C/C++ Materialpaket (Level C)
02c_FLOW_parse – Control Flow (Parse)

Prof. Dr. Carsten Link

Zusammenfassung

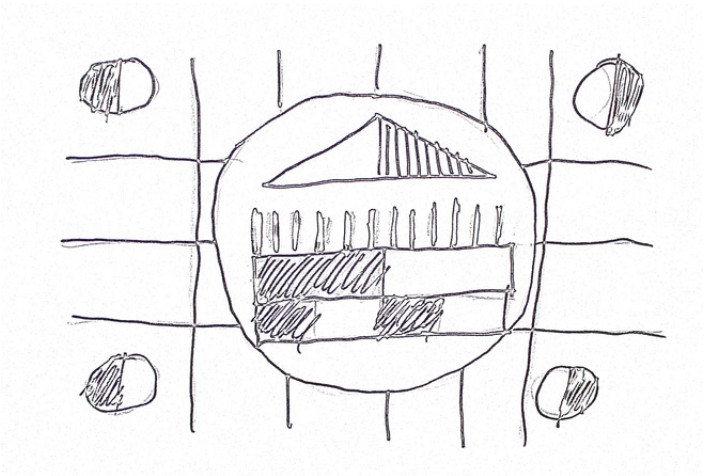

Abbildung 1: Testbild

Inhaltsverzeichnis

1 Kompetenzen und Lernegebnisse

Durch das Bearbeiten dieses Materialpaketes erwerben Sie diese Kompetenzen (Wissen, Fähigkeiten und Fertigkeiten zur Problemlösung):

Sie können einen Parser für einfache kontextfreie Grammatiken implementieren.

Die oben genannten Kompetenzen erwerben Sie, indem Sie Lernziele erreichen, welche sich prüfen lassen. Lernegebnisse: Sie können nachweislich[1]:

- erläutern, wie Stacks für Berechnungen eingesetzt werden kann (Rechenmodell)
- Erweiterte Backus-Naur-Form lesen
- einen einfachen rekursiv absteigenden Parser implementieren

2 Konzepte

Im Folgenden wird auf die Kontrollstruktur *Rekursion* eingegangen. Besonders wird beleuchtet, wie Rekursion sich besonders eignet, Berechnungen durchzuführen.

2.1 Stapel als Rechenmodell

In Physik und Mathematik hat sich die mathematische Notation[2] als Schreibweise durchgesetzt. Im einfachsten Fall werden Zahlen, Variablen und Konstanten mit Operatoren zu einem Ergebnis verknüpft. Hierbei haben Operatoren jeweils Prioritäten (Wertigkeit, Rangfolge), welche festlegen, in welcher Reihenfolge Berechnungen durchzuführen sind.

Der Ausdruck $1 + 2 * 3^4$ entspricht diesem Ausdruck ohne implizite Operatorprioritäten: $(1 + (2 * (3^4)))$ oder (von links nach rechts gerechnet) $3^4 * 2 + 1$. Operatorprioritäten regeln also, in welcher Reihenfolge Operatoren auf Operanden angewendet werden.

Ein Mensch berechnet solche Ausdrücke womöglich, indem er einen Operator mit höchster Priorität sucht, diesen auf dessen Operanden anwendet und das ganze durch das Ergebnis ersetzt. Dies wird so oft wiederholt, bis ein Endergebnis erhalten wird. Für Programme bzw. wissenschaftliche Taschenrechner ist dieses Verfahren viel zu aufwändig. Hier wird mit dem einfachen Stapel-Rechenmodell gearbeitet. Ein Stapel ist eine Liste, die nur an einem Ende bearbeitet wird; Elemente werden also nur an ein Ende Angefügt und von diesem Ende wieder entfernt - also genau wie ein Stapel mit Zetteln.

[1] Sie können das Erzielen der einzelnen Lernergebnisse beispielsweise bei einem Testat im Praktikum oder einer Aufgabe in der Modulprüfung nachweisen.

