

## Inhaltsverzeichnis

- 1 Einleitung
- 2 Backtracking
- 3 Einfachstes Program
- 4 Immer noch einfach
- 5 Schon etwas komplizierter
- 6 Taschenrechner
- 7 Klammern
- 8 Variablen
- 9 Hello World!
- 10 If und Schleifen
- 11 Funktionen
- 12 Parser Baum
- 13 Schlussbemerkung

## 1 Einleitung

Programmiersprachen sind eines der zentralsten Themen, mit denen sich Programmierer auseinandersetzen müssen. Trotzdem werden sie von vielen als magische Kiste, entwickelt von Spezialisten angesehen. In der Tat ist es ganz und gar nicht einfach einen guten Compiler zu entwickeln, vor allem sobald Optimierungen ins Spiel kommen. Die Grundlagen sind jedoch gar nicht so schwer. Dieser Artikel soll eine recht praxisnahe Einführung in das Thema Compilerbau darstellen und richtet sich an jeden der C++ kann und Grundkenntnisse in x86-Assemblerprogrammierung hat. Auf theoretische Konstrukte wie Grammatiken wird vereinzelt hingewiesen, jedoch braucht man sie nicht zu kennen, um folgen zu können.

Es muss irgendein GCC installiert sein, um die Beispiele ausführen zu können. Ich werde nämlich dessen Assembler und Linker einsetzen.

## 2 Backtracking

Bei einem Compiler handelt es sich um ein Programm, das eine Textdatei einliest und dann übersetzt. Es sollte offensichtlich sein, dass das Einlesen der einzelnen Zeichen ein wichtiger Baustein ist. Es muss möglich sein, in der Eingabe zurückzuspulen, da ein Übersetzer oft einfach ein paar Sachen durchprobieren muss, um herauszufinden, was ein gewisser Abschnitt darstellt. Zum Beispiel könnte ein Compiler an einer Stelle eine Logik ähnlich zu folgender anwenden: Wenn es keine Zahl ist dann wird es wohl ein String sein und wenn es dies auch nicht ist, dann handelt es sich um einen Fehler im Programmcode. Wenn es aber keine Zahl ist, so müssen aber alle Zeichen die nötig waren, um festzustellen, dass es keine Zahl ist, erneut interpretiert werden, um festzustellen ob es sich vielleicht um einen String handelt. Bei einer Zahl kann man es bereits beim ersten Buchstaben feststellen, jedoch gibt es auch komplexere Beispiele wo dies nicht so einfach ist. Hier benötigt man irgendeinen Weg, zurückspulen

zu können.

Es gibt viele Wege, dies nun zu realisieren. Zum Beispiel kann man Zeichenströme schreiben, welche immer nur das absolute Minimum von Zeichen im Speicher halten. Man muss ja nicht überall hin spulen können. Es reicht an ein paar bekannte Stellen zurück spulen zu können. Dies ermöglicht es selbst wahnsinnig große Quelldateien zu übersetzen und doch nur ein paar Byte Speicher zu benötigen.

In diesem Artikel wollen wir es allerdings nicht zu kompliziert machen und kopieren die Datei kurzerhand in den Speicher. Eine effiziente Implementierung eines Zeichenstroms wäre ein Thema, das in einen anderen Artikel gehört. Unser primitiver Strom wird aus zwei Zeigern bestehen. Einer zeigt auf das Zeichen, welches als nächstes verarbeitet werden soll und der andere auf das erste Zeichen nach dem Ende des Puffers. Es entspricht also einem Iteratorenpaar im Sinn der STL.

Die meisten Funktionen im Übersetzer werden der folgenden ähnlich sehen:

---

```
bool foo(..., const char*&pos, const char*end, ...){
    // Do stuff
}
```

---

Der erste Zeiger wird per Referenz übergeben, damit auch der Aufrufer weiß, bis wohin gelesen wurde. *end* dagegen ist immer gleich und muss deswegen nicht per Referenz übergeben werden. Der Rückgabewert gibt an, ob "es" geklappt hat. Was das nun im Detail heißt, hängt natürlich von der Funktion ab.

Da wir im Fehlerfall backtracken müssen und eine Ausnahme einen solchen Fall darstellt, werde ich Scopeguards einsetzen, um den Code ausnahmesicher und lesbar zu halten. Wir werden zwar keine Ausnahmen einsetzen, allerdings sollte man doch dafür sorgen, dass der Code ausnahmfreundlich ist.

---

```
class BacktrackGuard{
public:
    explicit BacktrackGuard(const char*&pos):
        pos(pos), start(pos){}

    ~BacktrackGuard(){
        if(start)
            pos = start;
    }

    void no_backtrack(){
        start = 0;
    }

private:
    const char*&pos;
    const char*start;

    BacktrackGuard(const BacktrackGuard&);
    BacktrackGuard&operator=(const BacktrackGuard&);
};
```

---

In vielen Compilern sind diese Zeichenströme auch dafür verantwortlich, die Position in der Datei zu merken. Das heißt konkret, welche Zeile und das wie viele Zeichen in dieser. Dies ist zum Beispiel nötig, um sinnvolle Fehlermeldungen ausgeben zu können. Wir werden es uns hier aber einfach machen und dieses Problem einfach ignorieren: Wir geben keine Positionsinformationen mit den Fehlern aus.

## 3 Einfachstes Program

Um anzufangen wollen wir einen Compiler schreiben, welche nur eine sehr einfache Sprache übersetzen kann, eine die so einfach ist, dass es schon an Spott grenzt, sie als Sprache zu bezeichnen. Ein Programm besteht aus einer einzigen Ziffer und diese muss eine '1' sein. Ein korrektes (und sogar das einzig korrekte) Programm wäre also:

---

1

---

Man sollte beachten, dass der Einfachheit zuliebe keine Leerzeichen erlaubt sind.

Das "1" soll bewirken, dass "1" auf der Konsole ausgegeben wird. Der Compiler hierfür sieht wie folgt aus:

---

```
#include <iostream>
#include <fstream>
#include <vector>
#include <iterator>
using namespace std;

bool parse_one(const char*&pos, const char*end, std::ostream&out){
    if(pos != end){
        if(*pos == '1'){
            ++pos;
            out
                <<"pushl $1"<<endl
                <<"call _print_int"<<endl
                <<"addl $4, %esp"<<endl;
            return true;
        }else
            cerr<<"Unknown command"<<endl;
    }else
        cerr<<"Unexpected end of file"<<endl;
    return false;
}

int main(){
    ifstream in("in.txt");
    vector<char>source_code;
    copy(
        istreambuf_iterator<char>(in.rdbuf()),
        istreambuf_iterator<char>(),
        back_inserter(source_code));

    ofstream out("out.s");
```

```

out
    <<" .text"<<endl
    <<" .globl _prog"<<endl
    <<"_prog:"<<endl;

const char*pos = &source_code.front();
const char*end = &source_code.back() + 1;

if(parse_one(pos, end, out)){
    if(pos != end)
        cerr<<"Unexpected trailing characters at end of program"<<endl;
    else
        out<<"ret"<<endl;
}
}

```

---

Auch wenn dieses Programm nicht allzu kompliziert sein dürfte, reicht es aber um die Grundstruktur zu verdeutlichen. Source code wird eingelesen, dann verarbeitet und der Assemblercode wird in eine Datei namens "out.s" geschrieben. Wo etwas fehlschlagen kann und warum, sollte durch die Fehlermeldungen deutlich werden.

Um eine ausführbare Datei zu erzeugen muss "out.s" gegen eine C-Sourcecode-Datei gelinkt werden, welche folgendermaßen aussieht:

---

```

#include <stdio.h>

void prog();

int main(){
    prog();
}

void print_int(int n){
    printf("%d", n);
}

```

---

Dieser Schnipsel C-Code stellt sozusagen die Laufzeitbibliothek unserer Sprache dar. Darum nenne ich die Datei "rt.c" (steht für **runtime**). Übersetzen kann man das ganze recht einfach.

---

```
gcc out.s rt.c -o out.exe
```

---

Manche GCC-Inkarnationen hängen nicht automatisch einen Unterstrich vor C-Funktionsnamen. Wenn die angegebene Kommandozeile einen Linkerfehler produziert, dann wird das wahrscheinlich der Grund sein. Ändern Sie in diesem Fall die Datei "rt.c" zu:

---

```

#include <stdio.h>

void _prog();

int main(){
    _prog();
}

void _print_int(int n){
    printf("%d", n);
}

```

---

Nun sollte es keine Probleme mehr geben.

## 4 Immer noch einfach

Man braucht kein Hellseher zu sein, um zu bemerken, dass unsere Sprache noch nicht zu viel zu gebrauchen ist. Dies wird vorerst aber leider auch so bleiben. Zuerst arbeiten wir nämlich an der Syntax unserer Sprache.

Es wäre doch gut, wenn man mehr Freiheiten beim Setzen der Leerzeichen hätten. Hierfür schreiben wir uns eine Funktion, welche Leerzeichen überspringt.

