MC97 Homework 2

Due March 15, 1996; Max Hailperin

1. Make a syntax-directed definition out of the expression grammar

$$\begin{aligned} E &\rightarrow E - T \mid T \\ T &\rightarrow (E) \mid \mathbf{id} \mid \mathbf{num} \end{aligned}$$

so that E and T have an attribute val, which is a "value specifier" indicating either the actual value of the expression, if it can be determined at compile time, or the location where the value will be stored at runtime.

Each value specifier can be a constant, a register assigned to a variable, or a temporary register. Assume you have available the following functions (where `val` is the type for value specifiers):

```
val makeConst(int value);
val makeVarReg(int register);
val makeTempReg(void); /* allocates a temporary register */
int isConst(val);
int isReg(val);  /* true of both variable and temporary registers */
int isTempReg(val);
int value(val); /* legal only if isConst returns true */
const char *regName(val); /* only legal if isReg returns true */
void deallocateTempReg(val); /* legal only if isTempReg returns true */
```

Assume also that the terminals **id** and **num** come equipt with integer attributes reg and $value$ respectively.

Your goal is to model the simple optimizations your compiler will be doing. Therefore, your syntax-directed definition should reflect the following assumptions:

- Variables are never copied from their "home register" to a temporary register. (The home register can be directly used instead.)
- Operators with both operands available as compile-time constants are evaluated at compile time, producing a constant result.
- When only one operand of an operator is constant, the constant is loaded into a temporary register if and only if it is the left operand.
- One `deallocateTempReg` is done for each operand that is in a temporary register.
- The result of a run-time computation is always generated in a temporary register allocated for the purpose, at least in this problem. (We'll consider "targeting" the results in our next problem.)

(a) Write the syntax-directed definition requested above. Note that you needn't do any actual code generation or other work not involved in computing the val attributes. Calls to `deallocateTempReg` *are* involved, in that they presumably influence the result of later `makeTempReg` calls.

(b) Are the val attributes synthesized or inherited?

(c) Is your syntax-directed definition S-attributed? L-attributed? Both? Neither?

(d) Can your definition be used in LL(1) parsing? LR(1)? Explain.

2. The main optimization we are going to do that is not addressed by problem one (or a simple variation on problem one) is "targeting." Targeting means arranging that the final operation producing the overall value of the right-hand side of an assignment puts its result directly into the destination location, rather than into a temporary register. (Similarly, the value of an expression to be printed should be computed directly into `$a0`.) In this problem we will consider how targeting can be expressed using syntax-directed definitions, and the difficulties presented for actual implementation.

   (a) Consider the following syntax-directed definition:

PRODUCTION	SEMANTIC RULES
$S \rightarrow \mathbf{id} = E\,;$	$E.target := \mathtt{makeVarReg}(\mathbf{id}.reg)$
$E \rightarrow T$	$T.target := E.target$
$E \rightarrow E_1 - T$	$E_1.target := \mathtt{makeTempReg}()$
	$T.target := \mathtt{makeTempReg}()$
$T \rightarrow (\,E\,)$	$E.target := T.target$
$T \rightarrow \mathbf{id}$	

   i. Is the underlying grammar LR(1)?

   ii. Are the attributes $E.target$ and $T.target$ synthesized or inherited?

   iii. Is the definition L-attributed? Explain.

   iv. Does this mean the attribute values can be computed during an LR(1) parse? Explain.

   (b) Perhaps we'd have better luck if we used an underlying grammar that was LL(1). After all, *any* L-attributed definition based on an LL(1) grammar can be used in LR(1) parsing. Consider the following definition:

PRODUCTION	SEMANTIC RULES
$S \rightarrow \mathbf{id} = E\,;$	$E.target := \mathtt{makeVarReg}(\mathbf{id}.reg)$
$E \rightarrow T\,E'$	$E'.target := E.target$
	$T.target :=$ if $E'.empty$ then $E.target$ else $\mathtt{makeTempReg}()$
$E' \rightarrow \epsilon$	$E'.empty := True$
$E' \rightarrow -\,T\,E'_1$	$E'.empty := False$
	$T.target := \mathtt{makeTempReg}()$
	$E'_1.target := E'.target$
$T \rightarrow (\,E\,)$	$E.target := T.target$
$T \rightarrow \mathbf{id}$	

   i. Is the underlying grammar LL(1)?

   ii. For each attribute, is it synthesized or inherited?

   iii. Is the definition L-attributed? Is it useful for LR(1) parsing? Explain.

(c) Perhaps some third syntax-directed definition, other than those of parts 2a and 2b, will allow us to do targeting during LR(1) parsing. To evaluate this possibility, consider the two statements `a=(b-c);` and `a=(b-c)-d;`. In both cases, there will be a point where we reduce the string `b-c` to E. (Actually, we may be reducing to E a sentential form that derives `b-c`, e.g. $E{-}T$.) At the moment when we do that reduction to E, we want to know the corresponding $E.target$ attribute.

i. At the time of this reduction, what will the lookahead symbol be for each of the two examples?

ii. At this point, is there any way for the LR(1) parser to distinguish which of the two examples it is parsing?

iii. For each example, what is the correct value of $E.target$ for this reduction? Are they the same?

iv. What does this suggest for our hopes of doing targeting during LR(1) parsing?

(d) Of course, if we can reformulate targeting in terms of an S-attributed LR(1) grammar, then we will have no problems with LR(1) parsing. To do this, we need to find a way to achieve targeting using only synthesized attributes. Below is a syntax-directed definition based on the same grammar as was used in part 2a, but which computes *val* attributes as in problem 1. However, we use one additional kind of value specifier, namely a "deferred operation." A deferred operation consists of an operation name and two value specifiers for the operands. The idea is that when we reduce $E{-}T$ to E (with at least one operand not a constant), we don't emit any code but instead produce a deferred operation specifier. Only when the resulting E in turn participates in a reduction by $S \to \mathbf{id}{=}E;$ or $E \to E{-}T$ will we actually emit the operation, by which time the target is known.

Production	Semantic Rules
$S \to \mathbf{id} = E\,;$	`moveToTarget`(`makeVarReg`($\mathbf{id}.reg$), $E.val$)
$E \to T$	$E.val := T.val$
$E \to E' - T'$	$E.val :=$ `makeDeferred`(`'-'`, $E'.val, T'.val$)
$T \to (\,E\,)$	$T.val := E.val$
$T \to \mathbf{id}$	$T.val :=$ `makeVarReg`($\mathbf{id}.reg$)
$E' \to E$	$E'.val :=$ `undeferIfDeferred`($E.val$)
$T' \to T$	$T'.val :=$ `undeferIfDeferred`($T.val$)

We are assuming the existence of procedures `makeDeferred`, `undeferIfDeferred`, and `moveToTarget`. The first of these creates a "deferred operation" value specifier, while the other two can each consume one of these specifiers. (That is, the latter two operations both undefer their argument if it is deferred.)

Prove that during an LR(1) parse based on this definition, there will be at most one deferred operation in existence at any one time (i.e., created but not yet undeferred). (This result legitimizes using statically allocated storage to hold the three components of the deferred operations.)