Лабораторная работа № 3. «Компиляция базового императивного языка»

Коновалов А.В.

8 апреля 2023 г.

1 Цель работы

Целью данной работы является освоение ознакомление к компиляцией базовых конструкций императивного языка программирования: функциями, операторами, выражениями.

2 Задание

Требуется написать компилятор базовой версии НУИЯПа, рассмотренной в лекции, и расширить его как минимум тремя средствами (на выбор студента) из списка ниже.

Интерфейс командной строки компилятора:

```
refgo MyCompiler+LibraryEx source.txt dest.txt
```

где `source.txt` — исходный текст на НУИЯПе, `dest.txt` — сгенерированный код на языке модельного ассемблера. Исходный текст должен загружаться при помощи `LoadExpr`.

Реализовывать поддержку битовых операций в константных выражениях не обязательно, т.к. средства битовой арифметики в Рефале-5 отсутствуют, а их имитация не входит в цель лабораторной работы.

Для каждого из добавленных средств нужно написать пример кода, демонстрирующий его работу.

Допустимо выполнять лабораторную работу и не на Рефале-5. В этом случае нужно будет самостоятельно спроектировать структуры данных для представления синтаксического дерева НУИЯПа на выбранном языке программирования (например, data-классы на Python, case-классы на Scala и т.д.).

2.1 Варианты заданий

1. Оператор множественного ветвления `switch`:

   ```
   t.Statement ::= …
     | (switch t.Expr e.Branches t.DefaultBranch)
   e.Branches ::= t.Branch*
   t.DefaultBranch ::= t.Branch
   t.Branch ::= (e.Code)
   ```

   Пусть веток кода `|e.Branches|` n штук. Тогда, если значение выражения `t.Expr` лежит в диапазоне от 0 до *n* – 1 включительно, переход осуществляется на ветку с соответствующим номером. Если значение выражения не входит в этот диапазон, переход осуществляется на ветку по умолчанию `t.DefaultBranch`.

   Для ветвей должен быть сгенерирован массив — если значение выражения является допустимым индексом, адрес перехода должен извлекаться из массива, иначе — переход на ветвь по умолчанию.

   ```
   [[ выражение для t.Expr ]]
   DUP <метка по умолчанию> JLT
   DUP <n> CMP <ветка по умолчанию> JGE
   <массив меток> ADD READ JMP
   :<массив меток>
   <L1> <L2> … <Ln>
   :<L1> [[ код ветки 1 ]] <выход> JMP
   …
   :<Ln> [[ код ветки n ]] <выход> JMP
   :<ветка по умолчанию>
   DROP [[ код ветки по умолчанию ]]
   :<выход>
   ```

2. Определение перечислений:

   ```
   t.Definition ::= …
     | (union s.Name t.ConstExpr*)
   ```

   Определяет константу `s.Name` со значением, равным наибольшему из значений константных выражений.

3. Оператор блока определяет локальные переменные:

   ```
   t.Statement ::= …
     | (block t.LocalVars e.Code)
   ```

   Область видимости локальных переменных ограничена этим блоком. При генерации кода текст для `e.Code` окружается инструкциями PUSHN и DROPN, резервирующими место локальных переменных на стеке.

4. Тернарная операция Си:

   ```
   t.Expr ::= …
     | (t.Cond "?" t.OnTrue ":" t.OnFalse)
   ```

5. Цикл `for` как в Паскале:

   ```
   t.Statement ::= …
     | (for t.Var "=" t.Start to t.End do e.Code)
   t.Var, t.Start, t.End ::= t.Expr
   ```

   Параметр `t.Var` определяет адрес переменной, являющейся параметром цикла (как правило, это будет просто именем переменной), значения `t.Start` и `t.End` определяют диапазон изменения значения этой переменной (включительно). Значение счётчика цикла на каждой итерации увеличивается на 1, поэтому если начальное значение больше конечного, цикл не должен выполниться ни разу.

6. Цикл `for` как в Си:

   ```
   t.Statement ::= …
     | (for (t.Init t.BoolExpr t.Step) e.Code)
   t.Init, t.Step ::= 0 | (t.Expr ";" t.Expr)
   ```

   Действие `t.Init` выполняется до первой итерации цикла, действие `t.Step` — после каждой из итераций. Действие может быть как оператором присваивания, так и макроцифрой 0, означающей, что действие отсутствует.

7. Поддержка процедур — функций без возвращаемого значения:

   ```
   t.Definition ::= …
     | (procedure s.Name (s.Name*) t.LocalVars? e.Code)
   t.Statement ::= …
     | (pcall t.Expr t.Expr*)
     | (preturn)
   ```

8. Цикл с постусловием:

   ```
   t.Statement ::= …
     | (repeat e.Code until t.BoolExpr)
   ```

   В отличие от цикла `while`, где условие проверяется перед каждой итерацией (и если перед первой итерацией оно ложно — цикл не выполнится ни разу), цикл `repeat` проверяет условие после итерации, благодаря чему выполняется как минимум один раз. Условие после `until` — условие выхода из цикла, т.е. цикл продолжается до тех пор, пока условие остаётся ложным.

