МУЛЬТИПАРАДИГМЕННЕ ПРОГРАМУВАННЯ

Лекція 16

Мультипарадигмена мова програмування Oz

Мова програмування Oz - логічна, програмування з обмеженнями, функціональна (як ледачі, так і «енергійні» обчислення), процедурна (імперативна), об'єктно-орієнтована, розподілена, паралельна.

Ця лекція знайомить з мовою програмування Oz та системою програмування Моцарт. Oz - мова з декількома парадигмами, призначена для вдосконалених, одночасних, мережевих, програмних програм реального часу та реактивних програм.

Oz надає основні можливості об’єктно-орієнтованого програмування, включаючи стан, абстрактні типи даних, об’єкти, класи та успадкування. Він забезпечує основні особливості функціонального програмування, включаючи композиційний синтаксис, першокласні процедури / функції та лексичний обсяг. Він забезпечує основні особливості логічного програмування та програмування обмежень, включаючи логічні змінні, обмеження, конструкції диз'юнкції та програмовані механізми пошуку. Це дозволяє користувачам динамічно створювати будь-яку кількість послідовних потоків. Потоки є потоками потоку даних у тому сенсі, що потік, що виконує операцію, буде призупинений, поки всі необхідні операнди не отримають чітко визначене значення.

Система програмування Моцарта була розроблена дослідниками з *DFKI (Німецький дослідницький центр штучного інтелекту)*, *SICS (Шведський інститут комп'ютерних наук)*, *Університет Саару, UCL (Université catholique de Louvain)* та ін.

Система Моцарта реалізує Oz 3, останню з сімейства Oz з багатьма парадигмами, засновану на моделі одночасних обмежень. Oz 3 майже повністю сумісний зі своїм попередником Oz 2. Основними доповненнями до Oz 2 є функтори (різновид програмного компонента) та ф'ючерси (для поліпшення поведінки потоків даних). Oz 2 сам по собі є спадкоємцем оригінальної мови Oz 1, реалізація якої вперше була оприлюднена в 1995 році. Якщо не зазначено інше, всі посилання на Oz у документації Моцарта стосуються Oz 3.

Oz 3 і система Моцарта були розроблені головним чином дослідницькими групами Герта Смолки при DFKI (Німецький дослідницький центр штучного інтелекту), Сейфа Харіді з SICS (Шведський інститут комп'ютерних наук) та Пітера Ван Роя з UCL (Університетський собор Лувена).

В основі всіх версій Oz лежить паралельна модель програмування з обмеженнями, розширена для підтримки обчислень із станом, тобто обчислень на змінних об'єктах. Оригінальна модель обчислення Oz, Oz 1, підтримує чітке поняття паралельності, де кожен вираз може бути виконаний одночасно. В результаті виходить дрібнозерниста модель, схожа на модель актора. Хороший виклад моделі програмування Oz 1 наведено в [Smo95]. Наш досвід використання Oz 1 показав, що цей тип моделі, хоча і теоретично привабливий, ускладнює програміст управління ресурсами своєї програми. Крім того, дуже важко налагоджувати програми, і об'єктна модель стає надмірно незручною.

Oz 2 виправляє ці проблеми, використовуючи замість них потокову модель паралельності з явним створенням потоків. Була розроблена потужна нова об'єктна система та додані традиційні конструкції обробки винятків. Крім того, значно розширено можливості вирішення обмежень та можливості пошуку.

Oz 3 консервативно розширює Oz 2 двома поняттями, функторами та ф’ючерсами, а також виправляє кілька незначних синтаксичних проблем. Функтор - це свого роду програмний компонент. Він визначає модуль з точки зору інших необхідних йому модулів. Це підтримує додаткову побудову програм із компонентів, які можуть бути адресовані через Інтернет за допомогою URL-адрес.
Майбутнє - це логічна змінна, яку можна читати, але не писати. Це дозволяє безпечно синхронізувати потік даних через Інтернет.

Система Mozart підтримує розподілені та мережеві програми. Можна підключити обчислення Oz, розташовані на різних сайтах, що призводить до єдиного прозорого обчислення мережі. Mozart підтримує автоматичну передачу даних та коду без громадянства між сайтами, мобільні обчислення (об'єкти), передачу повідомлень, спільні логічні змінні та ортогональні механізми для виявлення несправностей та обробки для мережі та для сайтів.

