УДК 519.683.2

Проценко В. С.

ФОРМАЛЬНІ СПЕЦИФІКАЦІЇ ОПЕРАТОРА ПЕРЕХОДУ В МОВАХ ПРОГРАМУВАННЯ

Задаючи мову програмування, необхідно описати 'и синтаксис і семантику. Головне завдання синтаксису — описати всі конструкції, котрі розглядаються як елементи мови. З цією метою використовується конкретний синтаксис, що виділяє послідовності символів алфавіту, які вважаються синтаксично правильними. Найчастіше конкретний синтаксис — це скінченний набір правил, котрі дозволяють породити нескінченну множину всіх конструкцій мови.

Найбільш відомі формалізми задавання конкретного синтаксису — Бекуса-Наура форми (БНФ), контекстно вільні граматики, синтаксичні схеми. Конкретний синтаксис у вигляді синтаксичних схем складається з правил вигляду

$$A = B,$$

де А — ім'я синтаксичної конструкції, а В — послідовність, що її визначає. У *B* можуть входити символи алфавіту, імена синтаксичних конструкцій та метасимволи вигляду '|', '[', ']'. Метасимвол '|' відділяє можливі варіанти запису конструкції, а запис ...[c]... означає, що послідовність символів c або входить, або не входить в конструкцію.

З точки зору семантики ("значення") синтаксичної конструкції конкретний синтаксис, як правило, містить "надлишок" інформації: порядок наступності її компонент; використання розділових символів ';','(',')'; пріоритет і асоціативність операцій. Тому у випадку, коли необхідно коротко задати лише структуру синтаксичних конструкцій, а не їх зображення рядками символів алфавіту, віддають перевагу абстракному синтаксису. Інформації, котру задає абстракний синтаксис мови програмування, достатньо для формального опису її семантики. Для реальних мов програмування абстракний синтаксис значно коротший і більш наочний, ніж конкретний.

З кожною конструкцією мови програмування зв'язується деякий абстрактний об'єкт (множина, таблиця, функція і т. п.) — її денотат, котрий задає "значення" (семантику) конструкції. Денотати — це класи об'єктів, в термінах яких пояснюються всі конструкції мови програмування [1]. Головна задача денотаційної семантики для конкретної мови програмування — знайти математичні об'єкти, котрі будуть денотатами. Вдалий їх вибір дозволяє просто визначити семантику всіх конструкцій мови. При цьому необхідно, щоб виконувалося ДЕНОТАЦІЙНЕ правило:

Денотат складеної конструкції мови повинен визначатися лише в термінах денотатів її компонент.

Як приклад розглянемо найпростішу мову програмування, яка включає лише цілі вирази. Цілі вирази складаються з цілих чисел, знаків бінарних операцій — додавання '+', віднімання '-', множення '*', ціле ділення '/' та круглих дужок '(' і ')'. Конкретний синтаксис цієї мови у вигляді схем включає конструкції — вираз є, доданок t і множник f:

```
e == t | e + t | e – t,
t == f | t * f | t /f,
f == INTG | ( e ).
```

Ці схеми враховують пріоритет операцій '+', '–' і '*', '/' та їх ліву асоціативність; ***INTG*** позначає довільне ціле без знака.

Абстрактний синтаксис містить класи: вирази *Expr*, застосування бінарної операції *Infix*, операції *Op* і цілі значення ***INTG***:

```
Expr = Infix | Id | INTG,
Infix :: Expr Op Expr,
Op = '+' | '–' | '*' | '/'.
```

Область денотатів для цієї мови є множина цілих чисел ***INTG***. Значення мови задає функція *e-expr*, котра використовує допоміжну функцію *apply-bo*:

```
type: e-expr : Expr                    → INTG
      apply-bo : INTG Op INTG → INTG
e-expr(e) =
cases e:
(mk-Infix(e1,o,e2) → (let v1 = e-expr(e1)(s) in
                      let v2 = e-expr(e2)(s) in
                      apply-bo(v1,o,v2)
                      ),
T                  → e
)
apply-bo(v1, o, v2) =
(o = '+' → v1 + v2,
```

© ПроценкоВ. С.,2000

o = '–' → $v1 - v2$,
o = '*' → $v1 * v2$,
o = '/' → ***if*** $v2$!= 0 ***then*** $v1 / v2$ ***else*** 0
).

