Compilerbau

0.1 **Compiler** = Programm zur Übersetzung Programmiersprache → Maschinensprache.

→ imperative Programme (Kontrollstrukturen, Variablen, Zuweisungen, Datenstrukturen)
(nicht: deklarative " , nebenläufige P., objektorientierte P.)

MS: von-Neumann-Rechner: Reduced (RISC), Complex (CISC)

0.2 **Aspekte:** **Syntax:** formaler hierarchischer Aufbau
Semantik: Bedeutung, Zustände
Pragmatik: Formulierung, natürliche Sprache, Maschinen-Abh.

äquivalent = semantisch äquivalent.

0.3 **Struktur des Compilers**

n-Pass-Compiler: n = Läufe durch Programm

RegEx, CFG

lexikal. Analyse	Scanner →	lexeme → Symbole ~~(erst lexeme)~~	→ (Token + Attrib.)
syntakt. "	→	Ableitungsbaum	↑ syntaktische Analyse
semant. "	→	attributierter "	↑ semantische Analyse
Frontend: Zwischencode-Erz.	→	Zwischencode	
Backend: Codeoptimierung	→	"	
Backend: Code-Erzeugung	→	Maschinencode	

1. Lexikalische Analyse

Quellcode —Scanner (Regex)— Lexeme → Symbole (Token + Attr.) (auf Einmal)

Symbolklassen: ~~Be~~

- Bezeichner	Var1
- Zahlwörter	55
- Schlüsselwörter	begin
- Einfache Symbole	+
- Zusammengs. S.	:=
- Leerzeichen	␣
- Spezielle Symb.	/* */

Attribute:
- Zeiger in Symb. Tab.
- Binärdarst. Zahl
- leer

Σ_0	Zeichensatz
$a \in \Sigma_0$	Zeichen
$P \in \Sigma_0^*$	Quellprogramm
$RegE(\Sigma)$	Regex. E
$[\![\ ]\!]$	Abb. Regex → matchende Zeichenfolgen.
$\mathcal{A}$	Automat
Q	Zustandsmenge
Σ	Alphabet
δ	Transitionsfunktion
$\subseteq Q$ q_0	Anfangszustand
$\subseteq Q$ F	Endzustandsmenge.
Σ_ε	$\Sigma \cup \{\varepsilon\}$
$\subseteq Q$ T	
$\varepsilon(T)$	ε-Hülle von T
$L(\mathcal{A})$	durch $\mathcal{A}$ erkannte Sprache

$\mathcal{A} = \langle Q, \Sigma, \delta, q_0, F \rangle \in NFA(\Sigma)$

1.1. Scanner-Konstruktion: RegEx.

- $[\![\cdot]\!]$ = alle Zeichenfolgen, die die RegEx "." matcht.
- alle Einzelzeichen sind Regexe, sowie $(\alpha \vee \beta)$, $(\alpha \cdot \beta)$, (α^*)
- $[\![\alpha \vee \beta]\!] := [\![\alpha]\!] \cup [\![\beta]\!]$ // $[\![\alpha \cdot \beta]\!] := [\![\alpha]\!] \cdot [\![\beta]\!]$ //...

1.1.1. Das einfache Matching-Problem

Wird Zeichenfolge durch RegEx gematcht?
Entscheide mit NFA (nondeterministic finite automata).
- ε-Hülle: ohne Eingabe erreichbare Zustände.
- erweiterte Transitionsfunktion: Zustände, die durch Eingabewort erreichbar sind.

- DFA-Methode: RegEx —Thompson→ NFA —Potenzmengen-Konstruktion→ DFA.
  → Entscheidung in $|w|+1$ Schritten.
- NFA-Methode: Durchlauf durch NFA mit Wort, auf allen möglichen Wegen.
  → Platz: $O(|\alpha| + |w|)$, Zeit: $O(|\alpha| \cdot |w|)$ „DFA on the fly erzeugen"

1.1.2. Das erweiterte Matching-Problem : ~~matcht es auf irgendeine RegEx~~

Problem: Zerlegung des Eingabeprogramms (wort) finden, eindeutig!
Lösung: fkm-Analyse: Längster Match, im Zweifel Automat höchster Priorität.
Berechnung:
1. Konstruiere einen DFA je RegEx.
2. Mache daraus Produktautomaten, und zwar so, dass jeder Endzustand die Klasse des niedrigsten matchenden Automaten hat.
   (z.B.: Automat 3 und 5 würden matchen → Endzustand hat Klasse 3.)

