# 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

- ELI
  - Eigenschaften und Idee
  - Spezifikationen
    - Dateitypen für Anwender
    - Beispiele
    - Problemgetrieben (Ein- / Ausgabe für Werkzeuge)
  - Technik hinter ELI
    - Odin
    - LIDO
- SUIF
- Trimaran

# Idee / Eigenschaften

ELI ist eine aufgabenspezifische Programmierumgebung zur Generierung von vollständigen Sprach-Implementierungen.

Mit besonderem Augenmerk auf:

- Problemorientierung statt Werkzeugorientierung
  (Dies ist für den eher UNIX-Werkzeug orientierten Markt ungewöhnlich)
- Ein breites Spektrum an Aufgaben und Lösungen
- Geringes Vorwissen sollte ausreichen
- Integriertes System mit flexiblen Anpassungen
- Neuübersetzung auch teilweise (geringere Umlaufzeit)

# Anwendungsgebiete

- Syntaktische Analyse
  Die kontextfreie Struktur eines Eingabetextes feststellen
- Lexikalische Analyse
  Zeichenfolgen erkennen
- Attributierung
  Die Berechnung von Werten in Baumstrukturen
- Eigenschaften abspeichern
  Das Warten und Speichern von Entitäten
- Generierung von Text
  Produzieren von strukturierten Ausgaben

# Vorgehensmodell I

- Ein einfaches Übersetzermodell [Waite-Goos, Compiler Construction] definiert einen Satz von Teilproblemen, der zur Lösung des Gesamtproblems führt.
- Lösungen werden als Spezifikationen beschrieben;
  sie können entweder neu gemacht oder aus Bibliotheken entnommen werden.
- Der tatsächliche Übersetzer wird dann von einem Konfigurationssystem zusammengebaut, das den Werkzeugeinsatz versteckt.

- ELI setzt folgendes Übersetzermodell voraus:
  - Struktur-Phase
    - Lexikalische Analyse
    - Syntaktische Analyse
  - Übersetzungs-Phase
    - Semantische Analyse
    - Transformation
  - Codierungs-Phase
    - Code Generierung
    - Assemblierung

# Vorgehensmodell II

- Im Übersetzermodell von ELI werden
  4 Klassen von Datenobjekten unterschieden.
  - Quellprogramm
  - konkreter Syntaxbaum
  - Zielprogramm Baum (Zwischendarstellung)
  - Maschinenbefehle (Zielcode)
- Die Spezifikationen beschreiben diese Strukturen
  und die Beziehungen dazwischen.

# Benutzerinteraktionen / Produkte

- Interaktive zeilenorientierte Benutzerschnittstelle (ähnlich einer UNIX-shell)
- Das ELI System erlaubt es dem Benutzer, verschiedene Produkte aus den Spezifikationen abzuleiten
- Ein Produkt ist eine:
  - UNIX-Datei
  - Menge UNIX-Dateien in einem Verzeichnis
  - Liste von Dateinamen
- Ein Produkt wird angefragt (und erstellt) durch Eingabe einer entsprechenden Zeile in die Benutzerschnittstelle

# Benutzerinteraktion - Beispiele

- `Minilax.specs :parsable`
  Prüfe, ob die konkrete Syntax den Erfordernissen des Zerteilergenerators genügt.
- `Minilax.specs +arg=(test.mla) :stdout`
  Rufe den generierten Übersetzer auf, übergebe als Argument `test.mla` und zeige die Compilerausgabe an.
- `Minilax.specs +arg=(test.mla) :stdout :err`
  Wie oben, zeigt aber nur Fehler an (stderr).
- `Minilax.specs :exe > Minilax.exe`
  Erstelle eine ausführbare Version des Übersetzers als `Minilax.exe`.
- `Minilax.specs :source > src`
  Erzeuge alle Quellen für den Minilax-Übersetzer in dem Verzeichniss `src`.

# Spezifikationen

- ELI verarbeitet diese Eingaben bzw. Spezifikationen:
  - .specs Sammlung von Teilproblembeschreibungen
  - .gla Beschreibung der Symbolstruktur
  - .con Beschreibung der Syntax („Phrasenstruktur")
  - .lido Beschreibung der Struktur eines Baumes und der Berechnungen, die auf dem Baum ausgeführt werden sollen
  - .map Beschreibung der Abbildung zwischen Zerteilung und AST „tree grammar"
  - .ctl Optionen für die Auswertergenerierung
  - .h, .c Module für benutzerdefinierte Funktionen, Variable, Typen
  - .ptg Beschreibung von strukturiertem Ausgabetext
  - .pdl Sprache zur Beschreibung der Eigenschaften von Entitäten
  - .oil Beschreibung zur Überladung von Operatoren
  - .cpl Beschreibung der Kommandozeilen-Optionen der generierten Übersetzer
  - .fw Stellt eine Sammlung von zusammengehörigen Spezifikationen samt Dokumentation ihrer Abhängigkeiten dar
  - .delit Ordnet Terminalen in .con Dateien eine Spezialsemantik zu
  - .gnrc Definiert ein generisches Spezifikationsmodul