Розглянемо цілий вираз 5 + 4 * 3, абстрактний синтаксис якого має вигляд:

mk-*Infix*(5, '+', ***mk***-*Infix*(4, '*', 3)).

Функція *e-expr*(5 + 4 * 3) вираховує його денотат наступним чином:

e-expr(5 + 4 * 3) =
(***let*** *v1* = *e-expr*(5) ***in***
let *v2* = *e-expr*(4 * 3) ***in***
apply-bo(*v1*, '+', *v2*)
) =
(***let*** *v1* = 5 ***in***
let *v2* = (***let*** *v1* = *e-expr*(4) ***in***
  let *v2* = *e-expr*(3) ***in***
  apply-bo(*v1*, '*', *v2*)
) ***in***
v1 + *v2*
) =
(***let*** *v2* = (***let*** *v1* = 4 ***in***
let *v2* = 3 ***in***
v1 * *v2*
) ***in***
5 + *v2*
) =
(***let*** *v2* = 4 * 3 ***in***
5 + *v2*
) =
(***let*** *v2* = 12 ***in***
5 + *v2*
) =
5 + 12 = 17.

Розглянемо розширення цієї простої мови, додаючи цілі скалярні змінні та оператори присвоювання, умовний та складений. Наступні схеми описують конкретний синтаксис конструкцій мови: p — програма, il — список ідентифікаторів, sl — список операторів, s — оператор, e — вираз, t — доданок і f — множник:

p == (**dcl** il; **in** id; **out** id; s)
il == id | il, id
sl == s | sl; s
s == id := e | **if** e **then** s **else** s | (sl)
e == t | e + t | e — t
t == f | t * f | t / f
f == **INTG** | id | (e)

id і **INTG** позначають ідентифікатор і ціле без знаку.

Кожна програма (**dcl** il; **in** iv; **out** ov; s) має на вході деяке значення v, яке на початку виконання програми стає початковим значенням змінної iv, і по закінченні виконання програми (оператора s) виводить значення змінної ov.

Абстрактний синтаксис вводить класи: програма (*Program*), оператор (*Stmt*), складений оператор (*Comp*), оператор присвоювання (*Assign*), умовний оператор (*If*) і ідентифікатор (*Id*).

Program :: *Id-set s-in:Id s-out:Id Stmt*
Stmt = *Comp* | *Assign* | *If*
Comp :: *Stmt*+
Assign :: *Id Expr*
If :: *Expr Stmt Stmt*
Expr = *Infix* | *Id* | ***INTG***
Infix :: *Expr Op Expr*
Op = '+' | '–' | '*' | '/'
Id :: ***TOKEN***.

Для того, щоб визначити денотати операторів і виразів, не порушуючи денотаційного правила, вводиться семантична область станів *St*. Кожний стан *s* @ *St* — це частково визначена функція (таблиця) з ідентифікаторів змінних *Id* в множину значень ***INTG***, яких вони можуть набути. Стан *s* можна розглядати як таблицю, входи в яку — це ідентифікатори змінних, а зв'язані з ними виходи — їх значення:

St = *Id* – *m* → ***INTG***.

Денотати операторів — це функції зміни стану (об'єкти із області *St* → *St*), а денотати виразів — функції із області *St* → ***INTG***. Денотат програми — це функція із області ***INTG*** → ***INTG***. Ці денотати будуються семантичними функціями: *i-program, i-stmt, i-comp, i-assign, i-if, i-stmt-list*, а також функцією *e-expr*:

