Compilerbau

Wintersemester 2009 / 2010

Dr. Heiko Falk

Technische Universität Dortmund

Lehrstuhl Informatik 12

Entwurfsautomatisierung für Eingebettete Systeme

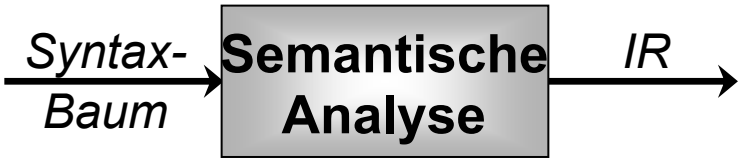
Kapitel 5

Semantische Analyse

Gliederung der Vorlesung

- Kapitel 1: Compiler – Abhängigkeiten und Anforderungen
- Kapitel 2: Interner Aufbau von Compilern
- Kapitel 3: Lexikalische Analyse (Scanner)
- Kapitel 4: Syntaktische Analyse (Parser)
- **Kapitel 5: Semantische Analyse**
  - Einführung
  - Interne Zwischendarstellungen
  - Semantische Analyse mittels Symboltabellen und BISON
- Kapitel 6: Instruktionsauswahl
- Kapitel 7: Register-Allokation
- Kapitel 8: Code-Optimierung
- Kapitel 9: Ausblick

Aufgabe der Semantischen Analyse (1)



Semantische Analyse *(IR Generator)*:

- Namensanalyse (z.B. Gültigkeitsbereiche von Symbolen)
- Prüfung, dass jeder Ausdruck korrekten Typs ist *(Typanalyse)*
- Aufbau von Symboltabellen (Abbildung von Bezeichnern zu deren Typen und Positionen)
- Erzeugung einer Internen Zwischendarstellung *(Intermediate Representation, IR)* zur weiteren Verarbeitung

Aufgabe der Semantischen Analyse (2)

Typische Fragestellungen:

- Ist `x` eine skalare Variable, Feld oder Funktion? Ist `x` deklariert?
- Werden Bezeichner benutzt, die nicht deklariert sind? Deklariert, aber nicht benutzt?
- Welche Deklaration gehört zu einer konkreten Nutzung von `x`?
- Ist der Ausdruck `x * y + z` typ-konsistent?
- Hat `a` drei Dimensionen für Ausdruck `a[i, j, k]`?
- Wo kann `z` gespeichert werden? (Register, Stack, global, static?)
- Wie ist die Zahl 15 in `f = 15;` zu repräsentieren?
- Wie viele Argumente benötigt `foo()`?
- Wurde `x` definiert, bevor es genutzt wird?

Kontextabhängige Analyse

Fragestellungen jenseits der Syntax:

- Fragestellungen von voriger Folie gehen über reine Syntax einer Programmiersprache weit hinaus.
- Antworten können nur berechnet werden, wenn Ausdrücke, Bezeichner, etc. in ihrem jeweiligen Kontext im zu übersetzenden Programm analysiert werden.

☞ Analysen auf Basis kontextfreier Grammatiken, wie z.B. während des Parsens, hierfür nicht mächtig genug.

☞ Aufteilung von syntaktischer und semantischer Analyse in zwei getrennte Compilerphasen.

Aufgabe der Semantischen Analyse (3)

Interne Zwischendarstellungen:

- Könnte der Syntaxbaum des Parsers nicht zentrale Datenstruktur für alle weiteren Compilerphasen sein? Wozu eine IR?
- Syntaxbaum ist zu fokussiert auf kontextfreie Grammatik der Programmiersprache: alle inneren Knoten sind mit den Nichtterminalen der Grammatik markiert *(☞ Kapitel 4)*. Insbes. durch Grammatik-Transformationen werden viele neue Nichtterminale eingefügt, die sich im Inneren der Syntaxbäume wiederfinden.
- Informationen sind für übrigen Compiler nach Parsen irrelevant, verbergen bloß reale Struktur des zu übersetzenden Programms.
- Syntaxbäume haben nur wenig Möglichkeiten, semantische, kontextabhängige Daten zu ermitteln, speichern und abzurufen.

☞ Separate, effiziente Datenstrukturen sind notwendig!

Ansätze zur Semantischen Analyse

Der formale Weg:

- Verwendung von kontextsensitiven Grammatiken oder
- von attributierten Grammatiken

Der praktische Weg:

- Aufbau von Symboltabellen und
- Nutzung von ad-hoc in die Syntaxanalyse eingeflochtenem Code zur semantischen Analyse.

☞ Während sich bei lexikalischer und syntaktischer Analyse die Formalismen durchgesetzt haben, wird die semantische Analyse i.d.R. ad-hoc realisiert.

- Im folgenden: Übersicht über IRs und über semantische Aktionen in BISON.