---

```

#include <cctype>
//...
void skip_spaces(const char*&pos, const char*end){
    while(pos != end && isspace(*pos))
        ++pos;
}

```

---

In einem komplexeren Compiler würde diese Funktion auch Kommentare überspringen. Der Rest des Compilers ändert sich nur leicht:

---

```

bool parse_one(const char*&pos, const char*end, std::ostream&out){
    skip_spaces(pos, end);          // Ich bin neu (1)
    if(pos != end){
        if(*pos == '1'){
            ++pos;
            out
                <<"pushl $1"<<endl
                <<"call _print_int"<<endl
                <<"addl $4, %esp"<<endl;
            return true;
        }else
            cerr<<"Unknown command"<<endl;
    }else
        cerr<<"Unexpected at end of file"<<endl;
    return false;
}

```

```

int main(){
    ifstream in("in.txt");
    vector<char>source_code;
    copy(istreambuf_iterator<char>(in.rdbuf()),
        istreambuf_iterator<char>(),
        back_inserter(source_code));

    ofstream out("out.s");
    out
        <<".text"<<endl
        <<".globl _prog"<<endl
        <<"_prog:"<<endl;

    const char*pos = &source_code.front();
    const char*end = &source_code.back() + 1;

    if(parse_one(pos, end, out)){
        skip_spaces(pos, end);          // Ich auch (2)
        if(pos != end)
            cerr<<"Unexpected trailing characters at end of file"<<endl;
        else
            out<<"ret"<<endl;
    }
}

```

---

Wozu die erste neue Zeile dient, sollte selbsterklärend sein. Die zweite sorgt dafür, dass Leerzeichen sich auch am Ende des Programms befinden können.

Als nächstes werden wir es möglich machen, die 1 in Buchstaben auszuschreiben. Dazu schreiben wir eine Funktion welche überprüft, ob wir uns vor einem Wort befinden oder nicht. Falls ja wird dieses auch gleich übersprungen.

---

```

bool ignore(const char*&pos, const char*end, const char*what){
    BacktrackGuard guard(pos);
    skip_spaces(pos, end);

    while(pos != end && *pos == *what){
        ++what;
        ++pos;
        if(*what == '\0'){
            guard.no_backtrack();
            return true;
        }
    }
    return false;
}

```

---

Nun müssen wir nur noch parse\_one anpassen.

---

```

void write_print_one_code(std::ostream&out){
    out
        <<"pushl $1"<<endl
        <<"call _print_int"<<endl
        <<"addl $4, %esp"<<endl;
}

bool parse_one(const char*&pos, const char*end, std::ostream&out){
    if(ignore(pos, end, "1") || ignore(pos, end, "one")){
        write_print_one_code(out);
        return true;
    } else {
        cerr<<"Unknown command"<<endl;
        return false;
    }
}

```

---

Wer genau hinschaut, bemerkt, dass sich in *parse\_one* kein *skip\_spaces* mehr befindet. Dadurch, dass wir in *ignore* einen Aufruf platziert haben, werden beinahe alle anderen überflüssig und damit ist der Code schon viel weniger fehleranfällig und lesbarer.

Und schon können wir Programme wie folgendes übersetzen.

---

```
one
```

---

Jedoch hat sich bereits der erste Bug eingeschlichen. Betrachten wir mal:

---

```
onehundred
```

---

Das Programm wird zwar wie erwünscht abgelehnt, aber nicht aus dem richtigen Grund. Der Compiler liest nur bis zum Ende von "one", übersetzt dies dann auch und das Programm wird nur abgelehnt, weil der Compiler nichts mit "hundred" hinter dem Ende des erkannten Programms anzufangen weiß. Das heißt, der Compiler reißt Wörter ohne Gewissensbisse entzwei. Es hängt natürlich von der Sprache ab, ob dies ein Fehler ist oder nicht, allerdings nimmt ein C-Compiler kein "staticintfoo" an, sondern verlangt, dass diese schön mit Leerzeichen getrennt werden. Dies hat auch seinen guten Grund, denn dadurch werden Zweideutigkeiten vermieden. Allerdings übersetzt ein C-Compiler auch "1+2" obwohl zwischen der "1" und dem "+" kein Leerzeichen ist. Man könnte nun für Zahlen und Operatoren eigene Funktionen schreiben. Die eine würde zusätzlich noch überprüfen, ob sich ein Leerzeichen hinter dem Wort befindet.

Es geht allerdings auch einfach.

---

```

bool ignore(const char*&pos, const char*end, const char*what){
    BacktrackGuard guard(pos);
    skip_spaces(pos, end);

    while(pos != end && *pos == *what){
        ++what;
    }
}

```

```

    ++pos;
    if(*what == '\0'){
        --what;
        if(isalpha(*what))
            if(isalnum(*pos))
                return false;
        guard.no_backtrack();
        return true;
    }
}
return false;
}

```

---

Streng genommen ist dies zwar ein Hack, aber er macht doch vieles einfacher und angenehmer.

## 5 Schon etwas komplizierter

So, nun wollen wir auch mal eine Sprache verwenden, welche ansatzweise eine Verwendung hat. Ziel ist es, einen Compiler zu schreiben, der Programme wie folgendes annimmt:

```

print 1
print 2 print 3
print 4 print
5

```

---

Das Programm besteht aus einer Reihe von "print" Befehlen, welche jeweils eine positive Ganzzahl ausgeben.

Als erstes ersetzen wir `parse_one` (und auch den Aufruf in `main`) durch folgende Funktion:

```

bool parse_program(const char*&pos, const char*end, ostream&out){
    while(parse_command(pos, end, out))
        {}
    return true;
}

```

---

Wie bereits erwähnt, besteht ein Programm aus einer Reihe von Befehlen und darum sollte die Funktion, die ein Programm liest, auch nur dies tun. `parse_program` gehen die Details eines Befehls nichts an. Dadurch halten wir die Funktionen klein und den Compiler übersichtlich und verständlich.

Um das Lesen eines Befehls kümmert sich die Funktion `parse_command`, welche dann wieder auf eine Reihe von Unterfunktionen zurückgreift.

```

bool parse_command(const char*&pos, const char*end, ostream&out){
    BacktrackGuard guard(pos);
    if(ignore(pos, end, "print")){
        int n;

```

```

        if(read_number(pos, end, n)){
            out
                <<"pushl $"<<n<<endl
                <<"call _print_int"<<endl
                <<"addl $4, %esp"<<endl;
            guard.no_backtrack();
            return true;
        }else
            cerr<<"Argument to print is missing"<<endl;
    }
    return false;
}

bool read_number(const char*&pos, const char*end, int&n){
    skip_spaces(pos, end);
    if(pos != end && isdigit(*pos)){
        n = 0;
        do{
            n *= 10;
            n += *pos - '0';
            ++pos;
        }while(pos != end && isdigit(*pos));
        return true;
    }else
        return false;
}

```

---

*read\_number* könnte man nun natürlich auch das Lesen von Hexadezimal- oder Oktalzahlen beibringen. Wir wollen uns aber mit einfachen Dezimalzahlen begnügen, um nicht den Rahmen zu sprengen.

In welche und wie viele Unterfunktionen man aufbrechen soll, hängt von der eigenen Erfahrung ab. Hier kann ich höchstens als Faustregel angeben, dass man alles auslagern soll, für was man einen passenden Funktionsnamen findet.

## 6 Taschenrechner

Als nächstes nehmen wir uns vor, komplexere Ausdrücke zu unterstützen. Also zum Beispiel folgendes Programm.

---

```

print 1+1
print 5*8
print 4 + 8*7

```

---

Jeder, der mit dem Überladen von Operatoren in C++ vertraut ist, der weiß, dass man Operatoren auch als Funktionsaufrufe ansehen kann. Den Code oben könnte man auch wie folgend umschreiben.

---

```
print add(1, 1)
print mul(5, 8)
print add(4, mul(8, 7))
```

---

Auf diese Weise wollen wir unsere Operatoren auch umsetzen, als inline-Minifunktionen. Wer schon einmal x86-Assembler geschrieben hat, der weiß, dass man das Register `eax` für Rückgabewerte nutzt. Dies wollen wir auch hier tun. Jede Minifunktion schreibt den Rückgabewert in `eax`. Eine Funktion mit mehreren Argumenten wertet die Argumente nacheinander aus und nutzt den Stack um Werte zwischenzuspeichern.

Allerdings, bevor wir uns um die Generierung des Assemblercodes kümmern, müssen wir zuerst dafür sorgen, dass wir den Ausdruck richtig lesen können. Hier hilft es, sich eine Reihe von Fragen zu stellen:

Woraus besteht ein Program? Aus Befehlen

Woraus besteht ein Befehl? Aus "print" gefolgt von einem Ausdruck

Woraus besteht ein Ausdruck? Aus einer Summe oder Differenz von Termen

Woraus besteht ein Term? Aus einem Produkt, Quotient oder Modulo von Faktoren

Woraus besteht ein Faktor? Aus einer Zahl mit vielleicht einem Vorzeichen

Wer sich für den theoretischen Hintergrund dieser Fragen interessiert, der kann mal nach EBNF<sup>[1]</sup> suchen. In diesem Artikel werde ich aber nicht weiter darauf eingehen.

