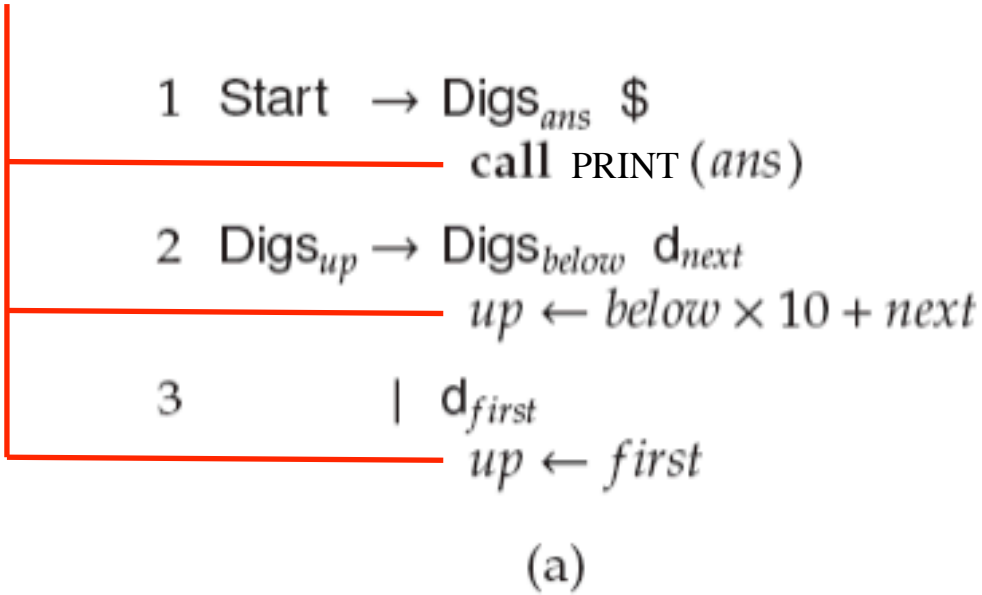
Base-10 waarde van een string digits (bottom-up)

ans, *up*, *below*, *next*, *first*: dit zijn tags – ze dienen om een onderscheid te maken tussen occurrences van hetzelfde symbol

