

CSc 553

Principles of Compilation

X11 : 11 Cocktail and BEG

Department of Computer Science
University of Arizona

collberg@gmail.com

Copyright © 2011 Christian Collberg

Code Generation with BEG

BEG and Cocktail I

- We'll talk about a code generator generating system called BEG.
- BEG is part of a larger compiler construction toolset called Cocktail. Cocktail has tools that construct parsers, scanners, semantic analysers, intermediate code generators and machine code generators.
- Most of the Cocktail tools are now commercial products, but there are still some free early versions around on the net.
- BEG is a program which takes as input a specification consisting of **code generation** rules, a description of the target architecture, and a description of the intermediate representation.

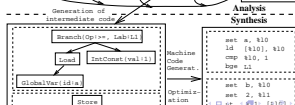
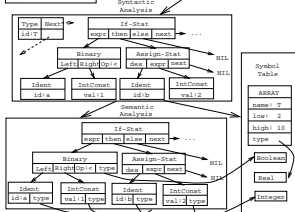
BEG and Cocktail II

- Based on the machine description BEG produces a code generator which operates by pattern-matching on the intermediate representation.
- One can choose between several (at least two...) types of register allocators.
- The code generator constructed by BEG takes as its input a sequence of expression trees. Output can be just about anything; we will produce textual assembly code.
- The code generator takes care of (almost) all aspects of code generation: instruction selection, register allocation, register assignment, and register spilling.
- The next slide shows the overall structure of a compiler built by Cocktail.

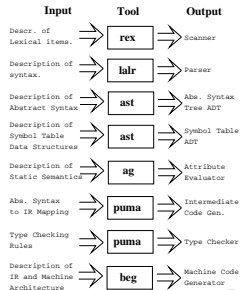
```

TYPE T =
  ARRAY[2..10] OF REAL
...
Lexical Analysis
IF Ident = T, ARRAY, [...]
  IF Ident = a,
    IntConst = 1, THEN Ident = b
  | =, IntConst = 2, END, ...
  ...
IF a = 1 THEN b = 2 END
...

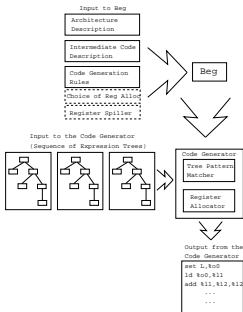
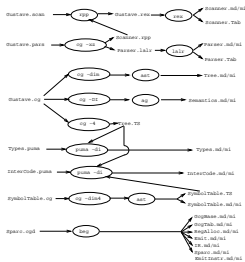
```



Cocktail Tools



Cocktail Generated Files



Integer (value: INTEGER) -> IntExp;
 GlobalVar (id: Identifier) -> IntExp;
 Load IntExp -> IntExp;
 Store IntExp * IntExp;
 Plus IntExp + IntExp -> IntExp;



- The + (*)-sign makes the operator (non-)commutative (Plus is commutative since $a + b \equiv b + a$).
- Parentheses indicate input attributes, data passed from the front end to the code generator.

Navigation icons: back, forward, search, etc.

Navigation icons: back, forward, search, etc.

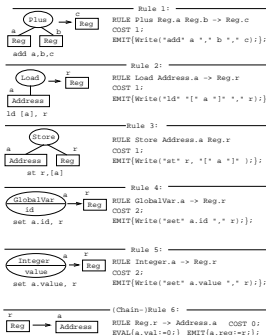


Navigation icons: back, forward, search, etc.

Navigation icons: back, forward, search, etc.

There are several alternative ways of thinking about the actions taken by BEG during code generation:

- Find a set of patterns that covers the input tree, such that the total cost is minimized.
- Build a tree (identical to the input tree) out of the available patterns, such that the total cost of the new tree is minimized.
- Apply the patterns to the input tree, replacing the matched part of the tree with the right hand side of the pattern, until the tree is consumed. Choose the set of patterns with minimal cost.



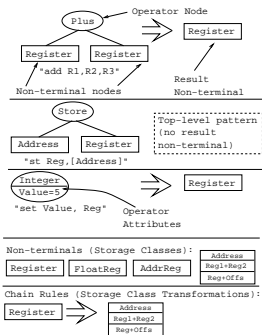
Navigation icons: back, forward, search, etc.