Gliederung der Vorlesung

- Kapitel 1: Compiler – Abhängigkeiten und Anforderungen
- Kapitel 2: Interner Aufbau von Compilern
- Kapitel 3: Lexikalische Analyse (Scanner)
- Kapitel 4: Syntaktische Analyse (Parser)
- **Kapitel 5: Semantische Analyse**
  - Einführung
  - Interne Zwischendarstellungen
  - Semantische Analyse mittels Symboltabellen und BISON
- Kapitel 6: Instruktionsauswahl
- Kapitel 7: Register-Allokation
- Kapitel 8: Code-Optimierung
- Kapitel 9: Ausblick

Abstraktionsniveaus von IRs

```
float a[20][10];
... a[i][j+2] ...;
```

- **High-Level**

  `t1 ← a[i,j+2]`

- **Medium-Level**

  t_1 ← j+2

  t_2 ← i*10

  t_3 ← t_1+t_2

  t_4 ← 4*t_3

  t_5 ← addr a

  t_6 ← t_5+t_4

  t_7 ← *t_6

- **Low-Level**

  r1 ← [fp-4]

  r2 ← r1+2

  r3 ← [fp-8]

  r4 ← r3*10

  r5 ← r4+r2

  r6 ← 4*r5

  r7 ← fp-216

  f1 ← [r7+r6]

Abstraktionsniveaus von IRs

High-Level IRs:

- Repräsentation sehr nah am Quellcode
- Oft: Abstrakte Syntaxbäume
- Variablen & Typen zur Speicherung von Werten
- Erhaltung komplexer Kontroll- & Datenflussoperationen (insbes. Schleifen, if-then / if-else Ausdrücke, Array-Zugriffe `[]`)
- Rücktransformation der High-Level IR in Quellcode leicht

*[**S. S. Muchnick,** Advanced Compiler Design & Implementation, Morgan Kaufmann, 1997]*

Abstraktionsniveaus von IRs

Medium-Level IRs:

- Drei-Adress-Code: $a_1 \leftarrow a_2$ *op* a_3;
- IR-Code unabhängig von Quell-Sprache & Ziel-Prozessor
- Temporäre Variablen zur Speicherung von Werten
- Komplexe Kontroll- & Datenflussoperationen vereinfacht (Labels & Sprünge, Zeiger-Arithmetik)
- Kontrollfluss in Form von *Basisblöcken*

Definition: Ein *Basisblock* $B=(I_1, ..., I_n)$ ist eine Befehlssequenz maximaler Länge, so dass

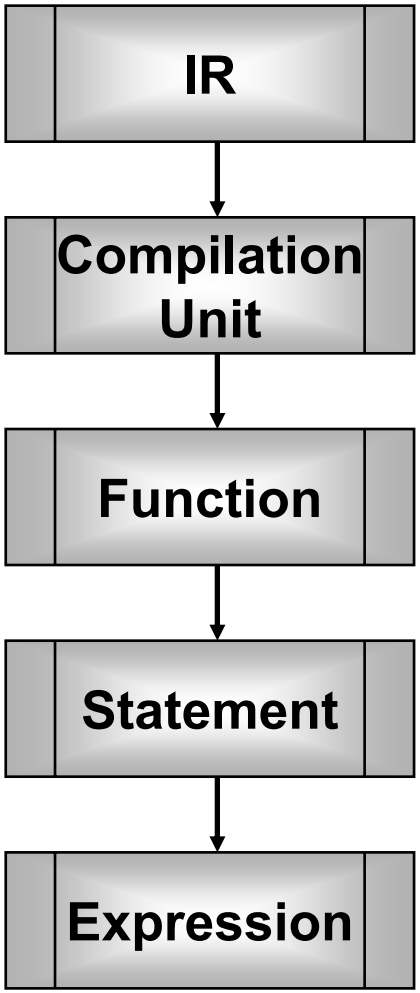
- B nur durch die erste Instruktion I_1 betreten wird, und
- B nur durch die letzte Instruktion I_n verlassen wird.

Abstraktionsniveaus von IRs

Low-Level IRs:

- Repräsentation von Maschinen-Code
- Operationen entsprechen Maschinenbefehlen
- Register zur Speicherung von Werten
- Transformation der Low-Level IR in Assemblercode leicht

