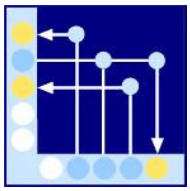
Hochschule Aalen
Fakultät Elektronik und Informatik
Studienbereich Informatik



Compilerbau-Praktikum

Lehrveranstaltung im Sommersemester 2021
Prof. Dr. habil. Christian Heinlein

2. Aufgabenblatt (29. März 2021)

Aufgabe 2: C++-Bibliothek *lib*$_{\texttt{CH}}$: Offene Typen

Arbeiten Sie sich anhand der Dokumentation zur Bibliothek *lib*$_{\texttt{CH}}$ in das Thema offene Typen ein! Wichtig sind zunächst vor allem die Teilabschnitte 6.1.1 bis 6.1.4 sowie 6.1.10. Für den letzten Teil von Teilaufgabe b ist außerdem Teilabschnitt 6.1.7 wichtig.

Implementieren Sie mit dem erworbenen Wissen u. a. folgendes:

✎

```
#include <iostream>
using namespace std;

#define USING_CH
#include "seq.ch"
#include "open.ch"
using namespace CH;
```

✐

a) Der offene Typ `Expr` repräsentiert arithmetische Ausdrücke `x` unterschiedlicher Art, konkret konstante Ausdrücke mit Wert `x(value)`, Variablen mit Namen `x(name)`, unäre Ausdrücke mit Operator `x(oper)` (entweder `"+"` oder `"-"`) und Operand `x(body)` sowie binäre Ausdrücke mit Operator `x(oper)` (`"+"`, `"-"`, `"*"` oder `"/"`) und Operanden `x(left)` und `x(right)`.
Der Typ des Attributs `value` soll `double` sein, der Typ der Attribute `name` und `oper` soll `str` sein, und der Typ der Attribute `body`, `left` und `right` soll wiederum `Expr` sein.

Für einen `double`-Wert `v` liefert `expr(v)` einen konstanten Ausdruck mit Wert `v`.
Für einen Variablennamen `n` liefert `expr(n)` eine Variable mit dem Namen `n`.
Für einen Operatornamen `op` und einen Ausdruck `x` liefert `expr(op, x)` einen unären Ausdruck mit Operator `op` und Operand `x`.
Für einen Operatornamen `op` und Ausdrücke `x` und `y` liefert `expr(x, op, y)` einen binären Ausdruck mit Operator `op` und Operanden `x` und `y`.

Für einen Ausdruck `x` berechnet `eval(x)` den Wert des Ausdrucks und liefert ihn zurück.
Der Wert einer Variablen wird hierfür aus einer globalen Tabelle mit Typ `std::unordered_map<str, double>` ermittelt.

Für einen Ausdruck `x` liefert `fmt(x)` eine Zeichenkettendarstellung des Ausdrucks, z. B. `(1+(a*2))`.

✎

```
// Arithmetischer Ausdruck.
TYPE(Expr)

// Wert eines konstanten Ausdrucks.
ATTR1(value, Expr, double)

// Name einer Variablen.
ATTR1(name, Expr, str)

// Operator eines unären oder binären Ausdrucks.
ATTR1(oper, Expr, str)

// Operand eines unären Ausdrucks.
ATTR1(body, Expr, Expr)

// Operanden eines binären Ausdrucks.
ATTR1(left, Expr, Expr)
ATTR1(right, Expr, Expr)

// Konstanten Ausdruck mit Wert v liefern.
Expr expr (double v) {
    return Expr(value, v);
}

// Variable mit Namen n liefern.
Expr expr (str n) {
    return Expr(name, n);
}

// Unären Ausdruck mit Operator op und Operand x liefern.
Expr expr (str op, Expr x) {
    return Expr(oper, op)(body, x);
}

// Binären Ausdruck mit Operator op und Operanden x und y liefern.
Expr expr (Expr x, const str& op, Expr y) {
    return Expr(oper, op)(left, x)(right, y);
}

// Tabelle zur Abbildung von Variablennamen auf die Werte der Variablen.
unordered_map<str, double> tab;
```

```
// Ausdruck x auswerten.
double eval (Expr x) {
    if (x(oper) == "+") {
        // Unäres Plus oder Addition.
        if (x(body)) return eval(x(body));
        else return eval(x(left)) + eval(x(right));
    }
    else if (x(oper) == "-") {
        // Unäres Minus oder Subtraktion.
        if (x(body)) return -eval(x(body));
        else return eval(x(left)) - eval(x(right));
    }
    else if (x(oper) == "*") {
        // Multiplikation.
        return eval(x(left)) * eval(x(right));
    }
    else if (x(oper) == "/") {
        // Division.
        return eval(x(left)) / eval(x(right));
    }
    else if (x(name)) {
        // Variable.
        return tab[x(name)];
    }
    else {
        // Konstante.
        return x(value);
    }
}

// Zeichenkettendarstellung des Ausdrucks x.
str fmt (Expr x) {
    if (x(oper)) {
        // Unärer oder binärer Ausdruck.
        if (x(body)) return x(oper) + "(" + fmt(x(body)) + ")";
        else return "(" + fmt(x(left)) + x(oper) + fmt(x(right)) + ")";
    }
    else if (x(name)) {
        // Variable.
        return x(name);
    }
    else {
        // Konstante.
        char buf [20];
        sprintf(buf, "%g", x(value));
        return buf;
    }
}
```