Гарне запитання, чому Oz. Ну, одна коротка відповідь полягає в тому, що, порівняно з іншими існуючими мовами, це магія! Він надає програмістам та розробникам систем широкий спектр програмних абстракцій, що дозволяє швидко та надійно розробляти складні програми. Oz об’єднує кілька напрямків проектування мови програмування в єдиний цілісний дизайн. Більшість з нас знають переваги різних парадигм програмування, будь то об'єктно-орієнтоване, функціональне або обмежувальне логічне програмування. Коли ми починаємо писати програми будь-якою існуючою мовою, ми швидко опиняємося в обмеженості поняттями основної парадигми. Oz вирішує цю проблему узгодженим дизайном, який поєднує програмні абстракції різних парадигм чітко і просто.

Отже, перед тим, як відповісти на вищезазначене питання, давайте подивимось, що таке Oz. Це знову важке запитання, на яке можна відповісти кількома реченнями. Отже, ось перший знімок. Це мова програмування високого рівня, розроблена для сучасних передових, одночасних, інтелектуальних, мережевих, програмних програм реального часу, паралельних, інтерактивних та активних програм. Як бачите, досі важко зрозуміти, що означає весь цей жаргон. Більш конкретно:

Oz поєднує в собі основні риси об'єктно-орієнтованого програмування, надаючи стан, абстрактні типи даних, класи, об'єкти та успадкування.

Oz надає основні риси функціонального програмування, надаючи композиційний синтаксис, першокласні процедури та лексичний обсяг. Насправді кожна сутність Oz є першим класом, включаючи процедури, потоки, класи, методи та об'єкти.

Oz надає основні можливості логічного програмування та програмування обмежень, надаючи логічні змінні, диз'юнктивні конструкції та програмовані стратегії пошуку.

Oz - паралельна мова, де користувачі можуть динамічно створювати будь-яку кількість послідовних потоків, які можуть взаємодіяти один з одним. Однак, на відміну від звичайних одночасних мов, кожен потік Oz є потоком даних. Виконання оператора в Oz триває лише тоді, коли всі реальні залежності потоку даних від задіяних змінних вирішені.

Система Моцарта підтримує прозорий в мережі розподіл обчислень Oz. Кілька сайтів Oz можуть з’єднуватися разом і автоматично поводитись як одне обчислення Oz, обмінюючись змінними, об’єктами, класами та процедурами. Сайти відключаються автоматично, коли посилання між об’єктами на різних сайтах перестають існувати.

У розподіленому середовищі Oz забезпечує мовну безпеку. Тобто всі мовні сутності створюються та передаються явно. Додаток не може підробляти посилання або посилання на доступ, які йому явно не надані. Основне представлення мовних сутностей недоступне програмісту. Це наслідок наявності абстрактного сховища та лексичного обсягу. Поряд із першокласними процедурами, ці концепції є важливими для реалізації політики безпеки на основі можливостей, що важливо у відкритих розподілених обчисленнях.

Ядро мови

Цей розділ дає короткий, але точний вступ до мови ядра Oz. Повну мову Oz можна розглядати як синтаксичний цукор для мови невеликого ядра. Мова ядра представляє важливу частину мови.

```
<Statement>  ::= <Statement1> <Statement2>
               |  X = f(l1:Y1 ... ln:Yn)
               |  X = <number>
               |  X = <atom>
               |  X = <boolean>
               |  {NewName X}
               |  X = Y
               |  local X1 ... Xn in S1 end
               |  proc {X Y1 ... Yn} S1 end
               |  {X Y1 ... Yn}
               |  {NewCell Y X}
               |  Y=@X
               |  X:=Y
```

```
|   {Exchange X Y Z}
|   if B then S1 else S2 end
|   thread S1 end
|   try S1 catch X then S2 end
|   raise X end
```

Рис.1.1. Ядро мови Oz

Модель виконання Oz складається з потоків даних, що спостерігають за спільним сховищем. Потоки містять послідовності операторів Si і взаємодіють через спільні посилання в магазині. Потік є потоком даних, якщо він виконує свій наступний оператор лише тоді, коли доступні всі значення, які потрібні оператору. Якщо для оператора потрібне значення, яке ще не доступне, тоді потік автоматично блокується, поки він не зможе отримати доступ до цього значення. Як ми побачимо, доступність даних у моделі Oz реалізується за допомогою логічних змінних.

Спільне сховище - це не фізична пам’ять, це скоріше абстрактне сховище, яке дозволяє лише операції, які є законними для суб’єктів, тобто не існує прямого способу перевірки внутрішніх представництв об’єктів. Зберігання містить незв’язані та зв’язані логічні змінні, комірки (з іменами, що змінюються покажчиками, тобто явний стан), і процедури (з іменами, що мають лексично обмежене закриття, які є першокласними сутностями). Змінні можуть посилатися на назви процедур та комірок. Клітини вказують на змінні. Зовнішні посилальні процедури є змінними. Коли змінна пов'язана, вона зникає, тобто всі потоки, на які вона посилається, автоматично посилатимуться на прив'язку замість неї.

