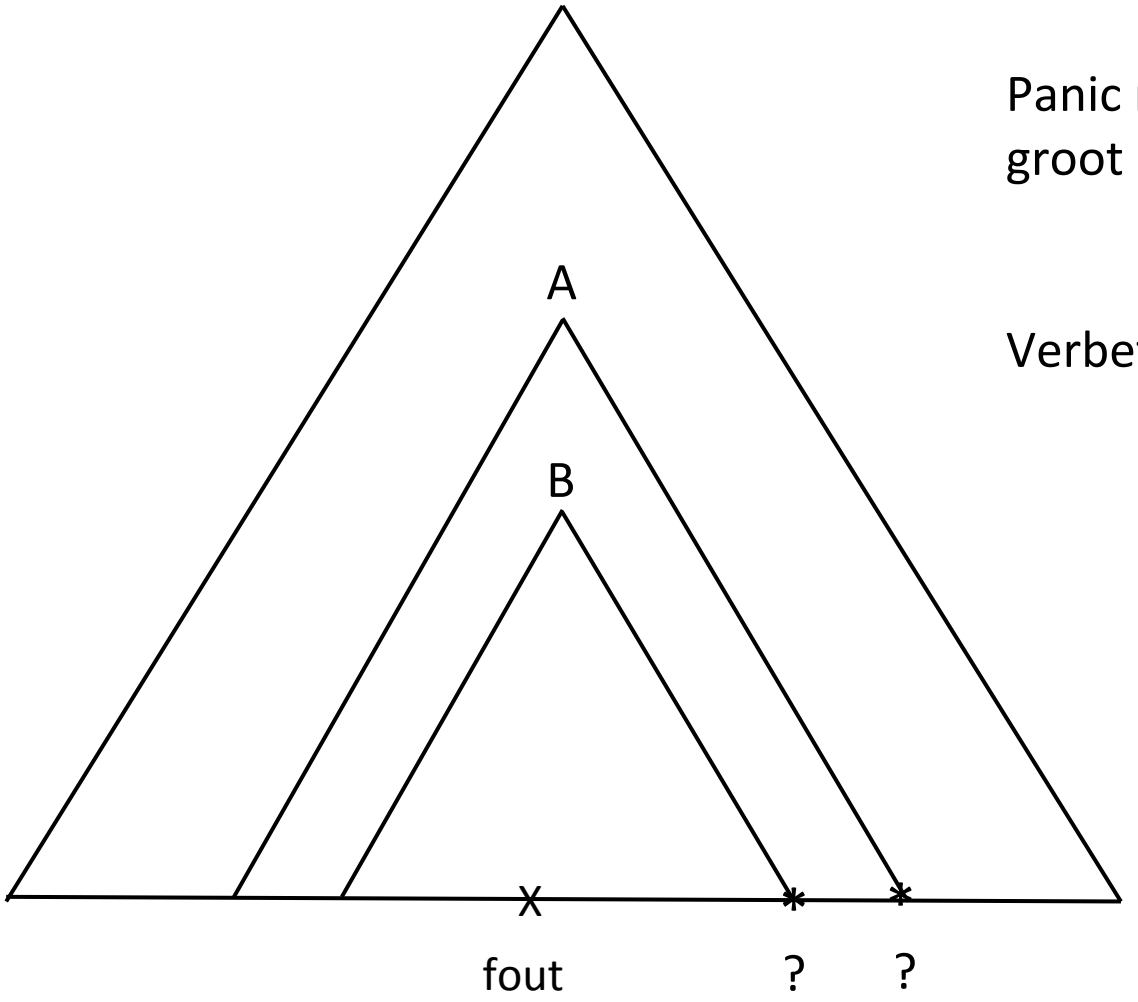
Error recovery in top-down parsers

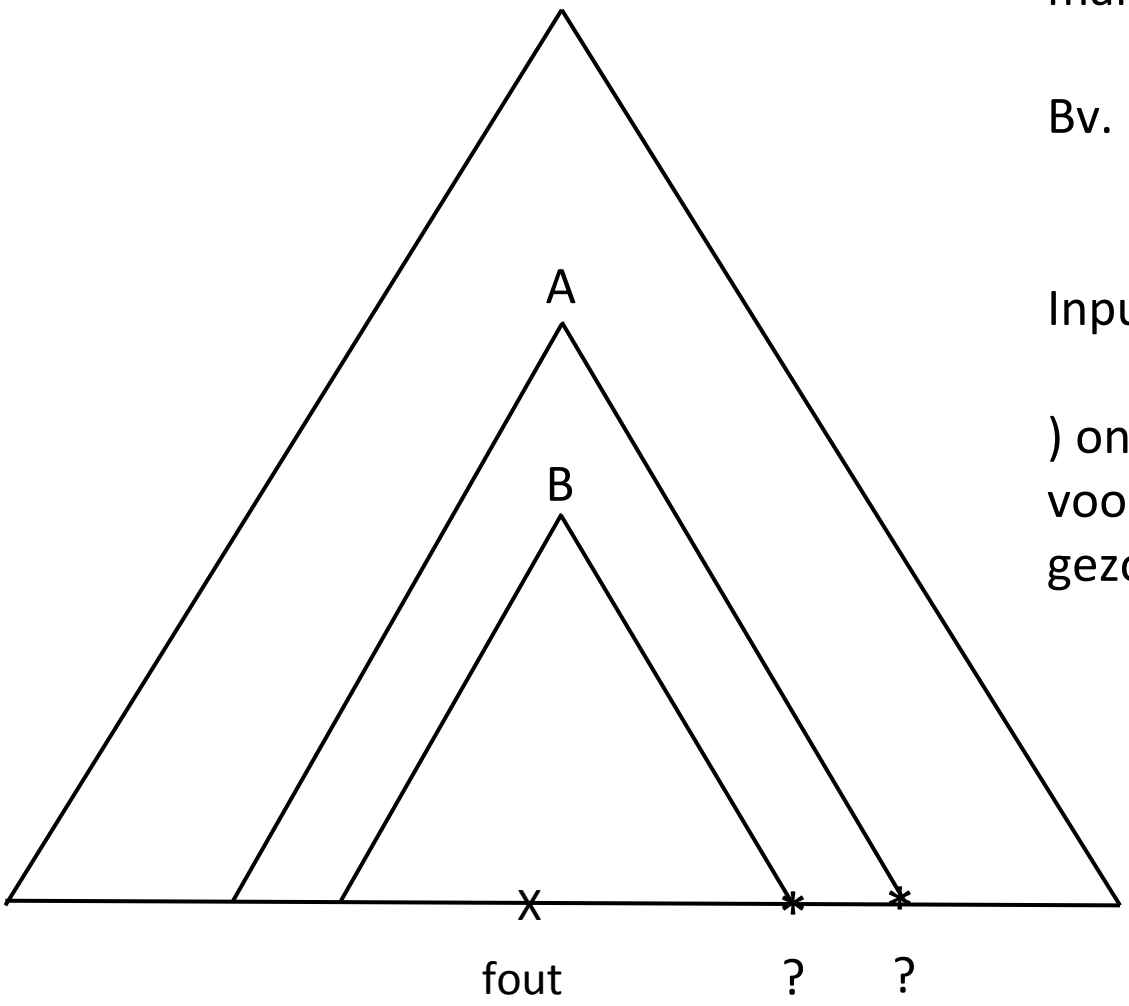
LL(1) parsers hebben de extendible prefix property: elke prefix die geaccepteerd wordt kan aangevuld worden tot een geldig woord. Dit suggereert om die prefix dan ook niet te corrigeren. Wat kan men dan wel doen?

Panic mode: skip tot aan een eindsymbool van de huidige nonterminal (bv. “)”of “;”of ”fi”). Als dat niet te vinden is, pop nonterminals van de stack en zoek verder.



Panic mode: negeert vaak (te) groot deel van de invoer

Verbetering: anker-symbolen



anker-symbolen

markeren eind van subexpressies

Bv. A = <conditional-stat>,
B = <expr>

Input: if (Id < while

) ontbreekt, maar while is anker voor A. Er wordt niet verder gezocht naar)

Error recovery in bottom-up parsers

Een syntaxfout geeft aanleiding tot een error-configuratie: een paar

$$(\phi\ q,\ a_i\ a_{i+1} \ldots a_n\ )$$

waar $\phi\ q$ de stackinhoud is, q de actuele toestand en $a_i\ a_{i+1} \ldots a_n$ de nog te lezen input . We proberen door één symbool te veranderen in de input een configuratie te bereiken van waaruit tenminste het eerstvolgende symbool verwerkt kan worden. Meer precies zijn er 3 mogelijkheden:

