Vorlesungsplan

Einführung (1)
Werkzeuge (1-2)
- Cocktail, BEG, ELI, SUIF, Trimaran

Sequentielle Optimierungen (3-7)
- SSA Konstruktion
- Optimierungen auf SSA-Form:
  Operatorvereinfachung, Eliminierung gemeinsamer Teilausdrücke (CSE), Eliminierung partieller Redundanzen (PRE)
- Speicher SSA
- Globale kontextsensitive Wertanalyse auf SSA-Form

Cache Optimierung (7-10)
- Caches und ihre Problematik
- Techniken zur Cacheoptimierung (und zur Parallelisierung):
  Schleifenoptimierungen für Reihungen, Optimierungen für dynamische Datenstrukturen, Vorladen und Befehlsanordnung

Weitere Optimierungen (10-11)
- Nachoptimierung, Registerzuteilung, Befehlsanordnung

Nebenläufige Sprachen (11-12)
- Begriffe und Konzepte
- Parallele Hardware-Architekturen
- Implementierung von Parallelität

Inhalt – Werkzeuge

Werkzeuge

- Generelle Frage: WAS wird hier WIE spezifiziert?
- Cocktail
  - REX, ELL, LALR, Puma, AST, AG
  - BEG

Lernziele:
- Welche Werkzeuge sind für welche Aufgabe geeignet? (Schwerpunkt z..B. auf Optimierungen, Backend, ...)

Werkzeuge

Vorteile von Werkzeugen bei der Konstruktion eines Übersetzers im Vergleich zur Handprogrammierung:

- Aufwand bei Konstruktion eines Übersetzers wesentlich geringer
- Es wird eine viel kürzere Spezifikation entwickelt
- Verbesserung der Leistung
- Prüfen der Spezifikation auf Konsistenz bewirkt:
  - Reduktion der Anzahl möglicher Fehler
  - Erhöhung der Zuverlässigkeit des resultierenden Übersetzers

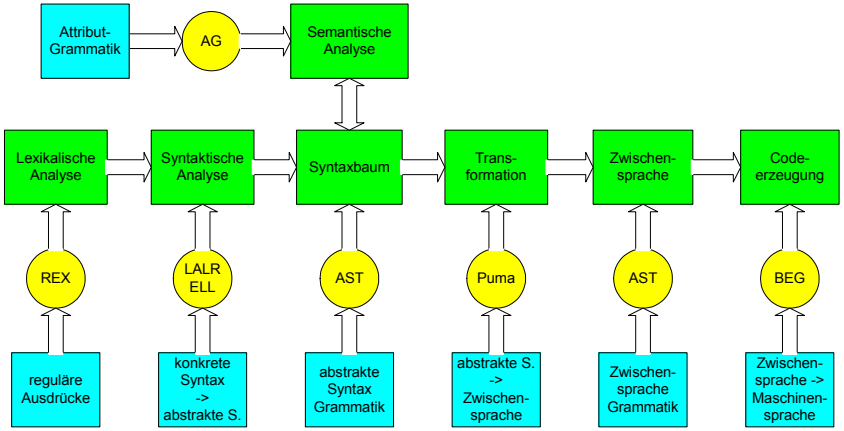
Entwurfsziele von Cocktail

- praktische Brauchbarkeit für realistische Programmiersprachen
- automatische Generierung von Übersetzern mit Produktionsqualität
- wesentliche Steigerung der Übersetzerbau-Produktivität
- mit Handprogrammierung vergleichbare Qualität der erzeugten Übersetzer

Werkzeuge von Cocktail

- REX Generator für lexikalische Analysatoren
- LALR LALR(1) Zerteilergenerator
- ELL LL(1) Zerteilergenerator
- AST Generator für abstrakte Syntaxbäume
- AG Generator für Attributauswerter
- Estra/Puma Transformation abstrakter Syntaxbäume
- BEG Generator für Codegeneratoren

Cocktail Deployment



Gemeinsamkeiten der Spezifikationen

- Die Zielsprache der Generatoren ist C oder Modula-2.
- Spezifikationen können Abschnitte mit Zielsprachanweisungen enthalten.
  - Die Erfahrung zeigt, dass während der Konstruktion realistischer Übersetzer eine Reihe kleiner Sonderprobleme auftritt, die nicht mit den Werkzeugen gelöst werden können.
  - Deswegen sind Schlupflöcher nötig, also Möglichkeiten, die es dem Werkzeugbenutzer erlauben, leicht handgeschriebene Teile einzufügen.
- Die Werkzeuge sind größtenteils von einander unabhängig.
  - Keines der erzeugten Module besitzt eine festgelegte Ausgabe.
  - Stattdessen wird die Ausgabe mittels Anweisungen der Zielsprache spezifiziert und kann somit beliebig gewählt werden.

