# COMPILERS

D. Janssens, 2015-2016

Nota: een deel van de figuren komt uit volgende handboeken:

Compiler Design - Virtual Machines, R. Wilhelm and H. Seidl, Springer,
ISBN 978-3-642-14908-5

Crafting a Compiler, C.N. Fischer, R.K. Cytron and R.J. LeBlanc, Pearson,

ISBN 987-0-13-801785-9

# Overzicht

- Introductie
- Specificatie van de vertaling
  - ✓ Bron: een imperatieve taal (C, Pascal, Algol,…)
  - ✓ Doel: virtuele machine (P-machine)
- Werking van een compiler
  - ✓ Fazen
  - ✓ Hulpstructuren (AST, symbooltabel)
  - ✓ Optimalisatie
- Project: een compiler voor een imperatieve taal

We concentreren ons op de vertaling van een hoog-niveau programmeertaal  naar een laag-niveau programmeertaal, ook al kan men ook andere vertaalprogramma's als compilers beschouwen. Technieken als parsing of flow analyse hebben ook andere toepassingen.

Hoog-niveau: C, java, C++, Pascal, Python, Scheme, Prolog, …

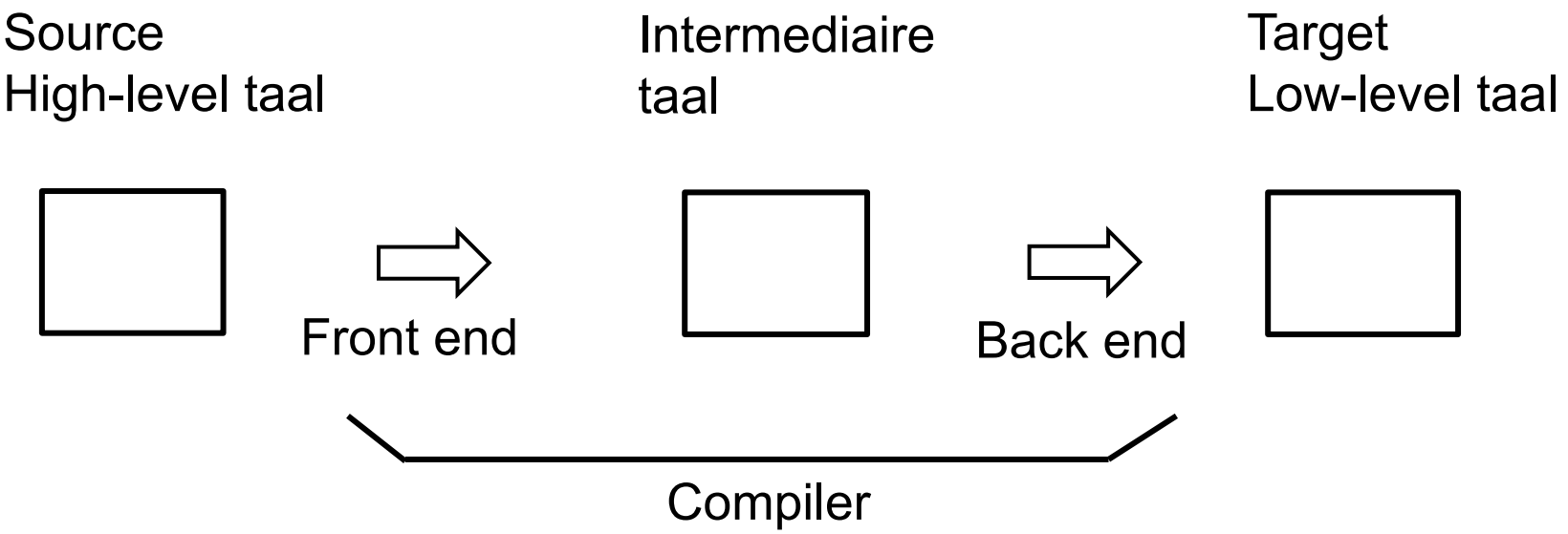
- ✓ Complexe datastructuren en controlestructuren
- ✓ Procedures, subprogramma's
- ✓ Scoping: locale en globale variabelen

Laag-niveau: dicht bij machine

- ✓ Machinetaal
- ✓ Relocatable code
- ✓ Assembleertaal

# Intermediaire taal

Een compiler wordt vaak verdeeld in een front-end en een back-end; die zijn verantwoordelijk voor, respectievelijk, de analyse van het bron-programma en de synthese van de target code



# Intermediaire taal en virtuele machines

- VM = conceptueel eenvoudig machinemodel, “op maat” van de gewenste hogere programmeertaal, makkelijk te interpreteren (P-machine, C-machine, Java Virtual Machine,…)
- Extra voordeel: portability: het is relatief makkelijk een interpreter te schrijven voor een gegeven platform. Eens die beschikbaar is volstaat het de hogere programmeertaal te vertalen in de taal van de VM.
- In het volgende deel van de cursus gebruiken we als doeltaal de taal van zo een VM: de P(ascal) Machine.