[2] https://en.wikipedia.org/wiki/Mathematical_notation

Es werden zwei Stapel verwendet (siehe auch Abbildung 1): einer um Operanden zu speichern und einer, um Operatoren zu speichern, falls keine sofortige Berechnung möglich sein sollte (dies geschieht, wenn ein Operator mit höherem Rang folgt). Ist die Berechnung möglich, so kann der oberste Operator auf die obersten Operatoren angewendet werden. Dabei wird der Operator entfernt und die beiden Operatoren werden durch das Ergebnis ersetzt.

Ein Beispiel: Für den Ausdruck $1 + 2 * 3^4$ sehen die beiden Stapel wie folgt aus:

```
1

2
1    +

3
2    *
1    +

4
3    ^
2    *
1    +


81
2    *
1    +

162
1    +

163  =
```

Hingegen bilden sich bei Ausdrücken mit homogener Operatorenpriorität kein große Stapel – hier der Ausdruck $1 + 2 + 3 + 4$:

```
1

2
1    +

3

3
3    +

6
```

```
12
13  4
14  6    +
15
16  10   =
```

Experimentieren Sie zur Übung mit einem wissenschaftlichen Taschenrechner und beobachten, unter welchen Umständen der Taschenrechner ein Zwischenergebnis berechnet.

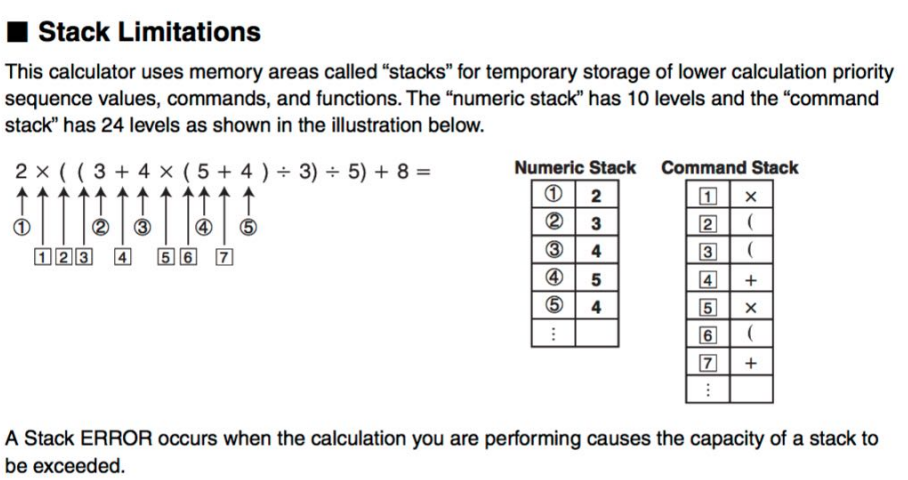

Abbildung 2: Wissenschaftliche Taschenrechner verwenden zwei Stapelspeicher, um Operanden und Operatoren zwischenzuspeichern, falls diese aufgrund von Operatorenrangfolge noch nicht verwendet werden können (aus [CasioFX50]). Klammern erhöhen die Prioritäten der umschlossenen Operatoren.

Der Großteil der imperativen und objektorientierten Programmiersprachen ist mittels eines call stacks implementiert, welcher die beiden Stapel Parameter Stack und Return Stack vereint (siehe Wikipedia: Call Stack[3]).

Es ist sehr wichtig, verinnerlicht zu haben, wie dieser Call Stack arbeitet, um nachvollziehen zu können, wie Berechnungen innerhalb eines Programms zur Laufzeit ablaufen. Der Leser soll hierzu eine Vorstellung entwickeln, wie sich der statische Quelltext zur Laufzeit dynamisch verhält.

Auf den Call Stack der Programmiersprache C++ wird in den nachfolgenden Kapiteln eingegangen.