Error recovery in bottom-up parsers

Delete: vind een stackinhoud ψ p zo dat

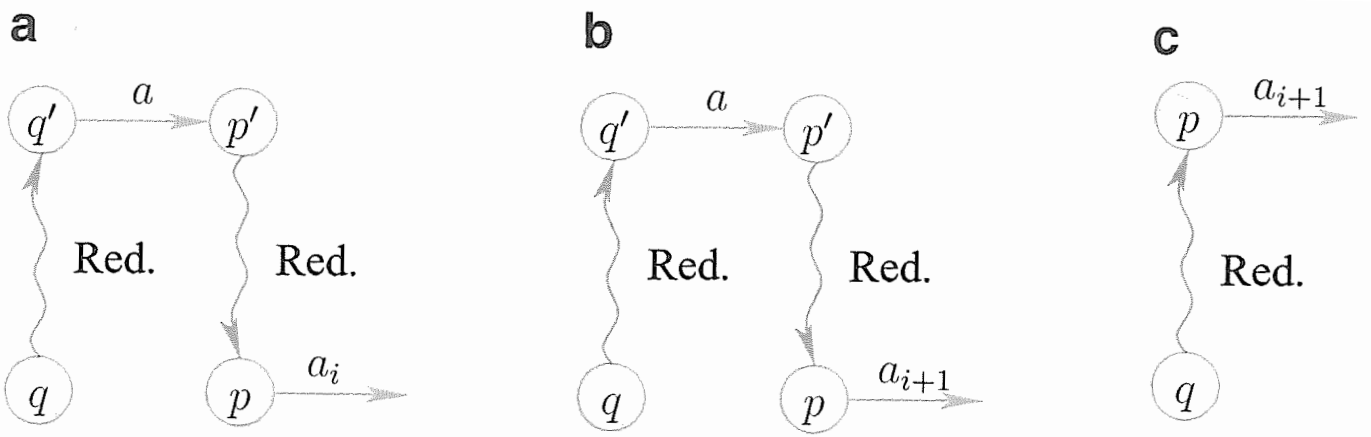
$(\phi\ q, a_{i+1} \ldots a_n) \vdash^* (\psi\ p, a_{i+1} \ldots a_n)$ en *action*$[p, a_{i+1}]$ = *shift*

Replace: vind een stackinhoud ψ p en een symbool a zo dat

$(\phi\ q, a\ a_{i+1} \ldots a_n) \vdash^* (\psi\ p, a_{i+1} \ldots a_n)$ en *action*$[p, a_{i+1}]$ = *shift*

Insert: vind een stackinhoud ψ p en een symbool a zo dat

$(\phi\ q, a\ a_i \ldots a_n) \vdash^* (\psi\ p, a_i \ldots a_n)$ en *action*$[p, a_i]$ = *shift*



Closing the gap for error correction, **a** by insertion, **b** by replacement, or **c** by deletion of a symbol

Het zoekproces kan veel versneld worden door precomputatie:

Volgende sets kunnen berekend worden at compile time:

Succ(q,a) is de set van reductie-successors van p onder a: alle states die vanuit q bereikt kunnen worden door enkel reducties, rekening houdend met de lookahead a

Sh(q,a) is de set van toestanden die vanuit *Succ*(q,a) bereikt kunnen worden door a te shiften

Bridge(q,a_i) is de set van symbolen a zo dat er een state q' is in *Sh*(q,a) waarvoor *Sh*(q',a_i) niet leeg is.

$$
\begin{array}{lcl}
S' & \rightarrow & S \\
S & \rightarrow & L = R \mid R \\
L & \rightarrow & *R \mid \mathsf{Id} \\
R & \rightarrow & L
\end{array}
$$

States van de karakteristieke automaat:

$$
\begin{aligned}
S_0 = \{ & [S' \rightarrow .S], \\
& [S \rightarrow .L = R], \\
& [S \rightarrow .R], \\
& [L \rightarrow . * R], \\
& [L \rightarrow .\mathsf{Id}], \\
& [R \rightarrow .L] \}
\end{aligned}
$$

$$
S_1 = \{ [S' \rightarrow S.] \}
$$

$$
\begin{aligned}
S_2 = \{ & [S \rightarrow L. = R], \\
& [R \rightarrow L.] \}
\end{aligned}
$$

$$
S_3 = \{ [S \rightarrow R.] \}
$$

$$
\begin{aligned}
S_4 = \{ & [L \rightarrow *.R], \\
& [R \rightarrow .L], \\
& [L \rightarrow . * R], \\
& [L \rightarrow .\mathsf{Id}] \}
\end{aligned}
$$

$$
S_5 = \{ [L \rightarrow \mathsf{Id}.] \}
$$

$$
\begin{aligned}
S_6 = \{ & [S \rightarrow L = .R], \\
& [R \rightarrow .L], \\
& [L \rightarrow . * R], \\
& [L \rightarrow .\mathsf{Id}] \ \}
\end{aligned}
$$

$$
S_7 = \{ [L \rightarrow *R.] \}
$$

$$
S_8 = \{ [R \rightarrow L.] \}
$$

$$
S_9 = \{ [S \rightarrow L = R.] \}
$$

Deze grammatica is niet geschikt voor LR(0) parsing:

er is een shift/reduce conflict in S_2

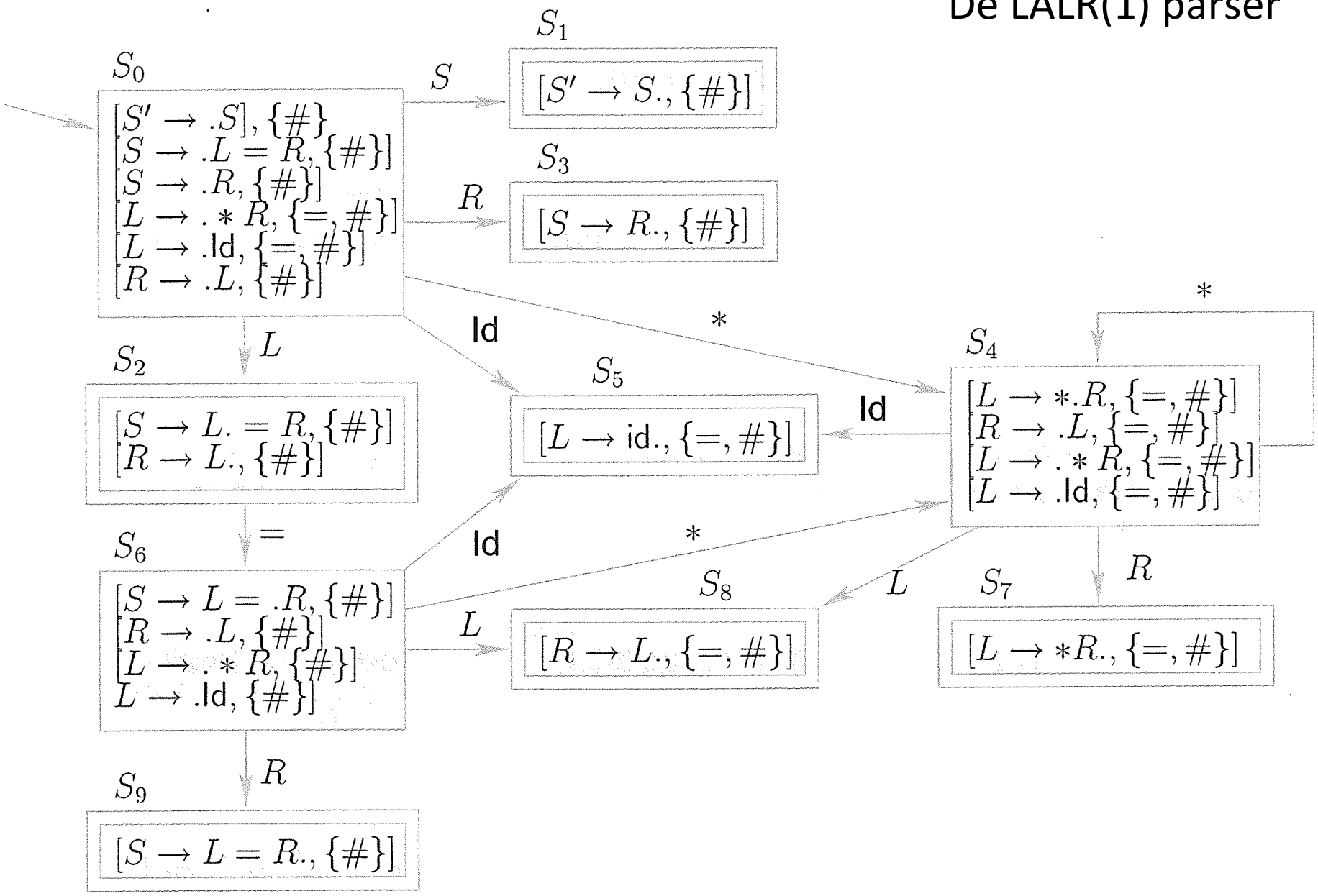
en ook niet voor SLR(1) parsing:

Follow(R) bevat het symbool = en volstaat dus niet om het conflict op te lossen

maar wel voor LALR(1) parsing:

reductie is in S_2 alleen gewenst als de lookahead het symbool # is

De LALR(1) parser



$Succ(q, a)$:

q	$=$	$*$	Id
S_0	S_0	S_0	S_0
S_1	S_1	S_1	S_1
S_2	S_2	S_2	S_2
S_3	S_3	S_3	S_3
S_4	S_4	S_4	S_4
S_5	S_2, S_5	S_5	S_5
S_6	S_6	S_6	S_6
S_7	S_2, S_7	S_7	S_7
S_8	S_2, S_8	S_8	S_8
S_9	S_9	S_9	S_9

$Sh(q, a)$:

q	$=$	$*$	Id
S_0		S_4	S_5
S_1			
S_2	S_6		
S_3			
S_4		S_4	S_5
S_5	S_6		
S_6		S_4	S_5
S_7	S_6		
S_8	S_6		
S_9			

$Bridge(q, a)$:

q	$=$	$*$	Id
S_0	Id	$*$	$*$
S_1			
S_2		$=$	$=$
S_3			
S_4	Id	$*$	$*$
S_5		$=$	$=$
S_6	Id	$*$	$*$
S_7		$=$	$=$
S_8			
S_9			

Gebruik van deze informatie voor correcties:

Input	Error configuration	Bridge	Correction
$* = \mathsf{Id}\ \#$	$(S_0 S_4, = \mathsf{Id}\ \#)$	$bridge(S_4, =) = \{\mathsf{Id}\}$	Insertion of Id
$\mathsf{Id} == \mathsf{Id}\ \#$	$(S_0 S_2 S_6, = \mathsf{Id}\ \#)$	$Bridge(S_6, \mathsf{Id}) = \{*\}$	Replacement of = by *

Input	Error configuration	Bridge	Correction
$\mathsf{Id}\ \mathsf{Id} = \mathsf{Id}\ \#$	$(S_0 S_5, \mathsf{Id} = \mathsf{Id}\ \#)$	$Sh(S_5, =) \neq \emptyset$	Deletion of Id

Elke correctie leidt tot het lezen (shift) van tenminste één symbool, er komt dus geen oneindige lawine van correcties.