type: *i-program* : *Program* → (***INTG*** → ***INTG***)
  i-stmt : *Stmt* → (*St* → *St*)
  i-comp : *Comp* → (*St* → *St*)
  i-assign : *Assign* → (*St* → *St*)
  i-if : *If* → (*St* → *St*)
  i-stmt-list : *Stmt* + N_1 → (*St* → *St*)
  e-expr : *Expr* → (*St* → ***INTG***)
  apply-bo : ***INTG*** *Op* ***INTG*** → ***INTG***
i-program(***mk***-*Program*(*ids, in, out, st*))(*v*) =
  (***let*** *s1* = [*id* → 0 | *id* @ *ids*] ***in***
  let *s2* = *i-stmt*(*st*)(*s1* + [*in* → v]) ***in***
  s2(*out*)
)
i-stmt(st)(s) =
  (***is***-*Comp*(*st*) → *i-comp*(*st*)(*s*),
  is-*Assign*(*st*) → *i-assign*(*st*)(*s*),
  is-*If*(*st*) → *i-if*(*st*)(*s*),
)
i-comp(***mk***-*Comp*(*sl*))(*s*) =
  i-stmt-list(*sl*, 1)(*s*)
i-assign(***mk***-*Assign*(*id, e*))(*s*) =
  (***let*** *v* = *e-expr*(*e*)(*s*) ***in***
  s + [*id* → *v*]
)

```
i-if(mk-If(e, s1, s2))(s) =
        (let v = e-expr(e)(s) in
        if v > 0  then i-stmt(s1)(s)
        else i-stmt(s2)(s)
        )
i-stmt-list(sl,i)(s) =
    if i = len sl then i-stmt(sl[i])(s) else
        (let s1 = i-stmt(sl[i])(s) in
        i-stmt-list(sl,i+1)(s1)
        )
e-expr(e)(s) =
    cases e:
    (mk-Infix(e1, o, e2) → (let v1 = e-expr(e1)(s) in
                             let v2 = e-expr(e2)(s) in
                             apply-bo(v1, o, v2)
                             ),
    mk-Id(id) → s(id),
    T → e
    )
apply-bo(v1, o, v2) = ...
```

Для демонстрації використання семантичних функцій вирахуємо денотат наступної програми:

```
mk-Program({a,b}, a, a,
    mk-Compl( < mk-Assign(b, mk-Infix(a, "+", 1)),
    mk-If(a,
    mk-Assign(b, mk-Infix(a, "+", 5)),
    mk-Comp( < mk-Assign(a, mk-Infix(b, "*", a)),
               mk-Assign(b, mk-Infix(b,"-",1))
             >)
    )
             >)
    ).
```

Якщо записати цю програму неформально, використовуючи традиційний синтаксис, то отримаємо:

```
(dcl a, b; in a; out a;
    (b := a + 5; if a then b := a + 5
                      else (a := b * a; b := b – 1)
    )
).
```

При використанні семантичних функцій їх аргументи — програма або окремі її елементи — відомі, тому можна частково виконати ці функції в тих місцях, котрі залежать тільки від синтаксичних конструкцій, а також використовуючи всюди, де це можливо, синтаксичну підстановку.

Нижче наводяться результати застосування цих функцій до окремих елементів програми, а також послідовність обрахування семантичної функції *e-program*:

```
e-expr(5)(s) = 5
e-expr(a)(s) = s(a)
e-expr(a + 5)(s) = s(a) + 5
e-expr(b * a)(s) = s(b) * s(a)
e-expr(b – 1)(s) = s(b) – 1
i-assign(b := a + 5)(s) = s + [b → s(a) + 5]
i-assign(a := b * a)(s) = s + [a → s(b) * s(a)]
i-assign(b := b – 1)(s) = s + [b → s(b) – 1]
i-stmt-list(< a := b * a, b := b – 1 >, 1)(s) =
    s + [a → s(b) * s(a), b → s(b) – 1]
i-if(if a then b := a + 5 else (a := b * a; b := b – 1))(s) =
    if s(a) > 0 then s + [b → s(a) + 5]
    else s + [a → s(b) * s(a), b → s(b) – 1]
i-stmt-list(< b := a + 5, if a then b := a + 5
    else (a := b * a; b := b – 1) >, 1)(s) =
    if s(a) > 0
    then s + [b → s(a) + 5]
    else s + [a → (s(a) + 5) * s(a), b → (s(a) + 5) – 1]
```