Semantische acties



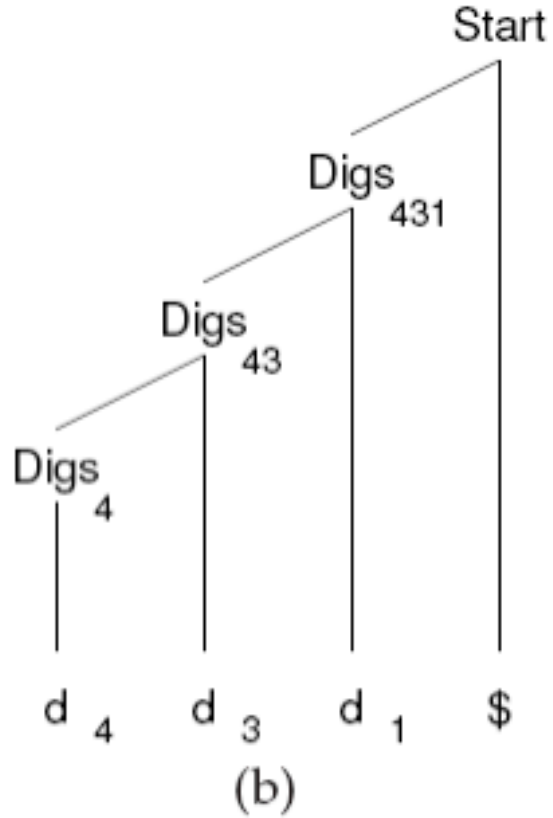


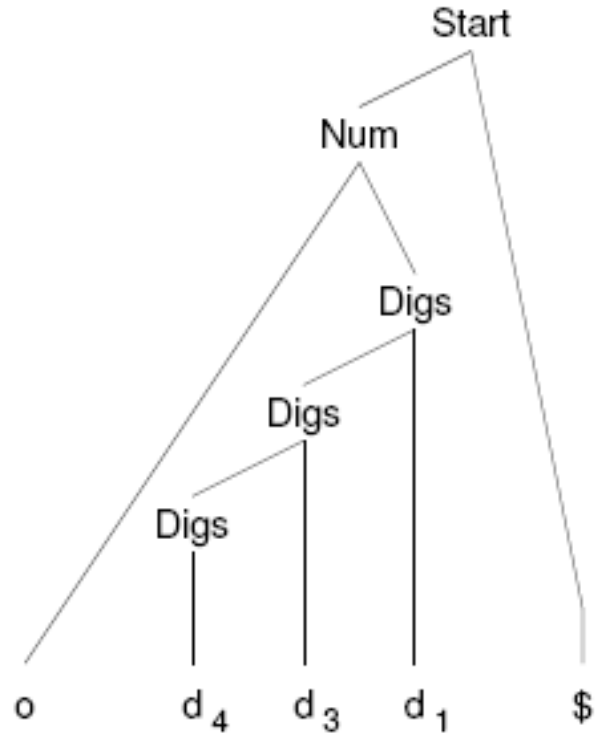
Figure 7.3: (a) Grammar with semantic actions; (b) Parse tree and propagated semantic values for the input 4 3 1 $.

Voeg een indicator toe voor octale notatie: o573 (octal) = 380 (decimal)

Probleem?

1 Start → Num $
2 Num → o Digs
3 | Digs
4 Digs → Digs d
5 | d

(a)



(b)

Figure 7.4: (a) Grammar and (b) parse tree for the input o 4 3 1 $.

Problematisch want – bij bottom-up parsing – komt de informatie dat de string octaal is “te laat”: ze komt pas ter beschikking wanneer een reduce operatie uitgevoerd is voor productie 2

Oplossing: pas de grammatica aan, bv door producties te clonen

```
1  Start          → Num_ans $
                    call PRINT(ans)
2  Num_ans        → o OctDigs_octans
                    ans ← octans
3                 | DecDigs_decans
                    ans ← decans
4  DecDigs_up     → DecDigs_below d_next
                    up ← below × 10 + next
5                 | d_first
                    up ← first
6  OctDigs_up     → OctDigs_below d_next
                    if next ≥ 8                                  (1)
                    then ERROR("Non-octal digit")
                    up ← below × 8 + next
7                 | d_first
                    if first ≥ 8                                 (2)
                    then ERROR("Non-octal digit")
                    up ← first
```

Figure 7.5: Grammar with cloned productions.

Producties en nonterminals zijn nu verschillend vooor het decimale en het octale geval, dus de parser kanze onderscheiden.

Nadeel: grotere grammatica!

Semantische acties forceren door λ-producties

1 Start $\rightarrow$ Num_{ans} \$
  call PRINT(ans)

2 Num_{ans} $\rightarrow$ SignalOctal Digs_{octans}
  $ans \leftarrow octans$

3 | SignalDecimal Digs_{decans}
  $ans \leftarrow decans$

4 SignalOctal $\rightarrow$ o
  $base \leftarrow 8$

5 SignalDecimal $\rightarrow \lambda$
  $base \leftarrow 10$

6 Digs_{up} $\rightarrow$ Digs_{below} d_{next}
  $up \leftarrow below \times base + next$

7 | d_{first}
  $up \leftarrow first$

Figure 7.6: Use of λ-rules to force semantic action.

Verdere uitbreiding: strings met optionele basis

		(decimale) waarde:	
4 3 1 $	431_{10}	431	(default)
x 8 4 3 1 $	431_{8}	281	
x 5 4 3 1 $	431_{5}	116	

```
1  Start          → Num_ans $
                    call PRINT (ans)
2  Num_ans        → x SetBase Digs_baseans
                    ans ← baseans
3                  | SetBaseTen Digs_decans
                    ans ← decans
4  SetBase        → d_val
                    base ← val
5  SetBaseTen     → λ
                    base ← 10
6  Digs_up        → Digs_below d_next
                    if next ≥ base                                  (3)
                    then ERROR( "Digit outside allowable range" )
                    up ← below × base + next
7                  | d_first
                    if first ≥ base                                 (4)
                    then ERROR( "Digit outside allowable range" )
                    up ← first
```

Figure 7.7: Strings with an optionally specified base.