Nun wissen wir schon, welche Funktionen gebraucht werden. Einige der Funktionen sind bereits implementiert und darum werde ich sie nicht wiederholen.

---

```
bool parse_command(const char*&pos, const char*end, ostream&out){
    BacktrackGuard guard(pos);
    if(ignore(pos, end, "print")){
        if(parse_expression(pos, end, out)){
            out
                <<"pushl %eax"<<endl
                <<"call _print_int"<<endl
                <<"addl $4, %esp"<<endl;
            guard.no_backtrack();
            return true;
        }else
            cerr<<"Argument to print is missing"<<endl;
    }
    return false;
}
```

---

Diese Funktion sollte selbsterklärend sein.

---

```
bool parse_expression(const char*&pos, const char*end, ostream&out){
    BacktrackGuard guard(pos);
    if(parse_term(pos, end, out)){
        for(;;){
            if(ignore(pos, end, "+")){
                out<<"push %eax"<<endl;
            }
        }
    }
}
```

```

        if(!parse_term(pos, end, out)){
            cerr<<"Expected term after +"<<endl;
            return false;
        }
        out
            <<"addl (%esp), %eax"<<endl
            <<"addl $4, %esp"<<endl;
    }else if(ignore(pos, end, "-")){
        out<<"push %eax"<<endl;
        if(!parse_term(pos, end, out)){
            cerr<<"Expected term after -"<<endl;
            return false;
        }
        out
            <<"subl %eax, (%esp)"<<endl
            <<"pop %eax"<<endl;
    }else
        break;
    }
    guard.no_backtrack();
    return true;
}else
    return false;
}

```

---

Diese Funktion sieht auf den ersten Blick bedrohlich aus, allerdings ist sie an sich recht einfach. Die einzelnen Terme der Summe werden nacheinander ausgewertet und zusammengezählt beziehungsweise voneinander abgezogen.

Die Multiplikation wird ähnlich behandelt.

---

```

bool parse_term(const char*&pos, const char*end, ostream&out){
    BacktrackGuard guard(pos);
    if(parse_factor(pos, end, out)){
        for(;;){
            if(ignore(pos, end, "*")){
                out<<"push %eax"<<endl;
                if(!parse_factor(pos, end, out)){
                    cerr<<"Expected factor after *"<<endl;
                    return false;
                }
            }
            out
                <<"imull (%esp), %eax"<<endl
                <<"addl $4, %esp"<<endl;
        }else if(ignore(pos, end, "/")){
            out<<"push %eax"<<endl;
            if(!parse_factor(pos, end, out)){
                cerr<<"Expected factor after /"<<endl;
                return false;
            }
        }

        out
            <<"movl %eax, %ebx"<<endl
            <<"pop %eax"<<endl
    }
}

```

```

        <<"movl $0, %edx"<<endl
        <<"idiv %ebx"<<endl;
    }else if(ignore(pos, end, "%")){
        out<<"push %eax"<<endl;
        if(!parse_factor(pos, end, out)){
            cerr<<"Expected factor after /"<<endl;
            return false;
        }

        out
            <<"movl %eax, %ebx"<<endl
            <<"pop %eax"<<endl
            <<"movl $0, %edx"<<endl
            <<"idiv %ebx"<<endl
            <<"movl %edx, %eax"<<endl;
    }else
        break;
    }
    guard.no_backtrack();
    return true;
}
}

```

---

Die Funktion, um Faktoren auszuwerten, ist recht einfach, da ein Faktor nur aus einer Zahl bestehen kann.

---

```

bool parse_factor(const char*&pos, const char*end, ostream&out){
    BacktrackGuard guard(pos);
    int n;
    bool negate = false;
    if(ignore(pos, end, "-"))
        negate = true;

    if(read_number(pos, end, n))
        out<<"movl $"<<n<<"", %eax"<<endl;
    else
        return false;

    if(negate)
        out<<"negl %eax"<<endl;
    guard.no_backtrack();
    return true;
}

```

---

## 7 Klammern

Als nächstes wollen wir geklammerte Ausdrücke unterstützen. Fragen wir uns erneut, woraus ein Faktor bestehen kann. Wirklich nur aus einer Zahl? Nein, unter Umständen auch aus einem geklammerten Ausdruck. Ein Ausdruck besteht - über Umwege - aber wieder aus einem Faktor. Es mag auf den ersten Blick nicht offensichtlich sein, allerdings stellen solche rekursiven Definitionen an sich kein Problem dar. Man muss die entsprechende Funktion lediglich rekursiv aufrufen.

---

```

bool parse_factor(const char*&pos, const char*end, ostream&out){
    BacktrackGuard guard(pos);
    bool negate = false;
    if(ignore(pos, end, "-"))
        negate = true;

    int n;
    if(read_number(pos, end, n))
        out<<"movl $"<<n<<" , %eax"<<endl;
    else if(ignore(pos, end, "(")){
        parse_expression(pos, end, out);
        if(!ignore(pos, end, ")"))
            cerr<<"Missing )" <<endl;
    }else
        return false;

    if(negate)
        out<<"negl %eax"<<endl;
    guard.no_backtrack();
    return true;
}

```

---

## 8 Variablen

Das nächste, was jeder Compiler beherrschen sollte, sind Variablen. Wir wollen uns hier auf lokale Variablen beschränken. Das heißt, alle werden auf den Stack gepackt und keine kommen in statischen Speicher. Um die Sache noch weiter zu vereinfachen, sind alle Variablen vom Typ int. Dies ist eine ziemlich starke Einschränkung, aber irgendwo muss ich ja streichen damit dieser Artikel nicht zu groß wird. 😊

Das erste was benötigt wird, ist eine Datenstruktur welche die Variablen verwaltet. Hierfür definiere ich eine Klasse.

---

```

class VarManager{
public:
    bool create_var(string name, ostream&out);
    bool is_var_defined(string name)const;
    string get_var_address(string name)const;
    void push_scope();
    void pop_scope(ostream&out);
};

```

---

Auf die Implementierungsdetails werde ich nicht eingehen. Eine fertige Implementierung findet man hier<sup>[2]</sup>.

Es soll möglich sein, in höheren Scopes Variablen zu definieren, welche Variablen aus den unteren Scopes verdecken. Dieses Verhalten ist natürlich sprachabhängig allerdings wollen wir uns hier einfach an die C-Regeln halten. Mit ihnen dürfte jeder Leser vertraut sein.

- `create_var` erzeugt eine Variable im obersten Scope mit einem gewissen Namen. Falls es bereits eine solche Variable gibt, so wird `false` zurückgegeben, andernfalls wird `true` zurückgegeben.
- Mit `is_var_defined` kann man ermitteln, ob es eine Variable bereits gibt oder nicht. Hier sollte darauf hingewiesen werden, dass diese Funktion nichts über die Erfolgsaussichten von `create_var` aussagt, da Variablen verdeckt werden können.
- `get_var_address` gibt die Stackadresse der Variable aus. Adressen werden von `ebp` aus gezählt. Wir müssen also auch die `main` Funktion anpassen damit `ebp` einen brauchbaren Wert enthält.
- `push_scope` sollte aufgerufen werden, wenn sich der Compiler in einen neuen Scope begibt.
- Beim Verlassen eines Scopes sollte `pop_scope` aufgerufen werden. Alle Variablen dieses Scopes werden zerstört.

Der `out`-Parameter ist der Zeichenstrom der den Assemblercode ausgibt. Methoden mit ihm können Code generieren.

Nun müssen wir noch dafür sorgen, dass `VarManager` überall zugänglich ist. Eine Möglichkeit wäre, eine Instanz global anzulegen oder ein Singleton zu basteln. Dies ist aber nicht wirklich die feine Art und führt zu Problemen, sobald man den Code parallelisieren möchte. Eine andere Möglichkeit ist eine Instanz in `main` anzulegen und dann eine Referenz auf diese von Funktion zu Funktion weiter zu geben. Diese Lösung ist viel flexibler, allerdings sind Variablen nicht das einzige, was verwaltet werden will.

Wer sich ein bisschen mit Schleifen auskennt, der weiß, dass dort Sprungadressen verwaltet werden müssen. Wenn man Variablen in Register packen will, so müssen diese auch verwaltet werden. Wenn wir jetzt auf alles eine Referenz übergeben, dann verlieren wir uns bald in endlosen Parameterlisten. Ein noch viel schlimmeres Problem ist, dass man keine neue Struktur einführen kann ohne den ganzen Code umzubauen.