Нехай

```
sl = < b := a + 5, if a then b := a + 5
                   else (a := b * a; b := b – 1) >.
i-program(mk-Program({a,b}, a, a, sl))(v) =
(let s1 = [a → v, b → 0] in
let s2 = i-stmt-list(sl, 1)(s1) in
s2(a)
) =
(let s1 = [a → v, b → 0] in
let s2 = (if s1(a) > 0
then s1 + [b → s1(a) + 5]
else s1 + [a → (s1(a) + 5) * s1(a), b → (s1(a) + 5) – 1])
in
s2(a)
) =
(let s2 = if v > 0 then [a → v, b → a + 5]
else [a → (v + 5) * v], b → (v + 5) — 1] in
s2(a)
) =
if v > 0 then v else (v + 5) * v
```

Розширимо цю мову, додаючи оператор переходу та поняття мітки. Вважаємо, що мітка l може з'явитися перед довільним оператором в списку, що складає тіло складеного оператора. Перехід на таку мітку l може викликати оператор переходу goto l, який або безпосередньо входить у список операторів, або розташований в середині одного з них.

У конкретному синтаксисі змінюються конструкції: sl – список операторів і s — оператор:

sl == [id:] s | sl; [id:] s

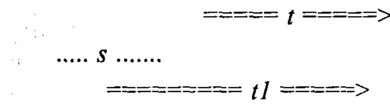
s == id := e | **if** e **then** s **else** s | (sl) | **goto** id.

В абстрактному синтаксисі змінюються класи: оператор (*Stmt*) та складений (*Comp*) і з'являються нові класи — помічений оператор (*Nmst*) та оператор переходу (*Goto*):

```
Stmt   = Comp | Assign | If | Goto
Comp   :: Nmst +
Nmst   :: s-n: [Id] s-b: Stmt
Goto   :: Id.
```

Функцій зміни стану (об'єкти області $St \to St$) вже не досить для того, щоб описати ефект оператора переходу **goto** l — передачі управління через

структуру — послідовність операторів. Для того, щоб описати семантику операторів, використовують продовження (*continuations*) [2] — функції зміни стану — об'єкти з області *Tr*. Денотат оператора тоді є об'єкт із області *Tr* → *Tr*. З кожною міткою m зв'язується деяке продовження — функція зміни стану від мітки m і до кінця програми. А денотат оператора переходу **goto** m — для довільного продовження — це є продовження, зв'язане з міткою m. Нехай денотат деякого оператора s із програми є функція *d @ Tr* → *Tr*. Ця функція для довільного продовження *t @ Tr*, що йде за оператором s — це функція зміни стану, зв'язана з виконанням усіх операторів до кінця програми, задає загальне продовження *t1*, тобто функцію зміни стану, яку задає оператор s і продовження *t*. Ця ситуація графічно показана на малюнку нижче, причому t1 = d(t):



Через те, що s може бути оператором перехода **goto** m, то в результаті цю функцію *t1* не можна задати просто як композицію двох функцій зміни стану (друга з яких t).

Для зв'язування (асоціації) мітки з її продовженням, яке застосовується при знаходженні денотата оператора переходу, використовується середовище (*Env*):

Env = *Id-m* → *Tr*
Tr = *St* → *St*

type: *i-program : Program* → (***INTG*** → ***INTG***)
i-stmt : Stmt → (*Env* → (*Tr* → *Tr*))
i-comp : Comp → (*Env* → (*Tr* → *Tr*))
i-assign : Assign → (*Env* → (*Tr* → *Tr*))
i-if : If → (*Env* → (*Tr* → *Tr*))
*i-stmt-list : Stmt** N_1 → (*Env* → (*Tr* → *Tr*))
i-goto : Goto → (*Env* → (*Tr* → *Tr*))
e-expr : Expr → (*St* → ***INTG***)
apply-bo : ***INTG*** *Op* ***INTG*** → ***INTG***