   Семантически цикл эквивалентен следующему коду на Python:

   ```
   while True:
       …e.Code…
       if …t.BoolExpr…:
           break
   ```

9. Операторы `break` и `continue`:

   ```
   t.Statement ::= …
     | (break)
     | (continue)
   ```

   Оператор `break` осуществляет выход из цикла, `continue` — к началу следующей итерации. В случае реализации циклов `for` и/или `repeat`, операторы `break` и `continue` должны поддерживаться и в них.

10. Оператор выделения памяти на стеке произвольного размера:

    ```
    t.Statement ::= …
      | (alloca t.Expr t.Expr)
    ```

    Оператор резервирует на стеке место размером, описанным вторым параметром и присваивает его адрес слову с адресом первого параметра: `(alloca some_variable (n "*" 5))`:

    ```
    [[ код для выражения второго параметра ]]
    PUSHN
    GETSP
    [[ код для выражения первого параметра ]]
    SWAP WRITE
    ```

11. Функции с переменным числом параметров (variadic functions):

    ```
    t.Definition ::= …
      | (vfunction s.Name (s.Name*) t.LocalVars? e.Code)
    t.Statement ::= …
      | (vcall t.Expr t.Expr*)
      | (vreturn t.Expr)
    t.Expr ::= …
      | (vcall t.Expr t.Expr*)
    ```

    В отличие от обычных функций, параметры со стека удаляет не вызываемый код, а вызывающий. Компиляция вызова функции может выглядет так:

    ```
    [[ код для параметра N ]]
    …
    [[ код для параметра 1 ]]
    [[ код для имени функции ]]
    CALL
    ; на вершине стека лежит возвращаемое значение, под ним параметры
    SWAP DROP … SWAP DROP ; N раз — удаляем параметры
    ```

    вместо генерации 2 · *N* инструкций SWAP DROP можно выполнить присваивание и DROPN (сами подумайте, как).

12. Операция присваивания, т.е. `(t.Expr ":=" t.Expr)` может встречаться в выражениях. Значением операции присваивания должно быть присвоенное значение.

13. Сокращённые операторы присваивания `(t.Expr s.BinOp ":=" t.Expr)`.

14. Операции инкремента и декремента в стиле Си: `("++" t.Expr)`, `(t.Expr "++")`, `("--" t.Expr)`, `(t.Expr "--")`. Функции принимают адрес и возвращают значение: функции пре*крементa возвращают *крементированное значение, функции пост*крементa — до модификации.

15. Операция блока внутри выражения (`let t.LocalVars e.Code t.Expr`).

16. Оптимизация хвостовой рекурсии:

    ```
    t.Statement ::= …
      | (return-tail-call t.Expr t.Expr*)
    ```

    Семантически эквивалентно (`return (call t.Func e.Args)`), но новый фрейм стека не создаёт.

    Решение этого задания засчитывается за два.

17. Добавить в компилятор два варианта генерации логических выражений: наивный и оптимальный. Переключение между ними должно осуществляться параметром командной строки `+gen-bool-expr=…`:

    ```
    refgo MyCompiler+LibraryEx +gen-bool-expr=naive source.txt dest.txt
    refgo MyCompiler+LibraryEx +gen-bool-expr=fast source.txt dest.txt
    ```

18. Добавить в компилятор оптимизацию свёртки констант: если выражение `t.Expr` содержит только значения, известные во время компиляции (макроцифры, имена констант, включая размеры структур и смещения полей), то его значение должно вычисляться во время компиляции.

    Оптимизация должна включаться параметром командной строки `+opt-const-fold`:

    ```
    refgo MyCompiler+LibraryEx +opt-const-fold source.txt dest.txt
    ```

    См. также https://ru.wikipedia.org/wiki/Свёртка_констант.

19. Добавить в компилятор оптимизацию распространения констант: если в последовательности операторов есть присвоение константы локальной переменной

    ```
    e.Code-B (s.LocalVarName "=" s.NUMBER) e.Code-E
    ```

    или

    ```
    e.Code-B (s.LocalVarName "=" ("-" s.NUMBER)) e.Code-E
    ```

    и в `e.Code-E` переменная `s.LocalVarName` используется только «для чтения» (т.е. только как операнд операции разыменования — `(i s.LocalVarName)`), то все операции разыменования сделать заменить этим константным значением.

    Оптимизация должна включаться параметром командной строки `+opt-const-prop`:

    ```
    refgo MyCompiler+LibraryEx +opt-const-prop source.txt dest.txt
    ```

    См. также https://ru.wikipedia.org/wiki/Свёртка_констант.

3 Отчёт по лабораторной работе

Отчёт выполняется в разметке Markdown по следующему шаблону:

```
% Лабораторная работа № 3. «Компиляция базового императивного языка»
% 8 апреля 2023 г.
% Вася Пупкин, ИУ9-63Б
# Цель работы
<перепишите цель работы из задания>
# Индивидуальный вариант
<перепишите выбранные расширения языка>
# Реализация и тестирование

xxxxx

# Вывод
<напиите, чему научились>
```

В отчёте приведён лишь необходимый минимум. Можно писать больше и интереснее — интересные и вдумчивые отчёты поощряются дополнительным баллом.

Шаблон отчёта

Ваш отчёт будет конвертирован в PDF при помощи pandoc следующей командой:

```
pandoc \
  --pdf-engine=xelatex \
  -V 'mainfont:Liberation Serif' \
  -V 'monofont:Liberation Mono' \
  "$SOURCE" -o "$PDF"
```

Язык реализации: Markdown

Код решения

Из файла

Отправить