b) Der offene Typ `Person` repräsentiert Personen `p` mit beliebig vielen Vornamen `p(firstnames)` und Nachname `p(name)`, jeweils mit Typ `str`.

Für eine Person `p` liefert `fmt(p, abbr)` eine Zeichenkettendarstellung der Person, die für `abbr` gleich `false` aus allen Vornamen und dem Nachnamen der Person, jeweils getrennt durch ein Leerzeichen, besteht. Für `abbr` gleich `true` wird jeder Vorname durch seinen ersten Buchstaben und einen Punkt ersetzt. Der Parameter `abbr` ist optional; wenn er fehlt, ist sein Wert `false`.

Ergänzen Sie ein einwertiges Attribut `spouse_` mit Zieltyp `Person`, das nicht direkt verwendet werden soll, sowie ein virtuelles einwertiges Attribut `spouse` mit folgender Bedeutung:
Für Personen `p` und `q` setzt `p(spouse, q)` oder `q(spouse, p)` den Ehepartner `p(spouse)` von `p` auf `q` und den Ehepartner `q(spouse)` von `q` auf `p`.
Wenn `p` und/oder `q` bereits einen anderen Ehepartner `pp` bzw. `qq` besitzt, wird die Verbindung zu diesem gelöst, d. h. `pp` und/oder `qq` hat anschließend keinen Ehepartner mehr.
`p` oder `q` kann auch nil sein, um lediglich den aktuellen Ehepartner von `q` bzw. `p` zu entfernen. (Wenn beide nil sind, sind die Operationen wirkungslos.)
Zum Beispiel:

```
Person p1 = true;
Person p2 = true;
Person q1 = true;
Person q2 = true;

p1(spouse, q1);          // p1 <-> q1
q2(spouse, p2);          // p2 <-> q2

p1(spouse, q2);          // p1 <-> q2, q1 und p2 haben keinen Ehepartner mehr
```

✎

```
// Person mit Name und Vornamen.
TYPE(Person)
ATTR1(name, Person, str)
ATTRN(firstnames, Person, str)

// Zeichenkettendarstellung der Person p, ggf. mit abgekürzten Vornamen.
str fmt (Person p, bool abbr = false) {
    str r;
    for (str s : p(firstnames)) {
        r += (abbr ? s(A|1) + "." : s) + " ";
    }
    return r += p(name);
}

// Ehepartner einer Person.
ATTR1(spouse_, Person, Person)

// Virtuelles Attribut spouse.
ATTR(spouse)

// Die Lesefunktion von spouse
// liefert den Wert des echten Attributs spouse_.
Person FUNC (spouse, Person p) {
    return p(spouse_);
}
```

```
    // Die Schreibfunktion von spouse verbindet p und q
    // über das echte Attribut spouse_ gleichzeitig in beide Richtungen.
    // Eventuell vorhandene Verbindungen von p oder q werden vorher gelöst.
    Person FUNC (spouse, Person p, Person q) {
        // Verbindung zu eventuell vorhandenen Ehepartnern lösen.
        // Wenn p oder q oder p(spouse_) oder q(spouse_) nil ist,
        // ist die entsprechende Operation wirkungslos.
        p(spouse_)(spouse_, nil);
        q(spouse_)(spouse_, nil);

        // Verbindung zwischen p und q in beiden Richtungen herstellen.
        // Wenn p oder q nil ist,
        // ist die entsprechende Operation wirkungslos.
        p(spouse_, q);
        q(spouse_, p);

        // "Aufrufobjekt" p zurückliefern.
        return p;
    }
```