Kommentarfolie

Kommentare zur vorangegangenen Folie:

- Beispiele für Schlupflöcher:
  - Fortran: keine rein syntaktische Zerlegung (vgl. Ü-Bau I)
  - Stringrepräsentation
  - Sonderzeichenbehandlung
  - Kommentare (wenn z.B geschachtelt oder länger als eine Zeile
- Aufgabenbereiche außerhalb des Übersetzerbaus:
  - Software-Engineering: z.B. automatische Bestimmung der Codequalität mit Metriken
  - XML-Verarbeitung, z.B. lazy evaluation
  - Graphikoptimierungen: z.B. Verdeckte Objekte nicht berechnen
  - Jahr-2000-Umstellung
  - Software-Sanierung im allgemeinen

REX

REX (regular expression tool) ist ein Generator für lexikalische Analysatoren

- Seine Spezifikationen basieren auf:
  - regulären Ausdrücken
  - semantischen Aktionen, die in C oder Modula-2 geschrieben werden.
- Erkennen der Eingabe des erzeugten lexikalischen Analysators:
  - Wenn eine einem regulären Ausdruck entsprechende Zeichenkette erkannt wurde, werden die zugehörigem Aktionen ausgeführt.
  - Falls zur eindeutigen Erkennung der Symbole der Kontext betrachtet werden muss, so kann der rechte Kontext durch einen zusätzlichen regulären Ausdruck spezifiziert werden, der linke Kontext wird mit Start-Zuständen behandelt.
  - Falls mehrere reguläre Ausdrücke auf die aktuelle Eingabe zutreffen, so wird der Ausdruck mit der längsten Zeichenkette bevorzugt.
  - Falls es immer noch mehrere Möglichkeiten gibt, so wird der zuerst in der Spezifikation stehende Ausdruck gewählt.

REX II

- Die erzeugten lexikalischen Analysatoren berechnen automatisch Zeile und Spalte in der Quelle für jedes erkannte Symbol und enthalten einen Mechanismus für Include-Dateien.
- Bezeichner und Schlüsselwörter können effizient in Groß- oder Kleinbuchstaben normalisiert werden.
- Es gibt Regeln, um Leerstellen, Tabulatoren und Zeilenwechsel zu überlesen.
- Die generierten lexikalischen Analysatoren sind tabellengesteuerte, deterministische endliche Automaten.
- Die Tabellen werden mit der sogenannten Kammvektortechnik komprimiert.

Kommentarfolie

Kammvektortechnik:

- Robert Endre Tarjan, Andrew Chi-Chih Yao:
  Storing a Sparse Table,
  Communications of the ACM, Volume 22, Number 11,
  November 1979
- Effizientes Speichern und Durchsuchen großer, spärlich besetzter Tabellen mit Methoden der linearen Algebra

LALR

Lalr ist ein LALR(1) Zerteiler-Generator der Grammatiken

- Seine Spezifikationen basieren auf:
  - erweiterter BNF Form
  - Die Grammatikregeln können mit semantischen Aktionen versehen werden, die direkt in einer Zielsprache formuliert sind.
- Erkennen der Eingabe läuft wie folgt:
  - Immer wenn eine Grammatikregel erkannt wird, wird die zugehörige semantische Aktion ausgeführt.
  - Der Generator stellt einen Mechanismus zur S-Attributierung zur Verfügung, d. h. synthetisierte Attribute können während der Zerteilung berechnet werden.
  - Im Falle von LR-Konflikten liefert Lalr die Mengen von Situationen bestehender Zustände und druckt einen Ableitungsbaum.
  - Konflikte können durch die Angabe von Priorität und Assoziativität für Operatoren und Produktionen gelöst werden.

LALR II

- Die generierten Zerteiler beinhalten eine automatische Fehlerbehandlung mit Fehlermeldungen und -reparatur. Zur Fehlerbehandlung wird die vollständige, rücksetzungsfreie Methode von Röhrich verwendet.
- Die Zerteiler sind tabellengesteuert und wie im Falle von Rex werden die Tabellen mit der Kammvektortechnik komprimiert.
- Die Eingabesprachen von Rex und Lalr sind hinsichtlich der Syntax gegenüber Lex und Yacc lesbarer gestaltet. Mit Hilfe zweier Präprozessoren können Rex und Lalr auch Eingaben für Lex und Yacc verarbeiten. Dadurch sind REX und LALR in Bezug auf die Benutzerschnittstelle kompatibel mit den UNIX-Werkzeugen.