i-program(***mk****-Program*(*ids, in, out, st*))(*v*) =
(***let*** *s1* = [*id* → 0 | *id @ ids*] ***in***
let *Ft* = *i-stmt*(*st*)([ ]) ***in***
let *fs* = *Ft*(*!s.s*) ***in***
let *s2* = *fs*(*s1* + [*in* → v]) ***in***
s2(*out*)
)
i-stmt(*st*)(*env*){*t*} =
(***is****-Comp*(*st*) → *i-comp*(*st*)(*env*){*t*},
is*-Assign*(*st*) → *i-assign*(*st*)(*env*){*t*},
is*-If*(*st*) → *i-if*(*st*)(*env*){*t*},
is*-Go*(*st*) → *i-go*(*st*)(*env*){*t*}
)
i-comp(***mk****-Comp*(*nsl*))(*env*){t} =
(***let*** *nenv* = *env* + [*s-n*(*nsl*[*i*]) →
→ *i-stmt-list*(*nsl, i*)(*nenv*){*t*}
| *i @* {*1* ... ***len*** *nsl*} & *s-n*(*nsl*[*i*] != ***nil***] ***in***
i-stmt-list(*nsl*, 1)(*nenv*){*t*}
)
i-assign(*mk-Assign*(*id, e*))(*env*){*t*} =
!s.(***let*** *v* = *e-expr*(*e*)(*s*) ***in***
t(*s* + [*id* → *v*])
)
i-if(***mk****-If*(*e, s1, s2*))(*env*){*t*} =
!s.(***let*** *v* = *e-expr*(*e*)(*s*) ***in***
if *v* > 0 ***then*** *i-stmt*(*s1*)(*env*){*t*}(*s*)
else *i-stmt*(*s2*)(*env*){*t*}(*s*)
)
i-stmt-list(*nsl, i*)(*env*){*t*} =
(***let*** *mk-Nmst*(*id, st*) = *nsl*[*i*] ***in***
if *i* = ***len*** *nsl* ***then*** *i-stmt*(*st*)(*env*){*t*} ***else***
(***let*** *t1* = *i-stmt-list*(*nsl, i* + 1)(*env*){*t*} ***in***
i-stmt(*st*)(*env*){*t1*}
)
)
i-goto(*mk-Goto*(*id*))(*env*){*t*} =
env(*id*)
e-expr(*e*)(*s*) = ...
apply-bo(*v1, o, v2*) = ...

Семантичні функції *e-expr*(*e*) і *apply-bo*(*v1, o, v2*) не змінилися, оскільки семантика виразу не змінилася, і тому тут не повторюються.

Для демонстрації використання семантичних функцій вирахуємо денотат наступної програми, записаної неформально, використовуючи традиційний синтаксис:

(**dcl** a,b; **in** a; **out** a;
(**if** a **then** b := a **else goto** l2;
l1: a := a – 1;
if a **then** (b := b * a; **goto** l2)
else goto l3;
l2: b := 1;
l3: a := b
)
).

Ця програма на вході v <= 0 обчислює 1, а для v > 0 обчислює v! (факторіал v).

Припустимо, що
s @ St, a @ dom s, b @ dom s і *t @ Tr*.
i-assign(b := b * a)(*env*){*t*} = *!s.t*(*s* + [*b* → *s*(*b*) * *s*(*a*)])
i-assign(a := a – 1)(*env*){*t*} = *!s.t*(*s* + [*a* → *s*(*a*) – 1])
i-assign(b := a)(*env*){*t*} = *!s.t*(*s* + [*b* → *s*(*a*)])
i-assign(b := 1)(*env*){*t*} = *!s.t*(*s* + [*b* → 1])
i-assign(a := b)(*env*){*t*} = *!s.t*(*s* + [*a* → *s*(*b*)])
i-stmt-list(< b := 1; a := b >,*1*)(*env*){*t*} =
!s.t((*s* + [*b* → 1, *a* → 1]).

