GEORG-AUGUST-UNIVERSITÄT
GÖTTINGEN

Informatik II

SS 2005

Kapitel 7: Compilerbau

Teil 1: Lexer und Parser

Dr. Michael Ebner
Dipl.-Inf. René Soltwisch

Lehrstuhl für Telematik
Institut für Informatik

Inhalte

- Organisation von Compilern für moderne Programmiersprachen (Teil 7.1)
  - Einführung
  - Lexer
  - Parser
  - Zusammenführung (Bau eines ausführbaren Programms)
- Grundlegende Konzepte von Programmiersprachen (Teil 7.2)
  - Einführung Programmiersprachen
  - Namen, Bindungen und Gültigkeitsbereiche
  - Speichermanagement und Implementierung
  - Kontrollfluss

Hinweise

- Als Grundlage
  - dient das Buch "*Programming Language Pragmatics"* von Michael L. Smith,
  - sowie das vorherige Kapitel „Automaten und Sprachen".
- Vertiefung in weiterführender Vorlesung „Konzepte der Programmiersprachen und des Compilerbaus" von Prof. Dr. Tiziana Margaria-Steffen

Literatur (1/2)

- **Bücher**
  - Michael L. Scott : „Programming Language Pragmatics", MKP 2000, ISBN 1-55860-578-9
    http://www.cs.rochester.edu/u/scott/pragmatics/
  - Klassiker der Automatentheorie von Hopcroft/Ullman/Motwani
    - Hopcroft, Motwani, Ullman : „Introduction to Automata Theory, Languages, and Computation", http://www-db.stanford.edu/~ullman/ialc.html
    - Hopcroft, Motwani, Ullman: „Einführung in die Automatentheorie, Formale Sprachen und Komplexitätstheorie", Pearson Studium, ISBN 3827370205
  - Drachenbuch
    - Alfred V. Aho, Ravi Sethi, Jeffrey D. Ullman: „Compilers - Principles, Techniques and Tools". Addison-Wesley 1988, ISBN 0-201-10088-6
    - Alfred V. Aho, Ravi Sethi, Jeffrey D. Ullman: „Compilerbau." Oldenbourg Verlag 1999, Teil 1: ISBN 3-486-25294-1, Teil 2: ISBN 3-486-25266-6

Literatur (2/2)

- Skripte
  - Compilerbau-Skript von Prof. Dr. Goltz, Universität Braunschweig
    http://www.cs.tu-bs.de/ips/ss04/cb/skript_cp.ps.gz
  - Informatik-Skripte von Prof. Dr. Waack, Universität Göttingen
    http://www.num.math.uni-goettingen.de/waack/lehrmaterial/
- Folien
  - Informatik II - SS2003 Folien dienen als Grundlage und wurden übersetzt und ev. teilweise ergänzt. Es wird aber auch komplett neue Teile geben!!!
    http://user.informatik.uni-goettingen.de/~info2/SS2003/
  - Übersetzerbau I – Prof. Dr. Goos, Universität Karlsruhe
    http://www.info.uni-karlsruhe.de/lehre/2003WS/uebau1/
- WWW: Wikipedia
  - http://en.wikipedia.org/wiki/Compilers:_Principles,_Techniques_and_Tools
  - http://de.wikipedia.org/wiki/Compiler
- Erfahrungsberichte von Studentenseite sind erwünscht

Weitere Quellen

- Katalog von Konstruktionswerkzeugen für Compiler
  http://wwwold.first.gmd.de/cogent/catalog/
  - ANTLR, ANother Tool for Language Recognition: http:/www.antlr.org
  - Populärste Compilergeneratoren: JavaCC, ANTLR, Eli, Byacc und Coco/R
- Konferenzen und Journale
  - ACM Transactions on Programming Languages and Systems
  - ACM SIGPLAN Conference on Programming Language Design and Implementation
  - ACM SIGPLAN Conference on Programming Language Principles

Überblick

- **Einführung**
- Lexer
- Parser
- Zusammenführung (Bau eines ausführbaren Programms)

Warum Programmiersprachen und Compiler studieren?

- Nach Aussage von Michael Scott (siehe Literaturangabe)
  - Verstehe schwer verständliche Spracheigenschaften
  - Wähle zwischen alternativen Wegen um etwas auszudrücken
  - Mache guten Gebrauch von Debuggern, Assemblern, Linkern und andere verwandte Werkzeuge
  - Simuliere nützliche Eigenschaften (features) welche in einer Sprache fehlen
- Nach Aussage von Kevin Scott (vorheriger Dozent)
  - Compiler sind große und komplexe Programme: studieren dieser Programme hilft dir „große Software“ besser zu verstehen
  - Viele Programme enthalten „kleine Programmiersprachen“
    - Unix shells, Microsoft Office Anwendungen, etc.
    - Es ist nützlich etwas über Sprachdesign und –implementierung zu wissen, so dass Sie kleine Sprachen in die eigene Software einbauen können

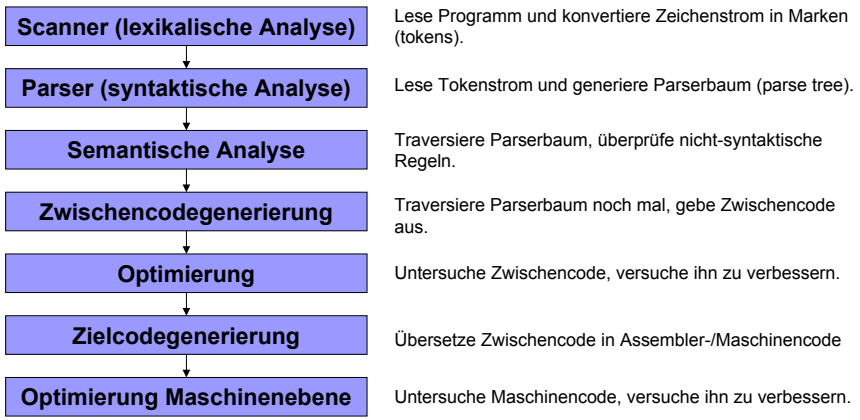
Weitere Fragen zum Nachdenken

- Was macht eine Programmiersprache erfolgreicher als andere?
- Werden Programmiersprachen mit der Zeit besser?
- An welchen Eigenschaften (features) mangelt es deiner bevorzugten Sprache um Sie
  - mächtiger,
  - zuverlässiger,
  - einfacher in der Verwendung zu machen?

Entwurf eines Compilers

- Compiler sind gut untersuchte, aber auch sehr komplexe Programme
  - Daher sollte man nicht davon ausgehen, dass Compiler immer fehlerfrei arbeiten!!!
- Die Komplexität wird durch die Aufteilung der Compilerarbeiten in unabhängige Abschnitte oder Phasen bewältigt
- Typischerweise **analysiert** eine Phase eine Repräsentation von einem Programm und **übersetzt** diese Repräsentation in eine andere, welche für die nächste Phase besser geeignet ist
- Das Design dieser **Zwischen**repräsentationen eines Programms sind kritisch für die erfolgreiche Implementierung eines Compilers

Der Kompilationsprozess (-phasen)



Lexikalische Analyse

- Eine Programmdatei ist nur eine Sequenz von Zeichen
- Falsche Detailebene für eine Syntaxanalyse
- Die lexikalische Analyse gruppiert Zeichensequenzen in **Tokens**
- Tokens sind die kleinste „Bedeutungseinheit“ (units of meaning) im Kompilationsprozess und sind die Grundlage (foundation) fürs Parsen (Syntaxanalyse)
- Die Compilerkomponente zum Ausführen der lexikalischen Analyse ist der **Scanner**, welcher oftmals ausgehend von höheren Spezifikationen automatisch generiert wird
  - Mehr über Scanner in der nächsten Vorlesung

Beispiel für lexikalische Analyse

Ein GGT Programm in C

```
int gcd (int i, int j) {
  while (i != j) {
    if (i > j)
      i = i - j;
    else
      j = j - i;
  }
  printf("%d\n",i);
}
```

Token

int	gcd	(
int	i	,
int	j	)
{	while	(
i	!=	j
)	{	if
(	i	>
j	)	i
=	i	-
j	;	else
j	=	j
-	i	;
}	printf	(
"%d\n"	,	I
)	;	}

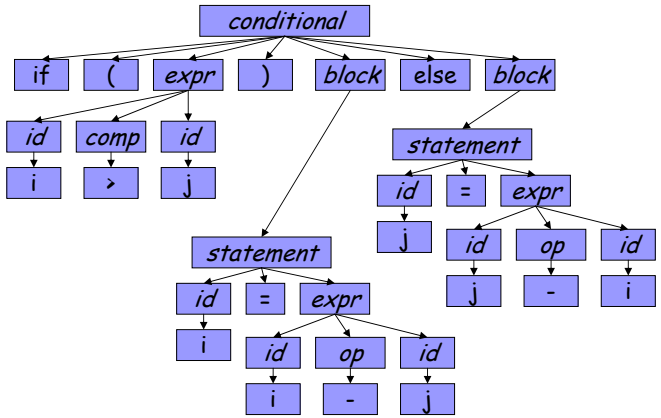
Syntaktische Analyse

- Die lexikalische Analyse erzeugt einen Strom von Tokens
- Falsche Detailebene für die semantische Analyse und Codegenerierung
- Die Syntaxanalyse gruppiert eine Zeichenfolge von Tokens in **Parserbäume**, was durch die **kontextfreie** Grammatik gelenkt wird, die die **Syntax** der zu kompilierenden Sprache spezifiziert
  - `conditional -> if ( expr ) block else block`
- Parserbäume repräsentieren die **Phrasenstruktur** eines Programmes und sind die Grundlage für die semantische Analyse und Codegenerierung
- Die Compilerkomponente zum Ausführen der syntaktischen Analyse ist der **Parser**, welcher oftmals ausgehend von höheren Spezifikationen automatisch generiert wird
  - Mehr über kontextfreie Grammatiken und Parser später