[3]https://en.wikipedia.org/wiki/Call_stack

2.2 Erweiterte Backus-Naur-Form (EBNF)

Um die Grammatik von einfachen formalen Sprachen beschreiben (spezifizieren) zu können hat sich die erweiterte Backus-Naur-Form (EBNF[4]) etabliert, die Teile der Wirth-Syntax-Notation[5] in sich vereint. Diese Schreibweise soll im Folgenden verwendet werden, da diese sich in Teilen mit der Schreibweise für reguläre Ausdrücke deckt.

Die Grammatik einer Sprache wird durch eine Menge von *Produktionen* definiert. Diese Regeln werden Produktionen genannt, da man bei dieser Sprechweise davon ausgeht, dass ein Generator mit diesen Regeln Wörter der durch die Grammatik beschriebenen Sprache erzeugen kann. Sie eignen sich allerdings auch dazu, Wörter und Teile davon zu erkennen.

Kontextfreie Grammatiken in Wirth-Syntax-Notation sind mächtiger als reguläre Ausdrücke. Das heißt: reguläre Ausdrücke lassen sich darin auch beschreiben. Darüber hinaus lassen sich Sprachen beschreiben, welche Rekursion aufweisen (insbes. geklammerte Ausdrücke). Einfach gesagt, besteht dann eine Produktion P aus Kombinationen, welche eine andere Produktion enthält, die wiederum P enthält. Die Grammatik von Programmiersprachen weist in der Regel Rekursion auf. Dieser Sachverhalt ist bereits bei einfachen arithmetischen Ausdrücken vorhanden. In einem arithmetischen Ausdruck kann in Klammern ein weiterer arithmetischer Ausdruck enthalten sein. Beispielsweise ist in $a * (b + c)$ der Teil $b + c$ wieder ein vollständiger arithmetischer Ausdruck.

Im Folgenden wird ein Parser vorgestellt, der einfache arithmetische Ausdrücke erkennen kann. Darüber hinaus kann dieser Parser während des Erkennungsvorgangs den Wert des Ausdrucks berechnen.

Parser orientieren sich anhand eines Stroms von sogenannten Token. Bei unserem einfachen Parser ist es ausreichend, das jeweils nächste Token zu betrachten. Token haben einen der folgenden Werte:

```
enum Token_value {
    NUMBER, // '0' ... '9'
    PLUS='+',   MINUS='-',  MUL='*',    DIV='/',
    PRINT='\n', POWER='#',  LP='(',     RP=')'
};
```

Mittels dieser Token läßt sich nun die Grammatik für einfache arithmetische Ausdrücke aufschreiben:

```
start:
  expression PRINT { expression PRINT }    // PRINT is \n

expression:
  term { '+' | '-' term }
```

[4] https://en.wikipedia.org/wiki/Extended_Backus–Naur_form
[5] https://en.wikipedia.org/wiki/Wirth_syntax_notation

```
6
7   term:
8     power { '*' | '/' power }
9
10  power:
11    primary { '#' primary }         // char '^' produces problems with terminal input
12
13  primary:
14     NUMBER | '(' expression ')'
```

In der Grammatik sind

- `start` das Startsymbol
- `NUMBER` ganze Zahlen aus den Ziffern `0...9`
- Ein `primary` ist entweder eine Zahl oder ein in Klammern gegebener Ausdruck. Hier ist die Rekursion in der Sprache ersichtlich (Ein Ausdruck enthält einen Ausdruck enthält einen Ausdruck enthält …).

Um Berechnungen mit geklammerten Ausdrücken durchführen zu können, ist der bereits erwähnte Call Stack nützlich. Auf diesem werden Übergabe- und Zwischenwerte gespeichert. Ebenso wird dort gespeichert, an welcher Stelle im Code die Ausführung fortgesetzt wird nachdem eine Zwischenberechnung abgeschlossen wurde.

