A Linguagem Simples

José de Oliveira Guimarães
Departamento de Computação
UFSCar - São Carlos, SP
Brasil
e-mail: jose@dc.ufscar.br

September 13, 2004

A linguagem Simples é um subconjunto de Green [8] e suporta todos os conceitos de orientação a objetos tais como classes, herança e polimorfismo. Simples é também um superconjunto de S2.[1] De fato, um programa em S2 pode ser considerado como o corpo de um método de uma classe em Simples.

Um exemplo de um pequeno programa nesta linguagem é mostrado na Figura 1. Todo programa deve ter uma classe com nome `Program` com um método sem parâmetros chamado `run`. Para iniciar a execução do programa, o sistema de tempo de execução cria um objeto da classe `Program` e envia a ele a mensagem `run`. O código em C++ equivalente seria:

```
  Program *p;

p = new Program;
p->run();
```

Nas seções a seguir definiremos os principais elementos da linguagem Simples.

1 Classes

Uma classe possui o formato

```
class Nome
  parte privada
  parte publica
end
```

onde `Nome` é um identificador diferente do nome de todas as classes declaradas previamente. A parte privada é opcional mas, se presente, começa com

```
private:
```

que pode ser seguido por declarações de métodos e variáveis de instância, em qualquer ordem, como mostra a Figura 2. A parte pública é também opcional e pode conter apenas métodos. O método `run` da classe `Program` deve ser público.

A sintaxe para declaração de um método é esboçada na Figura 3. O tipo de retorno é opcional. A sintaxe da declaração das variáveis locais e das instruções do corpo do método é igual à da linguagem

[1] S2 é semelhante ao compilado 10 visto no semestre anterior, um Pascal simplificado.

```
class Store
  private:
    var n : integer;
  public:
    proc get() : integer
      begin
      return self.n;
      end
    proc set( pn : integer )
      begin
      self.n = pn;
      end
end


class Program
  public:
    proc run()
      var s : Store;
          a : integer;
    begin
    s = Store.new();
    read(a);
    s.set(a);
    write( s.get() );
    end
end
```

Figure 1: Um programa em Simples

```
class A
  private:
    var x : integer;
    proc soma( y : integer ) : integer
      begin
      return self.x + y;
      end
    var ok : boolean;
    proc setOk( pOk : boolean )
      begin
      self.ok = pOk;
      end
  public:
    ...
end
```

Figure 2: Um exemplo de classe em Simples

```
proc nomeMet( lista-parametros ) : TipoRetorno
  Declaracao de Variaveis Locais
begin
  Corpo do Metodo
end
```

Figure 3: Sintaxe para declaração de um método

S2 e descrita na seção seguinte. O nome de um método ou variável de instância deve ser diferente do nome dos métodos e variáveis de instância da mesma classe.

Variáveis e parâmetros cujo tipo é uma classe são similares a ponteiros em C++. Portanto, em

```
...
  var s, t : Store;
begin
...
s = Store.new();
t = s;
...
end
```

a declaração de `s` e `t` não cria objetos da classe `Store`. Um objeto desta classe deve ser criado dinamicamente como em

```
s = Store.new();
```

A desalocação de memória é feita automaticamente pelo coletor de lixo.

Uma variável cujo tipo é uma classe pode receber `nil` em uma atribuição. `nil` é uma variável global da classe `Nil` que não possui métodos.

1.1 Herança

Herança de uma classe `A` por uma classe `B` é indicada por

```
class B subclassOf A
   ...
end
```

A classe `B` herda todas as variáveis de instância e métodos de `A`. Um método público de `A` pode ser redefinido em `B`, desde que a sua assinatura[2] não seja modificada. O método redefinido deve também ser público.

A classe `B` não possui acesso à parte privada de `A`. A instrução

`super.m(`p_1`, `p_2`, ... `p_n`)`

dentro de um método de `B` faz o compilador procurar por um método `m` começando na superclasse de `B`, `A`. Se ele não for encontrado, a busca continua na superclasse de `A` e assim por diante. Os parâmetros p_1, p_2, ... p_n devem ser convertíveis[3] para os parâmetros formais do método `m` encontrado.

O envio de mensagem a `super` resulta em uma chamada direta a um método de uma superclasse de `B`, pois a busca é feita em tempo de compilação.

1.2 Envio de Mensagens

A instrução

```
s.calc(b)
```

é o envio da mensagem "`calc(b)`" ao objeto apontado por `s`. Este envio de mensagem será válido se:

1. a classe declarada de `s` ou suas superclasses tiver um método público com nome `calc`;
2. este método tomar um único parâmetro e o tipo de `b` puder ser convertido[4] para o tipo do parâmetro formal.

[2] Nome, tipos dos parâmetros e tipo do valor de retorno.
[3] Leia a seção 2.5.
[4] Leia a seção 2.5.