Beispiel Syntaxanalyse

Token

if	(	i
>	j	)
i	=	i
-	j	;
else	j	=
j	-	i
;	}	



Semantische Analyse

- Bestimmt die Bedeutung eines Programms basierend auf der Repräsentation des Parserbaumes
- Setzt Regeln durch, welche nicht durch die Syntax der Programmiersprache verwaltet werden
  - Konsistente Verwendung von Typen, z.B.
    - int a; char s[10]; s = s + a; **illegal!**
  - Jeder Bezeichner (identifier) muss vor der ersten Verwendung deklariert sein
  - Unterprogrammaufrufe müssen die richtige Argumentanzahl und Argumenttyp haben
  - etc.
- Bringt die Symboltabelle auf den aktuellen Stand, welche neben anderen Dingen den Typ von Variablen, deren Größe und den Gültigkeitsbereich in welchen die Variablen erklärt wurden notiert

Zwischencodegenerierung

- Parserbäume sind die falsche Detailebene für die Optimierung und Zwischencodegenerierung
- Zwischencodegenerierung verwandelt den Parsebaum in eine Sequenz von Anweisungen (statements) der **Zwischensprache** welche die Semantik des Quellprogramms verkörpert
- Die Zwischensprache ist genauso Mächtig, aber einfacher, wie die höhere Sprache
  - z.B. die Zwischensprache könnte nur einen Schleifentyp (goto) haben, wogegen die Quellsprache mehrere haben könnte (for, while, do, etc.)
- Eine einfache Zwischensprache macht es einfacher nachfolgende Compilerphasen zu implementieren

Zielcodegenerierung

- Das Endziel eines Compilerprozesses ist die Generierung eines Programms welches der Computer ausführen kann
  - Dies ist die Aufgabe der Zielcodegenerierung
- Schritt 1: durchlaufe (traverse) die Symboltabelle, weise Variablen einen Platz im Speicher zu
- Schritt 2: durchlaufe (traverse) den Parsebaum oder Programm in der Zwischensprache um arithmetische Operationen, Vergleiche, Sprünge und Unterprogrammaufrufe auszugeben, sowie Lasten und Vorräte von Variablenreferenzen

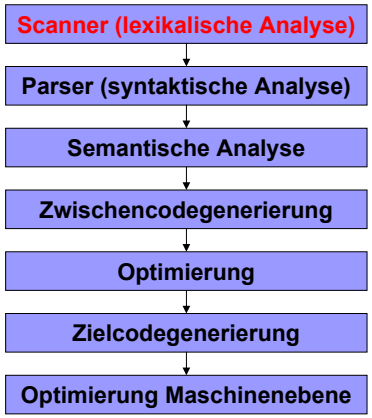
Optimierung

- Zwischencode und Zielcode ist typischerweise nicht so effizient wie er sein könnte
  - Einschränkungen erlauben es dem Codegenerator sich auf die Codeerzeugung zu konzentrieren und nicht auf die Optimierung
- Ein Optimierer kann aufgerufen werden um die Qualität des Zwischencodes und/oder Zielcodes nach jeder dieser Phasen zu verbessern
- Die Compilerkomponente zur Verbesserung der Qualität des generierten Codes wird **Optimierer** (optimizer) genannt.
- Optimierer sind die kompliziertesten Teile eines Compilers
- Optimierungsalgorithmen sind oftmals sehr ausgefeilt, benötigen erheblich viel Speicher und Zeit für die Ausführung und erzeugen nur kleine Verbesserungen der Programmgröße und/oder Leistung der Laufzeit
- Zwei wichtige Optimierungen
  - Registerzuteilung – entscheide welche Programmvariablen zu einem bestimmten Zeitpunkt der Programmausführung in Registern gehalten werden können
  - Unbenutzten Code eliminieren – entferne Funktionen, Blöcke, etc., welche niemals vom Programm ausgeführt würden

Überblick

- Einführung
- **Lexer**
- Parser
- Zusammenführung (Bau eines ausführbaren Programms)

Der Kompilationsprozess (-phasen)



Lese Programm und konvertiere Zeichenstrom in Marken (tokens).

Lese Tokenstrom und generiere Parserbaum (parse tree).

Traversiere Parserbaum, überprüfe nicht-syntaktische Regeln.

Traversiere Parserbaum noch mal, gebe Zwischencode aus.

Untersuche Zwischencode, versuche ihn zu verbessern.

Übersetze Zwischencode in Assembler-/Maschinencode

Untersuche Maschinencode, versuche ihn zu verbessern.

Lexikalische Analyse

- Die lexikalische Analyse gruppiert Zeichensequenzen in **Tokens** (Marken) bzw. Symbole
- Tokens sind die kleinste „Bedeutungseinheit“ (units of meaning) im Kompilationsprozess und sind die Grundlage (foundation) fürs Parsen (Syntaxanalyse)
- Die Compilerkomponente zum Ausführen der lexikalischen Analyse ist der **Scanner**, welcher oftmals ausgehend von höheren Spezifikationen automatisch generiert wird

Beispiel für lexikalische Analyse

Ein *GGT Programm in C*

```
int gcd (int i, int j) {
  while (i != j) {
    if (i > j)
      i = i - j;
    else
      j = j - i;
  }
  printf("%d\n",i);
}
```

Token

```
int     gcd     (
int     i       ,
int     j       )
{       while   (
i       !=      j
)       {       if
(       i       >
j       )       i
=       i       -
j       ;       else
j       =       j
-       i       ;
}       printf  (
"%d\n"  ,       I
)       ;       }
```


2 Fragen

- Wie beschreiben wir die lexikalische Struktur einer Programmiersprache?
  - Mit anderen Worten, was sind die Tokens (Symbole)
- Wie implementieren wir den Scanner nachdem wir wissen was die Tokens sind?

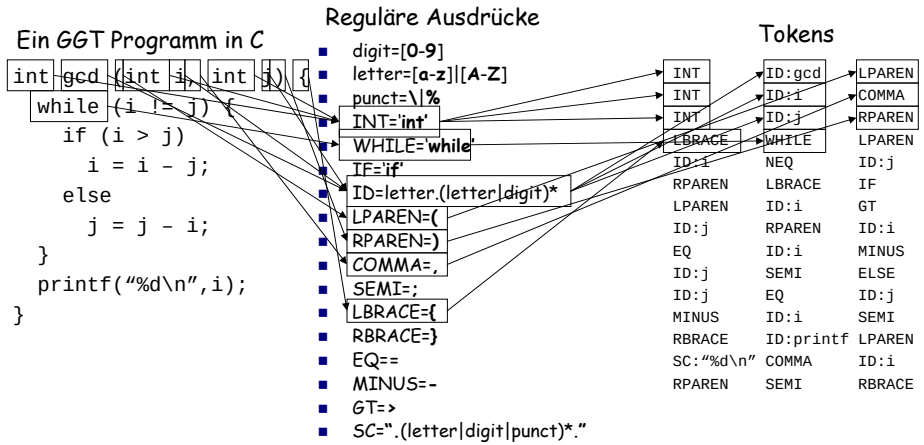
Wie beschreiben wir die lexikalische Struktur? (1/2)

- 1. Versuch: Liste aller Tokens
  - `if else long int short char ; , : ( ) { }` …
  - Aber was ist mit den Konstanten (Ganzzahlen, Fliesskommazahlen, Zeichenketten)?
    - Es können nicht alle aufgelistet werden, es gibt ~8 Milliarden 32-bit integer und floating-point Konstanten und eine **unendliche** Anzahl von Zeichenfolgenkonstanten
    - Das gleiche Problem gilt für Bezeichner (Variablen, Funktionen und benutzerdefinierte Typnamen)
    - Lösung: Wir brauchen einen Weg um kurz und prägnant Klassen von Tokens zu beschreiben, welche eine große Anzahl von verschiedenen Werten abdecken können

Wie beschreiben wir die lexikalische Struktur? (2/2)

- 2. Versuch: Reguläre Ausdrücke
  - Muster (patterns) welche zum Auffinden von passendem Text verwendet werden können
  - Werden mit folgenden Ausdrücken rekursiv ausgedrückt
    - Ein Zeichen
    - Der leeren Zeichenfolge ε
    - Der Verkettung zweier regulärer Ausdrücke
      - `r1.r2` ist der Wert von r1 **gefolgt** vom Wert von r2
    - Der Alternative zweier regulärer Ausdrücke
      - `r1|r2` ist der Wert von r1 **oder** der Wert von r2
    - Der Kleenesche Hülle * (ode einfach Hülle oder Stern)
      - `r*` ist **kein oder mehrere** Vorkommen des Wertes von r1
  - Runde Klammern können zum Gruppieren von regulären Ausdrücken verwendet werden, um Zweideutigkeiten bei Kombinationen auszuschließen
    - z.B. bedeutet r1.r2|r3 nun (r1.r2)|r3 oder r1.(r2|r3)???