Keuze tussen de verschillende mogelijke correcties door ontwerper vvan compiler

Semantische analyse

Doel:

- Declaration processing: relateer applied occurrences met de correcte defining occurrences, opbouw environment
- Typechecking (evt type inference)

Technieken:

- Symbooltabel
- Syntax –directed translation (SDT), attribute grammars

Abstract syntax tree (AST)

Is een vereenvoudigde versie van de parse tree (= concrete syntax tree), met weglating van knopen en takken die in het verdere compilatieproces niet meer nodig zijn

De AST wordt gebruikt voor zowel semantische analyse (e.g. type checking) als voor code generatie. Om dat laatste mogelijk te maken worden de knopen gedecoreerd met bijkomende informatie.

Deze informatie bestaat uit semantische waarden geassocieerd met knopen van de AST, en de manipulatie ervan gebeurt door semantische acties .

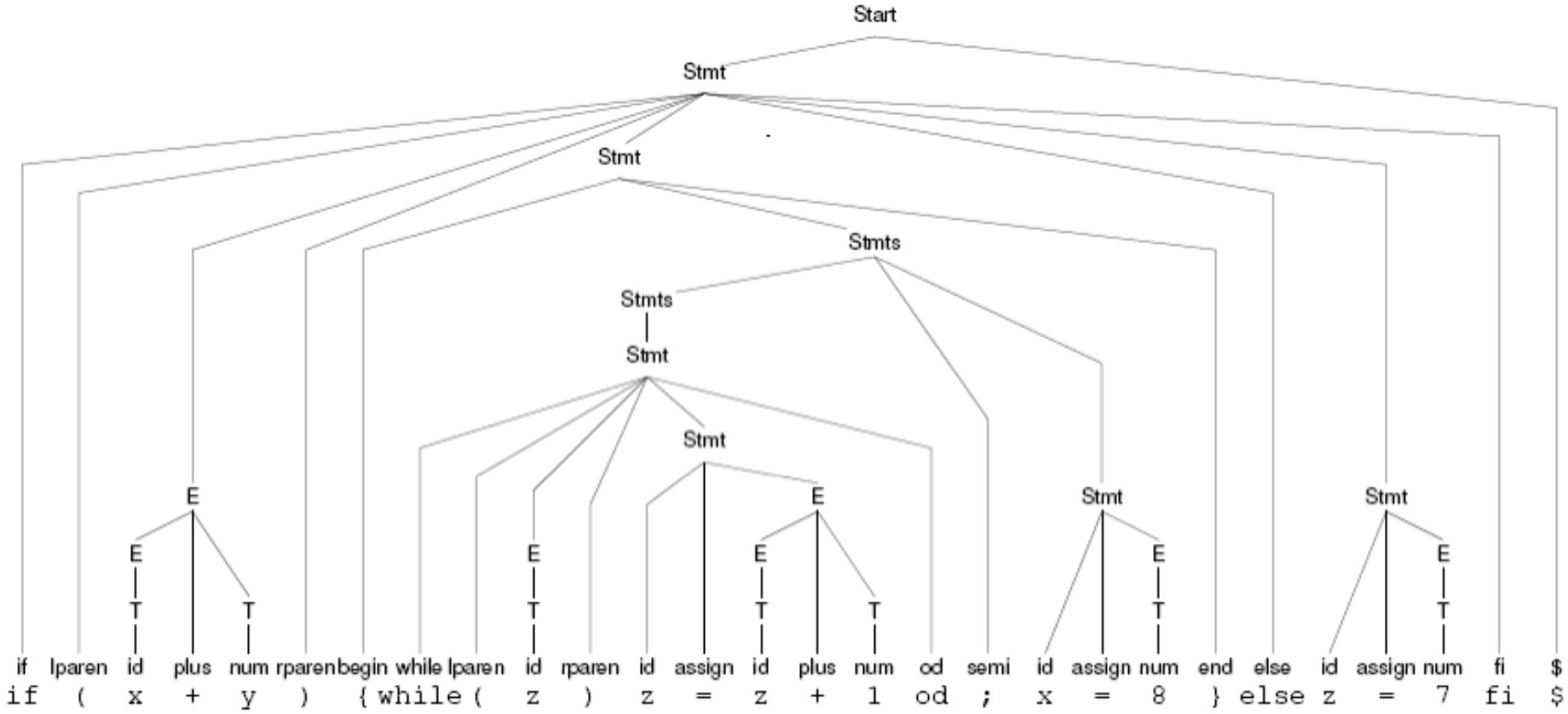


Figure 7.18: Concrete syntax tree.

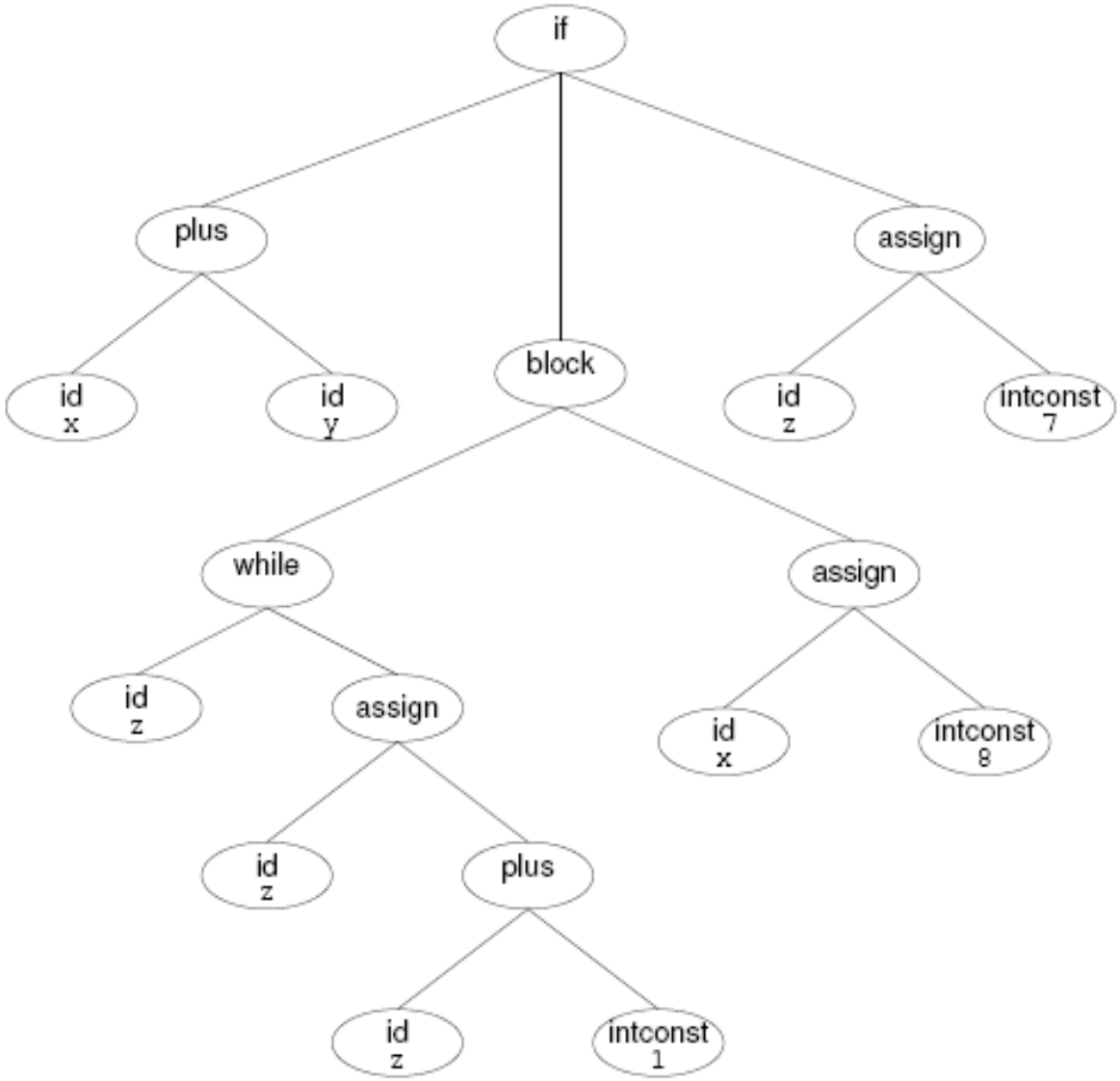


Figure 7.19: AST for the parse tree in Figure 7.18.

