Compilerbau

CompB Zusammenfassung

Danilo Bargen

Stand: 2013-09-08

https://github.com/HSR-Stud/CompB/

Inhaltsverzeichnis

1 Lexikalische Analyse

1.1 Aufgaben eines Lexers

Der Lexer liesst den Eingabestrom und unterteilt ihn in Lexeme. Zusätzlich trägt er Bezeichner in eine Symboltabelle ein.

Wichtig sind dabei folgende Begriffe:

Lexem Lexeme sind Zeichenketten, welche die kleinsten syntaktischen Einheiten einer Programmiersprache bilden.

Token Ein Token besteht aus einem Namen und einem optionalen Attributwert. Tokens formen eine syntaktische Kategorie für eine Klasse von Lexemen. Mögliche Tokens sind Schlüsselwörter, Bezeichner, Operatoren oder Konstanten.

Pattern Ein Pattern beschreibt den Aufbau eines Lexems eines Tokens. Häufig werden die Patterns als Reguläre Ausdrücke dargestellt.

Symboltabelle In der Regel werden die Tokens nicht als Text referenziert, sondern als Symbol/Zahl. Die Symboltabelle enthält Mappings von Symbolen zu Tokens.

Beispiel

Aus dem Programmcode rechts kann man die Lexeme und Tokens links herauslesen.

Lexem	Token
x	ID
=	ASSIGNMENT
foo	ID
-	ARITHMETIC
1	NUMBER
;	SEMICOLON

```
x = foo - 1;
```

1.2 Probleme, Schwerigkeiten

Viele Sprachkonstrukte sind in der Phase der Lexikalischen Analyse noch nicht eindeutig identifizierbar. Die Zeichenkette `fi` kann zum Beispiel als Identifier oder als falsch geschriebenes Schlüsselwort `if` auftreten. Dadurch ist es für einen Lexer auch sehr Schwer, Fehler im Programm zu erkennen.

Zudem spielt auch die Verarbeitungsreihenfolge eine wichtige Rolle. Werden beispielsweise Identifier vor den Schlüsselwörtern verarbeitet, so wird `while` als Variablenname und nicht als Schlüsselwort registriert.

2 Endliche Automaten

2.1 Umwandlung NFA → CFG

Ein regulärer Ausdruck in Form eines endlichen Automaten (NFA) kann auch als Kontextfreie Grammatik (CFG) dargestellt werden:

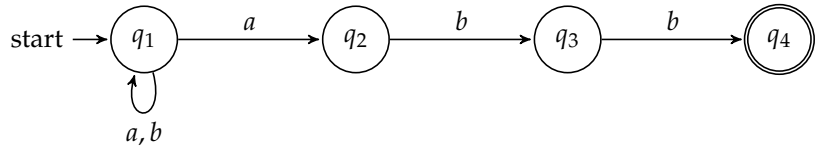
- Jeder Zustand des NFA wird zu einem Nicht-Terminal A_i
- Falls i der Startzustand ist, wird A_i das Startsymbol
- Ein Zustandsübergang $i \rightarrow j$ bei Eingabe von a wird zur Produktion $A_i \rightarrow aA_j$ (bei ε-Produktionen $A_i \rightarrow A_j$)
- Falls i ein Akzeptierzustand ist: $A_i \rightarrow \varepsilon$

Nachfolgend eine Beispielumwandlung:

Regex

$(a|b) * abb$

NFA



CFG

$A_0 \rightarrow aA_0 \mid bA_0 \mid aA_1$
$A_1 \rightarrow bA_2$
$A_2 \rightarrow bA_3$
$A_3 \rightarrow \varepsilon$

3 Syntaktische Analyse

3.1 Aufgaben eines Parsers

Die Aufgabe des Parser besteht darin, die Eingabesymbole, die der Lexer liefert, auf Korrektheit bezüglich der Grammatik der Sprache zu überprüfen.

Für korrekte Eingaben erstellt der Parser einen Syntaxbaum, welcher anschliessend weiterverarbeitet wird. Bei Syntaxfehlern sollte der Parser möglichst gute Fehlermeldungen liefern und falls möglich mit der Verarbeitung der Eingabe fortfahren.