Lexikalische Analyse: Reguläre Ausdrücke bei der Arbeit



Ein genauerer Blick auf die lexikalische Analyse

- Wie behandelt der lexikalische Analysator Leerzeichen, Kommentare und Konflikte zwischen regulären Ausdrücken?
  - Leerzeichen
    - `int gcd (int i, int j) {`
  - Kommentare einer Programmiersprache
    - `/* gcd */ int gcd (int i, int j) {`
  - Konflikte zwischen regulären Ausdrücken
  - Gegeben:
    - WHILE='**while**'
    - ID=letter.(letter|digit)*
  - Beide reguläre Ausdrücke decken die Zeichenfolge „while“ ab. Welcher Ausdruck soll aber nun gewählt werden?

Handhabung von Leerzeichen

- Leerzeichen können als Token durch folgende Regel erkannt werden
  - `WS=(\n|\r|\t|\s)*`
    - \n ist ein „escape“ Zeichen für „newline“ (neue Zeile)
    - \r ist ein „escape“ Zeichen für „carriage return“ (Wagenrücklauf)
    - \t ist ein „escape“ Zeichen für „tab“ (Tabulator)
    - \s ist ein „escape“ Zeichen für „space“ (Leerzeichen)
- Das Leerzeichentoken WS ist normalerweise unwichtig für die Syntax einer Programmiersprache, weshalb es einfach vom Tokenstrom gelöscht werden kann

Handhabung von Kommentaren (1/2)

- Alternative 1: Präprozessoren
  - Spezielles Programm welches eine Datei einliest, Kommentare entfernt, andere Operationen wie Makro-Expansion ausführt und eine Ausgabedatei schreibt, welche vom lexikalischen Analysator gelesen wird.
  - Quellprogramme können auch Steueranweisungen enthalten, die nicht zur Sprache gehören, z.B. Makro-Anweisungen. Der lexikalische Analysator behandelt die Steueranweisungen und entfernt sie aus dem Tokenstrom.
  - Präprozessor-Anweisungen in C und C++
    - z.B. `#include` und `#define`

Handhabung von Kommentaren (2/2)

- Alternative 2: Kommentartoken
  - In Abhängigkeit von der Komplexität von Kommentaren ist eine Beschreibung via regulärer Ausdrücke vielleicht möglich
    - Zeilenkommentare (single line comments) können mit regulären Ausdrücken gefunden werden
      - `SLC='//'.*$`
        - $ ist ein spezielles Symbol, welches das Ende einer Zeile bedeutet
      - Findet Texte wie
        - `// Dies ist ein Kommentar`
    - Einige Kommentare sind zu kompliziert um durch reguläre Ausdrücke gefunden zu werden
      - Willkürlich verschachtelte Kommentare
        - `/* level 1 /* level 2 */ back to level 1 */`
      - Wird Normalerweise vom Präprozessor behandelt

Handhabung von Konflikten

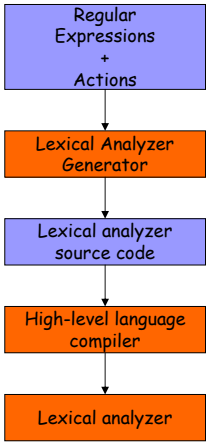
- Gegeben sind zwei reguläre Ausdrücke r1 und r2, welche eine Teileingabe p='c1..ck' finden. Welche soll nun ausgewählt werden?
  - Alternative 1: Längster Fund
    - Nehme solange Eingabezeichen hinzu bis weder r1 noch r2 passen. Entferne ein Zeichen und entweder r1 oder r2 muss passen. Die Teileingabe p ist der längste Fund und wenn nur einer von r1 oder r2 passt, dann wähle ihn.
    - Beispiel:
      - r1='while'
      - r2=letter.(letter|digit)*
      - Eingabe
        - int while48; …
      - Wenn p='while', beide, r1 und r2 passen
      - Wenn p='while48;' weder r1 noch r2 passen
      - Wenn p='while48' nur r2 passt, wähle r2 aus
  - Alternative 2: Regelpriorität
    - Wenn der längste Fund immer noch in einem Konflikt endet, dann wähle den erste regulären Ausdruck aus der lexikalischen Definition der Sprache

Weitere spezielle Probleme

- Reservierte Schlüsselwörter
  - Schlüsselwörter dürfen nicht in Bezeichnern (Namen) verwendet werden
- Groß-/Kleinschreibung
  - intern nur eine Repräsentation verwenden, weshalb eine Anpassung notwendig ist
- Textende
  - Das Textende muss dem Syntaxanalysator mitgeteilt werden, weshalb ein eind-of-text Symbol (eot) eingefügt werden muss
- Vorgriff (lookahead) um mehrere Zeichen
  - gelesene aber nicht verwendete Zeichen müssen für nächsten Test berücksichtigt werden
- Lexikalische Fehler
  - Die Verletzung der Syntax (z.B. falscher Wertebereich) wird gemeldet und trotzdem an den Syntaxanalysator weitergegeben

Implementierung eines lexikalischen Analysators

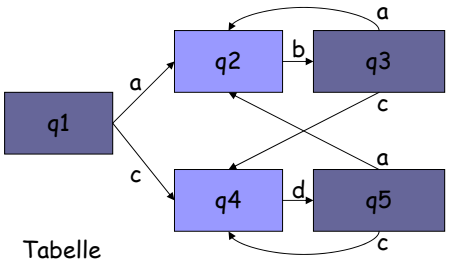
- Wie übertragen wir reguläre Ausdrücke in einen lexikalischen Analysator?
  - Konvertiere reguläre Ausdrücke zu einem deterministischen endlichen (finite) Automaten (DFA)
    - Warum??? DFAs sind einfacher zu simulieren als reguläre Ausdrücke
  - Schreibe ein Programm zum Simulieren eines DFAs
    - Der DFA **erkennt** die Tokens im Eingabetext und wird der lexikalische Analysator
    - Wenn ein Token erkannt wurde, dann kann eine benutzerdefinierte Aktion ausgeführt werden
      - z.B. überprüfe, ob der Wert einer Ganzzahlkonstante in eine 32-bit integer passt
  - Die Konvertierung von regulären Ausdrücken zu DFAs und das Schreiben eines Programms zum Simulieren des DFA kann entweder **von Hand** vorgenommen werden oder von einem anderen Programm, welches **lexikalischer Analysegenerator** genannt wird.



Bau des lexikalischen Analysators: DFA zu Code

- DFAs können effizient simuliert werden indem ein tabellenbasierter Algorithmus verwendet wird

```
void dfa (char *in) {
  s = in;
  state = start_state;
  while(1) {
    c = *s++;
    state = table[state][c];
    if (final[state]]) {
      printf("Accepted %s\n",in);
      break;
    }
  }
}
```



Tabelle

	a	b	c	d
q1	q2	q6	q4	q6
q2	q6	q3	q6	q6
q3	q2	q6	q4	q6
q4	q6	q6	q6	q5
q5	q2	q6	q4	q6
q6	q6	q6	q6	q6

Bau des lexikalischen Analysators: Letzter Schritt

- DFA Simulatorcode wird der Kern des lexikalischen Analysators
- Wenn der DFA in einem Endzustand ist
  - Führe mit dem letzten, passenden regulären Ausdruck, entsprechend dem **längsten Fund** und/oder der **Regelpriorität**, die verbundene, benutzerdefinierte Aktion aus
  - Merke aktuelle Stelle im Eingabestrom und gebe Token an Tokenkonsument (parser) weiter

Eine reale JLex lexikalische Spezifikation für einen Kalkulatorsprache

```
1
2       import java.io.IOException;
3
4       %%
5
6       %public
7       %class Scanner
8
9       %type   void
10      %eofval{
11              return;
12      %eofval}
13      %{      public static void main (String args []) {
14        Scanner scanner = new Scanner(System.in);
15        try {
16          scanner.yylex();
17        } catch (IOException e) { System.err.println(e); }
18       }
19      %}
20
21      comment = ("#".*)
22      space   = [\ \t\b\015]+
23      digit   = [0-9]
24      integer = {digit}+
25      real    = ({digit}+"."{digit}*|{digit}*"."{digit}+)
26      IF      = ("if")
27      THEN    = "then"
28      ELSE    = else
29
30      %%
31
32      {space}         { System.out.println("space");
33              break;
34          }
35      {comment}       { System.out.println("comment");
36              break;
37          }
38      {integer}       { System.out.println("Integer CONSTANT\t" + yytext());
39              break;
40          }
41      {real}          { System.out.println("REAL    CONSTANT\t" + yytext());
42              break;
43          }
44      {IF}            { System.out.println("IF      Token\t" + yytext());
45              break;
46          }
47      {THEN}          { System.out.println("THEN    Token\t" + yytext());
48              break;
49          }
50      {ELSE}          { System.out.println("ELSE    Token\t" + yytext());
51              break;
52          }
53      \n              { System.out.println("NL");
54              break;
55          }
56      "+"             { System.out.println("ADD");
57              break;
58          }
59      "-"             { System.out.println("SUB");
60              break;
61          }
62      "*"             { System.out.println("MUL");
63              break;
64          }
65      "/"             { System.out.println("DIV");
66              break;
67          }
68      "%"             { System.out.println("MOD");
69              break;
70          }
71      "("             { System.out.println("LPAR");
72              break;
73          }
74      ")"             { System.out.println("RPAR");
75              break;
76          }
77      .               { System.out.println("error" + "+" + yytext() + "+");
78              break;
79          }
```


