L-attributierte Grammatiken

- zugelassene Attributarten:
  - synthetisierte Attribute
  - ererbte Attribute mit folgender Eigenschaft
    betrachte die Produktion $N \rightarrow M_1, ..., M_n$
    ein ererbtes Attribut von M_i ist nur bestimmt durch
    - ererbte Attribute von N
    - synthetisierte Attribute von $M_1, ..., M_{i-1}$
- Konsequenz (Abb. 3.31)
  - Abarbeitung von links nach rechts: alle Attributwerte verfügbar
  - relevante ererbte Information liegt auf dem Pfad zur Wurzel
  - notwendig für "narrow"-Compiler
- Beispiel: Subskripte in EQN
  (Drachenbuch: Bsp. 5.13, Abb. 5.22, Abb. 5.23)

L-Attributierung und Parsing

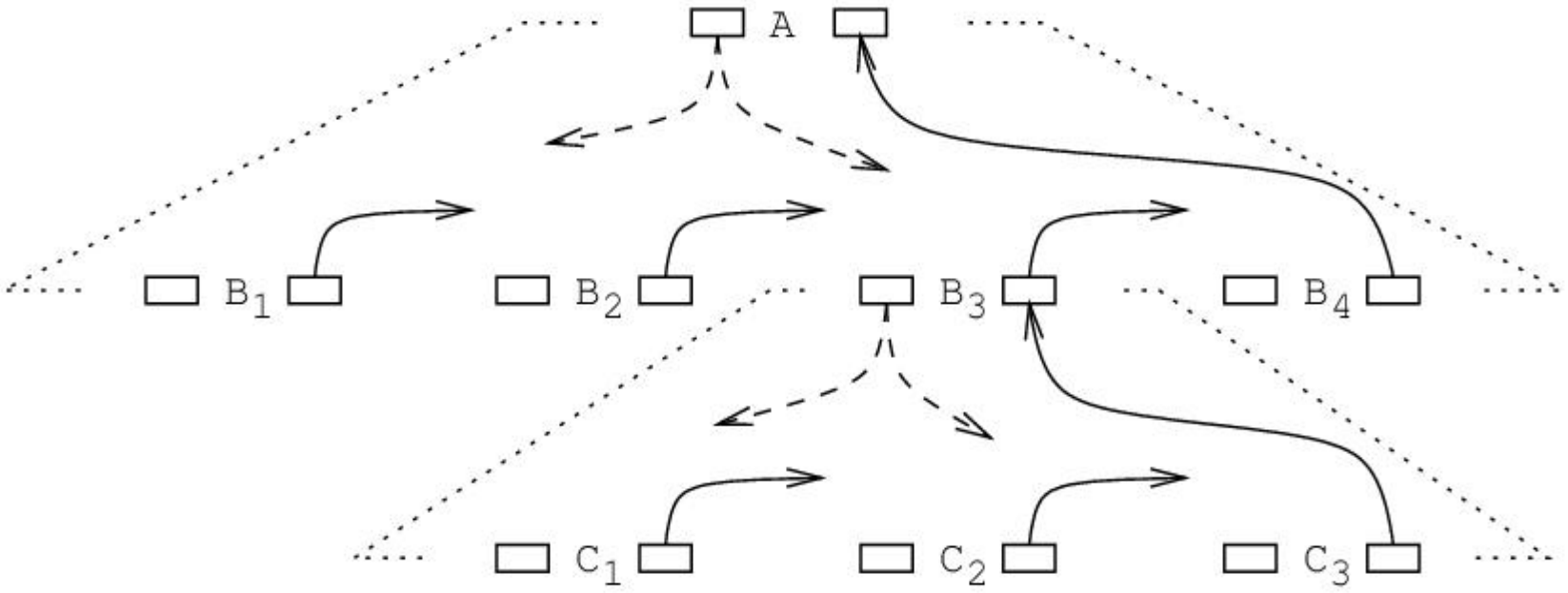


Figure 3.31 Data flow in part of a parse tree for an L-attributed grammar.

```
{
#include    "symbol_table.h"
}

%lexical get_next_token_class;
%token IDENTIFIER;
%token DIGIT;

%start Main_Program, main;

main {symbol_table sym_tab; int result;}:
    {init_symbol_table(&sym_tab);}
    declarations(sym_tab)
    expression(sym_tab, &result)
    {printf("result = %d\n", result);}
;

declarations(symbol_table sym_tab):
    declaration(sym_tab) [ ',' declaration(sym_tab) ]* ';'
;

declaration(symbol_table sym_tab) {symbol_entry *sym_ent;}:
    IDENTIFIER {sym_ent = look_up(sym_tab, Token.repr);}
    '=' DIGIT  {sym_ent->value = Token.repr - '0';}
;

expression(symbol_table sym_tab; int *e) {int t;}:
    term(sym_tab, &t)                {*e = t;}
    expression_tail_option(sym_tab, e)
;

expression_tail_option(symbol_table sym_tab; int *e) {int t;}:
    '-' term(sym_tab, &t)            {*e -= t;}
    expression_tail_option(sym_tab, e)
|
;

term(symbol_table sym_tab; int *t):
    IDENTIFIER      {*t = look_up(sym_tab, Token.repr)->value;}
;
```

Figure 3.32 *LLgen* code for an L-attributed grammar for simple expressions.