Navigation icons: back, forward, search, etc.

- These simple rules all contain three parts: a **pattern**, a **cost**, and an **emit**-part. The **pattern**-part describes a sub-tree which the rule matches, the **emit**-part describes the instructions to be generated by the rule, and the **cost**-part describes the cost of executing these instructions.

Explanations:

Rule 1, for example, states that on the SPARC there is an add-instruction which adds two registers into a third, and which executes in one (1) machine cycle.

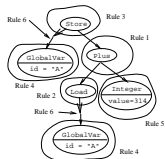


- Rule 2** says that we can load the value at a given address into a register using the ld-instruction. Address is a SPARC addressing mode which has several different forms.
- Rule 5** shows how we can load an integer value into a register using the set pseudo-operation.
- Rule 6** is known as a chain-rule. It generates no code but simply states how we may transform a value from one form to another. In this case the rule states how we can transform a register into the addressing mode Address. The Modula-2-code in the **EVAL**-part is executed during the code generator's cover phase, whereas the code in the **EMIT**-part is executed during the output phase.

Beg Tree Patterns II

- A Beg pattern is normally a sub-tree where the leaves are non-terminals.
- A non-terminal represents a *storage class* on the machine, i.e. a register or some other addressing mode.
- Most patterns have a *result non-terminal*, which describes the type of storage class in which the instruction returns its result.
- Top-level patterns are distinguished by the fact that they do not have a result non-terminal. They therefore only match the root of an input expression tree.

- A chain rule transforms a value from one storage class to another. Often this can be done without generating any code. For example, if one instruction returns its result in a register, and the next instruction requires its argument in an address (e.g. a Sparc addressing mode which is either a register, the sum of two registers, or the sum of a register and a small offset), then we can use a chain rule that maps a register non-terminal to an address (without producing any code) rather than introducing an explicit rule for each instruction of that kind.
- We can use the same technique to handle machines with more than one register class (Address and Data Registers for the MC68XXX architecture, for example).



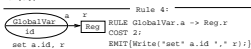
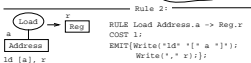
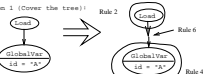
```

set A, %10
ld [%10], %10
set 314, %11
add %10, %11, %10
set A, %11
st %10, [%11]
    
```

Definition 3 (Consume the tree):



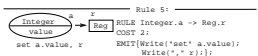
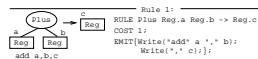
Definition 1 (Cover the tree):

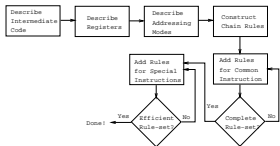


Definition 3 (Consume the tree):

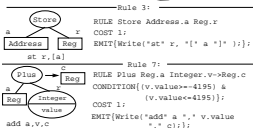
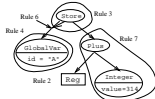


Definition 1 (Cover the tree):





- A working BEG specification can be extended with new rules that produce better code in special cases.



Without Rule 7:

```
set A, %10
ld [%10], %10
set 314, %11
add %10, %11, %10
set A, %11
st %10, [%11]
```

With Rule 7:

```
set A, %10
ld [%10], %10
add %10, 314, %10
set A, %11
st %10, [%11]
```

Structure of Sparc.cgd

- The main BEG specification for the SPARC architecture is stored in a file Sparc.cgd. It has the following global structure:

```
CODE_GENERATOR_DESCRIPTION Sparc;
INTERMEDIATE_REPRESENTATION
NONTERMINALS IntExp;
OPERATORS
    Intermediate code format
REGISTERS
    Description of registers
AVAIL (Available registers);
NONTERMINALS
    Description of addressing modes
Code generation rules
INSERTS
    Code to be inserted in generated modules
END CODE_GENERATOR_DESCRIPTION
```

Structure of BEG Specifications

- This is how the tree-nodes are described:

```
CODE_GENERATOR_DESCRIPTION Sparc;
INTERMEDIATE_REPRESENTATION;
NONTERMINALS IntExp;
```

OPERATORS

```
Integer (value : INTEGER) -> IntExp;
GlobalVar (id : Identifier) -> IntExp;
Load IntExp -> IntExp;
Store IntExp * IntExp;
Plus IntExp + IntExp -> IntExp;
```

...

```
END CODE_GENERATOR_DESCRIPTION ...
```

⏪ ⏩ ⏴ ⏵ ⏶ ⏷ ⏸ ⏹ ⏺ ⏻ ⏼ ⏽ ⏾ ⏿ 🔍

⏪ ⏩ ⏴ ⏵ ⏶ ⏷ ⏸ ⏹ ⏺ ⏻ ⏼ ⏽ ⏾ ⏿ 🔍

- We need to know the addressing modes the machine has and which registers are available.

