Prozeduren und Funktionen

Übersicht

- Terminologie

echte Prozedur, Unterprogramm	Prozedur ohne Ergebnis
Funktionsprozedur	Prozedur mit Ergebnis
Methode	Prozedur in OO-Sprachen
Prozedurvariable, formale Prozedur	Variable mit Prozedur als Wert

- Vereinbarung

geschachtelt	Algol Sprachen (Algol, Pascal, ...)
nicht geschachtelt	Fortran, C, OO-Sprachen

Prozeduren und Funktionen

Übersicht II

- Aufruf
  - Parameterlos: `p()` oder `p` (uniform referent)
  - Argumente nach Position `p(x1,...,xn)`
  - Argumente mit Bezeichner `p(file="hallo.txt", mode=read)`
- Rückgabe
  - Zugewiesen an Prozedurbezeichner: `p := Ergebnis`
  - Mitgeteilt durch `return Ergebnis`
  - Zugewiesen an Spezialvariable `res := Ergebnis`
  - Ergebnis kann ignoriert werden: Fortran, C, viele OO-Sprachen

Prozeduren und Funktionen

Vereinbarung

- Funktion (mit Schlüsselwortparametern):

```
Handle open(string file = "stdin",
            Mode mode = Open,
            int buffers = 2)
...
```

`file`, `mode`, `buffers` heißen formale Parameter

- Parameterart kennzeichnen:
  - kein Kennzeichen: Eingabeparameter
  - `out` Ausgabeparameter
  - `in out` Transienter Parameter
- Parameterart in manchen Sprachen auch im Aufruf gekennzeichnet
- Fehlerbehandlung
  - mit ganzzahligem Fehlercode als Ergebnis oder
  - mit Ausnahmebehandlung (später)
- mögliche Ausnahmen als Teil der Signatur (Java)!

Prozeduren und Funktionen

Aufruf

```
f(a1, ..., an)
```

oder

```
y := f(a1, ..., an)
```

Die Ausdrücke `a1,...,an` heißen aktuelle/tatsächliche Parameter oder auch Argument

Prozeduren und Funktionen

Parameterübergabe

Verfahren

- Wertaufruf (call-by-value)
- Ergebnisaufruf (call-by-result)
- Referenzaufruf (call-by-reference)
- Namensaufruf (call-by-name)

Zeitpunkt der Auswertung

- Strikter (Wert-)Aufruf (strict evaluation)
- Fauler Aufruf (call-by-need, lazy evaluation)

Gültigkeitsbereich von Parameterbezeichnern

- wie lokale Variable im Rumpf der Prozedur
  (in den meisten Sprachen, z.B. Java)
- wie lokale Variable in einem Block, der den Rumpf umgibt
  (C, C++)

Prozeduren und Funktionen

Parameterübergabe – Prinzipien

	Parameter ist ...	Wird bei ...
Wertaufruf	lokale Variable	Aufruf mit Argument initialisiert
Ergebnisaufruf	lokale Variable	Rückkehr an das Argument zugewiesen
Referenzaufruf	lokale Konstante	Aufruf mit Referenz auf Argument initialisiert
Namensaufruf	Ausdruck	*siehe folgende Folien*

Prozeduren und Funktionen

Namensaufruf

- Übergebe gesamten Ausdruck an Prozedur/Funktion
- Implementierung durch Zugriffsfunktion, die die Berechnung des Aufrufs implementiert
- Wenn an den Parameter zugewiesen wird, muss der Ausdruck eine gültige linke Seite darstellen
- Erstmals in Algol 60
- Auch verwendet in TCL

Prozeduren und Funktionen

Wertaufruf

Beispiel

```
int m = 1;
int n;

int p(int j, int k) {
  j = j+1;
  m = m+k;
  return j+k;
}
n = p(m, m+3);
```

Variable	m	n	j	k
Wert	5	6	1	4

Prozeduren und Funktionen

Wert-/Ergebnisaufruf

Beispiel

```
int m = 1;
int n;

int p(out int j, int k) {
  j = j+1;
  m = m+k;
  return j+k;
}
n = p(m, m+3);
```

Variable	m	n	j	k
Wert	2	6	1	4

Prozeduren und Funktionen

Referenzaufruf

Beispiel

```
int m = 1;
int n;

int p(ref int j, int k) {
  j = j+1;
  m = m+k;
  return j+k;
}
n = p(m, m+3);
```