Produktiver Zustand = Zustand, aus dem Endzustand erreichbar ist.

Leseband mit Backtrack-Kopf und LookAhead-Kopf.
- Normal Mode: Match suchen. Bei Endzustand → Backtrack-Mode.
  (entlang produktiver Z.) Bei nicht produktivem Zustand: lexerr.
- Backtrack Mode: längsten Match suchen. Bei Endzustand → Backtrack-Fahne neu setzen.
  (entlang Produktiver Z.). Bei nicht produktivem Z. → zurück zur Fahne.
- Eingabeende: Endzustand → Token ausgeben.
  Prod. Zustand → lexerr
  $O(|w|^2)$ Prod. Zustand im Backtrack-Mode → zurück zur Fahne.

1.2. Praktische Aspekte der Scanner-Konstruktion

Lex-Spezifikation $\xrightarrow{lex}$ Scanner in C $\xrightarrow{cc}$ Ausführbarer Scanner

- Pascal: 2-Lookahead, Whitespace → 1-Lookahead
- Attributberechnung: 2. Lauf: Zahlenumwandlungen, Bezeichner-Gleichheit prüfen, Schlüsselwörter erstmal als Bezeichner betrachten.

2. Syntaktische Analyse

... durch Parser. Erstelle Baumstruktur.
Syntaktische Einheiten: Variablen, Ausdrücke, Anweisungen, ...
Beschreibung durch CFG (BNF, EBNF, Syntaxdiagramme)
Erkennung durch Kellerautomat mit Ausgabe. ⇒deterministisch!
(deterministischer Kellerautomat mit input-look-ahead), linearer Bedarf Platz + Zeit.)
- Top-Down-Analyse: Linksanalyse
- Bottom-Up-Analyse: gespiegelte Rechtsanalyse

2.1. Kontextfreie Grammatiken

$A, a, u, \alpha,\ A \to \alpha \qquad S \overset{*}{\underset{\ell,r}{\Rightarrow}} w$

Grammatik G genau dann eindeutig, wenn es genau einen Ableitungbaum gibt,

ℓ-Analyse von α: $S \overset{z}{\underset{\ell}{\Rightarrow}} \alpha$. z = Folge von Regeln i. (r-Analyse analog.)

Syntaxanalyse von α: ℓ/r-Analyse, oder SYNTAX ERROR.
Vorr: • G reduziert, d.h. • NT-Symbole auf rechter Seite erreichbar: $S \overset{*}{\Rightarrow} \alpha A \beta$
• NT-Symbole " linker Seite produktiv: $A \overset{*}{\Rightarrow} w$.

2.2. TopDown-Analyse mit LL(k)-Grammatiken

Automat: • Ableitungsschritte (nicht deterministisch)
• Vergleichsschritte (deterministisch)
$(w, S, \varepsilon) \vdash (\varepsilon, \varepsilon, z)$ (Eingabe, Keller, Ausgabe)
Ziel: Nichtdeterminismus durch k-Lookahead beseitigen

CFG	kontextfreie Grammatik
G	Grammatik
Σ	Eingabealphabet
Π	Kelleralphabet
$1, \dots, p$	Ausgabealphabet
$\Sigma^* \times \Pi \times [p]^*$	Konfigurationsmenge.