L-Attributierung und Parsing

- Top-Down:
  - Tiefensuche des Baums folgt dem Datenfluss der L-Attributierung
  - Bsp.: Ausdrucksauswertung (Drachenbuch, Bsp. 5.14)
- Bottom-Up:
  - Trick: bringe semantische Aktionen ans Ende einer Regel:

    ```
    A   → B {C.inh := f(B.syn); } C
    ```

    wird zu :

    ```
    A   → B A_1 C
    A_1 → ε {C.inh := f(B.syn); }
    ```

  - Bsp.: Typdeklarationen (Drachenbuch, Bsp. 5.17)
  - Vorhersage der Position von ererbten Attributen im Stack (Drachenbuch, Bsp. 5.18-19)
  - Achtung: kann die LALR(1)-Eigenschaft verletzen! (Vorsicht bei ε-Regeln!, Drachenbuch Abb., 5.38)

Äquivalenz von L- und S-Attributierung (1)

- Konversion: L- nach S-Attributierung
  - ersetze ererbtes Attribut durch neues synthetisiertes Attribut S
    S = Funktion/Programmcode + dessen synthetisierte Attribute (Abb. 3.33)
  - oder transformiere die Grammatik, Bsp. (aus Drachenbuch):

```
D -> L : T                 L erbt Attribut
T -> integer | char        type von T (nicht
L -> L , id | id           einmal L-attributiert)
```

wird zu:

```
D -> id L                  L synthetisiert
L -> , id L | : T          Attribut type von T
T -> integer | char        (S-attributiert)
```

- Anmerkungen:
  - kann zu extensivem Postprocessing führen
  - Praktikabilitätsgrenzen
    - Elimination des ererbten Attributs *Symboltabelle*
    - vernünftige Fehlermeldungen

Äquivalenz von L- und S-Attributierung (2)

```
Declaration →
    Type_Declarator(type)  Declared_Idf_Sequence(type)  ';'
;

Declared_Idf_Sequence(INH type) →
    Declared_Idf(type)
|
    Declared_Idf_Sequence(type)  ','  Declared_Idf(type)
;

Declared_Idf(INH type) →
    Idf(repr)
    ATTRIBUTE RULES:
        Add to symbol table (repr, type);
;
```

Figure 3.33 Sketch of an L-attributed grammar for `Declaration`.

Äquivalenz von L -und S-Attributierung (3)

```
Declaration →
    Type_Declarator(type)  Declared_Idf_Sequence(repr list)  ';'
    ATTRIBUTE RULES:
        FOR EACH repr IN repr list:
            Add to symbol table(repr, type);

Declared_Idf_Sequence(SYN repr list) →
    Declared_Idf(repr)
    ATTRIBUTE RULES:
        SET repr list TO Convert to list (repr);
|
    Declared_Idf_Sequence(old repr list)  ','  Declared_Idf(repr)
    ATTRIBUTE RULES:
        SET repr list TO Append to list (old repr list, repr);
;

Declared_Idf(SYN repr) →
    Idf(repr)
;
```

Figure 3.34 Sketch of an S-attributed grammar for `Declaration`.

Attributierungen, Wertefluss

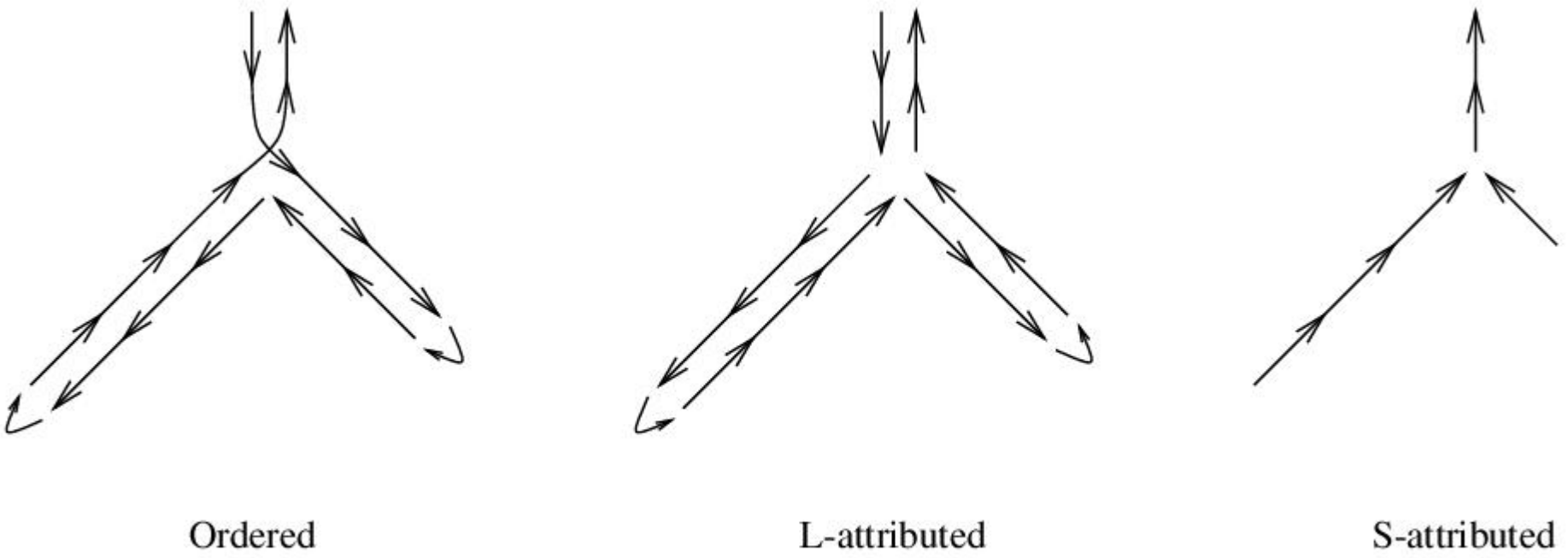


Figure 3.35 Pictorial comparison of three types of attribute grammars.