2.3 Rekursiv absteigender Parser

Es gibt verschiedene Möglichkeiten, einen Parser für eine Grammatik zu konstruieren; Ein rekursiv absteigender Parser ist die einfachste Implementierungsform. Aus einer EBNF-Grammatik läßt sich ein rekursiv absteigender Parser konstruieren, indem für jede Produktion eine (gleichnamige) Funktion erstellt wird:

```
100 unsigned int primary();     // Numbers or ( expression )
101 unsigned int power();       // x ^ y
102 unsigned int term();        // multiply or divide
103 unsigned int expression()   // add or subtract
```

Unser Parser für arithmetische Ausdrücke weist eine extrem einfache Struktur auf[6]: es gibt keine Variablen (die syntaktisch von Zahlen unterschieden werden müssen und mit Zuweisungsoperator einhergehen) und Operatoren haben immer zwei Operanden in Infixnotation (der Operator steht zwischen den Operanden). Aus diesen Gründen haben die Funktionen `expression`, `term` und `power` dieselbe Struktur (Pseudo-Code):

```
1 unsigned int ops_level_a()              // e.g. add or subtract
2 {
3     unsigned int left = ops_level_b();  // e.g. multiply or divide (which in turn calls ops_
```

[6] siehe https://karmin.ch/ebnf/examples für komplexere arithmetische Ausdrücke

```
    unsigned int right = 0;

    for (;;)
        switch (global_token) {
            case TOKEN_LEVELA_1:{
                right = ops_level_b();  // e.g. multiply or divide
                left = left TOKEN_LEVELA_1 right;
                break;
            }
            case TOKEN_LEVELA_2:{
                right = ops_level_b();  // e.g. multiply or divide
                left = left TOKEN_LEVELA_1 right;
                break;
            }
            default:         // an operator of another (lower) level, or \n
                return left; // there is no further operator (and operands) of this level
        }
}
```

Es ist in obigem Code ersichtlich, dass jede Operatorfunktion (hier `ops_level_a()`) zunächst mittels der höherwertigen Operatorfunktion den Wert des linken Operanden ermittelt (`left = ops_level_b()`, dies sorgt implizit für die Einhaltung der Operatorenrangfolge). Falls nach dem linken Operanden ein weiterer folgt, wird dessen Wert mittels der höherwertigen Operatorfunktion ermittelt und `right` zugewiesen. Das Statement `switch (global_token)` unterscheidet die Fälle `+,-` beziehungsweise `/,*`.

Hier der vollständige Code der Operatorfunktionen:

```
int main()
{
    while (cin){
        cout << expression() << '\n';
    }
    return no_of_errors;
}
```

```
unsigned int expression()       // add or subtract
{
    unsigned int left = term();
    unsigned int right = 0;

    for (;;)                // ``forever''
        switch (global_token) {
            case PLUS:{
                right = term();
                if (doPrintCalculations) printf("%u + %u\n", left, right);
```

```
                left += right;
                break;
            }
            case MINUS:{
                right = term();
                if (doPrintCalculations) printf("%u - %u\n", left, right);
                left -= right;
                break;
            }
            default:
                return left;
        }
}
```

```
unsigned int term()     // multiply or divide
{
    unsigned int left = power();
    unsigned int right = 0;

    for (;;)
        switch (global_token) {
            case MUL:{
                right = power();
                if (doPrintCalculations) printf("%u * %u\n", left, right);
                left *= right;
                break;
            }
            case DIV:{
                if ((right = power())) {
                    if (doPrintCalculations) printf("%u / %u\n", left, right);
                    left /= right;
                    break;
                }
                return error("divide by 0");
            }
            default:
                return left;
        }
}
```

```
unsigned int power()        // 2 ^ 3
{
    unsigned int left = primary();
    unsigned int right = 0;

    for (;;)
```

```
        switch (global_token) {
            case POWER:{
                right = primary();
                if (doPrintCalculations) printf("%u # %u\n", left, right);
                unsigned int base = left;
                left = 1;
                for (int i=0; i<right; i++)
                    left *= base;
                break;
            }
            default:
                return left;
        }
}
```

```
unsigned int primary()      // handle primaries
{
    Token_value current_token = get_token();

    switch (current_token) {
        case NUMBER:{
            unsigned int v = number_value;
            get_token();        // proceed global_token to next token (i.e. operator or '\n
            return v;
        }
        case LP:{
            unsigned int e = expression();
            if (global_token != RP) return error(") expected");
            get_token();        // eat ')' in order to proceed global_token to next token
            return e;
        }
        default:
            return error("primary expected");
    }
}
```

