Grammatica voor Programmeertalen

2021

Werkwijze

2 traps aanpak:

- vind eerst de losse tekens: keywords, getallen, identifiers, etc en gebruik hiervoor de reguliere expressie aanpak
  - het resultaat is een lijst van tokens
- gebruik nu een parser om mbv een grammatica het programma te herkennen

Waarom?

- maakt het parseren eenvoudiger
- parsetrees blijven overzichtelijk
- eerste fase wordt lexicale analyse genoemd (mbv een zgn scanner)

Voorbeeld rekenkundige expressies

```
expressie -> term   (('+' | '-') term)*
term      -> factor (('*' | '/') factor)*
factor    -> num | '(' expressie ')'
```

de invoer:

```
3 + 4 * 5
```

wordt na lexicale analyse

```
[3,'+',4,'*',5]
```

de parser werkt op deze input lijst!

Voorbeelden eenvoudig deel van Python

```
def fac(n):
  if n < 1:
    return 0
  else:
    return n * fac(n-1)
```

```
def sum(n):
  s = 0
  while n > 0:
    s = s + n
    n = n - 1
  return s
```

Simpele functies

- alleen if en while met eenvoudige conditie: var oper expressie
- assignment en return
- alleen getal bewerkingen (geen strings, lijsten, etc)

Wat maakt Python lastig?

```
def f(n,m):
  if n < 1:
    if m > 3:
       return 4
  else:
    return 5
  return 6
```

```
def f(n,m):
  if n < 1:
    if m > 3:
       return 4
    else:
      return 5
  return 6
```

Wat is het verschil?

- let op inspringen en terugspringen
- een inspring levert een indent token op
- een terugspring een dedent
- links: na return 4: 2 dedents en rechts maar 1!
- de scanner moet hiervoor zorgen

Grammatica

```
function    -> def iden '(' args? ')' ':' newline block

args        -> iden (',' iden)*

block       -> indent statement+ dedent

statement   -> complexstat | simplestat newline

simplestat  -> retstat | assign
assign      -> iden '=' expr
retstat     -> return expr

complexstat -> ifStat | whileStat

ifStat      -> if cond ':' newline block elsePart?
elsePart    -> else ':' newline block

whileStat   -> while cond ':' newline block

cond        -> iden relop expr
relop       -> '==' | '<' | '<=' | '>' | '>=' | '!='

expr        -> term (('+' | '-') term)*
term        -> factor (('*' | '/') factor)*
factor      -> num | idenorcall | '(' expr ')'

idenorcall  -> iden ('(' callargs? ')')?
callargs    -> expr (',' expr)*

```