Змінні можуть бути прив’язані до будь-якої сутності, включаючи інші змінні. Зберігання змінних та процедур є монотонними, тобто інформація може бути лише додана до них, а не змінена або видалена.

Рисунок 1.1 визначає абстрактний синтаксис виразу S мовою ядра. Коротко визначаємо кожне можливе твердження. Послідовності операторів послідовно зменшуються всередині потоку. Значення (записи, числа тощо) вводяться явно і можуть бути прирівняні до змінних. Всі змінні - це логічні змінні, оголошені в явному обсязі, визначеному локальним оператором. Процедури визначаються під час виконання з інструкцією proc і посилаються на змінну. Застосування процедур блокуються, поки їх перший аргумент не посилається на процедуру.

Стан створюється явно NewCell, який створює комірку, оновлюваний покажчик на сховище змінних. Клітини зчитуються @ та оновлюються: = або альтернативно Exchange. Умовники використовують ключове слово if і block, поки змінна умови B не буде істинною чи хибною у сховищі змінних. Потоки створюються явно за допомогою оператора thread. Обробка винятків обробляється динамічно і використовує оператори try and rise.

Повна мова Oz визначається шляхом перетворення всіх її тверджень на цю мову ядра. Це буде детально пояснено в цьому документі. Oz підтримує такі ідіоми, як об'єкти, класи, блокування та порти. Система реалізує їх ефективно, дотримуючись їх визначень. В якості вступу ми дамо короткий підсумок визначення кожної ідіоми. Для наочності на цьому етапі ми зробили невеликі концептуальні спрощення.

Клас - це, по суті, запис, що містить таблицю методів та імена атрибутів. Клас визначається за допомогою багаторазового успадкування, і всі конфлікти вирішуються під час визначення під час побудови таблиці методів.

Об’єкт - це, по суті, спеціальний запис, що має ряд компонентів. Одним із компонентів є клас об’єкта. Іншим компонентом є процедура з одним аргументом, яка посилається на клітинку, яка прихована за допомогою лексичного обсягу. Клітина містить стан об’єкта. Застосування об'єкта Obj до повідомлення M застосовує процедуру об'єкта до M. Аргумент індексується в таблиці методів. Метод - це процедура, яка отримує посилання на комірку стану. Загалом це змінює стан об'єкта.

Повторне блокування - це, по суті, процедура з одним аргументом {Lck P}, що використовується для явного взаємного виключення, наприклад, тіл методів в об’єктах, що використовуються одночасно. Блоки, що повертаються, використовують комірки та логічні змінні для досягнення своєї поведінки. P - процедура з нульовим аргументом, що визначає критичний розділ. Повторне введення означає, що одному і тому ж потоку дозволено входити в замок. Тому дзвінки до блокування можуть бути вкладеними. Блокування звільняється автоматично, якщо різьба в корпусі закінчується або викликає виняток, який виходить із корпусу блокування.

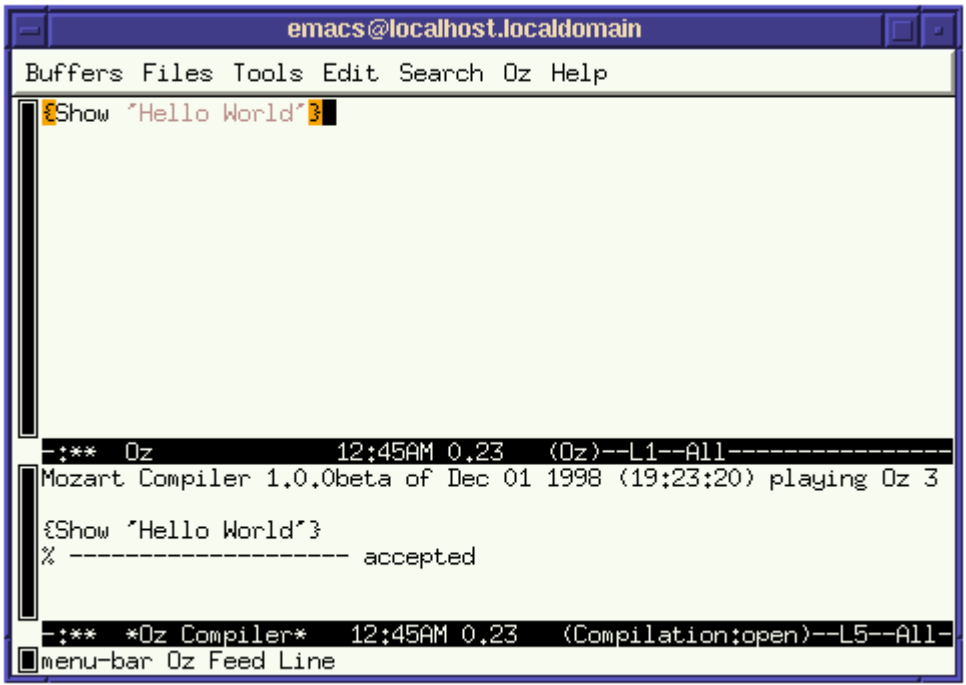
Порт - це асинхронний канал, який підтримує багатосторонній зв'язок. Порт P інкапсулює потік S. Потік - це список із незв'язаним хвостом. Операція {Надіслати P M} додає M до кінця S. Послідовні надсилання з того самого потоку відображаються в тому порядку, в якому вони були відправлені.