Die Lösung ist recht einfach. Man baut eine Hilfsstruktur, welche diese Referenzen verwaltet. Dies sieht dann etwa folgendermaßen aus:

---

```
class Env{
public:
    explicit Env(VarManager&var):
        var(var){}

    VarManager&var;
};
```

---

Eine Referenz auf diese Klasse wird dann von Funktion zu Funktion übergeben. Wenn eine neue Verwaltungsstruktur hinzugefügt wird, so braucht man nur `Env` und `main` zu verändern. Der gewählte Name steht für **En**vironment. Als Environment verstehe ich die Zusammenfassung aller Verwaltungsstrukturen.

Man kann die Ausgabe auch als Teil von `Env` ansehen.

---

```

class Env{
public:
    explicit Env(VarManager&var, ostream&out):
        var(var), out(out){}

    VarManager&var;
    ostream&out;
};

```

---

Die main-Funktion wird also wie folgend abgeändert:

---

```

int main(){
    ifstream in("in.txt");
    vector<char>source_code;
    copy(istreambuf_iterator<char>(in.rdbuf()),
        istreambuf_iterator<char>(),
        back_inserter(source_code));

    ofstream out("out.s");
    out
        <<".text"<<endl
        <<".globl _prog"<<endl
        <<"_prog:"<<endl
        <<"push %ebp"<<endl
        <<"mov %esp, %ebp"<<endl;

    const char*pos = &source_code.front();
    const char*end = &source_code.back() + 1;

    VarManager var;
    Env env(var, out);

    if(parse_program(env, pos, end)){
        skip_spaces(pos, end);
        if(pos != end)
            cerr<<"Unexpected trailing characters at end of file"<<endl;
        else
            out
                <<"pop %ebp"<<endl
                <<"ret"<<endl;
    }
}

```

---

*parse\_command* muss nun auch geändert werden, da unsere Sprache ja nun zusammengesetzte Befehle mittels Scopes unterstützt. Wir werden hier wie in C Akkoladen verwenden um Scopes zu begrenzen. Um die Funktion klein zu halten, lagere ich Teile der Funktion in eine neue Funktion namens *parse\_block* aus.

---

```

bool parse_block(Env&env, const char*&pos, const char*end){
    BacktrackGuard guard(pos);
    if(ignore(pos, end, "{")){
        env.var.push_scope();
    }
}

```

```

while(parse_block(env, pos, end))
    {}

if(ignore(pos, end, "}")){
    env.var.pop_scope(env.out);
    guard.no_backtrack();
    return true;
}else{
    cerr<<"Missing }"<<endl;
    return false;
}
}else{
    guard.no_backtrack();
    return parse_command(env, pos, end);
}
}

bool parse_command(Env&env, const char*&pos, const char*end){
    BacktrackGuard guard(pos);
    if(ignore(pos, end, "print")){
        if(parse_expression(env, pos, end)){
            env.out
                <<"pushl %eax"<<endl
                <<"call _print_int"<<endl
                <<"addl $4, %esp"<<endl;
            guard.no_backtrack();
            return true;
        }else
            cerr<<"Argument to print is missing"<<endl;
        }
    return false;
}
}

```

---

*parse\_program* können wir dann auch abändern, da mehrere Befehle durch die Benutzung von Akkoladen möglich sind.

---

```

bool parse_program(Env&env, const char*&pos, const char*end){
    return parse_block(env, pos, end);
}

```

---

Nun ist es möglich, Programme wie folgendes zu übersetzen.

---

```

{
    print 1
    print 2
    {
        print 3
        {}
    }
    print 4
}

```

---

Die äußeren Akkoladen sind nötig.

Auch wenn das schon nicht schlecht aussieht, haben wir doch einen wesentlichen Bestandteil vergessen. Es gibt noch keine Variablen!

Zuerst brauchen wir eine Funktion, die Namen von Variablen liest.

---

```
bool read_identifier(const char*&pos, const char*end, string&identifier){
    skip_spaces(pos, end);
    if(pos != end && (isalpha(*pos) || *pos == '_')){
        identifier = "";
        do{
            identifier += *pos;
            ++pos;
        }while(pos != end && (isalnum(*pos) || *pos == '_'));
        return true;
    }else
        return false;
}
```

---

Das nächste, was wir brauchen, ist ein Befehl, um Variablen zu erschaffen und einen weiteren, um ihnen etwas zuzuweisen.

---

```
bool parse_command(Env&env, const char*&pos, const char*end){
    BacktrackGuard guard(pos);
    if(ignore(pos, end, "print")){
        if(parse_expression(env, pos, end)){
            env.out
                <<"pushl %eax"<<endl
                <<"call _print_int"<<endl
                <<"addl $4, %esp"<<endl;
            guard.no_backtrack();
            return true;
        }else{
            cerr<<"Argument to print is missing"<<endl;
            return false;
        }
    }else if(ignore(pos, end, "var")){
        string identifier;
        if(read_identifier(pos, end, identifier)){
            if(env.var.create_var(identifier, env.out)){
                guard.no_backtrack();
                return true;
            }else
                cerr<<"Variable redefined"<<endl;
        }else
            cerr<<"var must be followed by a variable name"<<endl;
        return false;
    }else{
        string identifier;
        if(read_identifier(pos, end, identifier)){
            if(ignore(pos, end, "=")){
```

```

        if(parse_expression(env, pos, end)){
            env.out<<"movl %eax, "
                <<env.var.get_var_address(identifizier)<<endl;
            guard.no_backtrack();
            return true;
        }else
            cerr<<"Missing right operant of ="<<endl;
    }else
        cerr<<"Missing = in assignment"<<endl;
    }
    return false;
}
}

```

---

Jetzt brauchen wir nur noch eine Möglichkeit, lesend auf eine Variable zuzugreifen. Dazu verändere ich *parse\_factor*. Hier lagere ich auch wieder einen Teil der Funktion aus.

---

```

bool parse_literal(Env&env, const char*&pos, const char*end){
    int n;
    if(read_number(pos, end, n)){
        env.out<<"movl $"<<n<<" , %eax"<<endl;
        return true;
    }
    string identifizier;
    if(read_identifizier(pos, end, identifizier)){
        if(env.var.is_var_defined(identifizier)){
            env.out<<"movl "<<env.var.get_var_address(identifizier)
                <<" , %eax"<<endl;
            return true;
        }else{
            cerr<<"Undefined variable "<<identifizier<<endl;
            return false;
        }
    }
    return false;
}

bool parse_factor(Env&env, const char*&pos, const char*end){
    BacktrackGuard guard(pos);
    bool negate = false;
    if(ignore(pos, end, "-"))
        negate = true;

    if(parse_literal(env, pos, end)){
    }else if(ignore(pos, end, "(")){
        parse_expression(env, pos, end);
        if(!ignore(pos, end, ")"))
            cerr<<"Missing )" <<endl;
    }else
        return false;
}

```

```
    if(negate)
        env.out<<"negl %eax"<<endl;

    guard.no_backtrack();
    return true;
}
```

---

Langsam aber sicher entwickelt sich unsere Sprache weiter. Ein Programm sieht jetzt schon aus, als könnte es für irgendetwas zu gebrauchen sein.

---

```
{
    var a
    var b
    a = 5
    b = a + 5
    {
        a = a-1
        var c
        c = b
        print c
    }
    print a+b
}
```

---

## 9 Hello World!

Im vorherigen Kapitel habe ich gesagt, alle Variablen wären vom Typ int. Daran werde ich jetzt auch nichts ändern, allerdings benötigt jede Sprache wenigstens eine minimale Unterstützung für Strings. Ohne kann man noch nicht einmal ordentliche Beispielprogramme schreiben. Noch nicht einmal ein "Hello world!"-Programm ist möglich.

Ziel dieses Kapitels ist es, den print-Befehl ein wenig aufzumotzen. Er soll auch Stringlitterale als Parameter übernehmen können und mehrere durch Kommas getrennte Parameter sollen auch unterstützt werden. Er wird aber einzigartig bleiben in dem Sinn, dass nur er Strings als Argument nehmen kann.

Als erstes benötigen wir eine Funktion, welche Stringlitterale parsen kann.

---

```
char unescape_character(char c){
    switch(c){
        case 'n':return '\n';
        case 't':return '\t';
        case '\\':return '\\';
        case '\\@0@Unknown escape sequence \\@1@') {
            ++pos;
            str = "";
            do{
                if(*pos == '\\'){
                    ++pos;
                    if(pos == end){
```

```

        cerr<<"EOF in string literal"<<endl;
        return false;
    }
    str += unescape_character(*pos);
} else
    str += *pos;
    ++pos;
} while (pos != end && *pos != '\\');
++pos;
guard.no_backtrack();
return true;
} else
    return false;
}

```

---

Als nächstes brauchen wir eine Möglichkeit, Zeichenketten in den Code einzufügen. Diese werden wir in einer Data-Sektion unterbringen. Wir brauchen also eine weitere Klasse, welche die Stringlitterale verwaltet.