Manuelle Methoden

- Automatische Generierung von Algorithmen zur Kontextanalyse:
  mittels attributierter Grammatiken
  - Kontextanalyse während des Parsevorgangs
    (L- und S-Attributierung)
  - Kontextanalyse im Anschluss an den Parsevorgang
    (Multi-Visit-Grammatiken)
- Manuelle Generierung von Algorithmen zur Kontextanalyse:
  (a) symbolische Interpretation
  (b) Lösung von Datenflussgleichungen (teilweise Toolunterstützung)
  - Voraussetzung: Durchlauf des Syntaxbaums in der Reihenfolge des Kontrollflusses
- ”Threading” des abstrakten Syntaxbaums:
  - Erstellung des Kontrollflussgraphen für den Baum

Threading des abstrakten Syntaxbaumes

- Prinzip: Postorderdurchlauf (Abb. 3.36)
- Komplikation: Verzweigungen (Abb. 3.39)
- Lösung:
  - Einfügen von zusätzlichen Knoten (Abb. 3.40)
  - Einfügen von Code zur Auswertung der Verzweigungsbedingung

Postorderdurchlauf

Beispiel: `b*b-4*a*c`

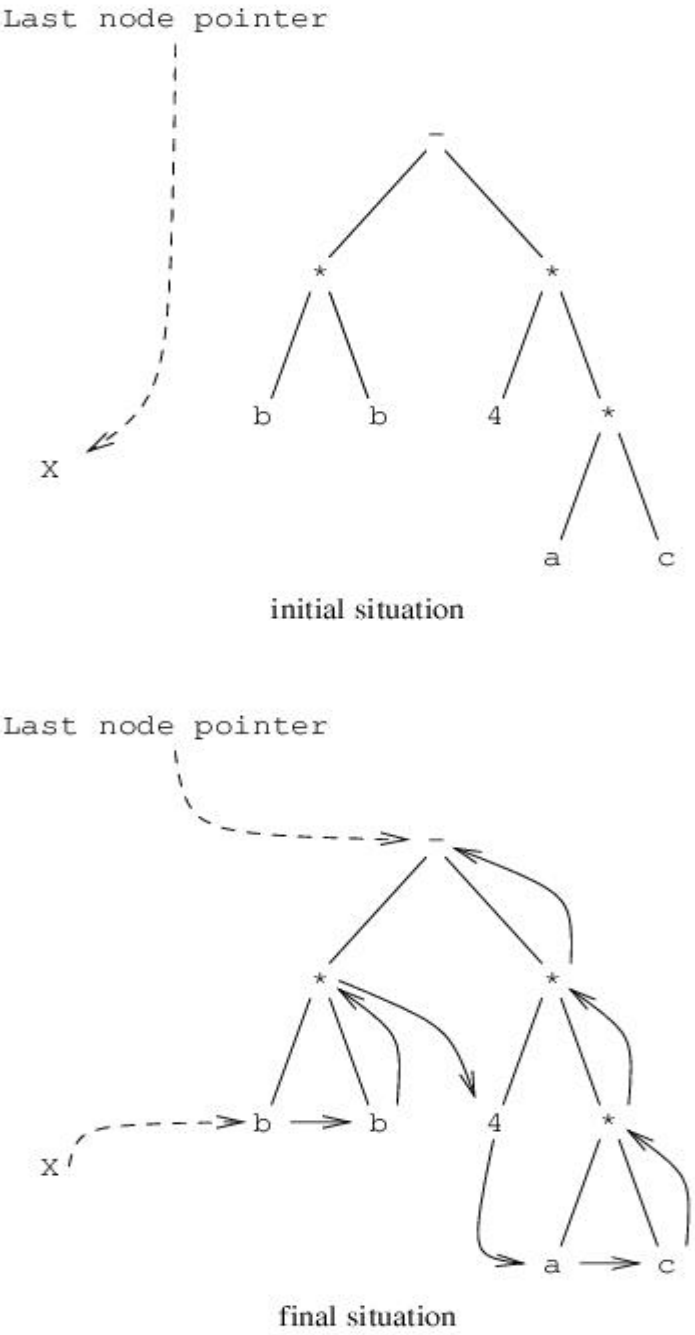


Figure 3.36 Control flow graph for the expression `b*b - 4*a*c`.

Postorderdurchlauf

```
#include    "parser.h"      /* for types AST_node and Expression */
#include    "thread.h"      /* for self check */
                                        /* PRIVATE */
static AST_node *Last_node;

static void Thread_expression(Expression *expr) {
    switch (expr->type) {
    case 'D':
        Last_node->successor = expr; Last_node = expr;
        break;
    case 'P':
        Thread_expression(expr->left);
        Thread_expression(expr->right);
        Last_node->successor = expr; Last_node = expr;
        break;
    }
}
                                        /* PUBLIC */
AST_node *Thread_start;

void Thread_AST(AST_node *icode) {
    AST_node Dummy_node;

    Last_node = &Dummy_node; Thread_expression(icode);
    Last_node->successor = (AST_node *)0;
    Thread_start = Dummy_node.successor;
}
```

Figure 3.37 Threading code for the demo compiler from Section 1.2.

Verzweigungen (1)

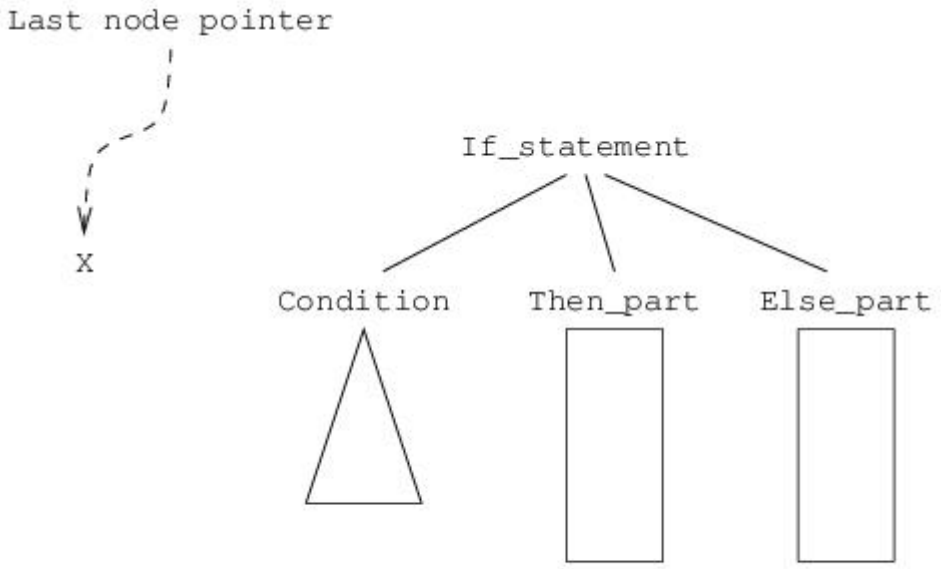


Figure 3.39 AST of an if-statement before threading.