High-Level IR: ICD-C

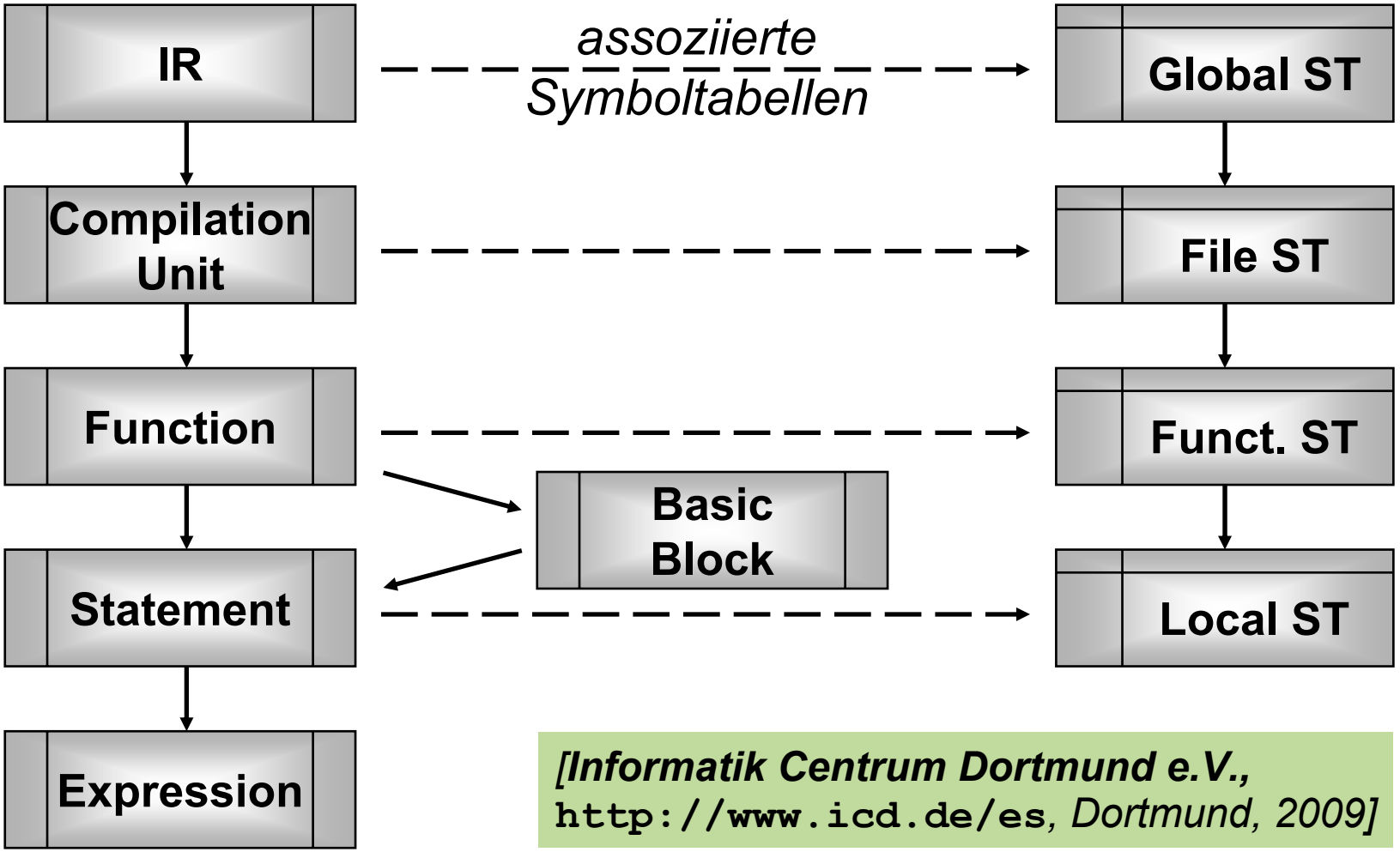


- 1 C-File bei gleichzeitiger Übersetzung mehrerer Quell-Dateien

- Loop Statements (for, do-while, while-do)
- Selection Statements (*if*, *if-else*, *switch*)
- Jump Statements (*return*, *break*, *continue*, ...)
- ...

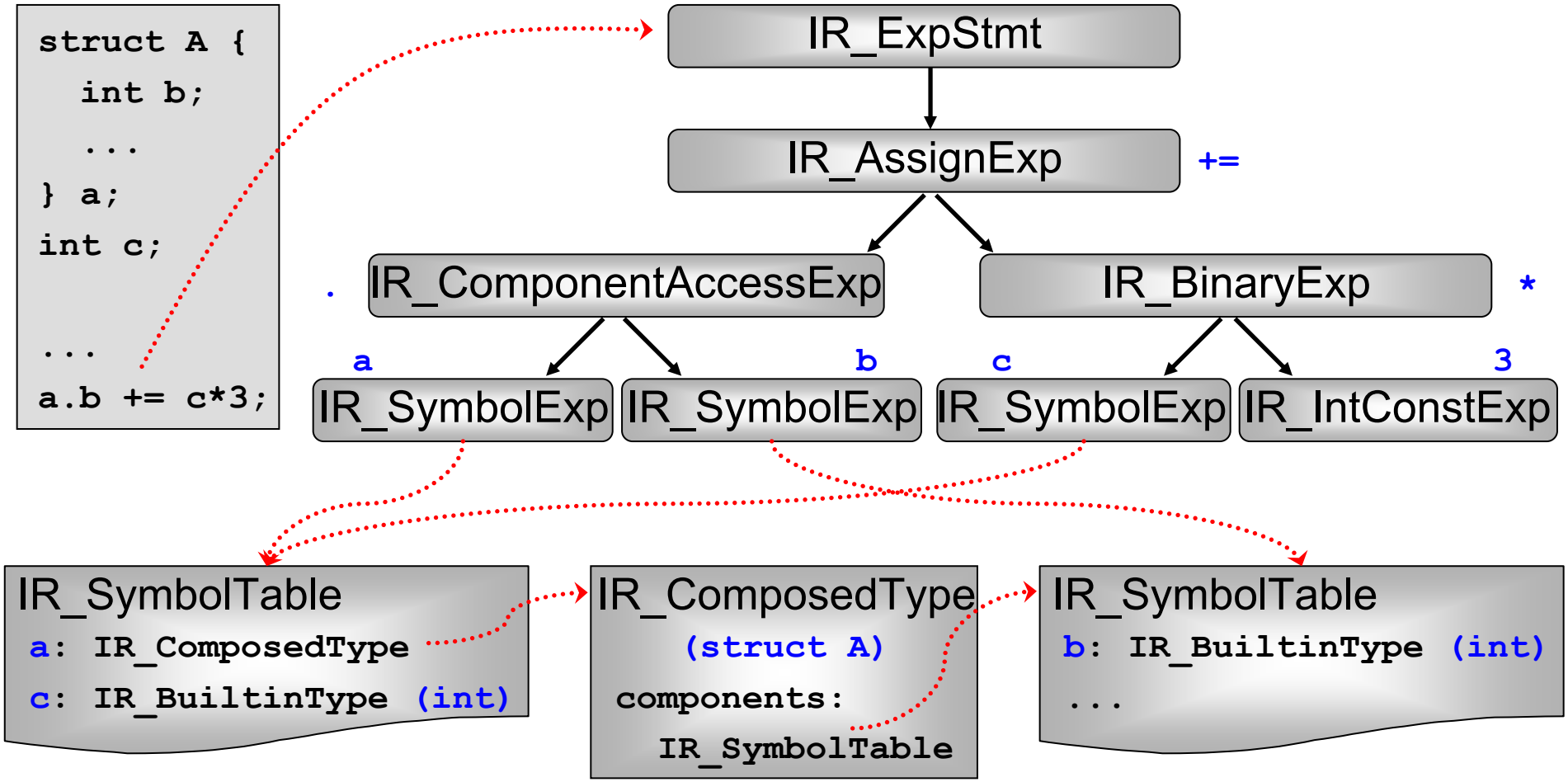
- Binary & Unary Expressions (+, -, *, /, ...)
- Zuweisungsoperatoren (=, +=, -=, ...)
- Index & Komponentenzugriff (*a[x]*, *a.x*, ...)
- ...