Rückblick

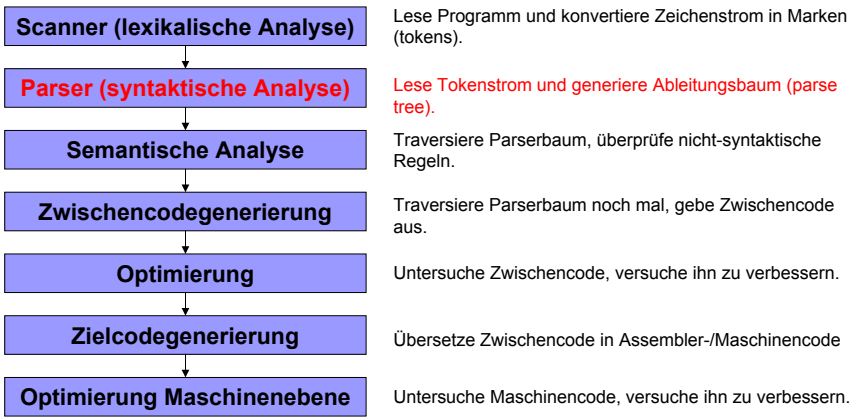
- Wichtige Algorithmen
  - *Konvertierung von regulären Ausdrücken zu nichtdeterministischen endlichen Automaten (NFA) (inklusive Beweise)*
  - *Konvertierung von nichtdeterministischen endlichen Automaten zu deterministischen endlichen Automaten (DFA)*
  - Tabellenbasierte Simulation von DFAs
- Lexikalische Analyse und Scanner
  - Verwenden reguläre Ausdrücke zur Definition der lexikalischen Struktur (Symbole/Token) einer Sprache
  - Verwenden die Theorie der regulären Sprachen zur Erzeugung eines Scanners ausgehend von der Beschreibung der lexikalischen Struktur einer Programmiersprache anhand von regulären Ausdrücken

Zu reguläre Sprachen, reguläre Ausdrücke, und (deterministische und nichtdeterministische) endliche Automaten siehe vorheriges Kapitel

Überblick

- Einführung
- Lexer
- **Parser**
- Zusammenführung (Bau eines ausführbaren Programms)

Der Kompilationsprozess (-phasen)

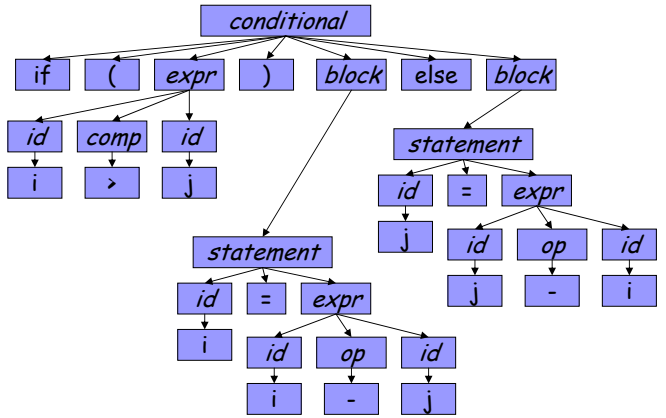


Syntaktische Analyse

- Die lexikalische Analyse erzeugt einen Strom von Symbolen (Tokens)
- Falsche Detailebene für die semantische Analyse und Codegenerierung
- Die Syntaxanalyse gruppiert eine Zeichenfolge von Tokens in **Ableitungsbäume (Struktur-/Parser-/Syntaxbäume)**, was durch die **kontextfreie Grammatik** gelenkt wird, die die **Syntax** der zu kompilierenden Sprache spezifiziert
  - `conditional -> if ( expr ) block else block`
- Ableitungsbäume repräsentieren die **Phrasenstruktur** eines Programms und sind die Grundlage für die semantische Analyse und Codegenerierung
- Die Compilerkomponente zum Ausführen der syntaktischen Analyse ist der **Parser**, welcher oftmals ausgehend von höheren Spezifikationen automatisch generiert wird

Beispiel Syntaxanalyse

Token		
if	(	i
>	j	)
i	=	i
-	j	;
else	j	=
j	-	i
;	}	



Syntaxbeschreibung

- Warum können wir **nicht** reguläre Ausdrücke zum Beschreiben der Syntax einer Programmiersprache verwenden?
- Betrachte die folgenden Beschreibungen:

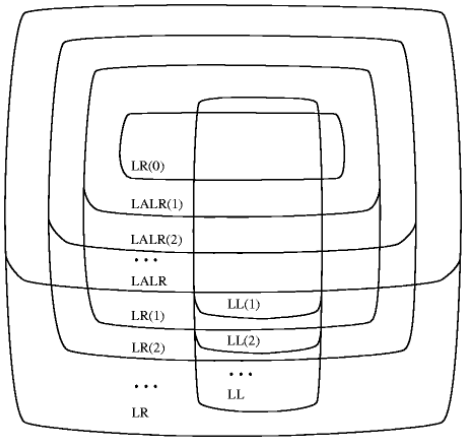
- Identitäten:
  - digit=[0-9]
  - letter=[a-z]
  - id=letter.(letter|digit)*
- Kann Identitäten von id durch Substitution entfernen:
  - id=[0-9].([a-z]|[0-9])*
  - id ist ein regulärer Ausdruck

- Identitäten:
  - digits=[0-9]+
  - sum=expr.'+'.expr
  - expr=('('.sum.')') | digits
- Kann **nicht** Identitäten von expr durch Substitution entfernen:
  - expr ist durch **Rekursion** definiert
  - expr ist kein regulärer Ausdruck

Grammatiken und Parser

- Kontextfreie Grammatiken **erzeugen** durch den Ableitungsprozess Strings (oder Sätze)
- Die kontextfreien Sprachen sind definiert durch
  - L_{CF}={L(G): Alle kontextfreien Grammatiken G}
  - Mit anderen Worten: Die Menge aller Sprachen von allen kontextfreien Grammatiken
- Ein Parser für eine kontextfreie Grammatik **erkennt** Strings in der Grammatiksprache
- Parser können automatisch aus einer kontextfreien Grammatik generiert werden
- Parser zum Erkennen von allgemeinen kontextfreien Sprachen können langsam sein
- Parser, die nur eine Untermenge von kontextfreien Sprachen erkennen können, können so gestaltet werden, dass sie schneller sind (deterministische Kellerautomaten)

Klassen von Grammatiken und Parsern



- LL(k) Parser
  - Eingabe wird von **links-nach-rechts** (1. L) abgearbeitet
  - **linksseitige Ableitung** (2. L)
  - "top down" oder "prädiktive" (voraussagende) Parser genannt
- LR(k) parsers
  - Eingabe wird von **links-nach-rechts** (1. L) abgearbeitet
  - **rechtsseitige Ableitung** (2. R)
  - "bottom up" oder "schiebe-reduziere" (shift-reduce) Parser genannt
- "k" steht für die Anzahl von Symbolen (token) für die in der Eingabe **vorausgeschaut** werden muss um eine Entscheidung treffen zu können
  - LL(k) – welche nächste Produktion auf der rechten Seite ist bei einer linksseitigen Ableitung zu wählen
  - LR(k) – ob zu schieben oder reduzieren

Top-down versus Bottom-up Syntaxanalyse (1/2)

- Top-down oder LL-Syntaxanalyse
  - Baue den Ableitungsbaum von der Wurzel aus bis hinunter zu den Blättern auf
  - Berechne in jedem Schritt voraus welche Produktion zu verwenden ist um den aktuellen nichtterminalen Knoten des Ableitungsbaumes aufzuweiten (expand), indem die nächsten k Eingabesymbole betrachtet werden
- Bottom-up oder LR-Syntaxanalyse
  - Baue den Ableitungsbaum von den Blättern aus bis hinauf zu der Wurzel auf
  - Ermittle in jedem Schritt, ob eine Kollektion von Ableitungsbaumknoten zu einem einzelnen Vorgängerknoten zusammengefasst werden kann oder nicht

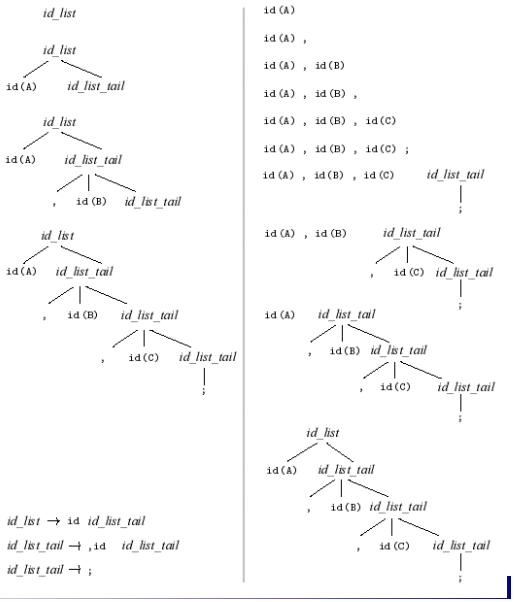
Top-down versus Bottom-up Syntaxanalyse (2/2)

- Grammatik:

id_list -> **id** id_list_tail
id_list_tail -> **,** **id** id_list_tail
id_list_tail -> **;**

- Beispiel Strings:

A;
A, B, C;



Syntaxanalyse durch rekursiven Abstieg (1/4)

- Rekursiver Abstieg ist ein Weg um LL (top-down) Parser zu implementieren
- Es ist einfach von Hand zu schreiben
  - Es wird kein Parsergenerator benötigt
- Jedes nichtterminale Symbol in der Grammatik hat einen Prozeduraufruf
- Es muss im Stande sein die nächste, anzuwendende, linksseitige Ableitung zu bestimmen (predict), indem nur die nächsten k Symbole angeschaut werden
  - k ist üblicherweise 1

Syntaxanalyse durch rekursiven Abstieg (2/4)

- Rekursiver Abstieg ist ein Weg um LL(1)-Parser zu implementieren:
  - Erinnerung: LL(1)-Parser machen linksseitige Ableitungen, unter Verwendung von höchstens 1 Symbol in der Vorausschau, um zu entscheiden welche rechte Seite einer Produktion verwendet wird, um ein linksseitiges Nichtterminal in einer Satzform zu ersetzen.