Verzweigungen (2)

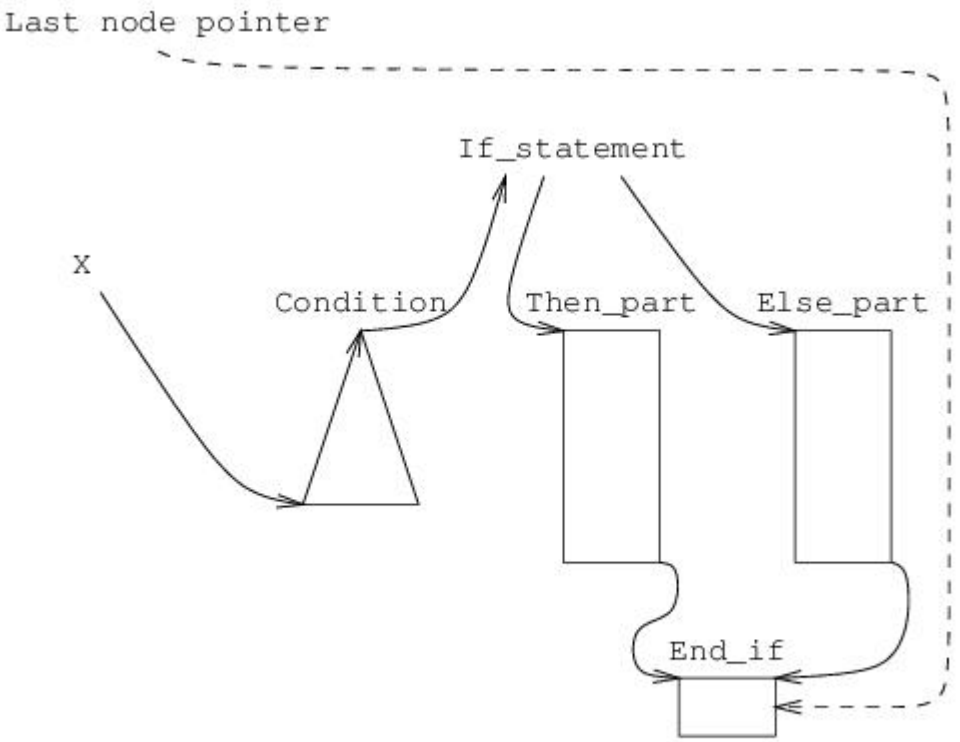


Figure 3.40 AST and control flow graph of an if-statement after threading.

Verzweigungen (3)

```
PROCEDURE Thread if statement (If node pointer):
    Thread expression (If node pointer .condition);
    SET Last node pointer .successor TO If node pointer;

    SET End if join node TO Generate join node ();

    SET Last node pointer TO address of a local node Aux last node;
    Thread block (If node pointer .then part);
    SET If node pointer .true successor TO Aux last node .successor;
    SET Last node pointer .successor TO address of End if join node;

    SET Last node pointer TO address of Aux last node;
    Thread block (If node pointer .else part);
    SET If node pointer .false successor TO Aux last node .successor;
    SET Last node pointer .successor TO address of End if join node;

    SET Last node pointer TO address of End if join node;
```

Figure 3.38 Sample threading routine for if-statements.

Verzweigungen (4)

- Implementierung mit einer attributierten Grammatik
  (Abb. 3.41, ausführliche Beispiele im Drachenbuch, Abschnitt 8.6)
  - ererbtes Attribut: Zeiger zur Verkettung der Durchlaufliste
  - synthetisiertes Attribut: Einstiegspointer
- Zur besseren Navigation
  - Kontrollflussgraph oft doppelt verkettet (Abb. 3.42)

Threading mit Attributregeln

```
If_statement(INH successor, SYN first) →
    'IF'  Condition  'THEN'  Then_part  'ELSE'  Else_part  'END'  'IF'
    ATTRIBUTE RULES:
        SET If_statement .first TO Condition .first;
        SET Condition .true successor TO Then_part .first;
        SET Condition .false successor TO Else_part .first;
        SET Then_part .successor TO If_statement .successor;
        SET Else_part .successor TO If_statement .successor;
```

Figure 3.41 Threading an if-statement using attribute rules.

Doppelte Verkettung

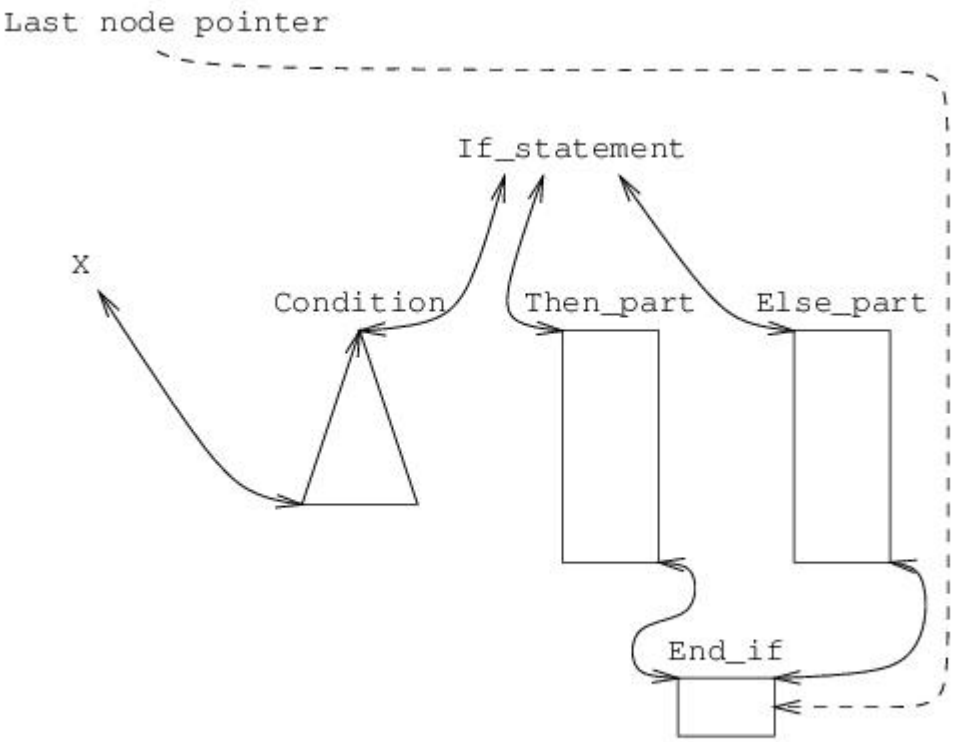


Figure 3.42 AST and doubly-linked control flow graph of an if-statement.