Wenn die globale Variable `doPrintCalculations` den Wert `true` hat, gibt der Parser die Zwischenberechnungen aus.

Aufgabe: Kompilierieren Sie die Datei `AP2-dc.cpp` und experimentieren Sie mit verschiedenen einfachen Ausdrücken, um zu verstehen, zu welchen Zeitpunkten welche Berechnungen durchgeführt werden: `1+1+1+1`, `2*2*2*2`, `1+2*3#4`, `1+2*3#4+5`.

Wichtig: Es ist zu beachten, dass jede der Funktionen `expression()`, `term()`, `power()` über eigene Exemplare (Ausprägungen, Instanzen) der lokalen Variablen `left` und `right` verfügt, solange sie ausgeführt wird (der Compiler sorgt dafür,

dass diese lokalen Variablen angelegt und freigegeben werden). Nicht so offensichtlich ist die Tatsache, dass sich dies insbesondere bei `expression()` so verhält: wenn der Parser einen Ausdruck verarbeitet, welcher eingeklammerte Teilausdrücke enthält (z.B. `1+(1+(1+1))`), so wird `expression()` in `primary()` erneut aufgerufen – es wird also eine weitere Ausprägung der Funktion `expression()` gestartet, obwohl diese bereits in Ausführung ist (indirekte Rekursion). Jede dieser `expression()`-Ausprägungen (-Aktivierungen) verfügt über einen eigenen Satz von lokalen Variablen (und Übergabeparametern). Ähnlich wie beim wissenschaftlichen Taschenrechner wird hier der Stapel als Rechenmodell verwendet. Der Compiler sorgt hierbei für das Stapeln der Werte. Siehe hierzu auch [WikipediaCallStack].

```
1
```

3 Material zum aktiven Lernen

Regelmäßiger Hinweis: Da eine Programmiersprache nur durch aktive Verwendung erlernt werden kann, werden im Folgenden Aufgaben zum praktischen Üben vorgestellt. Zunächst wird ein Grundgerüst (C/C++Programm) erstellt, welches dann auf mehrere Arten modifiziert wird. Insbesondere die Modifikationen ermöglichen es dem Lernenden (und auch dem Lehrenden), die Qualität des Kompetenzerwerbs bzgl. dieses Materialpakets bewerten zu können.

3.1 Aufgabe: Grundgerüst

Nehmen Sie die Datei `AP2-dc.cpp` als Vorlage. Erweitern Sie diese:

1. Es sollen die Ziffern `o` (kleines o für ein nicht gesetztes Bit) und `i` (kleines i für ein gesetztes Bit) erkannt werden
2. Es soll möglich sein, binäre Bitmuster einzugeben (z.B. `oioiiioo`)
3. Die Operatoren `&`, `v` sollen für Und- sowie Oder-Verknüpfung verwendbar sein