Em tempo de execução, o sistema de tempo de execução (STE) fará uma busca por método `calc` na classe do objeto apontado por `s`. Se ele não for encontrado lá, a busca continuará na superclasse da classe de `s`, superclasse da superclasse e assim por diante. Quando um método for encontrado, ele será executado. O STE sempre encontrará o método adequado exceto quando a variável apontar para `nil`.

Um método que não retorna um valor deve ser utilizado como uma instrução:

```
stack.print();
```

Um método que retorna um valor deve somente ser chamado em uma expressão, como em

```
if stack.getSize() > 0 then ... endif
```

1.3 self

A palavra chave `self` representa uma variável cujo tipo é a classe onde ela está sendo utilizada. `self` aponta para o objeto que recebeu a mensagem que causou a execução do método. `self` em Green é equivalente a `this` em Java ou C++. Utilizando a classe `Store` da Figura 1, o código

```
s = Store.new();
s.set(12);
```

causará a execução do método `set` de `Store`. Dentro deste método, `self` irá apontar para o mesmo objeto que `s`. Esta palavra chave é utilizada para manipular as variáveis de instância e chamar métodos privados e públicos da própria classe — observe a classe `Store`. O valor de `self` não pode modificado em uma atribuição.

2 Elementos Básicos de Simples

2.1 Comentários

Comentários na linguagem são colocados entre "`/*`" e "`*/`". Comentários aninhados não são permitidos, como

```
/*  comentario /* outro comentario */  fim primeiro */
```

O comentario termina no primeiro `*/` encontrado.

A linguagem Simples também admite comentários do tipo `//` iguais aos de C++. Qualquer coisa após `//` até o fim da linha é ignorado pelo compilador. Os símbolos `/*` e `*/` dentro de um comentário iniciado por `//` não significam comentário. O mesmo se aplica a `//` entre `/*` e `*/`.

2.2 Tipos e Literais Básicos

Existem apenas três tipos básicos em Simples, `integer`, `boolean` e `String`. Literais do tipo `integer` devem estar entre os limites `0` e `32767`, sendo que qualquer número de zeros antes de um número é permitido. Assim, os números

```
00000000000001
00000000000000
```

são válidos. O tipo `boolean` possui apenas dois valores: `false` e `true`.

Literais do tipo `String` devem aparecer entre aspas: `"Voce vem sempre aqui?"`. A barra `\` pode ser utilizada para retirar o significado de `"` e da própria barra. De fato, a string `"\c"` tem exatamente o mesmo significado que em C, independente do caráter `c`, que pode ser qualquer coisa.

```
and             begin           boolean
break           class           do
else            end             endif
false           if              integer
loop            new             nil
not             or              private
proc            public          read
return          self            String
subclassOf      super           then
true            var             while
write
```

Figure 4: As palavras chave de Simples

Os operadores `<`, `<=`, `>`, `>=`, `==`, `<>` de comparação podem ser aplicados a valores inteiros ou booleanos sendo que `false < true`. Naturalmente, ambos os operados de uma destas operações devem ser do mesmo tipo.

Os operadores `==` e `<>` podem também ser utilizados para comparar expressões cujo tipos são classes. O tipo do resultado é booleano, naturalmente. Neste caso, não é necessário existir nenhum relacionamento entre os tipos das expressões.

Os operadores `+`, `*`, `-` e `/` aplicam-se a valores inteiros resultando em inteiros. A semântica destes operadores é a mesma dos operadores da linguagem C.

Os operadores binários `and` e `or` e o unário `not` aceitam operandos booleanos e possuem o significado usual. A avaliação da expressão

```
expr1 and expr2
```

começa em `expr1`. Se `expr1` for `false`, toda a expressão será considerada falsa, mesmo sem o cálculo de `expr2`. Se for `true`, `expr2` será avaliada.

A expressão

```
expr1 or expr2
```

será considerada `true` se `expr1` for `true`. Caso contrário, esta expressão terá o valor de `expr2`.

O tipo de uma variável/parâmetro ou valor de retorno de um método deve ser um tipo básico ou uma classe. Se for classe, ela deve ter sido declarada previamente, podendo ser a classe corrente.

2.3 Identificadores

Identificadores são formados por letras, dígitos e o caráter sublinhado ("`_`"), iniciando-se por uma letra. Exemplos de identificadores válidos são:

```
getNum   x0   y1   get_Num
```

Os identificadores

```
_main    3ab    get$Num     write
```

são ilegais. "`write`" é ilegal por ser uma palavra chave. A lista das palavras chave da linguagem é exibida na Figura 4.

