4 The Olive Description Language

The SPAM compiler uses *Olive* as its machine description language. Olive was designed by Steve Tjiang, and is based on *Twig* and *iburg*. iburg is a code-generator generator developed by Dave Hanson and Christopher Fraser. Twig is a code-generator generator designed by Steve Tjiang, A.V. Aho and M. Ganapathi.

Olive is not a machine description language, in the strict sense of the word, given that it only supports structural description of the target processor *Instruction Set Architecture* (ISA). Olive does not capture behavioral information, as for example the pipeline timing. Issues associated with that, like the need to fill in *load delay slots*, have to be addressed by the developer separately during the design of the new compiler. On the other hand, Olive provides the developer with a set of powerful code generation algorithms, and most of all, it defines a methodology which is very helpful in the design of the compiler code generator module.

The following sections give a broad overview of Olive, and describe how to use Olive functions in the design of the processor Olive file. Olive takes this file as input to generate a code-generator for the target processor. The resulting code-generator reads an Olive expression tree (see the description of the *otree_node.h* file in this documentation) and returns machine code. Section 4.1 describes the history and the advantages of using Olive when compared with other code-generator generators. Sections 4.4 and 4.7 describe the syntax of Olive and its embedded functions. These sections have been adapted from the document *An Olive Twig*, a Synopsys Inc. technical report by Steve Tjiang.

4.1 Overview

This section describes Olive, a new version of Twig based on iburg. Twig is a tree-manipulation system intended primarily for code generation. Twig can be used to build a code generator that takes expression trees as input and emits assembly code. Andrew Appel used Twig to build several code generators in his compiler class at Princeton. He found that Twig was quite good at manipulating *Abstract Syntax Trees*[1] (ASTs).

Though intended for code generation, Twig's flexibility also suits other applications. Kurt Keutzer used Twig to build Dagon [14], a logic synthesis system used at Bell Labs. Twig's flexibility arises from dynamic programming and the generality of its cost computations, which can be practically any block of C. Dynamic programming offers a powerful method to choose between many different alternatives that arise during code generation and technology-mapping. General cost computations permit users to specify fairly sophisticated tests. Moreover, general cost computations allow dynamic programming to be performed even with non-integer costs.

Another key feature of Twig is the ease with which tree manipulations can be interleaved with pattern-matching. Twig's pattern-action rules can rewrite subtrees that can participate in further matching.

4.2 Advantages of iburg

Unfortunately, Twig's generality hurts pattern-matching performance. Because costs are opaque to the Twig compiler, powerful table-generation techniques such as BURS theory cannot be applied. The bottom line: general costs compel Twig to perform all dynamic programming during run-time instead of at table-generation time, saddling Twig with slow run-time performance. For example, in a VAX code generator designed with Twig, the pattern matching accounted for over 80% of the execution time.

Several new tree-matching systems – burg, iburg – have capitalized on these deficiencies. Burg restricts costs to integers and applies BURS theory to create very fast pattern-matchers. Unfortunately, the matchers are difficult to debug as the details are buried in inscrutable tables. To make debugging easier, Dave Hanson designed iburg for use in his compiler class at Princeton. The matcher-generator – iburg – has done away with tables, instead implementing the pattern matching directly with **IF** and **SWITCH** statements.

4.3 What New Olive Offers

Olive is Steve Tjiang's attempt to update Twig so that it becomes a better tool for tree-matching. Here are the areas in which Olive innovates:

[1] ASTs are a common intermediate representation used in compiler technology.

1 *A richer specification language:* Recent systems like burg and iburg have shied away from more powerful specification languages. The developers of burg and iburg intend that they be used as a back-end for tree-manipulation systems that do offer richer specification languages. While the developers' policy does make burg and iburg more generally applicable, it reduces their value as pre-packaged solutions. The aim of Olive is to ease the flow of information between rules. This flow could not be specified directly in Twig. Olive treats rules more like functions with arguments and will offer easier ways for one rule to invoke another, both features missing from Twig. In Olive, communication between rules can occur only through global variables or special fields in the nodes of the AST.

2 *Faster pattern matching:* Olive takes advantage of the open-coding style of iburg, i.e. code the table interpretation directly as **IF** and **SWITCH** statements. In addition, pattern matching overhead is minimized when pattern-matching is not used.

3 *Generalized costs:* Olive maintains general costs because of their usefulness as a means to write predicates. Although this will cost somewhat in performance, that impact will be much less than in Twig, and be more similar to that of iburg's.

The following sections describe the Olive language, shows how to interface to the pattern-matcher, and outlines the code that Olive generates.