- LL(1)-Parser Beispiel:
  - Grammatikfragment:
    - *factor* -> (*expr*) | [*sexpr*]
  - Wenn die Satzform “n1 … nk factor sm … sn” lautet, dann sollte die nächste Satzform folgende sein:
    - “n1 … nk (expr) sm … sn” oder
    - “n1 … nk [sexpr] sm … sn”

Syntaxanalyse durch rekursiven Abstieg (3/4)

Grammatik für eine Kalkulatorsprache

program ⟶ *stmt_list* $$
stmt_list ⟶ *stmt stmt_list* | ϵ
stmt ⟶ `id` := *expr* | `read id` | `write` *expr*
expr ⟶ *term term_tail*
term_tail ⟶ *add_op term term_tail* | ϵ
term ⟶ *factor factor_tail*
factor_tail ⟶ *mult_op factor factor_tail* | ϵ
factor ⟶ (*expr*) | `id` | `literal`
add_op ⟶ + | -
mult_op ⟶ * | /

Beispieleingabe:

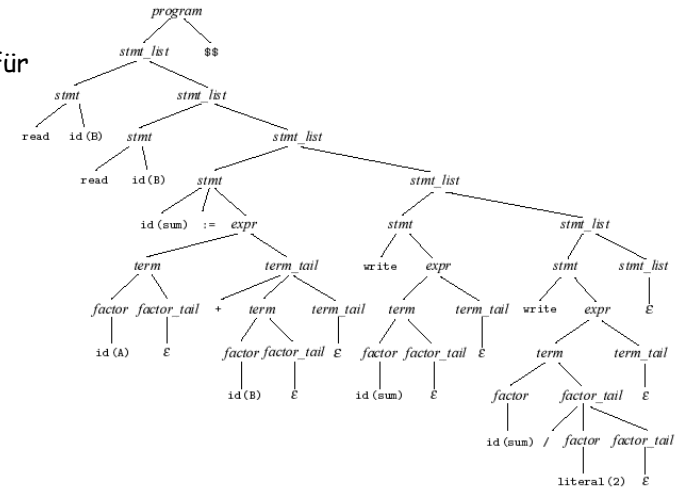
```
read A
read B
sum := A + B
write sum
write sum / 2
```

```
procedure match (expected)
    if input_token = expected
        consume input_token
    else error

-- this is the start routine:
procedure program
    case input_token of
        id, read, write, $$ :
            stmt_list
            match ($$)
        otherwise error

procedure stmt_list
    case input_token of
        id, read, write : stmt; stmt_list
        $$ : skip       -- epsilon production
        otherwise error

procedure stmt
    case input_token of
        id : match (id); match (:=); expr
        read : match (read); match (id)
        write : match (write); expr
        otherwise error

procedure expr
    case input_token of
        id, literal, ( : term; term_tail
        otherwise error

procedure term_tail
    case input_token of
        +, - : add_op; term; term_tail
        ), id, read, write, $$ :
            skip       -- epsilon production
        otherwise error

procedure term
    case input_token of
        id, literal, ( : factor; factor_tail
        otherwise error

procedure factor_tail
    case input_token of
        *, / : mult_op; factor; factor_tail
        +, -, ), id, read, write, $$ :
            skip       -- epsilon production
        otherwise error

procedure factor
    case input_token of
        id : match (id)
        literal : match (literal)
        ( : match (( ); expr; match ( ) )
        otherwise error

procedure add_op
    case input_token of
        + : match (+)
        - : match (-)
        otherwise error

procedure mult_op
    case input_token of
        * : match (*)
        / : match (/)
        otherwise error
```


Syntaxanalyse durch rekursiven Abstieg (4/4)

Ableitungsbaum für Beispieleingabe:

```
read A
read B
sum := A + B
write sum
write sum / 2
```



LL-Syntaxanalyse

- Finde zu einer Eingabe von Terminalsymbolen (tokens) passende Produktionen in einer Grammatik durch Herstellung von linksseitigen Ableitungen

- Für eine gegebene Menge von Produktionen für ein Nichtterminal, $X \to y_1|\ldots|y_n$, und einen gegebenen, linksseitigen Ableitungsschritt $\alpha X \beta \Rightarrow \alpha \gamma_i \beta$, müssen wir im Stande sein zu bestimmen welches γ_i zu wählen ist indem nur die nächsten k Eingabesymbole angeschaut werden

- Anmerkung:
  - Für eine gegebene Menge von linksseitigen Ableitungsschritten, ausgehend vom Startsymbol $S \Rightarrow \alpha X \beta$, wird der String von Symbolen α nur aus Terminalen bestehen und repräsentiert den passenden Eingabeabschnitt zu den bisherigen Grammatikproduktionen

Probleme mit der LL-Syntaxanalyse (1/4)

- Linksrekursion
  - Produktionen von der Form:
    - A -> Aα
    - A -> β
  - Wenn eine Grammatik linksrekursive Produktionen enthält, dann kann es dafür keinen LL Parser geben
    - LL Parser würden in eine Endlosschleife eintreten, wenn versucht wird eine linksseitige Ableitung in solch einer Grammatik vorzunehmen
  - Linksrekursion kann durch das Umschreiben der Grammatik ausgeschlossen werden
    - A -> βA'
    - A' -> αA' | ϵ

Probleme mit der LL-Syntaxanalyse (2/4)

- Linksrekursion: Nicht formale Rechtfertigung
  - Originalgrammatik:
    - A -> Aα
    - A -> β
  - Ableitungen:
    $A \Rightarrow A\alpha \Rightarrow A\alpha\alpha \Rightarrow A\alpha\alpha\alpha \Rightarrow^* \beta\alpha\alpha\alpha\ldots$
  - Umgeschriebene Grammatik:
    - A -> βA'
    - A' -> αA' | ϵ
  - Ableitungen:
    $A \Rightarrow \beta A' \Rightarrow \beta\alpha A' \Rightarrow \beta\alpha\alpha A' \Rightarrow^* \beta\alpha\alpha\alpha\ldots$
- Linksrekursion: Beispiel
  - Grammatik:
    - id_list -> id_list_prefix ;
    - id_list_prefix -> id_list_prefix , id | id
  - Linksrekursion kann durch das Umschreiben der Grammatik ausgeschlossen werden
    - id_list -> id id_list_tail
    - id_list_tail -> , id id_list_tail | ;

Probleme mit der LL-Syntaxanalyse (3/4)

- Gemeinsame Präfixe
  - Tritt auf wenn zwei verschiedene Produktionen mit der gleichen linken Seite mit den gleichen Symbolen anfangen
    - Produktionen der Form:
      - A -> bα
      - A -> bβ
    - LL(1) Parser kann nicht entscheiden welche Regel auszuwählen ist, wenn A in einem linksseitigen Ableitungsschritt zu ersetzen ist, weil beide rechten Seiten mit dem gleichen Terminalsymbol anfangen
  - Kann durch Faktorisierung ausgeschlossen werden:
    - A -> bA'
    - A' -> α | β

Probleme mit der LL-Syntaxanalyse (4/4)

- Gemeinsame Präfixe
  - Beispiel:
    - stmt -> id := expr
    - stmt -> id (argument_list)
  - Gemeinsame Präfixe können durch das Umschreiben der Grammatik ausgeschlossen werden
    - stmt -> id stmt_list_tail
    - stmt_list_tail -> expr | (argument_list)
- Der Ausschluss von Linksrekursion und gemeinsame Präfixe garantiert **nicht** das eine Grammatik LL wird
- Wenn wir keinen LL Parser für eine Grammatik finden können, dann müssen wir einen mächtigere Technik verwenden
  - z.B. LALR(1) – Grammatiken

Bau eines Top-Down Parsers mit rekursivem Abstieg (1/2)

- Für jedes Nichtterminal in einer Grammatik wird ein Unterprogramm erzeugt, welches einem einzelnen linksseitigen Ableitungsschritt entspricht, wenn es aufgerufen wird
  - Beispiel:
    - factor -> (expr)
    - factor -> [sexpr]

```
void factor (void) {
  switch(next_token()) {
    case '(':
      expr(); match(')'); break;
    case '[':
      sexpr(); match(']'); break;
  }
}
```

- Schwieriger Teil:
  - Herausbekommen welches Token den ‚case' Arm vom switch Befehl benennt

Bau eines Top-Down Parsers mit rekursivem Abstieg (2/2)