High-Level IR: ICD-C



[Informatik Centrum Dortmund e.V., `http://www.icd.de/es`*, Dortmund, 2009]*

ICD-C: Code-Beispiel



ICD-C: Features

- **ANSI-C Compiler Frontend:** C89 + C99 Standards
  GNU Inline-Assembler
- **Enthaltene Analysen:** Datenflussanalysen
  Kontrollflussanalysen
  Schleifenanalysen
  Zeigeranalyse
- **Schnittstellen:**
  - ANSI-C Dump der IR als Schnittstelle zu externen Tools
  - Schnittstelle zur Code-Selektion in Compiler-Backends
- **Interne Strukturen:**
  - Objektorientiertes Design (C++)

Medium-Level IR: MIR

- **MIR Program:** 1 – *N* Program Units *(d.h. Funktionen)*
- **Program Unit:** **`begin`** MIRInst* **`end`**

- **MIR-Instruktionen:**
  - Quadrupel: 1 Operator, 3 Operanden *(d.h. 3-Adress-Code)*
  - Instruktionstypen:
    Zuweisungen, Sprünge (**`goto`**), Bedingungen (**`if`**),
    Funktionsaufruf & -rücksprung (**`call`**, **`return`**),
    Parameterübergabe (**`receive`**)
  - Können MIR Ausdrücke *(Expressions)* enthalten

Medium-Level IR: MIR

- **MIR Ausdrücke:**
  - Binäre Operatoren: `+`, `-`, `*`, `/`, `mod`, `min`, `max`
  - Relationale Operatoren: `=`, `!=`, `<`, `<=`, `>`, `>=`
  - Schiebe- & Logische Operatoren: `shl`, `shr`, `shra`, `and`, `or`, `xor`
  - Unäre Operatoren: `-`, `!`, `addr`, `cast`, `*`

- **Symboltabelle:**
  - Enthält Variablen und symbolische Register
  - Einträge haben Typen: `integer`, `float`, `boolean`

*[**S. S. Muchnick,** Advanced Compiler Design & Implementation, Morgan Kaufmann, 1997]*

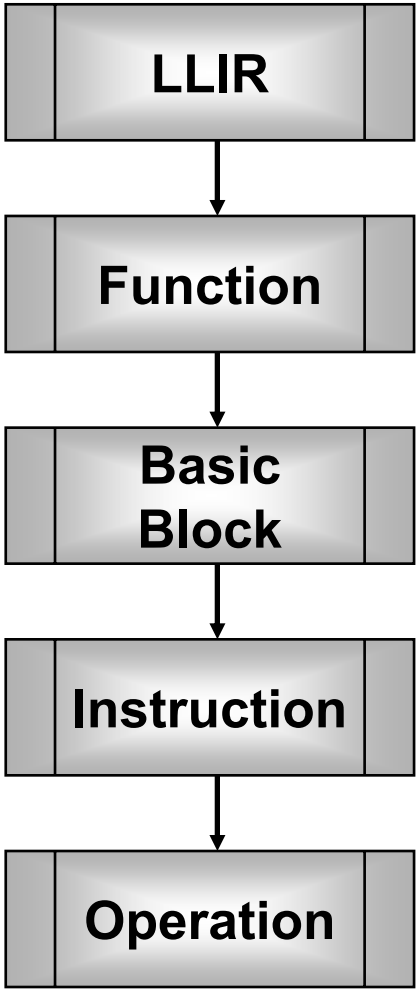
MIR: Eigenschaften

- **MIR ist keine High-Level IR:**
  - Nähe zur Quellsprache fehlt
  - High-Level Konstrukte fehlen: Schleifen, Array-Zugriffe, ...
  - Nur wenige, meist simple Operatoren präsent

- **MIR ist keine Low-Level IR:**
  - Nähe zur Zielarchitektur fehlt: Verhalten von Operatoren ist maschinenunabhängig definiert
  - Konzept von Symboltabellen, Variablen & Typen nicht low-level
  - Abstrakte Mechanismen zum Funktionsaufruf, Rücksprung und Parameterübergabe