Es gibt zwei Typen von Parsern: Top-Down und Bottom-Up Parser. Der Unterschied besteht darin, wie die Parse-Bäume aufgebaut werden, entweder von den Blättern zur Wurzel (Bottom-Up) oder umgekehrt (Top-Down).

3.2 Grammatik

Eine Grammatik besteht aus folgenden vier Elementen:

- Einer Menge Terminalsymbole (**Tokens**).
- Einer Menge Nicht-Terminalsymbole (**syntaktische Variablen**), aus welchen sich Terminalsymbole ableiten lassen.
- Einer Menge von Regeln (**Produktionen**). Auf der linken Seite einer Produktion steht ein Nicht-Terminalsymbol, auf der Rechten eine Folge von Terminalsymbolen und Nicht-Terminalsymbolen.
- Einem **Startsymbol**.

Nachfolgend eine Beispielgrammatik. Terminalsymbole sind in Kleinbuchstaben geschrieben, Nicht-Terminalsymbole mit Grossbuchstaben. Die erste Regel hat das Startsymbol auf der Linken Seite.

1. $E \rightarrow E + T$
2. $E \rightarrow T$
3. $T \rightarrow (E)$
4. $T \rightarrow id$

Es gibt verschiedene Typen von Grammatiken. Diese werden normalerweise mit $X_1X_2(k)$ bezeichnet:

- X_1: Leserichtung. Entweder L (links-nach-rechts) oder R (rechts-nach-links).
- X_2: Ableitungsrichtung. Entweder L (Linksableitung) oder R (Rechtsableitung).
- k: Anzahl Zeichen für Lookahead.

3.3 AST

Der AST (Abstract Syntax Tree) ist eine vereinfachte Variante des Parse-Trees, bei dem unnötige Details entfernt wurden. Mithilfe des ASTs werden Code-Generierung sowie Optimierungen durchgeführt.

3.4 Links- und Rechtsableitung

Links- oder Rechts- beschreibt die Reihenfolge, in der die Nicht-Terminale durch Terminale ersetzt werden. Bei der Linksableitung wird immer das erste Nichtterminal von links gesehen als erstes ersetzt.

3.5 Operator-Assoziativität

Links- oder Rechtsassoziativität beschreibt die Reihenfolge, in welcher Operatoren evaluiert werden. Dies ist vor allem bei arithmetischen Ausdrücken von Bedeutung. Als Beispiel: Addition ist linksassoziativ.

$$a + b + c + d = (((a + b) + c) + d)$$

Potenzierung ist rechtsassoziativ.

$$a^{b^c} = a^{(b^{(c)})}$$

Einige Operatoren können auch nichtassoziativ sein. Folgender Ausdruck ist beispielsweise in vielen Programmiersprachen nicht erlaubt:

$$a < b < c$$

3.6 Eliminieren von Linksrekursion

Einfache (direkte) Linksrekursionen können mithilfe des folgenden Algorithmus einfach entfernt werden:

- Es seien $A \to A\alpha_1 \mid A\alpha_2 \mid \ldots \mid A\alpha_m$ die linksrekursiven Regeln
- Es seien $A \to \beta_1 \mid \beta_2 \mid \ldots \mid \beta_n$ die nicht linksrekursiven Regeln
- Ersetze die erste Gruppe durch $A' \to \alpha_1 A' \mid \alpha_2 A' \mid \ldots \mid \alpha_m A'$
- Ersetze die zweite Gruppe durch $A \to \beta_1 A' \mid \beta_2 A' \mid \ldots \mid \beta_n A'$
- Füge folgende Regel hinzu: $A' \to \varepsilon$

Hinweis: Die Grammatik darf keine ε-Produktionen enthalten, ansonsten kann eine versteckte Linksrekursion auftreten.