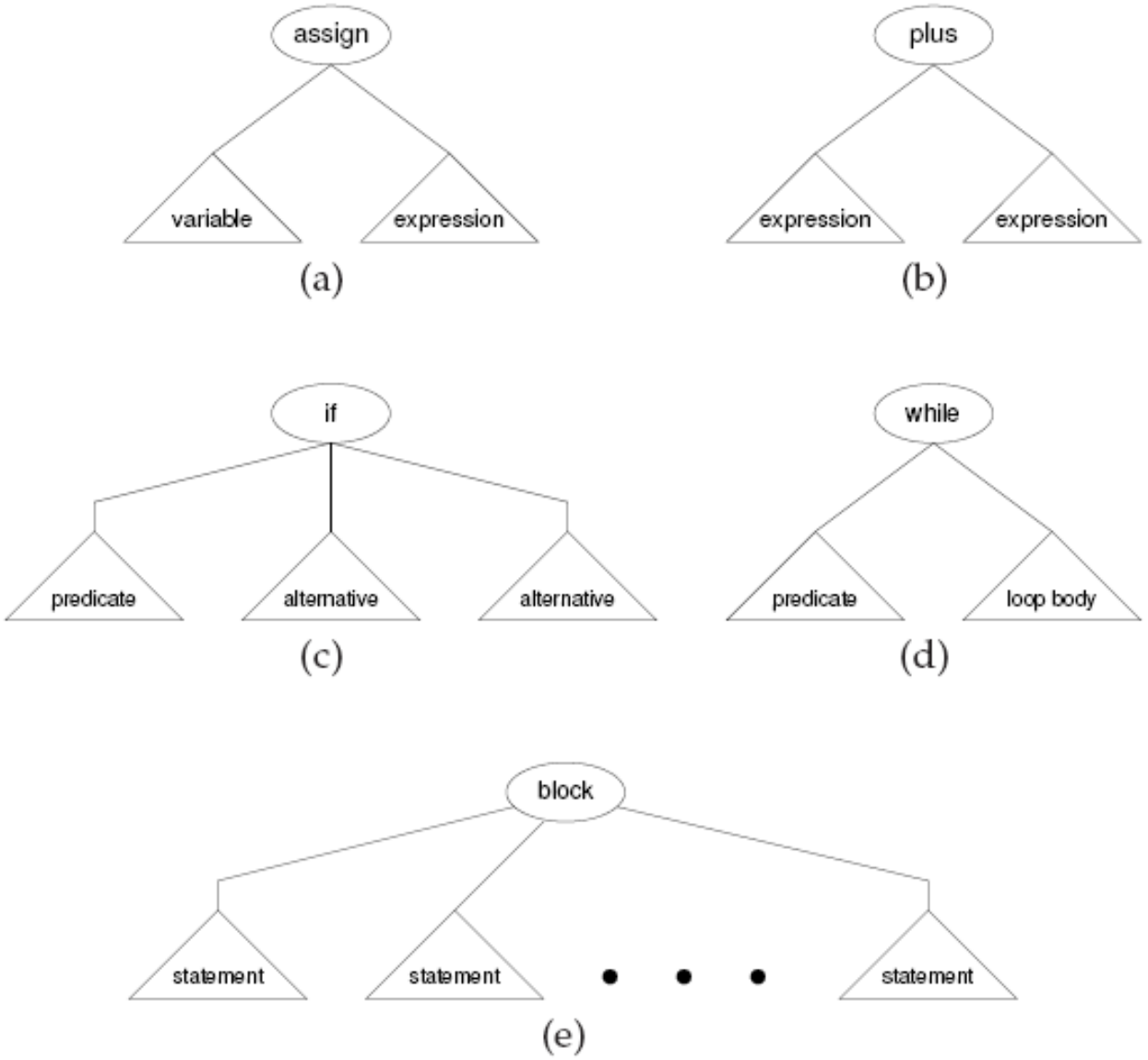


Figure 7.15: AST structures: A specific node is designated by an ellipse. Tree structure of arbitrary complexity is designated by a triangle.

Concrete parse tree

AST

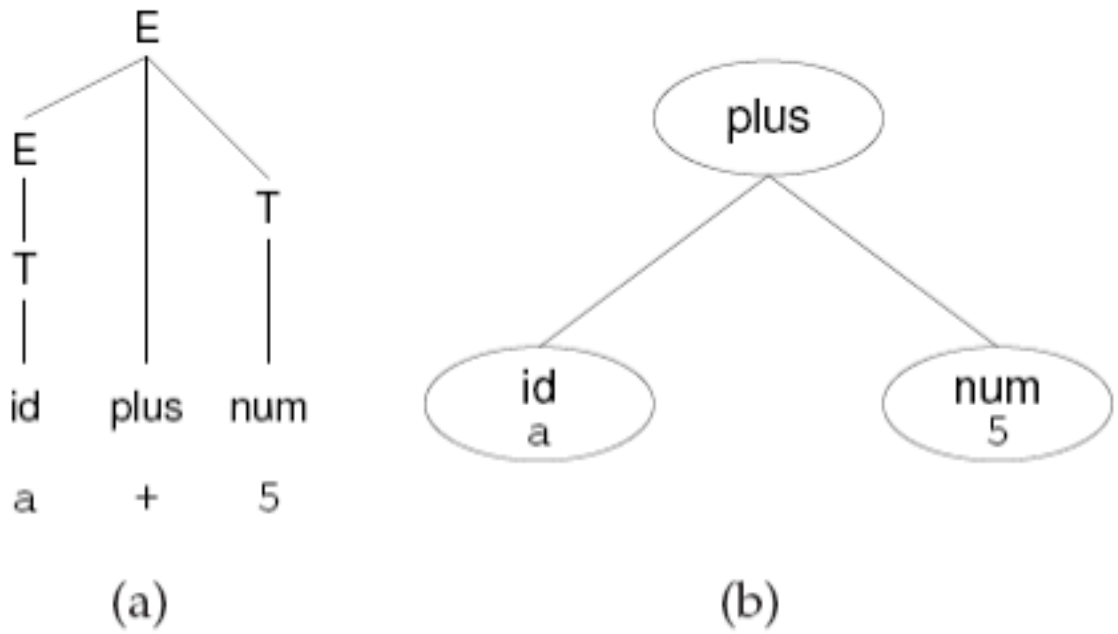


Figure 7.16: (a) Derivation of a + 5 from E;
(b) Abstract representation of a + 5.

Soms is het nuttig knopen toe te voegen:

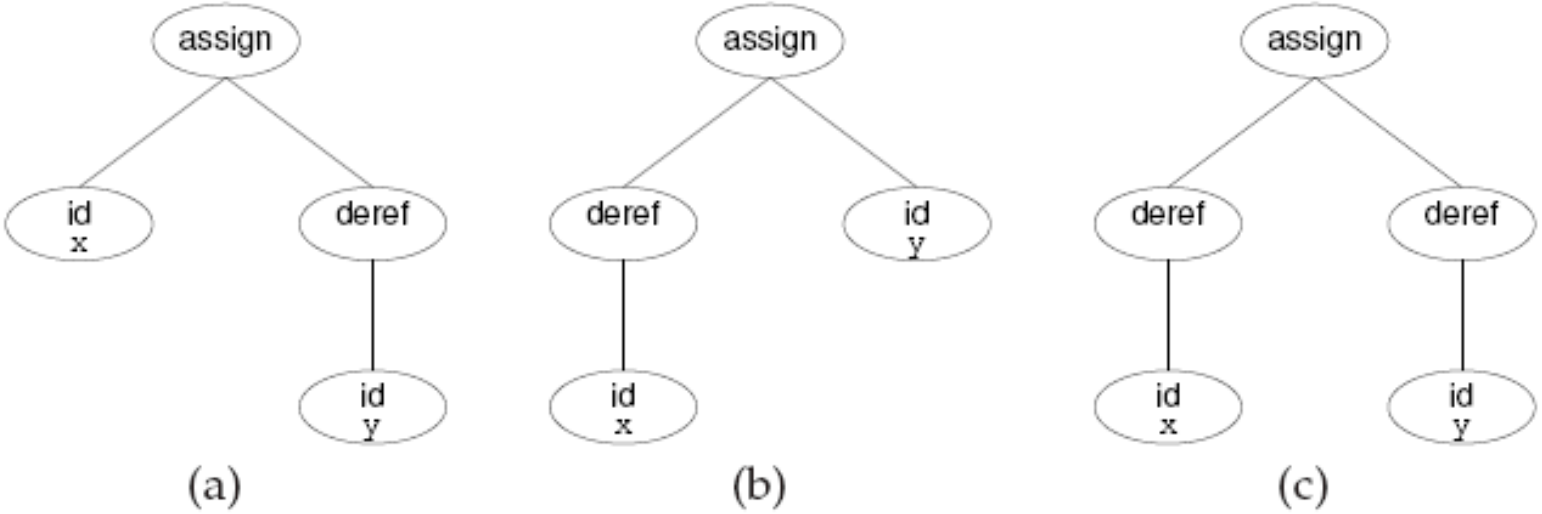


Figure 7.22: ASTs illustrating left and right values for the assignments:
(a) x = y
(b) x = &y
(c) ⋆x = y