Symbolische Interpretation

(nicht zu verwechseln mit *abstrakter Interpretation*,
welche formal sehr präzise und automatisch ist)

- Compilerphase
  - Context Handling, basierend auf Kontrollflussgraph
- Prinzip
  - Informationsgewinnung über Werte im Programm
    bereits zur Übersetzungszeit ... soweit möglich ...
- Vorgehen
  - Verfolgen des Kontrollflusses
  - Interpretation mit abstrakten Wertemengen oder partieller Information
- Ziele

1. "Optimierung" des Zielprogramms, meist

   Verringerung von Laufzeit und/oder Speicherbedarf

   Bsp.: `2*pi/180 → 3.49066e-2`

2. Erkennung von semantischen Fehlern

   Bsp.: Verwendung nicht definierter Variablen

Symbolische Interpretation / Vorgehen

jede Kante im Kontrollflussgraphen hat einen “Laufzeit”-Stack

- der Stack enthält alle an diesem Punkt sichtbaren Bezeichner
- jeder Eintrag enthält alle Informationen über den Bezeichner

Beispiel:

```
if y>0 then ...
```

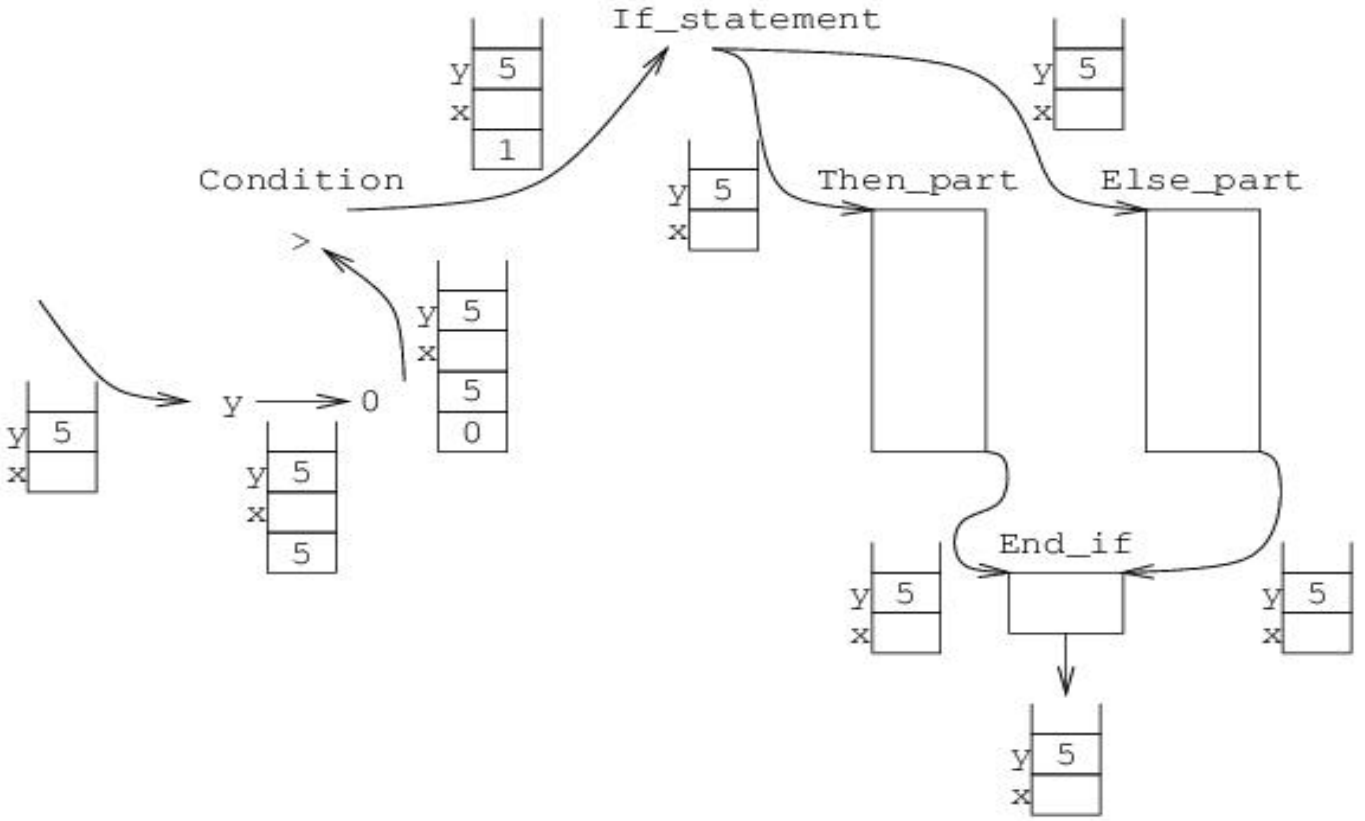


Figure 3.43 Stack representations in the control flow graph of an if-statement.

