Fragenkatalog: Compilerbau

1. Einführung

Was ist Übersetzerbau?

Ein Übersetzer (Compiler) ist ein Programm, das einen Eingabestrom liest und einen Ausgabestrom erzeugt. Der Eingabestrom besteht aus einem Programm der Quellsprache (source language), der Ausgabestrom aus einem Programm der Zielsprache (target language). Per Definition bleibt die Bedeutung eines Programms auch nach der Übersetzung erhalten.

Wozu braucht man Übersetzer?

Übersetzerbau beschäftigt sich mit der Übersetzung von Programmiersprachen in Maschinencode (klassischer Fall). Daneben gibt es in der Praxis viele grosse und kleine Übersetzungsaufgaben, bei denen Daten in einer Form (am bequemsten in Textform) vorliegen und in einer leicht anderen Form gebraucht werden (praktischer Fall).

Wozu braucht man die Theorie?

Das theoretische Wissen über Übersetzerbau und die dazu entwickelten Werkzeuge kann man in beiden, im klassischen sowie im praktischen Fall, verwenden.

Welche Werkzeuge gibt es?

um nur einige zu nennen:
sed – ein Stream Editor
lex (flex) – für die lexikalische Analyse
yacc (bison) – für die Generierung des Parsers
awk – Patterndirected Texttransformation

Welche Phasen gibt es?

Analysephase (Frontend):

lexikalische Analyse
syntaktische Analyse
semantische Analyse

Synthesephase (Backend):

Zwischencode-Erzeugung
Code-Optimierung
Maschinencode-Erzeugung

Was wird in den einzelnen Phasen gemacht?

lexikalische Analyse:

Der Eingabestrom wird in atomare Einheiten (Tokens) zerlegt.

Aufgaben der lexikalischen Analyse:

- Überprüfung, ob alle Zeichen aus dem Zeichenvorrat der Quellsprache sind
- Erkennen von Grundsymbolen, Token
- Entfernen von irrelevanten Zeichenfolgen (Kommentare, überflüssige Leerzeichen, Zeilenenden, Tabulatoren usw.)

syntaktische Analyse:

Hier wird die Struktur des Tokenstreams analysiert und überprüft ob syntaktische oder strukturelle Fehler vorliegen. Zur Darstellung der Struktur wird ein Baum (Parsetree) aufgebaut der die zur Verarbeitung relevante Struktur des Programms beschreibt. Dies geschieht typischerweise mit Hilfe einer Grammatik. Die Knoten des Baumes repräsentieren Operatoren oder Operanden. Die Blätter bezeichnet man als Terminal-Symbole.

semantische Analyse:

Weitere Eigenschaften der Eingabe werden extrahiert und überprüft. Liegt überhaupt ein übersetzbares Programm vor? Ist die Variable vor der Bedeutung definiert? Stimmt der Typ? Aufbau eines AST, d.h. eine kompakte symbolische Repräsentation der Eingabe.

Zwischencode-Erzeugung:

Erzeugung eines Zwischencodes. Vorteil: grössere Kompatibilität (d.h. leichtere Übertragung auf andere Maschinen).

Code-Optimierung:

Vor der Erzeugung des Codes werden verschiedene Optimierungen am AST vorgenommen um einen kompakten und schnellen Code zu erzeugen. Nach der Zwischencode-Erzeugung wird auch dieser noch weiter optimiert.

Maschinencode-Erzeugung:

Ausgehend vom Zwischencode wird ein Code spezifisch für die gegebene Maschine erzeugt gefolgt von weiteren maschinenspezifischen Optimierungen.

Wozu braucht man eine Symboltabelle?

Neben syntaktischen Eigenschaften, die durch die Zugehörigkeit durch eine Tokenklasse gegeben sind, haben viele Token ausserdem noch weitere semantische Attribute. Zahlen etwa haben einen Wert, Namen haben einen Typ oder sind selbst Typen. Diese Zusatzinformation, die bei der lexikalischen und syntaktischen Analyse anfällt wird für die spätere Verwendung in Symboltabellen festgehalten.

Welche Arten von Fehlern können in den verschiedenen Phasen auftreten?

