Compiler Construction

Lecture 18: Semantic Analysis IV (L-Attributed Grammars)/ Code Generation I (Introduction)

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/cc10/

Winter semester 2010/11

Outline

Attribute Evaluation Methods

Given:
- 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 (Syn_Σ)
   2. proceed by successive substitution
2. Recursive functions (for strongly noncircular AGs; later):
   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

Attribute Evaluation by Topological Sorting

Algorithm (Evaluation by topological sorting)

Input: *noncircular* $\mathfrak{A} = \langle G, E, V \rangle \in AG$, *syntax tree* t *of* G, *valuation* $v : Syn_\Sigma \to V$

Procedure:

1. *let* $Var := Var_t \setminus Syn_\Sigma$ (* *attributes to be evaluated* *)
2. *while* $Var \neq \emptyset$ *do*
   1. *let* $x \in Var$ *such that* $\{y \in Var \mid y \to_t x\} = \emptyset$
   2. *let* $x = f(x_1, \ldots, x_n) \in E_t$
   3. *let* $v(x) := f(v(x_1), \ldots, v(x_n))$
   4. *let* $Var := Var \setminus \{x\}$

Output: *solution* $v : Var_t \to V$

Remark: noncircularity guarantees that in step 2.1 at least one such x is available

Example

see Examples 15.1 and 15.2 (Knuth's binary numbers)

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 18.1 (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: $\mathfrak{A} \in LAG$).

Remark: note that no restrictions are imposed for $\beta \in Syn$ (for $i = 0$) or $\alpha \in Inh$ (for $j = 0$). Thus, in an L-attributed grammar,

- synthesized attributes of the left-hand side can depend on any outer variable and
- every inner variable can depend on any inherited attribute of the left-hand side.

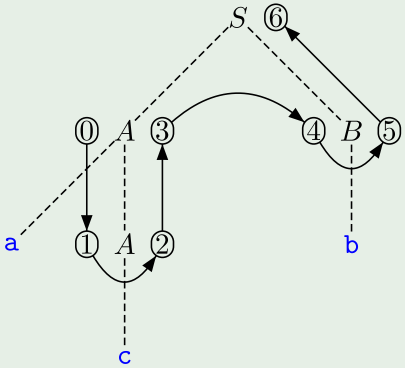
Corollary 18.2

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

Example 18.3

L-attributed grammar:

$$
\begin{array}{lrcl}
S \rightarrow AB & \color{red}{i.1} & \color{red}{=} & \color{red}{0} \\
 & \color{red}{i.2} & \color{red}{=} & \color{red}{s.1 + 1} \\
 & s.0 & = & s.2 + 1 \\
A \rightarrow \mathtt{a}A & \color{red}{i.2} & \color{red}{=} & \color{red}{i.0 + 1} \\
 & s.0 & = & s.2 + 1 \\
A \rightarrow \mathtt{c} & s.0 & = & i.0 + 1 \\
B \rightarrow \mathtt{b} & s.0 & = & i.0 + 1
\end{array}
$$



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 recursive-descent parser
- add variables and operations for attribute evaluation

Recursive-Descent Parsing and Evaluation I

Ingredients:
- variable `token` for current token
- function `next()` for invoking the scanner
- procedure `print(i)` for displaying the leftmost analysis (or errors)

Method: to every $A \in N$ we assign a procedure

$$\texttt{A(in: } \mathrm{inh}(A), \texttt{ out: } \mathrm{syn}(A)\texttt{)}$$

which
- declares local variables for synthesized attributes on right-hand sides,
- tests `token` with regard to the lookahead sets of the A-productions,
- prints the corresponding rule number and
- evaluates the corresponding right-hand side as follows:
  - for $a \in \Sigma$: check `token`; call `next()`
  - for $A \in N$: call `A` with appropriate parameters

Example 18.4 (cf. Example 18.3)

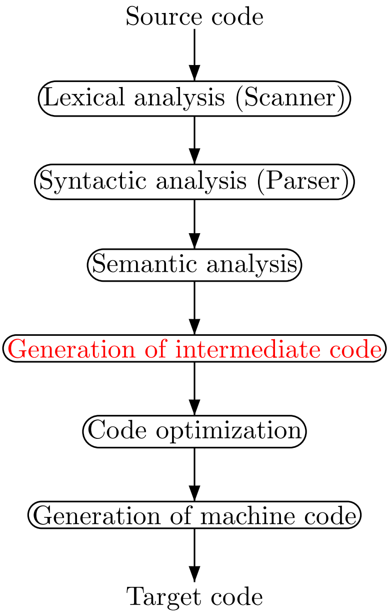
```
proc main();
  token := next(); S()
proc S();   (* S → A B *)
  if token in {'a','c'} then
    print(1); A(); B()
  else print(error); stop fi
proc A();  (* A → a A | c *)
  if token = 'a' then
    print(2); token := next(); A()
  elsif token = 'c' then
    print(3); token := next()
  else print(error); stop fi
proc B();   (* B → b *)
  if token = 'b' then
    print(4); token := next()
  else print(error); stop fi
```

Example 18.5 (cf. Example 18.3)

```
proc main(); var s;
  token := next(); S(s); print(s)
proc S(out s0); var s1,s2;   (* S → A B *)
  if token in {'a','c'} then
    print(1); A(0,s1); B(s1 + 1,s2); s0 := s2 + 1
  else print(error); stop fi
proc A(in i0,out s0); var s2;   (* A → a A | c *)
  if token = 'a' then
    print(2); token := next(); A(i0 + 1,s2); s0 := s2 + 1
  elsif token = 'c' then
    print(3); token := next(); s0 := i0 + 1
  else print(error); stop fi
proc B(in i0,out s0);   (* B → b *)
  if token = 'b' then
    print(4); token := next(); s0 := i0 + 1
  else print(error); stop fi
```

Outline

Conceptual Structure of a Compiler



Modularization of Code Generation I

Splitting of code generation for programming language PL:

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

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

Backend: code generates actual machine code (MC)

Advantages: IC machine independent $\implies$

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

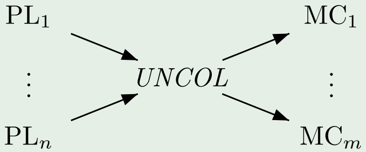
Fast compiler implementation: generating IC much easier than generating MC

Code size: IC programs usually smaller than corresponding MC programs

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

Example 18.6

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; ≈ 1980; `http://tack.sourceforge.net/`)
4. Java Virtual Machine (JVM; Sun; ≈ 1996; `http://en.wikipedia.org/wiki/Java_Virtual_Machine`)
5. Common Intermediate Language (CIL; Microsoft .NET; ≈ 2002; `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

Syntax of EPL

Definition 18.7 (Syntax of EPL)

The syntax of EPL is defined as follows:

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

Outline

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 18.8

```
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)

Dynamic Semantics of EPL

(omitting the details)

- To “run” a program, execute the main block in the state which is given by the input values
- Effect of statement = modification of state
  - assignment I `:=` A: update of I by value of A
  - composition C_1`;`C_2: sequential execution
  - branching `if` B `then` C_1 `else` C_2: test of B, followed by jump to respective brnach
  - iteration `while` B `do` C: execution of C as long as B is true
  - call I`()`: transfer control to body of I and return to subsequent statement afterwards
- Consequently, an EPL program $P =$ `in/out` I_1`,` $\ldots$ `,`I_n`;`K`.` $\in Pgm$ has as semantics a function

$$\mathfrak{M}[\![P]\!] : \mathbb{Z}^n \dashrightarrow \mathbb{Z}^n$$