Variable	m	n	j	k
Wert	6	10	6	4

Prozeduren und Funktionen

Namensaufruf

Beispiel

```
int m = 1;
int n;

int p(name int j, name int k) {
  j = j+1;
  m = m+k;
  return j+k;
}
n = p(m, m+3);
```

Variable	m	n	j	k
Wert	7	17	7	10

Namensaufruf

Jensen's Device

```
real  procedure sum(i,n,x);
      value n;
      integer i,n;
      real x;
begin
    real s;
    s := 0;
    for i := 1 step 1 until n do
        s := s + x;
    sum := s;
end;
```

Namensaufruf

Jensen's Device

```
real  procedure sum(i,n,x);
      value n;
      integer i,n;
      real x;
begin
    real s;
    s := 0;
    for i := 1 step 1 until n do
        s := s + x;
    sum := s;
end;
```

Fragen

- sum(i, n, i) =

Namensaufruf

Jensen's Device

```
real  procedure sum(i,n,x);
      value n;
      integer i,n;
      real x;
begin
    real s;
    s := 0;
    for i := 1 step 1 until n do
        s := s + x;
    sum := s;
end;
```

Fragen

- sum(i, n, i) $= \sum_{i=1}^{n} i$

Namensaufruf

Jensen's Device

```
real  procedure sum(i,n,x);
      value n;
      integer i,n;
      real x;
begin
    real s;
    s := 0;
    for i := 1 step 1 until n do
        s := s + x;
    sum := s;
end;
```

Fragen

- sum(i, n, i) = $\sum_{i=1}^{n} i$
- sum(i, n, a[i]*b[i]) =

Namensaufruf

Jensen's Device

```
real  procedure sum(i,n,x);
      value n;
      integer i,n;
      real x;
begin
    real s;
    s := 0;
    for i := 1 step 1 until n do
        s := s + x;
    sum := s;
end;
```

Fragen

- `sum(i, n, i)` $= \sum_{i=1}^{n} i$
- `sum(i, n, a[i]*b[i])` $= \sum_{i=1}^{n} a_i b_i$

Namensaufruf

Jensen's Device

```
real  procedure sum(i,n,x);
      value n;
      integer i,n;
      real x;
begin
    real s;
    s := 0;
    for i := 1 step 1 until n do
        s := s + x;
    sum := s;
end;
```

Fragen

- `sum(i, n, i)` $= \sum_{i=1}^{n} i$
- `sum(i, n, a[i]*b[i])` $= \sum_{i=1}^{n} a_i b_i$
- Was passiert, wenn `i` nicht per Name übergeben wird?

Prozeduren

Ausführungsprinzip

- Modellvorstellung, die die Sichtbarkeit und Zuordnung Bezeichner - Vereinbarung regelt, auch bei rekursivem Aufruf:
  1. Reserviere Speicher auf dem Keller für Parameter (ohne Reihungen) und Organisation
  2. Kopiere Prozedurtext
  3. Ersetze alle lokalen Bezeichner, einschl. Parameter, in der Kopie durch neue Bezeichner
  4. Berechne Argumente (im Kontext des Aufrufers) und weise sie an die neu bezeichneten Parameter zu
  5. Unterprogrammaufruf (Befehl)
  6. Speichere Rückkehradresse im Keller
  7. Ausführung der Prozedurkopie
  8. Prozedurrückkehr
  9. Zuweisung der Ergebnisparameter an ihre Argumente, falls vorhanden (auch vor Rückkehr möglich)
  10. Beseitige Prozedurkopie
- Analoges Schema für begin-end Blöcke mit lokalen Vereinbarungen

Ausnahmebehandlung

Ziel

Fehlerbehandlung so, daß eine Anweisung (ein Block) ordnungsgemäß zu Ende geführt wird.