# Bootstrapping

Invoering van L, gegeven een compiler voor K en een VM interpreter

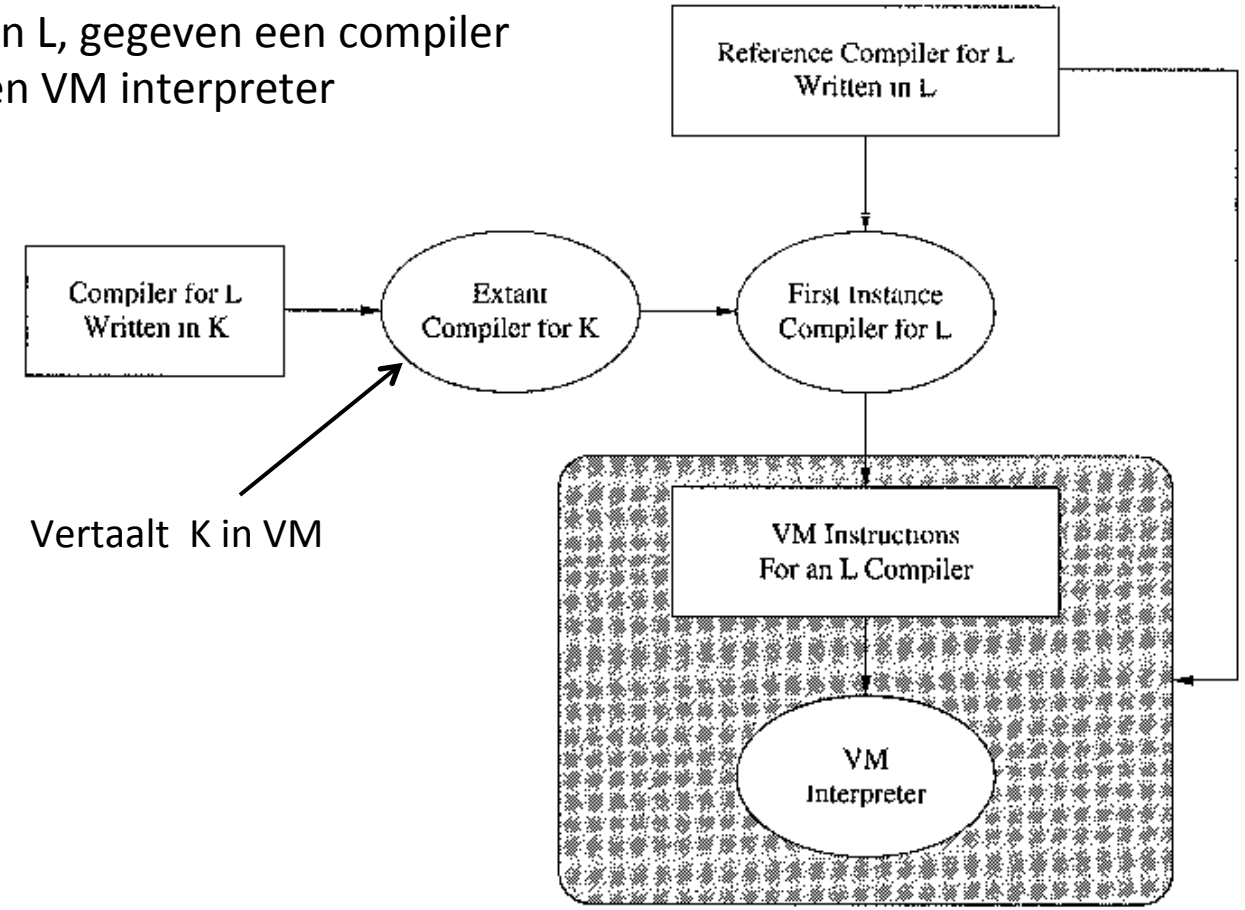



Figure 1.2: Bootstrapping a compiler that generates VM instructions. The shaded portion is a portable compiler for $L$ that can run on any architecture supporting the VM.

Compilers vs. interpreters:

Een compiler vertaalt, een interpreter voert ook uit!

# Interpreters:

- ✓ Control flow berust bij de interpreter
- ✓ Niet veel expliciete vertaling
- ✓ Goed voor debugging, talen met dynamische typering
- ✓ Veel trager: instructies in een loop worden bv. bij elke uitvoering opnieuw vertaald

