Lecture 14
Compiler Frameworks and Compiler Generators

A (non-exhaustive) survey
with a focus on open-source frameworks

Compiler Generators or CWS - Compiler Writing Systems

- A Compiler Generator or CWS is a program which, given a description of the source language and a description of the target language (object code description), produces a compiler for the source language as output.
- Different generators within CWS generate different phases of the compiler.

Compiler Generator Formalisms and generated Compiler Modules

- Regular expressions for L
- BNF grammar for L
- Attribute grammar or Denotational semantics for the language L
- Optimization specification
- Code generator specification for internal form I and machine A

Syntax-Based Generators

- LEX or FLEX – generates lexical analysers
- YACC or BISON – generates parsers
  - Compiler components that are not generated: semantic analysis and intermediate code gen.; the optimisation phase; code generators
- Note: YACC produces parsers which are bad at error management

Semantics-Based Generators
RML - A Compiler Generation System and Specification Language from Natural Semantics/Structured Operational Semantics

- **Goals**
  - Efficient code — comparable to hand-written compilers
  - Simplicity — simple to learn and use
  - Compatibility with "typical natural semantics/operational semantics" and with Standard ML

- **Properties**
  - Deterministic
  - Separation of input and output arguments/results
  - Statically strongly typed
  - Polymorphic type inference
  - Efficient compilation of pattern-matching
  - www.ida.liu.se/pelab/~rml

**Example Use: Generating an Interpreter Implemented in C, using rml2C**

<table>
<thead>
<tr>
<th>Formalism</th>
<th>Generator tool</th>
<th>Compiler phase</th>
<th>Program representation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Regular expressions</td>
<td>Lex</td>
<td>...</td>
<td>Text</td>
</tr>
<tr>
<td>BNF grammar</td>
<td>Yacc</td>
<td>...</td>
<td>Token sequence</td>
</tr>
<tr>
<td>Natural semantics in RML</td>
<td>rml2c</td>
<td>...</td>
<td>Abstract syntax</td>
</tr>
</tbody>
</table>

**RML Syntax**

- Goal: Eliminate plethora of special symbols usually found in Natural Semantics/Operational Semantics specifications
  - Software engineering viewpoint: identifiers are more readable in large specifications
- A Natural semantics/Operational semantics rule:
  
  \[
  \begin{align*}
  &H_1 \mid \ldots \mid H_n \mid -
  
  \begin{align*}
  &T_1 : R_1 \mid \ldots \mid T_n : R_n
  
  \text{if } \text{<cond>}
  
  \begin{align*}
  &H \mid -
  
  \begin{align*}
  &T : R
  
  \end{align*}
  
  \end{align*}
  \end{align*}
  \]

**Example Use: Generating a Compiler Implemented in C**

**Example: the Exp1 Expression Language**

- Typical expressions:
  - \(12 + 5 \times 3\)
  - \(-5 \times (10 - 4)\)

**Evaluator for Exp1**

- rule eval(s1) => v1 & eval(s2) => v2 & int_add(v1,v2) => v3
  
  eval(PPLUSop(s1,s2)) => v3

- rule eval(s1) => v1 & eval(s2) => v2 & int_sub(v1,v2) => v3
  
  eval(SPLUSop(s1,s2)) => v3

**Example: the Exp1 Expression Language**

- Abstract syntax tree of \(12 + 5 \times 3\)

- AXIOM eval(INTC) => int
- Evaluation of an integer constant is the integer itself:
  
  \[
  \text{eval(INTC)}(\text{eval}(\text{val})) \Rightarrow \text{val}
  \]

- Evaluation of an addition node
  
  **PLUSop** is v3, if v3 is the result of adding the evaluated results of its children s1 and s2
  
  **Subtraction, multiplication, division operators have similar specifications.**
  
  (we have removed division below)
Arithmetic Expression Translation Semantics

Simple Lookup in Environments Represented as Linked Lists

```
relation lookup: (Env,Ident) => Value =
("lookup returns the value associated with an identifier.
If no association is present, lookup will fail.
Identifier id is found in the first pair of the list, and value is returned."

rule id = id2
--------------------
lookup((id2,value)) => value

("id is not found in the first pair of the list,
and lookup will recursively search the rest of the list.
If found, value is returned.")

rule not id=id2 & lookup(rest, id) => value
--------------------------
lookup((id2,_) :: rest, id) => value
```

Translational Semantics of the PAM language
– Abstract Syntax to Machine Code

PAM language example:
beginning of RML Relation trans_expr:
relation trans_expr: Exp => Mcode list =
-----------------------
rule trans_expr(e1) => cod1 &
-----------------------
rule trans_expr(e2) => cod2 &
-----------------------
rule trans_expr(BINARY(e1,binop,e2)) => cod3
-----------------------
```
MB(opcode, operand2)
-----------------------
cod2
-----------------------
cod1
-----------------------
```