Einfaches Testprogramm

```
// Elemente der Sequenz s, getrennt durch Leerzeichen,
// auf einer Zeile ausgeben.
template <typename T>
void print (seq<T> s) {
    const char* sep = "";
    for (T x : s) {
        cout << sep << x;
        sep = " ";
    }
    cout << endl;
}

// Hauptprogramm.
int main () {
    // Ausdruck 1 + (a * 2) erzeugen.
    Expr x = expr(expr(1), "+", expr(expr("a"), "*", expr(2)));

    // Wert der Variablen a auf 3 setzen.
    tab["a"] = 3;

    // Ausdruck und seinen Wert ausgeben.
    cout << fmt(x) << " = " << eval(x) << endl; // (1+(a*2)) = 7
```

```
    // Person erzeugen und vollständig bzw. abgekürzt ausgeben.
    Person carh = Person(firstnames, "Charles", "Anthony", "Richard")
                                                    (name, "Hoare");
    cout << fmt(carh) << endl;          // Charles Anthony Richard Hoare
    cout << fmt(carh, true) << endl;    // C. A. R. Hoare

    // Vier Personen.
    Person p1(name, "p1");
    Person p2(name, "p2");
    Person q1(name, "q1");
    Person q2(name, "q2");

    // p1 und q1 sowie p2 und q2 verheiraten
    // und alle Personen zur Kontrolle ausgeben.
    p1(spouse, q1);
    q2(spouse, p2);
    for (Person p : { p1, p2, q1, q2 }) {
        cout << p(name) << " -> " << p(spouse)(name) << endl;
    }                   // p1 -> q1 | p2 -> q2 | q1 -> p1 | q2 -> p2

    // p1 und q2 verheiraten
    // und alle Personen zur Kontrolle ausgeben.
    p1(spouse, q2);
    for (Person p : { p1, p2, q1, q2 }) {
        cout << p(name) << " -> " << p(spouse)(name) << endl;
    }                   // p1 -> q2 | p2 -> | q1 -> | q2 -> p1

    // nil und q2 "verheiraten"
    // und alle Personen zur Kontrolle ausgeben.
    Person()(spouse, q2);
    for (Person p : { p1, p2, q1, q2 }) {
        cout << p(name) << " -> " << p(spouse)(name) << endl;
    }                   // p1 -> | p2 -> | q1 -> | q2 ->
}
```

Aufgabe 3: Zentrale Datenstrukturen des MOSTflexiPL-Compilers

Arbeiten Sie sich anhand der Dokumentation in die zentralen Datenstrukturen des MOSTflexiPL-Compilers ein! (Teilabschnitt 1.1.6 ist momentan noch nicht wichtig.)

Folgende Typen und Operatoren seien bereits definiert:

```
#include "data.h"

// Metatyp type mit zugehörigem Operator.
Type type_type = uniq;
Oper type_oper = Oper(res_, type_type)(sig_, Sig(Part(name_, "type")));
Type dummy = type_type(type_, type_type)(oper_, type_oper);

// Typ bool mit zugehörigem Operator.
Oper bool_oper = Oper(res_, type_type)(sig_, Sig(Part(name_, "bool")));
Type bool_type = Type(type_, type_type)(oper_, bool_oper);

// Typ int mit zugehörigem Operator.
Oper int_oper = Oper(res_, type_type)(sig_, Sig(Part(name_, "int")));
Type int_type = Type(type_, type_type)(oper_, int_oper);
```

a) „Übersetzen“ Sie mit dem erworbenen Wissen die folgenden MOSTflexiPL-Deklarationen in entsprechende `Oper`-Objekte (ohne Werte für die in der Dokumentation ausgegrauten Attribute) mit zugehörigen `Part`-Hilfsobjekten:

```
a : bool;
b : bool;
c : int;
d : int;
(int) "+" (int) -> (int)
print (int) [dec|oct|hex|bin] -> (bool)
sum of (int) { and (int) } end -> (int)
dnf (bool) { and (bool) } { or (bool) { and (bool) } } end -> (bool)
```