4.4 Lexical Conventions

Identifiers are defined in Olive as one might expect: they are strings of letters, digits or underscores, starting with either a letter or a percent sign (%). All Olive keywords begin with a percent sign. Olive uses the characters "**; :() , =**" as punctuations.

Like *YACC* grammar productions, Olive rules contain embedded C code that are to be integrated into the pattern matcher. The C code can come in two forms. They can be a correctly parenthesized C expression or they can be brace-enclosed blocks of C code as in YACC. Within the embedded code, C lexical conventions rule with the exception of special constructions that begin with a dollar sign (**$**), constructs with special meaning to Olive. They will be explained later in Section 4.5.2.

4.5 Rules and Tree Patterns

In the following, EBNF is used to describe the Olive language. A **nonterm** in tree-patterns corresponds to

rule	→	**nonterm : tree [cost] action;**
tree	→	**term (tree_list)**
	|	**term**
	|	**nonterm**
tree-list	→	**tree_list, child**
	|	**child**
child	→	**tree**
	|	**_**
cost	→	**C-code**
	|	**C-expr**
action	→	**C-code**

non-terminals in the grammar. A **term** corresponds to terminal symbols. For tree-patterns, **term** symbols label tree nodes. The special identifier "**_**" denotes a *don't care* situation which is used to bypass pattern-matching for particular subtrees. A rule matches a tree if and only if two things happen:

1 The tree pattern must match the subtree. *Matching* can be defined recursively. For a tree to match the pattern

$$x\ (t_1, t_2, \ldots, t_n)$$

the root has symbol x; there are n subtrees and each subtree matches a corresponding t_i. The leftmost subtree matches t_1; the next leftmost matches t_2; and so on.

2 The **cost** part of the rule must return a finite cost.

4.5.1 Costs

The **cost** part computes the cost of the rule when the tree pattern of the rule matches. The **cost** part can be used as a predicate: it can return a zero cost to accept a match or it can return an infinite cost to force a mismatch.

The **cost** part of a rule is either a C expression or C code. If the **cost** part is a C expression the expression is evaluated. If it evaluates to true then the rule matches with cost zero, otherwise the rule fails to match. If the **cost** part is C code the code is executed. Olive expects the code to return the cost of the match in special variables that begin with a dollar sign (`$`), as described in the following section.

4.5.2 Actions and Prototypes

Imagine every non-terminal symbol as being associated with a set of functions. These functions have the same prototype – they return values of the same type; they have the same number of arguments and each corresponding element has identical types. The developer must supply a prototype for a non-terminal function using a **%declare** statement.

The **code** and **action** parts of a rule can communicate with the pattern- matcher through special variables and functions whose names begin with a percent sign (`%`). The list below summarizes the conventions.

- `$n`
  The nth node in the tree pattern. Nodes in a tree pattern are numbered in the order in which they appear in a pre-order traversal. For example, in the rule
  ```
  L: A (B, C (D))
  ```
  the `A`, `B`, `C`, and `D` are numbered 1, 2, 3, and 4 respectively. The designator `$0` refers to `L`, left-hand-side of a rule.

- `$cost[n]`
  The cost of the nth node in the tree pattern. This can only be used inside the **cost** part of a rule. To return the cost, a rule should assign to `$cost [0]`.

- `$rule[n]`
  This function returns the **act** of the current rule. The **act** of a rule is an internal data structure that Olive uses to keep track of the non-terminal which was selected to match the current AST node. See action `$exec` below on how to use **act**.

- `$getcost[n]`
  The cost of the nth node in the tree pattern. This can only be used inside the **action** part of a rule[2].

- `$action[n](a, b, ...)`
  Explicitly perform the action part of the rule associated with the nth node in the tree pattern. This represents a call to the action part of the rule with arguments `a`, `b`, `c`, etc. The action part returns a value of the type specified in its prototype declaration. The action part of a rule can be executed more than once.

- `$immed[n,lhs](a, b, ...)`
  Explicitly perform the action part of the rule associated with the nth node in the tree pattern, which must be a "`_`". The action routine should have the same prototype as an action part of a rule whose left hand side is the non-terminal symbol `lhs`. Execution will fail if the nth node does not match `lhs`. The action part of a rule can be executed more than once.

- `$exec[act](a, b, ...)`
  Explicitly perform the action part of the rule which matched the node described in **act**. The action routine should have the same prototype as the action part of the rule matched in **act**. `$exec` can be used to emit code in advance for those subtrees in which the root has been spilled into memory (see Section 4.10).

- `$match [n, lhs]`
  Returns true if the nth node matches `lhs`. The nth node in the tree pattern must be a "`_`".

[2]In future versions of Olive, `$getcost` should be eliminated and `$cost[n]` used in the action part as well.

