

Disciplina: DIM0437 — Linguagens de Programação: Conceitos e Paradigmas

Docente: Umberto Souza da Costa

Discentes: Dogival Ferreira da Silva Junior

Felipe Cortez de Sá

Gabriel Sebastian von Conta

Phellipe Albert Volkmer

Vinícius Araújo Petch

Subproblema 7 — Subprogramas

1 Problema

1.1 Produto do problema

Definição da sintaxe e semântica intuitiva dos mecanismos que regem as formas de abstração de processamento (procedimentos e funções) da linguagem de programação a ser definida pelo grupo. Incluir a representação destas abstrações, assim como os mecanismos de passagem de parâmetros e a implementação sugerida.

1.2 Questões

1. Como serão definidos os procedimentos e funções de sua linguagem? Note que estes conceitos são diferentes, embora sejam tratados de forma unificada por algumas linguagens de programação. Quais são as diferenças entre esses conceitos?
2. Como serão definidos os parâmetros da linguagem? Quais dados poderão ser colocados como argumentos em chamadas a procedimentos e funções? Nomes de subprogramas poderão ser utilizados como parâmetros?
3. Quais serão as formas de passagem de parâmetros e como serão implementadas?
4. Sua linguagem verificará os tipos de parâmetros dos subprogramas?
5. A linguagem terá subprogramas sobrecarregados ou genéricos?
6. A linguagem deverá ter compilação separada ou independente?
7. Sua linguagem dará suporte a co-rotinas?

2 Resoluções

1. Na nossa linguagem, daremos suporte tanto a funções quanto procedimentos, procurando explicitar sua diferença através da sintaxe com o objetivo de deixar claros ambos os conceitos facilitando o aprendizado dos nossos usuários.

Tanto funções quanto procedimentos são sub-rotinas, ou seja, sequências de comandos reutilizáveis que podem ser alteradas sem a necessidade de sua repetição no código-fonte. A diferença conceitual entre procedimento e função reside na presença de variáveis de retorno para funções. Os procedimentos, por outro lado, são úteis apenas quando geram efeitos colaterais.

Em C, por exemplo, procedimentos possuem a mesma sintaxe de funções, com `void` no lugar do tipo:

```
int funcao(int a, int b) {
    return a + b;
}

void procedimento() {
    puts("Este comando gera um efeito colateral");
}
```

A sintaxe da nossa linguagem, portanto, possuirá a seguinte forma:

```
<function> ::= function <id> "(" <parameters_list> ")" "{" <stmts> <return_stmt> "}"
<procedure> ::= procedure <id> "(" <parameters_list> ")" "{" <stmts> "}"
```

2. Os parâmetros em nossa linguagem poderão ser de qualquer tipo da linguagem, inclusive os tipos criados pelo usuário. A utilização de parâmetros será de forma posicional, uma vez que, como a maior parte dos programas será curto, não são necessários parâmetros-chave, que diminuiriam a capacidade de escrita do nosso código e o tornaria muito extenso. Subprogramas não poderão ser utilizados como argumentos, uma vez que acarretaria em uma queda de legibilidade para o programa.
3. Como decidimos usar variáveis de referência em nossa linguagem, teremos necessariamente passagem por referência, ou seja, o endereço do argumento é passado para o subprograma e colocado na stack. Também teremos passagem por valor com cópia, em que uma cópia do argumento é criada na stack ao ser chamado o subprograma.
4. Sim, ela utilizará o método de protótipo, onde os tipos dos parâmetros são incluídos dentro da lista de parâmetros, e por meio deste o tipo de um parâmetro é checado. Em C99 e C++, é possível evitar a checagem de tipo para alguns parâmetros, porém essa funcionalidade não será utilizada.

Um exemplo de prototipagem em nossa linguagem seria:

```
int function1 (int par1, float par2) {
}
```

Em C, usar reticências após um primeiro argumento causa evasão de checagem de tipo. No exemplo seguinte, `printf` checará apenas o primeiro parâmetro. Após isso, tudo é permitido, permitindo que a função possa imprimir qualquer tipo suportado pela sintaxe de formatação implementada:

```
int printf(const char* format_string, ...);
```

Decidimos incorporar o mesmo recurso com a mesma sintaxe em nossa linguagem.

5. Subprograma genérico é um tipo de subprograma que funciona para diferentes tipos de entrada em diferentes ativações, é implementado com funções de template, definindo

um grupo de funções que pode ser gerado. Cabe ao compilador decidir que versão será criada e utilizada analisando os parâmetros de cada chamada.

Subprogramas por sobrecarga, por outro lado, baseiam-se em um grupo de subprogramas de mesmo identificador declarado várias vezes no esqueleto do programa. Uma função específica do grupo é usada dependendo dos parâmetros quando a função é chamada.

Na nossa linguagem optamos por usar somente sobrecarga, ignorando subprogramas genéricos. Essa decisão advém da maior legibilidade e entendimento em se definir sobrecarga em detrimento da simplicidade e facilidade de escrita obtida a partir de uma função de template, assim julgando mais adequado aos usuários de nossa linguagem (iniciantes).

6. Na compilação separada, partes do código podem ser compiladas em tempos diferentes, desde que essas partes não possuam outras dependências externas. Já na compilação independente, essa dependência não importa: qualquer unidade de código pode ser compilada sem se preocupar com dependências. Porém, isso faz com que unidades compiladas separadamente não tenham verificação quanto à coerência de tipo. Caso a linguagem não possua nenhum dos dois tipos de compilação citados, ou seja, compilação única, ela se tornará virtualmente inútil para aplicações industriais.

Nossa linguagem terá compilação separada, pois essa compilação é bastante prática, já que caso ocorra uma alteração no código, nem sempre será necessário compilar todo o código. A escolha da compilação separada ao invés da compilação independente se dá na falta de verificação de coerência de tipo desta.

7. Uma co-rotina é um tipo especial de subprograma. Ao invés de possuir uma relação mestre-escravo entre o subprograma que chama e o subprograma chamado, ambos estão em uma relação mais justa.

Um exemplo de co-rotina seria:

```
sub co1(){  
    ...  
    resume co2();  
    ...  
    resume co3();  
    ...  
}
```

Na primeira chamada de *co1*, a sua execução começa a partir da sua primeira declaração, até que *co2* seja retomado. Na próxima vez que *co1* seja retomado, sua execução continuará a partir da primeira declaração após a retomada de *co2*, até a retomada de *co2*. Na sua terceira retomada, *co1* continuará da primeira declaração após a retomada de *co3*.

Apesar de co-rotinas serem uma funcionalidade relevante a uma linguagem, a nossa linguagem possui um escopo educativo, portanto elas não serão utilizadas, já que exigem um domínio mais avançado de conceitos de linguagens de programação.