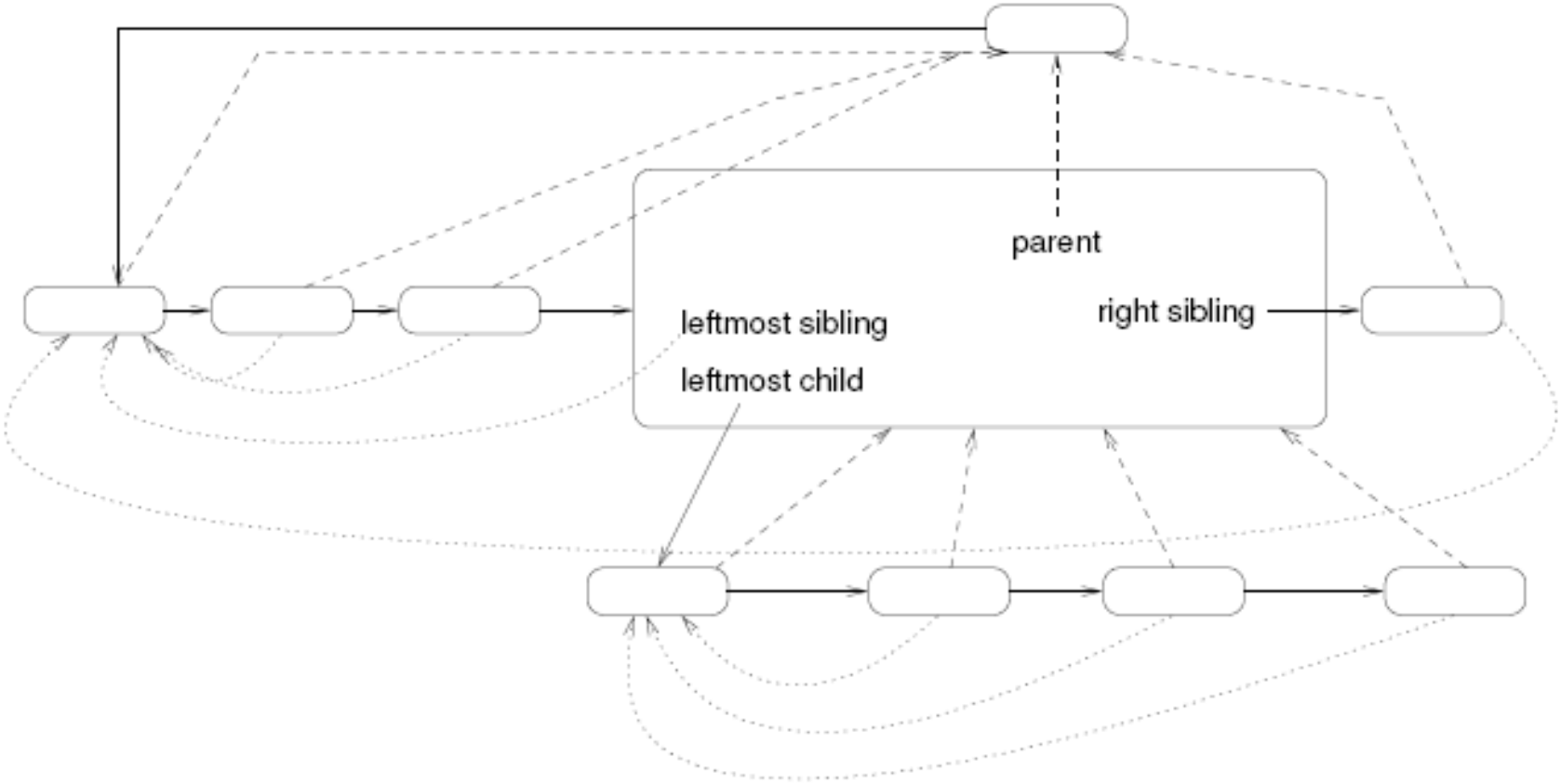
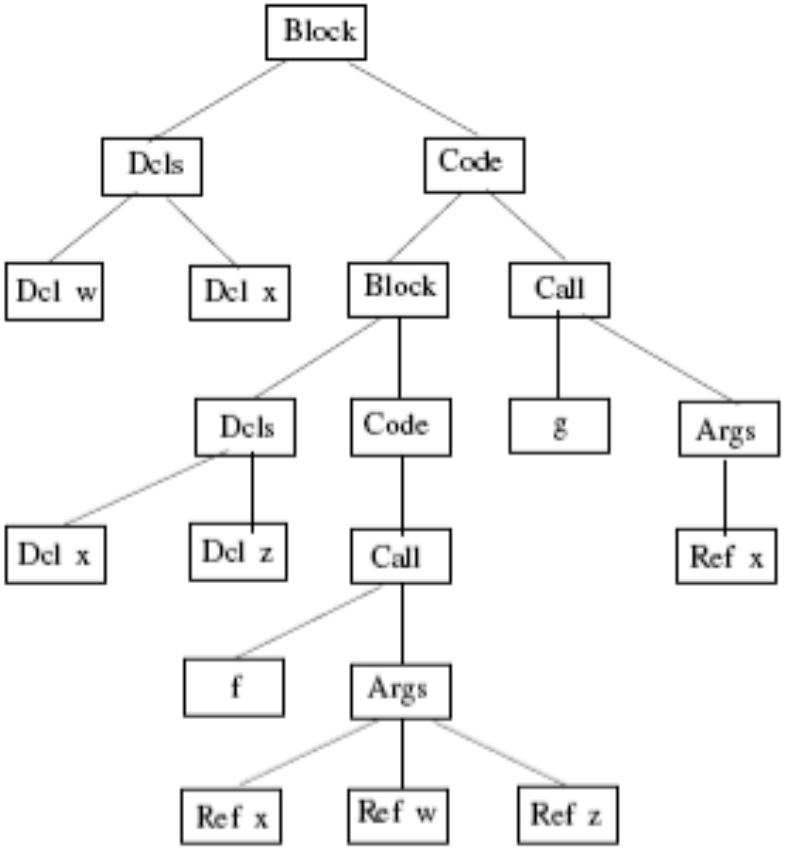


Figure 7.12: Internal format of an AST node. A dashed line connects a node with its parent; a dotted line connects a node with its leftmost sibling. Each node also has a solid connection to its leftmost child and right sibling.

Symbol Tables and Declaration Processing

```
import f(float, float, float)
import g(int)
{
  int w,x
  {
    float x,z
    f(x,w,z)
  }
  g(x)
}
```

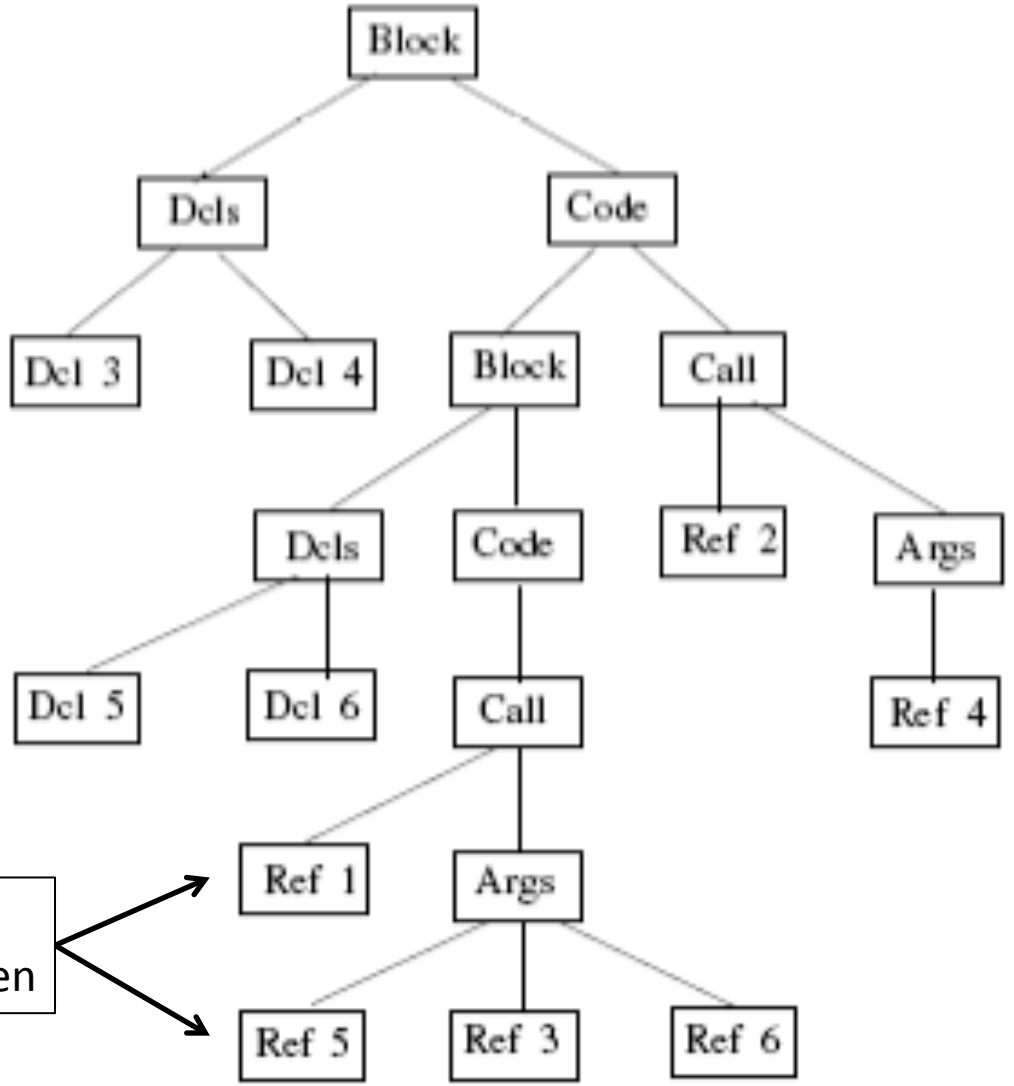
(a)