- **PREDICT**-Mengen (Vorhersagemengen)
  - PREDICT Mengen teilen uns mit, welche rechte Seite einer Produktion bei einer linken Ableitung auszuwählen ist, wenn mehrere zur Auswahl stehen
  - PREDICT-Mengen dienen somit als Grundlage für Ableitungstabellen (Parse-Tabellen) bzw. sind eine andere Teildarstellungsform für die Tabellen
  - Wird in Form von FIRST-, FOLLOW- und NULLABLE-Mengen definiert :
    - Sei A ein Nichtterminal und α beliebig, dann gilt
    - **PREDICT(A->α) = FIRST(α) ∪ FOLLOW(A) wenn NULLABLE(α)**
    - **PREDICT(A->α) = FIRST(α) wenn nicht NULLABLE(α)**
- **NULLABLE-**Mengen
  - Sei X ein Nichtterminal
  - NULLABLE(X) ist wahr wenn gilt $X \Rightarrow^* \epsilon$ (X kann den leeren String ableiten)

FIRST-Mengen

- Sei α eine **beliebige Folge** von Grammatiksymbolen (Terminale und Nichtterminale)
- FIRST(α) ist die Menge **aller Terminalsymbolen** a mit denen ein aus α abgeleiteter String **beginnen** kann:

FIRST(α) ist { a: $\alpha \Rightarrow^* a\beta$}

- Gilt $\alpha \Rightarrow^* \epsilon$, dann ist **auch** ϵ in FIRST(α)

Berechnung von FIRST-Mengen (1/2)

- Für alle Grammatiksymbole X wird FIRST(X) berechnet, indem die folgenden Regeln solange angewandt werden, bis zu keiner FIRST-Menge mehr ein neues Terminal oder ϵ hinzukommt:

1. Wenn X ein **Terminal** ist, dann ist FIRST(X)={X}
2. Wenn $X \rightarrow \varepsilon$ eine **Produktion** ist, dann füge ε zu FIRST(X) hinzu
3. Wenn X **Nichtterminal** und $X \rightarrow Y_1 Y_2 Y_3 \ldots Y_k$ eine **Produktion** ist, dann nehme a zu FIRST(X) hinzu, falls
   (a) a für irgendein i in FIRST(Y_i) und
   (b) ein ε in allen FIRST(Y_1), ..., FIRST(Y_{i-1}) enthalten ist ($Y_1 \ldots Y_{i-1}$ sind alle NULLABLE)

Berechnung von FIRST-Mengen (2/2)

- Folglich gilt:
  - Elemente aus FIRST(Y_1) gehören immer auch zu FIRST(X)
  - Ist ϵ **nicht** aus Y_1 ableitbar (**NICHT NULLABLE**), dann brauch nichts mehr hinzugefügt werden
  - Ist ϵ aus Y_1 ableitbar (**NULLABLE**), dann muss auch FIRST(Y_2) zu FIRST(X) hinzugefügt werden
  - Ist ϵ aus Y_2 ableitbar (**NULLABLE**), dann muss auch FIRST(Y_3) zu FIRST(X) hinzugefügt werden
  - usw.
  - ϵ wird nur zu FIRST(X) hinzugefügt, wenn es in **allen** Mengen FIRST(Y_1), ... ,FIRST(Y_k) enthalten ist

FOLLOW-Mengen

- Sei A ein **Nichtterminal**
- FOLLOW(A) ist die Menge **aller Terminalsymbole** a, die in einer Satzform **direkt rechts neben** A stehen können (sei S Startregel; α, β beliebig):

FOLLOW(A) ist { a:S $\Rightarrow^*$ αAaβ }

- **Achtung**: Zwischen A und a können während der Ableitung Symbole gestanden haben, die aber verschwunden sind, weil aus Ihnen ϵ abgeleitet wurde!
- Gibt es eine Satzform, in der A das am weitesten rechts stehende Symbol ist, dann gehört auch $ (die Endemarkierung) zu FOLLOW(A)

Berechnung von FOLLOW-Mengen

- Follow(A) wird für alle Nichtterminale A berechnet, indem die folgenden Regeln solange angewandt werden, bis keine Follow-Menge mehr vergrößert werden kann:

1. Sei S das Startsymbol und $ die Endemarkierung, dann nehme $ in FOLLOW(S) auf
2. Wenn es eine **Produktion** $A \to \alpha B \beta$ gibt, dann wird jedes Element von FIRST(β) mit Ausnahme von ε **auch** in FOLLOW(B) aufgenommen.
3. Wenn es **Produktionen** $A \to \alpha B$ oder $A \to \alpha B \beta$ gibt und FIRST(β) ε enthält (d.h. $\beta \Rightarrow^* \varepsilon$), dann gehört jedes Element von FOLLOW(A) **auch** zu FOLLOW(B)

Beispiel für NULLABLE-, FIRST- und FOLLOW-Mengen

S -> s$	B -> ϵ	A -> B
S -> A B S	B -> b	A -> a

Schritt 1: i=0

	NULLABLE	FIRST	FOLLOW
A	False	{**a**}	{}
B	True	{**b**, **ϵ**}	{}
S	False	{**s**}	{**$**}

Schritt 2: i=1

	NULLABLE	FIRST	FOLLOW
A	True	{a,**b**,ϵ}	{**b**}
B	True	{b, ϵ}	{**s**}
S	False	{s,**a**}	{$}

Schritt 3: i=2

	NULLABLE	FIRST	FOLLOW
A	True	{a,b,ϵ}	{b,**s**}
B	True	{b, ϵ}	{s,**a**}
S	False	{s,a,**b**}	{$}

Schritt 4: i=3

	NULLABLE	FIRST	FOLLOW
A	True	{a,b,ϵ}	{b,s,**a**}
B	True	{b, ϵ}	{s,a,**b**}
S	False	{s,a,b}	{$}

Beispiel für PREDICT-Mengen

S -> s$	B -> ϵ	A -> B
S -> A B S	B -> b	A -> a

	NULLABLE	FIRST	FOLLOW
A	True	{a,b,ϵ}	{b,s,a}
B	True	{b, ϵ}	{s,a,b}
S	False	{s,a,b}	{$}

	PREDICT
A->B	{a,b,ϵ,s}
A->a	{a}
B->ϵ	{a,b,s}
B->b	{b, ϵ}
S->s$	{s}
S->ABS	{a,b,s}

- PREDICT-Mengen zeigen uns welche Menge von look-ahead Symbolen die rechte Seite einer Produktion selektiert
- Diese Grammatik ist NICHT LL(1), da es **duplizierte** Symbole in den PREDICT-Mengen für alle drei Nichtterminale gibt
  - Siehe Hervorhebungen (**dick**, rot)

LL(k) Eigenschaften

- **Satz**: Jede kontextfreie Grammatik G ist genau dann LL(1), wenn für alle Alternativen $A \Rightarrow \alpha_1 | \alpha_2 | ... | \alpha_n$ gilt
  1. $FIRST(\alpha_1)$, ..., $First(\alpha_n)$ paarweise disjunkt,
  2. falls $\alpha_i \Rightarrow^* \epsilon$ gilt, dann $FIRST_1(\alpha_j) \cap FOLLOW_1(A) = \emptyset$ für $1 \le j \le n, j \ne i$
  - In Worten:
    - Aus $\alpha_1, \alpha_2, ...$ und α_n sind **jeweils** keine Strings ableitbar, wo **zwei** mit dem gleichen Nichtterminal anfangen
    - Der leere String ϵ kann **nicht sowohl** aus α_i und α_j für $i \ne j$ abgeleitet werden
    - Falls $\alpha_i \Rightarrow^* \epsilon$ gilt, dann **beginnt kein** aus α_i ableitbarer String mit einem Terminal aus FOLLOW(A)
- **Satz:** Sei G kontextfreie Grammatik, $k \ge 0$. G ist genau dann LL(k), wenn gilt:
  Sind $A \Rightarrow \beta$, $A \Rightarrow \zeta$ verschiedene Produktionen, dann
  $First_k(\beta\alpha) \cap FIRST_k(\zeta\alpha) = \emptyset$ für alle α, σ mit $S \Rightarrow^* \sigma A \alpha$

Ableitungsbäume und Parser mit rekursivem Abstieg (1/2)

- Die Beispielparser auf die wir bisher geschaut haben sind nur Erkenner
  - Sie bestimmen, ob eine Eingabe syntaktisch korrekt ist, aber bauen keinen Ableitungsbaum
- Wie konstruieren wir dann einen Ableitungsbaum?
  - In Parser mit rekursivem Abstieg machen wir für jede nichtterminale Funktion:
    - Konstruktion eines korrekten Ableitungsbaumknoten für sich selbst und Verbindungen zu seinen Kindern
    - Geben den konstruierten Ableitungsbaumknoten an den Aufrufer zurück

Ableitungsbäume und Parser mit rekursivem Abstieg (2/2)

- Beispiel: Jedes nichtterminale Unterprogramm konstruiert einen Ableitungsbaumknoten
  - factor -> (expr)
  - factor -> [sexpr]

