universität wien

Ein einfacher Rechner

Aufgabenstellung: Erstellen Sie ein Programm, das simple Ausdrücke (z.B. 2+3*1.5) einliest, berechnet und das Ergebnis ausgibt.

Die Werte haben den Datentyp double. Erlaubte Operatoren: + - * / % ( ). Es gelten die aus der Mathematik bekannten Regeln bezüglich Assoziativität.

Wie würden Sie vorgehen?

universität wien

Bevor wir loslegen

Das Problem ist nicht besonders neu. Es gibt also höchstwahrscheinlich schon Lösungen.

Allgemein werden wir versuchen, das Rad nicht immer wieder neu zu erfinden und eventuell existierende Lösungen verwenden bzw. adaptieren.

universität wien

Verwenden von vorhandenen Lösungen

Das Problem ist nicht besonders neu. Es gibt also höchstwahrscheinlich schon Lösungen.

Allgemein werden wir versuchen, das Rad nicht immer wieder neu zu erfinden und eventuell existierende Lösungen verwenden bzw. adaptieren.

Vorbehalt bei Übungsbeispielen:

- Lösung nachschlagen und kopieren 👎
- Eigene Lösung versuchen (eventuell scheitern) 👍
- Danach mit bekannten Lösungen und Alternativen vergleichen 👍
- Eventuell nachbessern 👍

Lexer und Parser

Es hat sich als vorteilhaft erwiesen, das Problem in zwei Stufen zu teilen:

Lexer:

Übernimmt die lexikalische Analyse. Trennt die einzelnen "Wörter" (sinntragende Einheiten) voneinander, z.B.: 5.2, + etc.

Liefert das Ergebnis in Form von sogenannten **Token** (das ist jeweils eine Kombination aus Typ und Wert) weiter, z.B. "Zahl/5.2" "Operator +/" (wie das letzte Beispiel zeigt, haben manche Token keinen Wert).

Parser:

Übernimmt die Analyse des gesamten Ausdrucks. Ergebnis ist in der Regel eine Datenstruktur (AST abstract syntax tree) die dann z.B. von einem Compiler weiterverwendet werden kann. Wir werden aber hier gleich das numerische Ergebnis des Ausdrucks ermitteln.

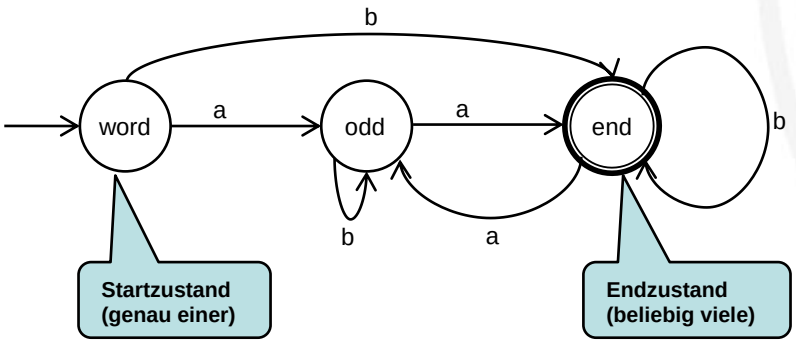
Beide bauen in der Regel auf vorgegebenen Grammatiken (vgl. EBNF aus der Vorlesung) auf.

Eine simple Grammatik

word = `'b'` end
word = `'a'` odd
odd = `'b'` odd
odd = `'a'` end
end = `'a'` odd
end = [`'b'` end]

word = `'b'` end | `'a'` odd
odd = `'b'` odd | `'a'` end
end = [(`'b'` end) | (`'a'` odd)]

Solche einfachen Grammatiken, bei denen (in der langen Fassung) links nur ein Nonterminalsymbol und rechts immer nichts oder ein Terminalsymbol eventuell gefolgt von einem Nichtterminalsymbol steht, heißen (rechts)**reguläre Grammatik**en. Sie lassen sich auch als endlicher Automat (FSM finite state machine) interpretieren, der wieder sehr einfach grafisch dargestellt und implementiert werden kann:



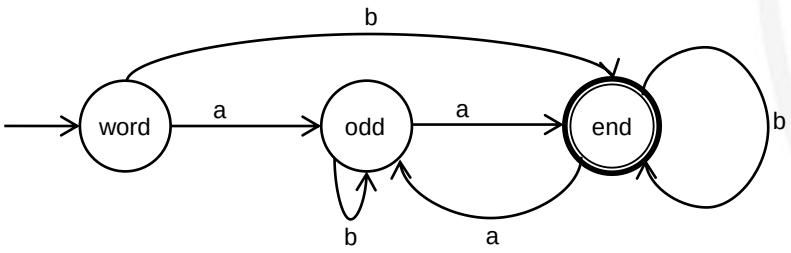

```
while (cin >> symbol) {
  switch (status) {
    case State::word:
      switch (symbol) {
        case 'a': status = State::odd; break;
        case 'b': status = State::end; break;
        default: throw runtime_error("ERROR");
      }
      break;
    ...
  }
}
```

Abbruch und Akzeptieren von Worten

Im mathematischen Modell ist es kein Problem, dass der Automat einfach nicht mehr weiterarbeitet ("verklemmt"), wenn falscher Input gelesen wird (der Automat akzeptiert ein Wort, wenn dieses vollständig gelesen wurde und er sich in einem Endzustand befindet).