Beispiel

$$A \to Aa \mid bA \mid c$$

...wird zu

$$A \rightarrow bAA' \mid cA'$$
$$A' \rightarrow aA' \mid \varepsilon$$

3.7 Linksfaktorisierung

Ein Parser liest den Tokenstream in der Regel von links nach rechts, deshalb macht es Sinn, wenn der Parser sofort entscheiden kann, welche Regeln er anwenden muss. Wenn zwei Produktionen auf der rechten Seite jedoch einen gemeinsamen Präfix haben, ist dies nicht möglich. Beispiel:

$$STMT \rightarrow if\ COND\ then\ STMT \mid if\ COND\ then\ STMT\ else\ STMT \mid OTHER$$

Der gemeinsame Präfix ist in diesem Fall *if COND then STMT*. Bei der Linksfaktorisierung wird der Teil nach diesem Präfix in ein separate Regel verschoben, damit der Entscheidungspunkt so weit verschoben wird, dass der Parser eine direkte Entscheidung treffen kann.

$$STMT \rightarrow if\ COND\ then\ STMT\ OPTELSE \mid OTHER$$
$$OPTELSE \rightarrow else\ STMT \mid \varepsilon$$

3.8 Top-Down Parser

Ein Top-Down Parser beginnt beim Startsymbol der Grammatik und baut den Parse-Baum von der Wurzel zu den Blättern sowie von links nach rechts auf.

3.8.1 Recursive Descent Parser

Eine speziell einfach implementierbare Variante eines Top-Down LL-Parsers ist der Recursive Descent Parser. Für jedes Nichtterminal wird eine Methode erstellt. Die Grammatikregeln werden als eine Folge von Aufrufen auf dieser Methode realisiert. Die Terminale werden in diesen Methoden konsumiert.

Für einen Recursive Descent Parser benötigt man keine Parsetabelle, jedoch muss mit Backtracking gearbeitet werden, was wiederum Performance-Einbussen bedeutet.

Beispiel

Gegeben ist folgende Grammatik:

$$E \rightarrow T \mid T + E$$
$$T \rightarrow int \mid int * T \mid (E)$$

Daraus ergeben sich folgende Tokens:

`INT`, `OPEN`, `CLOSE`, `PLUS`, `TIMES`

Nachfolgend der C-Code eines einfachen Recursive Descent Parsers:

```
bool term(TOKEN tok) { return *next++ ==  tok; }

bool E1() { return T(); }
bool E2() { return T() && term(PLUS) && E(); }
bool E() {
    TOKEN *save = next;
    return (next = save, E1()) || (next = save, E2());
}
bool T1() { return term(INT); }
bool T2() { return term(INT) && term(TIMES) && T(); }
bool T3() { return term(OPEN) && E() && term(CLOSE); }
bool T() {
    TOKEN *save = next;
    return (next = save, T1()) || (next = save, T2()) || (next = save, T3());
}
```

Das Problem bei diesem Algorithmus ist, dass er nicht alle Eingaben verarbeiten kann. Mit der Eingabe `3 * 4` beispielsweise denkt der Parser nach der Regel `T1` bereits, er hätte einen Match. Das folgende `*` kann er jedoch nicht mehr zuordnen und failt. Dies liegt daran, dass die Grammatik nicht linksfaktorisiert ist. Mit einer Linksfaktorisierung würde das Parsen aller Eingaben klappen. Aber auch ohne Linksfaktorisierung können Recursive Descent Parser mit erweitertem Backtracking implementiert werden, die mit allen Grammatiken klarkommt.

Ein grosser Vorteil des Recursive Descent Parsers ist die Einfachheit der Implementierung: Produktionen können quasi 1:1 im Code abgebildet werden.

3.9 Bottom-Up Parser

Ein Bottom-Up Parser folgt einer umgekehrten Rechtsableitung.

Beispiel

Gegeben ist folgende Grammatik:

$$E \to T \mid T + E$$
$$T \to int \mid int * T$$

Nachfolgend der Parse-Vorgang eines Bottom-Up Parsers für den Ausdruck $int * int + int$:

Schritt	Ausdruck	Anzuwendende Regel
1	$int * int + int$	$T \to int$
2	$int * \boldsymbol{T} + int$	$T \to int * T$
3	$\boldsymbol{T} + int$	$T \to int$
4	$T + \boldsymbol{T}$	$E \to T$
5	$T + \boldsymbol{E}$	$E \to T + E$
6	$\boldsymbol{E}$	

Wenn man diese Parse-Schritte von unten nach oben betrachtet, sieht man, dass immer das am weitesten rechts stehende Non-Terminalsymbol (fett hervorgehoben) ersetzt wurde, es ist also eine Rechtsableitung von unten nach oben ($\to$ Bottom-Up).