Deze oplossing kan nog verbeterd worden: ze gebruikt de globale variable *base*

Globale variabelen leiden gemakkelijk tot onvoorziene interacties tussen de semantische acties. Het is beter ze te vermijden.

Algemene werkwijze:

1. Schets de gewenste parse tree
2. Verander de grammatica
3. Verifieer dat hij nog geschikt is voor de parsing
4. Verifieer dat hij nog dezelfde taal genereert.

Semantische waarde wordt nu een paar (*base*, *val*)

1 Start $\rightarrow$ Digs_{ans} \$
  call PRINT(*ans.val*)

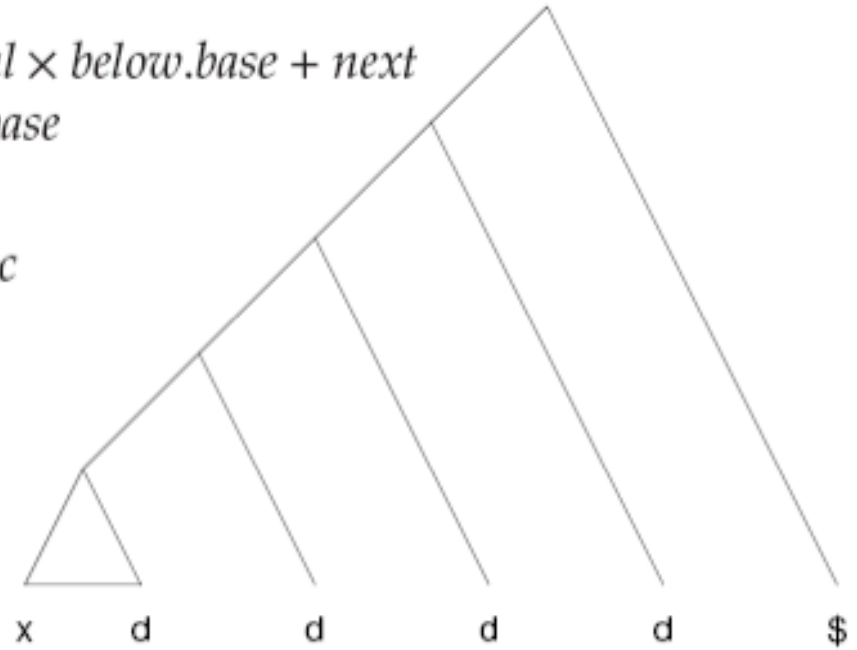
2 Digs_{up} $\rightarrow$ Digs_{below} d_{next}
  $up.val \leftarrow below.val \times below.base + next$
  $up.base \leftarrow below.base$

3 | $\text{SetBase}_{basespec}$
  $up.base \leftarrow basespec$
  $up.val \leftarrow 0$

4 $\text{Setbase}_n \rightarrow \lambda$
  $n \leftarrow 10$

5 | x d_{num}
  $n \leftarrow num$

(a)



(b)

Figure 7.8: (a) Grammar that avoids global variables; (b) Parse tree reorganized to facilitate bottom-up attribute propagation.

Top-down Syntax-Directed Translation

```
1  Start   → Value $
2  Value   → num
3          |  lparen Expr rparen
4  Expr    → plus Value Value
5          |  prod Values
6  Values  → Value Values
7          |  λ
```

Figure 7.9: Grammar for Lisp-like expressions.

