

# Code-generator generators

## Generating the code selector

Clément Vasseur <[clement.vasseur@lrde.epita.fr](mailto:clement.vasseur@lrde.epita.fr)>

LRDE seminar, June 2, 2004

<http://www.lrde.epita.fr/>



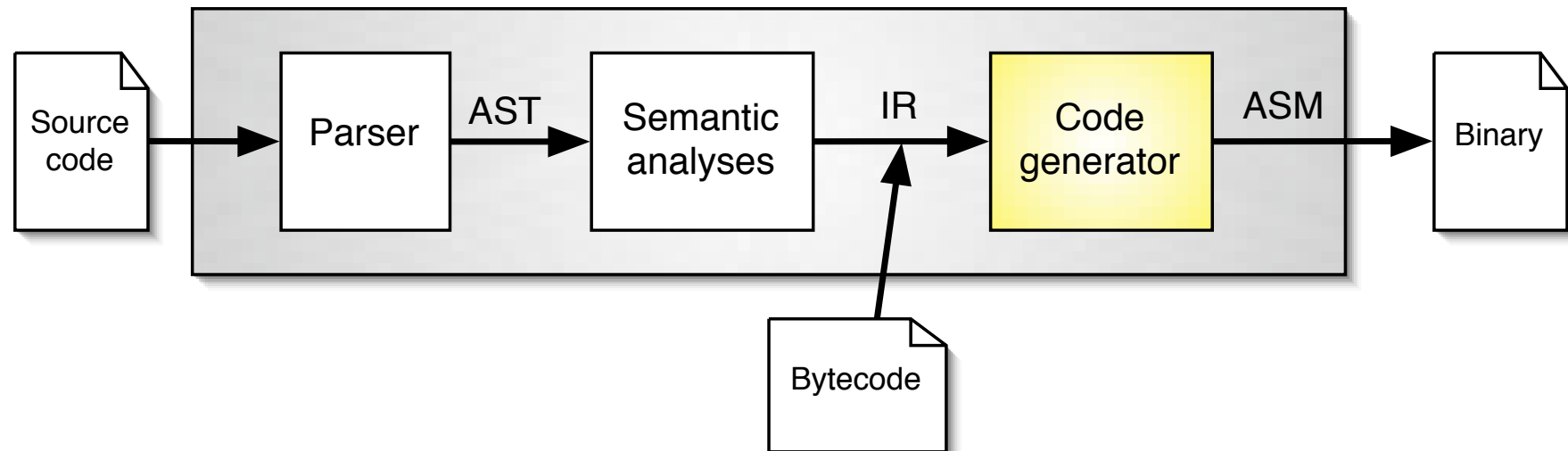
## Copying this document

Copyright © 2004 LRDE.

Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.2 or any later version published by the Free Software Foundation; with the Invariant Sections being just “Copying this document”, no Front-Cover Texts, and no Back-Cover Texts.

A copy of the license is provided in the file COPYING.DOC.

# Inside the compiler



**AST** Abstract Syntax Tree

**IR** Intermediate Representation

**ASM** Assembly instructions

# Introduction

Code selection is:

- a **critical** part of the compiler,
- **tricky** because the CPU instruction set must be known,
- **hard to debug** because the output is assembly code,
- **specific** for each architecture supported by the compiler.

Is it possible to **generate** the code generator?

# Table of Contents

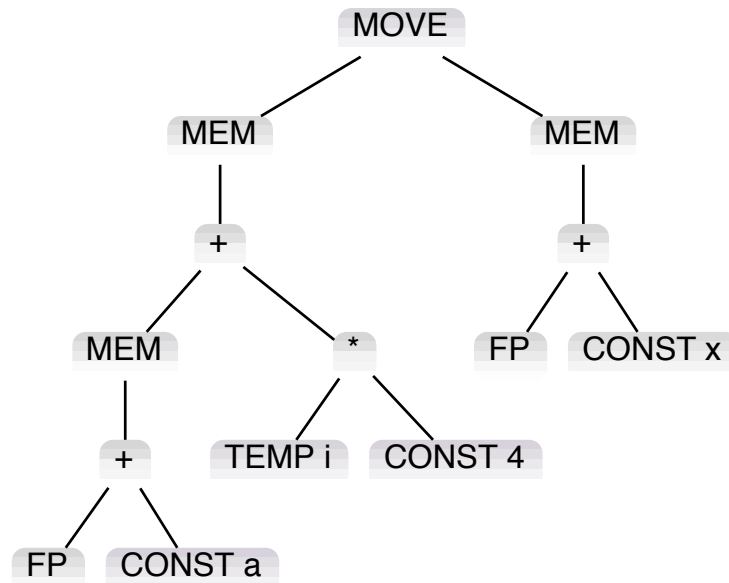
<b>Instruction selection</b> .....	<b>5</b>
<b>Generating the code generator</b> .....	<b>11</b>
<b>Other approaches</b> .....	<b>16</b>
<b>Tiger implementation</b> .....	<b>22</b>

# Instruction selection

Mapping IR trees to assembly instructions.