3.10 FIRST und FOLLOW Sets

3.10.1 FIRST-Set

Das FIRST-Set für ein Nicht-Terminal X besteht aus allen Terminalsymbolen, die auf X folgen können. Dabei werden nachfolgende Terminalsymbole aufgelöst.

Man betrachtet dafür alle Produktionen, bei welchen X auf der linken Seite steht:

1. Wenn auf der rechten Seite der Produktion als erstes ein Terminal steht, gehört dieses zum FIRST-Set von X.
2. Wenn auf der rechten Seite der Produktion ein ε steht, gehört dieses zum FIRST-Set von X.
3. Wenn auf der rechten Seite der Produktion als erstes ein Nicht-Terminal Y steht, gehört das FIRST-Set von Y (abzüglich allfälliger ε) zum FIRST-Set von X.
4. Wenn dieses FIRST-Set von Y ein ε enthält, berücksichtigt man auch alle Fälle, bei welchen Y weggelassen wird. Wenn es keine solchen Fälle gibt, gehört auch ε zum FIRST-Set von X.

Beispiel

Grammatik:

$S \to Ax \mid By \mid z$
$A \to 1CB \mid 2CB$
$B \to 3B \mid C$
$C \to 4 \mid \varepsilon$

Lösungsweg:

1. Das FIRST-Set von C ist $\{4, \varepsilon\}$ gemäss Regeln 1 und 2.
2. Das FIRST-Set von B enthält gemäss Regel 1 $\{3\}$.
3. Aufgrund der Produktion $B \to C$ und der Regel 3 enthält das FIRST-Set von B auch das FIRST-Set von C, abzüglich des ε.
4. Aufgrund der Regel 4 werden auch alle Produktionen berücksichtigt, bei denen das ε (und somit das C) „überlesen“ wird. Da auf C jedoch nichts mehr folgt, enthält FIRST(B) auch $\{\varepsilon\}$.
5. Das FIRST-Set von A ist gemäss Regel 1 $\{1, 2\}$.
6. Das FIRST-Set von S enthält gemäss Regel 1 $\{z\}$.
7. Das FIRST-Set von S enthält gemäss Regel 3 das FIRST-Set von A und B, abzüglich des ε in FIRST(B).
8. Da FIRST(B) ε enthält, wird zusätzlich der Fall berücksichtigt, bei dem B weggelassen wird. Da in der Produktion $S \to By$ auf das B ein y folgt, ist auch dieses in FIRST(S) enthalten. (Falls anstelle von y ein Non-Terminal folgen würde, müsste man auch dieses auflösen.)

Obige Lösungsschritte resultieren in folgenden FIRST-Sets:

$FIRST(S) = \{1, 2, 3, 4, y, z\}$
$FIRST(A) = \{1, 2\}$
$FIRST(B) = \{3, 4, \varepsilon\}$
$FIRST(C) = \{4, \varepsilon\}$

3.10.2 FOLLOW-Set

Im Gegensatz zum FIRST-Set betrachtet man beim FOLLOW-Set alle Produktionen, bei welchen X auf der rechten Seite steht.

1. Wenn X das Startsymbol ist, gehört \$ zum FOLLOW-Set von X.
2. Für jede Produktion $A \to \alpha X \beta$, füge FIRST(β) (ohne ε) zu FOLLOW(X) hinzu.
3. Falls ε in FIRST(β) ist, füge FOLLOW(A) zu FOLLOW(X) hinzu.
4. Für jede Produktion $A \to \alpha X$, füge FOLLOW(A) zu FOLLOW(X) hinzu.

Beispiel

Grammatik:

$S \to Ax \mid By \mid z$
$A \to 1CB \mid 2CB$
$B \to 3B \mid C$
$C \to 4 \mid \varepsilon$

Lösungsweg:

1. Das FOLLOW-Set von S enthält gemäss Regel 1 $\{\$\}$.
2. Das FOLLOW-Set von A enthält gemäss Regel 2 $\{x\}$.
3. Das FOLLOW-Set von B enthält gemäss Regel 2 $\{y\}$.
4. Das FOLLOW-Set von B enthält gemäss Regel 4 das FOLLOW-Set von A: $\{x\}$.
5. Das FOLLOW-Set von C enthält gemäss Regel 2 das FIRST-Set von B ohne ε: $\{3, 4\}$.
6. Das FOLLOW-Set von C enthält gemäss Regel 3 das FOLLOW-Set von A: $\{x\}$.
7. Das FOLLOW-Set von C enthält gemäss Regel 4 das FOLLOW-Set von B: $\{x, y\}$.