☞ ***MIR ist eine Medium-Level IR.***

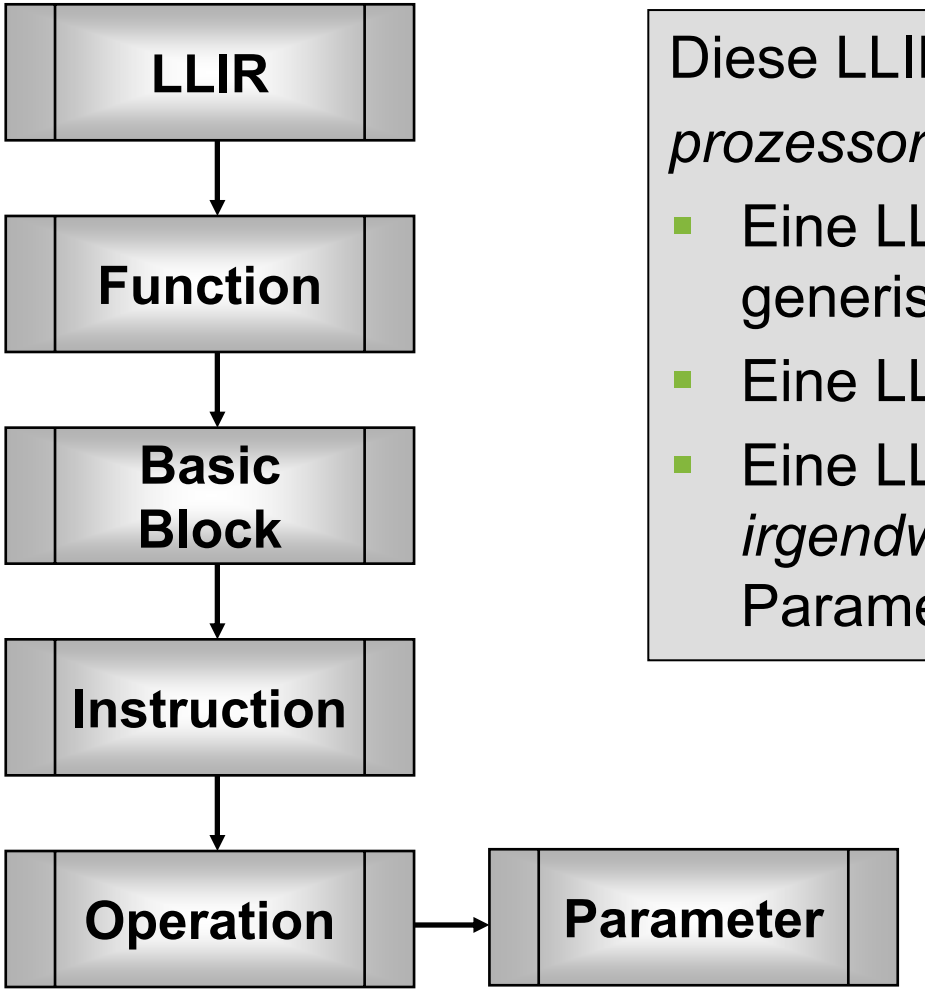
Low-Level IR: LLIR



- Maschinen-Instruktion
- Enthält 1-*N* Maschinen-Operationen
- Operationen werden parallel ausgeführt *(☞ VLIW)*

- Maschinen-Operation
- Enthält Assembler-Opcode (z.B. **`ADD`**, **`MUL`**, ...)
- Enthält 0-*M* Parameter

Low-Level IR: LLIR

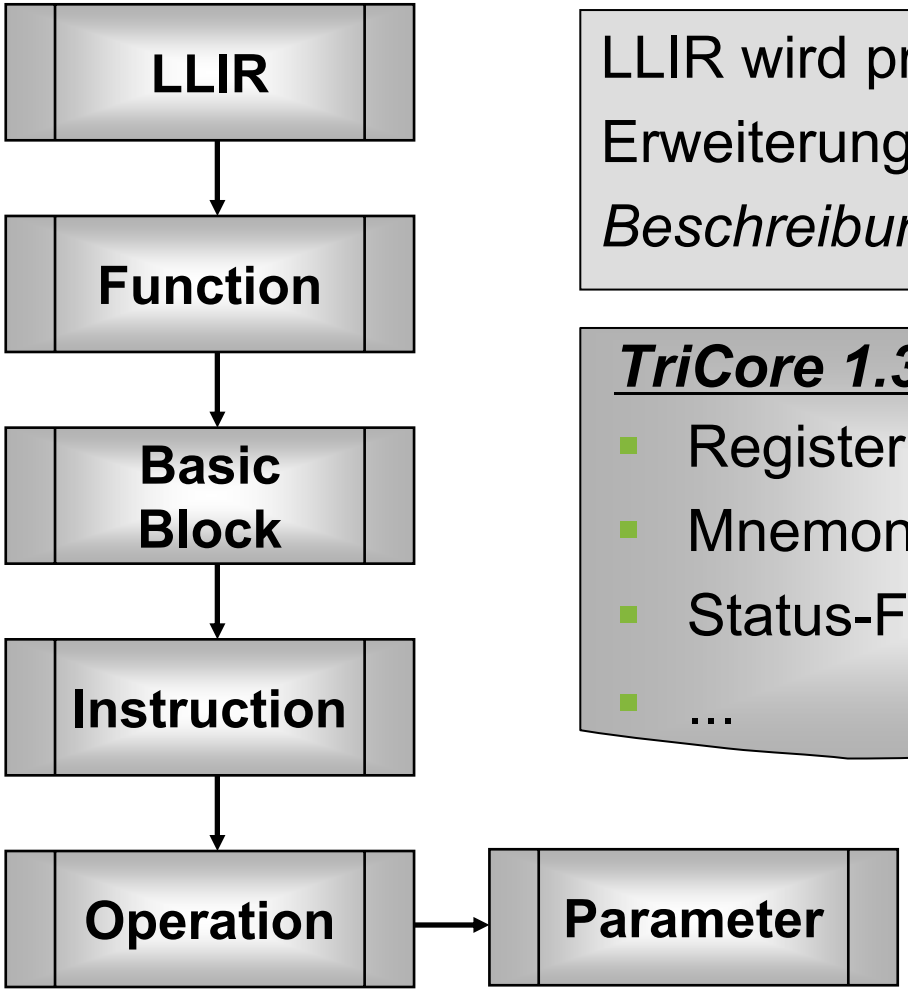


Diese LLIR-Struktur ist vollkommen *prozessor-unabhängig*:

