

Kapitel 2

Lexikalische Analyse

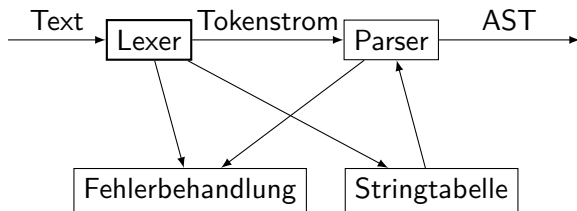
Kapitel 2: Lexikalische Analyse

- 1 Eingliederung in den Übersetzer / Zielvorgaben
- 2 Theoretische Grundlage: Endliche Automaten
- 3 Implementierung
 - Implementierung endlicher Automaten, Tabellenkompression
 - Der Tokenstrom
 - Implementierung der Stringtabelle
- 4 Einsatz von Generatoren
 - JLex/Flex

lexikalische Analyse: Aufgabe

- zerlegt Quellprogramm (*Text*) in Sequenz bedeutungstragender Einheiten (*Tokenstrom*)
- beseitigt überflüssige Zeichen(folgen) wie
 - Kommentare,
 - Leerzeichen, Tabulatoren usw.
- Modell: endlicher Automat aus programmieretechnischen Gründen: Geschwindigkeit höher

Eingliederung in den Übersetzer



Warum getrennte lexikalische Analyse?

- Durchschnittliche Komplexität einer Anweisung (Knuth und andere¹):
var := var + const;
 $\text{const} \in \{-1, 0, 1\}$
- 6 Symbole, aber ca. 40–60 Zeichen, viele (10–40) Leerzeichen pro Zeile wegen Einrückungen und Kommentaren
 - Informationskompression größer als in allen anderen Teilen des Übersetzers
 - deshalb Abtrennung
 - **Aber:** Kompression nur in Anzahl Symbolen, nicht unbedingt in Anzahl Bytes!
- Heute Geschwindigkeit der lexikalischen Analyse eher irrelevant (<5% der Compilerlaufzeit).
- Trennung aber Softwaretechnisch sinnvoll: Entkopplung, Modularisierung.

¹D.E. Knuth: An Empirical Study of FORTRAN Programs, Software P&E, 1(1971), 105-134

Weitere Gründe

- Beobachtung zeigt, dass bei modernen Programmiersprachen endliche Automaten ausreichen
- Umfangreiche Eingabe, daher effiziente Hilfsmittel (endl. Automat schneller als Kellerautomat)
- Sich selbst erfüllende Prophezeiung:
Weil endliche Automaten ausreichen, sind moderne Programmiersprachen so formuliert, dass endliche Automaten ausreichen.
- Ausnahmen:
 - endl. Automat mit Rücksetzen am Ende:
1.E1 \rightarrow 1.E1,
1.EQ. \rightarrow 1 .EQ.,
else 1. E...
(Fortran, other languages: +=, =:=, ...)
 - mehrere Automaten, gesteuert vom Parser: Fortran Formate
(Text ohne Anführungszeichen!)

Beim Sprachentwurf: Unabhängigkeit Lexer/Parser als Ziel

Beispiel: Ausnahmen

- Fortran 77
READ 5,ggg,...
ggg ist eine Formatanweisung, kein Bezeichner
- C
#pragma ...
Pragma in C: ... kann beliebiges enthalten; ist also insbesondere nicht im Sprachstandard definiert
- Pascal
(*D ...*)
Pragma in Pascal
- Zeichenketten in vielen Programmiersprachen

ADT Lexer

Gelieferte Operationen:

- `next_token`

Benötigte Operationen:

Eingabe:

- `next_char` oder `read_file`

Stringtabelle:

- `insert(text, key)`
- `find_or_insert(text) : (key,value)`
- `get_text(key) : text`

Fehlerbehandlung:

- `add_error(nr, text)`

Token können identifiziert werden durch:

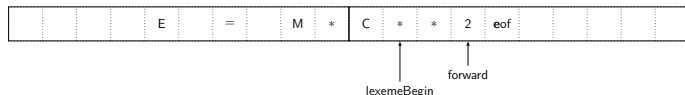
- Endzustand im Automaten (für jedes Token ein eigener Endzustand)
- Stringtabellen (durch Vergleich mit deren Einträgen)
- Hybrider Ansatz
 - Wortsymbole (z.B. „if“, „class“) und Bezeichner (z.B. Variablen- oder Funktionsnamen) erst in Stringtabelle unterschieden
 - andere Tokens mit unterschiedlichen Endzuständen