Fehler treten hauptsächlich in den ersten Phasen: lexikalische, syntaktische und semantische Analyse auf. So ist zum Beispiel „§“ kein gültiges Symbol in einem C Programm (lexikalischer Fehler). Die Kombination „int=double;“ ist syntaktisch nicht möglich

(syntaktischer Fehler) und „const int x; x=0;“ ist semantisch nicht korrekt (semantischer Fehler). Eine gute Fehlerbehandlung ist eine Kunst, da die genaue Ursache von Fehlern oft nicht aus dem Programmtext zu erschliessen ist.

2. Sprachen

Geben Sie Definitionen zum Thema Sprachen.
Die Frage ist, woraus bestehen Sprachen:
aus einem *Alphabet* von Zeichen und *Strings*, also Worten.

Alphabet:
Eingabesymbole wie {0,1} oder {+,-} oder beliebige andere Zeichen, meist ASCII.

String:
Ein String ist eine endliche Folge von Zeichen eines Alphabets. Es ist dann diesem Alphabet zugeordnet. Auch eine leere Folge von Zeichen (ε) ist ein String.

Sprache:
Eine Sprache ist eine Menge von Strings über ein Alphabet, ohne semantischen Gehalt. Es gibt auch die leere Sprache {}, die sich von der Sprache mit dem leere Folgestring {ε} unterscheidet. Im Gegensatz zum endlichen Alphabet und Strings kann eine Sprache auch unendlich sein.

Erläutern Sie den Zusammenhang zwischen String- und Sprachoperationen.
Es gibt zwei Operationen, die auf Strings angewendet werden können:
Aneinanderhängen (concatenation)
Exponentiation (n-maliges Wiederholen eines Eingabezeichens)

Da Sprachen Mengen sind, können hier zum Einen Mengenoperationen angewendet werden:
Vereinigung : $A \cup B$

Aneinanderhängen : A und B werden zu AB

Exponentiation : $A, AA, AAA, ..., A^n \quad n \geq 0$

Wiederholung : A^* bedeutet, dass alle Strings aus A hintereinander hängt

Klammern : Klammern definieren Assoziationen
Erweiterungen entstehen durch die Kombination dieser Basisoperationen.

3. Reguläre Ausdrücke

Welche Sprachen lassen sich mit regulären Ausdrücken definieren?

Eine Sprache, die sich durch einen regulären Audruck definieren lässt, heisst reguläre Sprache.

Bsp.: die Menge der geraden Zahlen als regulärer Ausdruck
gerade → [0|2|4|6|8]
ungerade → [1|3|5|7|9]
gerade Zahlen → (gerade | ungerade)* gerade

Welche Sprachen lassen sich mit regulären Ausdrücken nicht definieren?

Alle nicht regulären Sprachen lassen sich nicht mit regulären Ausdrücken definieren.
Beispiele:
- Primzahlen
- gleich viele schliessende wie öffnende Klammern → (*)*

Was ist ein Transition Diagramm und welcher Zusammenhang besteht zwischen regulären Ausdrücken, Transition Diagramms und endlichen Automaten?

Transition Diagramms besteht aus Knoten und gerichteten Kanten zwischen den Knoten. Es gibt einen Startknoten. Es gibt einen oder mehrere Endknoten. Die Kanten können mit einem Zeichen aus dem Alphabet markiert sein. Jeder Weg von Start- bis Endknoten beschreibt einen String, wobei die Markierungen entlang der Kanten die Zeichen sind. Die Menge aller möglichen, auf diese Weise gebildeten Strings, entspricht der Sprache, die das Diagramm beschreibt.
Jeder reguläre Ausdruck lässt sich in ein solches Transition Diagramm übersetzen. Ausserdem kann man ein Transition Diagramm auch als einen (meist nicht deterministischen) regulären Automaten betrachten. Dabei stellt man sich die Knoten als Zustände, und die Kanten als Zustandsübergänge der Maschine vor. Endliche Automaten werden zur Analyse regulärer Sprachen benutzt.

Was ist ein deterministischer / nicht deterministischer endlicher Automat?