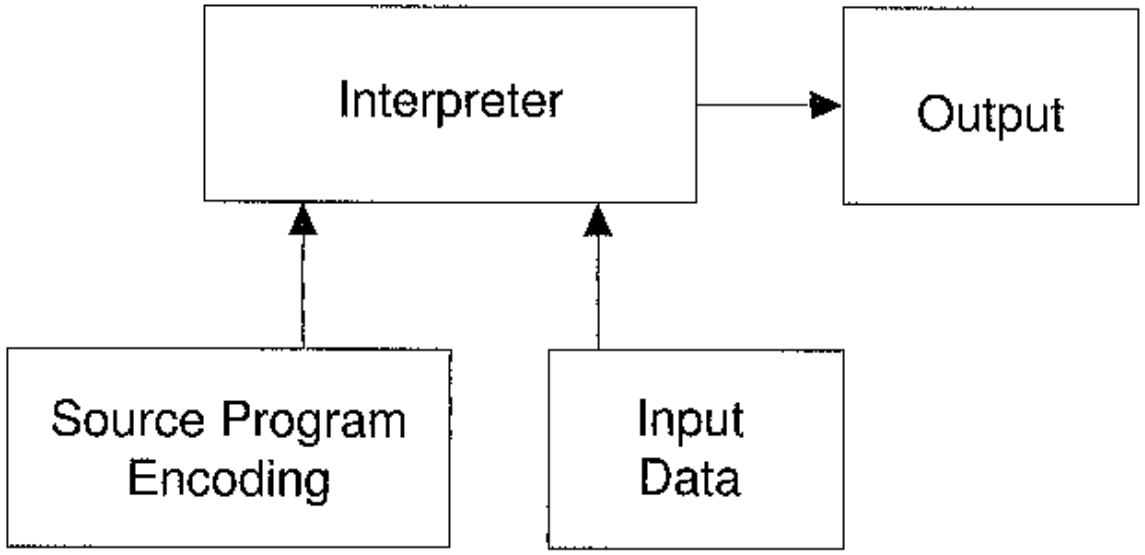



Figure 1.3: An interpreter.

# Voorbeeld: Design van een eenvoudige compiler

Brontaal: adding calculator ac

- 2 numerieke data types (integer, float)
- Berekening (plus, minus) infix
- Assignments aan (kleine) set van variabelen
- Printen van constante of variabele

- Automatische conversie van integer naar float

Brontaal:

- Integer: rij decimale cijfers
- float: ook 5 cijfers na de komma
- variables: roman letters, except f, i, p
- keywords: f, i, p

Voorbeeld:

```
f b i a a = 5 b = a + 3.2  p b
```

wat staat voor

```
Float b    integer a    a=5    b=a+3.2    print b
```

Doeltaal: taal voor berekening op stack, (dc)

met reverse polish notatie, dus a+b wordt ab+

- Data: Integer en float (zoals input)
- Variabelen (zoals input)
- Berekeningen: plus en minus (pop telkens de 2 argumenten en zet het resultaat op de stack)
- Instructies: push c, load a, store a (c constante , a variabele)
- Set/reset  precision: 5 k voor float en 0 k voor integer
- Print top of stack (p s i)

Voorbeeld:

f b i a a = 5 b = a + 3.2 p b

wordt

5 s a 0 k l a 5 k 3.2 + s b 0 k l b p s i

wat staat voor

push 5, store a, reset precision, load a , set precision, push 3.2, add, store b, reset precision, load b, print(en pop)

Vertaling:

- Infix naar postfix
- Set/reset precision toevoegen op basis van de typedeclaraties

# Syntax van de brontaal (contextvrije grammatica)

```
 1  Prog  → Dcls Stmts $
 2  Dcls  → Dcl Dcls
 3        | λ
 4  Dcl   → floatdcl id
 5        | intdcl id
 6  Stmts → Stmt Stmts
 7        | λ
 8  Stmt  → id assign Val Expr
 9        | print id
10  Expr  → plus Val Expr
11        | minus Val Expr
12        | λ
13  Val   → id
14        | inum
15        | fnum
```

Terminale symbolen zoals floatdcl, id, … noemen we tokens. Ze zullen nog verder verfijnd worden.

Figure 2.1: Context-free grammar for ac

| Step | Sentential Form | Production Number |
|---|---|---|
| 1 | ⟨Prog⟩ | |
| 2 | ⟨Dcls⟩ Stmts $ | 1 |
| 3 | ⟨Dcl⟩ Dcls Stmts $ | 2 |
| 4 | floatdcl id ⟨Dcls⟩ Stmts $ | 4 |
| 5 | floatdcl id ⟨Dcl⟩ Dcls Stmts $ | 2 |
| 6 | floatdcl id intdcl id ⟨Dcls⟩ Stmts $ | 5 |
| 7 | floatdcl id intdcl id ⟨Stmts⟩ $ | 3 |
| 8 | floatdcl id intdcl id ⟨Stmt⟩ Stmts $ | 6 |
| 9 | floatdcl id intdcl id id assign ⟨Val⟩ Expr Stmts $ | 8 |
| 10 | floatdcl id intdcl id id assign inum ⟨Expr⟩ Stmts $ | 14 |
| 11 | floatdcl id intdcl id id assign inum ⟨Stmts⟩ $ | 12 |
| 12 | floatdcl id intdcl id id assign inum ⟨Stmt⟩ Stmts $ | 6 |
| 13 | floatdcl id intdcl id id assign inum id assign ⟨Val⟩ Expr Stmts $ | 8 |
| 14 | floatdcl id intdcl id id assign inum id assign id ⟨Expr⟩ Stmts $ | 13 |
| 15 | floatdcl id intdcl id id assign inum id assign id plus ⟨Val⟩ Expr Stmts $ | 10 |
| 16 | floatdcl id intdcl id id assign inum id assign id plus fnum ⟨Expr⟩ Stmts $ | 15 |
| 17 | floatdcl id intdcl id id assign inum id assign id plus fnum ⟨Stmts⟩ $ | 12 |
| 18 | floatdcl id intdcl id id assign inum id assign id plus fnum ⟨Stmt⟩ Stmts $ | 6 |
| 19 | floatdcl id intdcl id id assign inum id assign id plus fnum print id ⟨Stmts⟩ $ | 9 |
| 20 | floatdcl id intdcl id id assign inum id assign id plus fnum print id $ | 7 |