Zielvorgaben

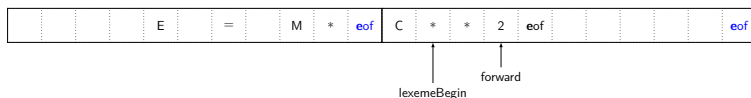
- soll höchstens 6%–10% der Gesamtlaufzeit eines nicht-optimierenden Übersetzers benötigen
- 15% einschl. syntaktischer Analyse
- Hauptaufwand: Einlesen der Quelle
 - zeichenweise: zu langsam wegen Prozeduraufruf/Systemaufruf für jedes gelesene Zeichen, nur bei Lesen von Tastatur
 - zeilenweise: Zeilen unbeschränkter Länge bei generiertem Code!? doppeltes Lesen wegen Suche nach Zeilenwechsel, mehrfaches Kopieren von Puffern
 - gepufferte Eingabe
 - Heutzutage: komplette Datei in virtuellen Hauptspeicher: hoher Speicherbedarf bei vielen offenen Dateien

Implementierung der zeichenweise Eingabe (1/2)

- Verwende 2 Puffer der Größe N (z.B. 4096 Byte)



- `lexemeBegin` zeigt auf Beginn des aktuellen Strings
- `forward` zeigt auf aktuelles Zeichen
- Verwende zusätzliche Wächter um Überlauf zu erkennen



Implementierung der zeichenweise Eingabe (2/2)

```
switch (*forward++) {  
    case eof:  
        if (forward is at end of first buffer) {  
            reload second buffer;  
            forward = beginning of second buffer;  
        }  
        else if (forward is at end of second buffer) {  
            reload first buffer;  
            forward = beginning of first buffer;  
        }  
        else /* eof within a buffer marks the end of input */  
            terminate lexical analysis;  
        break;  
    /* Cases for the other characters */  
}
```

Kapitel 2: Lexikalische Analyse

- 1 Eingliederung in den Übersetzer / Zielvorgaben
- 2 Theoretische Grundlage: Endliche Automaten
- 3 Implementierung
 - Implementierung endlicher Automaten, Tabellenkompression
 - Der Tokenstrom
 - Implementierung der Stringtabelle
- 4 Einsatz von Generatoren
 - JLex/Flex

Definition endlicher Automat

Ein endlicher Automat A ist ein Quintupel $(S, \Sigma, \delta, s_0, F)$, so dass:

- Σ ist das Eingabealphabet, eine endliche Menge von Symbolen
- $S \neq \emptyset$ ist eine endliche Menge von Zuständen
- Anfangszustand $s_0 \in S$
- $F \subseteq S$ sind die Endzustände
- $\delta : S \times \Sigma \rightarrow 2^S$ Übergangsrelation
- Elemente in δ : $sx \rightarrow s'$; $s \in S$, $x \in \Sigma$, $s' \in 2^S$

A akzeptiert die Zeichenketten $L(A) = \{\tau \in \Sigma^* \mid s_0\tau \xrightarrow{*} s, s \in F\}$.

Die Automaten A, A' sind äquivalent gdw. $L(A) = L(A')$.

A ist deterministisch, wenn δ eine (partielle) Funktion ist, d. h. zu jedem Zustand und jeder Eingabe höchstens ein Folgezustand existiert.

Prinzip des längsten Musters

Wann hört der Automat auf?

- Prinzip: der Automat liest immer so weit, bis das gelesene Zeichen nicht mehr zum Token gehören kann
 - bei Bezeichnern: bis ein Zeichen erreicht ist, das kein Buchstabe oder Ziffer (oder Unterstrich, . . .) ist
- Konsequenz: der Automat startet mit dem Zeichen, das er beim Vorgängertoken als letztes las
 - *Grundzustand: ein Zeichen im Puffer*

Lexikalische Analyse in Fortran 77

Problem

- Zwischenräume können auch in Symbolen vorkommen,
- Symboleinteilung abhängig davon, ob Anweisung mit Wortsymbol beginnt
- Dies kann erst später entschieden werden:
 - `DO 10 I = 1.5` ist Zuweisung an Variable `DO10I`
 - `DO10I = 1,5` ist Schleifensteuerung (Zähler `I`, Endmark `10`)

Verfahren

- Lies gesamte Anweisung. Anweisung beginnt mit Wortsymbol, wenn
 - Anweisung kein Gleichheitszeichen (außerhalb von Klammern) enthält
 - nach einem Gleichheitszeichen ein Komma (außerhalb von Klammern) folgt
 - ...
- Andernfalls ist die Anweisung eine Zuweisung, beginnend mit einem Bezeichner

Reguläre Ausdrücke

Gegeben: Vokabular V sowie die Symbole $\varepsilon, +, *, (,), [,]$, die nicht in V enthalten sind. Eine Zeichenkette R über V ist ein regulärer Ausdruck über V , wenn:

- R ist ein einziges Zeichen aus V oder Symbol ε , oder
- R hat die Form $(X), X + Y, XY, X^*, X^+, [X]_m^n$, wobei X und Y reguläre Ausdrücke sind
- Klammern können weggelassen werden:
 - * hat höchste Priorität, + eine niedrigere als Konkatination.

Satz:

Für jeden regulären Ausdruck R existiert ein endlicher Automat A , so dass $L(A) = L(R)$.