Ein endlicher Automat ist definiert durch ein Fünftupel:
A=(E, Q, Q0, Qf, P)
Dabei sind:
E: das Eingabealphabet des Automaten
Q: die Zustände des Automaten
Q0: ein ausgezeichneter Anfangszustand
Qf: ein Endzustand, der angibt, dass bei Erreichen dieses Zustandes die Eingabe akzeptiert worden ist
P: eine Übergangsfunktion P: Q x E → Q

Endliche Automaten werden mittels Zustandsgraphen dargestellt, wobei jeder Knoten einen Zustand, und die Pfeile zwischen den Knoten die Übergänge in andere Zustände auf Grund eines Eingabezeichens kennzeichnen.
Besitzt mindestens ein Zustand mehrere Zustandsübergänge für ein und dasselbe Zeichen so bezeichnet man den endlichen Automaten als nicht deterministisch. Sonst ist er deterministisch.

Wie definiert ein Automat eine Sprache?

Mit jedem Zustandsübergang ist ein Zeichen des Alphabets assoziert. Jeder mögliche Weg vom Startzustand zum Endzustand entspricht einem String, den die Maschine akzeptiert. Die Menge, der von der Maschine akzeptierten Strings, definiert die Sprache.

Wie werden reguläre Ausdrücke in nicht deterministische endliche Automaten übersetzt?

Jeder reguläre Ausdruck kann in Form eines Transition Diagramms dargestellt werden. Dieses kann wiederum als ein nicht deterministischer endlicher Automat aufgefasst werden, wenn man die Diagrammknoten als Zustände und die Kanten als Zustandsübergänge interpretiert.

Wie übersetzt man nicht deterministische Automaten für reguläre Sprachen in deterministische Automaten? Geht das immer?

Das Überführen eines nicht deterministischen in einen deterministischen Automaten ist immer automatisch möglich. Dazu nummeriert man die Zustände des alten Automaten. Für den ersten Zustand notiert man für jedes ausgehende Zeichen den Zustand oder die Zustandsmenge, in die es übergeht. Dann wiederholt man dieses Vorgehen für jeden Zustand oder Zustandsmenge die man sich auf diese Weise notiert hat. Danach entspricht jeder Zustand und jede Zustandsmenge einem Zustand des neuen deterministischen Automaten.

Fertigkeit im Schreiben von regulären Ausdrücken für einfache Sprachen

Literale

Besteht ein regulärer Ausdruck aus einer Zeichenfolge, die für sich selbst steht, so handelt es sich um ein Literal.
Im einfachsten Fall besteht die Zeichenfolge aus genau einem Zeichen. Der reguläre Ausdruck

a

erkennt folglich ein "a" im Eingabetext.
Ein Literal kann auch aus mehreren Zeichen bestehen:

abcd

paßt auf alle Zeichenketten der Eingabe, die "abcd" enthalten.

Zeichenklassen

Verschiedene Operatoren ermöglichen das Erkennen einer ganzen Klasse von Zeichen.
So ist der Punkt

.

ein Platzhalter für ein beliebiges Zeichen außer dem Zeilenendezeichen.

a.b

paßt damit auf alle Zeichenfolgen, die mit a anfangen, mit b aufhören und dazwischen ein beliebiges Zeichen aufweisen.
Die eckigen Klammern

[m]

sind ein Platzhalter für genau ein Zeichen aus der Menge m.
In den Klammern stehen Zeichenfolgen, als Beispiel paßt der Ausdruck

[abc]

auf "a" oder "b" oder "c".
Auch ganze Bereiche von Zeichen können angegeben werden, beispielsweise

[a-z]

was auf alle Zeichen im Bereich von "a" bis "z" paßt.
Die oben dargestellten Aufzählungen und Bereichsangaben können auch miteinander kombiniert werden:

[a-zAEIOU]

erkennt somit alle Buchstaben im Bereich von "a" bis "z" und einen der Buchstaben "A", "E", "I", "O" oder "U".
Soll das Zeichen "-" selbst in der gesuchten Zeichenklasse sein, so muß es am Anfang oder Ende der Klammer stehen. Alle anderen Spezialzeichen können innerhalb der eckigen Klammern für sich selbst stehen. Sollen eckige Klammern selbst innerhalb eines Zeichenklassenausdrucks stehen, so müssen sie gleich am Anfang des Ausdrucks stehen.
Aus der komplementären Zeichenfolge kann ausgewählt, indem "^" am Klammeranfang angegeben wird.