Почнемо з традиційного прикладу Hello World. В буфер Oz введіть наступне:

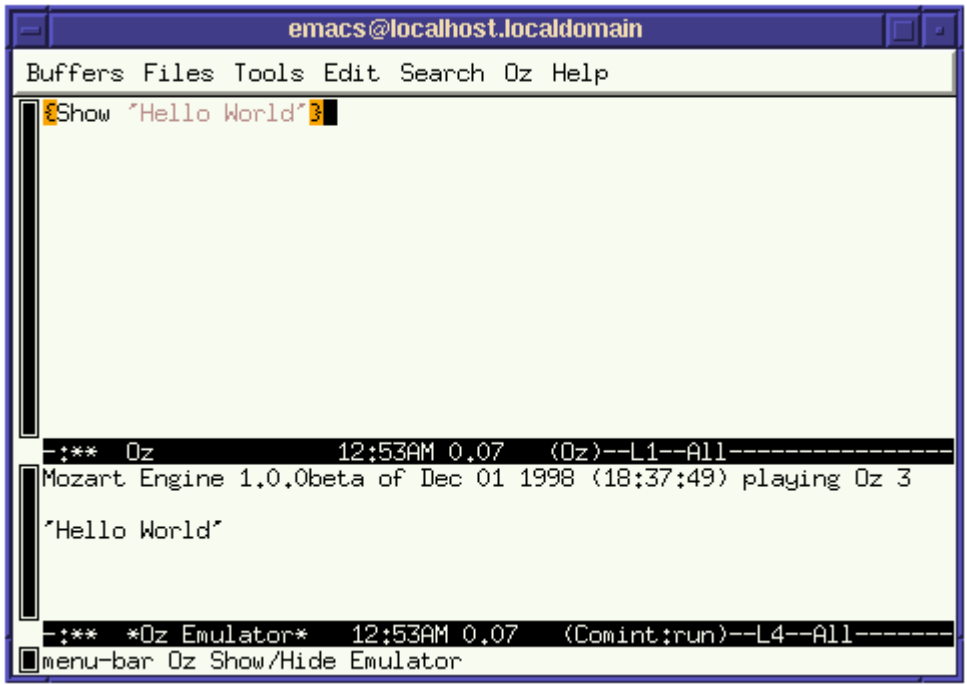
```
{Show 'Hello World'}
```

Цей приклад ілюструє нетрадиційний синтаксис виклику процедури в Oz: він позначений фігурними дужками. Тут процедура **`Show`** викликається як єдиний аргумент атомом **`"Hello World"`**.

Для того, щоб виконати цей фрагмент, ми розміщуємо точку на щойно набраному рядку та вибираємо **`Feed Line`** з меню Oz на панелі меню. Тепер ми бачимо:


emacs@localhost.localdomain
Buffers Files Tools Edit Search Oz Help
{Show 'Hello World'}
-:** Oz 12:45AM 0.23 (Oz)--L1--All----------
Mozart Compiler 1.0.0beta of Dec 01 1998 (19:23:20) playing Oz 3
{Show 'Hello World'}
% -------------------- accepted
-:** *Oz Compiler* 12:45AM 0.23 (Compilation:open)--L5--All-
menu-bar Oz Feed Line