Beispiel: Konstruktion eines endlichen Automaten

Regulärer Ausdruck:

$(b((b + z))^*)$

Einfügen von Zuständen:

$_0(b_1((b_2 + z_3))^*)$

Übergangsregeln zwischen allen benachbarten Zuständen:

$0b \rightarrow 1, 1b \rightarrow 2, 1z \rightarrow 3, 2\varepsilon \rightarrow 1, 3\varepsilon \rightarrow 1$

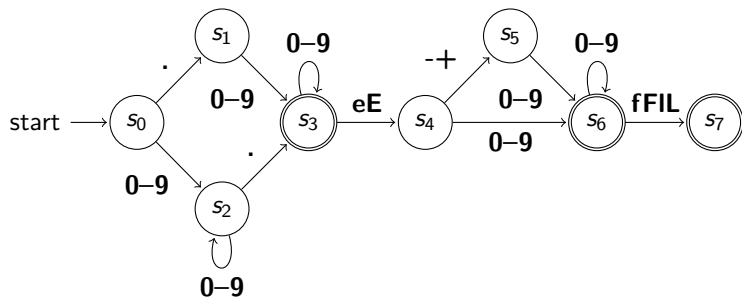
Hinweis: Es ist auch möglich die ε -Übergänge sofort zu eliminieren. (vgl. Vorlesungen über Informatik, Band 1, S. 91ff.)

Automat für Float-Konstanten in C²

Regulärer Ausdruck

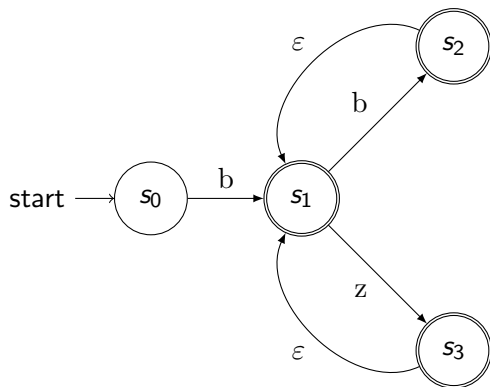
$$((0 + \dots + 9)^* \cdot (0 + \dots + 9)^+) + ((0 + \dots + 9)^+ \cdot)$$
$$(\epsilon + ((e + E)(+ + - + \epsilon)(0 + \dots + 9)^+))(\epsilon + f + F + l + L)$$

Automat



²siehe ISO/IEC 9899:1999 §6.4.4.2

Beispiel: Endlicher Automat



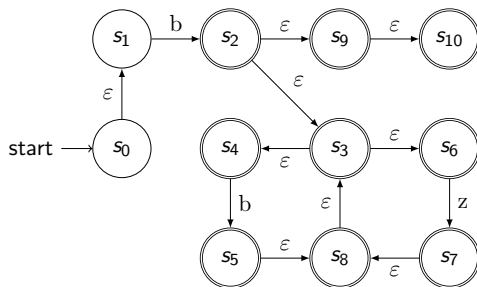
⊙ Endzustand ○ Durchgangszustand start → ○ Startzustand

Teilmengenkonstruktion

- *Teilmengenkonstruktion macht endl. Automaten deterministisch*
- Zustände in A' sind Mengen von Zuständen in A
 - Initial: $s'_0 = \{s_0\}$
 - ε -Übergänge:
Wenn $s_i \varepsilon \rightarrow_A s_j$ und $s_i \in s'$ dann $s' := s' \cup \{s_j\}$
 - Sonstige Übergänge:
Wenn $s_i a \rightarrow_A p_j$ dann erweitere $\{s_1, \dots, s_k\} a \rightarrow_{A'} \{p_1, \dots, p_l\}$
 - Endzustände:
Wenn $s_i \in s'_E$ Endzustand in A dann s'_E Endzustand in A'
- Komplexität praktisch linear, theoretisch (in pathologischen Fällen) exponentiell.
- pathologisch z.B.: $(a + b)^* a (a + b)^{n-1}$

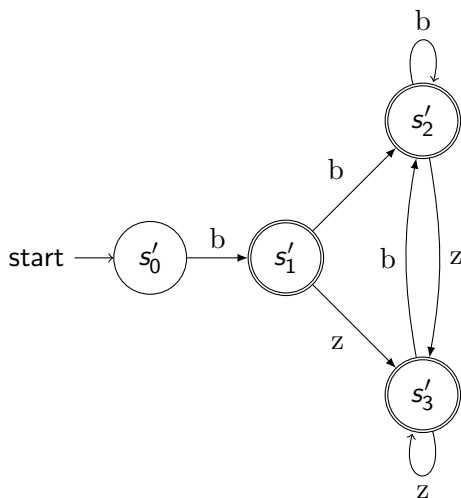
Beispiel: Teilmengenkonstruktion

Nichtdeterministischer Automat:



- Teilmengen: $0' = \{0, 1\}$, $1' = \{2, 3, 4, 6, 9, 10\}$,
 $2' = \{5, 8, 9, 10, 3, 4, 6\}$, $3' = \{7, 8, 9, 10, 3, 4, 6\}$
Anmerkung: Alter Zustand kann in mehreren neuen
vorkommen (z.B. 3)
- Übergänge: $0'b \rightarrow 1'$, $1'b \rightarrow 2'$, $1'z \rightarrow 3'$, $2'b \rightarrow 2'$, $2'z \rightarrow 3'$,
 $3'z \rightarrow 3'$, $3'b \rightarrow 2'$
- Endzustände: $1'$, $2'$, $3'$

Beispiel: Teilmengenkonstruktion (resultierender Automat)



Anmerkung: Dies ist nicht der einfachste mögliche Automat, sondern nur ein deterministischer.

