Compiler Construction

Lecture 16: Semantic Analysis IV & Code Generation I

Thomas Noll

Lehrstuhl für Informatik 2
(Software Modeling and Verification)

RWTH Aachen University

`noll@cs.rwth-aachen.de`

`http://www-i2.informatik.rwth-aachen.de/i2/cc08/`

Summer semester 2008

Outline

Attribute Evaluation Methods

Given:

- (strongly) noncircular attribute grammar $\mathfrak{A} = \langle G, E, V \rangle \in AG$
- syntax tree t of G
- valuation $v : Syn_\Sigma \to V$ where $Syn_\Sigma := \{\alpha.k \mid k \text{ labelled by } a \in \Sigma, \alpha \in \text{syn}(a)\} \subseteq Var_t$

Goal: extend v to (partial) solution $v : Var_t \to V$

Methods:

1. Topological sorting of D_t:
   1. start with attribute variables which depend at most on synthesized attributes of terminals
   2. proceed by successive substitution
2. Recursive functions (for strongly noncircular AGs):
   1. for every $A \in N$ and $\alpha \in \text{syn}(A)$, define evaluation function $g_{A,\alpha}$ with the following parameters:
      - the node of t where α has to be evaluated and
      - all inherited attributes of A on which α (potentially) depends
   2. for every $\alpha \in \text{syn}(S)$, evaluate $g_{S,\alpha}(k_0)$ where k_0 denotes the root of t
3. Special cases: S-attributed grammars (`yacc`), L-attributed grammars

Outline

L-Attributed Grammars I

In an L-attributed grammar, attribute dependencies on the right-hand sides of productions are only allowed to run from left to right.

Definition 16.1 (L-attributed grammar)

Let $\mathfrak{A} = \langle G, E, V \rangle \in AG$ such that, for every $\pi \in P$ and $\beta.i = f(\ldots, \alpha.j, \ldots) \in E_\pi$ with $\beta \in Inh$ and $\alpha \in Syn$, $j < i$. Then $\mathfrak{A}$ is called an L-attributed grammar (notation: $\mathfrak{A} \in LAG$).

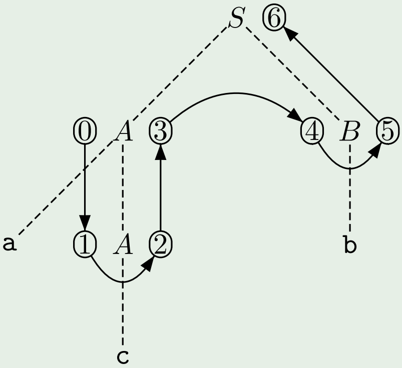
Corollary 16.2

Every $\mathfrak{A} \in LAG$ is noncircular.

Example 16.3

L-attributed grammar:

$S \to AB$	$i.1 = 0$	
	$i.2 = s.1 + 1$	
	$s.0 = s.2 + 1$	
$A \to \mathtt{a}A$	$i.2 = i.0 + 1$	
	$s.0 = s.2 + 1$	
$A \to \mathtt{c}$	$s.0 = i.0 + 1$	
$B \to \mathtt{b}$	$s.0 = i.0 + 1$	



Evaluation of L-Attributed Grammars

Observation 1: the syntax tree of an L-attributed grammar can be attributed by a depth-first, left-to-right tree traversal with two visits to each node

1. top-down: evaluation of inherited attributes
2. bottom-up: evaluation of synthesized attributes

Observation 2: visit sequence fits nicely with parsing

1. top-down: expansion steps
2. bottom-up: reduction steps

Idea: extend LL parsing to support reduction steps, and integrate attribute evaluation

$\Longrightarrow$ use $LR(0)$ items as stack alphabet
and store values of attribute variables in parsing stack

$LL(1)$ Parsing with Attribute Evaluation I

Definition 16.4 (Parsing automaton with attribute evaluation)

Let $\mathfrak{A} = \langle G, E, V \rangle \in LAG$ with $G = \langle N, \Sigma, P, S \rangle \in LL(1)$. The parsing automaton with attribute evaluation of $\mathfrak{A}$ is defined by the following components.