$first_k(\alpha)$: {die ersten k Symbole der möglichen Ableitungen}

$X \to i$	$first_3(simon) = \{sim\}$
$X \to a$	$first_3(sXmon) = \{sim, sam\}$

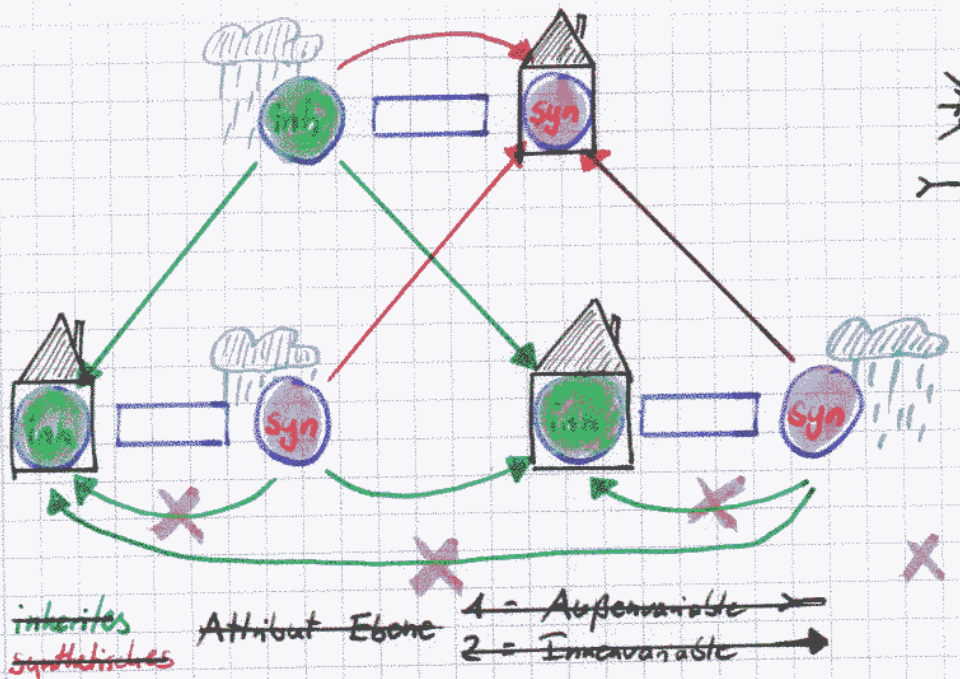
Folgerungen: 1. $first_k(\alpha) \neq \emptyset$, da G reduziert.
2. ε in $first_k \Rightarrow k=0$ oder $\alpha \overset{*}{\Rightarrow} \varepsilon$.
3. $\alpha \overset{*}{\Rightarrow} \beta \curvearrowright first_k(\beta) \subseteq first_k(\alpha)$
4. Wenn man von α nach x kommt, ist die first-Menge von x auch in der first-Menge von α enthalten. z.B.:

LL(k)-Grammatik: Lesen der Eingabe von links nach rechts.
- Linksableitungsschritt für $wA\alpha$ ist durch die nächsten auf w folgenden Symbole bestimmt.
- TD-Analyseautomat kann nun deterministisch mit k-Lookahead arbeiten.

Ziel: Bestimmung der A-Regel aus k-Look-ahead.
„aus den ersten k Zeichen des abgeleiteten Teilwortes ergibt sich, aus welchem Satz es abgeleitet ist.
$S \overset{*}{\underset{\ell}{\Rightarrow}} wA\alpha \overset{\ell}{\Rightarrow} w\beta\alpha \overset{*}{\underset{\ell}{\Rightarrow}} wx$
$\qquad\qquad \overset{\ell}{\Rightarrow} w\gamma\alpha \overset{*}{\underset{\ell}{\Rightarrow}} wy$
G ist LL(k)-Grammatik, wenn $\forall$ Linksabl. gilt: $first_k(x) = first_k(y) \curvearrowright \beta = \gamma$

- Lemma: Die $first_k$-Mengen von 2 möglichen Regeln haben keine Schnittmenge.
- Brauche Rechtskontext (follow-Menge), falls first-Menge ε ist.
- $fi(\beta) = first_k(\beta)$
- $fo(A) = follow_k(A)$
- $ea(A \to \beta) = fi(\beta\, fo(A))$

Berechnung fi-Mengen: auf was können die NT-Symbole ~~der rechten Seiten~~ abgeleitet werden?
Berechnung fo-Mengen: was kann hinter den NT-Symbolen der rechten Seite kommen?
Berechnung ea-Mengen: fi-Menge der rechten Seite. Wenn ε dabei, dann fo-Menge der linken Seite dazunehmen.



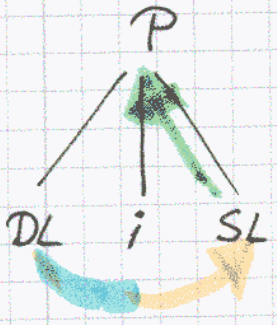
X = nicht erlaubt bei L-Attributgraum.

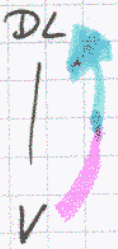
- inherite/synthetische Attribute sind disjunkt.
- Innen-/Außen- Variablen sind disjunkt.