Äquivalenzklassenbildung

Äquivalenzklassenbildung macht endl. Automaten minimal

Initial:

- $\{s_1, \dots, s_k\} = s'$ und $\{p_1, \dots, p_l\} = s_E$
- p_1, \dots, p_l Endzustände, s_1, \dots, s_k keine Endzustände
- wir betrachten die initialen Äquivalenzklassen $\{s_1, \dots, s_k\}$ und $\{p_1, \dots, p_l\}$

Rekursion:

- $s_i \equiv_{k+1} s_j$, wenn $s_i a \rightarrow p_i$, $s_j a \rightarrow p_j$, s_i, s_j sowie p_i, p_j jeweils in gleicher Klasse nach k Rekursionen, $k = 0, 1, 2, \dots$ bzw. $s_i a \rightarrow s_i$, $s_j a \rightarrow s_j$ und s_i, s_j in gleicher Klasse. Analog wird $p_i \equiv_{k+1} p_j$ definiert

Abschluß:

- Klasseneinteilung ändert sich nach m Schritten nicht mehr, $m < \max(l, k)$
- Äquivalenzklasse je ein Zustand, Übergänge und Endzustände wie bei Teilmengenkonstruktion

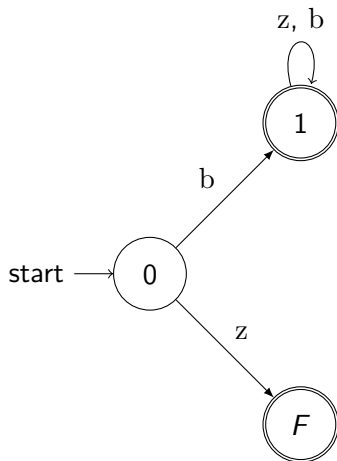
Beispiel: Äquivalenzklassenbildung

Deterministischer Automat (nicht minimal):

$0b \rightarrow 1, 1b \rightarrow 2, 1z \rightarrow 3, 2b \rightarrow 2, 2z \rightarrow 3, 3z \rightarrow 3, 3b \rightarrow 2$

- Zustände $\{0\}$ Endzustände $\{1, 2, 3\}$ Fehler $\{F\}$
- Partitionierung?
 $\{0\}b \rightarrow \{1, 2, 3\}, \{0\}z \rightarrow \{F\}, \{1, 2, 3\}b \rightarrow \{1, 2, 3\},$
 $\{1, 2, 3\}z \rightarrow \{1, 2, 3\}$
- Keine weitere Partitionierung

Beispiel: Äquivalenzklassenbildung (resultierender Automat)



Aufwand

- Deterministisch machen:
 - exponentiell bei pathologischen Beispielen
 - in der Praxis weniger als quadratisch
- Minimierung:
 - quadratisch
 - Es gibt auch $O(n \log n)$ -Algorithmen, praktisch nicht bewährt, Programmlogik kompliziert, die meisten unserer Automaten sind klein, daher nicht notwendig
- Automat unvollständig: minimaler Automat nicht eindeutig, Aufwand NP-vollständig
- Beispiel exponentieller Aufwand der Teilmengenkonstruktion:

$$(a + b)^* a (a + b)^{n-1}$$

Anzahl Zustände des deterministischen Automaten $\geq 2^n$

End- und Fehlerzustände

- Endzustand entsteht durch die Regel $A \rightarrow a$
- Jeder Fehlerzustand ist Endzustand
- Bei Minimierung müssen die End- und Fehlerzustände erhalten bleiben
 - Äquivalenzklassenbildung beginnt mit $n + 2$ Klassen
 - Anzahl der Endzustände: n
 - Ein Fehlerzustand
 - Eine Klasse aller anderen Zustände
- Beachte: Eigentlich sind alle Automaten auf den Folien und in Übersetzerbaubüchern falsch, wenn der Fehlerzustand nicht enthalten ist. Dies ist aber Konvention.

