ТЕОРІЯ КОМПІЛЯТОРІВ

Лекція 8

САМОВИЗНАЧЕННЯ СЕМАНТИКИ МОВ ПРОГРАМУВАННЯ

2020

Повний текст лекції буде розміщений на сайті baklaniv.at.ua

Рейтінг THIOBE

30	Kotlin
31	Logo
32	Lisp
33	F#
34	Fortran
35	Lua
36	Ada
37	Prolog
38	VBScript
39	LabVIEW
40	Apex
41	Haskell
42	TypeScript
43	PowerShell
44	ML
45	RPG
46	Scheme
47	Erlang
48	Groovy
49	Bash
50	Julia

Apr 2020	Apr 2019	Change	Programming Language
1	1		Java
2	2		C
3	4	^	Python
4	3	v	C++
5	6	^	C#
6	5	v	Visual Basic
7	7		JavaScript
8	9	^	PHP
9	8	v	SQL
10	16	^^	R
11	19	^^	Swift
12	18	^^	Go
13	13		Ruby
14	10	vv	Assembly language
15	22	^^	PL/SQL
16	14	v	Perl
17	11	vv	Objective-C
18	12	vv	MATLAB
19	17	v	Classic Visual Basic
20	27	^^	Scratch

Position	Programming Language
21	SAS
22	Delphi
23	Dart
24	Transact-SQL
25	D
26	COBOL
27	Rust
28	Scala
29	ABAP

Самовизначення LISP

```
(define (concat lst1 lst2)
    (cond   ((null? lst1) lst2)
                    (#t (cons (car lst1) (concat (cdr lst1) lst2)))))
```

List Operations

(car <list>)	return the first item in <list>
(cdr <list>)	return <list> with the first item removed
(cons <item> <list>)	add <item> as first element of <list>

Arithmetic Operations

(+ <e_1> <e_2>)	return sum of the values of <e_1> and <e_2>
(- <e_1> <e_2>)	return difference of the values of <e_1> and <e_2>
(* <e_1> <e_2>)	return product of the values of <e_1> and <e_2>
(/ <e_1> <e_2>)	return quotient of the values of <e_1> and <e_2>

Predicates

(null? <list>)	test if <list> is empty
(equal? <s_1> <s_2>)	test the equality of S-expressions <s_1> and <s_2>
(atom? <s>)	test if <s> is an atom

Function Definition and Anonymous Functions

(define (<name> <formals>) <body>)	allow user to define function <name> with formal parameters <formals> and function body <body>
(lambda (<formals>) <body>)	create an anonymous function
(let (<var-bindings>) <body>)	an alternative to function application; <var-bindings> is a list of (variable S-expression) pairs and the body is a list of S-expressions; let returns the value of last S-expression in <body>

Other

(quote <item>)	return <item> without evaluating it
(display <expr>)	print the value of <expr> and return that value
(newline)	print a carriage return and return ( )

```
(define (replace s r lst)
    (cond   ((null? lst) lst)
            ((equal? (car lst) s) (cons r (replace s r (cdr lst))))
            (#t (cons (car lst) (replace s r (cdr lst))))))
```

```
(define (fibonacci n)
    (cond   ((equal? n 0) 1)
            ((equal? n 1) 1)
            (#t (+(fibonacci (- n 1)) (fibonacci (- n 2))))))

(define (factorial n)
    (cond   ((equal? n 0) 1)
            (#t (* n (factorial (- n 1))))))
```

```
(define (micro-rep env)
    (let ((prompt (display ">> ")) (s (read)))
        (if   (equal? s 'quit)
            (begin (newline) (display "Goodbye") (newline))
            (cond
                ((atom? s)  (begin   (newline)
                                     (display (micro-eval s env))
                                     (newline)
                                     (micro-rep env)))
                ((equal? (car s) 'define)
                            (let ((newenv (updateEnv env
                                                (caadr s)
                                                (list 'lambda (cdadr s) (caddr s)))))
                                    (begin   (newline)
                                             (display (caadr s))
                                             (newline)
                                             (micro-rep newenv))))
```

```
(#t  (begin  (newline)
             (display (micro-eval s env))
             (newline)
             (micro-rep env)))))))
```

```
(define (applyEnv ide env) (cadr (assoc ide env)))
```

```
(define (micro-eval s env)
    (cond ((atom? s)
                (cond  ((equal? s #t) #t)
                       ((equal? s #f) #f)
                       ((number? s) s)
                       (else (applyEnv s env))))
          ((equal? (car s) 'quote) (cadr s))
          ((equal? (car s) 'lambda) s)
          ((equal? (car s) 'display)
                (let ((expr-value (micro-eval (cadr s) env)))
                       (display expr-value) expr-value))
          ((equal? (car s) 'newline) (begin (newline) '( )))
          ((equal? (car s) 'cond) (micro-evalcond (cdr s) env))
          ((equal? (car s) 'let)
                (micro-evallet (cddr s) (micro-let-bind (cadr s) env)))
          (else (micro-apply  (car s)
                              (map (lambda (x) (micro-eval x env )) (cdr s))
                              env))))
```

Самовизначення ПРОЛОГА

We first build a very simple meta-interpreter in Prolog that handles only the conjunction of goals and the chaining goals. A goal succeeds for one of three reasons:

1. The goal is true.
2. The goal is a conjunction and both conjuncts are true.
3. The goal is the head of a clause whose body is true.

```
prove(true).
prove((Goal1, Goal2)) :- prove(Goal1), prove(Goal2).
prove(Goal) :- clause(Goal, Body), prove(Body).
prove(Goal) :- fail.
```

```
memb(X,[X|Rest]).
memb(X,[Y|Rest]) :- memb(X,Rest).
```

```
:- prove((memb(X,[a,b,c]),memb(X,[b,c,d]))).
X = b ;   % semicolon requests the next answer, if any
X = c ;
no
:- prove((memb(X,[a,b,c]),memb(X,[d,e,f]))).
no
:- prove(((memb(X,[a,b,c]),memb(X,[b,c,d])),memb(X,[c,d,e]))).
X = c ;
no
```

Наступна лекція, яка відбудеться у наступну п’ятницю, присвячена трансляційним семантикам і атрибутним граматикам, а також традиційному операційному методу опису семантики мови програмування