Arithmetic Expression Translation Semantics

Simple Machine Instruction Set:

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>LOAD</td>
<td>Load accumulator</td>
</tr>
<tr>
<td>ADD</td>
<td>Add</td>
</tr>
<tr>
<td>SUB</td>
<td>Subtract</td>
</tr>
<tr>
<td>MUL</td>
<td>Multiply</td>
</tr>
<tr>
<td>DIV</td>
<td>Divide</td>
</tr>
<tr>
<td>GET</td>
<td>Input a value</td>
</tr>
<tr>
<td>PUT</td>
<td>Output a value</td>
</tr>
<tr>
<td>J</td>
<td>Jump</td>
</tr>
<tr>
<td>JN</td>
<td>Jump on negative</td>
</tr>
<tr>
<td>JP</td>
<td>Jump on positive</td>
</tr>
<tr>
<td>JNP</td>
<td>Jump on negative or positive</td>
</tr>
<tr>
<td>LAB</td>
<td>Label (no operation)</td>
</tr>
<tr>
<td>HALT</td>
<td>Halt execution</td>
</tr>
</tbody>
</table>

Example PAM program:
```
read x,y;
while x<>99 do
  ans := (x+1) - (y/2);
write ans;
read x,y;
end
```

Translated machine code
```
GET x, STD T2
GET y, STD T1
SUB 99, PUT AMX
GET x, STD XM
SUB 0, PUT XM
ADD T, J LT
LOAD y, STD LAB
LOAD y, STD HALT
DW 2
MGETI (x), MSTO (T2)
MGETI (y), MLOAD (T1)
MLOAD (y), MSTO (x)
MBMUL (N(9)), MPSTO (x)
MBMUL (N(2)), MPSTO (T2)
MJ (JPZ, T2)
MB (MADD, N(1)), MJMP (L1)
MLOAD (x), MSTO (T1)
MB (MSUB, T2), MPUT (I(ans))
MLABEL (L2)
MJ (MADD, T2), MPUT (I(ans))
MLABEL (L1)
MB (MSUB, N(99)), MLABEL (L2)
MJMP (L1)
MGET (I(y)), MLOAD (T1)
MJ (MADD, T2), MPUT (I(ans))
MLABEL (L2)
MJMP (L1)
MGET (I(x)), MLOAD (T1)
MB (MSUB, T2), MPUT (I(ans))
MLABEL (L1)
MJMP (L2)
```

Begin of RML Relation trans_expr:
```
relation trans_expr: Exp => Mcode list =
-----------------------
axiom trans_expr(INTEG(v)) => [MLOAD( N(v))]
axiom trans_expr(IDENT(id)) => [MLOAD( I(id))]
-----------------------
```

PAM Example Translation

```
PAM program:
read x,y;
while x<>99 do
  ans := (x+1) - (y/2);
write ans;
read x,y;
end

Translational Relations

Simple Machine Instruction Set:
```
...
Some Applications of RML

- Small functional language with call-by-name semantics (mini-Freja, a subset of Haskell)
- Almost full Pascal with some C features (Petrol)
- Mini-ML including type inference
- Specification of Full Java 1.2
- Specification of Modelica 2.0

Additional Performance Comparison

<table>
<thead>
<tr>
<th>#primes</th>
<th>RML</th>
<th>SICStus</th>
<th>SWI</th>
<th>Maude/MODS Tool</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>0.00</td>
<td>0.05</td>
<td>0.00</td>
<td>2.92</td>
</tr>
<tr>
<td>10</td>
<td>0.00</td>
<td>0.10</td>
<td>0.03</td>
<td>5.60</td>
</tr>
<tr>
<td>50</td>
<td>1.25</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>200</td>
<td>16.32</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

Execution time in seconds. The – sign represents out of memory.

Some RML and Semantics References

- Web page, open source: www.ida.liu.se/~rml

Some Attribute-Grammar Based Tools

- JASTADD – OO Attribute grammars
- Ordered Attribute Grammars.

LCC (Little C Compiler)

- Not really a Generator but uses IBURG
  - Dragon-book style C compiler implementation in C
  - Very small (20K Loc), well documented, well tested, widely used
  - Open source: http://www.cs.princeton.edu/software/lcc
  - Textbook A retargetable C compiler [Fraser, Hanson 1995] contains complete source code
  - One-pass compiler, fast
  - C frontend (hand-crafted scanner and recursive descent parser) with own C preprocessor
  - Low-level IR
    - Basic-block graph containing DAGs of quadruples
    - No AST
    - Interface to IBURG code generator generator
    - Example code generators for MIPS, SPARC, Alpha, x86 processors
    - Tree pattern matching + dynamic programming
    - Few optimizations only
    - Local common subexpr. elimination, constant folding
    - Good choice for source-to-target compiling if a prototype is needed soon
GCC 4.x