Kapitel 2: Lexikalische Analyse

- 1 Eingliederung in den Übersetzer / Zielvorgaben
- 2 Theoretische Grundlage: Endliche Automaten
- 3 Implementierung**
 - Implementierung endlicher Automaten, Tabellenkompression
 - Der Tokenstrom
 - Implementierung der Stringtabelle
- 4 Einsatz von Generatoren
 - JLex/Flex

Tabellendarstellung endlicher Automaten

Ziel: Effiziente Ausführung eines endlichen Automaten

- Ermitteln von Übergängen in $O(1)$

Alternativen

- Adjazenzliste: $O(\log(k))$, k maximaler Ausgangsgrad
- Adjazenzmatrix: $O(1)$ mit kleiner Konstante
- Ausprogrammieren (mit Fallunterscheidung): $O(1)$, aber keine Sprungvorhersage möglich

Größe der Adjazenzmatrix = $|S| * |\Sigma|$

- Für klassische Alphabete ist $|\Sigma| = 256$. ~ 40 echte Zeichen, alle andern führen in den Fehlerzustand
- $|S| \sim 100$
- Problem Speicherbedarf

Beispiel für Tabelle

Tabelle:

Zustand	<i>b</i>	<i>z</i>	<i>Trennzeichen</i>
0	1	<i>fehler</i>	0
1	1	1	<i>Ende</i>

```
int state = 0;
```

```
char cur;
```

```
while (!isFinal(state)  
      && !isError(state))
```

```
{
```

```
    cur = next_char();
```

```
    state = table[state, cur];
```

```
}
```

```
if (isError(state))
```

```
    return ERROR;
```

```
return find_or_insert(text());
```

Achtung: Wir haben hier nur einen Endzustand.

Beispiel für Programm

```
int state = 0; char cur;
while (true) {
    cur = next_char();
    switch(state) {
    case 0:
        switch(cur) {
            case b: state = 1; break;
            case z: return ERROR;
            case Trennzeichen: state = 0; break;
        } break;
    case 1:
        switch(cur) {
            case b, z: state = 1; break;
            case Trennzeichen: return find_or_insert(text());
        } break;
    }
}
```


Tabelle vs. Programm: Bewertung

- Pragmatisch: Generatoren können Tabellen besser verwenden
- Programmierte Version schneller und kleiner
- Tabelle übersichtlicher, systematischer, änderungsfreundlicher, aber langsamer
Begründung: Tabelle ist implementiert durch Schleife mit Abfrage nach allen Eventualitäten, führt zu nicht vermeidbaren Leerprozeduren
- Erfahrung: Programmcode in beiden Fällen ähnlich groß, Tabelle kommt extra dazu

Tabellenkomprimierung

- partitioniere Σ in Äquivalenzklassen von Zeichen die stets im gleichen Kontext benutzt werden.
- lege „ähnliche“ Spalten zusammen: Benutze „neue“ Zeichen J für Übergänge im endlichen Automaten
- Optimiere **nach** deterministisch Machen und Minimieren
- erfordert zusätzliche Indirektion zur Laufzeit
- Kompression reduziert Tabelle auf 5 bis 10% der ursprünglichen Größe
- Synergetische Effekte durch Prozessorcachel

Pragma: Kommentar zur Steuerung der Übersetzung kann eigene, nicht reguläre Syntax enthalten. Behandlung nicht allein durch lexikalische Analyse möglich.

Vorgehen bei Pragmas:

- Pragmatext als Eintrag in Stringtabelle
- Sonderbehandlung (Entschlüsselung während lexikalischer Analyse oder danach?)
- abhängig von Implementierung der Tokens

schwieriges Problem, hier nicht weiter behandelt

Entschlüsseln von UNICODE

Wozu UNICODE?

- zur Internationalisierung (z.B. mit Resource-Dateien)
- Standards wie XML, Java usw. erfordern es

Probleme: $|\Sigma| = 2^{16}$ bei UTF-16, bei UTF-8 variable Zeichenlänge.
Meist zu hoher Speicher- und Laufzeitbedarf für deterministisch
Machen und Minimieren (Generatorlaufzeit)

Lösung:

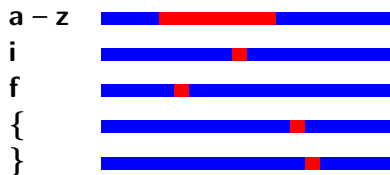
- Partitioniere Σ vorher
- Grundidee:
reguläre Ausdrücke $(X + Y)$, (XY) , $(X)^*$ mit Zeichenmengen
 $X, Y \subseteq \Sigma$

Beobachtung: bedeutungstragend weiterhin nur ca. 40
Zeichenmengen

Vorgehen

Jede Zeichenmenge Z zerlegt Σ in Z und $\Sigma \setminus Z$. Für eine Menge M von Zeichenmengen Z bestimme die induzierte Partition von Σ .

