Simples
Referência Básica
V 2

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 não existe diferença entre letras minúsculas e 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 */.

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 um ponto. Uma constante cadeia é uma sequência de zero ou mais caracteres e espaços cercados por “ (aspas duplas).

As palavras reservadas são: pare, continue, para, fpara, enquanto, fenquanto, faça, se, fse, verdadeiro, falso, tipo, de, limite, global, local, inteiro, real, cadeia, valor, ref, retorne, nulo, início, fim.

Os símbolos são , : ; ( ) [ ] { } . + - * / == != < <= > >= & | := =

2 Programa

Um programa Simples tem o seguinte formato:

programa: declarações
ação

declarações:
lista_declaração_de_tipo
lista_declarações_de_globais
lista_declarações_função

lista_declaração_de_tipo:
// vazio
*| **tipo** : lista_declaração_tipo*

lista_declaração_de_globais:
// vazio
*| **global** : lista_declaração_variável*

lista_declaração_de_funções:
// vazio
*| **função** : lista_declaração_função*

ação:
***ação** : lista_comandos*

Em Simples é possível não existir declarações e apenas um conjunto de ações, o qual contém um ou mais comandos (ponto de entrada).

2.1 Tipos

declaração_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 não podem ser redefinidos.

As três formas de *tipo* são:

1. nome: cria uma referência a um determinado tipo
2. registros: que podem possuir N campos de tipos distintos
3. 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 fim, Simples permite tipos mutuamente recursivos.

2.2 Variáveis

declaração_variável:
id : tipo_id := inicialização

inicialização:
expr
| { criação_de_registro }

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

Em Simples variáveis são inicializadas na declaração. Um registro é inicializado atribuindo uma lista de expressões a declaração, na seguinte forma: {campo = valor, campo = valor, ...}.

2.3 Funções

declaração_função:
    id (args) = corpo
    | id (args) : id = corpo

args : modificador id : id

modificador: ***valor*** | ***ref***

corpo:
    declarações_de_locais
    ação *: lista_comandos*

declarações_de_locais:
    // vazio
    | ***local*** *: lista_declaração_variável*

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 (com o uso da palavra reservada **valor**) ou referência (com o uso da palavra reservada **ref**).

Variáveis locais e argumentos compartilham o mesmo escopo, ou seja o corpo da função (ou procedimento). Desta forma, não pode existir argumentos com o mesmo nome de variáveis locais.

O corpo da função é composto por dois blocos: declaração de locais e uma lista de comandos.

Em Simples uma função pode retornar qualquer tipo definido no programa, seja por valor ou referência.

Em Simples é possível existir chamadas à funções ainda não declaradas (ou seja, cuja declaração está adiante no código).

Tipos, variáveis e funções possuem ambientes diferentes. Portanto, o exemplo a seguir é válido.

```
tipo:
    a = inteiro

global:
    a : a := 10

função:
    a(a:a) =
        ação: imprimei(a)
ação:
    a(10)
```

3 Comandos

lista_comandos:
comando
| lista_comandos ; comando

comando:
local := expr
| chamada_de_função
*| **se** expr **verdadeiro** lista_comandos **fse***
*| **se** expr **verdadeiro** lista_comandos **falso** lista_comandos **fse***
*| **para** id **de** expr **limite** expr **faça** lista_comandos **fara***
*| **enquanto** expr **faça** lista_comandos **fenquanto***
*| **pare***
*| **continue***
*| **retorne** expr*

Uma lista de comandos contém um ou mais comandos (neste último caso separados por ;). As palavras reservadas **pare** e **continue** somente podem aparecer dentro de um laço. Por fim a palavra reservada **retorne** só aparece uma função.

Atribuições apenas são permitidas em variáveis do mesmo tipo. Outro detalhe é o fato de não ser permitido chamar uma função sem armazenar o valor de retorno em uma variável, isto é um erro semântico.

4 Expressões

expr: *expressão_lógica* *// &, |*
| expressão_relacional *// ==, !=, <, <=, >, >=*
| expressão_aritmética *// +, -, *, /*
| criação_de_registro *// {a = 10, b= 20, ...}*
| nulo *// nulo*
| expressão_com_parênteses *// (expr)*
| chamada_de_função *// id (args)*
| local_de_armazenamento *// variável*
| literal *// inteiro, real, cadeia*

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

Expressões sempre retornam um valor, o qual sempre deve ser armazenado em um local_de_armazenamento. Este representam:

1. uma variável simples
2. um campo de um registro
3. uma posição em um vetor

Os elementos de um vetor são acessados por 0, 1, 2, … n - 1.

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. erro semântico.

6 Operadores

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

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

Os operadores lógicos == 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 possuem o mesmo tipo e 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 aritméticos e lógicos são associativos a esquerda. Os operadores relacionais não são associativos, ou seja a==b==c é um erro, mas a==(b==c) é legal.

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.

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 utilizado para criar um registro indeterminado. Os exemplos a seguir são legais.

a : rec := **nulo**
a := **nulo**
se a != **nulo verdadeiro** …
se a == **nulo verdadeiro** …
f(p: rec) -> f(**nulo**)

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

se nulo == **nulo verdadeiro** …

9 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.

10 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

11 Considerações

- Embora este documento não apresente a gramática completa, na forma Backus-Naur, ele a descreve por completo. Portanto, com este documento é possível especificar (implementar) toda a gramática da linguagem Simples.
- Observe que algumas construções necessitam de precedência; portanto é necessário implementar isto na gramática.
- É possível implementar esta gramática sem nenhum conflito. Lembre, conflitos shift-reduce indicam que o parser pode usar shift ou reduce dado o contexto da pilha e o lookahead. Por sua vez, conflitos reduce-reduce indicam que o parser pode reduzir por pelo menos duas regras distintas. O primeiro caso (shift-reduce) é resolvido dando prioridade a shift, desta forma é ***possível*** suportar tais conflitos. O segundo caso (reduce-reduce) é um erro que não deve ser negligenciado, pois indica um erro grave na gramática, o que pode ocasionar saídas diferentes do que a esperada. Ambos conflitos são removidos alterando a gramática.