CSc 553

Principles of Compilation

22 : Code Generation — Tree Matching

Department of Computer Science
University of Arizona

collberg@gmail.com

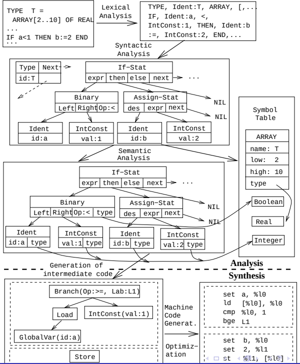
Code Generation with BEG

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.

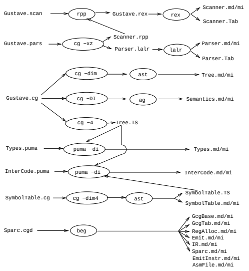


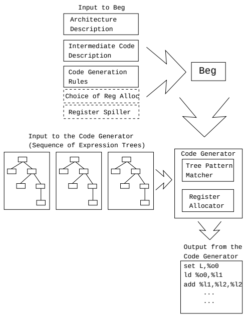
```
set  a, %l0
ld   [%l0], %l0
cmp  %l0, 1
bge  L1

set  b, %l0
set  2, %l1
st   %l1, [%l0]
```

Cocktail Tools

Input	Tool	Output
Descr. of Lexical items.	**rex**	Scanner
Description of syntax.	**lalr**	Parser
Description of Abstract Syntax	**ast**	Abs. Syntax Tree ADT
Description of Symbol Table Data Structures	**ast**	Symbol Table ADT
Description of Static Semantics	**ag**	Attribute Evaluator
Abs. Syntax to IR Mapping	**puma**	Intermediate Code Gen.
Type Checking Rules	**puma**	Type Checker
Description of IR and Machine Architecture	**beg**	Machine Code Generator

Cocktail Generated Files

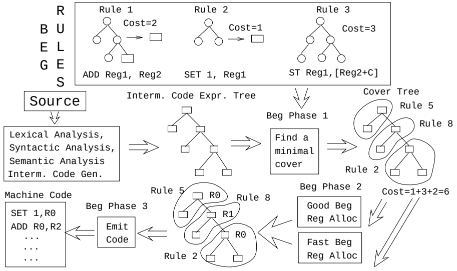




```
set L,%o0
ld %o0,%l1
add %l1,%l2,%l2
  ...
  ...
```

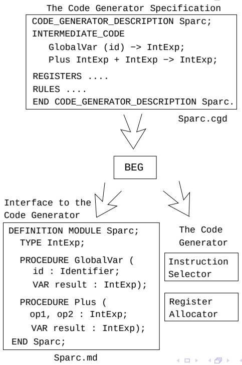
The Phases of Beg I

- The code generator generated by BEG runs in three phases.
- The first phase uses pattern matching to construct a **minimal cover** of the input tree: the code generator will find the set of code generator rules which will produce the cheapest (fastest) code sequences. BEG does this by **covering** the input tree with the cheapest sequence of patterns.
- The second phase allocates register.
- The third phase generates the actual code.
- Another way of thinking about a minimal cover is to say that the selected patterns **reduce** the input tree to the **empty tree** by deleting matched parts of the input tree.



Intermediate Code

- When we build the code generator with BEG, BEG generates a module Sparc.md/mi which is the interface to the code generator.
- At compile-time we call procedures in this module to build the intermediate code tree.
- The code generator generated by BEG assumes that the intermediate code is in a tree format. The structure of the tree-nodes is described by a specification.

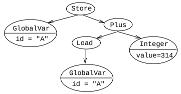


```
CODE_GENERATOR_DESCRIPTION Sparc;
INTERMEDIATE_CODE
   GlobalVar (id) -> IntExp;
   Plus IntExp + IntExp -> IntExp;
REGISTERS ....
RULES ....
END CODE_GENERATOR_DESCRIPTION Sparc.
```

```
DEFINITION MODULE Sparc;
  TYPE IntExp;
  PROCEDURE GlobalVar (
    id : Identifier;
    VAR result : IntExp);
  PROCEDURE Plus (
    op1, op2 : IntExp;
    VAR result : IntExp);
END Sparc;
```

Intermediate Code II

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

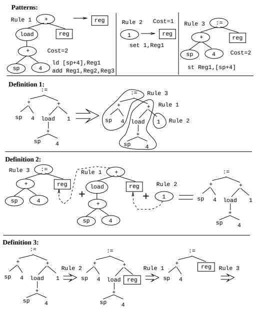


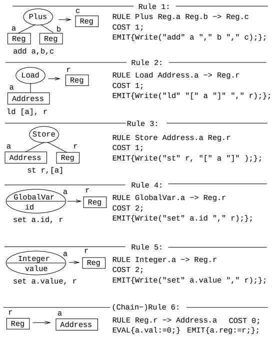
- The $+$ ($*$)-sign makes the operator (non-)commutative (Plus is commutative since $a + b \equiv b + a$).
- Parentheses indicate input attributes, data passed from the front end to the code generator.