Припустимо, що продовження *tl1, tl2* і *tl3* з *Tr* є денотати міток *l1, l2* і *l3*, при цьому *env* = [*l1* → *tl1*, *l2* → *tl2*, *l3* → *tl3*, ...].

```
i-stmt-list((b := b * a; goto l1),1)(env){t} =
!s.tl1(s + [b → s(b) * s(a)])
i-if(if a then (b := b * a; goto l1)
         else goto l3)(env){t} =
!s.(if s(a) > 0 then tl1(s + [b → s(b) * s(a)]) else tl3(s))
i-if(if a then b := a else goto l2)(env){t} =
!s.(if s(a) > 0 then t(s + [b → s(a)]) else tl2(s))
i-stmt-list((if a then (b := b * a; goto l1) else goto l3;
b := 1; a := b),1)(env){t} =
!s.(if s(a) > 0 then tl1(s + [b → s(b) * s(a)]) else tl3(s))
i-stmt-list((a := a – 1;
if a then (b := b * a; goto l1) else goto l3;
b := 1; a := b),1)(env){t} =
!s. if s(a) – 1 > 0
then tl1(s + [a → s(a) – 1,b → s(b) * (s(a) – 1)])
else tl3(s + [a → s(a) – 1])
i-stmt-list(if a then b := a else goto l2; a := a – 1;
if a then (b := b * a; goto l1) else goto l3;
b := 1; a := b),1)(env){t} =
!s.(if s(a) > 0 then
if s(a) – 1 > 0
then tl1(s + [a → s(a) – 1, b → s(a) * (s(a) – 1)])
else tl3(s + [a → s(a) – 1, b → s(a)])
else tl2(s)
).
```

Нехай

```
cmp =   (if a then b := a else goto l2; l1:a := a – 1;
         if a then (b := b * a; goto l1) else goto l3;
         l2:b := 1; l3:a := b
         ),
```

тоді

```
i-comp(cmp)(env){t} =
(let nenv = env + [l1 → i-stmt-list(sl4, 2)(nenv){t},
     l2 → i-stmt-list(sl4,4)(nenv){t},
     l3 → i-stmt-list(sl4,5)(nenv){t}] in
i-stmt-list(sl4,1)(nenv){t}
) =
(let tl1 = !s.if s(a) – 1 > 0
then tl1(s + [a → s(a) – 1, b → s(b) * (s(a) – 1)])
else tl3(s + [a → s(a) – 1]) in
let tl2 = !s.t((s + [b → 1, a → 1]) in
let tl3 = !s.t(s + [a → s(b)]) in
!s.(if s(a) > 0 then
if s(a) – 1 > 0
then tl1(s + [a → s(a) – 1, b → s(a) * (s(a) – 1)])
else tl3(s + [a → s(a) – 1, b → s(a)])
else tl2(s)
)
).
```

Нехай програма pr = (**dcl** a,b; **in** a; **out** a; cmp), тоді

```
i-program(pr)(v) =
(let s1 = [a → 0, b → 0] in
let Ft = i-stmt(cmp)([]) in
let fs = Ft(!s.s) in
let s2 = fs(s1 + [a → v]) in
s2(a)
) =
(let s2 = (let tl1 = !s.if s(a) – 1 > 0
then tl1(s + [a → s(a) – 1, b → s(b) * (s(a) – 1)])
else tl3(s + [a → s(a) – 1]) in
let tl2 = !s.t((s + [b → 1, a → 1]) in
let tl3 = !s.t(s + [a → s(b)]) in
if v > 0 then
if v – 1 > 0
then tl1([a → v – 1, b → v * (v – 1)])
else tl3([a → v – 1, b → v])
else tl2([a → v, b → 0])
) in
s2(a)
).
```


1. *Процент В. С, Чаленко П. Й.* Формальні специфікації мов програмування. Навч. пос.— К.: Либідь, 1994.— 184 с.

2. *Bjorner D.* Experiments in Block-structured GOTO language modeling: Exits versus Continuations // Lecture Notes in Computer Sciences.— 1980.— Vol. 86.— P. 216—247.

Protsenko V. S.

THE FORMAL SPECIFICATIONS OF THE OPERATOR GOTO IN THE PROGRAMMING LANGUAGES

Assigning a language of programming it is necessary to describe its syntax and semantics. The main task of syntax is to describe all constructions, which are considering as language elements. A special place among the main operators of programming languages belongs to goto operators. In the work described the formal specifications of the goto operators.