- Input alphabet Σ
- Pushdown alphabet $\Gamma := \bigcup_{\pi \in P \cup \{\to S\}} (LR(0)_\pi(G) \times Val_\pi)$ where
  - $LR(0)_\pi(G) := \{[A \to \delta_1 \cdot \delta_2] \mid \pi = A \to \delta_1\delta_2\}$ and
  - $Val_\pi := \{v \mid v : Out_\pi \dashrightarrow V\}$
- Configurations $\Sigma^* \times \Gamma^*$
  - initial configuration: $(w, ([\to \cdot S], v_\emptyset))$
  - final configurations: $\{(\varepsilon, ([\to S \cdot], v)) \mid v \in Val_{\to S}\}$

Definition 16.4 (continued)

- Transitions:

expand: (evaluate inherited attributes of expanded symbol)
if $x \in \text{la}(B \to \delta')$, then

$$(xw, ([A \to Y_1 \ldots Y_{i-1} \cdot B\delta], v)\gamma)$$
$$\vdash (xw, ([B \to \cdot\delta'], v')([A \to Y_1 \ldots Y_{i-1} \cdot B\delta], v)\gamma)$$

where $v' := [\beta.0 \mapsto f(v(\alpha_1.i_1), \ldots, v(\alpha_n.i_n))]$ for $\beta \in \text{inh}(B)$ and $\beta.i = f(\alpha_1.i_1, \ldots, \alpha_n.i_n) \in E_{A \to Y_1 \ldots Y_{i-1} B\delta}$

match:

$$(aw, ([A \to \delta_1 \cdot a\delta_2], v)\gamma)$$
$$\vdash (w, ([A \to \delta_1 a \cdot \delta_2], v)\gamma)$$

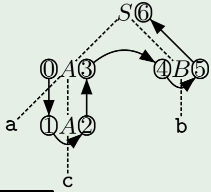
reduce: (evaluate synthesized attributes of reduced symbol)

$$(w, ([B \to \delta' \cdot], v')([A \to Y_1 \ldots Y_{i-1} \cdot B\delta], v)\gamma)$$
$$\vdash (w, ([A \to Y_1 \ldots Y_{i-1} B \cdot \delta], v'')\gamma)$$

where $v'' := v[\alpha.i \mapsto f(v'(\alpha_1.i_1), \ldots, v'(\alpha_n.i_n))]$ for $\alpha \in \text{syn}(B)$ and $\alpha.0 = f(\alpha_1.i_1, \ldots, \alpha_n.i_n) \in E_{B \to \delta'}$

Example 16.5 (cf. Example 16.3)

$S \to AB$
$A \to \mathtt{a}A$
$A \to \mathtt{c}$
$B \to \mathtt{b}$



acb

$[\to \cdot S]$	$-$

$\vdash$ acb

$[\to \cdot S]$	$-$
$[S \to \cdot AB]$	$-$

$\vdash$ acb

$[\to \cdot S]$	$-$
$[S \to \cdot AB]$	$-$
$[A \to \cdot \mathtt{a}A]$	$i.0 = 0$

$\vdash$ cb

$[\to \cdot S]$	$-$
$[S \to \cdot AB]$	$-$
$[A \to \mathtt{a} \cdot A]$	$i.0 = 0$

$\vdash$ cb

$[\to \cdot S]$	$-$
$[S \to \cdot AB]$	$-$
$[A \to \mathtt{a} \cdot A]$	$i.0 = 0$
$[A \to \cdot \mathtt{c}]$	$i.0 = 1$

