Kapitel 2

Lexikalische Analyse

1 Eingliederung in den Übersetzer / Zielvorgaben

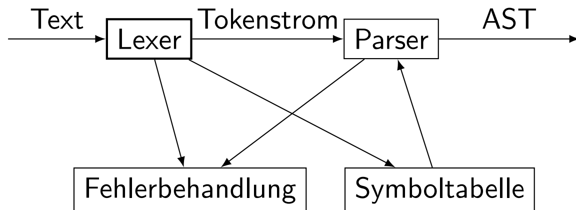
2 Theoretische Grundlage: Endliche Automaten

3 Implementierung
- Implementierung endlicher Automaten, Tabellenkompression
- Der Tokenstrom

lexikalische Analyse: Aufgabe

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

Eingliederung in den Übersetzer



Warum getrennte lexikalische Analyse?

- Durchschnittliche Komplexität einer Anweisung (Knuth und andere[1]):

  var := **var** + **const**;

  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 unrelevant ($<$5% der Compilerlaufzeit).
- Trennung aber Softwaretechnisch sinnvoll: Entkopplung, Modularisierung.

[1] D.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 → 1.E1,
    1.EQ. → 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`

Symboltabelle:

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

Fehlerbehandlung:

- `add_error(nr,text)`

Tokenidentifikation

Token können identifiziert werden durch:

- Endzustand im Automaten (für jedes Token ein eigener Endzustand)
- Symboltabellen (durch Vergleich mit deren Einträgen)
- Hybrider Ansatz
  - Wortsymbole (z.B. „if", „class") und Bezeichner (z.B. Variablen- oder Funktionsnamen) erst in Symboltabelle 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

1 Eingliederung in den Übersetzer / Zielvorgaben

2 Theoretische Grundlage: Endliche Automaten

3 Implementierung
- Implementierung endlicher Automaten, Tabellenkompression
- Der Tokenstrom

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
- $\delta : S \times \Sigma \to 2^S$ Übergangsrelation
- Elemente in δ: $sx \to s'$; $s \in S$, $x \in \Sigma$, $s' \in 2^S$

A akzeptiert die Zeichenketten $L(A) = \{\tau \in \Sigma^* | s_0\tau \to^* s, s \in F\}$.
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 $\epsilon, +, *, (,), [,]$, die nicht in V enthalten sind. Eine Zeichenkette R über V ist ein regulärer Ausdruck über V, wenn:

- R ist ein einiziges 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 Konkatenation.

Satz:

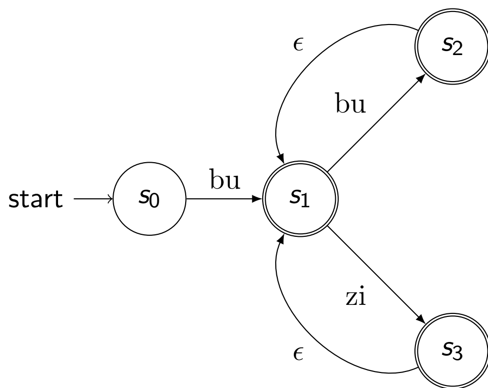
Für jeden regulären Audruck R existiert ein endlicher Automat A, so dass $L(A) = L(R)$.

Beispiel: Konstruktion eines endlichen Automaten

Regulärer Ausdruck:
$(bu((bu + zi))^*)$
Einfügen von Zuständen:
$_0(bu_1((bu_2 + zi_3))^*)$
Übergangsregeln zwischen allen benachbarten Zuständen:
$0bu \rightarrow 1$, $1bu \rightarrow 2$, $1zi \rightarrow 3$, $2\epsilon \rightarrow 1$, $3\epsilon \rightarrow 1$

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

Beispiel: Endlicher Automat



Endzustand Durchgangszustand start → Startzustand

Teilmengenkonstruktion

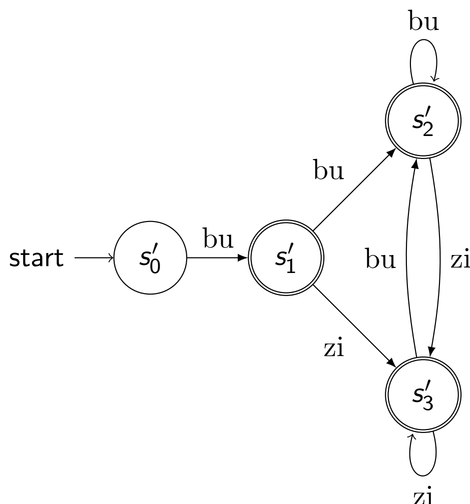
- *Teilmengenkonstruktion macht endl. Automaten deterministisch*
- Zuständer in A' sind Mengen von Zuständen in A
  - Initial: $s'_0 = \{s_0\}$
  - ϵ-Übergänge:
    Wenn $s_i\epsilon \rightarrow_A s_j$ und $s_i \in s'$ dann $s' := s' \cup \{s_j\}$
  - Sonstige Übergänge:
    Wenn $s_i a \rightarrow_A p_j$ dann erweitere $\{s_1, \ldots, s_k\}a \rightarrow_{A'} \{p_1, \ldots, p_l\}$
  - 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:
$0\epsilon \rightarrow 1$, $1bu \rightarrow 2$, $2\epsilon \rightarrow 3$, $3\epsilon \rightarrow 4$, $3\epsilon \rightarrow 6$, $4bu \rightarrow 5$, $6zi \rightarrow 7$, $5\epsilon \rightarrow 8$, $7\epsilon \rightarrow 8$, $8\epsilon \rightarrow 3$, $2\epsilon \rightarrow 9$, $9\epsilon \rightarrow 10$