**Example:**  $a[i] := x$

$r_1 \leftarrow r_0 + a$   
 $r_1 \leftarrow fp + r_1$   
 $r_1 \leftarrow M[r_1 + 0]$   
 $r_1 \leftarrow r_0 + 4$   
 $r_1 \leftarrow r_i \times r_2$   
 $r_1 \leftarrow r_1 + r_2$   
 $r_2 \leftarrow r_0 + x$   
 $r_2 \leftarrow fp + r_2$   
 $r_2 \leftarrow M[r_2 + 0]$   
 $M[r_1 + 0] \leftarrow r_2$

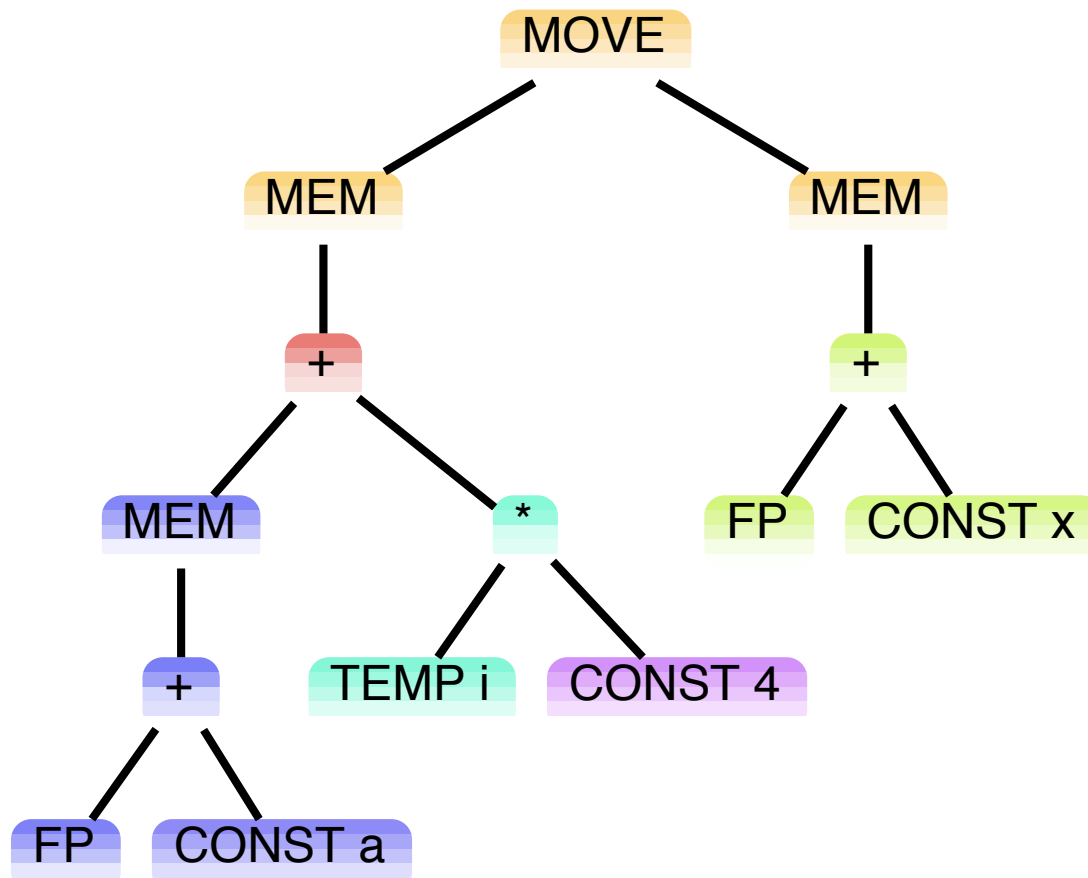


$r_1 \leftarrow M[fp + a]$   
 $r_2 \leftarrow r_0 + 4$   
 $r_2 \leftarrow r_i \times r_2$   
 $r_1 \leftarrow r_1 + r_2$   
 $r_2 \leftarrow fp + x$   
 $M[r_1] \leftarrow M[r_2]$

## Maximal munch

- Algorithm:
  - Starting at the root node, **find the largest tile** that fits.
  - **Cover the root node**, and perhaps several other nodes, with this tile.
  - **Repeat** for each subtree.
- **Easy** to understand and to implement (pattern matching).
- Good result with a **RISC** instruction set, but...
- **Not an optimum** tiling.