---

```

class DataSection{
public:
    void add_string_data(string label, string data);
    void write_data_section(ostream&);
};

```

---

Auch hier werde ich auf eine fertige Implementierung verweisen<sup>[3]</sup>. Es gibt den offensichtlichen Weg, diese Klasse zu implementieren oder man kann versuchen, ein paar Bytes zu sparen, indem man versucht, die Strings so anzuordnen, dass sie sich überlappen. Dies geht natürlich nur, solange die originalen Zeichenketten in der Data-Sektion nicht verändert werden dürfen.

- Mit *add\_string\_data* kann man einen String ablegen. Es wird ein Nullbyte angefügt und Label wird so gesetzt, dass es auf das erste Byte zeigt.
- *write\_data\_section* macht das, was der Name sagt.

Da *DataSection* ein Label braucht, brauchen wir auch jemanden, der diese verwaltet.

---

```

class LabelManager{
public:
    string make_unique();
};

```

---

- *make\_unique* erzeugt einen eindeutigen Labenamen und gibt diesen zurück.

Die Implementierung gibt es hier<sup>[4]</sup>.

*main* und *Env* müssen auch angepasst werden:

---

```

class Env{
public:
    explicit Env(VarManager&var, ostream&out,
                DataSection&data, LabelManager&label):
        var(var), out(out), data(data), label(label){}

    VarManager&var;
    ostream&out;
    DataSection&data;
    LabelManager&label;
};

int main(){
    ifstream in("in.txt");
    vector<char>source_code;
    copy(istreambuf_iterator<char>(in.rdbuf()),
        istreambuf_iterator<char>(),
        back_inserter(source_code));

    ofstream out("out.s");
    out
        <<" .text"<<endl
        <<" .globl _prog"<<endl
        <<" _prog:"<<endl
        <<"push %ebp"<<endl
        <<"mov %esp, %ebp"<<endl;

    const char*pos = &source_code.front();
    const char*end = &source_code.back() + 1;

    VarManager var;
    DataSection data;
    LabelManager label;
    Env env(var, out, data, label);

    if(parse_program(env, pos, end)){
        skip_spaces(pos, end);
        if(pos != end)
            cerr<<"Unexpected trailing characters at end of file"<<endl;
        else{
            out
                <<"pop %ebp"<<endl
                <<"ret"<<endl
                <<" .data"<<endl;
            data.write_data_section(out);
        }
    }
}

```

---

Schlussendlich muss auch noch "rt.c" angepasst werden, damit Zeichenketten ausgegeben werden können.

---

```

#include <string.h>

void print_string(const char*str){
    fwrite(str, 1, strlen(str), stdout);
}

```

---

*printf* will ich nicht benutzen, da man bei dieser Funktion auf Prozentzeichen achten muss. Durch die Benutzung von *fwrite* gehe ich allen möglichen Problemen aus dem Weg, selbst denen die durch Printf-Format-Erweiterungen entstehen könnten.

Da der print-Befehl komplizierter geworden ist, lagere ich ihn aus.

---

```

bool parse_print(Env&env, const char*&pos, const char*end){
    BacktrackGuard guard(pos);
    if(ignore(pos, end, "print")){
        do{
            string str;
            if(read_string(pos, end, str)){
                string label = env.label.make_unique();
                env.data.add_string_data(label, str);
                env.out
                    <<"pushl $"<<label<<endl
                    <<"call _print_string"<<endl
                    <<"addl $4, %esp"<<endl;
            }else if(parse_expression(env, pos, end)){
                env.out
                    <<"pushl %eax"<<endl
                    <<"call _print_int"<<endl
                    <<"addl $4, %esp"<<endl;
            }else{
                cerr<<"Argument to print is missing"<<endl;
                return false;
            }
        }while(ignore(pos, end, ","));
        guard.no_backtrack();
        return true;
    }else
        return false;
}

```

```

bool parse_command(Env&env, const char*&pos, const char*end){
    BacktrackGuard guard(pos);
    if(parse_print(env, pos, end)){
        guard.no_backtrack();
        return true;
    }else if(ignore(pos, end, "var")){
        string identifier;
        if(read_identifier(pos, end, identifier)){
            if(env.var.create_var(identifier, env.out)){
                guard.no_backtrack();
                return true;
            }else
                cerr<<"Variable redefined"<<endl;
        }
    }
}

```

```

        }else
            cerr<<"var must be followed by a variable name"<<endl;
        return false;
    }else{
        string identifier;
        if(read_identifier(pos, end, identifier)){
            if(ignore(pos, end, "=")){
                if(parse_expression(env, pos, end)){
                    env.out<<"movl %eax, "
                        <<env.var.get_var_address(identifier)<<endl;
                    guard.no_backtrack();
                    return true;
                }else
                    cerr<<"Missing right operant of ="<<endl;
            }else
                cerr<<"Missing = in assignment"<<endl;
        }
        return false;
    }
}

```

---

Nun endlich können wir ein "Hello world"-Programm schreiben.

---

```

{
    print "Hello" , " world!\n"
    var a
    var b
    a = 3
    b = 5
    print a, " + ", b, " = ", a+b
}

```

---

## 10 If und Schleifen

Was jetzt noch fehlt, sind Abfragen und Schleifen. Diese sind auch nicht sonderlich schwer. Zuerst benötigen wir eine Funktion, um ein if einzulesen. Der Einfachheit halber werden wir nicht versuchen, die Werte im Statusregister zu halten und wenn möglich wiederzuverwenden. If nutzt einfach wie überall eax und nimmt natürlich auch ein int als Bedingung. Dies ist recht ineffizient, allerdings macht es die Sache doch wesentlich einfacher.

---

```

bool parse_if(Env&env, const char*&pos, const char*end){
    BacktrackGuard guard(pos);
    if(ignore(pos, end, "if")){
        string label = env.label.make_unique();
        if(parse_expression(env, pos, end)){
            env.out
                <<"testl %eax, %eax"<<endl
                <<"jz " <<label<<endl;
            if(parse_block(env, pos, end)){

```

```

        env.out<<label<<": "<<endl;
        guard.no_backtrack();
        return true;
    }
}
return false;
}

```

---

So, nun muss nur noch *parse\_command* leicht angepasst werden.

---

```

bool parse_command(Env&env, const char*&pos, const char*end){
    BacktrackGuard guard(pos);
    if(parse_print(env, pos, end)){
        guard.no_backtrack();
        return true;
    }else if(parse_if(env, pos, end)){
        guard.no_backtrack();
        return true;
    }else
        ...
}

```

---

An sich war es das schon, allerdings ist ein Vergleich, ob denn nun zwei Werte gleich sind, noch ziemlich haarig.

---

```

{
    var a
    var b
    a = 4
    b = 3
    if a - b
        print "nicht gleich"
}

```

---

Deswegen muss *parse\_expression* verändert werden. Als erste nenne ich *parse\_expression* in *parse\_numeric\_expression* um und schreibe ein paar weitere Funktionen. Diese kümmern sich um das Einlesen der Vergleichsoperatoren und der logischen Verknüpfungen. Dadurch werden die Funktionen nicht endlos lang.

---

```

bool parse_compare_expression(Env&env, const char*&pos, const char*end){
    if(ignore(pos, end, "true")){
        env.out<<"movl $1, %eax"<<endl;
        return true;
    }else if(ignore(pos, end, "false")){
        env.out<<"movl $0, %eax"<<endl;
        return true;
    }else if(parse_numeric_expression(env, pos, end)){
        BacktrackGuard guard(pos);
    }
}

```

```

string condition_code;
if(ignore(pos, end, "="))
    condition_code = "e";
else if(ignore(pos, end, "!="))
    condition_code = "ne";
else if(ignore(pos, end, "<="))
    condition_code = "be";
else if(ignore(pos, end, ">="))
    condition_code = "ae";
else if(ignore(pos, end, "<"))
    condition_code = "b";
else if(ignore(pos, end, ">"))
    condition_code = "a";
else
    cerr<<"Unknown compare operator"<<endl;
env.out<<"pushl %eax"<<endl;
if(parse_numeric_expression(env, pos, end)){
    env.out
        <<"cmpl %eax, (%esp)"<<endl
        <<"movl $0, %eax"<<endl
        <<"set"<<condition_code<<" %al"<<endl
        <<"addl $4, %esp"<<endl;
    guard.no_backtrack();
    return true;
}
}
return false;
}

bool parse_boolean_expression(Env&env, const char*&pos, const char*end){
    BacktrackGuard guard(pos);
    bool not_operation = false;
    if(ignore(pos, end, "!"))
        not_operation = true;
    if(parse_compare_expression(env, pos, end)){
        for(;;){
            string operation;
            if(ignore(pos, end, "&"))
                operation = "and";
            else if(ignore(pos, end, "|"))
                operation = "or";
            else if(ignore(pos, end, "^"))
                operation = "xor";
            else{
                if(not_operation)
                    env.out<<"notl %eax"<<endl;
                guard.no_backtrack();
                return true;
            }
        }
        env.out<<"pushl %eax"<<endl;
        if(!parse_compare_expression(env, pos, end))
            return false;
        env.out
            <<operation<<" %eax, (%esp)"<<endl
            <<"addl $4, %esp"<<endl;
    }
}

```

```

    }
}
return false;
}

```

---

Man sollte beachten, dass die Reihenfolge der Abfragen in *parse\_compare\_expression* nicht egal ist. Wenn mehrere Operatoren mit dem gleichen Zeichen anfangen, so muss man zuerst auf den längsten testen, ansonsten wird der Operator entzweierteilt. Aus "*<=*" würde ein "*<*", gefolgt von einem "*=*" und dies würde zu einem Fehler führen.