Obige Lösungsschritte resultieren in folgenden FOLLOW-Sets:

$FIRST(S) = \{\$\}$
$FIRST(A) = \{x\}$
$FIRST(B) = \{x, y\}$
$FIRST(C) = \{3, 4, x, y\}$

3.11 LL(1) Parsetabellen

Damit man eine deterministische Parsetabelle erstellen kann, braucht man eine LL(1) Grammatik. Folgende Kriterien garantieren, dass eine Grammatik nicht LL(1) ist:

- Die Grammatik ist linksrekursiv
- Die Grammatik ist nicht linksfaktorisiert
- Die Grammatik ist mehrdeutig

Dies sind jedoch nicht die einzigen Kriterien. Um sicher zu gehen, dass eine Grammatik LL(1) ist, muss man eine deterministische Parsetabelle aufstellen.

Die Parsetabelle ist horizontal mit den Nicht-Terminalsymbolen sowie mit $ und vertikal mit den Terminalsymbolen beschriftet.

Folgender Algorithmus wird auf jede Produktion $A \to \alpha$ angewendet:

1. Für jedes Terminalsymbol t im FIRST-Set von α gilt: $T[A, t] = \alpha$.
2. Falls ε im FIRST-Set von α ist, gilt für jedes Terminalsymbol im FOLLOW-Set von A: $T[A, t] = \alpha$.
3. Falls ε im FIRST-Set von α ist und $ im FOLLOW-Set von A ist, gilt: $T[A, \$] = \alpha$.

Beispiel

Grammatik:

$S \to Ax \mid By \mid z$
$A \to 1CB \mid 2CB$
$B \to 3B \mid C$
$C \to 4 \mid \varepsilon$

Parsetabelle:

	1	2	3	4	x	y	z	$
S	Ax	Ax	By	By		By	z	
A	$1CB$	$2B$						
B			$3B$	C	C	C		
C			ε	$4 \mid \varepsilon$	ε	ε		

Mithilfe dieser Parsetabelle sieht man, dass die gegebene Grammatik aufgrund Mehrdeutigkeiten nicht LL(1) ist: Im Feld $T[C, 4]$ gibt es zwei mögliche Produktionen, welche man anwenden könnte.

4 Globale Analyse

4.1 Datenflussanalyse

Die Datenflussanalyse verknüpft mit jedem Programmpunkt einen Datenflusswert. Die Datenflusswerte vor und hinter einer Anwendung p sind $IN[p]$ und $OUT[p]$. Die Relation zwischen IN und OUT wird als *Transferfunktion* bezeichnet.

Innerhalb eines Grundblocks mit den Anweisungen $s_1, \ldots, s_n$ gilt: $IN[s_{i+1}] = OUT[s_i]$.

4.2 Liveness-Analyse

Regeln

1. Eine Variable **nach** einem Statement ist lebendig (alive), wenn sie **vor** einem nachfolgenden Statement lebendig ist.

   $$L(p, x, out) = \vee\{L(s, x, in) | s \text{ is a successor of } p\}$$

2. Wenn eine Variable in einem Statement gelesen wird, ist sie **vor** dem Statement lebendig.

   $L(s, x, in) = true$ if s refers to x on the rhs

3. Wenn eine Variable in einem Statement geschrieben, aber nicht gelesen wird, ist sie **vor** dem Statement tot (dead).

   $L(x := e, x, in) = false$ if e does not refer to x

4. Wenn in einem Statement eine Variable weder gelesen noch geschrieben wird, ist die Liveness **vor** dem Statement gleich wie **nach** dem Statement.

   $L(s, x, in) = L(s, x, out)$ if s does not refer to x

Algorithmus

1. Zu Beginn ist die Liveness überall $false$.

2. Wähle ein Statement s, welches eine der Regeln 1-4 nicht erfüllt. Korrigiere die Liveness. Wiederhole diesen Schritt bis alle Statements die Regeln 1-4 erfüllen.