(b)

Symbooltabel: voor elke declaratie, de relevante info

Symbol Number	Symbol Name	Attributes
1	f	void func(float,float,float)
2	g	void func(int)
3	w	int
4	x	int
5	x	float
6	z	float



Nodes referen naar specifieke declaraties, niet enkel naar namen

(d)

Hiervoor is het nodig bij elke applied occurrence een lijst te hebben met, voor de zichtbare namen, de op dat punt geldige defining occurrences.

Enige voorzichtigheid is nodig omdat namen geherdeclareerd kunnen worden in blocks e.d. (scopes)

Om de symbooltabel te kunnen gebruiken heeft men bv. volgende operaties

Openscope()

Closescope()

Entersymbol(name, type)

Retrievesymbol(name)

Symbooltabel: mogelijke implementaties

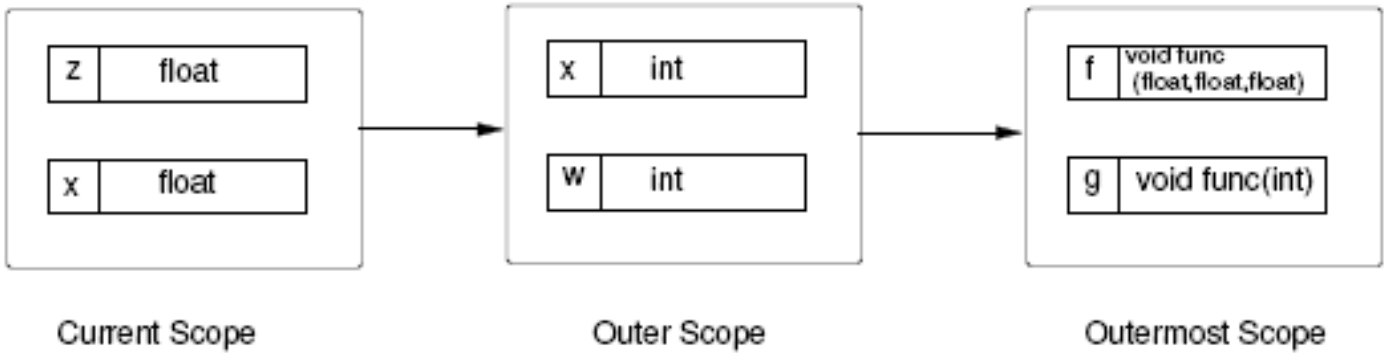


Figure 8.3: A stack of symbol tables, one per scope

Symbooltabel: mogelijke implementaties

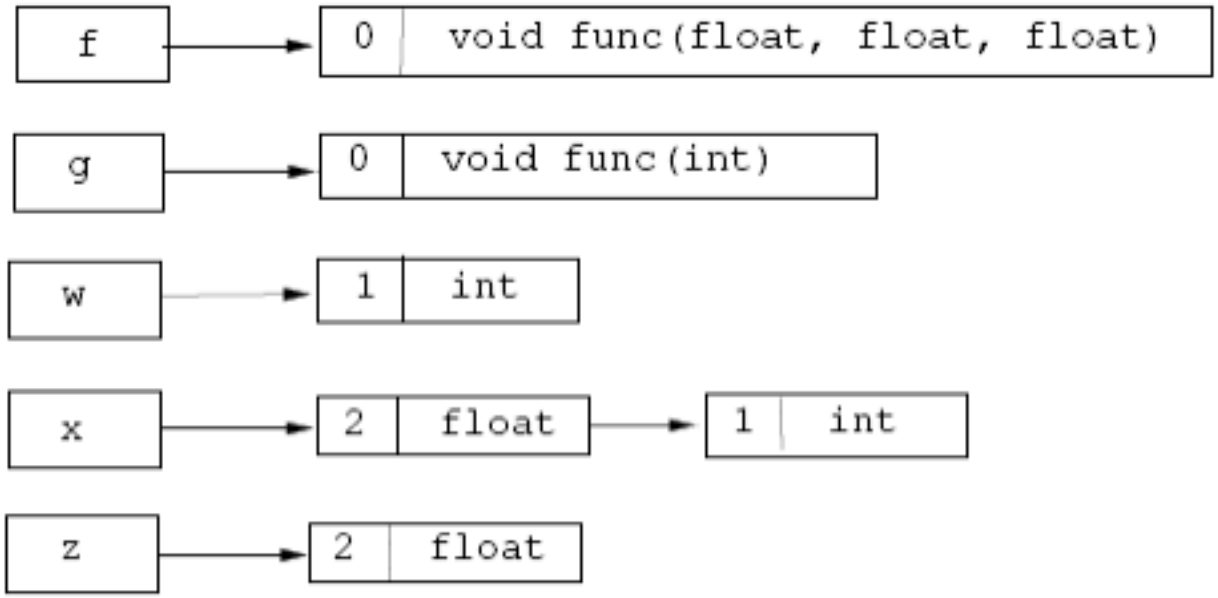


Figure 8.4: An ordered list of symbol stacks

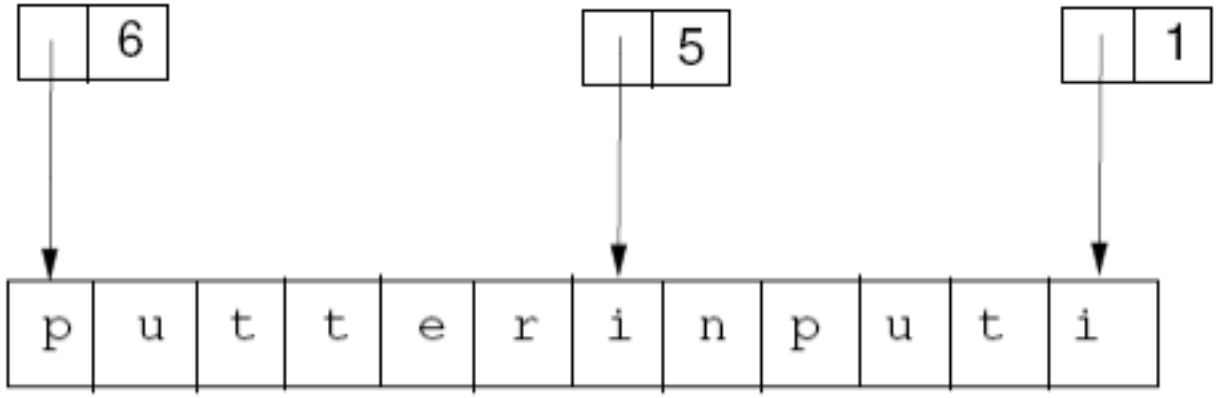


Figure 8.5: Name space for symbols *putter*, *input*, and *i*

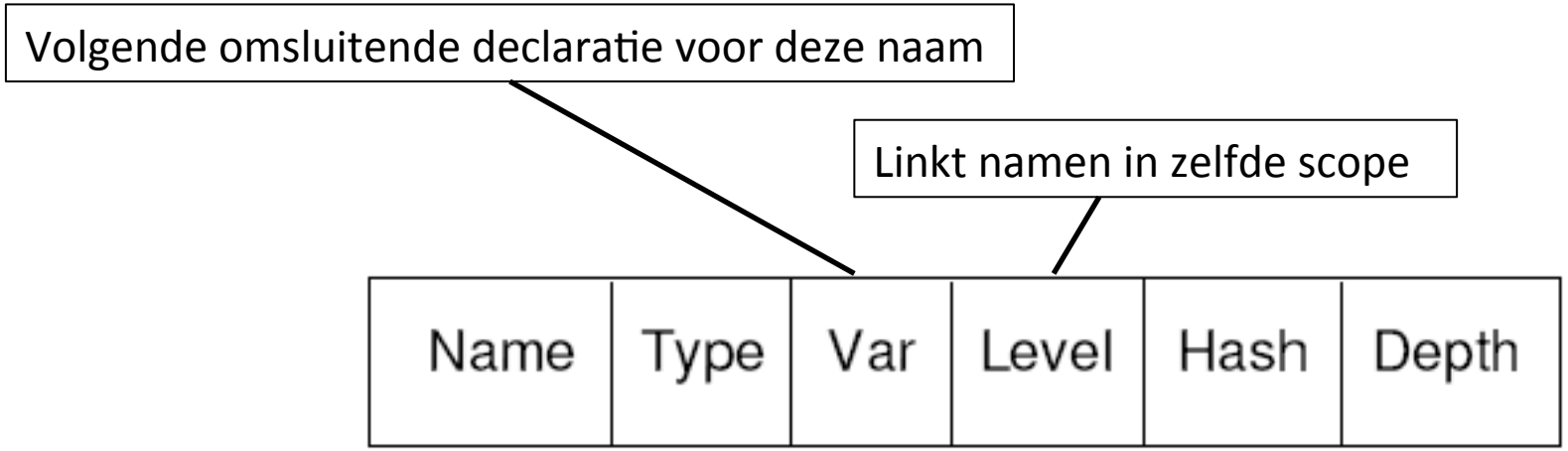


Figure 8.6: A symbol table entry

f en g, w en z hebben zelfde hash locatie

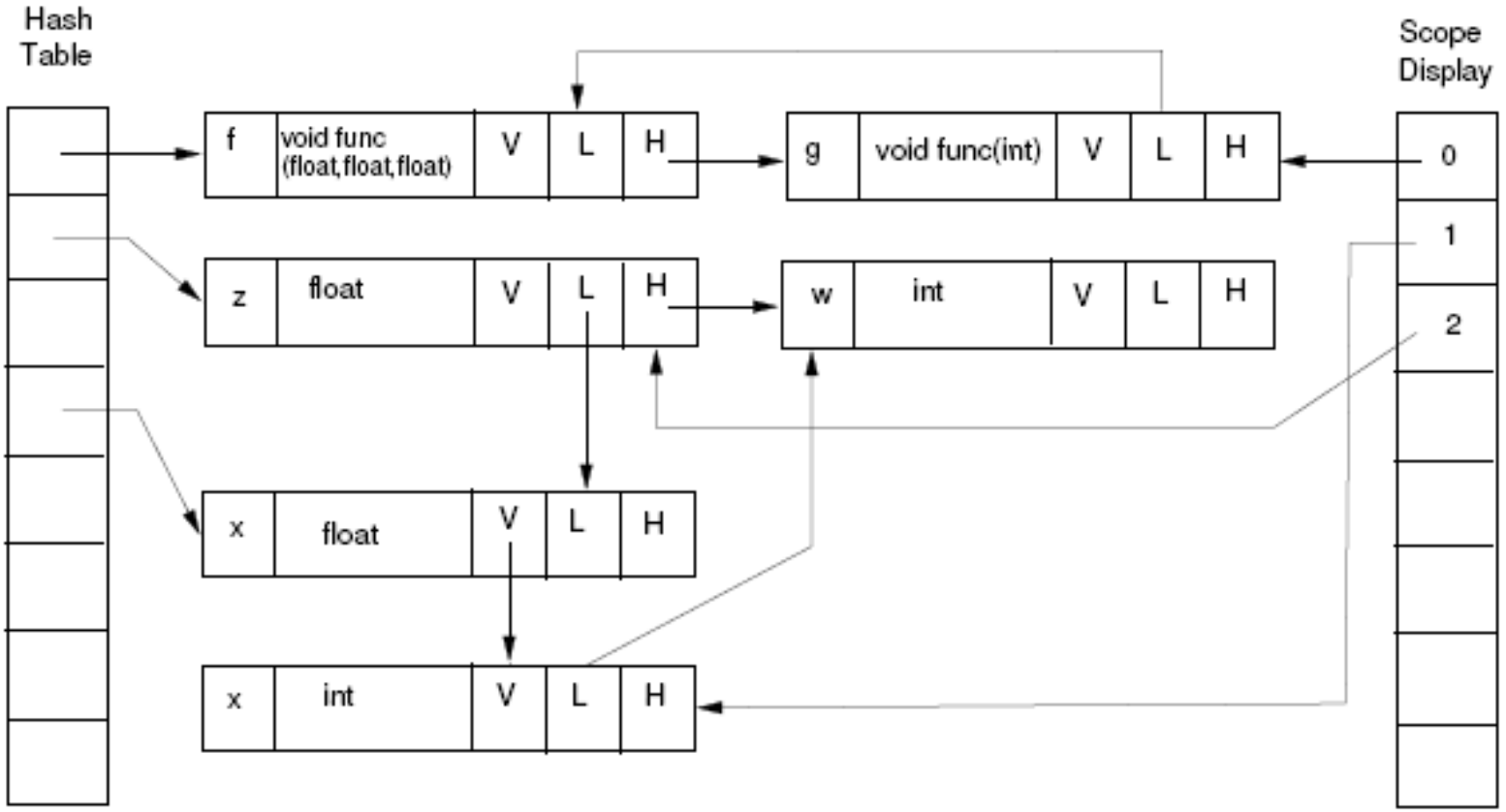


Figure 8.8: Detailed layout of the symbol table for Figure 8.1. The V, L, and H fields abbreviate the Var, Level, and Hash fields, respectively