Voorbeeld: Recursive descent

Inherited attributen: parameters van de parsing procedures, bv. Values()

Synthesized attributes: berekend bij return van de parsing procedures, bv. Value()

ts: token stream geproduceerd door de scanner

```
procedure START( )
    switch (...)
        case ts.PEEK( ) ∈ {num, lparen}
            ans ← VALUE( )
            call MATCH($)
            call PRINT(ans)
end
function VALUE( ) returns int
    switch (...)
        case ts.PEEK( ) ∈ {num}
            call MATCH(num)
            ans ← num.VALUEOF( )
            return (ans)
        case ts.PEEK( ) ∈ {lparen}
            call MATCH(lparen)
            ans ← EXPR( )
            call MATCH(rparen)
            return (ans)
end
```

```
function EXPR( ) returns int
    switch (...)
        case ts.PEEK( ) ∈ {plus}
            call MATCH(plus)
            op1 ← VALUE( )
            op2 ← VALUE( )
            return (op1 + op2)
        case ts.PEEK( ) ∈ {prod}
            call MATCH(prod)
            ans ← VALUES(1)
            return (ans)
end
function VALUES(thusfar) returns int
    case ts.PEEK( ) ∈ {num, lparen}
        next ← VALUE( )
        ans ← VALUES(thusfar × next)
        return (ans)
    case ts.PEEK( ) ∈ {rparen}
        return (thusfar)
end
```

thusfar: neemt de to dusver berekende attributen in rekening

AST design en constructie

```
 1  Start  → Stmt $
 2  Stmt   → id assign E
 3           | if lparen E rparen Stmt else Stmt fi
 4           | if lparen E rparen Stmt fi
 5           | while lparen E rparen do Stmt od
 6           | begin Stmts end
 7  Stmts  → Stmts semi Stmt
 8           | Stmt
 9  E      → E plus T
10           | T
11  T      → id
12           | num
```

Figure 7.14: Grammar for a simple language.

Semantische acties voor de productie van de AST

1. Start → $Stmt_{ast}$ $
   return(ast)
2. $Stmt_{result}$ → id_{var} assign E_{expr}
   result ← MakeFamily(assign, var, expr)
3. | if lparen E_p rparen $Stmt_s$ fi
   result ← MakeFamily(if, p, s, MakeNode())
4. | if lparen E_p rparen $Stmt_{s1}$ else $Stmt_{s2}$ fi
   result ← MakeFamily(if, p, s1, s2)
5. | while lparen E_p rparen do $Stmt_s$ od
   result ← MakeFamily(while, p, s)
6. | begin $Stmts_{list}$ end
   result ← MakeFamily(block, list)
7. $Stmts_{result}$ → $Stmts_{sofar}$ semi $Stmt_{next}$
   result ← sofar.MakeSiblings(next)
8. | $Stmt_{first}$
   result ← first
9. E_{result} → E_{e1} plus T_{e2}
   result ← MakeFamily(plus, e1, e2)
10. | T_e
   result ← e
11. T_{result} → id_{id}
   result ← MakeNode(var)
12. | num_{val}
   result ← MakeNode(val)

In het algemeen is een attributengrammatica noch bottom-up, noch top-down. Dan kan de evaluatie in 2 stappen:

(1) Maak de graph van dependencies:

- knopen = occurrences van attributen
- pijl van a naar b als a nodig is voor de semantische actie die b invult

(2) Indien deze graph acyclisch is, en de beginwaarden zijn beschikbaar, dan kunnen alle waarden berekend worden (eventueel in meerdere doortochten)

Typechecking

E ⟶ var = E
E ⟶ E aop T
E ⟶ T
T ⟶ T mop F
T ⟶ F
F ⟶ (E)
F ⟶ const
F ⟶ var
F ⟶ var ( )

$$
\begin{array}{lcl}
E \longrightarrow \mathsf{var}\ '='\ E & & \\
E[1].env & = & E[0].env \\
E[0].typ & = & E[0].env\ \mathsf{var}.id \\
E[0].ok & = & \quad \textbf{let}\ x = \mathsf{var}.id \\
 & & \textbf{in let}\ \tau = E[0].env\ x \\
 & & \textbf{in} \quad (\tau \neq \mathsf{error}) \wedge (E[1].type \sqsubseteq \tau)
\end{array}
$$

let $x = e$ **in** R: substitueer e voor x in R

$$
\begin{array}{lcl}
E \longrightarrow E\ \mathsf{aop}\ T & & \\
E[1].env & = & E[0].env \\
T.env & = & E[0].env \\
E[0].typ & = & E[1].typ \sqcup T.typ \\
E[0].ok & = & (E[1].typ \sqsubseteq \mathsf{float}) \\
 & & \wedge\ (T.typ \sqsubseteq \mathsf{float})
\end{array}
$$

$$
\begin{array}{lcl}
E \longrightarrow T & & \\
T.env & = & E.env \\
E.typ & = & T.typ \\
E.ok & = & T.ok
\end{array}
$$