Jede Stelle kann bei diesem Algorithmus höchstens ein mal die Liveness wechseln, zudem kann die Liveness nur von $false$ zu $true$ wechseln, nicht umgekehrt.

5 Optimierung

5.1 Lokale / Globale Optimierung

Lokale Optimierung ist unabhängig vom globalen Zustand eines Programmes.

Globale Optimierung hängt von der Kenntnis einer Eigenschaft *P* zu einem bestimmten Zeitpunkt der Programmausführung ab. Damit die Eigenschaft *P* zu *jedem* Ausführungszeitpunkt überprüft werden kann, muss das gesamte Programm bekannt sein. Aus Sicherheitsgründen sollte man deshalb eher konservativ optimieren. Im Zweifelsfall ist es besser nichts zu tun.

5.2 Algebraic Simplifications

Einige algebraische Anweisungen können vereinfacht werden. Beispiel:

```
X := 0 * A              X := 0
Y := B ** 2      ⇒      Y := B * B
Q := C * 8              Q := C << 3
```

5.3 Constant Folding

Ausdrücke mit konstanten Werten werden zur Compile-Zeit evaluiert. Beispiel:

```
X := 3 * 2 + 4      ⇒      X := 10
```

5.4 Constant Propagation

Konstanten werden wo immer möglich eingesetzt. Beispiel:

```
X := 3          ⇒       X := 3
Q := X + Y              Q := 3 + Y
```

5.5 Copy Propagation

Falls in einem Block `W := X` vorkommt, kann man in allen folgenden Einsätzen von `W` anstelle von `W` gleich `X` verwenden. Beispiel:

```
X := 7          ⇒       X := 7
Q := A + X              Q := A + 7
```

5.6 Dead Code Elimination

Ungenutzer Code wird entfernt. Beispiel:

```
X := 3
Y := Z * W      ⇒       Y := Z * W
Q := 3 + Y              Q := 3 + Y
```

6 Registerallokation

Die Registerallokation gehört zu den globalen Optimierungen. Sie hat zum Ziel, möglichst viele der Variablen so lange wie möglich in den Prozessorregistern zu halten. Hat man mehr Variablen zur selben Zeit als Register, muss man Werte auslagern (spillen).

6.1 Einfacher Top-Down Ansatz

Die Register werden nach Anzahl Verwendungen der Variablen vergeben.

Algorithmus

1. Lies das gesamte Programm (linear in Zeit) und bestimme, wie oft jede Variable verwendet wird. Jede Variable erhält eine Priorität gemäss der Häufigkeit ihres Auftretens.
2. Sortiere die Variablen gemäss Prioritäten.
3. Ordne den Variablen Register zu.
   - Falls genug Register vorhanden sind: jede Variable bekommt ein Register.
   - Sonst: verteile gemäss Priorität und reserviere Register für das Einlagern und Auslagern aus dem Memory.
4. Schreib den modifizierten Code wieder in eine Datei.

Vorteile

- Sehr einfach.
- Funktioniert, falls genug Register vorhanden sind.

6.2 Einfacher Bottom-Up Ansatz

Von unten nach oben versucht man einfach beim Auftreten einer Variable zu bestimmen:

- Ist die Variable in einem Register? Wenn ja: okay!
- Wenn nein: ist ein Register frei? Wenn ja: okay!
- Wenn nein: spill. Es wird die Variable ausgelagert, welche am längsten nicht mehr benötigt wird.

6.3 Linear Scan

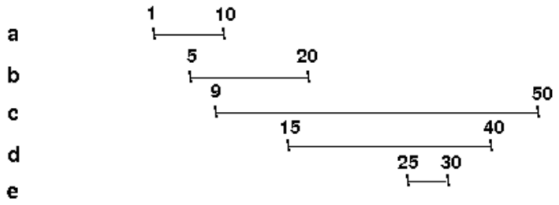
Algorithmus

1. Liveness-Analyse $\Rightarrow$ Liste mit Live Intervallen
2. Sortiere gemäss Startpunkt der Live Intervalle.

3. Intervall-Analyse:

   a) Prüfe, ob Variablen nicht mehr benötigt werden. Falls ja: gib diese Register frei.

   b) Falls genügend Register vorhanden sind: weiterfahren (Register zuordnen). Falls zuwenig Register vorhanden sind: Spilling (beispielsweise jene Variablen, welche den spätesten Endpunkt haben).