Figure 2.2: Derivation of an ac program using the grammar in Figure 2.1.

# Verfijning van de tokens, met behulp van reguliere expressies

| Terminal | Regular Expression |
|---|---|
| floatdcl | `"f"` |
| intdcl | `"i"` |
| print | `"p"` |
| id | `[a – e] \| [g – h] \| [j – o] \| [q – z]` |
| assign | `"="` |
| plus | `"+"` |
| minus | `"-"` |
| inum | $[0-9]^+$ |
| fnum | $[0-9]^+ . [0-9]^+$ |
| blank | $(" ")^+$ |

Figure 2.3: Formal definition of ac tokens.

# Parse tree (afleidingsboom)

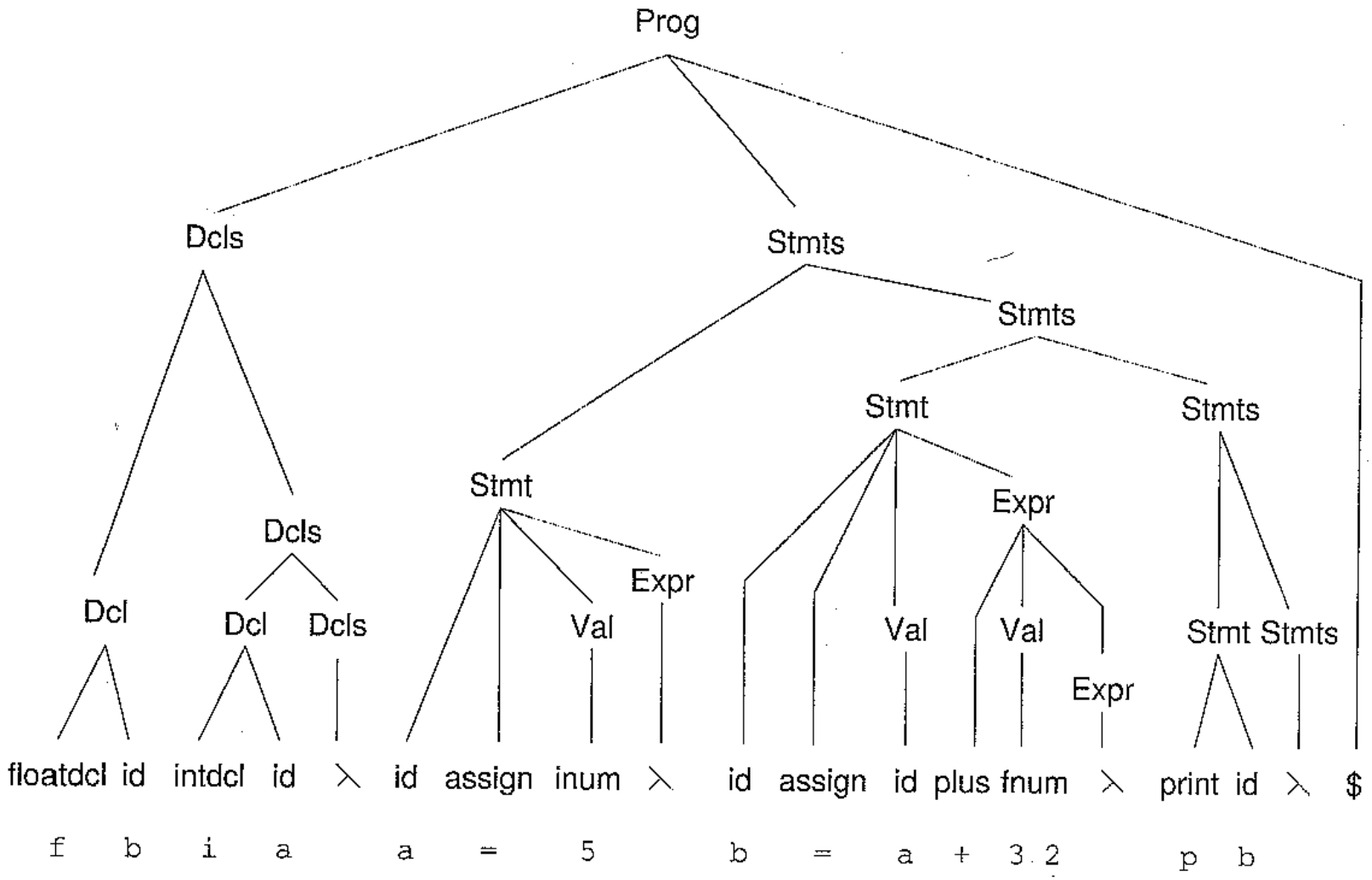



Figure 2.4: An ac program and its parse tree.

