COMPILERBAU UND FORMALE SPRACHEN

Aufgabenblatt 2

Aufgabe 1. *Lexer Token mit clang ausgeben*

Clang ist das am weitesten verbreitete Frontend für das LLVM Compiler Framework. Es unterstüzt die Programmiersprachen C, C++, Objective-C und Objective-C++ sowie diverse APIs[1] (z.B. OpenMP, OpenCL, und CUDA). Für das Backend benutzt Clang das LLVM Compiler-Framework. LLVM ist vor allem bekannt wegen seines modularen Aufbaus und einer Vielzahl von Möglichkeiten zur Code-Analyse und Optimierung.

In dieser Aufgabe sollen Sie den Clang Compiler verwenden, um die Ausgaben der ersten Phase des Compilers anzuzeigen. Clang bietet eine Vielzahl von Optionen, um Zwischenrepräsentationen, die während der Übersetzung einer Quelldatei erstellt werden, anzuzeigen. Für den Lexer sind diese vor allem die beiden folgenden Optionen `-dump-tokens` und `-dump-raw-tokens`.

Legen Sie ein C-Quellcode Datei an und schreiben Sie darin eine ”kleine” C-Funktion. Sie können z.B. die unten angegebene Funkton `sum` verwenden. Rufen Sie dann den Compiler mit `clang -cc1 -dump-tokens sum.c` auf. Gehen Sie die Ausgaben durch und versuchen Sie die Ausgaben des Lexers nachzuvollziehen. Was ändert sich, wenn Sie die Option `-dump-raw-tokens` verwenden?

```
1 #define ergebnis return
2 unsigned sum (unsigned n){
3         unsigned j, s=0;
4         for (j = 1; j <= n; j++)
5         s += j;
6         ergebnis s;
7 }
```

[1] https://de.wikipedia.org/wiki/Programmierschnittstelle

Aufgabe 2. *Zwischencode mit GCC ausgeben*

In der Analysephase transformieren moderne Compiler Quellcode in eine Zwischenrepräsentation, auf dessen Grundlage verschiedene Optimierungsläufe durchgeführt werden können. Das bekannte gcc Framework besitzt gleich drei Zwischenrepräsentationen, GENERIC, GIMPLE und RTL. Die GIMPLE Repräsentation wird dabei für plattformunabhängige Optimierungen verwendet.

In dieser Aufgabe wollen wir uns anschauen, welche Auswirkungen Compiler-Optionen zur Codeoptimierung haben. Genau wie auch Clang besitzt auch gcc eine Vielzahl von Optimierungsläufen, den sogenannten *Passes*. Dabei stellt der Programmierer normalerweise nicht per Hand ein, welche der Passes ausgeführt werden sollen. Es gibt eine allgemeine Option `-O`, mit der man den Optimierungsgrad für einen Übersetzungsvorgang angeben kann. Beim gcc gibt es 5 Stufen, von `-O0` (keine Optimierung) bis `-O3` (höchste Optimierung) und `-Os` für die Optimierung der Code-Größe.

Schreiben Sie eine einfache C-Funktion, Sie können hier auch die Funktion aus Aufgabe 3 wiederverwenden.

1. Schauen Sie sich zuerst den Assembler Code an, den gcc mit den verschiedenen Optimierungsstufen für ihre C-Datei generiert. Mit `gcc -O<L> -S <datei>.c` übersetzen Sie die Datei `<datei>.c` in eine Datei `<datei>.s` , die die generierten Assemblerbefehle beinhaltet. Der Platzhalter `<L>` steht dabei für den Optimierungs-Level (`0-3`, bzw. `s`).
2. Um einen Eindruck zu erhalten, welche Optimierungsläufe durchgeführt werden, können Sie beim Übersetzen die Option `-fdump-passes` angeben. Also etwa so:
   gcc -Os -S sum.c -fdump-passes
3. Die Ausgabe von Zwischencode erreichen Sie durch die Compiler-Optionen `-c -fdump-tree-<format>`. Als `<format>` können Sie z.B. *gimple* oder *ssa* wählen. Die *Static Single Assignment Form* (SSA) ist eine Form der Zwischenrepräsentation, die wir im späteren Teil des Moduls noch aufgreifen werden. Die zusätzliche Option `-c` benötigen Sie an dieser Stelle, damit gcc den Code nur Compiliert, aber nicht versucht zu linken.
   Durch die `fdump-tree` Optionen generiert der Compiler Ausgaben in neue Dateien. Schauen Sie sich den Inhalt an. Warum kann man sagen, dass der Zwischencode eine vereinfachte (bzw. eine verallgemeinerte) Darstellung des Quellcodes ist?
4. Eine Option, die sehr viele Ausgaben erzeugt ist `-fdump-tree-all-graph`. Damit generieren Sie graphische Repräsentationen für den Zustand des Zwischencodes nach diversen Analyse- und Optimierungsläufen. Die Ausgaben erfolgen dabei im `dot`-Format. Um daraus Abbildungen zu erzeugen, benötigen Sie das Programmpaket *Graphviz* mit dem Programm `dot`. Führen Sie z.B. folgende Anweisungen aus:
   ```
   gcc -c -O3 -fdump-tree-all-graph sum.c
   dot sum.c.020t.ssa.dot -Tpng > sum_ssa.png
   dot sum.c.227t.optimized.dot -Tpng > sum_opt.png
   ```
   Vergleichen Sie die beiden Abbildungen. Was ist mit dem Programm passiert?

Aufgabe 3. *Flex-Scanner für Kommentare*

1 Entwickeln Sie einen Scanner mit Flex, der aus C-Programmmen die Zeilenkommentare extrahiert und folgendermaßen ausgibt:

```
Zeilenkommentar bei Zeile 7:
// 2 ist die kleinste Primzahl
Zeilenkommentar bei Zeile 13:
// Ist die Zahl durch i teilbar?
Zeilenkommentar bei Zeile 19:
// Die Zahl ist nicht durch andere Ints >1 teilbar -> Prim!
```

2 Versuchen Sie, die Zeilennummer über die Lex-Variable `yylineno` mit auszugeben. Dazu setzen Sie im Definitionsteil des Scanners die Opion `%option yylineno`.

3 Bisher liest Ihr Scanner die Eingaben über den Umleitungsoperator `<` direkt von der Standardeingabe. Erweitern Sie Ihren Scanner so, dass sie einen Dateinamen als Parameter übergeben können. Testen Sie in der `main`-Funktion, ob ein Parameter übergeben wurde. Falls ja setzen Sie die Variable `yyin` auf den übergebenen Dateinamen.

4 Erweitern Sie Ihren Scanner so, dass er auch in der Lage ist, Blockkommentare (`/* */`) zu erkennen.

```c
/* Einfache Funktion zum Testen, ob
 * eine ganze Zahl eine Primzal ist
 */
void ist_prim(int zahl) {
    int i;
        // 2 ist die kleinste Primzahl
        if (zahl < 2) return 0;
        /* Teste alle Zahlen bis n/2.
         * Es würde genügen bis sqrt(n) zu testen.
         */
    for ( i=2; i<zahl/2; i++){
                // Ist die Zahl durch i teilbar?
        if(zahl%i==0){
            printf("%d ist keine Primzahl (%d/%d=%d)\n",
                        zahl, zahl, i, zahl/i);
            return;
        }
    }
        // Die Zahl ist nicht durch andere Ints >1 teilbar -> Prim!
        printf("%d ist eine Primzahl\n", zahl);
}
```