In *parse\_if* wird natürlich *parse\_boolean\_expression* verwendet und nicht *parse\_numeric\_expression*. Dadurch ist die doppelte Bedeutung von "*=*" auch nicht zweideutig. Aus dem Kontext heraus kann bestimmt werden, ob es ein Vergleich oder eine Zuweisung sein soll.

Nun fehlt nur noch ein *else*. Dafür müssen wir *parse\_if* anpassen.

---

```

bool parse_if(Env&env, const char*&pos, const char*end){
    BacktrackGuard guard(pos);
    if(ignore(pos, end, "if")){
        string false_label = env.label.make_unique();
        if(parse_boolean_expression(env, pos, end)){
            env.out
                <<"testl %eax, %eax"<<endl
                <<"jz " <<false_label<<endl;
            if(parse_block(env, pos, end)){
                if(!ignore(pos, end, "else")){
                    env.out<<false_label<<":"<<endl;
                    guard.no_backtrack();
                    return true;
                }
                string end_label = env.label.make_unique();
                env.out
                    <<"jmp " <<end_label<<endl
                    <<false_label<<":"<<endl;
                if(!parse_block(env, pos, end))
                    return false;
                env.out<<end_label<<":"<<endl;
                guard.no_backtrack();
                return true;
            }
        }
    }
    return false;
}

```

---

Schleifen sind auch nicht sonderlich verschieden von einem *if*. Als erstes schreibe ich eine *parse\_while*-Funktion und die wird dann in *parse\_command* genauso wie *parse\_if* aufgerufen.

---

```

bool parse_while(Env&env, const char*&pos, const char*end){
    BacktrackGuard guard(pos);
    if(ignore(pos, end, "while")){
        string
            reloop_label = env.label.make_unique(),
            end_label = env.label.make_unique();
        env.out<<reloop_label<<": "<<endl;
        if(parse_boolean_expression(env, pos, end)){
            env.out
                <<"testl %eax, %eax"<<endl
                <<"jz " <<end_label<<endl;
            if(parse_block(env, pos, end)){
                env.out
                    <<"jmp " <<reloop_label<<endl
                    <<end_label<<": "<<endl;
                guard.no_backtrack();
                return true;
            }
        }
    }
    return false;
}

```

---

Nun ist unser Compiler schon fähig, sinnvolle Programme zu übersetzen. Folgendes Programm zählt von 0 bis 10:

---

```

{
    var i
    i = 0
    while i<=10{
        print i, "\n"
        i = i+1
    }
}

```

---

Hier ist zum Beispiel ein Programm, das alle Primzahlen bis 100 sucht:

---

```

{
    print "Between 1 and 100 the following prime numbers exist : 2"
    var p
    p = 3
    while p < 100{
        var d
        d = 2
        var is_prime
        is_prime = 1
        while d<p{
            if p%d = 0{
                is_prime = 0
            }
            d = d+1
        }
    }
}

```

```
        if is_prime != 0{
            print ", ", p
        }
        p = p+1
    }
}
```

---

Aufmerksamen Lesern fällt wahrscheinlich auf, dass ich `is_prime` nicht mit `true` initialisiere. Nun, dies liegt daran, dass dies kein numerischer Wert ist. Diese strikte Unterscheidung zwischen boolschem und numerischem Wert ermöglicht aber auch, ein einfaches Gleichzeichen sowohl für eine Zuweisung als auch für einen Vergleich zu nehmen. Es hat also sowohl Vor- wie auch Nachteile. In einer kompletten Sprache gäbe es natürlich auch einen boolschen Typ und dadurch wäre das Problem elegant gelöst.

Eine for-Schleife ist nicht signifikant anders und ihre Implementierung wird dem Leser überlassen.

## 11 Funktionen

Funktionen sind die Bausteine jedes komplexeren Programms. Darum will ich sie auch in diesem Artikel nicht auslassen. Mit Funktionen kann man sich, was die Komplexität angeht, beliebig nach oben steigern. Templatefunktionen sind ein gutes Beispiel dafür. Hier will ich mich aber auf recht einfache Funktionen beschränken. Für jede Funktion speichern wir den Namen und die Anzahl der Parameter. Da es nur `int` als Variablentyp gibt, brauchen wir keine Informationen über die Typen der Parameter zu speichern. Der Rückgabewert ist auch immer ein `int`. Funktionen ohne Rückgabebetyp machen natürlich Sinn, aber das machen Templatefunktionen auch. Als Kompromiss zwischen Sinn und Einfachheit lassen wir es zu, dass ein Rückgabewert nicht initialisiert ist. Das heißt er enthält Schrott.

Funktionen müssen wie in C im Voraus deklariert werden und können mehrfach gleich deklariert werden. Sie können nicht überladen werden. Als Aufrufskonvention werden wir eine verwenden, die der des GCCs ähnlich ist. Dadurch können wir auch ohne weiteres gegen C-Code linken. Der Unterschied besteht darin, dass Argumente von links nach rechts auf den Stack gepackt werden. Beim GCC werden die Parameter von rechts nach links auf den Stack gepackt. Der Grund hierfür ist, dass es so für uns einfacher ist. Der Aufrufer ist dafür verantwortlich, die Argumente wieder vom Stack zu nehmen. Eine Funktion darf ihre Parameter verändern. Der Aufrufer darf also keine Argumente recyceln. Der Rückgabewert befindet sich in `eax`.

Als erstes brauchen wir eine Klasse, um die Informationen zu verwalten. Ich werde wieder einmal nur auf die Klassenschnittstelle eingehen und auf eine Implementierung verweisen<sup>[5]</sup>.

---

```
class FuncManager{
public:
    bool declare(string name, unsigned parameter_count);
    bool exists(string name) const;
    string get_label(string name) const;
    unsigned get_parameter_count(string name) const;
};
```

---

- *declare* macht eine Funktion bekannt.
- *exists* überprüft, ob eine Funktion bekannt gemacht wurde.
- *get\_label* gibt das Label des Funktionskörper zurück.
- Mit *get\_parameter\_count* findet man heraus, wie viele Parameter eine Funktion braucht.

Um auf Argumente zurückgreifen zu können und Werte zurückgeben zu können, müssen wir auch *VarManager* verändern. Konkret werden ein paar Methoden hinzugefügt und die Klasse sieht nun folgendermaßen aus:

---

```
class VarManager{
public:
    bool create_var(string name, ostream&out);
    bool add_arg(string name, unsigned pos);
    bool is_var_defined(string name)const;
    string get_var_address(string name)const;
    void push_scope();
    void pop_scope(ostream&out);
    void simulate_pop_all_scopes(ostream&out)const;
};
```

---

- *add\_arg* gibt einem Parameter einen Namen. *pos* gibt an, um das wie vielte Argument von links es sich handelt. Das erste Argument hat den Index 0. Der Rückgabewert gibt an, ob es bereits einen solchen Parameter gab oder nicht. Argumente werden in ihrem eigenen Scope angelegt und darum können sie mit gleichnamigen lokalen Variablen koexistieren. Sie werden jedoch von denen verdeckt.
- *simulate\_pop\_all\_scopes* generiert Code, um alle Scopes zu löschen, ohne dies jedoch zu tun. Dies wird benötigt, um ein *return* wie in C zu realisieren. Für andere Formen von Sprüngen, wie zum Beispiel *break*, würde man einen ähnlichen Mechanismus benötigen.

Eine Implementierung gibt es hier<sup>[6]</sup>.

In einem C-Programm unterscheidet man zwischen globalem Scope und Funktionskörper. In beiden Bereichen sind unterschiedliche Sprachkonstrukte erlaubt. So kann man zum Beispiel keine Funktion innerhalb einer anderen definieren. Schleifen können sich auch nicht im globalen Scope befinden. Diesen Unterschied wollen wir nun auch in unserer Sprache einführen.

Da wir keine globalen Variablen zulassen, kann man sich fragen, ob ein *VarManager* im globalen Scope Sinn macht. Meiner Meinung nach nicht. Deswegen gibt es auch für beide Teile des Programms unterschiedliche *Env*-Strukturen. Die eine nenne ich *FuncEnv*, da sie nur innerhalb einer Funktion Sinn macht und die andere einfach *Env*. Da wir uns bis jetzt nur um den Inhalt von Funktionen gekümmert haben, wird jedes alte *Env* in ein *FuncEnv* umbenannt.