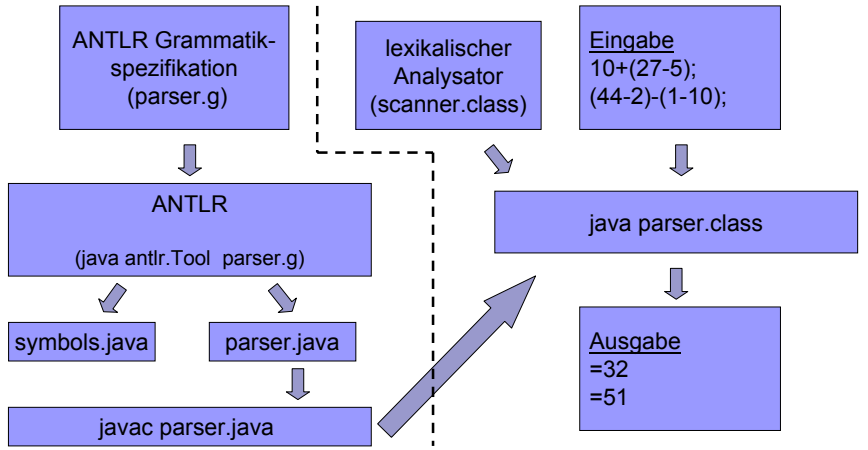
```
node *factor (void) {
  switch(next_token()) {
    case '(':
      node = factor_node(expr());
      match(')'); break;
    case '[':
      node = factor_node(sexpr());
      match(']'); break;
  }
  return node;
}
```

- Nicht alle Symbole werden zu einem Ableitungsbaumknoten
  - Beispiele: '(', ')', '[', ']'
- Diese Art von Ableitungsbaum wird „Abstrakter Syntaxbaum" (abstract syntax tree, AST) genannt

Parsergeneratoren und Syntaxanalyse

- Parsergeneratoren erzeugen ausgehend von der kontextfreien Grammatik einen Parser
- An Produktionen dürfen **semantische Aktionen** angehängt sein
  - Wenn ein Parser eine Produktion erkannt hat, dann wird die semantische Aktion aufgerufen
  - Wird hauptsächlich dazu verwendet einen Ableitungsbaum explizit zu konstruieren
- Die Ausgabe eines Parsergenerators ist ein Programm in einer Hochsprache (z.B. C, C++, oder Java) welches einen Symbolstrom (token stream) von einem Lexer (für die lexikalische Analyse) entgegen nimmt und welches einen Ableitungsbaum für die nachfolgenden Compilerphasen produziert

ANTLR als Beispiel eines Parsergenerators



Rückblick

- Wichtige Algorithmen:
  - Auflösung von Linksrekursion und gemeinsame Präfixe
- Syntaktische Analyse und Parser
  - Die Syntax einer Programmiersprache wird mit KFGs spezifiziert
  - Konstruktion eines Parsers mit rekursivem Abstieg anhand von KFGs
  - Automatische Generierung von Parsern anhand von KFGs
  - Parser erzeugen Ableitungsbäume für die Analyse und weitere Verarbeitung in den nachfolgenden Compilerphasen
- Unwichtig für Klausur, da nur für Satz über LL(k) notwendig
  - FIRST, FOLLOW, NULLABLE und PREDICT Mengen für kontextfreie Grammatiken

Zu kontextfreie Grammatiken (KFG), Sprachen (KFS) und Kellerautomat siehe vorheriges Kapitel

Überblick

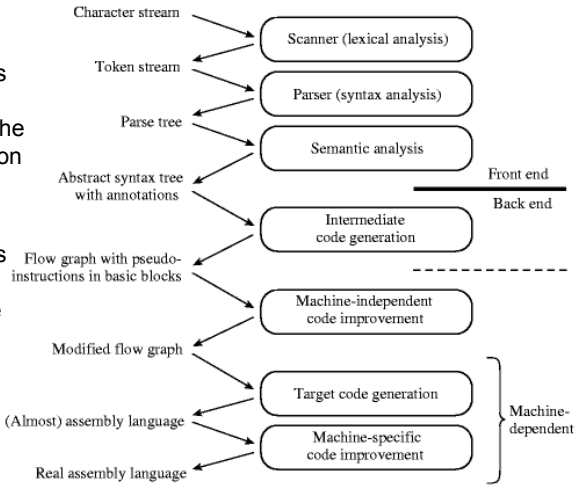
- Einführung
- Lexer
- Parser
- **Zusammenführung (Bau eines ausführbaren Programms)**

Der Kompilationsprozess (-phasen)

Scanner (lexikalische Analyse)	Lese Programm und konvertiere Zeichenstrom in Marken (tokens). **Theorie**: Reguläre Ausdrücke, endlicher Automat
Parser (syntaktische Analyse)	Lese Tokenstrom und generiere Ableitungsbaum (parse tree). **Theorie**: Kontextfreie Grammatiken, Kellerautomat
Semantische Analyse	Traversiere Parserbaum, überprüfe nicht-syntaktische Regeln.
Zwischencodegenerierung	Traversiere Parserbaum noch mal, gebe Zwischencode aus.
Optimierung	Untersuche Zwischencode, versuche ihn zu verbessern.
Zielcodegenerierung	Übersetze Zwischencode in Assembler-/Maschinencode
Optimierung Maschinenebene	Untersuche Maschinencode, versuche ihn zu verbessern.

Die Organisation eines typischen Compilers

- Frontend
  - Führt Operationen aus welche von der zu kompilierenden Sprache abhängen und nicht von der Zielmaschine
- Backend
  - Führt Operationen aus welche etwas Wissen über die Zielmaschine haben müssen



Schreiben des Programms

- Ein kleines Programm, geschrieben in Pascal, welches den größten gemeinsamen Teiler (ggT) von zwei Ganzzahlen berechnet

```
program gcd (input, output);
var i, j : integer;
begin
    read(i,j);
    while i <> j do
        if i > j then i := i - j
        else j :- j - i;
    writeln(i)
end.
```


Vom Text des Quellcodes zu den Tokens

Programmquelltext

```
program gcd (input, output);
var i, j : integer;
begin
   read(i,j);
   while i <> j do
      if i > j then i := i - j
      else j :- j - i;
   writeln(i)
end.
```

Tokens

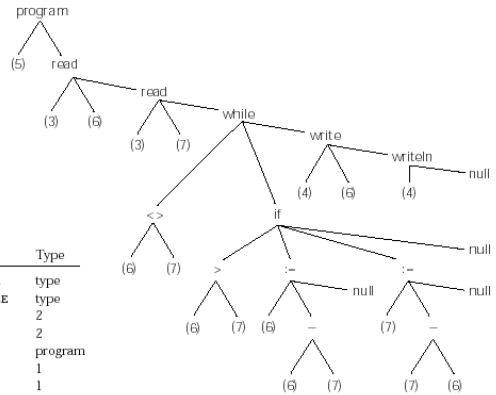
program	gcd	(
input	,	output
)	;	var
i	,	j
:	integer	;
begin	read	(
i	,	j
)	;	while
i	<>	j
do	if	i
…		

Von den Tokens zum Ableitungsbaum

Programmquelltext

```
program gcd (input, output);
var i, j : integer;
begin
   read(i,j);
   while i <> j do
      if i > j then i := i - j
      else j :- j - i;
   writeln(i)
end.
```

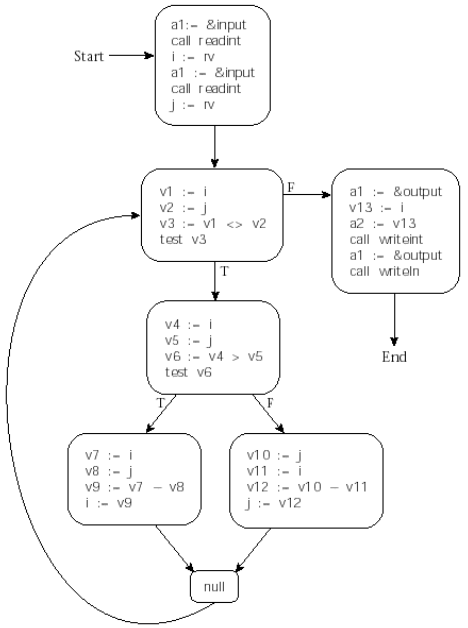
Ableitungsbaum und Symboltabelle



Index	Symbol	Type
1	INTEGER	type
2	TEXTFILE	type
3	INPUT	2
4	OUTPUT	2
5	GCD	program
6	I	1
7	J	1

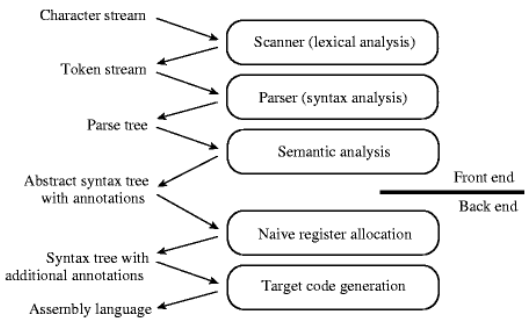
Zwischencode

- Der Ableitungsbaum wird zu einem **Kontrollflussgraphen** konvertiert
- Die Knoten des Kontrollflussgraphen sind **grundlegende Blöcke** und enthalten eine **pseudo-Assemblersprache**
- Die Kontrolle kann einen grundlegenden Block **nur** vom **Anfang** betreten und kann in **nur** am **Ende** wieder verlassen



Nicht optimierende Compiler

- Der Kontrollflussgraph wird zum **Zielcode** der Zielmaschine konvertiert
- Der Zielcode ist eine andere pseudo-Assemblersprache
- Der Kontrollfluss wird ausführlich gemacht durch:
  - Bezeichnen der Anfänge der grundlegenden Blöcke
  - Konvertieren der Kontrollflusskanten zu Sprung- (branch), Aufruf- (call) und Rückkehrinstruktionen
- Virtuelle Register werden durch reale Register ersetzt