Analyse

Eerste faze:
herkennen van de tokens in een stream van karakters

```
function SCANNER( ) returns Token
    while s.PEEK( ) = blank do call s.ADVANCE( )          skip blanks
    if s.EOF( )
    then ans.type ← $
    else
        if s.PEEK( ) ∈ {0, 1, ..., 9}                     lookahead
        then ans ← SCANDIGITS( )
        else
            ch ← s.ADVANCE( )                             1-symbol tokens
            switch (ch)
                case {a, b, ..., z} − {i, f, p}           a token has 2 components:
                    ans.type ← id                         type and value
                    ans.val ← ch
                case f
                    ans.type ← floatdcl
                case i
                    ans.type ← intdcl
                case p
                    ans.type ← print
                case =
                    ans.type ← assign
                case +
                    ans.type ← plus
                case -
                    ans.type ← minus
                case default
                    call LEXICALERROR( )
    return (ans)
end
```

Figure 2.5: Scanner for the ac language. The variable $s$ is an input stream of characters.

# Hulpfunctie ScanDigits herkent het volgende numerieke token

```
function ScanDigits( ) returns token
    tok.val ← " "
    while s.peek( ) ∈ {0, 1, . . . , 9} do
        tok.val ← tok.val + s.advance( )
    if s.peek( ) ≠ "."
    then tok.type ← inum
    else
        tok.type ← fnum
        tok.val ← tok.val + s.advance( )
        while s.peek( ) ∈ {0, 1, . . . , 9} do
            tok.val ← tok.val + s.advance( )
    return (tok)
end
```

concateneer met volgende karakter, lees verder

Figure 2.6: Finding inum or fnum tokens for the ac language.

# Analyse: parsing (opbouwen van de boom)

```
procedure STMT( )
    if ts.PEEK( ) = id
    then
        call MATCH(ts, id)
        call MATCH(ts, assign)
        call VAL( )
        call EXPR( )
    else
        if ts.PEEK( ) = print
        then
            call MATCH(ts, print)
            call MATCH(ts, id)
        else
            call ERROR( )
end
```

Procedure voor "herken een Stmt"

Predict set voor deze productie

komt overeen met de rechterkant van een productie

herken een token



Figure 2.7: Recursive-descent parsing procedure for Stmt. The variable *ts* is an input stream of tokens.

# Analoog voor de nonterminal Stmts

```
procedure SIMIS( )
    if ts.PEEK( ) = id or ts.PEEK( ) = print
    then
        call SIMI( )                    }
        call SIMIS( )                   }  eerste productie voor Stmts
    else
        if ts.PEEK( ) = $
        then
            /*   do nothing for λ-production
        else   call ERROR( )
end
```

Figure 2.8: Recursive-descent parsing procedure for Stmts.