- Auslösen der Ausnahme (raise, eventuell implizit durch Hardware) springt auf den Beginn der passenden Fehlerbehandlung.
- Wenn keine Fehlerbehandlung lokal vorhanden: Suche nach Fehlerbehandlung im dynamisch umfassenden Block, eventuell unter Beendigung der Prozedur, die den Fehler enthielt.
- Ausnahmebehandlung wie beschrieben in Ada, Modula-3, Sather, Java
- Java zählt Ausnahmen zur Signatur von Klassen/Prozeduren
- Allgemeinere Formen der Ausnahmebehandlung, z.B. mit Wiederaufsetzen, in MESA (Xerox PARC), vgl. auch J. Goodenough, CACM Juli 1975

Steuerparallelität

nebenläufige Programmierung, multithreading, multitasking

1. erzeuge neuen Prozeß (Faden, thread) p,
   einschl. Speicherreservierung für Keller
2. weise p Programmstück zu Funktionsaufruf (Normalfall), spezielle task (Ada)
3. starte Prozeß
4. Kommunikation oder Synchronisierung mit anderen Prozessen
5. Prozeßende:
   - Rückmeldung Ergebnis
   - Stop des Prozesses
   - Ende des Fadens
   - Ende Speicherreservierung Keller

Steuerparallelität

nebenläufige Programmierung, multithreading, multitasking

- 1-3 oft zusammengefaßt zu einer Operation (siehe `fork`)
- Kommunikation, Synchronisierung mit Hilfe von Semaphoren (gem. Speicher) oder Botschaften oder abgeleiteten Verfahren (Monitore, Fernaufruf, Rendezvous)
- Unterscheide beim Prozessende
  - Vater läuft simultan weiter, Sohn informiert Vater über Prozeßende
  - mehrere gleichzeitig gestartete Prozesse warten aufeinander, bevor Vater fortsetzt
  - erster fertiger Prozeß beendet alle anderen Söhne und setzt Vater fort
  - Sohn läuft unabhängig vom Vater, Vater eventuell früher fertig als Sohn

Datenparallelität

Parallelzuweisung

$$(v_1, \ldots, v_n) := (e_1, \ldots, e_n)$$

Auch Vertauschung $(x, y) := (y, x)$ erlaubt!

Parallele Schleife

```
for i := u step s until o do in parallel
  a[I(i)] := f(a[i], ..., a[i])
end
```

Führe alle $o - u + 1$ Durchläufe simultan aus

- Voraussetzung: Keine Datenabhängigkeit zwischen einzelenen Durchläufen
- Beispiel: Berechnung von `a[i]` darf nicht von der Kenntnis von `a[i+1]` abhängen
- Ausnahme: Selbststabilisierende Berechnungen:
  Sind korrekt, unabhängig davon, ob alter oder neuer Wert von `a[i+1]` verwendet wird

Programmstruktur, Abstraktionen

Ziele

- Gliederung des Programms in Module, Klassen, ...
- Unterstützung von Abstraktionen:

  zusammengesetzte Operation Hauptprogramm, Prozedur, Block

  zusammengesetztes Objekt Verbund

  benutzerdefinierter Typ Klasse, Vererbung
- Durchsetzung des Geheimnisprinzips

Blockstruktur

Block (Samelson 1958): Zusammenfassung von Anweisungen und zugehörigen Vereinbarungen zu größerer Anweisung

- eigenständiger Gültigkeitsbereich: globale, gleichbenannte Größen nicht zugänglich
- lokale Größen extern nicht sichtbar
- lokale Variable haben nur lokale Lebensdauer

Programm ist Block mit geschachtelten Unterblöcken

- Abbildung der Prozeduraufrufhierarchie auf dynamische Schachtelung von Blöcken

Prozedurale Abstraktion

Hauptprogramm:

- In ALGOL, Pascal, . . . :
  umfassender Block „enthält“logisch alle anderen Programmeinheiten
  (auch bei getrennter Übersetzung)
- In Fortran, C, und OO-Sprachen:
  Spezielle Prozedur; Parameter: Text der Kommandozeile

Prozedur:

- Zusammenfassung von Anweisungen
- Logische Einheit: abstrakte Operation
- Ist zusammen mit Typdefinitionen, anderen Operationen Teil einer umfassenden Abstraktion Modul
- OO-Sprachen: Teil einer Klasse, d.h. eines Objekttyps

Globale Variablen

Common Zonen

Common (Fortran) oder Data (Cobol):
In beliebigen Unterprogrammen (UP) möglich:

```
COMMON x, y, z
```

- Umbenennung der Variablen in jedem Unterprogramm möglich
- Unterschiedliche Typisierung möglich! Z.B.:
  - Statt `complex` zweimal `real`
  - Zusammenfassung zu Reihungen unterschiedl. Länge