$\vdash$ b

$[\to \cdot S]$	$-$
$[S \to \cdot AB]$	$-$
$[A \to \mathtt{a} \cdot A]$	$i.0 = 0$
$[A \to \mathtt{c} \cdot]$	$i.0 = 1$

$\vdash$ b

$[\to \cdot S]$	$-$
$[S \to \cdot AB]$	$-$
$[A \to \mathtt{a}A \cdot]$	$i.0 = 0, s.2 = 2$

$\vdash$ b

$[\to \cdot S]$	$-$
$[S \to A \cdot B]$	$s.1 = 3$

$\vdash$ b

$[\to \cdot S]$	$-$
$[S \to A \cdot B]$	$s.1 = 3$
$[B \to \cdot \mathtt{b}]$	$i.0 = 4$

$\vdash \varepsilon$

$[\to \cdot S]$	$-$
$[S \to A \cdot B]$	$s.1 = 3$
$[B \to \mathtt{b} \cdot]$	$i.0 = 4$

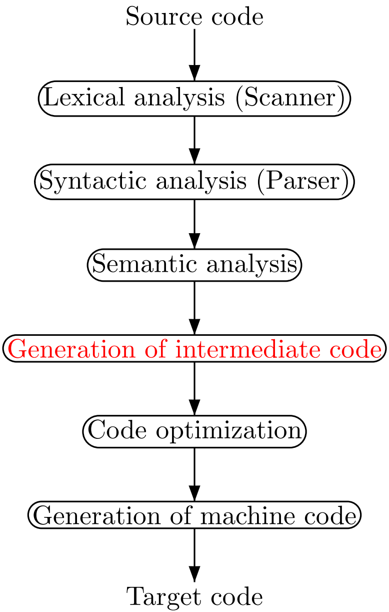
$\vdash \varepsilon$

$[\to \cdot S]$	$-$
$[S \to AB \cdot]$	$s.1 = 3, s.2 = 5$

$\vdash \varepsilon$

$[\to S \cdot]$	$s.1 = 6$

Conceptual Structure of a Compiler



Outline

Modularization of Code Generation I

Splitting of code generation for programming language PL:

$$\mathrm{PL} \xrightarrow{\mathrm{trans}} \mathrm{IC} \xrightarrow{\mathrm{code}} \mathrm{MC}$$

Frontend: trans generates machine-independent intermediate code (IC) for abstract (stack) machine

Backend: code generates actual machine code (MC)

Advantages: IC machine independent $\Longrightarrow$

Portability: much easier to write IC compiler/interpreter for a new machine (as opposed to rewriting the whole compiler)

Fast compiler implementation: generating IC much easier than generating MC

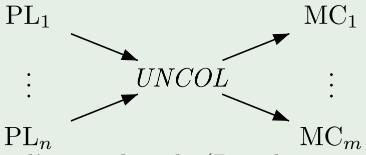
Code size: IC programs usually smaller than corresponding MC programs

Code optimization: division into machine-independent and machine-dependent parts

Modularization of Code Generation II

Example 16.6

1. UNiversal Computer-Oriented Language (UNCOL; ≈ 1960; `http://en.wikipedia.org/wiki/UNCOL`):
   universal intermediate language for compilers (never fully specified or implemented; too ambitious)



only $n + m$ translations (in place of $n \cdot m$)

2. Pascal's pseudocode (P-code; ≈ 1975; `http://en.wikipedia.org/wiki/P-Code_machine`)
3. The Amsterdam Compiler Kit (TACK; since 1980; `http://tack.sourceforge.net/`)
4. Java Virtual Machine (JVM; Sun; `http://en.wikipedia.org/wiki/Java_Virtual_Machine`)
5. Common Intermediate Language (CIL; Microsoft; `http://en.wikipedia.org/wiki/Common_Intermediate_Language`)

Language Structures I

Structures in imperative programming languages:
(object-oriented, declarative [functional/logic]: see special courses)