# .specs

Sammlung von Teilproblembeschreibungen (.specs)

- Die Spezifikationen für ein Projekt werden hier angegeben
- Benutzte Module und generische Instantiierungen werden bekannt gegeben
- ```
  IdemClass.fw /* Name analysis and symbol classification */
  TreeLido.fw  /* Generation of symbol computations */
  TreePtg.fw   /* Generation of rule templates */
  IdemTree.fw  /* Definition of the input language */

  idn.c   /* version of idn.c that includes chgidn function */
  ```
  ```

# .gla

Beschreibung der Symbolstruktur

- Es wird angegeben, ob und welche Berechnung ausgeführt werden soll, wenn dieses Symbol erkannt wird.
- ELI generiert aus dieser Spezifikation einen Symbolentschlüssler (scanner).
- ```
  ident : C_IDENTIFIER
  string: $' (auxPascalString)   [mkstr]
  numb  : $[0-9]                 [mkstr]
  ```
  ```

# .con

Beschreibung der Syntax („Phrasenstruktur")

- Jeder Grammatik-Regel können Berechnungen zugeordnet sein, die bei Anwendung der Regel ausgeführt werden.
- ELI generiert aus dieser Spezifikation einen Zerteiler.
- ```
  def: set_name '=' '{' body '}' .
  body: element+ .
  cond: 'if' exp 'then' stmt $'else' .
  ```
  ```

# .lido

Beschreibung der Struktur eines Baumes und der Berechnungen,
die auf dem Baum ausgeführt werden sollen

- ELI generiert hieraus einen baumtraversierenden Auswerter.
- `ATTR Sym: int;`

```
SYMBOL set_name INHERITS Entity END;
SYMBOL test COMPUTE
  PTGOut( PTGTable( CONSTITUENTS set_name.Sym
                    WITH (int, ADD, ONE, ZERO) ));
END;

RULE r_wall: wallspec ::= 'wall' pos ';'
COMPUTE
  wallspec.done = setwall(pos.x, pos.y);
END;
```

# .map

Beschreibung der Abbildung zwischen Zerteilung und AST („tree grammar")

- ELI benutzt diese Spezifikation, um die Baumaufbauoperationen festzulegen, die an die Zerteilergrammatik angeheftet werden.
- Es hilft ELI, den Übergang von der konkreten zur abstrakten Syntax nach den Wünschen des Benutzers zu gestalten;
  dabei können Symbole, Regeln und Regelketten mit einem Befehl abgebildet werden.
- ```
  MAPSYM
    Expression ::= Term Factor .
  ```
  ```

# .ctl

Optionen für die Auswertergenerierung

- ELI benutzt diese Spezifikationen, um das Verhalten bei der Generierung von Routinen anzupassen, die Auswertungen durchführen.
- Hier die Syntax der .ctl-Dateien:

```
Options   ::=   ( Option ';' )*
Option    ::=   'EXPAND'   ':' ExpandOpts /
                'ORDER'    ':' OrderOpts /
                'OPTIM'    ':' OptimOpts

ExpandOpts  ::=  ExpandOpt   ',' ExpandOpts / ExpandOpt
OrderOpts   ::=  OrderOpt    ',' OrderOpts / OrderOpt
OptimOpts   ::=  OptimOpt    ',' OptimOpts / OptimOpt

Expandopt   ::= 'INFO' / 'INCLUDINGS_SEPARATE' /
                'INCLUDING' 'ON' / 'INCLUDING' 'OFF'
OrderOpt    ::= 'PARTITION' Strategy / 'TOPOLOGICAL' Strategy /
                'GRAPH' Type ( ident ) * /
                'ARRANGE' arrangePart
Strategy    ::= 'EARLY' / 'LATE'
```

# .ctl II

```
Type          ::= 'DIRECT_SYMBOL' / 'TRANSITIVE_SYMBOL' /
                  'INDUCED_SYMBOL' / 'DIRECT_RULE' /
                  'TRANSITIVE_RULE' / 'INDUCED_RULE' /
                'PARTITIONED_RULE' / 'PARTITION' / 'VISIT_SEQUENCE'

arrangePart ::= 'AUTOMATICALLY' / 'FAST' /
                  'FOR' 'SYMB' ident 'EVAL' ident 'BEFORE' ident /
            'IN' 'RULE' ident 'EVAL' ident '[' intval ']' '.' ident
                  'BEFORE' ident '[' intval ']' '.' ident

OptimOpt      ::= 'OFF' / 'INFO' / 'MORE_GLOBALS' /
                  'NO_VARIABLES' / 'NO_STACKS' /
                  'GROUPING' 'VARIABLE' /
                'ATTRSPEZ' Type ( symbname '[' ( ident ) + ']' ) *

symbname      ::= ident / 'ANYSYMBOL'

Type          ::= 'GLOBAL VAR' / 'GLOBAL STACK' / 'TREE_NODE' /
                  'GROUP VAR' / 'GROUP STACK'
```