Стенограма з компілятора вказує, що {**Show 'Hello World'**} подано компілятору та прийнято, тобто. е. успішно проаналізовано та скомпільовано. Але чи було це виконано, і, якщо так, де вихід? Дійсно, він був виконаний, але вихід з'являється в іншому буфері, який називається *** Oz Emulator ***: він містить стенограму виконання. Якщо вибрати з меню Oz **Show / Hide -> Emulator**, тепер ми бачимо:


emacs@localhost.localdomain
Buffers Files Tools Edit Search Oz Help
{Show 'Hello World'}
-:** Oz 12:53AM 0.07 (Oz)--L1--All----------
Mozart Engine 1.0.0beta of Dec 01 1998 (18:37:49) playing Oz 3
'Hello World'
-:** *Oz Emulator* 12:53AM 0.07 (Comint:run)--L4--All-------
menu-bar Oz Show/Hide Emulator

Основи мови Oz

Спочатку ми обмежимося послідовним стилем програмування Oz. На цьому етапі ви можете думати про обчислення Oz як про виконання послідовного процесу, який виконує одне твердження за іншим. Ми називаємо цей процес потоком. Цей потік має доступ до магазину. Він може маніпулювати магазином, читаючи, додаючи та оновлюючи інформацію, що зберігається в магазині. Інформація доступна через поняття змінних. Потік може отримати доступ до інформації лише за допомогою видимих йому змінних, прямо чи опосередковано. Змінні Oz - це змінні одного присвоєння або більш доречні логічні змінні.

В імперативних мовах, таких як C та Java, змінну можна призначати кілька разів. На відміну від цього, змінні одного присвоєння можуть бути призначені лише один раз. Це поняття відоме багатьом мовам, включаючи мови потоків даних та паралельні мови програмування логіки. Одна змінна присвоєння має ряд фаз протягом свого життя. Спочатку він вводиться з невідомим значенням, а пізніше йому може бути присвоєне значення, і в цьому випадку змінна стає прив'язаною. Як тільки змінна пов'язана, вона сама не може бути змінена. Логічна змінна - це одна змінна призначення, яку також можна прирівняти до іншої змінної.

Використання логічних змінних не означає, що ви не можете моделювати зміну стану, оскільки змінна, як ви побачите пізніше, може бути прив'язана до комірки, яка має статус, тобто вміст комірки може бути змінений.
Потік виконання оператора:

```
local X Y Z in S end
```

введе три змінні єдиного присвоєння X, Y, Z та виконає оператор S у межах цих змінних. Змінна зазвичай починається з великої літери, за якою може йти довільна кількість буквено-цифрових символів.

Змінні також можуть бути представлені у текстовому вигляді як будь-який рядок символів для друку, укладених у символи зворотних лапок, наприклад `це **$** є змінною`. До виконання **S** заявлені змінні не матимуть жодних пов'язаних значень. Ми говоримо, що змінні незв’язані. Необхідно ввести будь-яку змінну в програмі Oz, за винятком певних конструкцій, що відповідають шаблонам, які будуть показані пізніше.

Інша форма декларації:

```
declare X Y Z in S
```

Це відкрите оголошення, яке робить **X**, **Y** та **Z** глобально видимими в **S**, а також у всіх твердженнях, що слідують за **S** текстово, якщо це не замінено іншим оголошенням змінних тих самих текстових змінних. **X, Y, Z** - глобальні змінні.

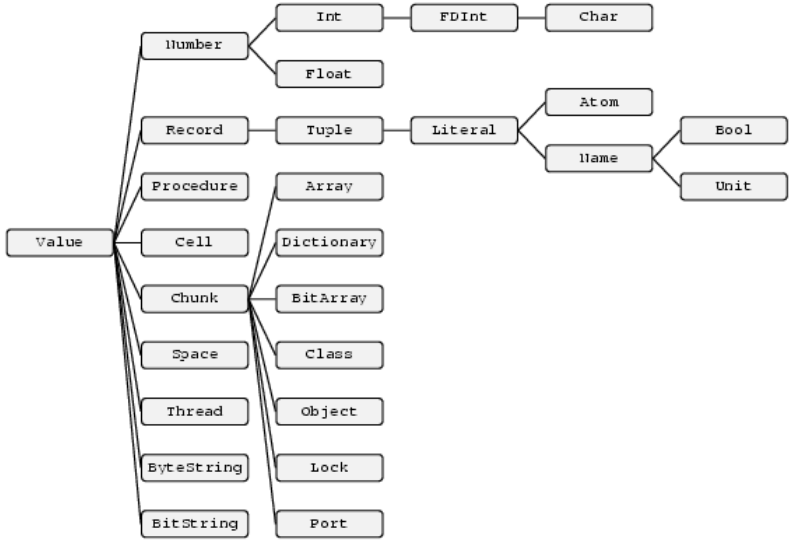


Рис 3.1. Ієрархія типів мови Oz.