# Maximal munch - example

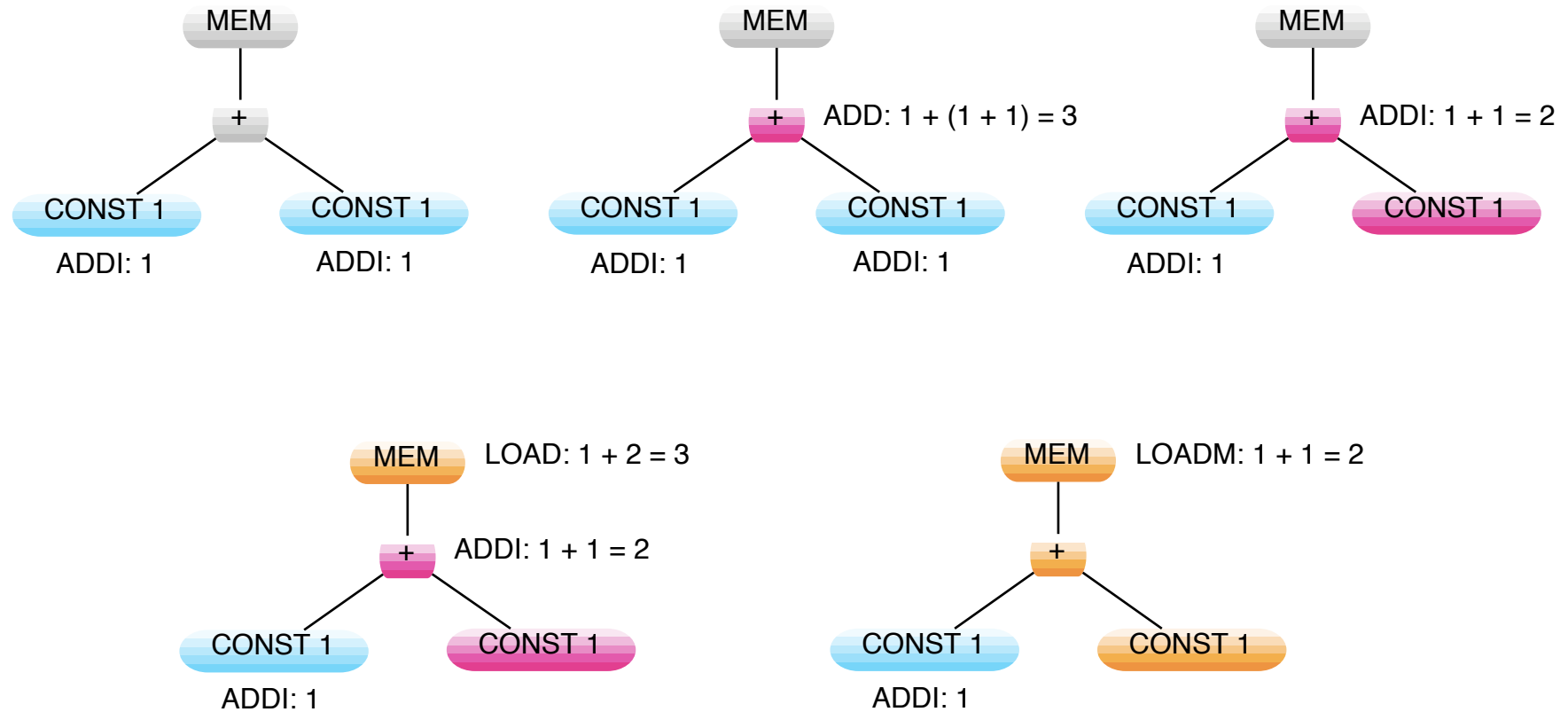


- $r_1 \leftarrow M[fp + a]$
- $r_2 \leftarrow r_0 + 4$
- $r_2 \leftarrow r_i \times r_2$
- $r_1 \leftarrow r_1 + r_2$
- $r_2 \leftarrow fp + x$
- $M[r_1] \leftarrow M[r_2]$

# Dynamic programming

- Assign a **cost** to every node in the tree.
- **Bottom-up** traversal:
  - for each tile  $t$  of cost  $c$  that matches at node  $n$
  - $c_i$  = cost of each subtree corresponding to the leaves of  $t$
  - cost of  $n = c + \sum c_i$
- **Top-down** traversal to select the minimum cost tiling.

# Dynamic programming - example



---

# Outline

<b>Instruction selection</b> .....	<b>5</b>
<b>Generating the code generator</b> .....	<b>11</b>
<b>Other approaches</b> .....	<b>16</b>
<b>Tiger implementation</b> .....	<b>22</b>

# Generating the code generator

## History

**1980** *Graham* and *Glanville*:

bottom-up shift-reduce parser, using  $LR_0$  grammars.