Zum Beispiel:

```
// Nachträgliche Änderung gegenüber dem ursprünglichen Aufgabenblatt:
// Die Namen aller Attribute enden mit einem Unterstrich.

// Konstante a mit Typ bool.
Oper a = Oper(res_, bool_type)(sig_, Sig(Part(name_, "a")));

// Anonyme Parameter p und q mit Typ int.
Par p = Par(res_, int_type);
Par q = +p;

// Binärer Plus-Operator.
Oper plus = Oper(res_, int_type)
    (sig_, Sig(Part(par_, p), Part(name_, "+"), Part(par_, q)));

// Variadischer sum-Operator.
Oper sum = ......
```

Definieren Sie sich bei Bedarf geeignete Hilfsobjekte und -funktionen zur Vereinfachung!

✎

```
// Konstante b mit Typ bool.
Oper b = Oper(res_, bool_type)(sig_, Sig(Part(name_, "b")));

// Konstanten c und d mit Typ int.
Oper c = Oper(res_, int_type)(sig_, Sig(Part(name_, "c")));
Oper d = Oper(res_, int_type)(sig_, Sig(Part(name_, "d")));

// Hilfsfunktionen zur Erzeugung von Signaturteilen aus irgendwelchen
// Dingen (Namen, Parameter oder Signaturteile).
Part part (str name) { return Part(name_, name); }
Part part (Par par) { return Part(par_, par); }
Part part (Part part) { return part; }

// Hilfsfunktionen zur Erzeugung von Signaturen aus irgendwelchen
// Dingen.
template <typename ... TT>
Sig sig (TT ... xx) { return Sig(part(xx) ...); }
Sig sig (Sig sig) { return sig; }

// Optionales bzw. wiederholbares Signaturteil mit einer Alternative
// liefern, die aus irgendwelchen Dingen besteht.
template <typename ... TT>
Part opt (TT ... xx) { return Part(opt_, true)(alts_, sig(xx ...)); }
template <typename ... TT>
Part rep (TT ... xx) { return opt(xx ...)(rep_, true); }

// Signaturteil liefern, das aus den Alternativen xx ... besteht.
template <typename ... TT>
Part alts (TT ... xx) { return Part(alts_, sig(xx) ...); }

// Operator mit Resultattyp res erzeugen,
// dessen Signatur aus irgendwelchen Dingen besteht.
template <typename ... TT>
Oper oper (Type res, TT ... xx) { return Oper(sig_, sig(xx ...)); }

// Ausgabe-Operator.
Oper pr = oper(bool_type, "print", p,
                       alts("dec", "oct", "hex", "bin")(opt_, true));

// Variadischer sum-Operator.
Oper sum = oper(int_type, "sum", "of", p, rep("and", q), "end");

// Disjunktive Normalform.
Par p1 = Par(res_, bool_type), p2 = +p1, p3 = +p1, p4 = +p1;
Oper dnf = oper(bool_type, p1, rep("and", p2),
                           rep("or", p3, rep("and", p4)), "end");
```

✐

b) „Übersetzen“ Sie entsprechend die folgenden MOSTflexiPL-Ausdrücke in entsprechende `Expr`-Objekte (ohne Werte für die in der Dokumentation ausgegrauten Attribute) mit zugehörigen `Pass`- und `Item`-Hilfsobjekten, die bei Bedarf die zuvor erstellen `Oper`-Objekte verwenden:

```
c
c + d
print c
print d hex
sum of c and d end
sum of c and c + d and d end
sum of sum of c and d end and sum of d and c end end
dnf a or a and b or a and b and print c end
```