Zeichenmengen M :



Durch M induzierte Partition:



Algorithmus:

- Stelle jedes Z als Intervall $[b, e)$ dar
- sortiere nach Anfangswert b
- Arbeite sortierte Liste ab und erstelle Partitionierung
- Ersetze Zeichenmengen durch Partitionsnummern J
- Erzeuge und benutze den Automaten wie bekannat (mit Abbildung $M \rightarrow J$)

- tokenstrom = Strom(Token)
- Token = (Key, Value, Position)
- Key:
 - das syntaktische Terminalsymbol des Parsers, definiert durch den Endzustand der lexikalischen Analyse oder vordefiniert für reservierte Bezeichner, z.B. Wortsymbole
- Value:
 - Für Bezeichner und Konstanten: Verweis auf den Eintrag in die Stringtabelle, mit dem man zumindest Text und Textlänge erhalten kann.
 - sonst: null (Key überflüssig, nicht definiert)

Umfang der Tokens

- Tokenstrom muss nicht tatsächlich als Datenstruktur vorliegen
- Key: Maschinenwortbreite (oft 32 bit)
- Value: Maschinenwortbreite (oft 32 bit)
- Position der Tokens benötigt für Fehlerausgabe
 - Datei, Zeilen- und Spalteninformation (alternativ Zeile und Relativadresse oder nur Relativadresse)
 - 32 bit für Zeile (oder Relativadresse)
 - mind. 16 bit für Spalte
 - Positionierung bei WYSIWYG Editoren, wenn Tabulatoren beliebig definiert werden können?
 - **Achtung:** Zeilenzählung bei generiertem Code problematisch
- **Summe:** etwa 12-16 Bytes pro Token

Tokenstrom: Implementierung

- Anfrage (Funktionsaufruf)
 - Parser ruft Lexer
 - Lexer ruft Parser
- Strom (Pipeline)
- Array (Tokens sind bereits abgelegt)
 - 20% schneller auf heutigen Architekturen wegen Cache
 - das kann sich ändern

Stringtabelle: Ziele und Kriterien

Ziele:

- Die Merkmale bzw. Schlüssel aller Symbole festlegen, die nicht durch Endzustände des Automaten bestimmt sind.
- Bezeichner und Konstanten durch Merkmal einheitlicher Länge codieren.
- Aufbewahrung der Bezeichner- und Konstantentexte für die weitere Bearbeitung und für Fehlermeldungen.
- Ankerpunkt, von dem aus verschiedene Vereinbarungen eines Bezeichners erreichbar sind (für semantische Analyse).

Kriterien:

- Anfangs unbekannte Anzahl von Bezeichnern und Konstanten unbeschränkter (!) Länge aufnehmen (Faustregel: pro 10 Zeilen 1 Bezeichner).
- Suche nach Bezeichnertexten wenn möglich mit Aufwand $O(1)$.

Hinweis: Für die lexikalische Analyse wird in der Regel eine Stringtabelle nicht unbedingt benötigt, sie dient als „Gedächtnis“.

ADT Stringtabelle

Gelieferte Operationen:

- `insert(text, key)`
- `find_or_insert(text) : (key,value)`
- `get_text(key) : text`

Benötigte Operationen:

keine

`insert` dient für Voreinträge (z.B. `while`)

Suchverfahren in Stringtabelle

- Sequentielle Suche (nie sinnvoll)
- Suchbaum (in Fortran 77 nötig)
- Hashen
 - Perfektes Hashen (nur bei vorher bekannter Bezeichnermenge)
 - Verkettetes Hashen (Aufwand?)
 - Hashen mit quadratischem Sondieren o.ä. (Aufwand?)

$$\text{index}_i := (h(x) + i^2 \times g(x)) \bmod |\text{Tabelle}|$$

i : i -te Kollision, h : Hashfunktion, g : optionale Hashfunktion (ggf. $g := 1$)

- Einträge: Verweise auf Symbole, gleichzeitig deren Merkmal

Achtung: Bestimmte Implementierung der Stringtabelle kann von der Quellsprache erzwungen werden, siehe Fortran

Umstiegspunkt für Stringtabelle

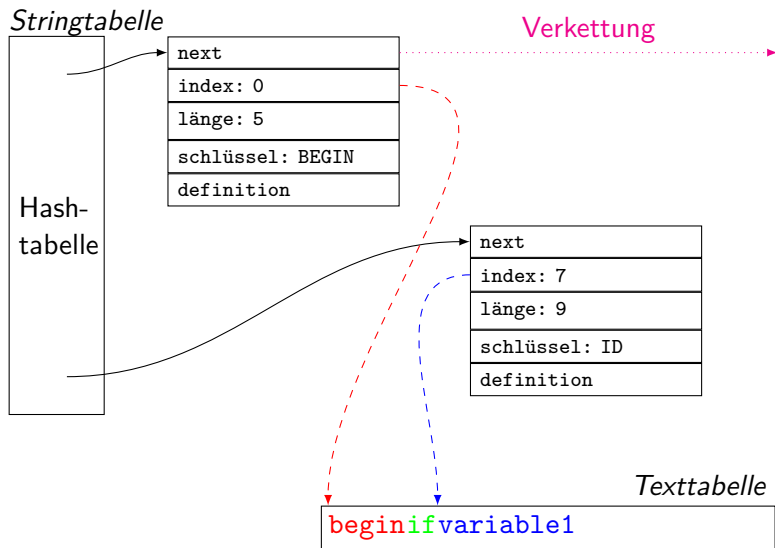
Alternativen:

- Perfektes Hashen: bestes Verfahren wenn anwendbar
- Sondieren (z.B. quadratisch):
Aufwand: Hashfunktion + # Kollisionen
- verkettetes Hashen: Aufwand: Hashfunktion + Kettenlänge
- Suchbaum: Aufwand Pfadlänge im Baum ($O(\log(|\text{Einträge}|))$)
in formatiertem FORTRAN sind Leerzeichen erlaubt,
ständiges Berechnen von Hashschlüsseln und Suchen in
Tabelle ist zu teuer, im Baum ist binäre Suche möglich

Wann welche Technik?