3.2 Aufgabe: Modifikationen

Regelmäßiger Hinweis: Weiter unten ist eine Liste mit Modifikationen gegeben, die zwei Zwecken dienen: 1) Sie dienen als Richtschnur für das Praktizieren und Üben der Inhalte dieses Materialpakets. 2) Die Modifikationen können im Rahmen eines Testats als Aufgabe verwendet werden, durch deren Lösung Studierende nachweisen können, dass sie den Stoff dieses Materialpakets beherrschen. Stellen Sie sicher, dass Sie jede einzelne der nachfolgenden Modifikationen innerhalb weniger Minuten (ca. 5 - 10) vor Zuschauern (Testatsituation) umsetzen können. Konkret sollen Sie im Testat in der Lage sein, das gegebene Grundgerüst um mindestens eine zufällig ausgewählte Modifikation zu erweitern. Bereiten Sie dazu auf ihrer Arbeitsumgebung ein Verzeichnis vor, welches ausschließlich das Grundgerüst enthält. **Arbeiten Sie also auf einer Kopie des Verzeichnisses `src-cpp-student/`!** Achten Sie darauf, dass der Text auf Ihrem Bildschirm in

heller Umgebung aus einem Meter Abstand heraus gut lesbar ist (light mode, große Schrift).

Modifikationen:

1. Der Operator `$` soll für die Verknüpfung exclusiv Oder verwendbar sein
2. Zahlen können mit vorangestelltem `x` auch hexadezimal eingegeben werden. Erstellen Sie hierzu die Funktion `AsciiHexToInt()` und verwenden diese
3. In `asciToInt()` soll eine `while`-Schleife verwendet werden
4. Ändern Sie `power()` so, dass der Operator `#` rechtsassoziativ ist (also $a^{b^c} = a^{(b^c)}$)
5. Die Ausgabe soll binär erfolgen. Verwenden Sie eine selbst geschriebene Funktion hierzu.
6. Die Ausgabe soll hexadezimal erfolgen. Verwenden Sie eine selbst geschriebene Funktion hierzu.

3.3 Verständnisfragen

Nach Bearbeitung des Kapitels "Konzepte", der Erstellung des Grundgerüsts sowie dem Üben der Modifikationen sollten Sie in der Lage sein, die folgenden Fragen zu beantworten.

1. Geben Sie die Reihenfolge der Berechnungen der Teilergebnisse eines wissenschaftlichen Taschenrechners an bei der Eingabe dieser Ausdrücke: $1 + 2 + 3 + 4 =$, $2 + 3 * 4^5 =$, $3 + 4^5 * 2 =$
2. Ersetzen Sie `a` bis `d` durch `+, -, *, /`, so dass sich die gewohnte Operatorenrangfolge für arithmetische Ausdrücke ergibt:

```
expression:
  expression { 'a' | 'b' term }

term:
  factor { 'c' | 'd' factor }

factor:
   NUMBER | '(' expression ')'
```

4. Ersetzen Sie `a` bis `c` durch angemessene Bezeichner, so dass sich eine Grammatik für einfache arithmetische Ausdrücke ergibt, welche über die gewohnte Operatorenrangfolge verfügt:

```
a:
  a { '+' | '-' b }

b:
  c { '*' | '/' c }

c:
   NUMBER | '(' a ')'
```

5. Freiwillig: Zeigen Sie auf, inwiefern das Piktogram auf der Titelseite dieses Materialpaketes den Inhalt zusammengefasst darstellt.

4 Nützliche Links

- OSU CSE Recursive-Descent Parsing: http://web.cse.ohio-state.edu/software/2231/web-sw2/extras/slides/27.Recursive-Descent-Parsing.pdf
- Wikipedia, Recursive_descent_parser: https://en.wikipedia.org/wiki/Recursive_descent_parser
- Boost Spirit (Parserspezifiation per C++-Templates): http://boost-spirit.com/home/

5 Literatur

- [PPP] Stroustrup, Bjarne: Programming - Principles and Practice using C++
- [TCPL] Stroustrup, Bjarne: The C++ Programming Language, Fourth Edition
- [CasioFX50] Casio: fx-50 Users Guide, http://support.casio.com/storage/en/manual/pdf/EN/004/fx-50F_PLUS_EN.pdf
- [WikipediaCallStack] Wikipedia Aufrufstapel: https://de.wikipedia.org/wiki/Aufrufstapel