Zum Beispiel:

```
// Nachträgliche Änderungen gegenüber dem ursprünglichen Aufgabenblatt:
// Die Namen aller Attribute enden mit einem Unterstrich.
// Statt branches_ heißt es jetzt branch_.

// Verwendung der Konstanten c und d.
// Hier fehlte im ursprünglichen Aufgabenblatt jeweils (row_, ...).
Expr c_ = Expr(oper_, c)(type_, int_type)(row_, Row(Item(word_, "c")));
Expr d_ = Expr(oper_, d)(type_, int_type)(row_, Row(Item(word_, "d")));

// c + d
Expr c_plus_d = Expr(oper_, plus)(type_, int_type)
    (row_, Row(Item(opnd_, c_), Item(word_, "+"), Item(opnd_, d_)));

// sum of c and c + d and d end
Expr sum_of_c_and_c_plus_d_and_d_end =
    Expr(oper_, sum)(type_, int_type)(row_, Row(
        Item(word_, "sum"),
        Item(word_, "of"),
        Item(opnd_, c_),
        Item(passes_,
            Pass(choice_, A)(branch_, Row(
                Item(word_, "and"),
                Item(opnd_, c_plus_d))),
            Pass(choice_, A)(branch_, Row(
                Item(word_, "and"),
                Item(opnd_, d_)))
        ),
        Item(word_, "end")
    ));
```

✎

```
// Hilfsfunktionen zur Erzeugung von Einträgen aus irgendwelchen Dingen
// (Zeichenfolgen, Operanden, Folgen von Durchläufen oder Einträge).
Item item (str word) { return Item(word_, word); }
Item item (Expr opnd) { return Item(opnd_, opnd); }
Item item (seq<Pass> passes) { return Item(passes_, passes); }
Item item (Item item) { return item; }

// Hilfsfunktionen zur Erzeugung von Reihen aus irgendwelchen Dingen.
template <typename ... TT>
Row row (TT ... xx) { return Row(item(xx) ...); }
Row row (Row row) { return row; }
```

```
// Durchlauf mit Position choice liefern,
// dessen Reihe aus irgendwelchen Dingen besteht.
template <typename ... TT>
Pass pass (posA choice, TT ... xx) {
    return Pass(choice_, choice)(branch_, row(xx ...));
}

// Ausdruck mit Operator oper liefern,
// dessen Hauptreihe aus irgendwelchen Dingen besteht.
template <typename ... TT>
Expr expr (Oper oper, TT ... xx) {
    return Expr(oper_, oper)(type_, oper(res_))(row_, row(xx ...));
}

// print c
Expr print_c = expr(pr, "print", c_, Item(uniq));

// print d hex
Expr print_d_hex = expr(pr, "print", d_, seq(pass(3*A, "hex")));

// sum of c and d end
Expr sum_of_c_and_d_end =
           expr(sum, "sum", "of", c_, seq(pass(A, "and", d_)), "end");

// sum of sum of c and d end and sum of d and c end end
Expr sum_of_d_and_c_end =
           expr(sum, "sum", "of", d_, seq(pass(A, "and", c_)), "end");
Expr sum_of_sum = expr(sum, "sum", "of", sum_of_c_and_d_end,
                       seq(pass(A, "and", sum_of_d_and_c_end)), "end");

// dnf a or a and b or a and b and print c end
Expr a_ = Expr(oper_, a)(type_, bool_type)(row_, Row(Item(word_, "a")));
Expr b_ = Expr(oper_, b)(type_, bool_type)(row_, Row(Item(word_, "b")));
Expr dnf_ = expr(dnf, "dnf", a_, seq<Pass>(),
    seq(pass(A, "or", a_, seq(pass(A, "and", b_))),
        pass(A, "or", a_, seq(pass(A, "and", b_)),
                          seq(pass(A, "and", print_c)))), "end");
```

✎

Einfaches Testprogramm

```
#include <iostream>
using namespace std;

#include "parser.h"

int main () {
    // Die erzeugten Ausdrücke zur Kontrolle
    // mit der Funktion print des Parsers ausgeben.
    for (Expr expr :
                { c_, c_plus_d, print_c, print_d_hex,
                  sum_of_c_and_d_end, sum_of_c_and_c_plus_d_and_d_end,
                  sum_of_sum, dnf_ }) {
        print(expr);
        cout << endl;
    }
}
```

✎