Universidade do Estado da Bahia – UNEB
Colegiado de Sistemas de Informação
Fundamentos de Compiladores – 2017.2
Prof. Antonio Atta

Especificação da Linguagem[1] Cmm

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**

caraccon ::= `'`*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).

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

comentario Comentários são como no C, i.e. uma sequência de caracteres precedida

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

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

decl ::= *tipo decl_var* { '**,**' *decl_var*}
| *tipo* **id** '**(**' *tipos_param*'**)**' { '**,**' **id** '**(**' *tipos_param*'**)**' }
| **semretorno id** '**(**' *tipos_param*'**)**' { '**,**' **id** '**(**' *tipos_param*'**)**' }

decl_var ::= **id**

tipo ::= **caracter**
| **inteiro**
| **real**
| **booleano**

tipos_param ::= **semparam**
| *tipo* (**id**) { '**,**' *tipo* (**id**) }

func ::= *tipo* **id** '**(**' *tipos_param*'**)**' '**{**' { *tipo decl_var*{ '**,**' *decl_var*} '**;**' } { *cmd* } '**}**'
| **semretorno id** '**(**' *tipos_param*'**)**' '**{**' { *tipo decl_var*{ '**,**' *decl_var*} '**;**' } { *cmd* } '**}**'

cmd ::= **se** '**(**' *expr* '**)**' *cmd* [**senao** *cmd*]
| **enquanto** '**(**' *expr* '**)**' *cmd*
| **para** '**(**' [*atrib*] '**;**' [*expr*] '**;**' [*atrib*] '**)**' *cmd*
| **retorne** [*expr*] '**;**'
| *atrib* '**;**'
| **id** '**(**' [*expr* { '**,**' *expr* }] '**)**' '**;**'
| '**{**' { *cmd* } '**}**'
| '**;**'

atrib ::= **id** = *expr*

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

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

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

fator ::= **id** | **intcon** | **realcon** | **caraccon** |
id '**(**' [*expr* { ',' *expr* }] '**)**' | '**(**' *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 uma operador de menor precedência. A precedência dos operadores diminuem, 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. Uma função pode ter no máximo uma assinatura (protótipo); uma função pode ser definida no máximo uma vez.
3. Se uma função possuir uma assinatura declarada, então os tipos dos parâmetros formais declarados na definição da função devem casar (ou seja, serem os mesmos) em número e ordem com os tipos presentes na assinatura. Da mesma forma, o tipo do valor de retorno na definição da função deve casar com o tipo do valor de retorno da assinatura da função. A assinatura, se presente, deve anteceder a definição da função.
4. Um identificador pode aparecer no máximo uma vez na lista dos parâmetros formais na definição da função.
5. Os parâmetros formais de uma função possuem escopo local a esta função.
6. Se uma função não possuir parâmetros formais, sua assinatura/definição deve indicar esta situação usando a palavra reservada **semparam** no lugar dos parâmetros formais.

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 **semretorno**. Qualquer função que é um comando (aparece isolada em um comando) deve possuir tipo de valor de retorno igual a **semretorno;**
2. Uma função cujo tipo é **semretorno** não pode retornar um valor, ou seja, ela não pode possuir um comando como "**retorne** *expr*;" em seu corpo;
3. Uma função cujo tipo não seja **semretorno** não pode conter um comando na forma "**retorne**;" – tais funções devem conter ao menos um comando na forma "**retorne** *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**, 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; 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 **retorne** em uma função deve ser compatível com o tipo do valor de retorno daquela função.
3. O tipo da expressão condicional em um comando **se**, **para**, e **enquanto** deve ser do tipo **booleano**.

---

4. Características Operacionais

A linguagem Cmm 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 Cmm como sendo o mesmo da linguagem C.

4.1. Dados

4.1.1. Escalares

Um objeto do tipo **inteiro** ocupa 32 bits; um objeto do tipo **caracter** ocupa 8 bits; um tipo **real** ocupa 32 bits e um tipo **booleano** ocupa 8 bits.

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 array 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.

Conversão de tipos

Um valor do tipo **caracter** deve ser convertido (extensão do sinal) para um valor de 32 bits antes de ser atribuído a um objeto do tipo **inteiro**.

Um valor do tipo **inteiro** deve ser convertido (truncado) para um valor de 8 bits, descartando os 24 bits mais significativos, antes de ser atribuído a um objeto do tipo **caracter**.

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

Cmm não aceita a passagem de valores escalares (constantes), apenas variáveis podem ser usadas como argumento em chamadas de funções. Variáveis são passadas **sempre** 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.

Se uma função que possuir um parâmetro formal do tipo **caracter**, este parâmetro será passado como um valor de 32 bits e deve ser convertido (truncado) para um valor de 8 bits antes que possa ser usado.

4.4.3. Retorno de uma função

Se uma função possui um tipo de retorno diferente de **semretorno,** ele deve possuir um comando de retorno (**retorne**) 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 **principal().** Todo programa deve obrigatoriamente possuir um e um único procedimento nomeado **principal().**