**1984** *Davidson* and *Fraser*:

macro-expansion of trees to naive Register Transfer Lists, followed by declarative code improvement (*GCC*).

**1989** *TWIG* - *Aho, Ganapathi, Tjiang*:

tree pattern matching and dynamic programming.

**1992** *BURG* - *Fraser, Henry, Proebsting*:

BURS (Bottom-Up Rewrite System).

## Burg tree-grammar specification

A **cost** and an **action** are specified for each rule.

```
stmt: ASGNI(disp,reg) = 4 (1);
stmt: reg = 5;
reg:  ADDI(reg,rc) = 6 (1);
reg:  CVCI(INDIRC(disp)) = 7 (1);
reg:  IOI = 8;
reg:  disp = 9 (1);
disp: ADDI(reg,con) = 10;
disp: ADDRLP = 11;
rc:   con = 12;
rc:   reg = 13;
con:  CNSTI = 14;
con:  IOI = 15;
```

## Generator implementation (*iburg*)

- Generalization of **dynamic programming**.
- The minimum-cost match at each node **for each non-terminal** of the grammar is computed.
- **label** and **reduce**:
  - `label` is generated.
  - `reduce` is trivial to write.

## The label function

The generated code generator implements the following pseudo-code.

```
function label(node p)
  label(p.left)
  label(p.right)

  p.rule[0..N] = 0
  p.cost[0..N] = +inf

  switch (p.op)
    case ADDRLP:
      c := 0
      rec(p, disp, c, 11)
      rec(p, reg, c + 1, 9)
      rec(p, rc, c + 1 + 0, 13)
      rec(p, stmt, c + 1 + 0, 5)

case CVCI:
  if p.left.op = INDIRC
    and p.left.left.rule[disp] != 0
  then
    c := p.left.left.cost[disp] + 1
    rec(p, reg, c, 7)
    rec(p, rc, c + 0, 13)
    rec(p, stmt, c + 0, 5)
  endif
```

---

# Outline

<b>Instruction selection</b> .....	<b>5</b>
<b>Generating the code generator</b> .....	<b>11</b>
<b>Other approaches</b> .....	<b>16</b>
<b>Tiger implementation</b> .....	<b>22</b>

## Other approaches

- burg
- mburg
- wburg
- gburg

# burg

- *Fraser, Henry and Proebsting* (1992).
- BURS: Bottom-up Rewrite System.
- Dynamic programming at compile-compile time.
- Encode the unbounded number of tree configurations into a finite set of equivalence classes.

# mburg

- *Gough* and *Ledermann* (1995).
- Similar to *iburg*.
- Produce code generators in Modula-2.
- Incremental labelling.
- Forced, preemptive reductions.

---

# wburg

- *Proebsting* and *Whaley* (1995).
- Optimal generation in a single bottom-up pass.
- Eliminate the need for an IR tree.
- Support a proper subset of the grammars handled by two-pass systems.

# gburg

- *Fraser* and *Proebsting* (1999).
- Greedy Bottom-Up Rewrite Generator.
- **Fast** and **small** code generators for *JIT* compilers.
- One pass on linearized postfix notation (stack code).
- Pattern matching using a finite state machine.

---

# Outline

<b>Instruction selection</b> .....	<b>5</b>
<b>Generating the code generator</b> .....	<b>11</b>
<b>Other approaches</b> .....	<b>16</b>
<b>Tiger implementation</b> .....	<b>22</b>

## Tiger implementation

- Code selection in the LRDE **Tiger** Compiler.
- Using *monoburg* (*iburg*).
- Tiger LIR  $\rightarrow$  **MIPS** assembly.
- RISC CPU  $\Rightarrow$  **small grammar** (under 100 rules).

### **To do:**

Modify *monoburg* to output the code generator in **C++**, in order to integrate it to TC.

## Conclusion

- Code-generator generation is **practical**.
- **Several techniques** exist, depending on the requirements.
- Improve **code quality**, **maintenance** and ease of **retargetting**.

Anyway ...

... the more we generate, the better it is.

## References

Fraser, C. W., Hanson, D. R., and Proebsting, T. A. (1992). Engineering a simple, efficient code-generator generator. *ACM Letters on Programming Languages and Systems*, 1(3):213–226.

Fraser, C. W. and Proebsting, T. A. (1999). Finite-state code generation. *ACM SIGPLAN Notices*, 34(5):270–280.

Gough, K. J. and Ledermann, J. (1997). Optimal code-selection using MBURG. *Australian Computer Science Comm.: Proc. 20th Australasian Computer Science Conf., ACSC*, 19(1):441–450.

Proebsting, T. A. and Whaley, B. R. (1995). One-pass, optimal tree parsing - with or without trees.

# Questions