# .ptg

Beschreibung von strukturiertem Ausgabetext

- ELI generiert hieraus eine Menge von Ausgabefunktionen.
- ```
  Seq: $ $
  List: $ ",\n\t" $
  ```
  ```

# .pdl

Sprache zur Beschreibung der Eigenschaften von Entitäten

- ELI generiert aus dieser Spezifikation ein Definitionstabellen-Modul.
- ```
  Code : mytype; "kcode.h"
  size : int;
  ```
  ```

# .oil

Beschreibung des Überladens von Operatoren

- ELI generiert hieraus ein Modul zur Operationsauswahl mit Typanpassung.
- Dieses Verfahren kann auch zur Codeerzeugung verwendet werden.
- ```
  OPER iAdd(integer, integer): integer;
  OPER rAdd(real, real): real;

  INDICATION Plus: iAdd, rAdd, sUnion;

  COERCION Float(integer): real;
  ```
  ```

# .cpl

Beschreibung der Kommandozeilen-Optionen der generierten Übersetzer

- ELI generiert ein Modul, das die Kommandozeile verarbeitet und zugreifbar macht.
- ```
  Speed "-s" int
  "-s determines steps per second";
  ```
  ```

# .fw

Stellt eine Sammlung von zusammengehörigen Spezifikationen
samt Dokumentation ihrer Abhängigkeiten dar

- ELI spaltet die Spezifikationen verschiedenen Typs auf und verarbeitet sie gemäß ihres Typs weiter.
- Aus Dateien dieses Typs kann automatisch eine druckbare oder html-Dokumentation erstellt werden.
- Das Vorgehen ist aus dem „literate programming" von D.E. Knuth abgeleitet.
- ```
  @0@<c.ptg@>@{
  Seq: $ $
  @}
  @0@<c.lido@>@{
  SYMBOL Entity INHERITS IdPtg END;
  @}
  ```
  ```

# .delit

Ordnet Literalen in .con Dateien eine Spezialsemantik zu

- Jede Zeile besteht aus einem regulären Ausdruck gefolgt von einem optionalen Bezeichner.
- Die regulären Ausdrücke definieren das besonders zu behandelnde Literal.
- In den Spezifikationen ist dann der Bezeichner synonym zu dem regulären Ausdruck verwendbar.

# .gnrc

Definiert ein generisches Spezifikations-Modul

- Generische Module können instantiiert werden,
  um eine Sammlung von Spezifikationen zu erhalten,
  die spezifische Probleme lösen.

# Mini-Beispiel I

```
Position:     Latitude Longitude .
Latitude:     NS Coordinate .
Longitude:    EW Coordinate .
NS:           'N' / 'S' .
EW:           'E' / 'W' .
Coordinate:   Integer Float .
```

---

```
Integer:    $[0-9]+                 [mkstr]
Float:      $[0-9]+"."[0-9]+        [mkstr]
```

---

```
RULE DegMin: Coordinate ::= Integer Float
COMPUTE
  Coordinate.minutes=Minutes(Integer, Float)
END;
```

# Mini-Beispiel II

bsp.specs
```
    bsp.gla
    bsp.con
    bsp.liga
```

bsp.con:
```
    Position:     Latitude Longitude .
    Latitude:     NS Coordinate .
    Longitude:    EW Coordinate .
    NS:              'N' / 'S' .
    EW:              'E' / 'W' .
    Coordinate:      Integer Float .
```

bsp.gla:
```
    Integer:      $[0-9]+                  [mkstr]
    Float:        $[0-9]+"."[0-9]+         [mkstr]
```

bsp.liga
```
    RULE DegMin: Coordinate ::= Integer Float
    COMPUTE
      Coordinate.minutes=Minutes(Integer, Float)
    END;
```

# Mini-Beispiel III

ELI starten und Übersetzer generieren:

```
eli
Eli Version 4.3 (? for help, ^D to exit) (local: type ? for help)
-> bsp.specs :exe > bsp.exe
** Copied up-to-date value into: /ben/rubino/work/bsp.exe
->
```

Übersetzer außerhalb von ELI testen:

```
i44pc20:~/work-> ./bsp.exe
N 45 5.6 W 45 4.56
44pc20:~/work->
```

# Inhalt – Werkzeuge

- ELI
  - Eigenschaften und Idee
  - Spezifikationen
    - Dateitypen für Anwender
    - Beispiele
    - Problemgetrieben (Ein- / Ausgabe für Werkzeuge)
  - Technik hinter ELI
    - Odin
    - LIDO
- SUIF
- Trimaran

# Erstellung von Produkten - Odin

- Das ELI System führt beliebige Verarbeitungen aus, um
  - die Ausgaben eines Werkzeugs
    als Eingaben eines anderen Werkzeugs anzupassen
  - um konsistente Eingaben für mehrere Werkzeuge
    aus einer Spezifikation zu erstellen
  - um die Ausgaben mehrerer Werkzeuge zu kombinieren
- Diese Verarbeitungen werden angestoßen, wenn aus einem Quellobjekt ein „abgeleitetes Objekt" produziert werden muß

# Quellobjekte ⇒ abgeleitete Objekte

- Quellobjekte können direkt vom Benutzer erstellt oder geändert werden. Sie können normale Dateien, Verzeichnisse oder symbolische Verweise sein. Quellobjekte werden nicht von Eli automatisch neu erstellt; sie sind die Grundbausteine, aus denen Eli alle weiteren Objekte ableitet.
- Jedem Quellobjekt wird ein Typ von Eli gegeben, der auf seiner Dateinamenserweiterung basiert. Dieser Typ legt fest, welche Objekte aus dem Quellobjekt produziert werden können.
- Abgeleitete Objekte sind Objekte, die aus Quellobjekten und anderen abgeleiteten Objekten durch den Aufruf von einem oder mehreren Werkzeugen produziert werden können.
- Werkzeuge werden nur aufgerufen, wenn es nötig ist, um ein spezifiziertes abgeleitetes Objekt zu erstellen. Eli speichert automatisch berechnete Objekte für Wiederverwendung in den zukünftigen Ableitungen zwischen. Abgeleitete Objekte werden nur von Eli selbst, nicht von den Benutzern erstellt und geändert.
- Odin [von G. Clemm] übernimmt die hier beschriebenen Steueraufgaben.

# Odin

Odin ist eine Weiterentwicklung von make
(„Konfigurationssystem der zweiten Generation")

- Odin berechnet vollständige Abhängigkeitsinformationen, was die „build scripts" kürzer und einfacher zu handhaben macht.
- Die Abhängigkeiten werden stückweise spezifiziert.
  - Odin baut daraus rückwärts (vom zu erzeugenden Endprodukt her) den jeweiligen Abhängigkeitsgraphen auf.
  - Zur Erzeugung des Endprodukts werden dann die einzelnen Werkzeuge vorwärts (von den Quellen her) aufgerufen.
- Odin erzielt seine Leistungsfähigkeit:
  - indem es sich den Projektstatus in einem Puffer (cache) merkt,
  - und dadurch die meisten Dateisystem-Statusabfragen beseitigt,
  - durch paralleles Generieren im verteilten System.

# LIDO / LIGA

- LIDO wird benötigt, um Berechnungen zu spezifizieren, die im AST ausgeführt werden. Formale Grundlage: OAGs.
- Mit LIDO kann beschrieben werden:
  - welche Berechnungen ausgeführt werden müssen,
  - wie der Kontext zu berücksichtigen ist,
  - wie die Berechnungen voneinander abhängen,
  - und welche Werte wie propagiert werden.
- Die Berechnungs-Funktionen und Typen werden in C implementiert.
- LIGA verarbeitet diese Beschreibungen.
  - Ergebnis ist ein C-Modul.
  - Es wird automatisch eine Baumdurchlaufstrategie gefunden, welche die geforderten Abhängigkeiten erfüllt.
  - Verwendet wird ein erweiterter OAG-Algorithmus.

# Beispiel

```
RULE p: Stmt ::= 'while' Expr 'do' Stmt COMPUTE
   Expr.postType = boolType
END;

SYMBOL Expr COMPUTE
   Compatible(THIS.preType, THIS.postType)
END;

ATTR preType, postType: DefTableKey;
```

# Besonderheiten I

- Es können Attributierungsregeln für Symbole und Regeln angegeben werden.
  Werden dabei die gleichen Attribute spezifiziert,
  kommen diese in den Regeln zum Tragen.
- Mittels `BOTTOMUP` ist es möglich anzugeben, dass die Berechnungen während des Baumaufbaus stattfinden sollen.
- An Ausdrücken können zusätzliche Abhängigkeiten angezeigt werden,
  die zum Beispiel aus Seiteneffekten herrühren, denn Abhängigkeiten, die durch Seiteneffekte in C-Aufrufen entstehen, werden nicht automatisch erkannt.