Oz - це мова, що динамічно набирається. На рисунку 3.1 показана ієрархія типів Oz. Будь-яка змінна, якщо вона коли-небудь отримає значення, буде прив'язана до значення одного з цих типів. Більшість типів здаються знайомими досвідченим програмістам, за винятком, ймовірно, **`Chunk`**, **`Cell`**, **`Space`**, **`FDInt`** та **`Name`**. Ми з часом обговоримо всі ці типи. Для нетерплячого читача ось кілька підказок. Тип даних **`Chunk`** дозволяє користувачам вводити нові абстрактні типи даних. Клітина вводить примітивне поняття стану-контейнера та модифікації стану. Місце буде потрібно для розширеного вирішення проблем за допомогою техніки пошуку.

`FDInt` - це тип кінцевого домену, який часто використовується при програмуванні обмежень та задоволенні обмежень. Назва представляє анонімні унікальні незабутні токени.

Мова динамічно набирається в тому сенсі, що при введенні змінної її тип, а також значення невідомі. Лише коли змінна прив’язана до значення Oz, її тип стає визначеним.

Числа

Наступна програма вводить три змінні I, F і C. Вона присвоює I ціле число, F плаваюче, а C символ t у такому порядку. Потім відображається список, що складається з I, F і C.

```
local I F C in
   I = 5
   F = 5.5
   C = &t
   {Browse [I F C]}
end
```

Oz підтримує двійкові, вісімкові, десяткові та шістнадцяткові записи для цілих чисел, які можуть бути довільно великими. Вісімкове починається з провідного 0, а шістнадцяткове починається з провідного 0x або 0X. Плаваючі числа відрізняються від цілих чисел і повинні мати десяткові крапки. Показані інші приклади плаваючих значень, де ~ - одинарний мінус:

```
~3.141    4.5E3     ~12.0e~2
```

Літерали

Іншою важливою категорією атомних типів, тобто типами, члени яких не мають внутрішньої структури, є категорія літералів. Літерали поділяються на атоми та імена. Атом - це символічна сутність, яка має ідентичність, що складається з послідовності літерно-цифрових символів, що починаються з малої літери, або довільних друкованих символів, укладених у лапки. Наприклад:

```
a foo '=' ': =' 'OZ 3.0' 'Hello World' '
```

Атоми мають впорядкування, засноване на лексикографічному впорядкуванні.

Інша категорія елементарних сутностей - ім’я. Єдиний спосіб створити ім’я - це виклик процедури `{NewName X}`, де `X` присвоюється нове ім’я, яке гарантовано буде унікальним у всьому світі. Імена не можна підробляти або друкувати. Як буде видно пізніше, імена відіграють важливу роль у безпеці програм Oz.

Підтипом **Name** є **Bool**, який складається з двох імен, захищених від перевизначення за допомогою зарезервованих ключових слів **true** і **false**. Таким чином, користувацька програма не може перевизначити їх, а також зіпсувати всі програми, спираючись на їх визначення. Існує також тип **Unit**, який складається з одиниці єдиного імені. Це використовується як маркер синхронізації в багатьох одночасних програмах.

```
local X Y B in
   X = foo
   {NewName Y}
   B = true
   {Browse [X Y B]}
end
```

Записи та кортежі

Записи - це структуровані складені сутності. Запис має мітку та фіксовану кількість компонентів або аргументів. Існують також записи із змінною кількістю аргументів, які називаються відкритими записами. Наразі ми обмежуємось «закритими» записами. Далі є запис:

```
tree(key: I value: Y left: LT right: RT)
```

Він має чотири аргументи та дерево міток. Кожен аргумент складається з пари **Функція: Поле**, тому ознаками наведеного вище запису є ключ, значення, ліворуч та праворуч. Відповідні поля - це змінні **I**, **Y**, **LT** і **RT**. Можна опустити особливості запису, зменшуючи його до того, що відомо з логічного програмування як складений термін. В країні Oz це називається кортеж. Отже, наступний кортеж має ту саму мітку та поля, що і вищезазначений запис:

```
tree(I Y LT RT)
```

Це просто синтаксичний запис для запису:

```
tree(1:I 2:Y 3:LT 4:RT)
```

де ознаками є цілі числа, починаючи від 1 до кількості полів у кортежі. Наступна програма покаже список, що складається з двох елементів, один із яких є записом, а інший - кортежем, що має однакову мітку та поля:

```
declare T I Y LT RT W in
T = tree(key:I value:Y left:LT right:RT)
I = seif
Y = 43
LT = nil
RT = nil
W = tree(I Y LT RT)
{Browse [T W]}
```

На дисплеї з'явиться:

```
[tree(key:seif value:43 left:nil right:nil)
 tree(seif 43 nil nil)]
```