- perfektes Hashen, wenn möglich
- quadratisches Sondieren mit Aufwand $O(1)$, wenn
Hashfunktion gleichverteilt und Hashtabelle nur halbvoll;
Abhilfe Tabellenverdopplung
- verkettetes Hashen: $O(1)$, wenn Hashfunktion ungefähr
gleichverteilt, keine Abhilfe, wenn letzteres nicht erfüllt ist
- Suchbaum: sonst

Organisation der Stringtabelle



ADT Stringtabelleneintrag

```
abstract class StringTableEntry is
    next: StringTableEntry; // Verkettung
    index: Integer;          // Index in Texttabelle
    length: Integer;         // Laenge in Texttabelle
    key: Key;                 // Art des Eintrags
    definition : SymbolTableEntry;
    // Momentan gueltige Definition
end StringTableEntry;
```

Verweise auf Objekte dieses Typs als Merkmal für Bezeichner usw.

- Tabellenlänge: Abstand zu $n \times 256!$
 - Primzahl
 - 2^p (Vermeiden von ganzzahliger Division), p kein Vielfaches von 8!
- Hashverfahren:
 - Hashen mit quadratischem Sondieren, ...
 - verlangt eventuell Tabellenverlängerung
 - verkettetes Hashen
- Faustregel: ca. ein Neueintrag für 10 Zeilen Quelltext
- Hashtabelle nach Ende der lexikalischen Analyse überflüssig, nur Symboleinträge und Texttabelle werden noch benötigt

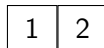
- Hashfunktion muss **schnell** berechnet werden können!
- Anfang/Ende vieler Bezeichner gleich (a1, a2, ...)
- $id = id_1 id_2 \dots id_k$
 - $h(id) = abs(id_1) + abs(id_k) + abs(id_{(k+1)/2})$
 - $h(id) = c_1 \times abs(id_1) + c_2 \times abs(id_k) + c_3 \times abs(id_{(k+1)/2})$
 - $c_i = 1, 4, 8?$ Spreizung über $0 \dots 255$, abhängig vom Zeichensatz
 - $h(id) = \sum_{i=1}^k c_i \times abs(id_i)$
 - ...
 - alle Berechnungen modulo Tabellenlänge
- bei Hashfunktionen kommt es auf Gleichverteilung an, nicht auf die aktuelle Rechenmethode
 - z.B. spart wortweises Addieren Zeit (verletzt aber die Typregeln)

Sei

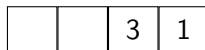
$$n = 2^k$$



Zugriffszeit für n Elemente:



$$n + n/2 + n/4 + n/8 + \dots < 2n \in O(n)$$



amortisiert für ein Element: $O(1)$



Betrachte **Knoten 5**: 4 Schreibzugriffe für Kopieren + eigentlicher Schreibzugriff

Achtung: Der Index kann nicht das Merkmal eines Symbols sein, da er sich offenbar ändern kann.

Kapitel 2: Lexikalische Analyse

- 1 Eingliederung in den Übersetzer / Zielvorgaben
- 2 Theoretische Grundlage: Endliche Automaten
- 3 Implementierung
 - Implementierung endlicher Automaten, Tabellenkompression
 - Der Tokenstrom
 - Implementierung der Stringtabelle
- 4 Einsatz von Generatoren
 - JLex/Flex

Generatoren für die lexikalische Analyse

- Scannergeneratoren werden eingesetzt um die lexikalische Analyse eines Compilers möglichst kompakt und einfach zu spezifizieren (eine Domain Specific Language).
- Eingabe: Ein Zeichenstrom (ASCII, UTF-8, ...)
- Ausgabe lässt sich meist in der Beschreibungssprache spezifizieren. Mehrere Möglichkeiten zur Anbindung eines Parsers:
 - Push-Schnittstelle: Bei erkanntem Token wird eine Methode des Parsers aufgerufen
 - Pull-Schnittstelle: Der Parser ruft eine Methode im Lexer auf um das nächste Token zu bekommen. Meist am einfachsten zu programmieren.
 - Pufferung: Tokenstrom wird vor Syntaxanalyse teilweise oder komplett vorberechnet. Dies kann günstiger für den Cache sein.
- Typischerweise keine automatische Unterstützung für Stringtabellen (da oft sehr sprachspezifisch) und Fehlerbehandlung.