```
GetProp(UseId.Key, 0) <- UseId.PropIsSet
printf("%s ", Opr.String) <- (Expr[2].printed, Expr[3].printed)
```

- Es können gemeinsame Berechnungen ausfaktorisiert werden.
  - Mehrfach-Vererbung
  - Bei Konflikten ist die jüngere Deklaration eines Attributs die gültige.

# Besonderheiten II

- Auf Attribute im Baum kann man über eine längere Distanz zugreifen.
  - `INCLUDING` erstellt einen Zugriff zu weiter oben liegenden Attributen.
  - `CONSTITUENT(S)` erstellt einen Zugriff zu weiter unten liegenden Attributen.
  - So können Werte weitergegeben oder Abhängigkeiten geschaffen werden.
  - Dadurch kann von der speziellen Baumstruktur abstrahiert werden, was zu generelleren Berechnungsmustern führt.
    $\Rightarrow$ Wiederverwendbare Spezifikationen
  - Durch Kurzschließen der Attributwerte werden zahlreiche Kopien im Baum überflüssig.
- Der initiale Baum kann mit vorberechneten Unterbäumen angereichert werden
  - Baume können an vordefinierten Stellen eingehängt werden, dies geschieht auch beim ganz normalen Baumaufbau.
- Man kann zyklische Abhängigkeiten angeben, die solange iteriert werden, bis eine bestimmte Bedingung zutrifft.

# Attributspeicherung

Ort der Attributspeicherung (Wdh. aus ÜBau I):

1. Explizit im AST → teuer, immer möglich
2. Im Prozedurkeller → effizienter als (1.) , aber nicht immer möglich
3. In separaten Attributkeller → u.U. effizienter als (2.) ,
   immer möglich wenn (2.) möglich
4. Als globale Variable → effizienter als (3.) , aber selten möglich
5. Durch globale Tabelle → praktisch das wichtigste Verfahren,
   aber nicht systematisch aus AG ableitbar

- ELI (LIGA) beherrscht die Arten 1. – 4. der Attributspeicherung
  sowohl automatisch als auch mit manuellen Hinweisen.
- Die Speicherung in der Definitionstabelle (5.) wird durch spezielle Anweisungen
  des Benutzers erreicht.

# Optionen für die AG-Auswertergenerierung I

- Die nachfolgenden Optionen werden von dem AG-Auswertergenerator in der Regel nur als Hinweis aufgefaßt und ignoriert, falls die Korrektheit verletzt wird.

- ```
  Options   ::=   ( Option ';' )*
  Option    ::=   'EXPAND'   ':' ExpandOpts /
                  'ORDER'    ':' OrderOpts /
                  'OPTIM'    ':' OptimOpts

  ExpandOpts  ::=  ExpandOpt   ',' ExpandOpts / ExpandOpt
  OrderOpts   ::=  OrderOpt    ',' OrderOpts / OrderOpt
  OptimOpts   ::=  OptimOpt    ',' OptimOpts / OptimOpt
  ```

- ```
  Expandopt   ::= 'INFO' / 'INCLUDINGS_SEPARATE' /
                  'INCLUDING' 'ON' / 'INCLUDING' 'OFF'
  ```
  Beeinflußt den Zugriff auf entfernte Attribute.
  Wird vor allem für die Fehlerkorrektur benötigt.

# Optionen für die AG-Auswertergenerierung II

- `OrderOpt ::= 'PARTITION' Strategy / 'TOPOLOGICAL' Strategy / 'GRAPH' Type ( ident )* / 'ARRANGE' arrangePart`
- `Strategy ::= 'EARLY' / 'LATE'`

  Gibt die Strategie zur Berechnung der Attributpartitionen vor.
- `Type ::= 'DIRECT_SYMBOL' / 'TRANSITIVE_SYMBOL' / 'INDUCED_SYMBOL' / 'DIRECT_RULE' / 'TRANSITIVE_RULE' / 'INDUCED_RULE' / 'PARTITIONED_RULE' / 'PARTITION' / 'VISIT_SEQUENCE'`

  Mit `GRAPH` kann der Abhängigkeitsgraph für Symbole und Regeln unter verschiedenen Aspekten angegeben werden.
- `arrangePart ::= 'AUTOMATICALLY' / 'FAST' / 'FOR' 'SYMB' ident 'EVAL' ident 'BEFORE' ident / 'IN' 'RULE' ident 'EVAL' ident '[' intval ']' '.' ident 'BEFORE' ident '[' intval ']' '.' ident`

  Legt die Auswertungsreihenfolge von Symbolen oder Regeln fest. Dies ist zur Beschleunigung verwendbar. Im Fall einer Nicht-OAG, die aber wohldefiniert ist, kann diese erst dadurch überhaupt verarbeitet werden.

# Optionen für die AG-Auswertergenerierung III

```
OptimOpt    ::= 'OFF' / 'INFO' / 'MORE_GLOBALS' /
                'NO_VARIABLES' / 'NO_STACKS' /
                'GROUPING' 'VARIABLE' /
                'ATTRSPEZ' Type ( symbname '[' ( ident )+ ']' )*
```

Mit diesen Optionen kann der optimierende Attributanordner beeinflußt werden.

- `OFF` überhaupt nicht optimieren.
- `INFO` Informationen über das Ergebnis ausgegeben.
- `MORE_GLOBALS` eine zusätzliche Optimierung, um mehr Globalisierung zu haben.
- `NO_VARIABLES` es werden keine globalen Variablen verwendet.
- `NO_STACKS` es wird kein Keller zum Speichern der Variablen verwendet.
- `GROUPING VARIABLE` mehrere Attr. werden zu einer Variable gruppiert.
- `ATTRSPEZ` einem Attribut wird manuell ein Speicherort zugewiesen.