---

```
struct FuncEnv{
    explicit FuncEnv(VarManager&var, ostream&out,
        DataSection&data, LabelManager&label,
        const FuncManager&func):
        var(var), out(out), data(data), label(label),
        func(func){}
```

```

    VarManager&var;
    ostream&out;
    DataSection&data;
    LabelManager&label;
    const FuncManager&func;
};

struct Env{
    explicit Env(ostream&out, DataSection&data,
                LabelManager&label, FuncManager&func):
        out(out), data(data), label(label), func(func){}

    ostream&out;
    DataSection&data;
    LabelManager&label;
    FuncManager&func;
};

```

---

Der *FuncManager* in *FuncEnv* ist konstant, da man keine Funktion innerhalb einer anderen erschaffen kann. In *main* wird eine Instanz von *FuncManager* angelegt wie wir es bereits für die meisten anderen Verwaltungsstrukturen gemacht haben.

Als nächstes brauchen wir eine Funktion zum Einlesen von Funktionsdefinitionen und Deklarationen.

---

```

bool read_arg_list(const char*&pos, const char*end, vector<string>&args){
    BacktrackGuard guard(pos);
    if(!ignore(pos, end, "("))
        return false;
    string arg_name;
    while(read_identifizier(pos, end, arg_name)){
        args.push_back(arg_name);
        if(!ignore(pos, end, ","))
            break;
    }
    if(!ignore(pos, end, ")"))
        return false;
    guard.no_backtrack();
    return true;
}

bool parse_function(Env&env, const char*&pos, const char*end){
    BacktrackGuard guard(pos);
    string func_name;
    if(!read_identifizier(pos, end, func_name))
        return false;

    vector<string>arguments;
    if(!read_arg_list(pos, end, arguments))
        return false;

    if(env.func.exists(func_name)){
        if(env.func.get_parameter_count(func_name)
           != arguments.size())
            cerr<<"Function " <<func_name

```

```

                <<" redeclared with wrong argument number"<<endl;
    }else
        env.func.declare(func_name, arguments.size());

    if(ignore(pos, end, ";")){
        guard.no_backtrack();
        return true;
    }

    VarManager var;

    for(unsigned j=0; j<arguments.size(); ++j)
        if(!var.add_arg(arguments[j], arguments.size()-j-1))
            cerr<<"2 arguments with same name "
                <<arguments[j]<<" in function "
                <<func_name<<endl;

    env.out
        <<".globl "<<env.func.get_label(func_name)<<endl
        <<env.func.get_label(func_name)<<": "<<endl
        <<"push %ebp"<<endl
        <<"mov %esp, %ebp"<<endl;
    ;

    FuncEnv fenv(var, env.out, env.data, env.label, env.func);
    if(parse_block(fenv, pos, end)){
        env.out
            <<"pop %ebp"<<endl
            <<"ret"<<endl;
        guard.no_backtrack();
        return true;
    }

    return false;
}

```

---

Als nächstes sorgen wir dafür, dass eine Funktion auch aufgerufen werden kann. Dafür schreiben wir uns als erstes eine Funktion, die einen Aufruf einliest.

---

```

bool parse_function_call(FuncEnv&env, const char*&pos, const char*end){
    BacktrackGuard guard(pos);

    string func_name;
    if(!read_identifier(pos, end, func_name))
        return false;
    else{
        if(ignore(pos, end, "(")){
            unsigned parameter_count = 0;
            if(!ignore(pos, end, ")")){
                do{
                    if(!parse_numeric_expression(env, pos, end))
                        return false;
                    env.out<<"pushl %eax"<<endl;
                    ++parameter_count;
                }
            }
        }
    }
}

```

```

        }while(ignore(pos, end, ","));
        if(!ignore(pos, end, "))")
            return false;
    }

    if(!env.func.exists(func_name))
        cerr<<"Unknown function "<<func_name<<endl;
    else if(env.func.get_parameter_count(func_name)
            != parameter_count)
        cerr<<"Wrong number of parameters to function "
            <<func_name<<endl;
    else{
        env.out<<"call "<<env.func.get_label(func_name)<<endl;
        if(parameter_count != 0)
            env.out<<"addl $"<<4*parameter_count
                << ", %esp"<<endl;
        guard.no_backtrack();
        return true;
    }
}
return false;
}
}

```

---

Eine Funktion kann innerhalb eines Ausdrucks und eines Befehls aufgerufen. Es müssen also *parse\_factor* und *parse\_command* verändert werden. In *parse\_factor* muss man *parse\_function\_call* vor *parse\_literal* aufrufen, da ansonsten der Funktionsname als Variable interpretiert wird.

Nun fehlt nur noch die Möglichkeit, einen Wert zurückzugeben. Die Funktion, die sich darum kümmert, sollte nicht zu kompliziert sein, wenn man es geschafft hat, bis hier zu folgen.

---

```

bool parse_return(FuncEnv&env, const char*&pos, const char*end){
    BacktrackGuard guard(pos);

    if(ignore(pos, end, "return")){
        if(parse_numeric_expression(env, pos, end)){
            env.var.simulate_pop_all_scopes(env.out);
            env.out
                <<"pop %ebp"<<endl
                <<"ret"<<endl;
            guard.no_backtrack();
            return true;
        }
    }
    return false;
}

```

---

Diese Funktion wird nun noch in *parse\_command* aufgerufen und schon können wir Funktionen übersetzen.

Nun müssen wir nur noch *parse\_program* und *main* anpassen.

---

```
bool parse_program(Env&env, const char*&pos, const char*end){
    while(parse_function(env, pos, end))
        {}

    return true;
}

int main(){
    ifstream in("in.txt");
    vector<char>source_code;
    copy(istreambuf_iterator<char>(in.rdbuf()),
        istreambuf_iterator<char>(),
        back_inserter(source_code));

    ofstream out("out.s");
    out<<".text"<<endl;

    const char*pos = &source_code.front();
    const char*end = &source_code.back() + 1;

    DataSection data;
    LabelManager label;
    FuncManager func;
    Env env(out, data, label, func);

    if(parse_program(env, pos, end)){
        skip_spaces(pos, end);
        if(pos != end)
            cerr<<"Unexpected trailing characters at end of file"<<endl;
        else
            data.write_data_section(out);
    }
}
```

---

Ein "Hello World"-Programm sieht nun wie folgend aus:

---

```
prog(){
    print "Hello World!"
}
```

---

Selbst rekursive Funktionen sind kein Problem. Folgendes Programm berechnet die Fakultät einer Zahl rekursiv:

---

```
fact(n){
    if n = 0
        return 1
    else
        return n*fact(n-1)
}

prog(){
    print fact(9)
}
```

---

Wir können aber nun auch gegen C-Funktionen linken. Zum Beispiel kann man so etwas von der Konsole einlesen.

---

```
int read_int(){
    int r = -1;
    scanf("%d", &r);
    return r;
}
```

---

```
read_int();

prog(){
    var a
    print "a = "
    a = read_int()
    var b
    print "b = "
    b = read_int()
    print "a + b = ", a+b
    print "a - b = ", a-b
    print "a * b = ", a*b
    print "a / b = ", a/b
    print "a % b = ", a%b
}
```

---

Man muss nur beachten, dass die Parameter in C eine andere Reihenfolge haben werden. Die Funktion  $test(a,b,c)$  würde der C-Funktion  $test(c,b,a)$  entsprechen.

## 12 Parserbaum

Wer schon einmal vorher mit der Materie in Kontakt gekommen ist, wird sich wahrscheinlich wundern, warum ich ihn bis jetzt noch nicht erwähnt habe, ja vielleicht sogar wundern, wie wir überhaupt so weit ohne ihn gekommen sind. Die Rede ist vom Parserbaum.

Die Idee ist es, das Interpretieren der Eingangszeichenkette und das Generieren der Assemblercodes zu trennen. Diese Trennung bietet viele Vorteile, wobei die Möglichkeit, elegant mehrere Prozessoren zu unterstützen nur die offensichtlichste ist.

Der Parser liest den Code und generiert einen Baum, welcher alle wichtigen Informationen enthält. Der Generator nimmt diesen Baum dann und generiert den entsprechenden Code. Dies macht Arbeitsteilung im Entwicklerteam viel einfacher, da beide Teile nun unabhängig voneinander sind.

Ein weiterer großer Vorteil ist, dass es möglich wird, Funktionen zu inlinen und auch andere Optimierungen durchzuführen. Ich wüsste nicht, wie man dem Compiler, so wie er hier vorgestellt wurde, beibringen soll, Funktionsaufrufe zu ersetzen. Wenn man den Code allerdings als Baum vorliegen hat, so muss man nur einen Ast des Baums kopieren und noch ein paar Umformungen bezüglich der Variablenamen durchführen. In einem Baum ist es auch leicht, Informationen über Unterausdrücke einzuholen. Zum Beispiel, ob ein Ausdruck noch etwas anderes tut außer den Rückgabewert zu berechnen. Dies ist wichtig um zum Beispiel  $0*x$  durch  $0$  zu ersetzen.