- Basic data types and basic operations
- Static and dynamic data structures
- Expressions and assignments
- Control structures (sequences, branching statements, loops, ...)
- Procedures and functions
- Modularity: blocks, modules, and classes

Use of procedures and blocks:

- FORTRAN: non-recursive and non-nested procedures
  $\Longrightarrow$ static memory management (memory requirement determined at compile time)
- C: recursive and non-nested procedures
  $\Longrightarrow$ dynamic memory management using runtime stack (memory requirement only known at runtime), no static links
- Algol-like languages (Pascal, Modula): recursive and nested procedures
  $\Longrightarrow$ dynamic memory management using runtime stack with static links

Language Structures II

Structures in machine code: (von Neumann/SISD)

Memory hierarchy: accumulators, registers, cache, main memory, background storage

Instruction types: arithmetic/Boolean/... operation, test/jump instruction, transfer instruction, I/O instruction, ...

Address modes: direct/indirect, absolute/relative, ...

Architectures: RISC (few [fast but simple] instructions, many registers), CISC (many [complex but slow] instructions, few registers)

Structures in intermediate code:

- Data types and operations like PL
- Data stack with basic operations
- Jumping instructions for control structures
- Runtime stack for blocks, procedures, and static data structures
- Heap for dynamic data structures

Outline

Structures of EPL:

- Only integer and Boolean values
- Arithmetic and Boolean expressions with strict and non-strict semantics
- Control structures: sequence, branching, iteration
- Nested blocks and recursive procedures with local and global variables
  ($\Longrightarrow$ dynamic memory management using runtime stack with static links)
- Procedure parameters and data structures later

Definition 16.7 (Syntax of EPL)

The syntax of EPL is defined as follows:

$\mathbb{Z}$:	z	(* z is an integer *)
Ide :	I	(* I is an identifier *)
$AExp$:	$A ::= z \mid I \mid A_1$ `+` $A_2 \mid \ldots$	
$BExp$:	$B ::= A_1$ `<` $A_2 \mid$ `not` $B \mid B_1$ `and` $B_2 \mid B_1$ `or` B_2	
Cmd :	$C ::= I$ `:=` $A \mid C_1$`;`$C_2 \mid$ `if` B `then` C_1 `else` $C_2 \mid$ `while` B `do` $C \mid I$`()`	
Dcl :	$D ::= D_C\ D_V\ D_P$ $D_C ::= \varepsilon \mid$ `const` I_1 `:=` z_1`,` $\ldots$ `,`I_n `:=` z_n`;` $D_V ::= \varepsilon \mid$ `var` I_1`,` $\ldots$ `,`I_n`;` $D_P ::= \varepsilon \mid$ `proc` I_1`;`K_1`;` $\ldots$ `;`I_n`;`K_n`;`	
$Block$:	$K ::= D\ C$	
Pgm :	$P ::=$ `in/out` I_1`,` $\ldots$ `,`I_n`;`K`.`	

Static Semantics of EPL I

- All identifiers in a declaration D have to be different.
- Every identifier occurring in the command C of a block $D\ C$ must be declared
  - in D or
  - in the declaration list of a surrounding block.
- Multiple declarations of an identifier in different blocks are possible. Each usage in a command C refers to the "innermost" declaration.
- Static scoping: the usage of an identifier in the body of a called procedure refers to its declaration environment (and not to its calling environment).

Static Semantics of EPL II

Example 16.8

```
in/out x;
  const c = 10;
  var y;
  proc P;
    var y, z;
    proc Q;
      var x, z;
      [... z := 1; P() ...]
    [... P() ... R() ...]
  proc R;
    [... P() ...]
  [... x := 0; P() ...] .
```

- "Innermost" principle
- Static scoping: body of `P` can refer to `x`, `y`, `z`
- Later declaration: call of `R` in `P` followed by declaration (in Pascal: **`forward`** declarations for one-pass compilation)