```
Type        ::= 'GLOBAL VAR' / 'GLOBAL STACK' / 'TREE_NODE' /
                'GROUP VAR' / 'GROUP STACK'
```

Spezifiziert den Speicherort von Attributen.

# Was fehlt in ELI

- Explizite Zwischendarstellung für und mit Optimierungen
- (Code-)Optimierungen
- Effektive Codegeneratoren
- Dokumente der (internen) Funktionsweise der Werkzeuge

# Inhalt – Werkzeuge

- ELI
  - Eigenschaften und Idee
  - Spezifikationen
    - Dateitypen für Anwender
    - Beispiele
    - Problemgetrieben (Ein- / Ausgabe für Werkzeuge)
  - Technik hinter ELI
    - Odin
    - LIDO
- SUIF
- Trimaran

# SUIF: Aufgabengebiete

I. Entwerfen von Übersetzern mit existierenden Läufen.

- Frontends + Läufe.
- Einfache Zusammenstellung von Übersetzerkomponenten (zur Laufzeit).

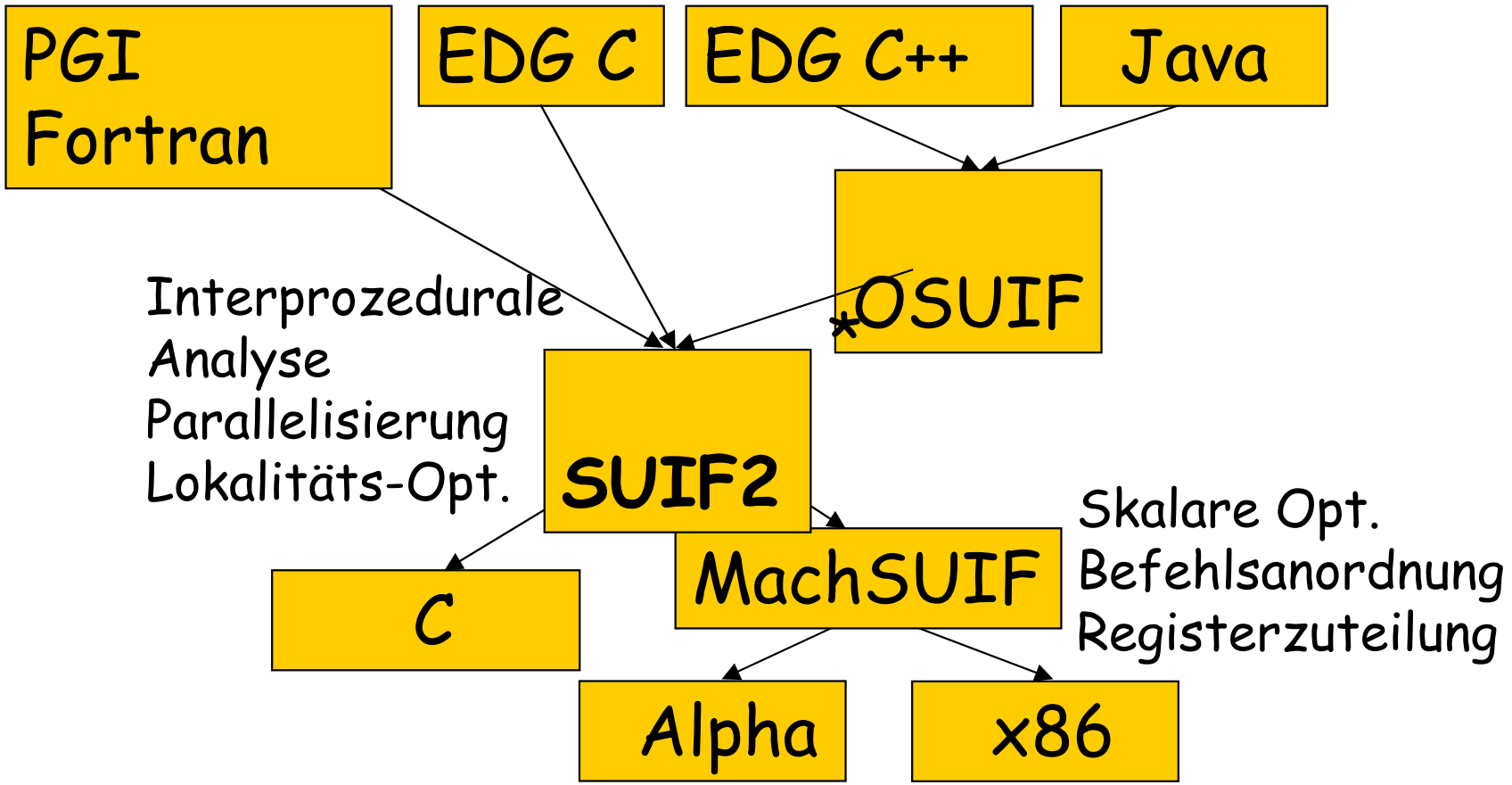
II. Entwickeln von neuen Läufen.

- Benutzer kann sich auf die algorithmischen Themen konzentrieren.
- Die Infrastruktur kümmert sich um die üblichen Funktionalitäten.
- Standard Zwischendarstellung (IR) und zugehörige Werkzeuge.
- Übliche Hilfsfunktionen und Datenstrukturen.

III. Entwickeln neuer IRs (neue Sprachkonstrukte oder Analysen).

- Einfache Definition von IR-Knoten.
- Alter Code arbeitet mit neuer IR ohne Neuübersetzung.

# Das SUIF-System




* C++ OSUIF nach SUIF ist unvollständig

# Komponenten von SUIF I

- Grundstrukturen
  - Erweiterbare Zwischensprache
  - Hoof: SUIF Objekt Spezifikationssprache
  - Standard IR
  - Modulares Übersetzersystem
  - Datenstrukturen (z.B. Hashtabellen)
- Objektorientierte Infrastruktur (OSUIF Repräsentation)
  - Java OSUIF $\rightarrow$ SUIF herunterbrechen
  - Objektanordnung und Methoden-Dispatch
- Backend Infrastruktur (MachSUIF Repräsentations- und Optimierungs-Rahmen)
  - Skalare Optimierungen
  - CSE
  - Eliminierung von totem Code
  - Nachoptimierungen
  - Registerzuteilung mit Graphenfärben
  - Alpha und x86 Codegeneratoren

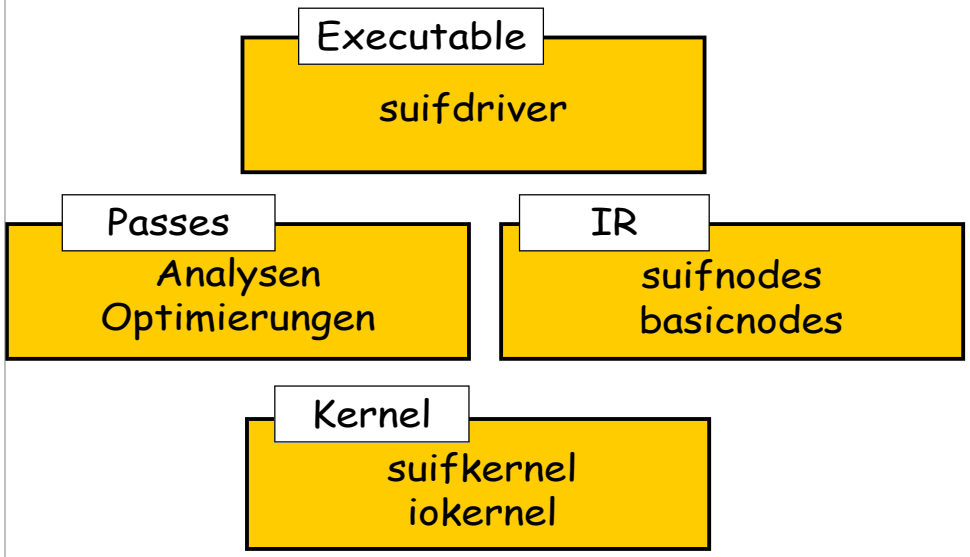
# Komponenten von SUIF II

- Infrastruktur für Analysen (Optimierungen) auf hoher Ebene
  - Graphen (Zwischendarstellung)
  - Iterierte Dominanzgrenzen
- Intraprozedurale Analysen
  - Kopienfortschreibung
  - Eliminierung von totem Code
  - Steensgaard's-Alias-Analyse
- Aufrufgraph
  - Rahmenwerk für Regionen
  - Rahmenwerk zur interprozeduralen Analyse
- Ablaufsteuerungsgraphen