ELL

ELL ist ein LL(1) Zerteiler-Generator:

- verarbeitet die gleiche Spezifikationssprache wie LARL.
  - Die Grammatiken müssen die LL(1)-Eigenschaft besitzen.
  - Während der Zerteilung kann eine L-Attributierung ausgewertet werden.
  - Die erzeugten Zerteiler beinhalten eine automatische Fehlerbehandlung mit Fehlermeldungen und -reparatur wie Lalr.
- Die Zerteiler sind nach dem Verfahren des rekursiven Abstiegs implementiert

AST

Ast ist ein Generator für abstrakte Syntaxbäume.

- Er generiert Programm-Module zur Bearbeitung attributierter Bäume.
- Den Knoten können beliebig viele Attribute von beliebigem Typ zugeordnet werden.
- Die Spezifikationen basieren auf erweiterten Baum-Grammatiken.
  - Sie können als gemeinsame Notation sowohl für konkrete und abstrakte Syntax als auch für attributierte Bäume und Graphen betrachtet werden.
  - Ein Erweiterungsmechanismus stellt einfache Vererbung zur Verfügung.
- Intern werden die Bäume durch verzeigerte Verbunde gespeichert.
- Operationen für Bäume und Graphen können von Ast erzeugt werden:
  - Konstruktoren kombinieren Aggregatschreibweise mit Speicherverwaltung.
  - Lese- und Schreibeprozeduren übertragen Graphen aus/in Dateien in lesbarem ASCII- oder internem Binärformat.
  - Die Reihenfolge von Teilbäumen in einer Liste kann umgekehrt werden.
  - Es werden Prozeduren für häufig benutzte Traversierungsstrategien wie top down oder bottom up zur Verfügung gestellt.
- Ein interaktiver Graph-Browser erlaubt die Inspektion von Graphen in lesbarer Weise und unterstützt so den Programmtest.

AG

Ag ist ein Generator für Attributauswerter.

- Er verarbeitet geordnete Attributgrammatiken (OAGs) und sogenannte higher-order Attributgrammatiken (HAGs).
- Basis ist die abstrakte Syntax; genauer: die von AST erzeugte Baummodule.
- Den Terminalen und Nichtterminalen können Attribute zugeordnet werden.
  - Diese werden mit den Typen der Zielsprache getypt.
  - Dabei sind auch baumwertige Attribute möglich.
  - Ag erlaubt regellokale Attribute und bietet einfache Vererbung für Attribute und Attributberechnungen an.
- Die Attributberechnungen werden in der Zielsprache formuliert und sollten in einem funktionalen Stil gehalten sein.
  - Es ist möglich externe Funktionen aufzurufen.
  - Nicht-funktionale Anweisungen sind möglich, aber gefährlich.
- Eine Attributgrammatik kann aus mehreren Modulen bestehen, wobei die kontextfreie Grammatik nur einmal spezifiziert wird.
- Es gibt Kurzschreibweisen für Kopierregeln und gefädelte Attribute, womit viele triviale Attribut-Berechnungen weggelassen werden können.

Kommentarfolie

Higher-order Attributgrammatik:

- Attribut kann Verweis auf einen Knoten im AST sein
- d.h. Baumwertige Attribute sind erlaubt

Puma / Estra

Estra ist ein Generator für Transformatoren von abstrakten Syntaxbäumen.

- Die erzeugten Transformations-Module haben als Eingabe einen attributierten Baum und bilden diesen auf eine Ausgabe beliebiger Art ab.
  - Die Ausgabe kann ein neuer Baum sein, eine lineare Zwischensprache ein Quellprogramm z. B. in Java oder eine Folge von Prozeduraufrufen.
  - In jedem Fall bleibt der Eingabebaum unverändert, um das Problem der Reattributierung aus Konsistenzgründen zu umgehen.
  - Teilbäume des Eingabebaums können zur Konstruktion eines Ausgabebaums wiederverwendet werden.
- Die beabsichtigten Anwendungsgebiete für die Transformationen sind die
  - Erzeugung von Zwischensprachen aus abstrakten Syntaxbäumen
  - Optimierer für interne Baumstrukturen jeden Niveaus.
- Estra arbeitet mit dem Werkzeug Ast zusammen, indem die Eingabebäume mittels AST erzeugten Modulen erstellt werden können.

