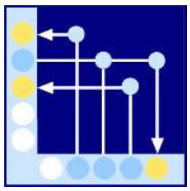
Hochschule Aalen
Fakultät Elektronik und Informatik
Studienbereich Informatik



Compilerbau-Praktikum

Lehrveranstaltung im Wintersemester 2022/2023
Prof. Dr. habil. Christian Heinlein

3. Aufgabenblatt (18. Oktober 2022)

Aufgabe 3: Zentrale Datenstrukturen des MOSTflexiPL-Compilers

Arbeiten Sie sich anhand der Dokumentation in die zentralen Datenstrukturen des MOSTflexiPL-Compilers ein! Wichtig sind im Moment die Abschnitte 1.1 bis 1.5.

a) „Übersetzen“ Sie mit dem erworbenen Wissen die folgenden MOSTflexiPL-Deklarationen in entsprechende `Oper`-Objekte mit zugehörigen `Part`- und `Par`-Hilfsobjekten! Die Typen der Operatoren und Parameter können momentan noch nicht ausgedrückt werden. Die in der Dokumentation ausgegrauten Attribute werden momentan noch nicht benötigt.

```
a : bool;
b : bool;
c : int;
d : int;
(p:int) "+" (q:int) -> (int = ...)
print (p:int) [dec|oct|hex|bin] -> (bool = ...)
sum of (p:int) { and (q:int) } end -> (int = ...)
dnf (p1:bool) { and (p2:bool) } { or (p3:bool) { and (p4:bool) } } end
                                                    -> (bool = ...)
```

Zum Beispiel:

```
#include "data.h"

// Konstante a.
Oper a = Oper(sig_, Part(name_, "a"))(stat_, true);

// Parameter p und q.
Par p = Par(sig_, Part(name_, "p"))(stat_, true);
Par q = Par(sig_, Part(name_, "q"))(stat_, true);

// Binärer Plus-Operator.
Oper pl = Oper(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!

✎

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

// Konstanten b, c und d.
Oper b = oper("b")(stat_, true);
Oper c = oper("c")(stat_, true);
Oper d = oper("d")(stat_, true);

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

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

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

✐

b) „Übersetzen“ Sie entsprechend die folgenden MOSTflexiPL-Ausdrücke in entsprechende `Expr`-Objekte mit zugehörigen `Pass`- und `Item`-Hilfsobjekten, die bei Bedarf die zuvor erstellen `Oper`-Objekte verwenden! Auch hier werden die in der Dokumentation ausgegrauten Attribute momentan noch nicht benötigt.

```
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:

```
// Verwendung der Konstanten c und d.
Expr c_ = Expr(oper_, c)(row_, Item(word_, "c"));
Expr d_ = Expr(oper_, d)(row_, Item(word_, "d"));

// c + d
Expr c_plus_d = Expr(oper_, pl)
    (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)(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)(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)(row_, Item(word_, "a"));
Expr b_ = Expr(oper_, b)(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");
```

✎

c) Implementieren Sie eine Funktion

```
void print (Expr expr, str ind = "");
```

die den Ausdruck `expr` baumartig mit Einrückung `ind` (eine zunächst leere Folge von Leerzeichen) ausgibt!

Idee: Durchlaufen Sie die Einträge der Reihe `expr(row_)` und geben Sie jeden Eintrag wie folgt aus:

- Wenn der Eintrag ein Wort enthält, wird es mit Einrückung `ind` auf einer Zeile ausgegeben.
- Wenn der Eintrag einen Operanden enthält, wird er rekursiv mit zwei Leerzeichen mehr Einrückung ausgegeben.
- Andernfalls enthält der Eintrag die Durchläufe durch eine Klammer. Jeder Durchlauf enthält in seinem Attribut `branch_` die Teilreihe für diesen Durchlauf, die wiederum rekursiv (mit unveränderter Einrückung) ausgegeben werden kann.

Für den Ausdruck `sum of c and c + d and d end` lautet die Ausgabe von `print` zum Beispiel:

```
sum
of
  c
and
    c
  +
    d
and
  d
end
```

✎

```
// Reihe row mit Einrückung ind ausgeben.
void print (Row row, str ind) {
    for (Item item : row) {
        if (str word = item(word_)) {
            cout << ind << word << endl;
        }
        else if (Expr opnd = item(opnd_)) {
            print(opnd(row_), ind + "  ");
        }
        else {
            for (Pass pass : item(passes_)) {
                print(pass(branch_), ind);
            }
        }
    }
}

// Ausdruck expr mit Einrückung ind ausgeben.
void print (Expr expr, str ind = "") {
    print(expr(row_), ind);
}
```

✎

✎

Einfaches Testprogramm

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

int main () {
    // Die erzeugten Ausdrücke mit der Funktion print 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;
    }
}
```

✎