$\sqcup$: gemeensch. supertype

$\sqsubseteq$: subtype

$T \longrightarrow T$ mop F

$$
\begin{array}{lcl}
T[1].env & = & T[0].env \\
F.env & = & T[0].env \\
T[0].typ & = & T[1].typ \sqcup F.typ \\
T[0].ok & = & (T[1].typ \sqsubseteq \mathsf{float}) \\
 & & \wedge\ (F.typ \sqsubseteq \mathsf{float})
\end{array}
$$

$T \longrightarrow F$

$$
\begin{array}{lcl}
F.env & = & T.env \\
T.typ & = & F.typ \\
T.ok & = & F.ok
\end{array}
$$

$F \longrightarrow (E)$

$$
\begin{array}{lcl}
E.env & = & F.env \\
F.typ & = & E.typ \\
F.ok & = & E.ok
\end{array}
$$

$F \longrightarrow \mathsf{const}$

$$
\begin{array}{lcl}
F.typ & = & \mathsf{const}.typ \\
F.ok & = & \mathsf{true}
\end{array}
$$

$F \longrightarrow \mathsf{var}$

$$
\begin{array}{lcl}
F.typ & = & F.env\ \mathsf{var}.id \\
F.ok & = & (F.env\ \mathsf{var}.id \neq \mathsf{error})
\end{array}
$$

$F \longrightarrow \mathsf{var}\ ()$

$$
\begin{array}{lcl}
F.typ & = & (F.env\ \mathsf{var}.id)\ () \\
F.ok & = & \textbf{match}\ F.env\ \mathsf{var}.id \\
 & & \textbf{with}\ \tau\ () \rightarrow \mathsf{true} \\
 & & \quad \mid\ \_ \rightarrow \mathsf{false}
\end{array}
$$

Declaration Processing (blocks en parameterloze procedures)

$\langle decl \rangle \longrightarrow \langle type \rangle$ var;

$\langle decl \rangle \longrightarrow$ void var () { $\langle block \rangle$ }

$\langle stat \rangle \longrightarrow$ var $= E$;

$\langle stat \rangle \longrightarrow$ { $\langle block \rangle$ }

$\langle block \rangle \longrightarrow \langle decl \rangle \ \langle block \rangle$

$\langle block \rangle \longrightarrow \langle stat \rangle \ \langle block \rangle$

$\langle block \rangle \longrightarrow \epsilon$

$\langle decl \rangle \longrightarrow \langle type \rangle$ var;

$\langle decl \rangle .new = (\text{var}.id, \langle type \rangle .typ)$
$\langle decl \rangle .ok \quad = \text{true}$

$\langle decl \rangle \longrightarrow \text{void var } () \{ \langle block \rangle \}$

$\langle block \rangle .same = \emptyset$
$\langle block \rangle .env \quad = \langle decl \rangle .env \oplus$
$\quad \{\text{var}.id \mapsto \text{void } ()\}$
$\langle decl \rangle .new \quad = (\text{var}.id, \text{void } ())$
$\langle decl \rangle .ok \quad = \langle block \rangle .ok$

$\langle block \rangle \longrightarrow \langle decl \rangle \ \langle block \rangle$