Symbolische Interpretation / Beispiel (1)

Beispiel:

```
if y>0 then ...
```

Vorbedingungen
- x initialisiert, aber unbekannt
- y=5

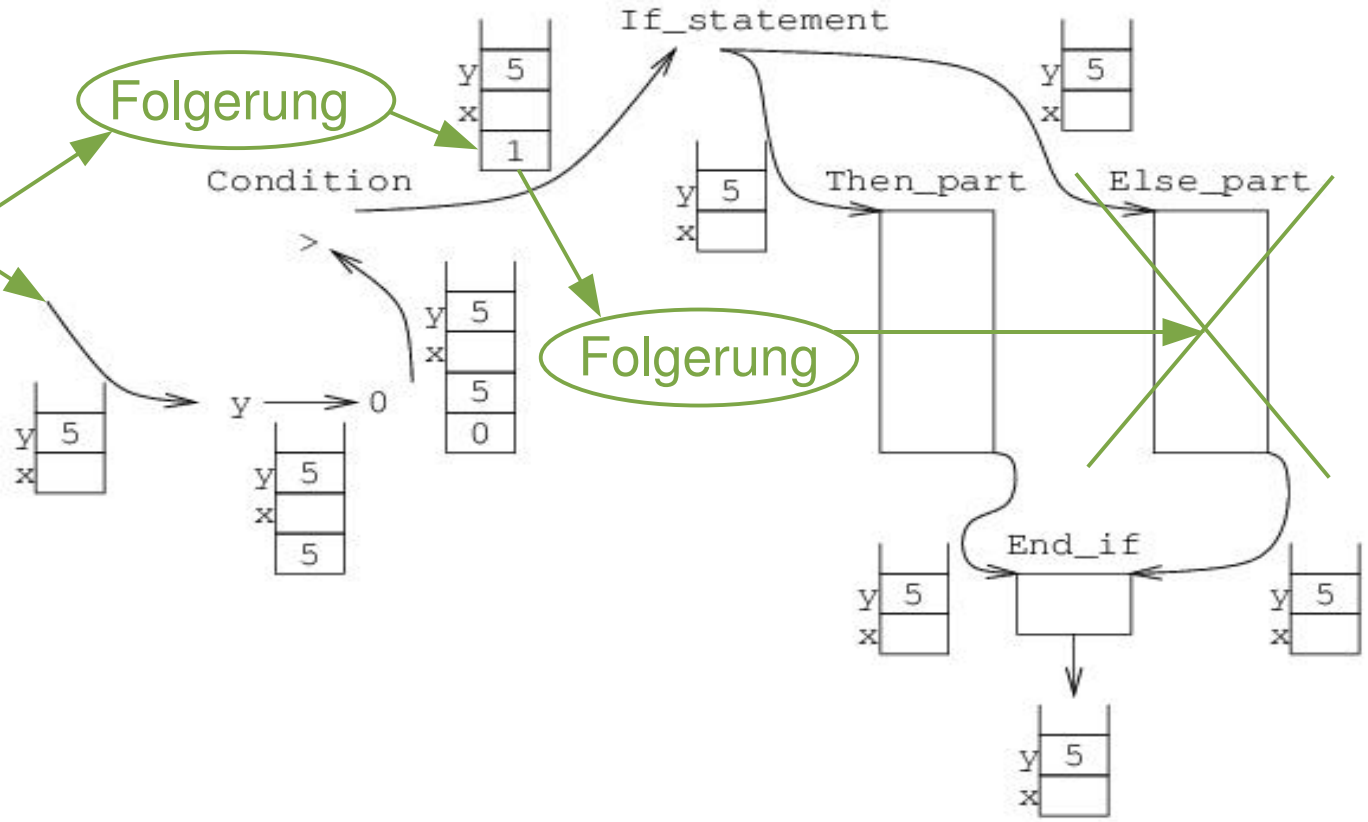
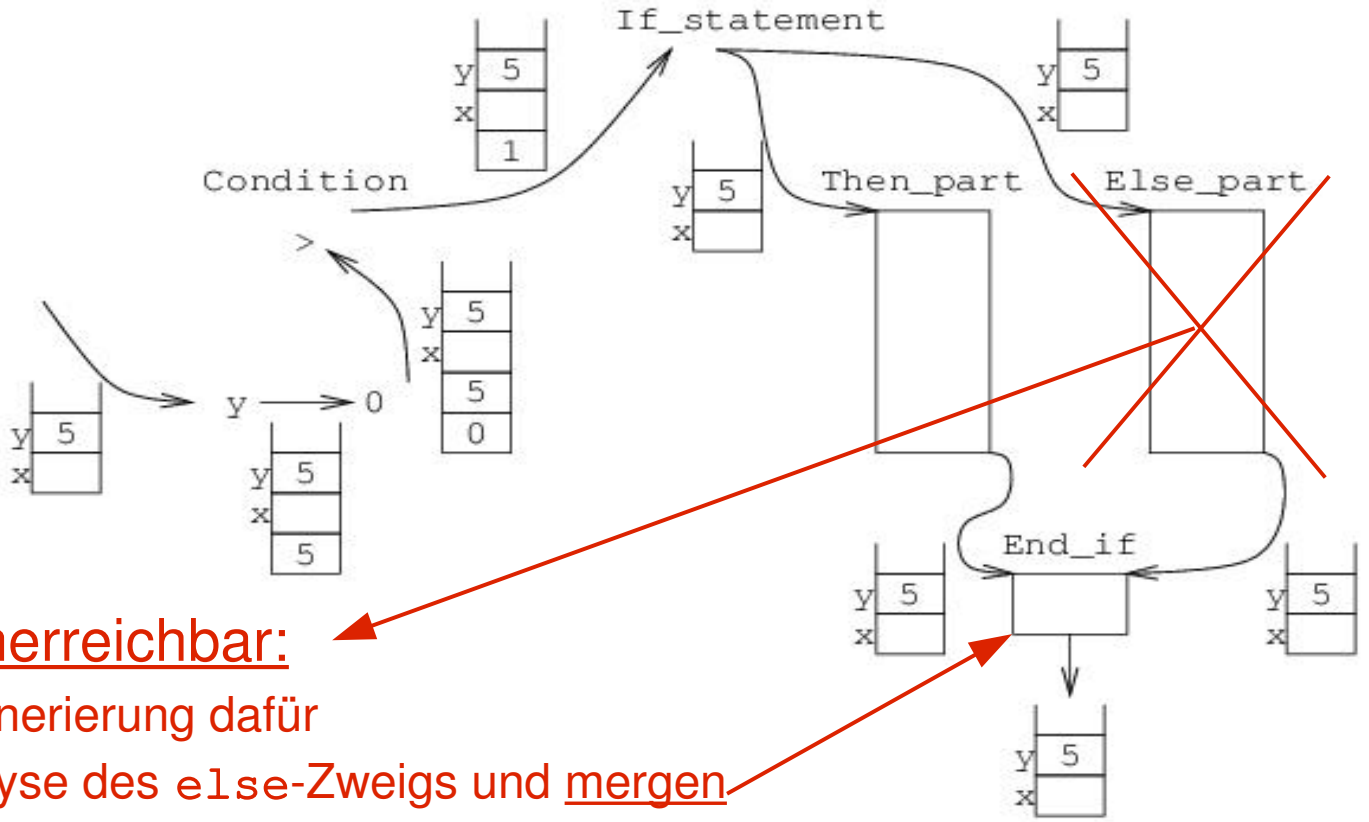