1 Integer Register

- global (%g0, %g1, ..., %g7)
- local (%l0, %l1, ..., %l7)
- input (%i0, %i1, ..., %i7)
- output (%o0, %o1, ..., %o7)

2 Floating Point Register

- (%f0, %f1, ..., %f31)

3 Address

- reg₁ + reg₂
- reg₁ + const₁₃
- const₁₃

4 Register or Immediate

- reg₁
- const₁₃

Registers and Addr. Modes

```
CODE_GENERATOR_DESCRIPTION Sparc;
REGISTERS
```

```
i0,i1,i2,i3,i4,i5,i6,i7, ..., i5,i6,i7;
AVAIL (i0..i7,i0..i5,i7,o0..o5,g1..g6);
```

NONTERMINALS

```
Register REGISTERS (i0,i1,i2,...,i5,...);
Greg REGISTERS (g1,g2,g3,g4,...);
Address ADRMODE
```

COND_ATTRIBUTES

```
(val : INTEGER)
```

```
(reg1 : Register; reg2 : Register);
```

```
RegOrImm ADRMODE
```

COND_ATTRIBUTES

```
(val : INTEGER)
```

```
(reg : Register);
```

```
IntConst13 COND_ATTRIBUTES
```

```
(val : INTEGER);
```

```
END CODE_GENERATOR_DESCRIPTION
```

⏪ ⏩ ⏴ ⏵ ⏶ ⏷ ⏸ ⏹ ⏺ ⏻ ⏼ ⏽ ⏾ ⏿ 🔍

⏪ ⏩ ⏴ ⏵ ⏶ ⏷ ⏸ ⏹ ⏺ ⏻ ⏼ ⏽ ⏾ ⏿ 🔍

BEG Rules

RULE Pattern consisting of register classes, addressing modes, operators, etc.

CONDITION Condition evaluated during cover phase. If result is FALSE rule will not be used.

COST Cost (in number of cycles) of using rule.

CHANGE Registers affected by generated code.

EVAL Code to evaluate attributes, build addressing modes, etc.

EMIT Code to generate target code.

```
RULE arg1 arg2 ... argn
      -> result
```

```
CONDITION { Boolean Modula-2-expression };
```

```
COST Integer;
```

```
CHANGE { Affected registers };
```

```
EVAL { Cover phase Modula-2-code };
```

```
EMIT { Output phase Modula-2-code };
```


Admissible Registers I:

```

RULE Mult Reg.a(o0) Reg.b(o1)->Reg.c(o0)
COST 4;
EMIT {Write("mul...");}

```

- The mul instruction requires its arguments to be in registers %o0 and %o1, and returns its result in register %o0.

Admissible Registers II:

```

RULE Instr Reg.a(o0..06) -> Reg.b(o0)
COST 4;
EMIT {Write("...");}

```

- In order to use this rule, Beg has to make sure that the argument is in one of the specified registers.

Registers Affected by Side-Effects:

```

RULE Call.c Arg;
COST 2;
CHANGE (g1..g6);
EMIT {Write("call...");}

```

- A procedure call affects all global registers.

Computed Registers:

```

RULE Instr Reg.a -> Reg.c({Mod-2 Expr})
COST 4;
EMIT {Write("...");}

```

- The expression returns a suitable register. This is useful when passing parameters in registers.

Constant Folding:

- The Modula-2 code in in the **EVAL** part of a rule is executed during the cover phase. It can therefore be used to evaluate (fold) constant expressions.
- For example, an expression $a + 2 * 3$ would be translated into code for $a + 6$, with $2 * 3$ evaluated at compile time.