6.4 Register Interference Graph

Zwei temporäre Variablen, welche gleichzeitig lebendig sind, können nicht das selbe Register belegen.

Register Interference Graph (RIG / IG):

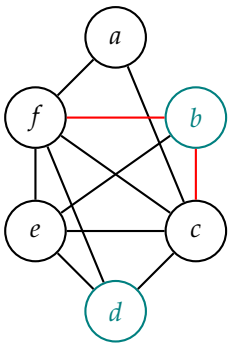
- Knoten repräsentieren (temporäre) Variablen.
- Kanten verbinden Knoten / Variablen, welche gleichzeitig lebendig sind.
- Zwei Variablen können das selbe Register belegen, falls sie keine verbindende Kante haben.

Beispiel

Live Sets:

```
{b,c,f}
{a,c,f}
{c,d,f}
{c,d,e,f}
{c,e}
{b,c,e,f}
{c,f}
{b}
```

Graph:



Schlussfolgerung:

- `{b,c,f}`: Die Variablen `b`, `f` und `c` können nicht im selben Register gespeichert werden.
- Die Variablen `b` und `d` können das selbe Register verwenden.

6.5 Graph Coloring

Graph Coloring ordnet den Knoten eines Graphen Farben zu. Benachbarte Knoten müssen unterschiedliche Farbe haben. Ein Graph ist `k`-färbbar, falls er mithilfe von `k` Farben gefärbt werden kann.

Das Problem des Register Interference Graphs kann auf das Graph Coloring Problem abgebildet werden. Dabei entsprechen die Farben den Registern und `k` ist die Anzahl Maschinen-Register.

Falls der RIG `k`-färbbar ist, dann existiert eine Registerdarstellung mit nicht mehr als `k` Registern.

Das Graph Coloring Problem (und somit das Register Allocator Problem) ist NP-Hart. Es gibt jedoch heuristische Lösungsverfahren, wie z.B. der Chaitin Algorithmus.

Wenn der optimistische Chaitin Algorithmus schief geht, müssen wir Variablen spillen. Auch für die Auswahl der auszulagernden Variablen gibt es nur heuristische Ansätze.

7 Weiterführende Links

Ausgezeichneter Online-Kurs zum Compilerbau von der Stanford Universität:

`https://class.coursera.org/compilers-2012-002/lecture/`

Einige Handouts zum *Dragon Book* von Stanford:

`http://dragonbook.stanford.edu/lecture-notes/Stanford-CS143/03-Lexical-Analysis.pdf`
`http://dragonbook.stanford.edu/lecture-notes/Stanford-CS143/06-Formal-Grammars.pdf`
`http://dragonbook.stanford.edu/lecture-notes/Stanford-CS143/07-Top-Down-Parsing.pdf`
`http://dragonbook.stanford.edu/lecture-notes/Stanford-CS143/08-Bottom-Up-Parsing.pdf`
`http://dragonbook.stanford.edu/lecture-notes/Stanford-CS143/09-SLR-Parsing.pdf`
`http://dragonbook.stanford.edu/lecture-notes/Stanford-CS143/11-LALR-Parsing.pdf`
`http://dragonbook.stanford.edu/lecture-notes/Stanford-CS143/12-Miscellaneous-Parsing.pdf`
`http://dragonbook.stanford.edu/lecture-notes/Stanford-CS143/13-Syntax-Directed-Translation.pdf`
`http://dragonbook.stanford.edu/lecture-notes/Stanford-CS143/14-Semantic-Analysis.pdf`
`http://dragonbook.stanford.edu/lecture-notes/Stanford-CS143/15-Runtime-Environments.pdf`
`http://dragonbook.stanford.edu/lecture-notes/Stanford-CS143/16-Intermediate-Rep.pdf`
`http://dragonbook.stanford.edu/lecture-notes/Stanford-CS143/18-Processor-Architectures.pdf`
`http://dragonbook.stanford.edu/lecture-notes/Stanford-CS143/19-Final-Code-Generation.pdf`
`http://dragonbook.stanford.edu/lecture-notes/Stanford-CS143/20-Optimization.pdf`