Puma / Estra II

- Die Spezifikation einer Transformation ist regelbasiert; Eine Regel besteht aus einem Muster, das ein Baumfragment beschreibt, und einer Aktion.
  - Aktionen bestehen aus Anweisungen der Zielsprache.
  - Es können mehrere Transformationen spezifiziert werden.
  - Teilbäume können in beliebiger Reihenfolge transformiert werden.
  - Teilbäume können mehrmals mit der selben oder mit verschiedenen Transformationen bearbeitet werden.
  - Die Aktionen haben Lesezugriff auf die Attribute des Eingabebaums.
  - Zusätzliche abgeleitete oder vererbte Attribute können während der Transformation berechnet werden.
  - Die Anwendung von Regeln lässt sich durch Bedingungen einschränken.
  - Mehrdeutigkeiten werden mittels Kostenangaben aufgelöst.
- Zum Mustervergleich können zwei Implementierungen gewählt werden:
  - Ein direkt codierter dynamischer Programmierungs-Algorithmus.
  - Tabellen-gesteuerter Baummuster-Vergleicher.

BEG - Codeauswahl

Beg (back end generator) erzeugt Module zur Codeauswahl und zur Registerzuteilung.

- Codeauswahl wird mittels Baummustervergleich durchgeführt.
  - Maschinenbefehle werden mit Regeln beschrieben, die Baummuster enthalten.
- Der erzeugte Codegenerator hat als Eingabe eine baumförmige Zwischensprache.
  - Ein Eingabebaum wird abgebildet durch die Überdeckung des Baums mit Mustern und der anschließenden Ausgabe der zugehörigen Maschinenbefehle.
  - Die Regeln sind mit Kosten versehen, wodurch der Codegenerator eine Überdeckung mit Regeln mit minimalen Kosten auswählen kann.
- Der Benutzer beschreibt auf eventuell mehrdeutige Art und Weise die Abbildung bestimmter Konstrukte der Zwischensprache.
  - Es ist günstig, bei der Entwicklung einer Codegenerator-Beschreibung erst einen Teil der Maschinenbefehle zu spezifizieren, der groß genug ist, um die ganze Sprache zu übersetzen. Dies führt zu einem funktionsfähigen Übersetzer, welcher eventuell ineffizienten Code erzeugt.
  - Später können nach und nach weitere Regeln hinzugefügt werden, was schließlich zu einem Übersetzer führt, der guten Code erzeugt.

BEG II

- Beg implementiert die Bestimmung einer Überdeckung mit minimalen Kosten durch eine direkt codierten Version des dynamischen Programmierungs-Algorithmus.

- Die Generierung eines Registerzuteilers ist von besonderer Bedeutung, da hier Handprogrammierung ziemlich schwer und lästig ist und weil Fehler in der Registerzuteilung schwer zu finden sind.
- Innerhalb der Regeln können die Eigenschaften eines Maschinenbefehls hinsichtlich der Registerzuteilung spezifiziert werden:
  - die erlaubten Register für jeden Operanden
  - die durch Seiteneffekte veränderten Register
  - ob es sich um einen Zweiadressbefehl handelt oder nicht.
- Zusätzlich wird der Registersatz der Zielmaschine beschrieben.
- Zwei Arten von Registerzuteilung sind möglich:
  - Die on the fly Registerzuteilung kann nur einfache Registersätze behandeln. Sie ist jedoch sehr effizient und liefert oft zufriedenstellende Ergebisse.
  - In manchen Fällen ist der allgemeine Registerzuteiler nötig, welcher eine Art von Vorschau durchführt und deshalb einen zusätzlichen Pass benötigt.

Vergleich verschiedener Zerteilergeneratoren

	Bison	yacc	PGS	Lalr	Ell
Grammatik Spezifiziert in	LALR(1) BNF	LALR(1) BNF	LALR(1) EBNF	LALR(1) EBNF	LL(1) EBNF
Geschwindigkeit in [10^3 Symbol / Sekunde]	8.93	15.94	17.32	34.94	54.64
Geschwindigkeit in [10^3 Zeilen / Minute]	150	270	290	580	910
Tabellengröße in [bytes] (komprimiert)	7724	9968	9832	9620	-
Zerteilergröße in [bytes]	10900	12200	14140	16492	18048

Eingabe: Modula-2 Code.
Hardware: PCS Cadmus mit MC68020 Prozessor (16.7 MHz).
Quelle: J. Grosch: "Lalr - A Generator for Efficient Parsers." GMD-Bericht 1988.