- 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:
  $0'bu \rightarrow 1'$, $1'bu \rightarrow 2'$, $1'zi \rightarrow 3'$, $2'bu \rightarrow 2'$, $2'zi \rightarrow 3'$, $3'zi \rightarrow 3'$, $3'bu \rightarrow 2'$
- 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:

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

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, $k = 0, 1, 2, \ldots$ 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:

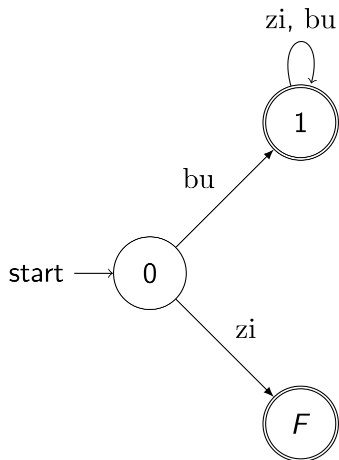
- Klasseneinteilung ändert sich nach m Schritten nicht mehr, $m < \max(l, k)$
- Äquivalenzklasse je ein Zustand, Übergänge und Endzustände wie bei Teilmengenkonstruktion

Beispiel: Äquivalenzklassenbildung

Deterministischer Automat (nicht minimal):
$0bu \rightarrow 1$, $1bu \rightarrow 2$, $1zi \rightarrow 3$, $2bu \rightarrow 2$, $2zi \rightarrow 3$, $3zi \rightarrow 3$, $3bu \rightarrow 2$

- Zustände $\{0\}$ Endzustände $\{1, 2, 3\}$ Fehler $\{F\}$
- Partitionierung?
  $\{0\}bu \rightarrow \{1, 2, 3\}$, $\{0\}zi \rightarrow \{F\}$, $\{1, 2, 3\}bu \rightarrow \{1, 2, 3\}$,
  $\{1, 2, 3\}zi \rightarrow \{1, 2, 3\}$
- 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.

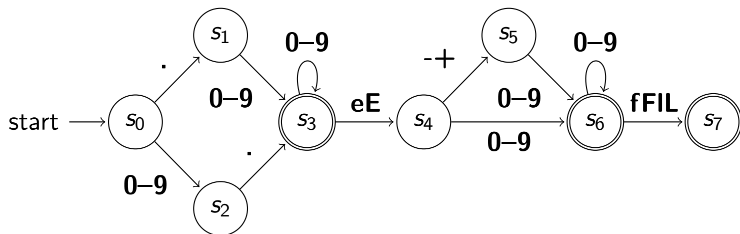
Automat für Float-Konstanten in C[2]

Regulärer Ausdruck

$$((\mathbf{0} + \ldots + \mathbf{9})^{*}\mathbf{.}(\mathbf{0} + \ldots + \mathbf{9})^{+}) + ((\mathbf{0} + \ldots + \mathbf{9})^{+}\mathbf{.})$$

$$(\epsilon + ((\mathbf{e} + \mathbf{E})(\mathbf{+} + \mathbf{-} + \epsilon)(\mathbf{0} + \ldots + \mathbf{9})^{+}))(\epsilon + \mathbf{f} + \mathbf{F} + \mathbf{l} + \mathbf{L})$$

Automat



[2] siehe ISO/IEC 9899:1999 §6.4.4.2

1 Eingliederung in den Übersetzer / Zielvorgaben

2 Theoretische Grundlage: Endliche Automaten

3 Implementierung
- Implementierung endlicher Automaten, Tabellenkompression
- Der Tokenstrom

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	*bu*	*zi*	*Trennzeichen*
0	1	*fehler*	0
1	1	1	*Ende*

```
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 $\mathit{bu}$: state = 1; break;
            case $\mathit{zi}$: return ERROR;
            case $\mathit{Trennzeichen}$: state = 0; break;
        } break;
    case 1:
        switch(cur) {
            case $\mathit{bu}, \mathit{zi}$: state = 1; break;
            case $\mathit{Trennzeichen}$: 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 Prozessorcache

Besonderheiten

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 Symboltabelle
- 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 Ressource-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:

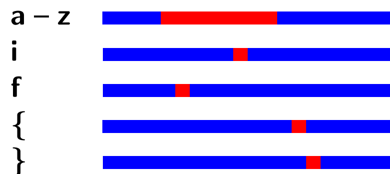
- Partitioniere Σ vorher
- Grundidee:
  reguläre Ausdrücke $(X + Y)$, (XY), $(X)^*$ mit Zeichenmengen $X, Y \subseteq \Sigma$

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 beknnat (mit Abbildung $M \rightarrow J$)

1 Eingliederung in den Übersetzer / Zielvorgaben

2 Theoretische Grundlage: Endliche Automaten

3 Implementierung
- Implementierung endlicher Automaten, Tabellenkompression
- Der Tokenstrom

Tokenstrom

- 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 Symboltabelle, 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