- Interprozedurale Regionen-gestützte Analyse
  - Reihungs-Abhängigkeit & Privatisierung
  - Eliminieren von Skalaren & Privatisierung

# Komponenten von SUIF III

- Interprozedurale Parallelisierung
  - Preßburger Arithmetik (omega)
  - Farkas Lemma
  - Paket zur Gauß- Elimination
- Affine Partitionierungen von Feldern für Parallelitäts- u. Lokalitäts-Erhöhung
  - Unimodulare Transformation (austauschen, umdrehen, zerstückeln)
  - Vereinigung und Aufspalten von Schleifen
  - Neu-Indizieren und Skalieren von Befehlen und Ausdrücken
  - Blockerzeugung für nicht perfekt geschachtelte Schleifen

# Die SUIF-Module




- Kernel: Stellt die Grundfunktionalität zur Verfügung
  - iokernel: implementiert I/O
  - Suifkernel:
    Versteckt den iokernel und führt die Modul-Unterstützung durch, ferner: cloning, Verarbeitung der Kommandozeilen, Liste von Fabriken, ...
- Module
  - Läufe: Stellt ein „Lauf"-Rahmenwerk zur Verfügung
  - IR: Grundlegende Programm-repräsentation
- Suifdriver
  - Stellt Kontrolle über die Ausführung der Module und Läufe zur Verfügung.

# Trimaran

Trimaran ist eine Infrastruktur zur Untersuchung von Parallelität auf Befehlsebene.

Es ist geeignet für Forscher, die interessiert sind an:

- Explicitly Parallel Instruction Computing (EPIC)
- Hochleistungs-Computer-Systemen
- Instruction-level parallelism (ILP)
- Übersetzer-Optimierungen
- Rechnerarchitektur
- Adaptive und eingebettete Systeme
- Sprachentwurf

# Eigenschaften

- Das Trimaran System ist ein integrierter Übersetzerbaukasten mit einer Infrastruktur zur Leistungsüberwachung.
- Ziel: Entwicklung von EPIC-Architekturen (explicitly parallel instruction computer)
- Der architektonische Freiraum, den Trimaran abdeckt, wird gekennzeichnet durch HPL-PD, eine prametrisierbare Prozessorarchitektur, die neuartige Funktionen unterstützt:
  - Konditionelle Ausdrücke (predication)
  - Ablaufssteuerung- und Datenspekulation
  - Übersetzer-kontrollierte Verwaltung der Speicherhierarchie
- Trimaran hat eine Menge an Analyse- und Optimierungs-Moduln sowie eine graphbasierte Zwischensprache.
- Optimierungen und Analysen können leicht hinzugefügt oder übersprungen werden.
- Trimaran verfügt über eine detailreiche Simulationsumgebung und ein flexibles Leisungsüberwachungssytem, das die Leistung verfolgt, während das Maschinenmodell variiert wird.