Abstract syntax tree (AST): parse tree, maar met overbodige knopen weggelaten

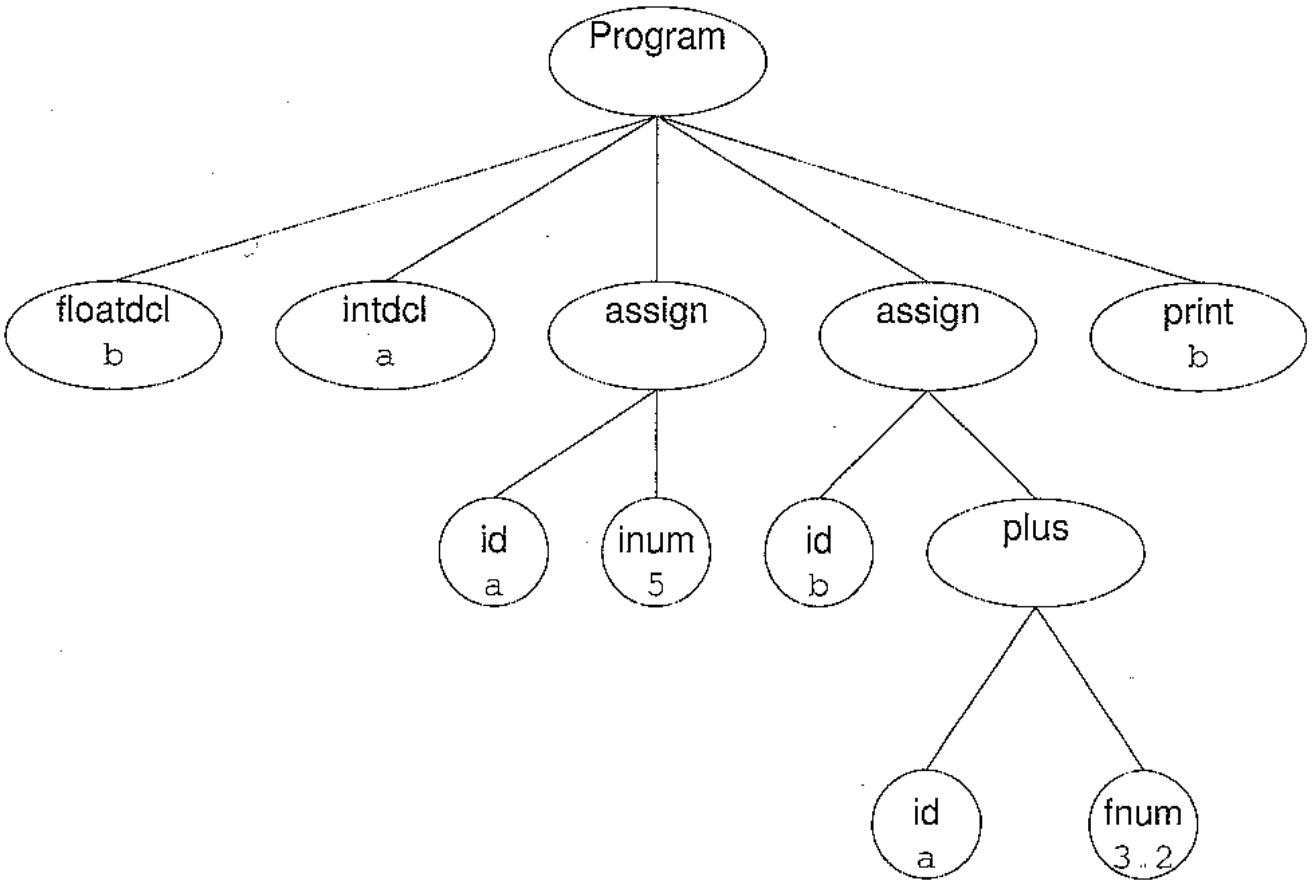



Figure 2.9: An abstract syntax tree for the ac program shown in Figure 2.4.

# Analyse – Verwerking van de declaraties: opbouw van de symbooltabel

| Symbol | Type | Symbol | Type | Symbol | Type |
|---|---|---|---|---|---|
| a | integer | k | null | t | null |
| b | float | l | null | u | null |
| c | null | m | null | v | null |
| d | null | n | null | w | null |
| e | null | o | null | x | null |
| g | null | q | null | y | null |
| h | null | r | null | z | null |
| j | null | s | null | | |

Figure 2.11: Symbol table for the ac program from Figure 2.4

# Opbouw en gebruik van de symbooltabel bij een doortocht van de boom

```
/*   Visitor methods
procedure VISIT(SymDeclaring n)
    if n.GETTYPE( ) = floatdcl
    then call ENTERSYMBOL(n.GETID( ), float)
    else call ENTERSYMBOL(n.GETID( ), integer)
end

/*   Symbol table management
procedure ENTERSYMBOL(name, type)
    if SymbolTable[name] = null
    then SymbolTable[name] ← type
    else call ERROR("duplicate declaration")
end

function LOOKUPSYMBOL(name) returns type
    return (SymbolTable[name])
end
```

Figure 2.10: Symbol table construction for ac.

```
/*    Visitor methods                                              */
procedure VISIT(Computing n)
    n.type ← CONSISTENT(n.child1, n.child2)
end
procedure VISIT(Assigning n)
    n.type ← CONVERT(n.child2, n.child1.type)
end
procedure VISIT(SymReferencing n)
    n.type ← LOOKUPSYMBOL(n.id)
end
procedure VISIT(IntConsting n)
    n.type ← integer
end
procedure VISIT(FloatConsting n)
    n.type ← float
end
/*    Type-checking utilities                                      */
function CONSISTENT(c1, c2) returns type
    m ← GENERALIZE(c1.type, c2.type)
    call CONVERT(c1, m)
    call CONVERT(c2, m)
    return (m)
end
function GENERALIZE(t1, t2) returns type
    if t1 = float or t2 = float
    then ans ← float
    else ans ← integer
    return (ans)
end
procedure CONVERT(n, t)
    if n.type = float and t = integer
    then call ERROR("Illegal type conversion")
    else
        if n.type = integer and t = float
        then
            /*    replace node n by convert-to-float of node n    */ (13)
        else /* nothing needed */
end
```

Vul type in

Bepaal type waarnaar de twee argumenten kunnen geconverteerd worden

Voeg conversie-knoop toe in AST als nodig

Figure 2.12: Type analysis for ac.