Verbreitete Generatoren

- *Lex*: Werkzeug zur Generierung von Scannern bekannt aus dem Unix-/C-Umfeld. Spezifikation von regulären Ausdrücken erzeugt Tabellengesteuerten Scanner.
- Flex ist eine Open-Source Implementierung von Lex.
- Zahlreiche Portierungen auf andere Programmiersprachen. Beispiele: *JLex* (Java), *CsLex* (C#), *Alex* (Haskell), ...
- Es gibt auch Scanner die nicht mit regulären Ausdrücken/endlischen Automaten arbeiten. Ein bekannter Vertreter ist *antlr* der LL(*k*) Grammatiken benutzt.

Vor- und Nachteile von Generatoren

Vorteile:

- Die Spezifikation von Scannern ist kompakt
→ verständlich und gut wartbarer Code, weniger Fehler
- Automatische Konsistenzprüfung
- Kurze Entwicklungszyklen, schnelles Prototyping möglich
- Scannergeneratoren erzeugen sehr performanten Code.

Nachteile:

- Neue Sprache/Werkzeug muss erlernt werden
- Integration in Buildsystem notwendig
- Fehlersuche oft sehr mühsam da generierter Code für Menschen schlecht lesbar
- Fehlende Flexibilität; In der Praxis gibt es häufig kleinere Ausnahmen die man nur umständlich oder gar nicht mit Generatoren umsetzen kann.

Scanner für XML

XML-Dateien lassen sich in Tokens zerlegen. Beispiel:

```
<?xml version='1.0'?>  
<!-- my personal books -->  
<books>  
    <book name='Goedel, Escher, Bach' />  
</books>
```

- : Strings
- : XML-Namen
- : weitere Tokens

Scanner für XML

XML-Dateien lassen sich in Tokens zerlegen. Beispiel:

```
<?xml version='1.0'?>  
<!-- my personal books -->  
<books>  
  <book name='Goedel, Escher, Bach' />  
</books>
```

- : Strings
- : XML-Namen
- : weitere Tokens

Reguläre Ausdrücke

Spezifikation Regulärer Ausdrücke in Flex (Auszug):

x	Zeichen 'x' erkennen
xy	Zeichenkette xy erkennen (ohne Interpretation)
\x	Zeichen 'x' erkennen (ohne Interpretation)
.	Jedes Zeichen außer Zeilenumbruch erkennen
[xyz]	Zeichen 'x', 'y' oder 'z' erkennen
[abj-oZ]	Zeichen 'a', 'b', 'Z' oder Zeichen von 'j' bis 'o' erkennen
[^A-Z]	Alle Zeichen außer den Zeichen von 'A' bis 'Z' erkennen
x y	x oder y erkennen
(x)	x erkennen (verändert die Bindung)
x*	0, 1 oder mehrere Vorkommen von x erkennen
x+	1 oder mehrere Vorkommen von x erkennen
x?	0 oder 1 Vorkommen von x erkennen
{Name}	Expansion der Definition <i>Name</i>
\t, \n, \r	Tabulator, Zeilenumbruch, Wagenrücklauf erkennen

Spezifikation

NameStartChar	[a-zA-Z]
NameChar	{NameStartChar} [\-\.0-9]
Name	{NameStartChar}({NameChar})*
Comment	"<!--"([^\-] "-"[^\-])*"-->"
String	\' [^\-']* \\' \" [^\-]* \"
Token	"<?\" \"</\" \"<\" \"/>\" \">\" \"?>\" \"=\"

```
%%  
{String}      printf("String: %s\n", yytext);  
{Name}       printf("Name: %s\n", yytext);  
{Comment}    /* skip */  
{Token}      printf("Token: %s\n", yytext);  
.|\n         /* skip */
```

```
%%  
int main(void) {  
    yylex();  
    return 0;  
}
```

Spezifikation

NameStartChar	[a-zA-Z]
NameChar	{NameStartChar} [\-\.0-9]
Name	{NameStartChar}({NameChar})*
Comment	"<!--"([^\-] "-"[^\-])*"-->"
String	\ '[^\']*'\ \"[^\"]*\"
Token	"<?\" \"</\" \"<\" \"/>\" \">\" \"?>\" \"=\"

```
%%  
{String}      printf("String: %s\n", yytext);  
{Name}       printf("Name: %s\n", yytext);  
{Comment}    /* skip */  
{Token}      printf("Token: %s\n", yytext);  
.|\n         /* skip */
```

```
%%  
int main(void) {  
    yylex();  
    return 0;  
}
```

Verwendung von Flex

```
$ flex -o scanner.c scanner.l
$ gcc scanner.c -lfl -o scanner
$ ./scanner < test.xml
Token: <?
Name: xml
Name: version
Token: =
String: "1.0"
Token: ?>
Token: <
Name: books
Token: >
Token: <
Name: book
Name: name
Token: =
String: 'Goedel , Escher , Bach '
Token: />
Token: </
Name: books
Token: >
```

test.xml:

```
<?xml version="1.0"?>
<!-- my personal books -->
<books>
  <book name=
    'Goedel , Escher , Bach' />
</books>
```