Bei Programmen ist es aber nicht wünschenswert, dass diese einfach nicht mehr weiterarbeiten. Wir werfen daher gegebenenfalls eine Exception. Wenn ein Wort ganz eingelesen wurde, wird überprüft, ob der aktuelle Zustand ein Endzustand ist. Falls ja, wird das Wort akzeptiert (der Lexer kann z.B. ein passendes Token erzeugen), andernfalls wird wieder eine Exception geworfen.

Wie im Falle einer Exception vernünftig weitergearbeitet werden soll, ist nicht so einfach. In der Regel werden Punkte in der Eingabe gesucht, wo man wieder aufsetzen kann (z.B. ; oder neue Zeile in C++). Wir beenden einfach das Programm.




```
while (cin >> symbol) {
  switch (status) {
   ...
  }
}
if (status == State::end)
  cout << "accept\n";
else
  throw runtime_error("ERROR");
```

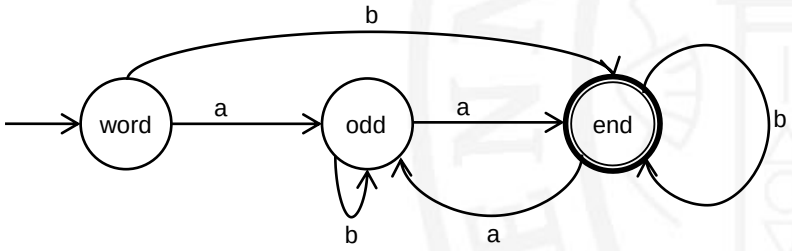
Akzeptierte Sprache

Die Menge aller Worte, die eine Grammatik erzeugt, bzw. eine entsprechender Automat akzeptiert wird als die von der Grammatik erzeugte bzw. vom Automaten akzeptierte Sprache bezeichnet.

Welche Sprache wird von unserem Automaten akzeptiert?

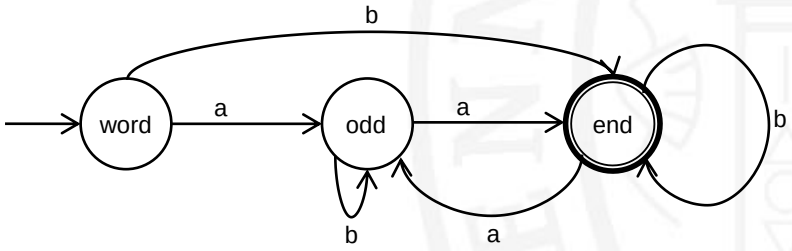
Versuchen Sie eine einfache Beschreibung aller Worte zu finden, die zur Sprache gehören.

Zum Testen Ihrer Hypothesen können Sie **/home/Xchange/ue4/fsm1_pr1** verwenden. (Sie können ein zu testendes Wort pro Zeile eingeben. Programmende durch EOF – Ctrl-D)



Akzeptierte Sprache (Lösung)

Der Automat akzeptiert alle Worte, die nur aus a und b bestehen, mindestens ein Zeichen lang ist und die eine gerade Anzahl von a's (auch 0) haben.

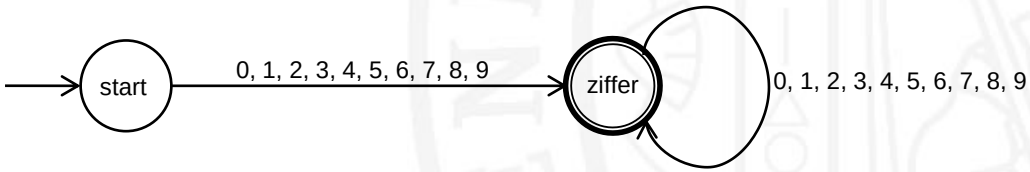


Ein Automat für natürliche Zahlen

Erstellen Sie ein Programm, das genau die natürlichen Zahlen akzeptiert.

Berechnen Sie den Wert der eingelesenen Zahl während der Abarbeitung und geben Sie diesen Wert aus, falls das eingegebene Wort akzeptiert wird, bzw. eine Fehlermeldung sonst.

Mögliche Probleme?



Unterlagen mit dem vorherigen Beispiel: /home/Xchange/ue4/bsp1.pdf
Zusatzaufgaben: /home/Xchange/ue4/zusatz1.pdf

universität wien

Mögliche Probleme

Was passiert, wenn der Zahlenbereich bei der Berechnung des Werts überschritten wird?