Cover Trees I

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

1. Find a set of patterns that covers the input tree, such that the total cost is minimized.
2. Build a tree (identical to the input tree) out of the available patterns, such that the total cost of the new tree is minimized.
3. 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. Chose the set of patterns with minimal cost.





```
Rule 1:
RULE Plus Reg.a Reg.b -> Reg.c
COST 1;
EMIT{Write("add" a "," b "," c);};

Rule 2:
RULE Load Address.a -> Reg.r
COST 1;
EMIT{Write("ld" "[" a "]" "," r);};

Rule 3:
RULE Store Address.a Reg.r
COST 1;
EMIT{Write("st" r, "[" a "]" );};

Rule 4:
RULE GlobalVar.a -> Reg.r
COST 2;
EMIT{Write("set" a.id "," r);};

Rule 5:
RULE Integer.a -> Reg.r
COST 2;
EMIT{Write("set" a.value "," r);};

(Chain-)Rule 6:
RULE Reg.r -> Address.a   COST 0;
EVAL{a.val:=0;} EMIT{a.reg:=r;};
```

Beg Example I (b) – Explanation

- 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:

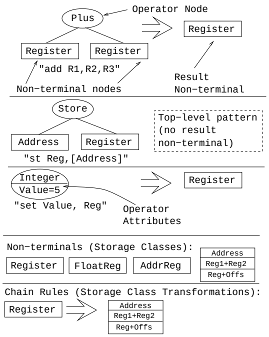
Rule 1, for example, states that on the SPARC there is an add-instruction which adds two registers into a third, and which executes in one (1) machine cycle.

Beg Example I (c) – Explanation

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.



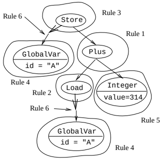
Beg Tree Patterns II

- 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.

Beg Tree Patterns III

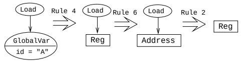
- A chain rule transforms a value from one storage class to another. Often this can be done without generating any code. For example, if one instruction returns its result in a register, and the next instruction requires its argument in an address (e.g. a Sparc addressing mode which is either a register, 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).

BEG Example I (d)

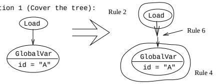


```
set A, %l0
ld  [%l0], %l0
set 314, %l1
add %l0, %l1, %l0
set A, %l1
st  %l0, [%l1]
```

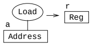
Definition3 (Consume the tree):



Definition 1 (Cover the tree):



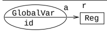
Rule 2:



ld [a], r

```
RULE Load Address.a -> Reg.r
COST 1;
EMIT{Write("ld" "[" a "]");
     Write("," r);};
```

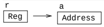
Rule 4:



set a.id, r

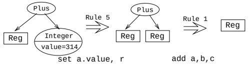
```
RULE GlobalVar.a -> Reg.r
COST 2;
EMIT{Write("set" a.id "," r);};
```

(Chain-)Rule 6:



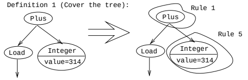
```
RULE Reg.r -> Address.a    COST 0;
EVAL{a.val:=0;} EMIT{a.reg:=r;};
```

Definition 3 (Consume the tree):

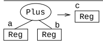


set a.value, r   add a,b,c

Definition 1 (Cover the tree):



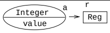
Rule 1:



add a,b,c

```
RULE Plus Reg.a Reg.b -> Reg.c
COST 1;
EMIT{Write("add" a "," b);
     Write("," c);};
```

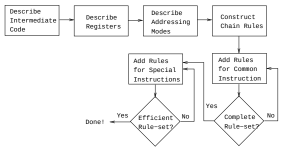
Rule 5:



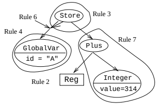
set a.value, r

```
RULE Integer.a -> Reg.r
COST 2;
EMIT{Write("set" a.value);
     Write("," r);};
```

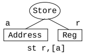
Building Beg Specifications


Describe Intermediate Code
Describe Registers
Describe Addressing Modes
Construct Chain Rules
Add Rules for Common Instruction
Add Rules for Special Instructions
Complete Rule-set?
Yes
No
Efficient Rule-set?
Yes
No
Done!

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

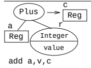


Rule 3:



```
RULE Store Address.a Reg.r
COST 1;
EMIT{Write("st" r, "[" a "]" );};
```

Rule 7:



```
RULE Plus Reg.a Integer.v->Reg.c
CONDITION{(v.value>=-4195) &
          (v.value<=4195)};
COST 1;
EMIT{Write("add" a "," v.value
           "," c);};
```

Without Rule 7:

```
set A, %l0
ld  [%l0], %l0
set 314, %l1
add %l0, %l1, %l0
set A, %l1
st  %l0, [%l1]
```

With Rule 7:

```
set A, %l0
ld  [%l0], %l0
add %l0, 314, %l0
set A, %l1
st  %l0, [%l1]
```

Structure of BEG Specifications

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 · · · .

Intermediate Format

- 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  ....
```

SPARC Architecture

- 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**
   - $reg_1 + reg_2$
   - $reg_1 + const_{13}$
   - $const_{13}$
4. **Register or Immediate**
   - reg_1
   - $const_{13}$

Registers and Addr. Modes

```
CODE_GENERATOR_DESCRIPTION Sparc;
REGISTERS
l0,l1,l2,l3,l4,l5,l6,l7, ..., i5,i6,i7;
AVAIL  (l0..l7,i0..i5,i7,o0..o5,g1..g6);
NONTERMINALS
   Register    REGISTERS       (l0,l1,l2,...,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 };
```

Advanced BEG Rules I

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.

Advanced BEG Rules II

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.

Advanced BEG Rules III

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.

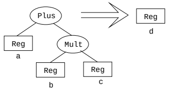
Rule 9:

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

Advanced BEG Rules IV

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 ...");}
```

Inserting Modula-2 Code I

- 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).

Inserting Modula-2 Code II

```
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

Example – IBM370 I

```
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;
```

Example – IBM370 II

```
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);
```

Example – IBM370 III

```
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);}"
```

Example – IBM370 IV

```
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

Readings and References

- Read the Tiger Book, Chapter 9, *Instruction Selection*.
- Read "Emmelmann, Schröer, Landwehr: *BEG – A generator for Efficient Back Ends*", PLDI '89.
- Or, read the Dragon Book, pp. 572–580.

Summary

- 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*.