[^0-9]

erkennt also jedes nichtnumerische Zeichen.
Soll das Zeichen "^" selbst gesucht werden, so darf es nicht am Anfang der eckigen Klammer stehen.

Wiederholungsoperatoren

Wenn a ein regulärer Ausdruck ist, dann machen die folgenden Operatoren Angaben über die Häufigkeit des vorangehenden Ausdrucks a und bilden damit selbst reguläre Ausdrücke.

a*

bedeutet, daß a Null-mal oder mehrmals vorkommt.

a*bc

erkennt also "bc", "abc", "aabc", usw..

a?

bedeutet, daß a Null-mal oder einmal vorkommt.
Der Ausdruck

a?bcd

erkennt die Zeichenketten "bcd" oder "abcd".

a+

bedeutet, daß a einmal oder mehrmals vorkommt.
Der Ausdruck

a+bcd

erkennt "abcd", "aabcd", usw..
Wiederholungsoperatoren können sich auch auf Zeichenklassenausdrücke beziehen.

Kontextoperatoren

Reguläre Ausdrücke erlauben - in beschränktem Umfang - die Angabe, in welchem Kontext der gesuchte Ausdruck vorkommen soll. So kann angegeben werden, ob der gesuchte Ausdruck am Anfang oder Ende einer Zeile vorkommt. Der so entstehende Ausdruck ist wieder regulär.

^a

bedeutet, daß der folgende Ausdruck nur am Zeilenanfang vorkommt.

^abcd

findet alle Zeilen mit "abcd" am Zeilenanfang.
Ein Konflikt mit dem Komplementzeichen ^ ist nicht möglich, da dieses nur innerhalb eines []-Klammerpaares auftreten kann.

a$

bedeutet, daß der vorhergehende Ausdruck nur am Zeilenende vorkommt.

abcd$

findet alle Zeilen mit "abcd" am Zeilenende.

Komplexe Ausdrücke

Wenn a und b reguläre Ausdrücke sind, dann ist ab ebenfalls ein regulärer Ausdruck. Die hiermit ausgedrückte Verkettungsoperation hat keinen eigenen Operator; sie wird nur durch die Aneinanderreihung von regulären Ausdrücken realisiert.
Alternativen von regulären Ausdrücken können gebildet werden, wenn nach mehreren Mustern in der Eingabe gesucht wird:

a|b

ist eine Verknüpfung von alternativen Ausdrücken, also ein "oder".
Die Abarbeitung von regulären Ausdrücken erfolgt grundsätzlich von links nach rechts. Die Reichweite der Operatoren ist auf die unmittelbare linke oder rechte Umgebung beschränkt. Die Operatoren werden in der Reihenfolge
[]
?
+
*
Verkettung
|
abgearbeitet. Abweichungen von der Verarbeitungsreihenfolge können nur durch Klammerung erreicht werden:
Maskierung von Spezialzeichen
Die Spezialbedeutung der Operatoren kann durch das Zeichen \ außer Kraft gesetzt werden. Soll der Backslash selbst gesucht werden, dann muß er verdoppelt werden, auß in eckigen Klammern, in denen er für sich selbst steht.

Regelsysteme

letter → [a|b|...|z]

digit → [0|1|...|9]

id → letter (letter | digit)*

number → digit digit*

exp → id | number

4. Kontextfreie Sprachen

Wie ist durch eine Grammatik eine Sprache definiert ?
Eine Grammatik definiert ein Set von Regeln (Produktionen) über eine Sprache.
Problem der Semantik bzw. Eindeutigkeit.

