HansenLite — Syntax-Directed Translation

CSIS 480 — Principles of Compiler Construction

1 Augmented *HansenLite* Grammar

statement →
identifier assignment_operator *expression* <store>
| if <gen_labels> *boolean_expression* then *statement* <goto_begin> <end_label> *else_clause* <begin_label> <pop_labels>
| while <gen_labels> <begin_label> *boolean_expression* do *statement* <goto_begin> <end_label> <pop_labels>
| print *print_expression* <print_printf>
| begin *statement_list* end
| variable identifier <declare>

else_clause →
else *statement*
| ϵ

statement_list →
statement separated_list

separated_list →
statement_separator *statement separated_list*
| ϵ

print_expression →
expression <print_ifmt>
| *string_const* <load> <print_sfmt>

boolean_expression →
expression relational_operator <push_op> *expression* <compute>

expression →
term addition

addition →
additive_operator <push_op> *term* <compute> *addition*
| ϵ

term →
factor multiplication

multiplication →
multiplicative_operator <push_op> *factor* <compute> *multiplication*
| ϵ

factor →

left_paren *expression* right_paren
| identifier <load>
| number <load>
| *signed_term*

signed_term →

additive_operator <push_op> *term* <sign>

To facilitate code-generation, your compiler may need 4 additional stacks:

1. An operand stack
2. An operator stack
3. One or two stacks for labels — either a *begin* and *end* label stack, or a combined stack where you can peek at the top pair of labels (e.g., [being,end]).

2 Semantic Actions

<***load***> For each of the following load operations, we need to push something (e.g., a `Symbol` from the symbol table or a temporary `Symbol`) onto the top of the compiler operand stack. The form of the load depends on the type of the thing being loaded:

local variable Push the Symbol from the symbol table onto the stack; undeclared variables will be missing from the symbol table indicating an error.

int constant A temporary `Symbol` with the numeric value can be pushed.

string constant Note that for string constants we need to keep track of a "Constant" pool (similar to a symbol table) that holds constant symbols. When a string constant is first seen, for example, we should add an entry (e.g., an anonymous `Symbol`) to the constant symbols with that string so we can generate unique ids for constants in the "data" section. You should keep a count of the constants and name constant symbols accordingly (e.g., `str_1`); the proper `Symbol` should be pushed.

<***store***> Emit "`str <Rd>, [fp, <offset>]`" by popping the operand stack and storing the symbol from its current register to the stack-relative memory location or the operand.

<***push_op***> Push the operator onto the compiler operator stack so that other semantic actions can retrieve the token later and take the action.

<***compute***> Pop an operation off the compiler operator stack, pop two operands off the compiler operand stack, and emit the proper instruction.

Arithmetic Operators : Pop two operands off the compiler operand stack and make sure they're loaded into registers and emit the operation with three registers (e.g., if the operation is "+" emit "`add <Rd>, <Rs>, <Rs>`"). Such an operation needs a "temporary" variable associated with the result; an easy way to do this is to let the temporary be an anonymous `Symbol` object that is **not** placed into the symbol table, but is assigned to a register; the temporary is then pushed onto the operand stack as the result for any subsequent operations.

If operands need to be loaded into a register, the load operation depends on the kind of operand:

local variable : Emit "`ldr <Rd>, [fp, <offset>]`" to load the variable from its stack-relative memory location into a register. `<offset>` will be a multiple of 4 bytes.

int constant Emit "`ldr <Rd>, =<value>`" to load the value of the operand into a register. Note that we do not use `mov` and `#0` here because ARM puts a very small size limit on immediate values and we want any arbitrary 32-bit value.

string constant Emit "`ldr <Rd>, =<id>`" to load the *location* (i.e., starting address) of the constant string into a register.

Logical Operators : Pop two operands off the compiler operand stack, load into a register as above if necessary, and emit "`cmp <Rs>, <Rs>`". Peek at the *end* label from the compiler label stack and emit a branch that is the **inverse** function of the stated test; e.g., if the operation is "<" then emit "`bge <label>`". It may be useful to place the code to be emitted into the token itself as part of the definition of the token.

<***sign***> Pop an operation off the compiler operator stack which should be a sign. If it's negation, then we can negate the value by popping the top of the compiler operand stack and emitting "`neg <Rd>, <Rs>`" to negate the number. We also need to create a "temporary" symbol associated with the destination register to push onto the operand stack for subsequent operations; note that this negated value should not be conflated with the non-negated value (i.e. if `<Rd>` and `<Rs>` are the same, you can not let the original symbol think it's still held in the register!)

<***declare***> Pop the identifier off the compiler operand stack and add it to the symbol table along with information about the stack-local variable to which this variable has been assigned (i.e., assign it a scope-relative number such as 1, 2, ...). Emit code to adjust the stack-pointer to make room for the variable: "`add sp, sp, #-4`" where the offset is variable-number * 4 (could combine multiple adjustments into one operation if you keep track on a per-scope basis and only do this action once).

<***print_printf***> Save the current state of r0–r3 and the lr by emitting "`push {r0-r3,lr}`"; pop the operand off the compiler operand stack and make sure it's in a register and emit "`mov r1, <Rs>`" followed by emit "`bl printf`" and restore r0–r3 and lr with "`pop {r0-r3,lr}`"

<***print_sfmt***> Emit "`ldr r0, =<string-format-name>`"

<***print_ifmt***> Emit "`ldr r0, =<int-format-name>`"

<***gen_labels***> Generate 2 new labels and push them onto compiler *begin* and *end* label stacks (or just push a label number onto a single label stack for beginN: and endN: labels)

<***pop_labels***> Pop the label stack(s).

<***goto_begin***> Emit "`b <label>`" where label is the label on top of the *begin* label stack.

<***goto_end***> Emit "`b <label>`" where label is the label on top of the *end* label stack.

<***begin_label***> Peek at label on the *begin* label stack and emit "`<label>:`"

<***end_label***> Peek at label on the *end* label stack and emit "`<label>:`".