Codegenerierung

- besteht generell aus drei Aspekten

  (A) Auswahl einer geeigneten Kombination von Maschinenbefehlen

  Minimierung der Summe an Taktzyklen aller Befehle

  (B) Zuweisung von Variablen an Register (stark gekoppelt mit A)

  Minimierung des Hauptspeicherzugriffs

  (C) Bestimmung der Reihenfolge von Befehlen

  Eigenheiten des Prozessors ausnutzen (Pipelining, Branch Prediction)
- wird ausführlich im zweiten Teil der Vorlesung behandelt
- jetzt: nur triviale Codegenerierung, basiert auf
  - Modifikation eines Interpreters, geschrieben in Host-Sprache
  - erzeuge Codestücke in Host-Sprache
  - übersetze sie mit dem Compiler für die Host-Sprache

Triviale Codegenerierung

- Idee:
  - iterativer Interpreter: pro Knoten eine Aktion (Abb. 4.6)
  - trivialer Compiler: (Abb. 4.12)
    - gleiche Schleife wie der Interpreter, aber generiert Code
    - Active Node-Pointer→ Befehlszähler
- Generierter Code: (Abb. 4.13)
- Beobachtungen:
  - auf beliebig komplizierte Quellsprachen anwendbar
  - bei existierendem (iterativem) Interpreter "kostenfreier" Compiler
  - konvertiert komplizierten Quellcode in einfachen Zielcode
  - generierter Code sehr naiv (hohe Repetitivität)
- Optimierungen:
  - Threaded Code
  - Partielle Auswertung

Vergleich: iterative Interpretation / triviale Codegenerierung

```
static void Interpret_iteratively(void) {
    while (Active_node_pointer != 0) {
        /* there is only one node type, Expression: */
        Expression *expr = Active_node_pointer;
        switch (expr->type) {
        case 'D':
            Push(expr->value);
            break;
        case 'P': {
            int e_left = Pop(); int e_right = Pop();
            switch (expr->oper) {
            case '+': Push(e_left + e_right); break;
            case '*': Push(e_left * e_right); break;
            }}
            break;
        }
        Active_node_pointer = Active_node_pointer->successor;
    }
    printf("%d\n", Pop());      /* print the result */
}
```

```
static void Trivial_code_generation(void) {
    printf("#include    \"stack.h\"\nint main(void) {\n");
    while (Active_node_pointer != 0) {
        /* there is only one node type, Expression: */
        Expression *expr = Active_node_pointer;
        switch (expr->type) {
        case 'D':
            printf("Push(%d);\n", expr->value);
            break;
        case 'P':
            printf("{\n\
                int e_left = Pop(); int e_right = Pop();\n\
                switch (%d) {\n\
                case '+': Push(e_left + e_right); break;\n\
                case '*': Push(e_left * e_right); break;\n\
                }}\n",
                expr->oper
            );
            break;
        }
        Active_node_pointer = Active_node_pointer->successor;
    }
    printf("printf(\"%%d\\n\", Pop()); /* print the result */\n");
    printf("return 0;}\n");
}
```

```
#include    "parser.h"      /* for types AST_node and Expression */
#include    "thread.h"      /* for Thread_AST() and Thread_start */
#include    "backend.h"     /* for self check */
                                        /* PRIVATE */
static AST_node *Active_node_pointer;

static void Trivial_code_generation(void) {
    printf("#include    \"stack.h\"\nint main(void) {\n");
    while (Active_node_pointer != 0) {
        /* there is only one node type, Expression: */
        Expression *expr = Active_node_pointer;
        switch (expr->type) {
        case 'D':
            printf("Push(%d);\n", expr->value);
            break;
        case 'P':
            printf("{\n\
                int e_left = Pop(); int e_right = Pop();\n\
                switch (%d) {\n\
                case '+': Push(e_left + e_right); break;\n\
                case '*': Push(e_left * e_right); break;\n\
                }}\n",
                expr->oper
            );
            break;
        }
        Active_node_pointer = Active_node_pointer->successor;
    }
    printf("printf(\"%%d\\n\", Pop()); /* print the result */\n");
    printf("return 0;}\n");
}
                                        /* PUBLIC */
void Process(AST_node *icode) {
    Thread_AST(icode); Active_node_pointer = Thread_start;
    Trivial_code_generation();
}
```

Figure 4.12 A trivial code generator for the demo compiler of Section 1.2.