Not a Generator – but wide-spread usage

- GNU Compiler Collection (earlier: GNU C Compiler)
- Compilers for C, C++, Fortran, Java, Objective-C, Ada ...
  + sometimes with own extensions, e.g. GNU-C
- Open-source, developed since 1985
- Very large
- 3 IR formats (all language independent)
  - GENERIC: tree representation for whole function (also statements)
  - GRIMPLE: simple version of GENERIC for optimizations
    based on trees but expressions in quadrant form.
  - RTL: (Register Transfer Language, low-level, Lisp-like) (the traditional GCC-IR)
    only word-sized data types, stack explicit; statement scope
- Many optimizations
- Many target architectures
- Version 4.x (since ~2004) has strong support for retargetable code generation
  - Machine description in .md file
  - Reservation tables for instruction scheduler generation

Open64 / ORC Open Research Compiler Framework

- Based on SGI Pro-64 Compiler for MIPS processor, written in C++,
  went open source in 2000
- Several tracks of development (Open64, ORC, ...)
- For Intel Itanium (IA-64) and x86 (IA-32) processors.
  Also retargeted to x86-64, Ceva DSP, Tensilica, XScale, ARM ...
- "simple to retarget" (?)
- Languages: C, C++, Fortran95 (uses GCC as frontend),
  OpenMP and UPC (for parallel programming)
- Industrial strength, with contributions from Intel, Pathscale, ...
- Open source: www.open64.net, ipf-orc.sourceforge.net
- 6-layer IR:
  - WHIRL (VH, H, M, L, VL) – 5 levels of abstraction
    - All levels semantically equivalent
  - Each level a lower level subset of the higher form
  - and target-specific very low-level CGIR

Open64 / ORC Open Research Compiler

- Multi-level IR
  + translation by lowering
  - Analysis / Optimization engines can work on
    the most appropriate level of abstraction
  - Clean separation of compiler phases
  - Framework gets larger and slower
- Many optimizations, many third-party contributed components

CoSy

A commercial compiler framework
Primarily focused on backends

www.ace.nl
A CoSy Compiler with Repository-Architecture

Parser

Optimizer

Lexer

CodeGen

Transformation

Common Intermediate representation repository

"Backboard architecture"  

Composite Engines in CoSy

- Built from simple engines or from other composite engines by combining engines in interaction schemes (Loop, Pipeline, Fork, Parallel, Speculative, ...)
- Described in EDL (Engine Description Language)
- View defined by the joint effect of constituent engines
- A compiler is nothing more than a large composite engine

Example for CoSy EDL (Engine Description Language)

- Component classes (engine class)
- Component instances (engines)
- Basic components are implemented in C
- Interaction schemes (cf. skeletons) form complex connectors
  - SEQUENTIAL
  - PIPELINE
  - DATAPARALLEL
  - SPECULATIVE
- EDL can embed automatically
  - Single-call-components into pipes
  - p<> means a stream of p-items
  - EDL can map their protocols to each other (p vs p<>)

Evaluation of CoSy

- The outer call layers of the compiler are generated from view description specifications
  - Adapter, coordination, communication, encapsulation
  - Sequential and parallel implementation can be exchanged
  - There is also a non-commercial prototype


- Access layer to the repository must be efficient (solved by generation of macros)
- Because of views, a CoSy-compiler is very simply extensible
  - That's why it is expensive
  - Reconfiguration of a compiler within an hour
More Frameworks

- LLVM (Univ. of Illinois at Urbana Champaign)
  - llvm.org
  - "Low-level virtual machine", IR
  - compiles to several target platforms: x86, Itanium, ARM, Alpha, SPARC
  - Open source

- Cetus
  - http://cobweb.ecn.purdue.edu/ParaMount/Cetus/
  - C/C++ source-to-source compiler written in Java.
  - Open source

- Tools and generators
  - TXL source-to-source transformation system
  - ANTLR frontend generator

More frameworks…

- Some influential frameworks of the 1990s
  - SUIF Stanford university intermediate format, suif.stanford.edu
  - Trimaran (for instruction-level parallel processors) www.trimaran.org
  - Polaris (Fortran) UIUC
  - Jikes RVM (Java) IBM
  - Soot (Java)
  - GMD Toolbox / Cocolab Cocktail™ compiler generation tool suite
  - and many others …

- And many more for the embedded domain …

The End (?)

"Now this is not the end. It is not even the beginning of the end. But it is, perhaps, the end of the beginning."
- W. Churchill

- Do you like compiler technology? Learn more?
  - TDDC86 Compiler optimizations and code generation 6hp
  - TDDC18 Component-based software 4.5hp
  - Thesis project (Exjobb) at PELAB, 30 hp

R.U. = (THANK YOU) (MERRY CHRISTMAS) (HAPPY NEW YEAR)