Was versteht man unter einer Herleitung eines Strings ?
Es geht darum, Worte bzw. Strings in einer kontextfreien Sprache zu erkennen.
* Für einen String kann es unterschiedliche Herleitungen geben
* Oft unterscheiden sich die Herleitungen nicht durch die Struktur sondern durch die Reihenfolge der Regelanwendung
* Unterschiede in der Struktur sind interessant, Unterschiede in der Reihenfolge nicht (hier geht´ s um Assotiation, also, welches Zeichen z.B. mehr bindet)

Was ist eine leftmost/rightmost Derivation ?
Das sind Methoden der Herleitung. Standard ist die leftmost Derivation.

leftmost Derivation LL(k)-Parser:
das Non Terminal, das am weitesten links steht wird expandiert.

rightmost Derivation LR(k).Parser:
das Non Terminal, das am weitesten rechts steht wird expandiert.
-> entspricht parsen von links nach rechts.

Was ist ein Parsetree ?
Dient zur Darstellung von Herleitungen.

Welcher Zusammenhang besteht zwischen Parsetree und Herleitung ?
Startsymbol ist die Wurzel
Hergeleitete Strings befinden sich an den Blättern
Kanten entsprechen Vorgängerbeziehungen und somit der Regel

Was ist eine kontextfreie Sprache ?
Eine kontextfreie Sprache ist eine Sprache, die durch eine kontextfreie Grammatik definiert wird, z.B. durch die Bachus-Naur-Form (BNF).

Da auf der linken Seite einer Produktion nur eine einzelne Variable stehen darf, kann diese Variable unabhängig vom Kontext ersetzt werden - daher die Bezeichnung kontextfreie Grammatik.

Die Verwendung einer Regel in einer Herleitung hängt also nur davon ab, ob das entsprechende Non Terminal vorkommt und nicht vom Kontext, in dem das Non Terminal vorkommt.

Was ist BNF, welche Erweiterungen bietet EBNF gegenüber BNF und wie kann man solche Erweiterungen auf einfache BNF zurückführen ?
In der BNF werden über die Eingabezeichen folgende Mengen gebildet:
* Menge von Terminal Symbolen
* Menge von Nicht Terminal Symbolen
* Menge von Produktionsregeln

* Ein Startsymbol

Dabei müssen die Mengen endlich sein und die Mengen der Terminal und Nicht Terminal Symbole vereinigt die gesamte Menge aller Symbole bilden.
In den Produktionsregeln muss:
* die Menge der Nicht Terminal Symbolen eindeutig einer Liste von Symbolen zuordenbar sein.
* Rekursion von Produktionsregeln möglich sein.
* die Definition mehrerer Regeln zu gleichen Terminal Symbolen möglich sein.
* Terminalsymbole kommen aus dem Alphabet oder Tokens (?)

EBNF erweitert die Symbolik der BNF Regeln, nicht deren prinzipiellen Möglichkeiten:
"*" bedeutet 0 oder mehrere Wiederholungen
"+" 1 oder mehrere Wiederholungen
"|" oder
"[...]" optional
"(...)" Gruppieren

Zurückführen:
S: AB*C

ohne *:
S:AC
S:AB+C

ohne +:
S:ARC
R:BR |;

ohne |:
S:ARC
R:BR
R:

Welche Sprachen lassen sich mit kontextfreien Grammatiken definieren, welche nicht ?
HÄÄÄ ??

Schreiben Sie eine Grammatik für Listen von Zahlen, die durch Komma getrennt sind. Zum Beispiel "1,2,3" oder "1" oder ε gültige Listen von Zahlen, aber "1,,2" oder "1,2 3" oder "1,2,3," nicht. Gehen Sie von den terminalen Symbolen Zahl und Komma aus.

liste: | nummernliste
nummerliste: *Zahl* | *Zahl Komma* nummerliste

5. Parsing

Wie funktioniert Shift-Reduce Parsing?

Die Eingabezeichenkette wird von links nach rechts, Zeichen für Zeichen, gelesen. Bevor der Parser ein Zeichen weitershifted, überprüft er ob sich innerhalb der bisher eingelesenen Zeichen eine Regel erkennen lässt, wie sie auf der rechten Seite der Grammatik vorkommt,

auf die der Parser basiert. Findet er eine solche Regel (Handle) dann werden die entsprechenden Zeichen durch die linke Seite dieser Regel ersetzt (reduziert). Dies wird solange durchgeführt, bis nur noch das Startsymbol übrig ist.