Операції над записами

Ми обговорюємо деякі основні операції із записами. Більшість операцій містяться в модулі **`Record`**. Для вибору поля компонента запису ми використовуємо оператор інфіксної крапки, тобто **`Record.Feature.`**

```
% Selecting a Component
{Browse T.key}
{Browse W.1}
% will show seif twice in the browser
seif
seif
```

Арність запису - це перелік особливостей запису, відсортований лексикографічно. Для відображення архітності запису ми використовуємо процедуру **Arity**. Застосування процедури **{Arity R X}** буде виконано, як тільки **R** буде прив'язано до запису, і прив'яже **X** до архітектури запису. Виконання наступних тверджень

```
% Getting the Arity of a Record
local X in {Arity T X} {Browse X} end
local X in {Arity W X} {Browse X} end
```

відображатиметься

```
[key left right value]
[1 2 3 4]
```

Ще одна корисна операція - умовний вибір поля запису. Операція **CondSelect** приймає запис **R**, функцію **F** і значення поля за замовчуванням **D**, а також аргумент результату **X**. Якщо функція **F** існує в **R**, **X** прив'язується до **RF**, інакше **X** прив'язується до значення **D**. **CondSelect** насправді не є примітивною операцією. Це можна визначити в країні Oz. Наступні твердження:

```
% Selecting a component conditionally
local X in {CondSelect W key eeva X} {Browse X}
end
local X in {CondSelect T key eeva X} {Browse X}
end
```

відображатиметься

```
eeva
seif
```

Поширеним інфіксним кортежним оператором, що використовується в Oz, є `#`. Отже, `1 # 2` - це кортеж з двох елементів, і зауважте, що `1 # 2 # 3` - це один кортеж із трьох елементів:

```
'#' (1 2 3)
```

а не пара `1 # (2 # 3)`. За допомогою оператора `#` ви не можете безпосередньо писати порожній або одиничний кортеж. Натомість вам слід повернутися до звичайного синтаксису запису префіксу: порожній кортеж повинен бути написаний `'#' ()` або просто `'#'`, а один елемент кортежу `'#' (X)`.

Операція **{AdjoinAt R1 FX R2}** прив'язує **R2** до запису, що виникає внаслідок примикання поля **X** до **R1** при ознаці **F**. Якщо **R1** вже має функцію **F**, отриманий запис **R2** ідентичний **R1**, за винятком поля **R1.F**, значення якого стає **X**. В іншому випадку аргумент **F: X** додається до **R1**, що призводить до **R2**.
Операція **{AdjoinList R LP S}** бере запис **R**, список пар об'єкт-поле і повертає в **S** новий запис такий, що:

- Мітка **R** дорівнює мітці **S**.
- **S** має компоненти, які вказані в **LP** на додаток до всіх компонентів у **R**, які не мають властивості, що зустрічається в **LP**.

Звичайно, ця операція визначається за допомогою **AdjointAt.**

```
local S in
   {AdjoinList tree(a:1 b:2) [a#3 c#4] S}
   {Show S}
end
% gives S=tree(a:3 b:2 c:4)
```

Списки

Як і в багатьох інших символічних мовах програмування, напр. Схеми та Пролог, списки утворюють важливий клас структур даних в країні Oz. Категорія списків не належить до одного типу даних в країні Oz. Вони радше концептуальна структура. Список - це або атом нуль, що представляє порожній список, або кортеж, що використовує оператор інфіксу | та два аргументи, які є відповідно головою та хвостом списку. Таким чином, список перших трьох натуральних чисел представлений у вигляді:

```
1 | 2 | 3 | nil
```

Інший зручний спеціальний запис для закритого списку, тобто е. список із визначеною кількістю елементів:

```
[1 2 3]
```

Вищезазначені позначення використовуються лише для закритого списку, тому виглядає список, першими двома елементами якого є **1** і **2**, але хвостом якого є змінна **X:**

```
1 | 2 | X
```

Можна також використовувати стандартну нотацію записів для списків:

```
'|' (1 '|' (2 X))
```

Подальший нотаційний варіант дозволений для списків, елементи яких відповідають кодам символів. Списки, написані в цьому позначенні, називаються рядками, напр.

```
"OZ 3.0"
```

- це список

```
[79 90 32 51 46 48]
```

або еквівалентно

```
[& O & Z & & 3 &. & 0]
```

Віртуальні рядки

Віртуальний рядок - це спеціальний кортеж, який представляє рядок із віртуальною конкатенацією, тобто конкатенація виконується тоді, коли це дійсно потрібно. Віртуальні рядки використовуються для вводу-виводу з файлами, сокетами та вікнами. Для складання віртуальних рядків можна використовувати всі атоми, крім `nil` та `'#'`, а також цифри, рядки або позначені символом `'#'`. Ось один приклад:

```
123 # "-" # 23 # "є" # 100
```