- Eine LLIR besteht aus *irgendwelchen* generischen Funktionen
- Eine LLIR-Funktion besteht aus...
- Eine LLIR-Operation besteht aus *irgendwelchen* generischen Parametern

- Register
- Integer-Konstanten & Labels
- Adressierungsmodi
- ...

Low-Level IR: LLIR



LLIR wird prozessor-spezifisch durch Erweiterung um eine *Prozessor-Beschreibung*:

TriCore 1.3:

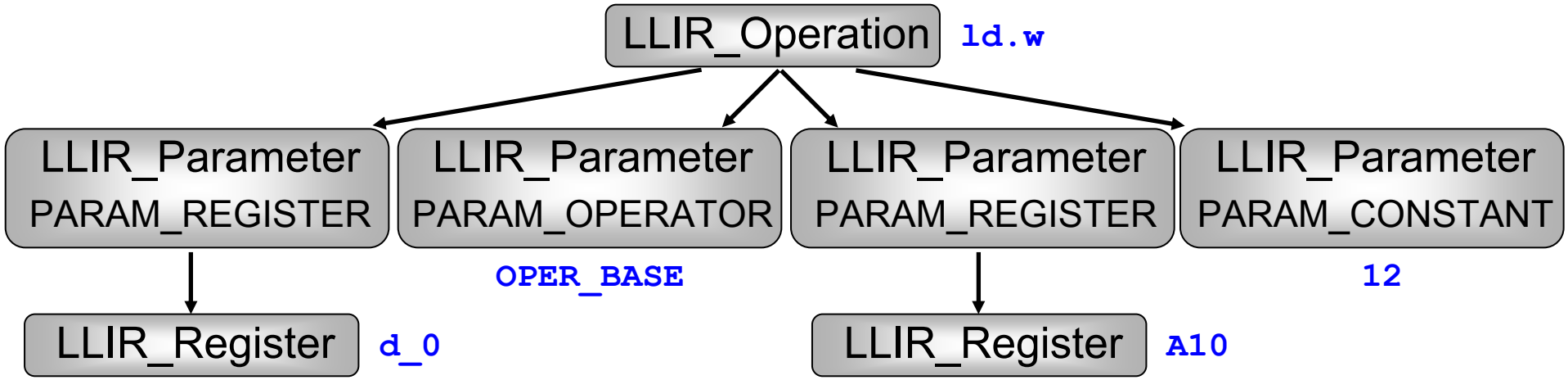
- Register = {`D0`, ..., `D15`, `A0`, ..., `A15`}
- Mnemonics = {`ABS`, `ABS.B`, ..., `XOR.T`}
- Status-Flags = {`C`, `V`, ..., `SAV`}
- ...

[Informatik Centrum Dortmund e.V., `http://www.icd.de/es`, *Dortmund, 2009]*

LLIR: Code-Beispiel *(Infineon TriCore 1.3)*

```
ld.w %d_0, [%A10] 12;
```

- Lade Speicherinhalt von Adresse `[%A10] 12` in Register `d_0`
- *Erinnerung:* Register `A10` = Stack-Pointer ☞ Phys. Register
- Adresse `[%A10] 12` = Stack-Pointer + 12 Bytes
  (sog. Base + Offset-Adressierung)
- TriCore hat kein Register `d_0` ☞ Virtuelles Datenregister



LLIR: Features

- **Retargierbarkeit:**
  - Anpassbarkeit auf verschiedenste Prozessoren (z.B. DSPs, VLIWs, NPUs, ...)
  - ☞ Modellierung verschiedenster Befehlssätze
  - ☞ Modellierung verschiedenster Registersätze
- **Enthaltene Analysen:**
  - Datenflussanalysen
  - Kontrollflussanalysen
- **Schnittstellen:**
  - Einlesen und Ausgabe von Assembler-Dateien
  - Schnittstelle zur Code-Selektion

Gliederung der Vorlesung

- Kapitel 1: Compiler – Abhängigkeiten und Anforderungen
- Kapitel 2: Interner Aufbau von Compilern
- Kapitel 3: Lexikalische Analyse (Scanner)
- Kapitel 4: Syntaktische Analyse (Parser)
- **Kapitel 5: Semantische Analyse**
  - Einführung
  - Interne Zwischendarstellungen
  - Semantische Analyse mittels Symboltabellen und BISON
- Kapitel 6: Instruktionsauswahl
- Kapitel 7: Register-Allokation
- Kapitel 8: Code-Optimierung
- Kapitel 9: Ausblick