Os primeiros 31 caracteres de um identificador são significativos e letras maiúsculas e minúsculas são consideradas diferentes. Assim, os identificadores

```
a12345678901234567890123456789012345678 90Um
a12345678901234567890123456789012345678 90Dois
```

são iguais. Todos os identificadores devem ser declarados antes de serem usados e nenhum pode ser declarado duas vezes.

Se a geração de código for feita para a linguagem C, podem ocorrer problemas na translação de nomes longos. Por enquanto, ignoraremos este problema.

2.4 Escopo

O escopo do nome de uma classe é do início da classe até o fim do arquivo.

O escopo de uma variável de instância ou método é do ponto onde foi declarada até o fim da declaração da classe. Contudo, as variáveis de instância e métodos devem ser manipulados pela palavra chave "`self`":

```
self.n = 0;
x = self.m(5);
self.p.set(0);
```

O escopo das variáveis locais e parâmetros de um método é todo o corpo do método. Como identificadores não podem ser redeclarados dentro do mesmo escopo, uma variável local não pode ter o mesmo nome que um parâmetro. Variáveis de instância e locais nunca se confundem porque as primeiras são sempre utilizadas precedidos por "`self`". As variáveis locais possuem precedência sobre os nomes globais de classes.

2.5 Atribuição

A atribuição de uma expressão `expr` a uma variável `aa` é

```
aa = expr
```

Esta instrução será válida se:

- o tipo de `aa` for um dos tipos básicos e igual ao tipo de `expr`;
- o tipo de `expr` for uma classe igual à classe declarada de `aa` ou subclasse desta;
- o tipo de `aa` for uma classe e `expr` for a variável `nil`.

As mesmas regras valem para passagem de parâmetros para métodos e retorno de valores por meio do comando `return` (veja seção 2.8). Isto é, as instruções

```
bb.m(pr);
return r;
```

serão válidas se as atribuições

```
pf = pr;
x = r;
```

estiverem corretas, onde `pf` possui o mesmo tipo que o parâmetro formal do método `m` e `x` é uma variável do mesmo tipo que o valor de retorno do método onde esta instrução está.

2.6 Comandos de Decisão

O comando `if` de Simples possui a forma

```
if expr
then
  StatementList
else
```

```
  StatementList
endif
```

onde a parte

```
else
  StatementList
endif
```

é opcional. `expr` é uma expressão booleana e `StatementList` é uma lista de zero ou mais comandos.

2.7 Comandos de Repetição

O comando

```
while expr do
  Statement;
```

repete `Statement` enquanto a avaliação da expressão booleana `expr` resultar em `true`. Este comando também possui a forma

```
while expr do
  begin
  StatementList
  end
```

onde `StatementList` possui zero ou mais comandos.

O comando `loop` é um laço sem fim, possuindo a seguinte sintaxe:

```
loop
  C1;
  C2;
  ...
  Cn;
end
```

Os comandos Ci são repetidos até que um comando `break` seja executado. O código acima é equivalente a

```
while true do
   begin
   C1;
   ...
   Cn;
   end
```

2.8 Retorno de Valores de Métodos

Um método retorna um valor com o comando

```
return exp;
```

Se o tipo (classe) do valor de retorno `exp` for `U` e o tipo declarado do valor do método for `T`, o tipo `U` deve ser convertível para `T`. Isto é, uma atribuição

```
t = u;
```

deve ser válida, onde os tipos de `t` e `u` são `T` e `U`, respectivamente. Um método sem o tipo do valor de retorno não pode ter nenhum comando `return`.

2.9 Entrada e Saída

A entrada de dados é feita pelo comando `read`:

```
read( IdList );
```

onde `IdList` é uma lista de uma ou mais variáveis do tipo `integer` ou `String`. O comando

```
read( a1, a2, ...  an )
```

é equivalente a

```
read(a1);
read(a2);
...
read(an);
```

onde `read(a)` é equivalente ao código

```
gets(s);
sscanf(s, "%d", &a);
```

em C se `a` for do tipo `integer` ou

```
{
   char _s[512];
   gets(_s);
   a = malloc(strlen(_s) + 1);
   strcpy(a, _s);
}
```

se `a` for do tipo `String`.

Cada variável é lida em uma linha separada da entrada padrão.

O comando

```
write( expr1, expr2, ...  exprn )
```

escreve as expressões na saída padrão, sendo equivalente a

```
write(expr1);
write(expr2);
...
write(exprn);
```

O número `n` de expressões deve ser maior do que zero. O comando `write(expr)` é equivalente ao código

```
printf("%d ", expr);
```

em C se `expr` for do tipo `integer` ou

```
puts(expr)
```

se `expr` for do tipo `String`. Expressões booleanas não podem ser parâmetros de `write`.