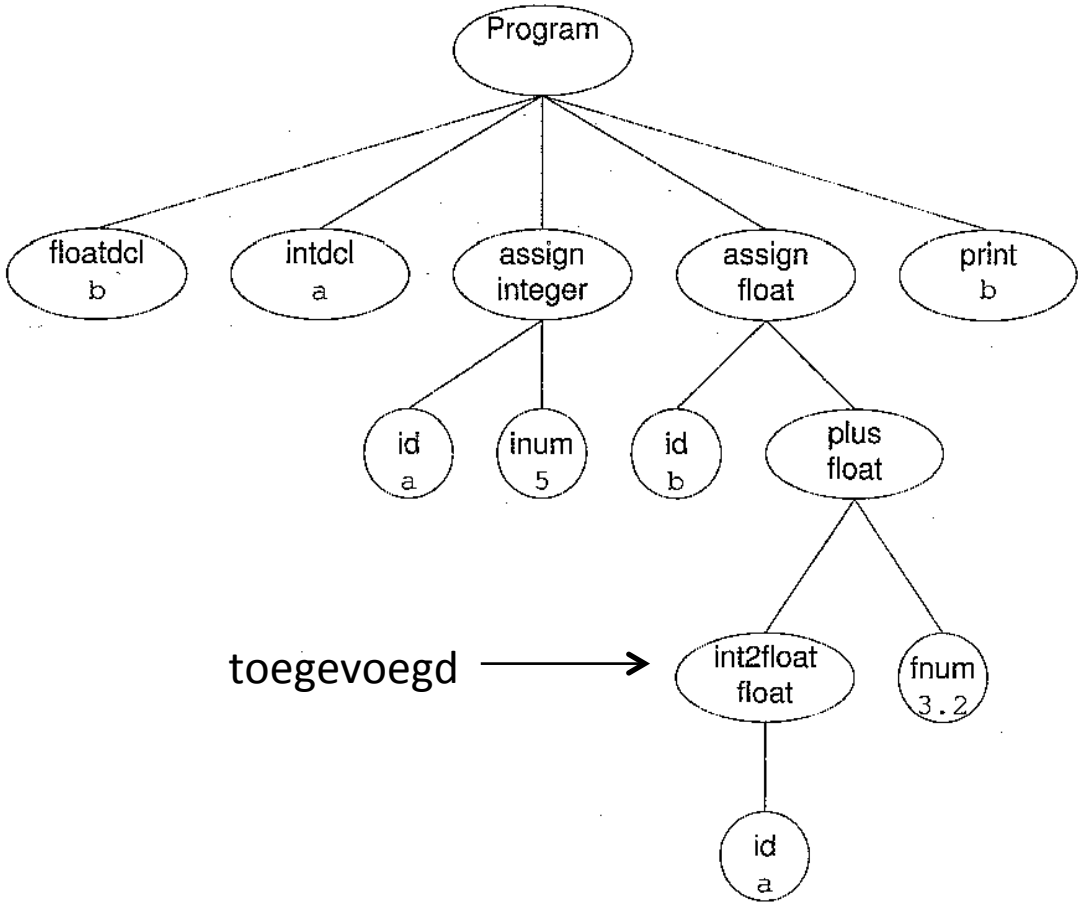



Figure 2.13: AST after semantic analysis.

# Synthese - Codegeneratie

```
procedure VISIT(Assigning n)
    call CODEGEN(n.child2)
    call EMIT("s")
    call EMIT(n.child1.id)
    call EMIT("0 k")
end
procedure VISIT(Computing n)
    call CODEGEN(n.child1)
    call CODEGEN(n.child2)
    call EMIT(n.operation)
end
procedure VISIT(SymReferencing n)
    call EMIT("l")
    call EMIT(n.id)
end
procedure VISIT(Printing n)
    call EMIT("l")
    call EMIT(n.id)
    call EMIT("p")
    call EMIT("si")
end
procedure VISIT(Converting n)
    call CODEGEN(n.child)
    call EMIT("5 k")
end
procedure VISIT(Consting n)
    call EMIT(n.val)
end
```

Figure 2.14: Code generation for ac

Bezoek de knopen van de "gedecoreerde" AST (depth –first, L naar R) en genereer voor elke bezochte knoop een stuk code

Store in i: truuk om effect van "pop" te krijgen

# Resultaat van de vertaling

| Code | Source | Comments |
|---|---|---|
| `5` | `a = 5` | Push 5 on stack |
| `sa` | | Pop the stack, storing (s) the popped value in register a |
| `0 k` | | Reset precision to integer |
| `la` | `b = a + 3.2` | Load (l) register a, pushing its value on stack |
| `5 k` | | Set precision to float |
| `3.2` | | Push 3.2 on stack |
| `+` | | Add: 5 and 3.2 are popped from the stack and their sum is pushed |
| `sb` | | Pop the stack, storing the result in register b |
| `0 k` | | Reset precision to integer |
| `lb` | `p b` | Push the value of the b register |
| `p` | | Print the top-of-stack value |
| `si` | | Pop the stack by storing into the i register |

Figure 2.15: Code generated for the AST shown in Figure 2.9.

