externas à classe.
11. Funções método declaradas na seção "**code:**" de uma classe são sempre públicos; por outro lado, funções método declaradas na seção "**intern:**" de uma classe são sempre privados (acessíveis por membros da classe apenas).
12. Os códigos das funções métodos de uma classe de objetos têm acesso direto aos atributos e funções métodos de um outro objeto que seja da mesma classe.
13. Os parâmetros formais de uma função possuem escopo local a esta função.
14. A chamada de toda função método deve ser precedida do nome do objeto associado (ou do apontador para esse objeto) e do operador ponto "**.**"
15. O comando "**delete**" só deve ser operado sobre um apontador.
16. Nesta versão, a linguagem suporta vetor de apontadores, mas esses apontadores devem apontar para tipos escalares apenas (tipos primitivos da linguagem).
17. O comando "**new**" só deve ser usado do lado direito de uma expressão e sempre associado à atribuição a um apontador do mesmo tipo (do operando do comando new) ou do tipo "**void**"
18. Nesta versão, a linguagem não dá suporte à aritmética de apontadores (apontadores não devem ser usados em expressões a menos que estejam associados com o operador de indireção).
19. Cada definição de classe objeto (tipo definido pelo usuário) deve apresentar ao menos dois métodos internos (privados) da classe; um construtor de objetos, que deve possuir o mesmo nome (**id**) da classe e um destrutor do objeto, que deve também possuir o mesmo nome da classe precedido do sufixo "~" (**~id**). Esses métodos devem ser definidos na seção "**intern:**" da classe, contendo instruções para serem executadas sempre que um objeto é criado ou destruído, respectivamente.
20. Outras regras semânticas 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. Funções devem ter os seus tipos de argumentos e valor de retorno especificados (via assinatura ou via definição), antes delas serem chamadas no corpo de outras funções. Se um identificador é declarado como possuindo escopo local a uma função (variável local), então todos os usos daquele identificador se referem a esta instância local do identificador. Se um identificador não é declarado localmente a uma função, mas é declarado como global, então qualquer uso daquele identificador dentro da função 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 come 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. Definição de Funções

1. qualquer função chamada de dentro de uma expressão não deve possuir tipo de valor de retorno igual a "**void**". Qualquer função que é um comando (aparece isolada em um comando) deve possuir tipo de valor de retorno igual a "**void**";
2. Uma função cujo tipo é "**void**"não pode retornar um valor, ou seja, ela não pode possuir um comando como "**return** *expr*;" em seu corpo;
3. Uma função cujo tipo não seja "**void**" não pode conter um comando na forma "**return**;" – tais funções devem conter ao menos um comando na forma "**return** *expr*;" (observe que, em tempo de execução, ainda é possível que uma função desse tipo falhe em retornar um valor por que o programador colocou o comando de retorno dentro de um comando condicional, por exemplo, o que provavelmente ocasionará um erro de execução)

3.2.2. Expressões

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

1. Se *e* é uma constante inteira, então seu tipo é inteiro.
2. Se *e* é um identificador, então o tipo de *e* é o tipo daquele identificador.
3. Se *e* é uma chamada de função, então o tipo de *e* é o tipo do valor de retorno daquela função.
4. Se *e* é uma expressão na forma *e1* + *e2*, *e1* - *e2*, *e1* * *e2*, *e1* / *e2*, ou -*e1*, então o tipo de *e* é compatível com os tipos dos elementos da expressão, retritos a inteiro, caracter e real (**float**), ou seja, em "*e1* + *e2*" se *e1* for caracter e *e2* 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 *e1* for um inteiro e *e2* for um real um conflito de tipos é estabelecido e uma mensagem de erro deve ser emitida;
5. Se *e* é uma expressão na forma *e1* >= *e2*, *e1* <= *e2*, *e1* > *e2*, *e1* < *e2*, *e1* == *e2*, ou *e1* != *e2* então o tipo de *e* é booleano.
6. Se *e* é uma expressão na forma *e1* && *e2*, *e1* || *e2*, ou !*e1*, então o tipo de *e* é booleano.

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

1. Cada argumento passado em uma chamada de função deve ser compatível com o parâmetro formal correspondente declarado na assinatura ou definição daquela função.

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.

3.3.3. Comandos

1. Apenas variáveis dos tipos básicos (inteiro, caracter, real e booleano) podem receber atribuições; objetos devem prover funções métodos específicas para esse tipo de operação. O tipo associado ao lado direito de um comando de atribuição deve ser compatível com o tipo do lado esquerdo daquele comando de atribuição.
2. O tipo de uma expressão em um comando "**return**" em uma função deve ser compatível com o tipo do valor de retorno daquela função.
3. O tipo da expressão condicional em um comando "**if**", "**for**", e "**while**" deve ser do tipo booleano.

---

4. Características Operacionais

A linguagem MinObjC 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 pontos não tratados explicitamente, deve-se considerar o comportamento de MinObjC como sendo o mesmo da linguagem C.

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, esta conversão deverá estender o sinal.

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 cujo último elemento é o character **NULL** ou '**0**'.

4.2. Expressões

4.2.1. Ordem de avaliação

- ***Expressões Aritméticas*** : Os operandos de uma expressão (uma chamada de função, 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 estes 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 será avaliada deverá respeitar as regras de precedência dos operadores estabelecidas na gramática.

4.4. Funções

4.4.1. Avaliação de argumentos de funções

A ordem em que os argumentos de uma função deverão ser avaliados antes da chamada da função obedecerá a ordem de ocorrência, ou seja, da esquerda para a direita.

4.4.2. Passagem de parâmetros

MinObjC não aceita a passagem para valores escalares (constantes), apenas variáveis podem ser usadas como argumento em chamadas de funções. Variáveis podem ser passadas por valor ou por referência na chamada de funções.

Um objeto do tipo caracter deve ser convertido (por extensão do sinal) para um objeto de 32 bits antes de ser passado como parâmetro para uma função.

4.4.3. Retorno de uma função

Se uma função possui um tipo de retorno diferente de "**void**"**,** ele deve possuir um comando de retorno ("**return**") cujo argumento deve possuir tipo compatível com o valor de retorno

da função. Se o programador não especificar o comando de retorno no corpo da função nenhum valor deve ser retornado.

4.5. Execução do Programa

A execução do programa inicia no procedimento nomeado **main().** Todo programa deve obrigatoriamente possuir um e um único procedimento nomeado **main().**