Symboltabellen

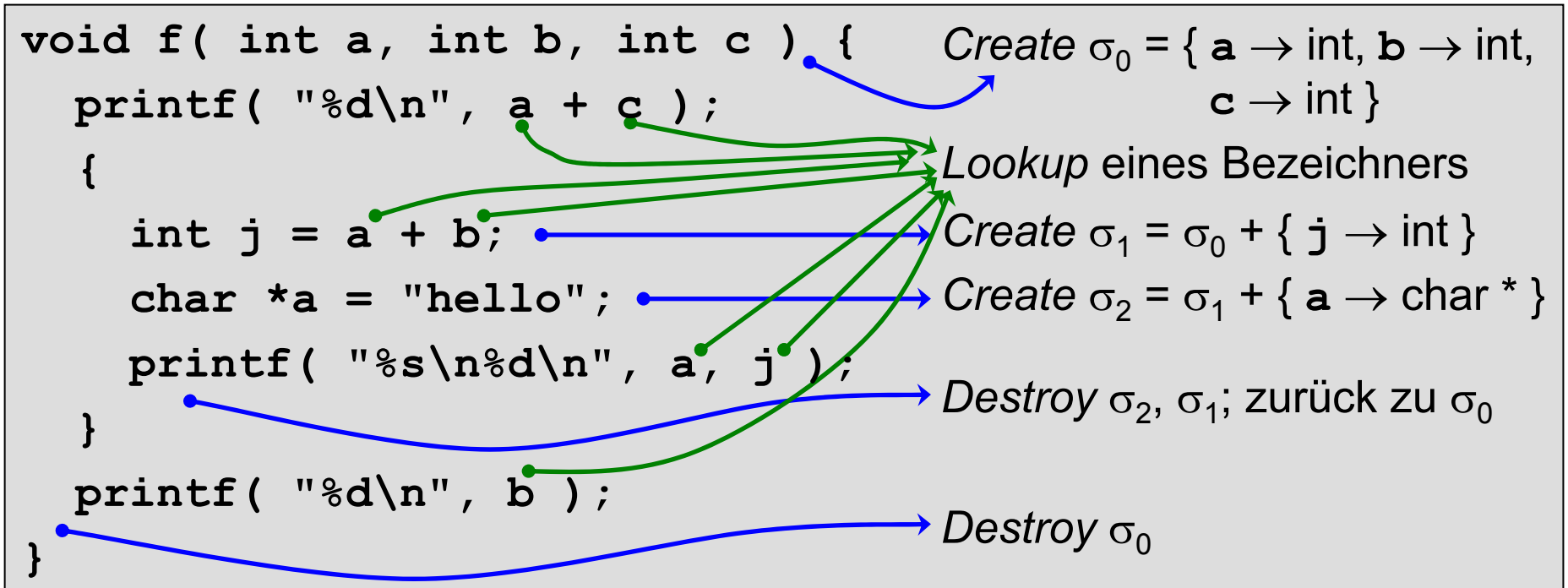
Symboltabellen:
Eine Symboltabelle ist eine Abbildung, die Bezeichner eines Programms, z.B. Variablennamen, Funktionsnamen etc., auf beliebige weitere Informationen, z.B. Typ-Information, abbildet.

Bemerkungen:

- I.d.R. sind für ein Programm in einer Quellsprache mehrere / viele Symboltabellen zu verwalten.
- Dies hängt stark von der Struktur der Quellsprache und insbes. von deren Sichtbarkeitsregeln ab.

☞ Beispiel: Ein Bezeichner `a`, der lokal in einer Funktion deklariert ist, verdeckt üblicherweise eine gleichnamige globale Variable `a`.

Operationen auf Symboltabellen



- Operator + muss Sichtbarkeitsregeln der Sprache umsetzen.
- *Destroy* kehrt zu „älteren“ Tabellen zurück ☞ *Stack-Organisation*
- *Lookup* von Bezeichnern muss sehr effizient (≈ $O(1)$) sein!

Organisation von Symboltabellen

Verwaltung und Umsetzung:

- Bei jeder *Deklaration* eines Bezeichners wird neue Symboltabelle kreiert, indem vorige Tabelle um neu deklarierte Bezeichner ergänzt wird und verdeckte Bezeichner „ausgeblendet“ werden. Dies geschieht kellerartig.
- Bei jeder *Verwendung* eines Bezeichners im Programmcode erfolgt ein Lookup. Wegen typischerweise vieler solcher Verwendungen muss Lookup sehr effizient sein. Lookups suchen ein Symbol stets in der jüngsten Tabelle auf dem Keller.
- Mit Verlassen eines *Sichtbarkeitsbereichs (Scope)* sind zuletzt eingekellerte Tabellen zu verwerfen.
- Datenstrukturen: Hash-Tabellen, binäre Suchbäume, Buckets, verkettete keller-organisierte Listen

Integration Semantische / Syntaktische Analyse

Teilaufgaben der Semantischen Analyse:

- Erzeugung und Zerstörung von Symboltabellen, Suche von Bezeichnern.
- Durchführung semantischer Tests, abhängig von der Semantik der Quellsprache, unter Nutzung der jeweils aktuellen Symboltabelle.
- Aufbau einer Intermediate Representation.

Regeln in BISON Parser-Spezifikation:

- `<Nichtterminal> : <rechte Seite>` { *semantische Aktion* }
- Die semantische Aktion kann beliebigen C-Code enthalten, der bei Verwendung einer Regel zum Bottom-Up Parsen ausgeführt wird.
- Hier ist der Code zur Symboltabellen-Verwaltung, zur Durchführung semantischer Tests und zum IR-Aufbau anzugeben.