# Komponenten von Trimaran I

- Das System ist derzeit auf EPIC-Architekturen orientiert und unterstützt Übersetzerforschung in typischen Codeerzeugungsbereichen wie Registerzuteilung, Befehlsanordnung und maschinenabhängige Optimierungen.

Die Übersetzerinfrastruktur Trimaran besteht aus folgenden Teilen:

- Eine Einrichtung zur Beschreibung von Maschinen, mdes, mit einer ILP-Architektur.
- Eine parameterisierbare ILP-Architektur gennant HPL-PD.
- Ein Übersetzerfrontend, genannnt IMPACT, für C.
  Es führt die Zerteilung, Typprüfung und eine breite Palette an maschinenunabängigen Optimierungen durch. Es gibt auch Optimierungen auf Befehlsebene zur Erhöhung der ILP.
- Ein Übersetzerbackend, genannt Elcor, parameterisiert durch eine Maschinenbeschreibung, das Befehlsanordnung, Registerzuteilung und machinenabängige Optimierungen durchführt. Jedes Phase des Backends kann von einem Übersetzerbauer leicht ersetzt oder geändert werden.

# Komponenten von Trimaran II

- Eine erweiterbare IR, die eine interne und Textdarstellung hat, mit Konvertierungsprogrammen zwischen den beiden. Die Textdarstellung wird „Rebel" genannt. Diese IR unterstützt moderne Übersetzertechniken, die u.a. auf Darstellen der Ablaufsteuerung, auf Daten- und Steuerabhängigkeiten, beruhen.
- Ein Simulator für die HPL-PD Architektur auf Maschinenzyklusebene, der durch eine Maschinenbeschreibung konfigurierbar ist. Er liefert als Ausgabe:
  - Ausführungszeit
  - Sprunghäufigkeit
  - Resourcenverbrauch

  Diese Informationen können für profilgetriebene Optimierungen, sowie zur Validierung neuer Optimierungen, verwendet werden.
- Eine integrierte graphische Benutzerschnittstelle (GUI) für das Konfigurieren und das Steuern des Ablaufs des Systems Trimaran stehen zur Verfügung. Die GUI umfasst Hilfsmittel für die graphische Sichtbarmachung der Programmzwischendarstellung und der Leistungsresultate.

# Besprochene Übersetzerbaukasten

Cocktail ein Übersetzerbaukasten:

- http://www.first.gmd.de/cocktail/

ELI ein Übersetzerbaukasten mit Betonung des Erstellungsprozess:

- http://www.uni-paderborn.de/project-hp/eli.html

SUIF ein Übersetzerbaukasten für Forschungszwecke
mit besonderem Augenmerk auf Optimierungen:

- http://suif.stanford.edu/

Trimaran ein System zur Simulierung von Hardware für experimentelle Zwecke:

- http://www.trimaran.org/