Rule 9:

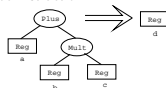
```

RULE Plus Integer.a Integer.b
-> Integer.c
COST 0;
EVAL {c.value := a.value + b.value}

```

Complex Patterns:

- A BEG rule can be given an arbitrarily complex pattern. The pattern part of a rule is actually a **preorder** traversal of a tree pattern.
- The following example shows how we can specify a multiply-and-add instruction:



Rule 10:

```

RULE Plus Reg.a Mult Reg.b Reg.c -> Reg.d
COST 2;
EMIT {write("MADD ...");}

```

- The last part of a BEG specification consists of instructions that inserts extra code into the generated modules.
- Often this code consists of **IMPORT** statements that makes it possible for the generated modules to find types and procedures defined by us.
- For every module M generated by BEG, you can insert code into its definition and implementation modules. This is done in so called **insertion points** (Ip):

```
IpM_d { Code inserted into M.def,
        after IMPORT statements. }
IpM_i { Code inserted into M.mod }
```

- There are special insertion points called IpInOut (for inserting IO routines) and IpNoReg (for inserting code to call when register allocation fails).

CODE_GENERATOR_DESCRIPTION Sparc; INSERTS

```
IpInOut {
  FROM AsmFile IMPORT
    Create, ..., FatalError;
}
IpIR_d {
  FROM GcgBase IMPORT
    Identifier, Condition;
}
IpIR_i {
  FROM GcgBase IMPORT Identifier, ...,
  PrintCondition;
}
IpNoReg {
  FatalError ("Reg alloc failed.");
  HALT;
}
```

Example – IBM370 I

Example: IBM370

```
CODE_GENERATOR_DESCRIPTION IBM370;
INTERMEDIATE_REPRESENTATION
NONTERMINALS Value;
OPERATORS
  Constant ( v : INTEGER ) -> Value;
  Plus      Value + Value -> Value;
  Mult      Value + Value -> Value;
  AddressPlus Value * Value -> Value;
  BlockBase                               -> Value;
  Content      Value -> Value;
  Assign       Value * Value;
```

```

REGISTERS
  R0,R1,R2,R3,R4,R5,R6,R7,R8,R9,R10,R11,
  R12,R13,R14,R15,
  D0(R0,R1),D2(R2,R3),D4(R4,R5),D6(R6,R7),
  D8(R8,R9),D10(R10,R11),D12(R12,R13),
  D14(R14,R15), F0,F1,F2,F3,F4,F5,F6,F7,
  DF0(F0,F1),DF2(F2,F3),DF4(F4,F5),DF6(F6,F7);
NONTERMINALS
  Register REGISTERS (R0,R1,R2,R3,R4,R5,R6,R7,
    R8,R9,R10,R11,R12);
  Double REGISTERS (D0,D2,D4,D6,D8,D10);
  RegSum ADRMODE (r:Register; s:Register);

```

```

RULE AddressPlus
  Register.i (R1..R15)
  Register.b (R1..R15) -> RegSum;
  COST 0;
  EMIT {RegSum.r := i; RegSum.s :=b};

RULE Double -> Register(R1,R3,R5,R7,R9,R11);
  COST 0; TARGET Double;

RULE Register (R0,R2,R4,R6,R8,R10) ->
  Double.d (D0,D2,D4,D6,D8,D10);
  COST 2;
  TARGET Register;
  EMIT {Write("SRDA ", d, 32);}

```

```

RULE Constant -> Register.r;
  COST 5;
  EMIT {Write("L ", r, "=A(", Constant.v,")");};

RULE Plus Register.s Register.r -> Register;
  COST 2;
  TARGET r;
  EMIT {Write("AR ", r, s);};

RULE Mult Register.a(R1,R3,R5,R7,R9,R11)
  Register.b
  -> Double.d (D0,D2,D4,D6,D8,D10);
  COST 20; TARGET a;
  EMIT {Write("MR ", d, b);};

```

Summary

- Read the Tiger Book, Chapter 9, *Instruction Selection*.
 - Read “Emmelmann, Schröder, Landwehr: *BEG – A generator for Efficient Back Ends*”, PLDI '89.
 - Or, read the Dragon Book, pp. 572–580.
- In the Sparc example, the val-attributes in the Address, RegOrImm, and IntConst13 addressing modes are specified as **condition attributes**. This means that this attribute gets its value during the code generator's *cover phase*. The register attributes, on the other hand, get their values during the *output phase*.