Syntax-Directed Translation

Syntax-gestuurde organizatie van (o.a.) de semantische analyse

Het is wenselijk ook de semantische analyse te kunnen specifiëren met behulp van grammatica.

De semantische waarden en semantische acties worden geintegreerd in de symbols en producties van de grammatica, en dus in het parsing proces.

Semantische waarden: grammaticale symbolen worden uitgerust met attributen, variabele velden voor semantische waarden. Er zijn 2 soorten attributen: synthesized attributen worden bottom-up berekend (attributen van de LHS worden berekend op basis van die van de RHS) en inherited attributen (attributen van de RHS worden berekend op basis van die van de LHS of van andere attributen van de LHS)

Semantische acties: worden geassocieerd met de producties. Kunnen evt Uitgevoerd worden wanneer de parser een nonterminal expandeert (top-down parsing) of een handle reduceert (shift-reduce parsing).

Voorbeeld: evaluatie van een expressie

Synthesized attribute “value”
voor alle symbolen behalve +,*,$

Voor terminale symbolen:
geinitialiseerd door de scanner

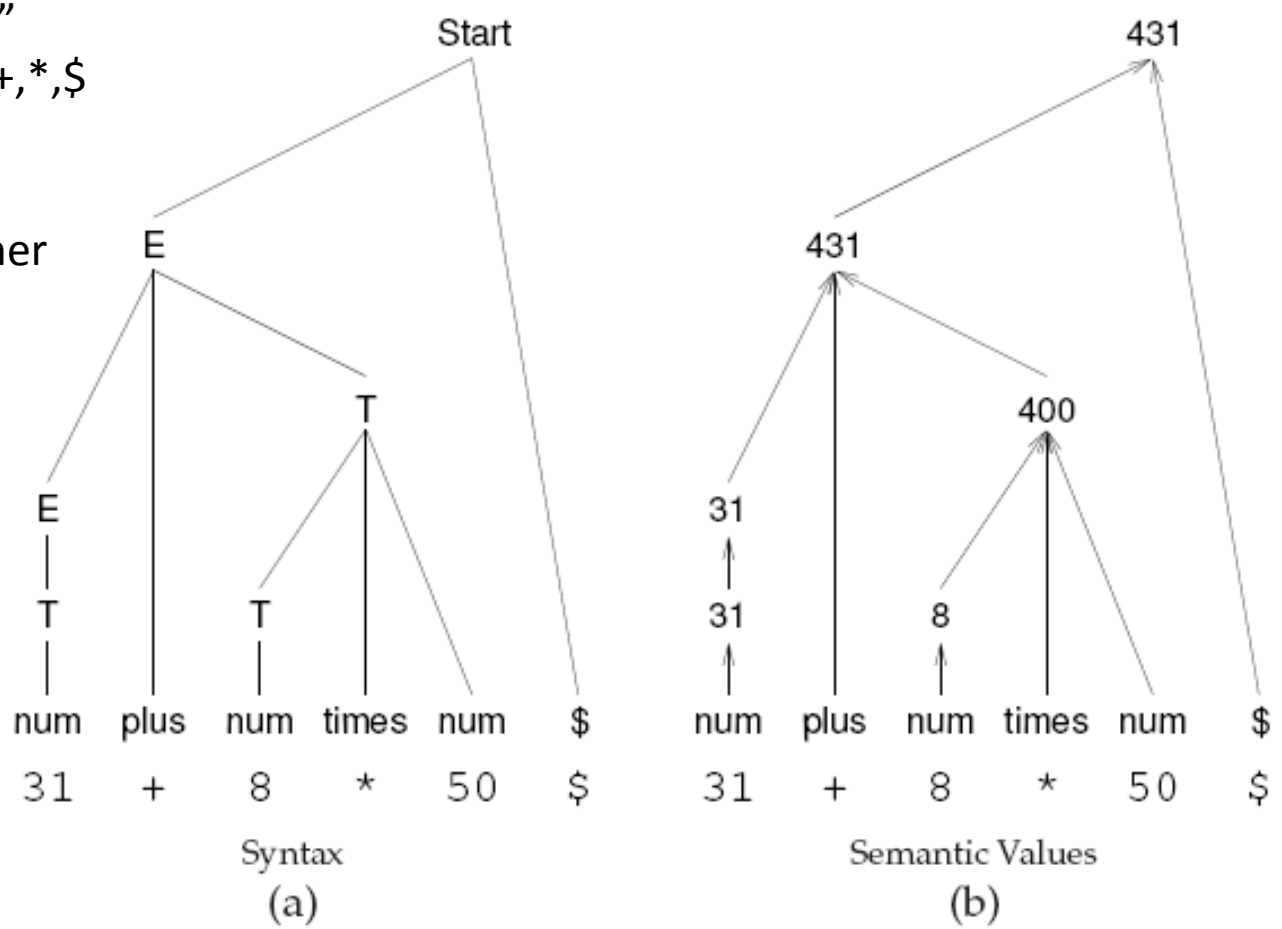


Figure 7.1: (a) Parse tree for the displayed expression;
(b) Synthesized attributes transmit values up the parse tree toward the root.