Sollte ebenfalls zu einer Fehlermeldung führen

Was erhalten Sie, wenn Sie zwei Zahlen getrennt durch einen Zeilenwechsel oder durch ein Leerzeichen eingeben?

Wahrscheinlich werden die Zeilenwechsel und Leerzeichen einfach ignoriert und Sie erhalten den entsprechenden Wert, der ohne diese Zeichen berechnet worden wäre.

Ursache: cin >> überliest automatisch alle Arten von Leerzeichen (whitespace). Ihr Programm erhält also diese Zeichen gar nicht.

Behebung: Einlesen der Zeichen in einen String (mit getline!) und dann Bearbeiten der Zeichen im String oder Verwendung von cin.get() (ein Zeichen einlesen) oder cin so umkonfigurieren, dass Leerzeichen nicht übersprungen werden (cin>>noskipws). Nicht vergessen, den Modus gegebenenfalls wieder auszuschalten, da andere Eingaben sonst nicht mehr wie gewohnt funktionieren (cin>>skipws).

universität wien

Zusatzbeispiele:

1) Erweitern Sie den Automaten für natürliche Zahlen, sodass ein Vorzeichen am Beginn (+ oder -) eingegeben werden kann und der Automat ganze Zahlen akzeptiert (und die korrekten Werte liefert).

2) Implementieren Sie einen Automaten, der reelle Zahlen (double) akzeptiert. Format wie in C++, also optionales Vorzeichen, dann Vorkommateil, Dezimalpunkt und Nachkommateil (Vor- oder Nachkommateil können leer sein, aber nicht beide. Beide entsprechen unserer natürlichen Zahl). Daran anschließend optional ein Exponent. Dieser besteht aus dem Buchstaben 'e' (groß oder klein) einem optionalen Vorzeichen und einer natürlichen Zahl). Z.B.: -3.6E+6

Wenn ein Exponent angegeben ist, dann kann auch das Komma entfallen (vor dem e muss dann der Vorkommateil der Zahl stehen) z.B. 6e-2.

3) Erweitern Sie den Automaten aus Beispiel 2 so, dass die entsprechende Zahl als Ergebnis ermittelt wird.

4) Beheben Sie die Probleme mit Leerzeichen / Zeilenwechseln in Ihren Automaten.

Parser

Während der Lexer meist mit regulären Grammatiken auskommt, wird für den Parser in der Regel eine kompliziertere Grammatik eingesetzt. Auch hier lässt sich aber die Grammatik relativ leicht in ein zugehöriges Programm übersetzen:

expression = term ('+' | '-') term

term = number

Wir wollen außerdem den Wert des Ausdrucks gleich auch berechnen. Das jeweils nächste Token kann mit ts.get() gelesen werden.

Token:

type (Ttype)
value (double)

```
double expression() {
  double result = term(); //linker Term

  Token tok = ts.get(); //Operator
  switch (tok.type) {
    //rechter Term
    case Ttype::plus: return result + term();
    case Ttype::minus: return result - term();
    default: throw runtime_error("+/- erwartet");
  }
}

double term() {
  Token tok = ts.get();
  if (tok.type != Ttype::number)
    throw runtime_error("Zahl erwartet");
  return tok.value;
}
```

Eine Grammatik für unsere Ausdrücke

Während der Lexer meist mit regulären Grammatiken auskommt, wird für den Parser in der Regel eine kompliziertere Grammatik eingesetzt. Auch hier lässt sich aber die Grammatik relativ leicht in ein zugehöriges Programm übersetzen:

expression = expression `('+' | '-')` term | term

term = term `('*' | '/' | '%')` primary | primary

primary = number | `'('` expression `')'`

Kopieren Sie die Datei mit dem Hauptprogramm: **cp /home/Xchange/ue4/exp1_pr1.cpp .**

Erstellen Sie die fehlende Funktion expression()

Die vordefinierten Konstanten für die Token-Typen finden Sie in **/home/Xchange/ue4/exp1_lib_pr1.h**

Kompilieren Sie mit: **g++ exp1_pr1.cpp /home/Xchange/ue4/exp1_lib_pr1.o –o exp1**

Unterlagen mit dem vorherigen Beispiel: /home/Xchange/ue4/bsp2.pdf
Zusatzaufgaben: /home/Xchange/ue4/zusatz2.pdf

universität wien

Abschließende Bemerkung

In der Praxis verwendet man Werkzeuge, die bei der Programmerstellung aus vorgegebenen Grammatiken unterstützen. Klassiker sind hier lex und yacc (flex und bison in der GNU Version).

Etwas moderner z.B. ANTLR oder Xtext.

Zusatzaufgabe

Schreiben Sie auch die Funktionen term() und primary() selbst. Eine Library, die nur die Funktionen des Token-Streams enthält, finden Sie unter /home/Xchange/ue4/exp1_lib1_pr1.o

Kompilieren Sie also mit: **g++ exp1_pr1.cpp /home/Xchange/ue4/exp1_lib1_pr1.o –o exp1**