Triviale Codegenerierung / Output

```
#include    "stack.h"
int main(void) {
Push(7);
Push(1);
Push(5);
{
    int e_left = Pop(); int e_right = Pop();
    switch (43) {
    case '+': Push(e_left + e_right); break;
    case '*': Push(e_left * e_right); break;
    }}
{
    int e_left = Pop(); int e_right = Pop();
    switch (42) {
    case '+': Push(e_left + e_right); break;
    case '*': Push(e_left * e_right); break;
    }}
printf("%d\n", Pop()); /* print the result */
return 0;}
```

Figure 4.13 Code for `(7*(1+5))` generated by the code generator of Figure 4.12.

Threaded Code

- Anmerkung: kein Zusammenhang mit *Threading* des abstrakten Syntaxbaums
- Idee: Zweiteilung des generierten Codes
  - vom Interpreter abgeleitete Bibliotheksroutinen (Abb. 4.15)
  - vom Quellprogramm abgeleitete Aufrufsequenz (Abb. 4.14)
- Vorteil: extreme Codekompression möglich
  - bei höchstens 256 Routinen: ein Byte pro Aufruf
  - Parameterkombinationen können ähnlich kodiert werden
  - Kodierung erfordert wieder minimalen Interpreter (!)
- Anwendungen
  - eingebettete Systeme
  - Prozesskontrollsysteme

Threaded Code / Beispiel

```
#include    "stack.h"
int main(void) {
Push(7);
Push(1);
Push(5);
{
    int e_left = Pop(); int e_right = Pop();
    switch (43) {
    case '+': Push(e_left + e_right); break;
    case '*': Push(e_left * e_right); break;
    }}
{
    int e_left = Pop(); int e_right = Pop();
    switch (42) {
    case '+': Push(e_left + e_right); break;
    case '*': Push(e_left * e_right); break;
    }}
printf("%d\n", Pop()); /* print the result */
return 0;}
```

Figure 4.13 Code for `(7*(1+5))` generated by the code generator of Figure 4.12.

```
int main(void) {
    Expression_D(7);
    Expression_D(1);
    Expression_D(5);
    Expression_P(43);   /* 43 = ASCII value of '+' */
    Expression_P(42);   /* 42 = ASCII value of '*' */
    Print();
    return 0;
}
```

Figure 4.14 Possible threaded code for `(7*(1+5))`.

```
#include    "stack.h"

void Expression_D(int digit) {
    Push(digit);
}

void Expression_P(int oper) {
    int e_left = Pop(); int e_right = Pop();
    switch (oper) {
    case '+': Push(e_left + e_right); break;
    case '*': Push(e_left * e_right); break;
    }
}

void Print(void) {
    printf("%d\n", Pop());
}
```

Figure 4.15 Routines for the threaded code for `(7*(1+5))`.

Verwendung von partieller Auswertung

- Idee: Zweiteilung des Interpreters
  - 1. Stufe: analysiert den Zwischencode (verschwindet durch partielle Ausw.)
  - 2. Stufe: enthält die eigentlichen Operationen des Programms
- Beispiel `switch`
  - Generator (Abb. 4.12), mit partieller Auswertung (Abb. 4.16)
  - generierter Code (Abb. 4.13), mit partieller Auswertung (Abb. 4.17)
- Visualisierung der Stufen im Generator:
  - 1. Stufe: run now (Abb. 4.18)
  - 2. Stufe: run later (Abb. 4.19)

Vgl.: Generator/Code, ohne/mit PA

```
case 'P':
    printf("{\n\
        int e_left = Pop(); int e_right = Pop();\n\
        switch (%d) {\n\
        case '+': Push(e_left + e_right); break;\n\
        case '*': Push(e_left * e_right); break;\n\
        }}\n",
        expr->oper
    );
    break;
```

aus Figure 4.12

```
Push(7);
Push(1);
Push(5);
{
    int e_left = Pop(); int e_right = Pop();
    switch (43) {
    case '+': Push(e_left + e_right); break;
    case '*': Push(e_left * e_right); break;
    }}
{
    int e_left = Pop(); int e_right = Pop();
    switch (42) {
    case '+': Push(e_left + e_right); break;
    case '*': Push(e_left * e_right); break;
    }}
```

aus Figure 4.13

```
case 'P':
    printf("{\n\
        int e_left = Pop(); int e_right = Pop();\n"
    );
    switch (expr->oper) {
    case '+': printf("Push(e_left + e_right);\n"); break;
    case '*': printf("Push(e_left * e_right);\n"); break;
    }
    printf("}\n");
    break;
```

Figure 4.16 Partial evaluation in a segment of the code generator.

```
#include    "stack.h"
int main(void) {
Push(7);
Push(1);
Push(5);
{int e_left = Pop(); int e_right = Pop(); Push(e_left + e_right);}
{int e_left = Pop(); int e_right = Pop(); Push(e_left * e_right);}
printf("%d\n", Pop()); /* print the result */
return 0;}
```

Figure 4.17 Code for `(7*(1+5))` generated by the code generator of Figure 4.16.