Inherited attribute: posities tellen in een string

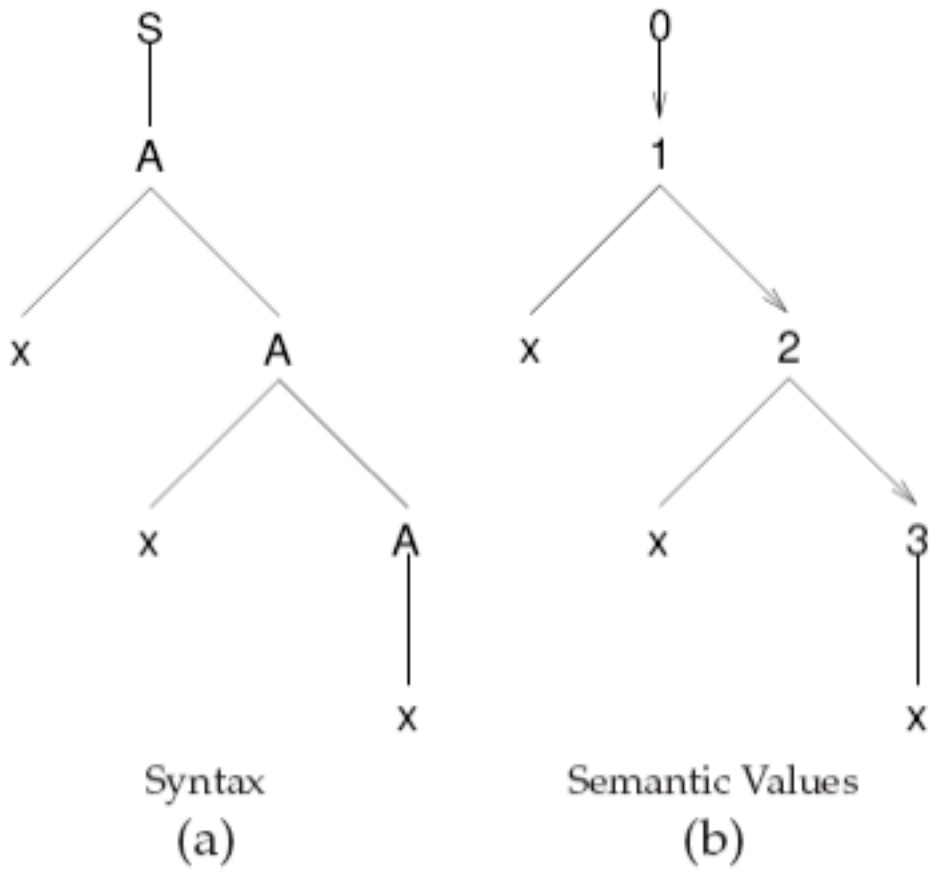


Figure 7.2: (a) Parse tree for the displayed input string; (b) Inherited attributes pass from parent to child.

Bottom-up Syntax-Directed Translation

Het speciale geval dat best geschikt is om te combineren met bottom-up parsing.

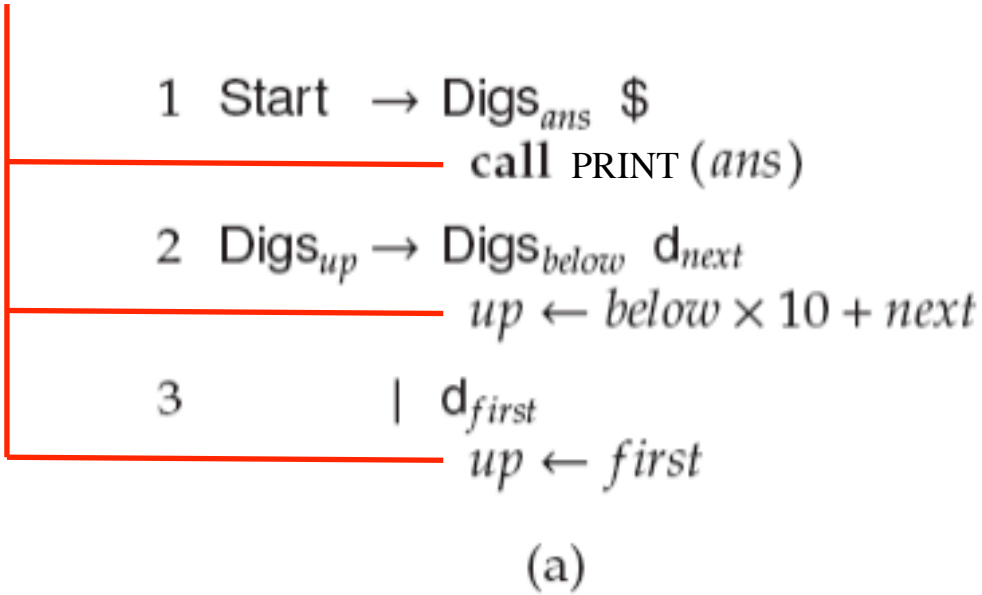
Semantische waarden kunnen op de stack bewaard worden (of op een stack die op dezelfde manier werkt als de syntactische stack : een shift operatie zorgt ervoor dat waarden op de stack gepushed worden, en een reduce operatie popt ze)

We bekijken een rij van met elkaar gerelateerde voorbeelden.

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



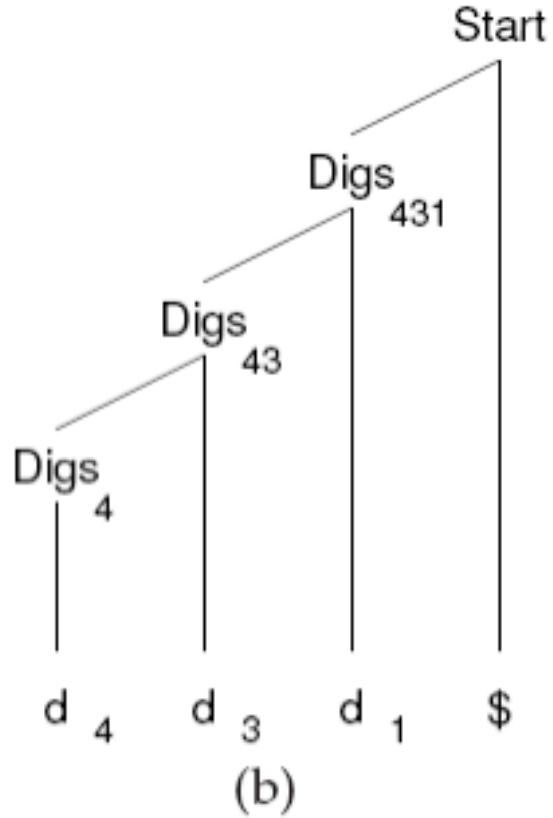


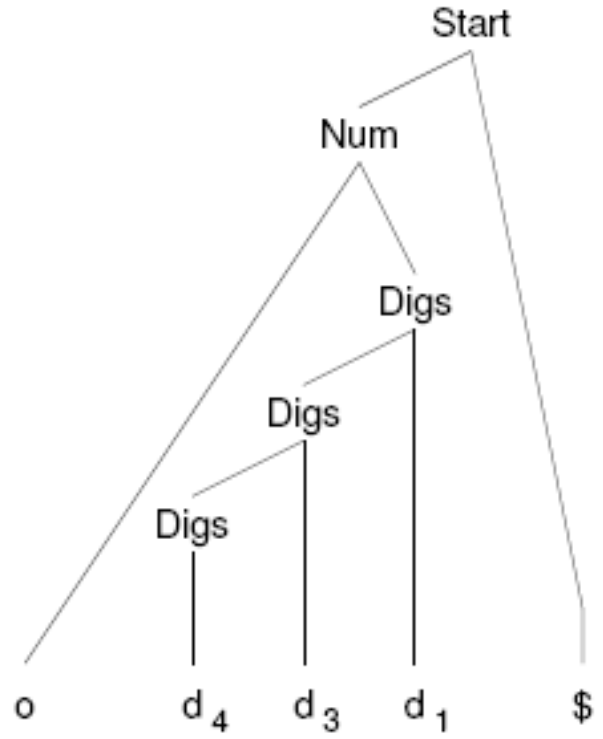
Figure 7.3: (a) Grammar with semantic actions; (b) Parse tree and propagated semantic values for the input 4 3 1 $.

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.