Figure 3.43 Stack representations in the control flow graph of an if-statement.

Symbolische Interpretation / Beispiel (2)

Beispiel:

```
if y>0 then ...
```



`else`-Zweig unerreichbar:

- keine Codegenerierung dafür
- trotzdem Analyse des `else`-Zweigs und mergen der Bedingungen

Figure 3.43 Stack representations in the control flow graph of an if-statement.

Code für `if`-Statement

```
FUNCTION Symbolically interpret an if statement (
    Stack representation, If node
) RETURNING a stack representation:
    SET New stack representation TO
        Symbolically interpret a condition (
            Stack representation, If node .condition
        );
    Discard top entry from New stack representation;
    RETURN Merge stack representations (
        Symbolically interpret a statement sequence (
            New stack representation, If node .then part
        ),
        Symbolically interpret a statement sequence (
            New stack representation, If node .else part
        )
    );
```

Figure 3.44 Outline of a routine `Symbolically interpret an if statement`.

Symbolische Interpretation / Arten

- einfache symbolische Interpretation
  - ein Durchlauf von Prozeduranfang bis -ende
  - folgt dem Aufbau des abstrakten Syntaxbaums
    - geeignet für "narrow" Compiler
  - nur für strukturierte Programme (kein goto) und bei bestimmten Bedingungen
- volle symbolische Interpretation
  - für jede Art von Kontrollfluss geeignet
  - braucht i.a. den gesamten abstrakten Syntaxbaum
  - Hüllenalgorithmus wg. zyklischer Abhängigkeiten

Einfache symbolische Interpretation
Beispiel: Initialisierungszustand von Variablen

- Stack auch genannt: *Property-Liste*
- Entwicklung der Property-Liste
  - Initialisierung
    - für **IN**- und **IN/OUT**-Parameter: **Init**
    - für **OUT**-Parameter: **Uninit**
  - Deklaration für Bezeichner x :
    - (x, **Init**) oder (x, **Uninit**)
  - Verzweigung:
    - kopiere Liste für jeden Zweig
    - vereinige die Kopien am *Merge*(*Join*, *Meet*)-Knoten
    - (x, **Init**) $\cup$ (x, **Uninit**) $\Rightarrow$(x, **Maybeinit**)

Entwicklung der Property-Liste (Fortsetzung)

- Zuweisung:
  - Zielvariable erhält Status **Init**, falls rechte Seite auswertbar
- Variablenzugriff:
  - Fehlermeldung, falls Status = **Uninit**
  - Warnung, falls Status = **Maybeinit**
  - Beispiel: (Annahme: Status von y ist **Uninit**)

    `if(x>=0){y=0;}` Status von y wird zu **Maybeinit**

    dann `if(x>0) {z=y;}` Warnung (hier unberechtigt)
- Programmaufruf:
  - Behandlung der aktuellen Parameterübergabe wie eine gewöhnliche Zuweisung

- Entwicklung der Property-Liste für Schleifen
  - Auswertung der Grenzausdrücke
  - Initialisierung der Schleifenvariablen
  - kopiere Property-Liste zur Loop-Exit-Liste (falls keine Iteration)
  - Rücksprung zum Schleifenbeginn muss (!) ignoriert werden