Stufen der partiellen Auswertung

```
case 'P':
    printf("{\n\
        int e_left = Pop(); int e_right = Pop();\n"
    );
    switch (expr->oper) {
    case '+': printf("Push(e_left + e_right);\n"); break;
    case '*': printf("Push(e_left * e_right);\n"); break;
    }
    printf("}\n");
    break;
```

Figure 4.18 Foreground (run-now) view of partially evaluating code.

```
case 'P':
    printf("{\n\
        int e_left = Pop(); int e_right = Pop();\n"
    );
    switch (expr->oper) {
    case '+': printf("Push(e_left + e_right);\n"); break;
    case '*': printf("Push(e_left * e_right);\n"); break;
    }
    printf("}\n");
    break;
```

Figure 4.19 Background (run-later) view of partially evaluating code.

Partielle Auswertung

- Herausforderung bei der partiellen Auswertung:

  sehr allgemeines Prinzip, wie kann man es automatisieren?

- Anwendungsgebiete:
  - Optimierungen im Compiler (`pow(x,3)→ x*x*x`)
  - automatische Programmgenerierung (siehe Projekt "Metaprogrammierung" des Lehrstuhls)
- Oft geht es auch einfacher:
  - Hauptproblem der partiellen Auswertung ist Analyse, was und wieweit ausgewertet werden soll (Nichttermination, Codeexplosion)
  - verwende "Generating Extension"
    - spezialisiert Code zielgerichtet, z.B.: `genpow("x",3)="x*x*x"`
    - triviale Codegenerierung: spezialisiere mit dem Quellprogramm (statisch)
    - Flexibilität (statisch↔dynamisch): *Generating Extension Generator*

Triviale Codegenerierung in der Übung

- OCaml: funktionale Sprache mit imperativen und Objekt-orientieren Features
- MetaOCaml: Annotationen (für Einteilung des Programms in Stufen)
  - Brackets schliessen Code ein, der später ausgeführt wird
  - Escape fügt in solchen Code anderen Code ein, der jetzt erzeugt wird (variable Wiederholung von erzeugtem Code durch Rekursion möglich)
  - Run übersetzt erzeugten Code (mit dem OCaml-Compiler) und führt ihn aus
- Anzahl der Stufen unbegrenzt, aber Programm-abhängig (Typinformation)
- triviale Codegenerierung ausschliesslich zur Laufzeit, ausführbares Programm enthält OCaml-Compiler
- Beim Einfügen der Stufen bleiben wichtige Teile der Semantik erhalten
  - Typbindung durch Typregeln für Annotationen
  - Lexical Scoping durch automatische Umbenennung lokaler Variablen

Effizienz des trivial generierten Codes

Programm: Berechnung der ersten 50 Primzahlen nach 1,000,000,000

Interpreter in bzw. Zielcode	Rechner	Pascal mit Übungscompiler interpretiert (s)	Pascal mit Übungscompiler triviale CG (s)	interpretiert / triviale CG	Programm in OCaml (s)	triviale CG / OCaml-Prg.
OCaml Bytecode	wagner	18.46	0.62	30	0.19	3.3
	neo	15.90	0.46	35	0.15	3.1
	trick	13.33	0.38	35	0.11	3.5
	ravel	13.16	0.42	31	0.13	3.2
Native Code erzeugt mit OCaml-Comp.	wagner	2.50	0.05	50	0.04	1.3
	neo	1.62	0.03	54	0.02	1.5
	trick	1.71	0.03	57	0.02	1.5
	ravel	1.59	0.03	53	0.02	1.5

- die Effizienz ist mglw. bereits ausreichend (hier nur ca. 30-50% schlechter gegenüber Implementierung in Standard-Programmiersprache)
- attraktive Lösung für das Rapid-Prototyping neuer Programmiersprachen