Semantische Aktionen in BISON

Bottom-Up Propagierung von semantischen Attributen:

- In der semantischen Aktion einer BISON-Regel können semantische Informationen / Attribute für den aktuellen Knoten im Syntaxbaum generiert und an den Knoten angehängt werden.
- Zum Generieren semantischer Attribute kann auf die Attribute der Kind-Knoten, d.h. auf die der Symbole auf der rechten Regel-Seite zugegriffen werden.
- ☞ Semantische Attribute pflanzen sich im Syntaxbaum entlang der Vorgehensweise des BISON-Parsers fort, d.h. Bottom-Up.
- Namenskonvention zum Zugriff auf Attribute des aktuellen und der Kind-Knoten in BISON:

  `$$` = Symbol auf linker Regel-Seite, d.h. aktueller Knoten

  `$1` = 1. Symbol auf rechter Regel-Seite, `$2` = 2. Symbol, …

Anwendungsfall: Variablendeklaration

```
vardecl : ID colon type semicolon
  { createVariableDeclaration( $1, $3 ); }
```

- Bei Anwendung dieser Regel während des Parsens wird eine Variablendeklaration erzeugt, die aus dem Bezeichner des Tokens `ID` (= `$1`) und dessen Typ in `$3` besteht.
- Für dieses Beispiel wird angenommen, dass die Verwaltung der Symboltabellen über globale Variablen geschieht, und dass `createVariableDeclaration` die erzeugte Deklaration direkt auf dem Stack der Symboltabellen einkellert.

Anwendungsfall: Verwendung von Variablen

```
var : ID
  { symbolTableEntry *p = lookup( $1->value );
    if ( p == NULL ) yyerror( "Undeclared identifier" );
    else if ( p->declarationType == procedureDecl )
      yyerror( "Declared identifier not a variable" );
    else
      $$ = createVariable( $1->value, p->type ); }
```

- Wird eine Variable durch diese Regel abgeleitet, wird geprüft, ob sie deklariert ist und ob die gefundene Deklaration auch eine Variablendeklaration ist.
- Danach wird die IR für diese Variable per **`createVariable`** aufgebaut und zur weiteren Verwendung an Knoten **`var`** im Syntaxbaum gehängt.

Anwendungsfall: Aufbau der IR

Grammatik: *(leicht abgewandelte G_2 aus ☞ Kapitel 4)*

$S \rightarrow var$ **:=** E $\quad E \rightarrow E$ **+** T $\quad T \rightarrow T$ ***** F $\quad F \rightarrow var$ $\quad var \rightarrow$ **ID**

$E \rightarrow E$ **-** T $\quad T \rightarrow F$ $\quad F \rightarrow$ **num**

$E \rightarrow T$ $\quad F \rightarrow$ **(** E **)**

```
Stmt : var assign Expr { $$ = createAssign( $1, $3 ); }
Expr : Expr plus Term { $$ = createAdd( $1, $3 ); }
Expr : Expr minus Term { $$ = createSub( $1, $3 ); }
Expr : Term { $$ = $1; }
Term : Term times Factor { $$ = createMul( $1, $3 ); }
Term : Factor { $$ = $1; }
Factor : var { $$ = $1; }
Factor : num { $$ = createNumber( $1->value ); }
Factor : lbracket Expr rbracket { $$ = $2; }
```

Einschränkungen

Folgen der Integration semantische Analyse ↔ Parser:

- Wegen bottom-up Strategie des LR-Parsers können semantische Informationen nur von den Blättern zur Wurzel des Syntaxbaums geleitet werden. Der umgekehrte Weg ist nicht möglich.
- *Analog:* Kind-Knoten kennen keine Daten der „Geschwister“.
- Deklarationen von Bezeichnern müssen im Programmcode vor Verwendungen von Bezeichnern stehen *(was aber in der Praxis für reale Programmiersprachen keine echte Einschränkung ist)*.

Ausweg:

- Werden mehr Freiheiten gebraucht, kann semantische Analyse komplett losgelöst vom Parser implementiert werden, so dass Informationen beliebig durch den Syntaxbaum fließen können.

Literatur

Interne Zwischendarstellungen:

- Steven S. Muchnick, *Advanced Compiler Design & Implementation*, Morgan Kaufmann, 1997.
  ISBN 1-55860-320-4
- Andrew W. Appel, *Modern compiler implementation in C*, Cambridge University Press, 1998.
  ISBN 0-521-58390-X

Literatur

Symboltabellen und Semantische Aktionen:

- Andrew W. Appel, *Modern compiler implementation in C*, Cambridge University Press, 1998. *(☞ Kapitel 5)*
  ISBN 0-521-58390-X
- J. Eliot B. Moss, Compiler Techniques – Fall 2008, 2008. *(☞ Foliensätze 14-15)*
  `http://www-ali.cs.umass.edu/~moss/610.html`

Zusammenfassung

- **Semantik einer Programmiersprache: geht über reine Syntax, spezifiziert in kontextfreien Grammatiken, hinaus.**
- **Semantische Aktionen in BISON: Erlauben Ausführung beliebigen C-Codes bei Verwendung einer Regel**
- **Typische semantische Aktionen: Erzeugung und Pflege von Symboltabellen, Suche von Bezeichnern in Symboltabellen, Aufbau der Zwischenrepräsentation**
- **Kombinierter Ansatz Parser ↔ semantische Analyse hat sich gegen formale Ansätze (Attribut-Grammatiken) durchgesetzt**