4.5.3 Declarations

Terminal and non-terminal symbols must be declared before they are used. A terminal symbol is declared using the `%term` statement. Each `%term` statement declares a list of identifiers as terminals. Olive assigns an index to each terminal symbol and creates a `#define` which can be used to refer to the terminal symbol. To declare a non-terminal symbol and to associate it with a prototype, the developer must use `%declare` statement. Each `%declare` statement looks like this:

`%declare<return type> nonterm<arguments>`

Note that angle brackets are used and not parenthesis. This is done to keep the Olive parser simple.

4.6 Interfacing to Olive

The user must provide two abstract data types to Olive, one for costs and one for tree nodes. This gives the user flexibility in choosing representations for costs and trees. If C++ is used then these could be classes. In C, however, they have to be provided using macro definitions and functions. This section lists the required definitions and functions for costs and trees.

4.6.1 Costs

Olive will support costs defined by the following components:

1 **COST** is a C type, defined either through a macro or a **typedef**.

2 **INFINITY** is the maximum attainable cost value.

3 **DEFAULT_COST** is the default cost returned by rules without a cost part.

4 **COST_LESS (x, y)** is a function that returns true if and only if the first argument costs less than the second.

The cost part of a rule set the costs for the rule by using the designator **$cost[0]**. It can also perform a "**return 0;**" explicitly to abort the consideration of that rule.

4.6.2 Trees

Olive uses the following definitions to access the trees.

1 **NODEPTR** is the type of a pointer to a node.

2 **NULL** denotes a non-existent node.

3 **GET_KIDS (r)** returns a vector of **NODE-POINTER**s that are the children of **r**.

4 **OP_LABEL (r)** returns the label of the node **r**.

5 **SET_STATE (r, s)** assigns the state label s to the node **r**.

6 **STATE_LABEL (r)** returns the state label previously assigned to the node **r**.

4.7 Invoking the Tree Matcher

Olive generates a routine called **burm_label** that is called to invoke the matcher, as shown below: The routine **burm_label** determines a covering for the tree, and may invoke actions on parts of the tree due to immediate rules. If **burm_label** returns a non-zero value, covering was successful. If **lhs** is the non-terminal that matches at the root, one can execute the actions of the rule that form the cover by calling the routine **lhs_action**.

```
if(burm_label(root) == 0)
     error ("matcher failed");
else
     lhs_action (STATE_LABEL (root), arguments);
```

4.7.1 Olive Flags

The following are Olive command line flags:

- **-L**
  Turns *off* the emission of "**#line**" directives.
- **-I**
  Tells Olive to generate **burm_string**, **burm_files**, **burm_file_numbers**, and **burm_line_numbers** which are tables used to map rule numbers back to the file and line number where the rule is defined. These tables are used for debugging.

4.8 The Register Allocator

SPAM contains a number of modules employed to ease the design of new compilers. `TheRegAllocator` is the register allocation module of SPAM. It contains all the functions the developer will need to allocate virtual and physical registers during the design of the Olive file. The functions available in `TheRegAllocator` are:

- `AsmRegister* GetReg (int arnum)`
  This function returns a virtual register numbered `arnum`.
- `AsmRegister* GetReg (AsmRegister* ar = NULL)`
  This function returns a virtual register if `ar` is NULL, and a virtual register with the same number as `ar` otherwise.
- `AsmRegIndir* GetRegIndir (int arnum)`
  This function returns a virtual address register numbered `arnum`.
- `AsmRegister* GetRegIndir(AsmRegIndir* ar = NULL)`
  This function returns a virtual address register if `ar` is NULL, and a virtual address register with the same number as `ar` otherwise.
- `AsmRegIndir* GetRegIndir (var_sym* var)`
  This function returns a virtual address register for pointer variable `var`.
- `AsmRegIndir* GetRegIndir (AsmOperandKind kind)`
  This function returns a virtual address register of type `kind`.
- `AsmRegIndir* GetRegIndir (otree_node* node, AsmRegIndir* dst = NULL)`
  This function returns a virtual address register for `node`. Node `node` must be of type `nVAR` and the variable associated with it must be a pointer variable. The function allocates a virtual address register to the variable if `dst` is `NULL`, or uses `dst` otherwise. The type of the register (e.g. auto-increment) is defined by an annotation attached to the node (see the description of the *annotations.h* file in this documentation).
- `AsmRegIndir* GetRegIndir (instruction* ins)`
  This function allocates an `AsmArrayRef` to instruction `ins` (see the description of the *asm.h* file in this manual).. This instruction must be of type `io_array`.
- `AsmRegIndir* GetRegIndir (AsmRegIndir* ar, int offset)`
  This function returns a virtual address register with the same number of `ar`, which can perform auto-modify operations in the range $\pm$ `offset`.