Simples
Referência Básica

Anderson Faustino da Silva
Universidade Estadual de Maringá
Departamento de Informática

1 Aspectos Léxicos

Um *identificador* é uma sequência de letras, dígitos que iniciam com uma letra. Em Simples letras minúsculas são diferentes de letras maiúsculas. Espaços em branco, tabulação, quebra de linha, retornos e comentários podem aparecer entre *tokens* e são ignorados. Um comentário inicia com /* e termina com */, porém não é permitido comentário aninhado.

Uma contante inteiro é uma sequência de um ou mais dígitos. Por sua vez uma contante real é formada por duas partes, uma inteira e uma decimal, separadas por uma vírgula.
Uma constante cadeia é uma sequência de zero ou mais caracteres e espaços cercados por “.

As palavras reservadas são: pare, continue, para, enquanto, faça, fun, se, verdadeiro, falso, tipo, de, limite, var, inteiro, real, cadeia, ref, retorne, nulo, início, fim.
Os símbolos são , : ; ( ) [ ] { } . + - * / <> < <= > >= & | := = == ?.

2 Declarações

Um programa Simples é uma lista de declarações (ou blocos de declarações).

lista_declarações:
declaração
| lista_declarações declaração:

declaração:
declaração_tipo:
| declaração_variável
| declaração_função

2.1 Tipos

declaração_tipo:
tipo *tipo_id = descritor_tipo*

descritor_tipo:
tipo_id
| {tipo_campos}
| [tipo_contantes] ***de*** *tipo_id*

tipo_campos:
tipo_campo
| tipo_campos , tipo_campo

tipo_campo:
id : tipo_id

tipo_constantes:
constante_inteiro
| tipo_constantes , constante_inteiro

Simples possui três tipos pré-definidos: inteiro, real e cadeia. Novos tipos podem ser definidos e tipos existentes podem ser redefinidos.

As três formas de *tipo* são referentes à: tipo (cria uma referência a um determinado tipo), registros (que podem possuir N campos de tipos distintos) e vetores (que podem ter N dimensões). A declaração de um tipo sempre cria um tipo distinto, ou seja, mesmo que dois tipos possuam a mesma estrutura estes são diferentes em Simples. Por sua vez, uma declaração *tipo a=b* cria uma referência.

Um bloco de declarações de tipo (consecutivas, ou seja, sem outro tipo de declarações no bloco) podem conter tipos mutuamente recursivos. Em um bloco, deste tipo, não pode existir duas declarações com o mesmo identificador. Cada ciclo recursivo deve passar por um tipo registro ou vetor.

Tipos possuem seu próprio espaço de nomes.

2.2 Variáveis

declaração_variável:
    var *id : tipo_id := inicialização*

inicialização:
    expr
    | { lista_expr }

Em Simples variáveis são inicializadas na declaração. Variáveis e funções compartilham do mesmo espaço de nomes. Um registro é inicializado atribuindo uma lista de expressões a declaração.

2.3 Funções

declaração_função:
    fun id (tipo_campos) = corpo
    | **fun** id (tipo_campos) : tipo_id = corpo

A primeira forma declara um procedimento, enquanto a segunda uma função (a qual deve terminar com o comando **retorne** seguido de uma expressão). Ambas as formas podem especificar uma lista de zero ou mais argumentos, os quais são passados por valor ou referência (com o uso da palavra reservada **ref**).

O escopo dos argumentos é o corpo da função (ou procedimento). Em Simples uma função pode retornar qualquer tipo definido no escopo do programa, seja por valor ou referência. O corpo da função é semelhante ao corpo de um programa, exceto que o corpo de uma função deve conter pelo menos um bloco de comandos.

Um bloco de declarações de funções (consecutivas, ou seja, sem outro tipo de declarações no bloco) podem conter tipos mutuamente recursivos. Em um bloco, deste tipo, não pode existir duas declarações com o mesmo identificador.

Em Simples não existem funções aninhadas e é possível existir chamadas à funções ainda não declaradas (ou seja, cuja declaração está adiante no código). Por fim, a função *inicial* é o ponto de entrada na execução do programa. Tal função não possui argumentos e retornar um valor inteiro.

3 Comandos e Expressões

3.1 Bloco

bloco:
    início *lista_comando* ***fim***

lista_comando:
    comando
    | lista_comando ; comando

comando:
    local := expr
    | ***se*** *expr* ***verdadeiro*** *bloco*
    | ***se*** *expr* ***verdadeiro*** *bloco* ***falso*** *bloco*
    | ***para*** *id* ***de*** *expr* ***limite*** *expr* ***faça*** *bloco*
    | ***enquanto*** *expr* ***faça*** *bloco*
    | id (lista_expr)
    | ***pare***
    | ***continue***

Um bloco contém uma lista de comandos separados por ;. Portanto, não é permitido uma expressão aparecer em um bloco.

