Compilerbau für die Common Language Run-Time

Syntaxanalyse

Grammatikgetriebener Compilerbau

- Grammatik Teil der Sprachdefinition
- Grammatik liefert Liste der Tokenklassen
- Grammatik hilft bei Strukturierung der abstrakten Syntax
- Parser kann automatisch aus Grammatik generiert werden (compiler compiler)
  - Sprachdefinition und Implementierung stimmen besser überein
  - Anpassung des Compilers bei Sprachänderung wird leichter

Parseralgorithmen

- universell und langsam
  - erlauben beliebige kontextfreie Grammatiken
  - Cocke-Younger-Kasami
  - Earley
- eingeschränkt und schnell
  - schränken Grammatiken ein
    - “künstliche” Syntaxfehler bei Konflikten
  - LALR(1)
  - LL(1)
- top-down vs. bottom-up

Mehrdeutige Grammatiken

- Mehrere verschiedene Parsebäume für die gleiche Eingabe

expr → variable | term | expr * term

term → variable | term + variable

- Grammatikkonflikte
  - “echte” Mehrdeutigkeit: Programme haben verschiedene Semantik
  - “syntaktische” Mehrdeutigkeit: Parsebäume sind nur strukturell verschieden
  - Parserbeschränkung: falsche Parserentscheidung führt letztlich zu Syntaxfehler

Eliminierung von Mehrdeutigkeiten

stmt → **if** expr **then** stmt
| **if** expr **then** stmt **else** stmt
| **other**

stmt → matched_stmt | unmatched_stmt
matched_stmt →
if expr **then** matched_stmt **else** matched_smt
| **other**
unmatched_stmt →
if expr **then** stmt
| **if** expr **then** matched_stmt **else** unmatched_stmt

Eliminierung von Linksrekursion

- Wichtig für Top-Down-Parser

$E \rightarrow E \; \mathbf{+} \; T \mid T$

$T \rightarrow T \; \mathbf{*} \; F \mid F$

$F \rightarrow \mathbf{(} \; E \; \mathbf{)} \mid \mathbf{id}$

- Idee: Überführung von $A \rightarrow A\,a \mid b$ in

$A \rightarrow b\,A'; \qquad A' \rightarrow a\,A' \mid \varepsilon$

$E \rightarrow T\,E' \qquad E' \rightarrow \mathbf{+}\;T\,E' \mid \varepsilon$

$T \rightarrow F\,T' \qquad T' \rightarrow \mathbf{*}\;F\,T' \mid \varepsilon$

$F \rightarrow \mathbf{(}\;E\;\mathbf{)} \mid \mathbf{id}$

Linksfaktorierung

- Problem: Zwei Alternativen fangen mit der gleichen Symbolfolge an

stmt → **if** expr **then** stmt **else** stmt
| **if** expr **then** stmt
| **other**

- Idee: Auslagerung gemeinsamer Anfangsstücken

stmt → **if** expr **then** stmt stmt' | **other**

stmt' → **else** stmt | ε

Top-Down-Parsing

- Rekursiver Abstieg
  - rekursive Funktionen, die Tokens konsumieren
  - Backtracking: Sequentielles Durchprobieren aller Alternativen
- Prädiktiver Parser
  - mit Lookahead wird Alternative der Grammatik ausgewählt
- LL(1)
  - Grammatikklasse, die für prädiktive Parser geeignet ist

Prädiktiver Parser: Beispiel

E → T E1 E1 → **+** T E1 | ε

T → F T1 T1 → ***** F T1 | ε F → **(** E **)** | **id**

```
def parse_E():
    return parse_T() and parse_E1();
def parse_E1():
    if lookahead==“+”:
        return consume(“+”) and parse_T() and parse_E1()
    return True
def parse_F():
    if lookahead==“(“:
        return consume(“(”) and parse_E() and consume(“)”)
    if lookahead == TOK_id:
        return consume(TOK_id)
    return False
```

FIRST und FOLLOW

- FIRST: Menge aller Terminalsymbole, mit der eine Regel beginnen kann
  - FIRST(a S1 S2 S3): a
  - FIRST(H S1 S2 S2) == FIRST(H)
    - Falls H => ε, dann muss auch FIRST(S1 S2 S3) berücksichtig werden
  - FIRST(H) = **U** FIRST(r) (r ist Alternative von H)
- FOLLOW(H): Menge aller Terminale, die in irgendeiner Ableitung von S auf H folgen können

LL(1)

- Left-to-right scan
- Leftmost derivation
- 1 Token Lookahead
- Wenn A → x | y Regeln sind, so gilt
  - Für kein Terminal a sind beginnen Ableitungen von x und y mit a (FIRST(x) und FIRST(y) sind disjunkt)
  - Höchstens eines von x und y lässt sich als ε ableiten
  - Falls y =*> ε, dann beginnt keine Ableitung von x mit einem Terminal aus FOLLOW(y)

LL(1)-Parser

- Parser: Input I, Stack S, Parse Table M
  1. Stack ist initial das Startsymbol
  2. Lies t aus Input
  3. Wenn TOS ein Terminal ist
     - Wenn TOS == t: Konsumiere t, pop()
     - sonst: Syntaxfehler
  4. Sonst: Wenn M[TOS, t] == TOS → Y1 Y2 Y3
     - pop()
     - push(Y3), push(Y2), push(Y1)
  5. Sonst: Syntaxfehler
  6. Setze mit 2. fort