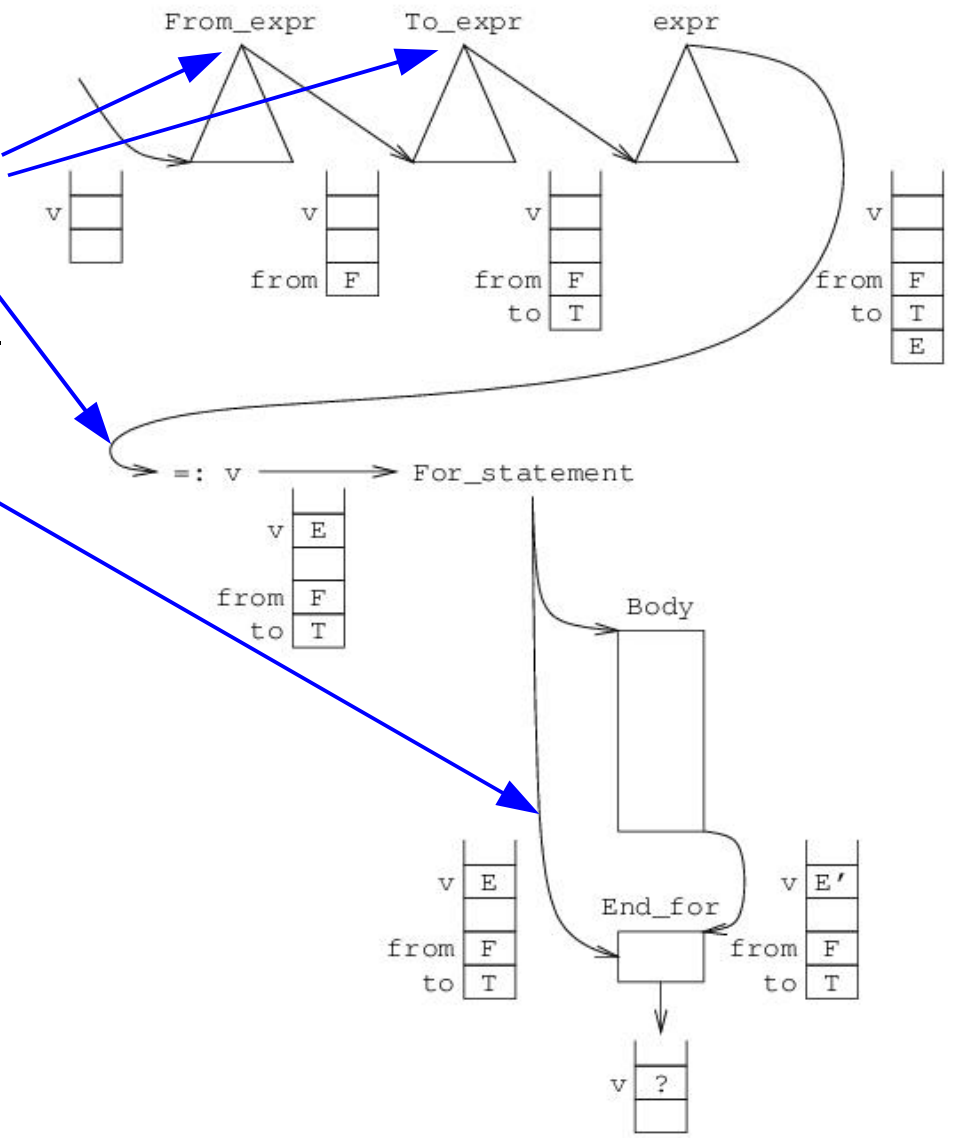


Figure 3.45 Stack representations in the control flow graph of a for-statement.

Entwicklung der Property-Liste (Fortsetzung)

- Schleifen-Austritt (“exit-loop”, “break”)
  - merge Property-Liste vor dem “break” mit der Liste am Schleifenende (Loop-Exit Liste)
  - Folgeknoten im Rumpf (nach dem “break”) hat (zunächst) leere Property-Liste
- Routinenrücksprung (“return”)
  - merge Property-Liste vor dem “return” mit der Liste am Prozedurende (Return-Liste)
  - Folgeknoten im Rumpf (nach dem “return”) hat (zunächst) leere Property-Liste
- Endknoten der Prozedur erreicht
  - Fehlermeldung, falls nicht alle OUT-Parameter einen definierten Wert haben (in der Return-Liste)
  - Warnung, falls nicht alle Variablen initialisiert sind

Beispiel: Konstantenpropagation

- Idee: propagiere statisch bekannte Werte im Programm
- Ziel:
  - erkenne und eliminiere Variablen, die nie ihren Wert ändern
  - gewinne Information über Ausdrücke wie Schleifengrenzen
- Problem: (Beispiel aus Abb. 3.46)

```
int i=0;
while (some_condition) {
  if (i>0) printf("Loop reentered: i = %d\n", i);
  i++;
}
```

  - Elimination der Ausgabe ist nicht gerechtfertigt
  - korrekte Analyse erfordert zweiten Durchlauf des Schleifenrumpfes
  - Bestimmung aller möglichen Werte von `i` führt sogar zur Nichttermination der Analyse

Einfache symbolische Interpretation

- Voraussetzungen:
  1. strukturierte Programme (Kontrollfluss mit einem Ein-/Ausgang)
  2. Werte der analysierten Eigenschaften müssen Verband bilden

     ($v_1,...,v_n$, sodass keine Aktion v_j in v_i mit $i<j$ transformiert)
  3. der Merge zweier Werte darf nicht kleiner als beide Werte sein
  4. Aktion auf v_i macht jede Aktion auf v_j ($v_i \leq v_j$) in gleicher Situation überflüssig
- Begründungen:

  1. erlaubt die unabhängige Betrachtung der Programmteile

  2.-4. erlaubt die Nichtbetrachtung des Rücksprungs in Schleifen

  2. $v_{in} \leq v_{out}$ (v_{in} Eigenschaft am Schleifeneingang, v_{out} am -ausgang)

  3. $(v_{new} = v_{out} \cup v_{in}) \Rightarrow v_{new} \geq v_{in}$

  4. daher sind Aktionen auf v_{new} überflüssig (z.B. zweiter Schleifendurchlauf)
- Beispiel: Uninit < Maybeinit < Init

  (Fehlermeldung dominiert Warnung, Warnung dominiert Nullaktion)

Volle symbolische Interpretation

- Idee: einfache symbolische Interpretation mit Hüllenbildung (Abb. 3.47)
- Beispiel: Sprungbefehle
  - eine zusätzliche Liste pro Sprungmarke (Initialisierung: leer)
  - bei Sprungbefehl auf Marke L
    - merge aktuelle Property-Liste zur Sprungliste von L
    - nächster Befehl: Fortsetzung mit leerer Property-Liste
  - bei mit L markiertem Befehl
    - merge aktuelle Property-Liste zur Sprungliste von L
    - aktueller Befehl: Fortsetzung mit neuer Sprungliste von L
  - bei Sprung zu einer textuell früheren Position
    - neuer Durchlauf der symbolischen Interpretation nötig

      ⇒ volle symbolische Interpretation
    - Analyseergebnis erst richtig, wenn Fixpunkt erreicht

      ⇒ Gefahr von Nichttermination

Volle symbolische Interpretation

Data definitions:

Stack representations, with entries for every item we are interested in.

Initializations:

1. Empty stack representations are attached to all arrows in the control flow graph residing in the threaded AST.
2. Some stack representations at strategic points are initialized in accordance with properties of the source language; for example, the stack representations of input parameters are initialized to `Initialized`.

Inference rules:

For each node type, source language dependent rules allow inferences to be made, adding information to the stack representation on the outgoing arrows based on those on the incoming arrows and the node itself, and vice versa.

Figure 3.47 Full symbolic interpretation as a closure algorithm.