# Fasen van een compiler

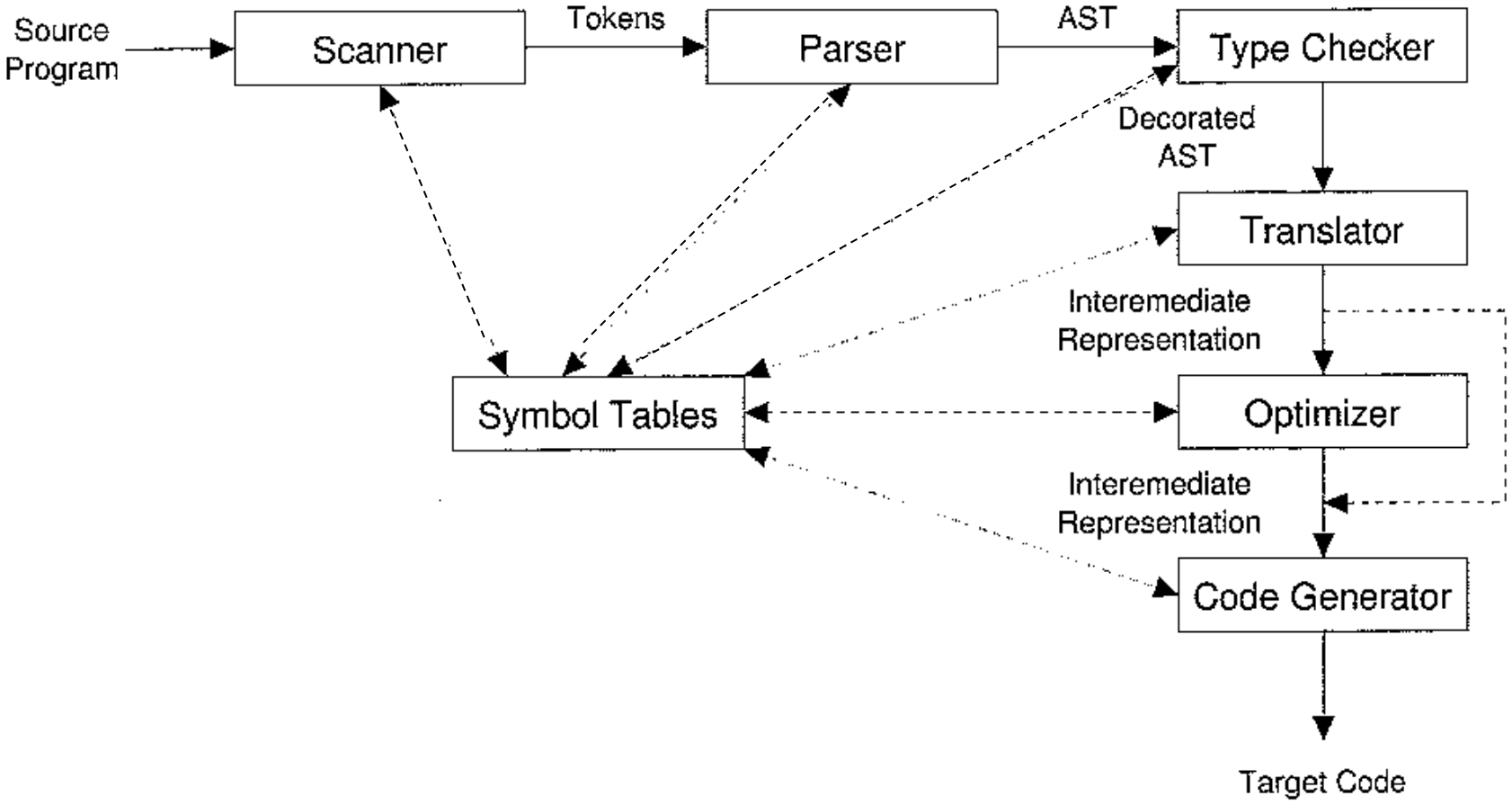



Figure 1.4: A syntax-directed compiler. AST denotes the Abstract Syntax Tree.

# Syntax en Semantiek

- Syntax: de vorm van een correct programma.

  Methoden:
  - ✓ Contextvrije grammatica's (CFG). Elk programma heeft een (abstracte)afleidingsboom of AST.
  - ✓ Reguliere expressies (structuur van de tokens)
  - ✓ Andere: statische semantiek (bv elke variabele heeft een type)
- Semantiek: de betekenis van een programma, m.a.w. het effect van de uitvoering ervan op een willekeurige input.

- Correctheid van een compiler: de vertaling moet de (runtime) semantiek bewaren. Ook de optimizaties moeten het gedrag behouden.

- Van ad-hoc benadering tot systematische opbouw
- Breed  toepasbaar:
  - ✓ Text editing systemen (Tex, postScript)
  - ✓ Mathematica
  - ✓ Silicon compilers
  - ✓ Database query languages
  - ✓ … (elke applicatie met text-oriented command set)
  - ✓ Verificatie tools
- Keuze van target talen:
  - ✓ Pure machine code
  - ✓ Augmented machine code
  - ✓ Virtual machine code
- Portability

# Vervolg: compilatie van een imperatieve taal

Nadruk op de HL features:

- Control flow
- Datatypes, inclusief arrays en structs
- Procedures
- Scoping

Centrale vraag:

Waar vinden de VM instructies hun data?