Was ist der Unterschied zwischen SLR und LR(k)?

LR(k) hat im Unterschied zu SLR die Möglichkeit k Zeichen in die Zukunft zu schauen um einwandfrei festzulegen, welche Regel als nächstes erfolgreich angewandt werden kann.

Gibt es Grammatiken, die man Top-Down aber nicht Bottom-Up parsen kann?

Die Bottom-Up Methode hat folgende Einschränkungen bezüglich der Grammatik:
- es gibt keine zwei gleichen rechten Seiten
- es gibt keine leeren rechten Seiten

Was ist ein Shift-Reduce Konflikt?

Als Beispiel sei folgende Grammatik gegeben:
R → bcc
R → b
Wird das Wort bcc eingelesen hat der Parser beim ersten Zeichen b die Möglichkeit sofort nach R zu reduzieren, oder weiterzushiften und am Ende des Wortes die erste Regel anzuwenden.

Was ist ein Reduce-Reduce Konflikt?

Als Beispiel sei folgende Grammatik gegeben:

start → a | b
a → A
b → A

Hier hat der Parser zwei Möglichkeiten zu reduzieren zwischen denen er nicht entscheiden kann.

Wie kann man Konflikte beheben?

Was ist der Zusammenhang zwischen regulären Sprachen und kontextfreien Sprachen?

Gibt es für jede kontext freie Sprache eine LR(k) Grammatik?

Gibt es zu einer Sprache eine, höchstens eine, oder mehrere Grammatiken?

6. Code Erzeugung

Was ist ein AST ?
AST bedeutet: Abstract Syntax Tree
Der Baum dient der Optimierung des Codes.
Er vereinfacht den Parse Tree, in dem er syntaktische Informationen zusammenfasst

Wozu braucht man intermediate Code ?
Intermediate Code ist ein abstrakter, symbolischer Code ohne Registerzuweisungen. Somit ist er maschinenunabhängig. Es können hier Optimierungen vorgenommen werden.

Was ist Alias Analysis ?
Es wird überprüft, ob zwei Ausdrücke dieselbe Speicherzelle betreffen:
also z.B.:
int i = 2;
int j = i;
int h = i*2
int g = j*4 <- j kann eigentlich durch i ersetzt werden ...

möglicherweise? -> keine Optimierung
sicher ! -> Optimierung
sicher nicht ! -> Common Subexpression

Was ist ein Basic Bock ?
Ein Basic Block ist eine Codesequenz mit einem Eingang am Anfang und ein oder mehrere Ausgänge am Ende (z.B. bedingter Sprung)

Welche Optimierungen gibt es? Erläutern Sie diese jeweils durch ein Beispiel.

Lokale Optimierungen (in einem Basic Block):

* Common Subexpression Elimination
Gleiche Anweisungen werden zusammengefasst
z.B.:
t6 = i*4;
t7 = i*4;
wird zu:
t6 = i*4
t7 = t6

* Copy Propagation
Variablen, die gleiche Werte haben, werden ersetzt
das oben genannte t7 wird in Anweisungen durch t6 ersetzt, wenn sich beide nicht geändert haben.

* Dead Variable Elimination
Variablen, die deklariert, aber nicht benutzt werden, werden eliminiert (also z.B. das t7)

* Strength Reduction
Es werden Operatoren optimiert.
z.B. wird t6 = i*4; bitgeshiftet, was sehr viel performanter ist: t6 = i<<2;

* Alias Analysis
siehe Frage oben

Globale Optimierungen (zwischen Basic Blocks):

* Constant Code Movement

Konstanten werden außerhalb von Schleifen definiert , wenn sie sich in den Schleifen nicht ändern (sie werden dadurch nur einmal definiert, und nicht bei jedem Schleifendurchlauf)

* Induction Variables Elimination
Das sind Laufvariablen, also Variablen, die sich immer um einen konstanten Wert ändern.
z.B. i++; in einer Schleife.
oder:

i++;
t2 = i*4
wird zu:
t2 = t2 + 4 (i wurde eliminiert inkl. Strength Reduction)
anschließend meist Dead Variable Elimination