Beispiel

```
SUBROUTINE foo
  REAL a(4)
  COMMON a
  ...
```

```
SUBROUTINE bar
  COMPLEX c(2)
  COMMON c
  ...
```

Globale Variablen

Benannte Common Zonen

Common Zonen können benannt werden

```
COMMON /a/ x, y, z
COMMON /b/ u, v, w
```

- Jeder Name definiert Gruppe von Unterprogrammen
- Unterprogramme kann zu mehreren Gruppen gehören

Beispiel

```
SUBROUTINE foo
  REAL x(4)
  COMMON /a/ x
  ...
```

```
SUBROUTINE bar
  COMPLEX y(2)
  COMMON /b/ y
  ...
```

```
SUBROUTINE zap
  COMPLEX x(2)
  REAL    z(4)
  COMMON /a/ x
  COMMON /b/ y
  ...
```

Unterprogramme mit mehreren Einstiegspunkten

Unterprogramme in Fortran können mehrere Prozedurköpfe enthalten mit unterschiedlicher Parametrisierung

- Gruppierung von Operationen, die lokale gemeinsame Daten benutzen
- Gruppierung von Operationen, die gemeinsame Codestücke enthalten

Alternative zum Gebrauch benannter Common Zonen ▶ Geheimnisprinzip

kontrollierter Zugang Jede Operation soll nur auf die nicht-lokalen Größen zugreifen können, die sie logisch benötigt

Abschottung Gruppen von Operationen sollen lokal die Größen vereinbaren können, die sie benötigen; andere haben keinen Zugang

Austauschbarkeit Implementierungen von Operationsgruppen sollen austauschbar ohne Rückwirkung auf andere, solange Schnittstelle gleich bleibt

Wiederverwendung Andere sollen Operationsgruppen nur mit der Kenntnis der Schnittstelle wiederverwenden können

Modularisierung

- Zuordnung von Operationen zu Daten
- Partitionierung der Prozeduren bezüglich verwendeter Bereiche
- Hervorhebung der Schnittstelle
  - Modula, Ada: Unterscheidung von Definition (Schnittstelle) und Implementierung
  - C: .h-Datei definiert Schnittstelle
  - OO-Sprachen: Unterscheidung der öffentlichen und der nur privaten Operationen
- Probleme:
  - Typüberprüfung an der Schnittstelle nötig
  - nur ein Exemplar jedes Moduls außer in OO-Sprachen

Programmieren im Großen

- Zugriff über definierte Schnittstellen (öffentliche Sicht), Konsistenzprüfung zur Übersetzungszeit
- Trennung der Schnittstelle von ihrer Implementierung (private Sicht), austauschbare Implementierungen

Beispiel: Keller (Integer) in C

Schnittstelle

```
#define ERROR INT_MIN
extern int top(void);
extern int pop(void);
extern void push(int);
```

Implementierung

```
static int *stack, mark, size;
static void *realloc(void *, size_t);
int top(void) {
  return mark?stack[mark]:ERROR;
}
int pop(void) {
  return mark?stack[mark--]:ERROR;
}
void push(int i) {
  if (++mark >= size)
    stack=realloc(stack, 2 * size);
  stack[mark]=i;
}
```

Modul – ADT Implementierung

- ADT = (Konstruktoren, Operationen, Axiome)
- Modul =

  Schnittstelle (öffentliche Sicht)

  - Definition von Konstruktoren (Typdeklaration)
  - Operationen

  Implementierung (private Sicht)

  - Konstruktoren und Operationen
  - Muss den Axiomen des ADTs genügen

Modul für ADT Keller in C

Schnittstelle

```
#define ERROR INT_MIN
typedef struct stack stack;
extern int top(stack *);
extern int pop(stack *);
extern void push(stack *, int);
```

Implementierung

```
struct stack {
  int *elms;
  int mark, size;
};

int top(stack *s) {
  return s->mark?s->elms[s->mark]:ERROR;
}
...
```

Generische Module

- Mehrsortige Algebren mit einigen unspezifizierten Sorten.
- Schnittstelle und Implementierung enthalten Typvariablen.
- Instanziierung mit konkreten Typen erzeugt Modul.
- Definiert durch textuelle Substitution der Typvariablen durch konkreten Typ
- Setzt keine Ibjektorientierung voraus (siehe z.B. Ada, ML)

Achtung

Generics in Java 1.5 funktionieren leicht anders!