Tabellenkonstruktion

- Für jedes Hilfssymbol H:
  - Für jedes Token t:
    - M[H,t] = H → Y1 Y2…, falls t ∈ FIRST(Y1 Y2…)
    - M[H,t] = H → ε, falls t ∈ FOLLOW(H)
    - M[H,t] = error sonst

LL(1)-Generatoren

- COCO/R: LL(k)
  - generiert Java, C#, C++, Delphi, Modula-2, Oberon, …
  - integriert Scanner und Parser
    - Parser wird in rekursiv absteigende Funktionen übersetzt
  - erweiterte BNF:
    - option: [h1 h2 h3]
    - iteration: { h1 h2 h3 }
  - Semantische Aktionen
    - (. Console.WriteLine(“n = “ + n); .)
  - Parameter für Hilfssymbole

    expr<out int value> =

    term<out T> expr1<out rest> (. value=combine(T, rest) .)

COCO/R-Konfliktlösung

- Grammatikumformulierung
  - u.U. Änderung der Sprache
  - u.U. Verschlechterung der Lesbarkeit
- expliziter Konfliktlöser

```
UsingClause = "using" [IF(IsAlias()) ident '='] Qualident ';'.
```

  - IsAlias ist nutzerdefiniert und liest zwei Token voraus

LL(k)-Generatoren

- ANTLR
  - Eigentliches Ziel: LL(k)
  - Tatsächlich realisiert: “linear approximation” LL(k)
- Lexik, Syntax, abstrakte Syntax
- Generiert Java, C++, C#
- EBNF
- Prädikate: syntaktische und semantische Konfliktlösung

YACC

- Yet Another Compiler Compiler
- Bison: GNU-Neuimplementierung
- LALR(1) (Bison: auch GLR)
- Eingabe: BNF + semantische Aktionen
  - Konvertierung von EBNF in BNF

LR(1)

- LR: Left-to-right scan, rightmost derivation
- Verschiedene Algorithmenvarianten
  - SLR: Simple LR
  - GLR: Generalized LR
  - LALR: Lookahead LR
- Parser: Input, Zustand, Stack, Aktionstabelle
  - Zustände sind “dotted productions”
  - Aktionen: Shift, Reduce, Goto
    - Für jeden Zustand und jedes Token eine Aktion
    - Shift konsumiert ein Token
    - Reduce ersetzt Symbole auf Stack; potentiell gefolgt von goto

Mehrdeutige Grammatiken

- Shift/Reduce-Konflikt:
  - In einem Zustand lässt sich eine Regel anwenden, oder aber das Parsen in einer anderen Regel fortsetzen
  - yacc: gelöst zugunsten von shift
    - entspricht üblicherweise der Sprache
    - Vorrang-Deklarationen (precedence) steuern Konfliktlösung
- Reduce/Reduce-Konflikt:
  - Mehrere Regeln alternativ anwendbar
  - yacc: gelöst zugunsten der textuell ersten Regel
  - Reduce/Reduce-Konflikte sollten behoben werden
- Konfliktlösung:
  - Umformulierung der Grammatik
  - Vergabe von Vorrängen

Parsing mit ANTLR

- Parsersyntax:

  class *Klassenname* extends Parser;

  //optional: Parser(“de.loewis.Basisklasse”)

  options

  tokens

  parser rules

- Parseroptionen:
  - k=2; // Lookahead
  - buildAST=true;
  - ...

Grammatikregeln

- EBNF

```
basicDecl :
      varDecl
    | constDecl
    | typeDecl
    ;
varDecl: “var” identList COLON typeName
     (BECOMES constantValue)? SEMI
;
```

-

Prädikate

- Auflösung von Mehrdeutigkeiten
  - ermöglicht Verzicht auf Refaktorierung
  - Alternativen in einer Regel von links nach rechts getestet
    - expr : “if” expr “then” statement (“else” statement) ?
- Syntaktische Prädikate
  - “probeweises Parsen”
  - (*predicate*) => regel
  - stat : (list “=”) => list “=” list
    | list
- Semantische Prädikate: Prädikat kann beliebiger Code sein

Semantische Werte

- in der Regel wird beim Parsen AST aufgebaut
- Alternativ: Hilfssymbole können nutzerdefinierte Werte haben
  - expr : atom (PLUS atom | MINUS atom)*;
    atom : INT | LPAREN expr RPAREN ;
  - expr returns [int value = 0]
    { int x; }
    : value=atom
    (PLUS x=atom { value+=x; }
    | MINUS x=atom { value-=x; }
)*;
    atom returns [int value = 0] :
    i:INT { value = Integer.parseInt(i.getText()); }
    | LPAREN value=expr RPAREN
    ;

Erzeugung des Baums der abstrakten Syntax (AST)

- Parseroption buildAST=true
- In Grammatikregeln: Zeichen ^ zeigt an, welchen Typ ein AST-Knoten bekommen soll, Zeichen ! zeigt an, welche Symbole im AST ignoriert werden sollen
  - expr: mexpr (PLUS^ mexpr)* SEMI! ;
  - mexpr: atom (STAR^ atom)* ;
  - atom: INT ;