представляє рядок

```
"123-23 - це 100"
```

Оператор тесту на рівність та рівність

До цього часу ми показали прості приклади твердження про рівність, напр.

```
W = tree(I Y LT LR)
```

Вони були досить простими, щоб інтуїтивно зрозуміти, що відбувається. Однак, що відбувається, коли дві незв’язані змінні прирівнюються до **`X = Y`**, або коли прирівнюються дві великі структури даних. Ось коротке пояснення. Ми можемо сприймати магазин як динамічно розширюваний масив слів пам'яті, які називаються вузлами. Кожен вузол позначений логічною змінною. Коли вводиться змінна **`X`**, у сховищі створюється новий вузол, позначений **`X`**, значення якого невідоме. На даний момент вузол не має жодної реальної цінності; він порожній як контейнер, який можна заповнити пізніше.

Змінна, що позначає вузол, значення якого невідоме, є незв'язаною змінною. Вузли досить гнучкі, щоб містити будь-яке довільне значення Oz. Операція

```
W = tree(1:I 2:Y 3:LT 4:LR)
```

зберігає структуру записів у вузлі, пов'язаному з **W**. Зверніть увагу, що ми просто отримуємо графічну структуру. Вузол містить запис із чотирма полями. Поля містять дуги, що вказують на вузли, позначені **I, Y, LT** та **LR** відповідно. Кожна дуга, у свою чергу, позначається відповідною ознакою запису. Враховуючи дві змінні **X** та **Y**, операція **X = Y** спробує об'єднати їх відповідні вузли.

Тепер ми можемо дати обґрунтований облік операції злиття **`X = Y`**, відомої як покрокове повідомлення або, як альтернатива, операція об'єднання.

Якщо **`X`** та **`Y`** позначають один і той же вузол, операція завершена успішно.

Якщо **`X`** (відповідно **`Y`**) не є зв'язаним, тоді об'єднайте вузол **`X`** (відповідно **`Y`**) з вузлом **`Y`** (відповідно **`X`**). Злиття означає заміну всіх посилань на вузол **`X`** посиланням на **`Y1`**. Концептуально вихідний вузол **`X`** відкинуто.

Якщо **X** та **Y** позначають різні вузли, що містять записи **Rx** та **Ry** відповідно:

Якщо **Rx** та **Ry** мають різні мітки, сутності або і те, і інше: операція завершена та висунуто виняток.

В іншому випадку аргументи **Rx** та **Ry** з однією і тією ж ознакою попарно об'єднуються у довільному порядку.

Коли змінна більше не доступна, процес, відомий як збір сміття, відновлює свій вузол.

Ось кілька прикладів успішних операцій щодо рівності:

```
local X Y Z in
   f(1:X 2:b) = f(a Y)
   f(Z a) = Z
   {Browse [X Y Z]}
end
```

покаже `[a b R14 = f (R14 a)]` у браузері.

`R14 = f (R14 a)` - зовнішнє представлення циклічного графіка.

Наступний приклад показує, що відбувається, коли змінні з несумісними значеннями прирівнюються.

```
declare X Y Z in
X = f(c a)
Y = f(Z b)
X = Y
```

Інкрементний тег `X = Y` прив’яже `Z` до значення `c`, але також спричинить виняток, який система вловлює, коли вона намагається прирівняти `a` та `b`.

Управляючі структури

Ми вже бачили деякі основні твердження в країні Oz. Представляємо нові змінні та послідовність тверджень:

```
S1 S2
```

Повторюючи ще раз, потік виконує оператори в послідовному порядку. Однак потік, на відміну від звичайних мов, може призупинятись у якомусь твердженні, тому вище, потік повинен завершити виконання `S1`, перш ніж запускати `S2`. Насправді `S2` може взагалі не виконуватися, якщо в `S1` встановлено виняток.

skip

Оператор **`skip`** - це порожній оператор.

If твердження

Oz надає просту форму умовного висловлювання, що має такий вигляд:

```
if B then S1 else S2 end
```

`B` має бути булевим значенням.

Процедурна абстракція

Визначення процедури є основною абстракцією в країні Oz. Процедуру можна визначити, передати як аргумент іншій процедурі або зберегти в записі. Визначення процедури - це твердження, яке має таку структуру.

```
proc {P X1 ... Xn} S end
```

Наступна лекція буде присвячена елементам функціонального, ООП та логічного програмування в мові Oz.