3 A Gramática de Simples

Esta seção define a gramática da linguagem. As palavras reservadas e símbolos da linguagem são mostrados entre “ e ”. Qualquer seqüência de símbolos entre { e } pode ser repetida zero ou mais vezes e qualquer seqüência de símbolos entre [e] é opcional. Parênteses agrupam símbolos. Por exemplo,

D ::= (A|B) { C }

significa A ou B seguido de qualquer número de C's. O prefixo Un em um nome significa a união de duas ou mais regras.

O não terminal `StringValue` representa uma string de caracteres entre aspas como em C++/Java: `"Ola !!!"`. Este não terminal **não** está presente na gramática abaixo.

```
Assignment        ::= LeftValue "=" Expression
BasicType         ::= "integer" | "boolean" | "String"
BasicValue        ::= IntValue | BooleanValue | StringValue
Block             ::= "begin" StatementList "end"
BooleanValue      ::= "true" | "false"
ClassDec          ::= "class" Id [ "subclassOf" Id ]
                      UnPubPri "end"
Digit             ::= "0" | ... | "9"
Expression        ::= SimpleExpression [ Relation SimpleExpression ]
ExpressionList    ::= Expression { "," Expression }
Factor            ::= BasicValue | RightValue | MessageSend | "(" Expression ")" |
                      "not" Factor | "nil"
FormalParamDec    ::= ParamDec { ";" ParamDec }
HighOperator      ::= "*" | "/" | "and"
Id                ::= Letter { Letter | Digit | "_" }
IdList            ::= Id { "," Id }
IfStat            ::= "if" Expression "then" StatementList
                      [ "else" StatementList ] "endif"
InstVarDec        ::= "var" InstVarDec2 { InstVarDec2 }
InstVarDec2       ::= IdList ":" Type ";"
IntValue          ::= Digit { Digit }
LeftValue         ::= [ "self" "." ] Id
Letter            ::= "A" | ... | "Z" | "a" | ... | "z"
LocalDec          ::= "var" VarDec { VarDec }
LowOperator       ::= "+" | "-" | "or"
LoopStat          ::= "loop" StatementList "end"
MessageSend       ::= ReceiverMessage "." MethodId "("
                      [ ExpressionList ] ")"
MethodDec         ::= ProcHeading [ LocalDec ] Block
MethodId          ::= Id | "new"
ParamDec          ::= IdList ":" Type
PrivatePart       ::= ( InstVarDec | MethodDec ) { InstVarDec | MethodDec }
ProcHeading       ::= "proc" Id "(" [ FormalParamDec ] ")"
                      [ ":" Type ]
Program           ::= ClassDec { ClassDec }
PublicPart        ::= MethodDec { MethodDec }
ReadStat          ::= "read" "(" LeftValue { "," LeftValue } ")"
ReceiverMessage   ::= "super" | Id | "self" | "self" "." Id
Relation          ::= "==" | "<" | ">" | "<=" | ">=" | "<>"
ReturnStat        ::= "return" Expression
RightValue        ::= "self" [ "." Id ] | Id
Signal            ::= "+" | "-"
SignalFactor      ::= [ Signal ] Factor
SimpleExpression  ::= Term { LowOperator Term }
```

Statement	::= Assignment “;” | IfStat | WhileStat | MessageSend “;” | ReturnStat “;” | ReadStat “;” | WriteStat “;” | LoopStat | “break” “;” | “;”
StatementList	::= { Statement }
Term	::= SignalFactor { HighOperator SignalFactor }
Type	::= BasicType | Id
UnPubPri	::= [“private” “:” PrivatePart] [“public” “:” PublicPart]
UnStatBlock	::= Statement | Block
WriteStat	::= “write” “(” ExpressionList “)”
VarDec	::= IdList “:” Type “;”
WhileStat	::= “while” Expression “do” UnStatBlock

References

[1] Stroustrup, Bjarne. *The C++ Programming Language*. Second Edition, Addison-Wesley, 1991.

[2] Lippman, Stanley B. *C++ Primer*. Addison-Wesley, 1991.

[3] Deitel, H.M. e Deitel P.J. *C++ How to Program*. Prentice-Hall, 1994.

[4] Aho, Alfred e Sethi, Rave e Ullman, Jeffrey. *Compilers: Principles, Techniques and Tools*. Addison-Wesley Publishing Company, 1986.

[5] Gamma, Erich; Helm, Richard; Johnson, Ralph e Vlissides, John. *Design Patterns: Elements of Reusable Object-Oriented Software*. Professional Computing Series, Addison-Wesley, Reading, MA, 1994.

[6] Zorzo, Sérgio Donizetti. Notas de aula de Construção de Compiladores, 1995.

[7] Pittman, Thomas; Peter, James. *The Art of Compiler Design: Theory and Practice*. Prentice Hall, 1992.

[8] Guimarães, José de Oliveira. *The Green Language*. http://www.dc.ufscar.br/ jose/green/green.htm