Studienarbeit

- Thema:
  Leistungsvergleich verschiedener Zerteilergeneratoren
- Messungen auf vorangegangener Folie auf aktuellen Prozessoren für aktuelle Programmiersprachen durchführen
- Dabei auch neuere Zerteilergeneratoren testen:
  - Cup (LR)
  - JavaCC (LL)
  - ANTLR (LL System in Java, früher C, mit beliebiger, aber statisch fixierter Vorausschau)

Beispiel - REX

Im Folgenden einige fragmentarische Beispiele für Spezifikationen in den Cocktail Sprachen

- REX:

```
#STD# digit + "." digit + (E {+\-}? digit +) ?
   : {GetWord(Word);
      Attribute.IntegerConst.Integer := StringToInt(Word);
      RETURN RealConst;}
```

Beispiel - LALR / ELL

```
Stat       = <
   Assign  = Adr ':=' Expr .
   Call0   = Ident .
   Call    = Ident '(' Actuals ')' .
   If      = 'IF' Expr 'THEN' Then: Stats 'ELSE' Else: Stats 'END' .
   Read    = 'READ' '(' Adr ')' .
>.

...

Assign = {Tree := mAssign (NoTree, Adr:Tree, Expr:Tree,
                           ':=':Position);} .
Call0  = {Tree := mCall (NoTree, mNoActual (Ident:Position),
                         Ident:Ident, Ident:Position);} .
Call   = {Tree := mCall (NoTree, ReverseTree (Actuals:Tree),
                         Ident:Ident, Ident:Position);} .
If     = {Tree := mIf (NoTree, Expr:Tree, ReverseTree (Then:Tree),
                       ReverseTree (Else:Tree));} .
Read   = {Tree := mRead (NoTree, Adr:Tree);} .
```


Beispiel - AST / AG

```
Stats = <
   NoStat = .
   Stat = Next: Stats REV <
      Assign = Adr Expr [Pos: tPosition] .
      Call = Actuals [Ident: tIdent] [Pos: tPosition] .
      If = Expr Then: Stats Else: Stats .
      While = Expr Stats .
      Read = Adr .
      Write = Expr .
   >.
>.
...
If   = { Expr: Co := Coerce (Expr:Type, Reduce (Expr:Type)) ; } .
If   = { Label1 := Else: CodeSizeIn ;
         Label2 := Else: CodeSizeOut ; }
Call = { Object := Identify (Ident, Env) ; } .
Call = { Actuals: Formals := GetFormals (Object) ;
         => { CheckParams (Actuals, Actuals:Formals); } ; } .
Call = { CHECK Object^.Kind # NoObject
            ==> Error ("identifier not declared" , Pos) ;
         CHECK IsObjectKind (Object, Proc)
            ==> Error ("only procedures can be called" , Pos) ; } .
```


Beispiel - Puma

```
Assign (Next, Adr, Expr, _) ?
   Code (Adr); Code (Adr::Co);
   Code (Expr); Code (Expr::Co);
   Emit (STI, 0, 0);
   Code (Next);
.

If (Next, Expr, Then, Else, Label1, Label2) ?
   Code (Expr); Code (Expr::Co);
   Emit (FJP, Label1, 0);
   Code (Then);
   Emit (JMP, Label2, 0);
   Code (Else);
   Code (Next);
.
```


Beispiel - BEG

```
NONTERMINALS Value;
OPERATORS
  Constant    ( v : INTEGER )                -> Value;
  Plus                        Value + Value  -> Value;
  AddressPlus                 Value * Value  -> Value;
  BlockBase                                  -> Value;
  Content                     Value          -> Value;
  Assign                      Value * Value;

REGISTERS
  R0,R1,R2,R3,R4,R5,R6,R7,R8,R9,R10,R11,R12,R13,R14,R15;

NONTERMINALS
  Register REGISTERS (R0,R1,R2,R3,R4,R5,R6,R7,R8,R9,R10,R11,R12)
```

Beispiel - BEG II

```
RULE  Plus
        Content
          AddressPlus
            BlockBase
            Constant
        Register.r -> Register;
  COST 4; TARGET r;
  EMIT { WriteString (' A ');
         WrRegister (r.register); Write (',');
         WriteInt (Constant.v,1);
         WriteString ("(11)");
         WriteLn };

RULE  Constant -> Register;
  CONDITION { (Constant.v>=0) AND (Constant.v<=4095) }
  COST 3;
  EMIT { WriteString(' LA '); WrRegister (Register.register);
         Write(",",); WriteInt (Constant.v,1); WriteLn };
```


Beispiel - BEG III

```
RULE  Assign
        AddressPlus
          BlockBase
          Constant
        Register.r;
  COST 4;
  EMIT { WriteString (' ST ');
         WrRegister (r.register); Write (',');
         WriteInt (Constant.v,1);
         WriteString ("(11)");
         WriteLn };
```