Zielcode

- Der Zielcode ist beinahe Assemblercode
  - Der Kontrollfluss ist ausführlich
  - Der Code referenziert nur reale Registernamen
  - Anweisungen zum Speicherreservieren sind vorhanden
- Zielcode ist einfach zu Assemblercode zu übersetzen

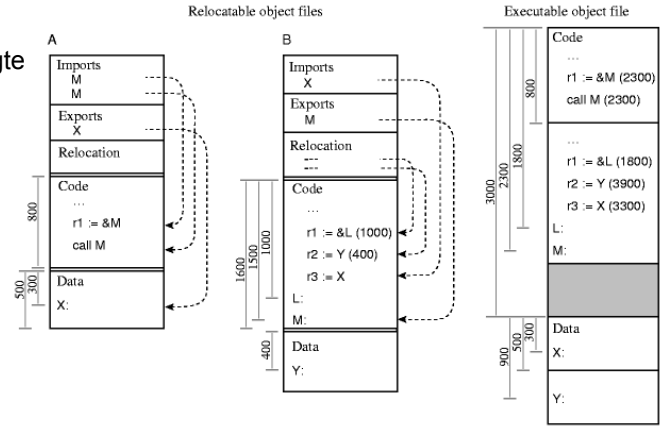
```
    -- first few lines generated during symbol table traversal
    .data          -- begin static data
    .word i        -- reserve one word to hold i
    .word j        -- reserve one word to hold j
    .text          -- begin text (code)
    -- remaining lines accumulated into program.code
main:
    a1 := &input -- "input" and "output" are file control blocks
                 -- located in a library, to be found by the linker
    call readint -- "readint", "writeint", and "writeln" are library subroutines
    i := rv
    a1 := &input
    call readint
    j := rv
    goto L1
L2: r1 := i      -- body of while loop
    r2 := j
    r1 := r1 > r2
    if r1 goto L3
    r1 := j      -- "else" part
    r2 := i
    r1 := r1 – r2
    j := r1
    goto L4
L3: r1 := i      -- "then" part
    r2 := j
    r1 := r1 – r2
    i := r1
L4:
L1: r1 := i      -- test terminating condition
    r2 := j
    r1 := r1 <> r2
    if r1 goto L2
    a1 := &output
    r1 := i
    a2 := r1
    call writeint
    a1 := &output
    call writeln
    goto exit    -- return to operating system
```

Vom Zielcode zum Assemblercode

- Normalerweise einfach:
  - r10 := r8 + r9 -> add $10, $8, $9
  - r10 := r8 + 0x12 -> addi $10, $8, 0x12
- Manchmal auch zu einer Folge von Instruktionen erweitert:
  - r14 := 0x12345abc ->
    - lui $14, 0x1234
    - ori $14, 0x5abc

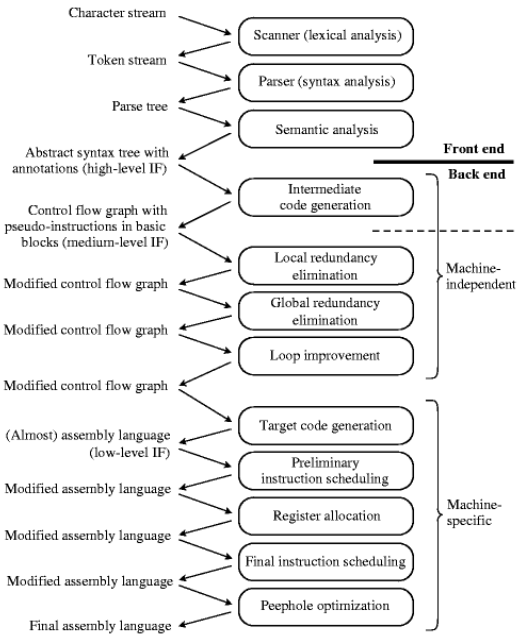
Binden (linking)

- Beim Binden werden mehrere durch den Assembler erzeugte **Objektdateien** zu einer einzelnen, **ausführbaren Datei** kombiniert, welche durch ein Betriebssystem lauffähig ist



Optimierende Compiler

- Optimierung ist ein komplexer Prozess
- Verbessert die Qualität des generierten Codes auf Kosten von zusätzlicher Compilezeit
- Optimierer sind schwierig zu schreiben und einige Optimierungen verbessern das fertige Programm vielleicht nicht
- **Praxistip**: Die jeweilige Einstellung der Optimierungstiefe eines Compilers genau auf korrekte Funktion kontrollieren, da diese öfters fehleranfällig sind (lieber erstmal auf Optimierung verzichten und erst am Ende austesten!)



Peephole-Optimierung

- Sehe dir den Zielcode an, wenige Instruktionen gleichzeitig, und versuche einfache Verbesserungen zu machen
- Versucht kurze, sub-optimale Folgen von Instruktionen zu finden und ersetzt diese Sequenzen mit einer „besseren“ Sequenz
- Sub-Optimale Sequenzen werden durch Muster (patterns) spezifiziert
  - Meistens heuristische Methoden — Es gibt keinen Weg um zu überprüfen ob das Ersetzen eines sub-optimalen Musters tatsächlich das endgültige Programm verbessern wird
- Einfach und ziemlich wirksam

Peephole-Optimierungstypen (1/3)

- **Redundante load/store Instruktionen beseitigen**

```
r2 := r1 + 5                    r2 := r1 + 5
i  := r2          wird zu       i  := r2
r3 := i                         r4 := r2 x 3
r4 := r3 x 3
```

- **Konstantenfaltung (constant folding)**

```
r2 := 3 x 2       wird zu       r2 := 6
```

- **Entfernung gemeinsamer Teilausdrücke (common subexpression elimination)**

```
r2 := r1 x r5                   r4 := r1 x r5
r2 := r2 + r3     wird zu       r2 := r4 + r3
r3 := r1 x r5                   r3 := r4
```


Peephole-Optimierungstypen (2/3)

- **Fortpflanzung von Konstanten (constant propagation)**

```
r2 := 4                         r2 := 4
r3 := r1 + r2     wird zu       r3 := r1 + 4     und    r3 := r1 + 4
r2 := …                         r2 := …                 r2 := …
```

- **und auch**

```
r2 := 4
r3 := r1 + r2     wird zu       r3 := r1 + 4     und
r3 := *r3                       r3 := *r3               r3 := *(r1+4)
```

- **Fortpflanzung von Zuweisungen (copy propagation)**

```
r2 := r1                        r2 := r1
r3 := r1 + r2     wird zu       r3 := r1 + r1    und    r3 := r1 + r1
r2 := 5                         r2 := 5                 r2 := 5
```


Peephole-Optimierungstypen (3/3)

- **Algebraische Vereinfachung (strength reduction)**

```
r1 := r2 x 2                    r1 := r2 + r2
r1 := r2 / 2      wird zu       r1 := r2 >> 1    oder   r1 := r2 << 1
```

- **Beseitigung von unnötigen Instruktionen**

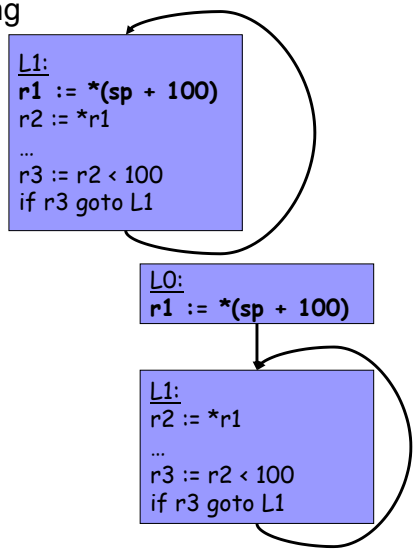
```
r1 := r1 + 0
r1 := r1 - 0      wird zu       (wird komplett beseitigt)
r1 := r1 * 1
```


Komplexe Optimierungen

- Es ist für den Optimierer erforderlich den Datenfluss zwischen Registern und Speicher zu „verstehen“
  - Wird durch **Datenflussanalyse** (data flow analysis) bestimmt
    - Beispiel: Finde die Variablenmenge welche einen grundlegenden Block auf dem Weg zu einem anderen Block „durchfliesst“ und von diesem anderen Block verwendet wird (**lebendige Variablen**)
  - Ist wichtig für Optimierungen welche datenverändernde Instruktionen einfügen, löschen oder bewegen
  - Der ursprüngliche Programmdatenfluss kann nicht geändert werden!
- Es ist für den Optimierer erforderlich die Struktur des Kontrollflussgraphen (control flow graph) zu „verstehen“
  - Wird durch **Kontrollflussanalyse** bestimmt
  - Ist wichtig für die Leistungsoptimierung von Schleifen oder schleifenähnlichen Strukturen in dem Kontrollflussgraphen

Beispiel für komplexe Optimierung

- Schleifeninvarianter Code (Loop Invariant Code Motion)
  - Bewege Berechnungen, dessen Werte während allen Schleifeniterationen gleich bleiben (invariant sind), aus der Schleife raus



Rückblick

- Compiler sind komplexe Programme welche eine höhere Programmiersprache in eine Assemblersprache umwandeln um danach ausgeführt zu werden
- Compilerprogrammierer handhaben die Komplexität des Kompilationsprozesses durch:
  - aufteilen des Compilers in unterschiedliche Phasen
  - verwenden, ausgehend von der Spezifikation, eine Theorie für den Bau von Compilerkomponenten