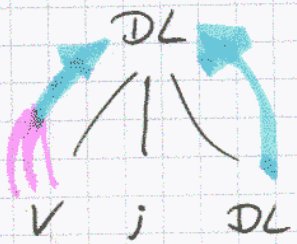
	inherit	synthetisch
Ebene 1	A	I
Ebene 2	I	A

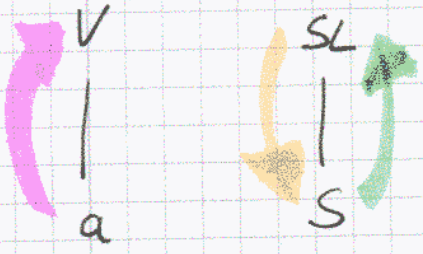
Zirkularitäten beim Verkleben.

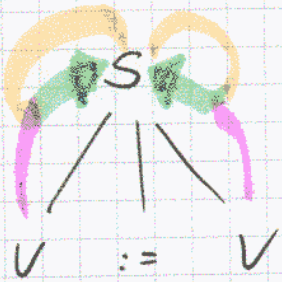
D(A, t) = Abhängigkeiten in der Wurzel A (für Baum t)



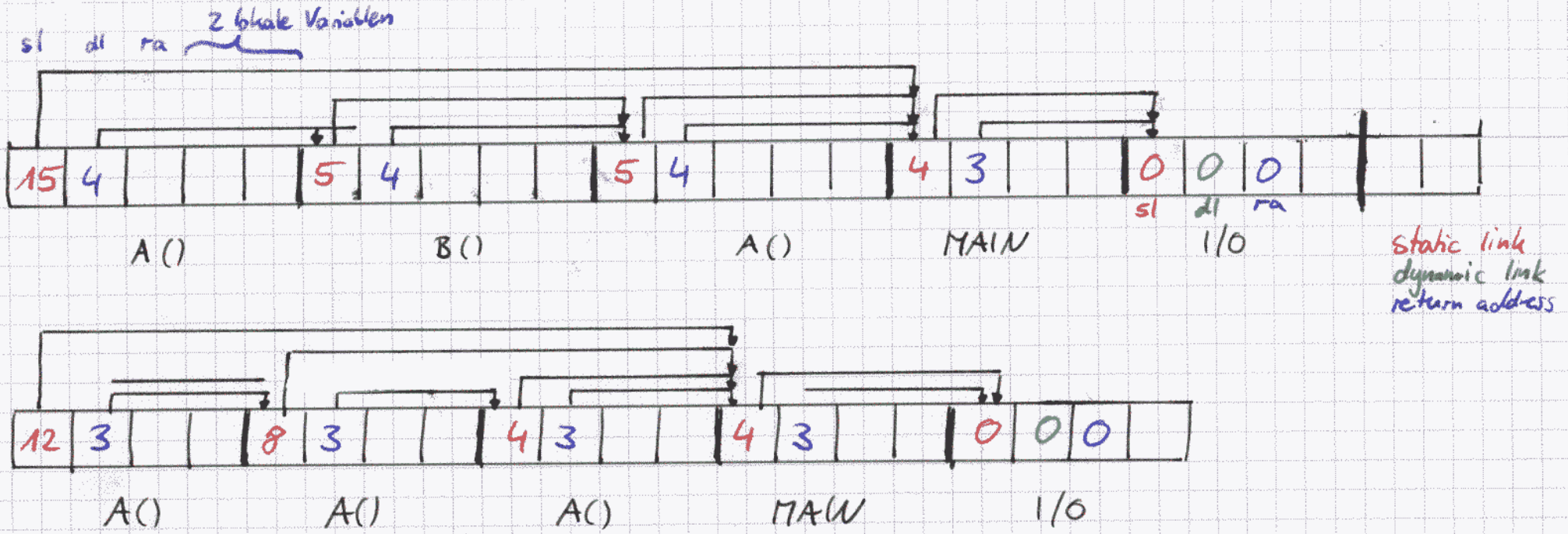








- DECL
- ENV
- DV
- V


sl dl ra
2 lokale Variablen
15 4 5 4 5 4 4 3 0 0 0
sl dl ra
A() B() A() MAIN I/O
Static link
dynamic link
return address
12 3 8 3 4 3 4 3 0 0 0
A() A() A() MAIN I/O

Symbol - Tabelle

			(Wert)		
Tisch	→	const,	5		
Stuhl	→	const,	7		
			(Level)	(Offset)	
Papier	→	var,	2	5	
Stift	→	var	1	3	
			(Adresse)	(Level)	(Anzahl Variablen) (Offset) i.d. Proz.
Haus	→	proc	1002	2	5
Boot	→	proc	1005	1	3