Generische Klassen in Java 1.5

```
class Stack<T> implements Collection<T> {
  private T[] elems;
  ...
  public T top() {
    return elems[mark];
  }
}
```

- Typparameter dient ausschließlich zur typsichereren Programmierung
- Keine Instanziierung von Typvariablen vom Programmierer erlaubt (wie in C++)
- `T` wird vomn Übersetzer nach `Object` umgesetzt
- Übersetzer fügt nötige Typanpassungen selbst ein
- `T` zur Laufzeit unbekannt, deswegen

  ```
  elems = new T[size];
  ```

  nicht möglich!

Generischer Keller in C

```
#define STACK(type) \
typedef struct type ##_stack { \
  type *elms; \
  int mark, size; \
} type ##_stack; \
\
type type ##_top(type ##_stack *s){ \
  return s->mark ? s->elms[s->mark] : ERROR; \
} \
...
#define TOP(type,stack) type ##_top(stack)
```

Namensbildung durch Makrosubstitution

Generischer Keller in Java 1.5

Schnittstelle

```
public interface Stack<T> {
  T top();
  T pop();
  void push();
}
```

Implementierung

```
public class ArrayStack<T> implements Stack<T> {
  private T[] elms;
  int mark;
  ...
  T top() {
    ...
  }
}
```

Abstraktes Programmieren

- Typen: minimale Eigenschaften von Objekten im Kontext einer Anwendung
- Untertypbeziehungen
  - Objektmengen (Typen), die diesen Anforderungen genügen
  - Entspricht struktureller Konformität
- Typschranken und Typparameter:
  - minimale Eigenschaften von Typen

Weitere Konzepte

Namensräume und Sichtbarkeit

Namensräume

Pakete in Java, namespaces in C++

Zugriffskontrolle

- Objekt
- Klasse, Superklasse, Freundklasse
- Paket, Freund

Metaprogrammieren, Metaklassen, Reflexion

Probleme mit Sichtbarkeit

```
class A {
  public static int x;
}

class B {
  class A {
    int x;
  }
  int foo() {
    return A.x;
  }
  int bar(A A) {
    return A.x;
  }
  ...
}
```

Metainformationen

Metadaten Daten über Programme

Reflexion Besichtigen von Metadaten zur Laufzeit

Metaprogrammieren

- Erstellen und Modifizieren von Programmen
- Programme = Daten (LISP)

Kommentare

Aufgaben

- Entwicklungsdokumentation (für Test, Wartung)
- Schnittstellendokumentation (Moduln, Klassen, Prozeduren)
- Benutzerdokumentation (bei kleinen Programmen)
- Änderungsdokumentation und Versionskontrolle
- Übersetzersteuerung

Unterschiede

- geklammerte Kommentare (mit Schachtelungsmöglichkeit): zum „Auskommentieren“
- Zeilenkommentare (Kommentarende = Zeilenende): Standard
- Beginn Zeilenkommentar nach Aufgabe gegliedert, z.B. /*, /**, etc.

Kommentare

Merke

- Kommentare werden während des Programmentwurfs geschrieben, oder überhaupt nicht
- schlechte Kommentare sind besser als keine Kommentare
- Schreibe erst den Kommentar, dann den Code

Pragmas

Pragmas sind Kommentare zur Übersetzer-/Laufzeitsteuerung

- eingeleitet mit speziellem Kommentarsymbol, z.B. `pragma` (Ada)
  - gültig für Übersetzungseinheit, Modul/Klasse/Prozedur/Anweisung
  - nicht an beliebiger Stelle einsetzbar
- Pragmas sind implementierungsspezifisch, nicht Teil der Sprache!

Aufgabenbeispiele

- Steuerung Sprachumfang (Zulassen zusätzlicher Sprachelemente
- Feinsteuerung Speicherverwaltung
- Unterdrücken von Warnungen
- Hinweise an den Optimierer

Bedingungen

- Übersetzung auch dann korrekt, wenn Pragmas ignoriert werden!?
- Pragmas werden während Symbolentschlüsselung decodiert, daher nur reguläre Syntax!