TD10 - November 30

Note : This TD has been prepared by Mioara Joldes for the 2009-2010 Compilation course

Exercise 1 Instruction Selection - A systematic approach : Graham - Glanville

In the following exercise we will analyse a syntax-directed technique for automatic generation of code generators. In fact, we want to automatically construct a code generator from a machine description. Such a generator, tipically consists ot three steps :

- **intermediate-language transformations** : from front-end to suitable form for pattern matching
- **pattern - matcher** : sequence of reductions to consume the input intermediate code
- **instruction generation** : actually generates an appropriate sequence of instructions + register allocation

Each instruction of the computer is described as a prefix expression together with a certain *sematic* information and an assembly (or machine language template). One example of such an instruction set description is presented in Figure 1.

We consider that the *front end* of the compiler in a **linearized intermediate representation** of the source program. It consists of a series of parathesis-free Polish-prefix expressions. For example,

The code generation algorithm performs a pattern-match similar to **parsing** in which the IR sequence of prefix expressions is translated into a sequence of instructions. However, there are several differences :

- since most operators can access their operands in several ways, the machine description is usually ambiguous.
- the reduce move is considerably more complicated since it selects among a variety of instructions or instructions sequences on the basis of both syntactic and semantic information.
- error situations signaled by the code generator always signify compiler bugs.

Question 1 The code generator

1. For the Graham-Glanville machine description rules in Figure 1, construct an adaptation of an **SLR(1)-like deterministic shift-reduce parser** to emit machine instructions instead of parse trees or intermediate code. In essence, this parser recognizes a language whose productions are machine description rules with "ϵ" replaced by a nonterminal N and the additional productions $S \Rightarrow N^*$.
   Note that ϵ is treated as a nonterminal symbol, rules with ϵ on the left-hand side are allowed. For that, we treat the target machine description as context-free grammar rules (ignoring semantics) and construct first a set of LR(0) states. Note that in the example, the only non-terminal is r.

$r.2 \leftarrow r.1$	$r.2 \Rightarrow r.1$	$r.1, 0, r.2$
$r.3 \leftarrow r.1 + r.2$	$r.3 \Rightarrow +r.1r.2$	add $r.1, r.2, r.3$
$r.3 \leftarrow r.1 + k.2$	$r.3 \Rightarrow +r.1k.2$	add $r.1, k.2, r.3$
$r.2 \leftarrow [r.1]$	$r.2 \Rightarrow\uparrow r.1$	ld $[r.1], r.2$
$[r.2] \leftarrow r.1$	$\epsilon \Rightarrow\leftarrow r.2r1$	st $r.1, [r.2]$
(a)	(b)	(c)

FIGURE 1 – (a) - LIR instructions ; (b) - Graham-Glanville machine description rules ; (c) corresponing SPARC instructions templates

$$\leftarrow + r1\ 2 +\ \uparrow\ r3\ 3$$

2. Given the above differences, it is normal that there will be some "shift-reduce” and "reduce-reduce“ conflicts. Give some heuristics to eliminate these conflicts.
3. Parse the string given in Figure 1. Show the stack, the input and the action taken.

Question 2 Table Construction Algorithm

Now, we are interested in constructing an automatic generator for the code generator. (Similar to YACC with respect to syntactic analyzers). Due to ambiguities, the conflict-resolution rules do not necessarily yield to a recognizer for the entire language generated by the grammar.

In what follows, we resolve the conflicts in the following ways :
– **Shift-Reduce** conflicts are resolved by shifting
– SLR(1) lookahead is used to ensure that **reduce** are included where needed
– **reduce-reduce** conflicts are resolved by semantic restrictions, if not the longest instruction is used
Glanville has proven that if an instruction set is *uniform* then this policy for conflict resolution is language preserving.

An instruction is said to be *uniform* if :

Any left operand of a binary operator b is a valid left operand of b in any prefix expression containing b. Similar defs are for right operands and unary operators. The essential idea of uniformity is that operands to an operator are independent of context. For further reading please see the table construction algorithm in the article joint to this exercise.

Give an algorithm that constructs automatically the parser automaton (represented by a table) and checks whether the grammar rules are uniform.

Question 3 Eliminating Chain Loops

In parsing grammars it is relatively rare to find chain loops, i.e. sets of nonterminals s.t. each of them can derive the others. On the other hand, such loops are extremely common in machine descriptions. As an example of the effect of chain loops consider the simple grammar consisting of the following rules :

$r \Rightarrow \uparrow r$
$r \Rightarrow s$
$s \Rightarrow t$
$t \Rightarrow r$
$\epsilon \Rightarrow \leftarrow st$

1. Show the parsing automaton for this grammar. What happens when the string $\leftarrow r1 \uparrow r2$ is parsed ? Based on this example, by what type of instruction can looping be caused ?
2. Give an algorithm for eliminating chain-loops in a preprocessing phase for the grammar.
3. There are situations when chain-loops may seem desirable. For example, for moving values between integer and floating-point registers in either direction. How can they be accommodated in a different way ?