In diesem Artikel wollen wir aber keinen Parserbaum mehr entwickeln. Es würde einfach den Rahmen sprengen.

## 13 Schlussbemerkung

Ziel dieses Artikels war es, den Leser in das Gebiet des Compilerbaus einzuführen und zwar ohne ihn mit Theorie zu bombardieren. Ich hoffe, Sie haben jetzt einen Überblick über die Materie und können sich vorstellen, wo man die Aussagen der Theorie praktisch einsetzen kann. Wer sich für das Thema interessiert, dem sei gesagt, dass es sich hier nur um die Spitze des Eisbergs handelt. Es gibt noch sehr viel, was in diesem Artikel noch nicht einmal mit einem Wort erwähnt wurde.

Ob es von meiner Seite einen weiteren Artikel zu diesem Thema geben wird, ist ungewiss. Es fällt mir schwer, einzuschätzen, ob ich in Zukunft noch genug Zeit und Motivation aufbringen kann. Natürlich hängt es auch davon ab, wie viele Leute sich dafür interessieren.

## Verweise

- <sup>[1]</sup> [http://de.wikipedia.org/wiki/Erweiterte\\_Backus-Naur-Form](http://de.wikipedia.org/wiki/Erweiterte_Backus-Naur-Form)

## Quellcode

[2] [http://www.c-plusplus.de/magazin/bilder/Compiler/var\\_man1.cpp](http://www.c-plusplus.de/magazin/bilder/Compiler/var_man1.cpp)

---

```
#include <vector>
#include <string>
#include <ostream>
#include <sstream>
#include <cassert>
#include <algorithm>
using namespace std;
```

```

class VarManager{
public:
    bool create_var(string name, ostream&out);
    bool is_var_defined(string name)const;
    string get_var_address(string name)const;
    void push_scope();
    void pop_scope(ostream&out);

private:
    vector<string>var_name;
    vector<unsigned>scope_start;
};

bool VarManager::create_var(string name, ostream&out){
    assert(!scope_start.empty());
    if(std::find(var_name.begin() + scope_start.back(), var_name.end(), name) == var_name.end()){
        var_name.push_back(name);
        out<<"subl $4, %esp"<<endl;
        return true;
    }else
        return false;
}

bool VarManager::is_var_defined(string name)const{
    assert(!scope_start.empty());
    return std::find(var_name.begin(), var_name.end(), name) != var_name.end();
}

string VarManager::get_var_address(string name)const{
    assert(!scope_start.empty());
    for(unsigned i_plus_one = var_name.size(); i_plus_one > 0; --i_plus_one)
        if(var_name[i_plus_one-1] == name){
            ostream out;
            out<<"-"<< 4 *(i_plus_one)<<"(%ebp)";
            return out.str();
        }
    assert(false);
}

void VarManager::push_scope(){
    scope_start.push_back(var_name.size());
}

void VarManager::pop_scope(ostream&out){
    assert(!scope_start.empty());
    unsigned to_pop = 4*(var_name.size() - scope_start.back());
    if(to_pop != 0){
        out<<"addl $"<<to_pop<<" , %esp"<<endl;
        var_name.erase(var_name.begin() + scope_start.back(), var_name.end());
    }
    scope_start.pop_back();
}

```

---

**[3]** [http://www.c-plusplus.de/magazin/bilder/Compiler/data\\_sec.cpp](http://www.c-plusplus.de/magazin/bilder/Compiler/data_sec.cpp)

---

```
#include <string>
#include <map>
#include <ostream>
#include <cassert>
using namespace std;

class DataSection{
public:
    void add_string_data(string label, string data);
    void write_data_section(ostream&)const;

private:
    map<string, string>string_table;
};

void DataSection::add_string_data(string label, string data){
    assert(string_table.find(label) == string_table.end());
    string_table.insert(make_pair(label, data));
}

string escape_string(string in){
    string out;
    for(unsigned i=0; i<in.length(); ++i)
        switch(in[i]){
            case '\n':out+="\\n";break;
            case '\t':out+="\\t";break;
            case '\\':out+="\\";break;
            default:out+=in[i];break;
        }
    return out;
}

void DataSection::write_data_section(ostream&out)const{
    typedef map<string, string>::const_iterator iter;
    for(iter i=string_table.begin(); i!=string_table.end(); ++i)
        out<<i->first<<":.ascii \\"<<escape_string(i->second)<<"\\0\\"<<endl;
}
```

---

**[4]** [http://www.c-plusplus.de/magazin/bilder/Compiler/label\\_man.cpp](http://www.c-plusplus.de/magazin/bilder/Compiler/label_man.cpp)

---

```
#include <string>
#include <sstream>
using namespace std;

class LabelManager{
public:
    LabelManager();

    string make_unique();

private:
```

```

    unsigned i;
};

LabelManager::LabelManager():i(0){}

string LabelManager::make_unique(){
    ostringstream out;
    out<<"L"<<i++;
    return out.str();
}

```

---

[5] [http://www.cplusplus.de/magazin/bilder/Compiler/func\\_man.cpp](http://www.cplusplus.de/magazin/bilder/Compiler/func_man.cpp)

---

```

#include <map>
#include <string>
#include <ostream>
#include <sstream>
#include <cassert>
using namespace std;

class FuncManager{
public:
    bool declare(string name, unsigned parameter_count);
    bool exists(string name)const;
    string get_label(string name)const;
    unsigned get_parameter_count(string name)const;

private:
    map<string, unsigned>table;
};

bool FuncManager::declare(string name, unsigned parameter_count){
    map<string, unsigned>::iterator found = table.find(name);
    if(found == table.end()){
        table.insert(make_pair(name, parameter_count));
        return true;
    }else{
        return found->second == parameter_count;
    }
}

bool FuncManager::exists(string name)const{
    map<string, unsigned>::const_iterator found = table.find(name);
    return found != table.end();
}

string FuncManager::get_label(string name)const{
    return "_" + name;
}

unsigned FuncManager::get_parameter_count(string name)const{
    map<string, unsigned>::const_iterator found = table.find(name);
    assert(found != table.end());
    return found->second;
}

```

```
#include <vector>
#include <string>
#include <ostream>
#include <sstream>
#include <cassert>
#include <algorithm>
using namespace std;

class VarManager{
public:
    bool create_var(string name, ostream&out);
    bool add_arg(string name, unsigned pos);

    bool is_var_defined(string name) const;
    string get_var_address(string name) const;
    void push_scope();
    void pop_scope(ostream&out);

    void simulate_pop_all_scopes(ostream&out) const;
private:
    vector<string>var_name;
    vector<unsigned>scope_start;
    vector<string>args;
};

bool VarManager::create_var(string name, ostream&out){
    assert(!scope_start.empty());
    if(std::find(var_name.begin() + scope_start.back(), var_name.end(), name) == var_name.end()){
        var_name.push_back(name);
        out<<"subl $4, %esp"<<endl;
        return true;
    }else
        return false;
}

bool VarManager::add_arg(string name, unsigned pos){
    // Does argument with the same name exist?
    if(find(args.begin(), args.end(), name) != args.end())
        return false;

    args.resize(max(args.size(), pos+1));

    // Is the stack position still empty?
    if(args[pos].empty()){
        args[pos] = name;
        return true;
    }else
        return false;
}

bool VarManager::is_var_defined(string name) const{
    assert(!scope_start.empty());
    if(std::find(var_name.begin(), var_name.end(), name) != var_name.end())
```

```

        return true;
    if(std::find(args.begin(), args.end(), name) != args.end())
        return true;
    return false;
}

string VarManager::get_var_address(string name)const{
    assert(!scope_start.empty());
    for(unsigned i_plus_one = var_name.size(); i_plus_one > 0; --i_plus_one)
        if(var_name[i_plus_one-1] == name){
            ostream out;
            out<<"-"<< 4 *(i_plus_one)<<"(%ebp)";
            return out.str();
        }

    for(unsigned i=0; i<args.size(); ++i)
        if(args[i] == name){
            ostream out;
            out<< 4 *(i+2)<<"(%ebp)";
            return out.str();
        }
    assert(false);
}

void VarManager::push_scope(){
    scope_start.push_back(var_name.size());
}

void VarManager::pop_scope(ostream&out){
    assert(!scope_start.empty());
    unsigned to_pop = 4*(var_name.size() - scope_start.back());
    if(to_pop != 0){
        out<<"addl $"<<to_pop<<" , %esp"<<endl;
        var_name.erase(var_name.begin() + scope_start.back(), var_name.end());
    }
    scope_start.pop_back();
}

void VarManager::simulate_pop_all_scopes(ostream&out)const{
    assert(!scope_start.empty());
    unsigned to_pop = 4*var_name.size();
    if(to_pop != 0)
        out<<"addl $"<<to_pop<<" , %esp"<<endl;
}

```

---