$\langle decl \rangle .env = \langle block \rangle [0].env$
$\langle block \rangle [1].same$
$=$ **let** $(x, \_) = \langle decl \rangle .new$
in $\langle block \rangle [0].same \cup \{x\}$
$\langle block \rangle [1].env$
$=$ **let** $(x, \tau) = \langle decl \rangle .new$
in $\langle block \rangle [0].env$
$\oplus \{x \mapsto \tau\}$
$\langle block \rangle [0].ok$
$=$ **let** $(x, \_) = \langle decl \rangle .new$
in **if** $\neg(x \in \langle block \rangle [0].same)$
then $\langle decl \rangle .ok \wedge \langle block \rangle [1].ok$
else false

$$\langle stat \rangle \longrightarrow \mathsf{var} = E;$$

$$E.env = \langle stat \rangle .env$$
$$\langle stat \rangle .ok = E.ok \wedge \langle stat \rangle .env(\mathsf{var}.id) = E.typ$$

$$\langle stat \rangle \longrightarrow \{ \langle block \rangle \}$$

$$\langle block \rangle .env = \langle stat \rangle .env$$
$$\langle block \rangle .same = \emptyset$$
$$\langle stat \rangle .ok = \langle block \rangle .ok$$

$$\langle block \rangle \longrightarrow \langle stat \rangle \ \langle block \rangle$$

$$\langle stat \rangle .env = \langle block \rangle [0].env$$
$$\langle block \rangle [1].env = \langle block \rangle [0].env$$
$$\langle block \rangle [1].same = \langle block \rangle [0].same$$
$$\langle block \rangle [0].ok = (\langle stat \rangle .ok \wedge \langle block \rangle [1].ok)$$

$$\langle block \rangle \longrightarrow \epsilon$$

$$\langle block \rangle .ok = \mathsf{true}$$

Codegeneratie (evaluatie van boolean expressies)

Generatie van minimale code: zonder boolean operatoren, enkel met “load” instructies en testen

Codegeneratie (evaluatie van boolean expressies)

Generatie van minimale code: zonder boolean operatoren, enkel met “load” instructies en testen

$$\langle if\_stat \rangle \longrightarrow \text{if } (E) \ \langle stat \rangle \ \text{else} \ \langle stat \rangle$$

$$E \rightarrow T$$
$$E \rightarrow E \text{ or } T$$

$$T \rightarrow F$$
$$T \rightarrow T \text{ and } F$$

$$F \rightarrow (E)$$
$$F \rightarrow \text{not } F$$

$$F \rightarrow \text{var}$$

$\langle if\_stat \rangle \longrightarrow$ if (E) $\langle stat \rangle$ else $\langle stat \rangle$

$E.tsucc = \text{new}()$

$E.fsucc = \text{new}()$

$\langle if\_stat \rangle .code =$ **let** $t = E.tsucc$
in let $e = E.fsucc$
in let $f = \text{new}()$
in $E.code \,\hat{}\, \text{gencjump}(\neg E.jcond, e) \,\hat{}$
$t : \hat{} \langle stat \rangle [1].code \hat{} \text{jump } f \,\hat{}$
$e : \hat{} \langle stat \rangle [2].code \hat{}$
$f :$

$E \rightarrow T$

$E \rightarrow E$ or T

$E[1].tsucc = E[0].tsucc$ $\qquad$ $T.tsucc = E[0].tsucc$

$E[1].fsucc = \text{new}()$ $\qquad$ $T.fsucc = E[0].fsucc$

$E[0].jcond = T.jcond$

$E[0].code =$ **let** $t = E[1].fsucc$
in $E[1].code \hat{} \text{gencjump}(E[1].jcond, E[0].tsucc) \,\hat{}$
$t : \hat{} T.code$

$T \rightarrow F$

$T \rightarrow T$ and F

$T[1].tsucc = \mathsf{new}()$ $\qquad$ $F.tsucc = T[0].tsucc$

$T[1].fsucc = T[0].fsucc$ $\qquad$ $F.fsucc = T[0].fsucc$

$T[0].jcond = F.jcond$

$T[0].code = \textbf{let } f = T[1].tsucc$

$\textbf{in } T[1].code \hat{}\, \mathsf{gencjump}\,(\neg T[1].jcond, T[0].fsucc) \hat{}$

$f : \hat{} F.code$

$F \rightarrow (E)$

$F \rightarrow$ not F

$F[1].tsucc = F[0].fsucc$

$F[1].fsucc = F[0].tsucc$

$F[0].code = F[1].code$

$F[0].jcond = \neg F[1].jcond$

$F \rightarrow$ var

$F.jcond = \mathsf{true}$

$F.code = \mathsf{load\ var}.id$