3.2 Expressões

expr:
    constante_inteira
    | constante_real
    | constante_cadeia
    | local
    | expr operador_binário expr
    | expr ? expr : expr
    | id (lista_expr)
    | (expr)
    | { lista_expr }

lista_expr:
    expr
    | lista_expr, expr

Expressões sempre retornam um valor, o qual sempre deve ser armazenado em uma localização. Portanto não é permitido chamar uma função sem armazenar seu retorno em uma localização.

Simples possui uma expressão ternária (?), a qual avalia uma expressão e retorna o valor da expressão após o operador caso a

avaliação seja verdadeira, ou o valor da última expressão caso contrário.

3.3 Local de Armazenamento

local:
id
| local . Id
| local [lista_expr]

Um local representa uma localização de armazenamento a qual pode ser atribuído um valor: variáveis, parâmetros, campos de estruturas e elementos de vetores. Os elementos de um vetor são indexados por 0, 1, 2, … n - 1.

3.4 Registros e Vetores

Registros contém de 1 a *N* campos. Vetores podem conter *N* dimensões. Ambos são alocados de forma contínua.

3.5 Chamada de Função

A invocação de uma função *pode* retornar um valor, e esta pode ter zero ou mais argumentos separados por ,. Quando uma função é invocada os argumentos atuais (parâmetros) são avaliados da esquerda para a direita e alocados nos argumentos formais utilizando as regras de escopo estático. Uma função que não retorna valor não pode ser usada como uma expressão, tal caso acarreta um erro semântico.

3.6 Operadores

Os operadores binários são + - * / == <> < > <= >= & |.

Parênteses agrupam expressões da forma usual.

Os operados binários +, -, * e / requer operandos do mesmo tipo (inteiros ou reais) e retornam um resultado do mesmo tipo dos operandos.

Os operados binários >, <, >= e >= compara seus operandos e produz um inteiro 1 se a comparação obteve sucesso ou 0 caso contrário. Comparação de strings é feita utilizando a ordem lexicográfica ASCII.

Os operadores binários == e <> comparam operandos do mesmo tipo e retornam 0 ou 1. Strings são iguais se contem os mesmos caracteres. Dois registros ou vetores são iguais se todos os campos são iguais.

Os operadores lógicos & e | são operadores preguiçosos sobre inteiros. Tais operadores não avaliam o argumento da direita se o da esquerda determina o resultado. Zero é considerado falso, qualquer outro valor é considerado verdadeiro.

A ordem de procedência é * e /, + e -, ==, <>, >, <, >=, <=, &, |, :=.

Os operadores +, -, * e / são associativos a esquerda. Os operadores de comparação não são associativos, ou seja a==b==c é um erro, mas a==(b==c) é legal.

3.7 Atribuição

Uma atribuição avalia a expressão da direita e então vincula o valor resultante a uma localização. Atribuições não produzem valores, então a := b := 1 é ilegal.

3.8 Nulo

Uma expressão nula (**nulo**) representa um valor que pode ser atribuído a um registro. Acessar um campo de um registro nulo é um erro de execução. Nulo pode ser usado um tipo registro pode ser determinado, desta forma os exemplos a seguir são legais.

var a : rec := nulo
a := nulo
se a <> **nulo verdadeiro** …
se a == **nulo verdadeiro** …
func f(p: rec) = f(nulo)

Porém, o exemplo a seguir é ilegal (erro semântico).

se nulo == **nulo verdadeiro** …

3.8 Controle de Fluxo

O comando **pare** interrompe um laço, enquanto o comando **continue** executa um salto para o início do laço. Portanto, tais comandos somente são válidos dentro de laços.

4 Biblioteca Padrão

fun imprime(c : cadeia)
Imprime uma cadeia na saída padrão.

fun imprimei(v : inteiro)
Imprime uma contante inteiro na saída padrão.

fun imprimer(v : real)
Imprime uma constante real na saída padrão.

fun emite()
Execute um *flush* na saída padrão.

fun lc() : cadeia
Lê uma cadeia da entrada padrão.

fun li() : inteiro
Lê uma contante inteiro da entrada padrão.

fun lr() : real
Lê uma constante real da entrada padrão.

fun ordem(c : cadeia) : inteiro
Retorna o valor ASCII do primeiro caracter da cadeia ou -1 se a cadeia é vazia.

fun chr(i : inteiro) : cadeia
Retorna o caracter para o valor ASCII i. Erro de execução se o valor é inválido.

fun tamanho(c : cadeia) : inteiro
Retorna a quantidade de caracteres da cadeia.

fun subcadeia(c : cadeia, i : inteiro, n : inteiro) : cadeia
Retorna a subcadeia de c iniciando no caracter i, composta por n caracteres. O primeiro caracter inicia na posição 0.

fun concatene(c1 : cadeia, c2 : cadeia) : cadeia
Retorna uma nova cadeia formada por c1 seguida por c2.

fun inverter(i : inteiro) : inteiro
Retorna 1 se a i for zero, ou 0 caso contrário.

fun termine(i : inteiro)
Termina a execução do programa com o código i.

fun gere_inteiro() : inteiro
Retorna um número inteiro aleatório

fun gere_real() : real
Retorna um número real aleatório