Разработка стекового калькулятора с использованием виртуальной машины

Матвей Андриенко, 10 класс

Научный руководитель: И.Р. Дединский, МФТИ

0. Введение

Работа над проектом началась по заказу одного из московских вузов. Требовалось создание языка программирования и виртуальной машины для исполнения кода на этом языке, а также предоставление всех сопутствующих средств разработки, таких как отладчик и декомпилятор. Планировалось применение программного комплекса при проведении олимпиад по программированию: использование единого языка программирования и средств разработки ставило всех участников олимпиады в равное положение, а также значительно упрощало работу оргкомитета. Основными требованиями, предъявляемыми к разрабатываемому продукту, являлись:

1. Высокая скорость исполнения кода
2. Наличие системы информирования пользователя как об ошибках компиляции, так и времени исполнения
3. Наличие отладчика

В качестве языка программирования был выбран язык C++, как наиболее подходящий для низкоуровневого взаимодействия с памятью и работы с бинарными файлами.

1. Интерпретатор и архитектура стекового процессора

Основным компонентом разработанного программного комплекса является интерпретатор. Интерпретатор последовательно исполняет все инструкции из предоставленного байт-кода, одну за другой. Некоторые инструкции языка (например, *goto*) позволяют «подменить» адрес текущей команды, что вызовет пропуск участка кода или его повторное исполнение (рис. 1).

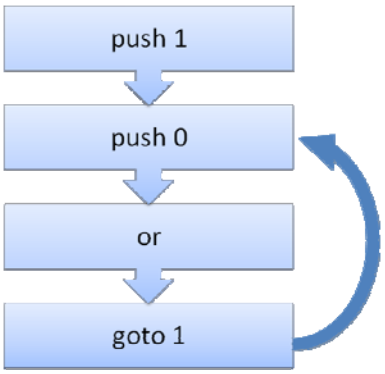

Рисунок 1. Исполнение кода интерпретатором

Память интерпретатора разделена на три изолированные друг от друга части: *основной стек*, *дополнительный буфер* и *стек адресов возврата*. Кроме того, в памяти хранятся несколько *флагов*: адрес текущей команды, флаг аварийного завершения исполнения программы и флаг, переводящий интерпретатор в режим отладки (рис. 2).

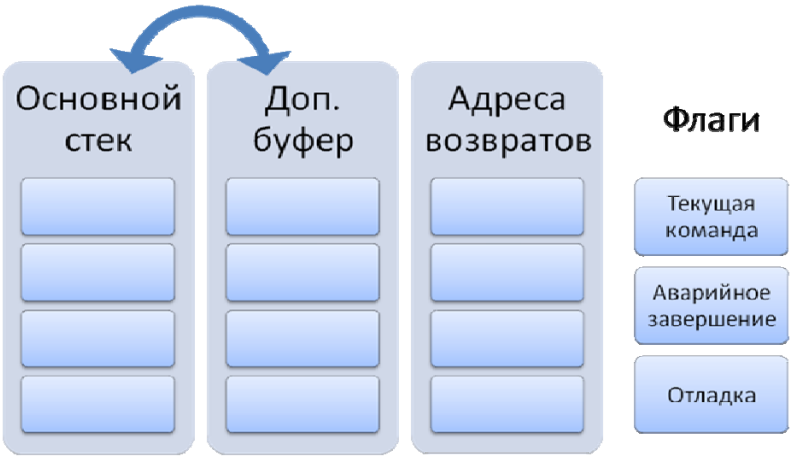

Рисунок 2. Память стекового процессора

Изоляция этих участков памяти друг от друга повышает надежность системы. Благодаря ней, к примеру, при переполнении одного из стеков не произойдет повреждение другого.

Все значения, необходимые пользователю для выполнения вычислений, помещаются в основной стек. Так как стек позволяет получить доступ только к последнему сохраненному в нем элементу, а пользователю может потребоваться значение, которое было создано ранее, был предусмотрен *дополнительный буфер*, также представленный в виде стека. Пользователь может переместить значение из основного стека в дополнительный буфер, а потом вернуть ее обратно.

Участок кода можно выделить в *функцию*. После выполнения функции интерпретатор должен продолжить выполнение кода с момента вызова, поэтому при вызове функции с помощью инструкции *call* будет сохранен *адрес возврата*. Все адреса возвратов также помещаются в соответствующий стек. Если в коде встретится нескольких вложенных вызовов функций, последовательный переход по сохраненным адресам возвратов позволит «распутать» цепочку вызовов (рис. 3).

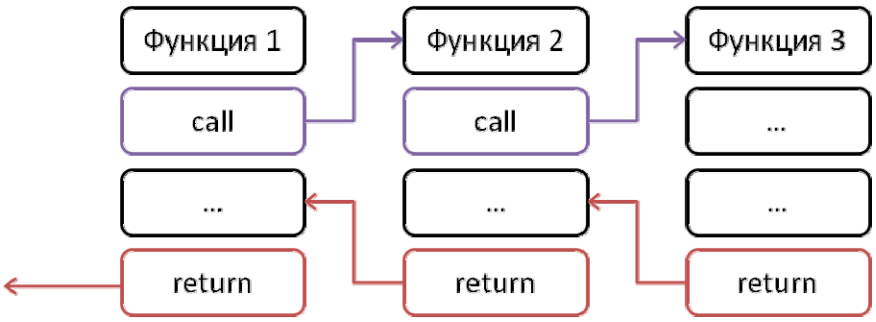

Рисунок 3. Переход по множественным адресам возврата

2. Компилятор и предварительная обработка кода препроцессором

Интерпретатор для идентификации инструкций использует не их текстовые названия, такие как *debug*, *save*, *dupl*, а сопоставленные им числовые коды. Кроме того, исполняемый интерпретатором байт-код хранится в бинарном файле. Это позволяет значительно повысить скорость исполнения кода, так как операции сравнения строк, чтения из файла, последующий разбор чисел, записанных в текстовом виде, отнимают много времени. Для автоматической трансляции кода на языке стекового калькулятора в байт-код был разработан компилятор (рис. 4).

Компилятор добавляет в язык несколько конструкций, облегчающих написание и понимание кода: во время предварительной обработки кода из него удаляются комментарии, и формируется словарь меток. Метки позволяет создавать синонимы для номеров инструкций. Например, объявив в начале функции метку *MY_FUNCTION*, вы сможете вызывать функцию по имени, а не по адресу.

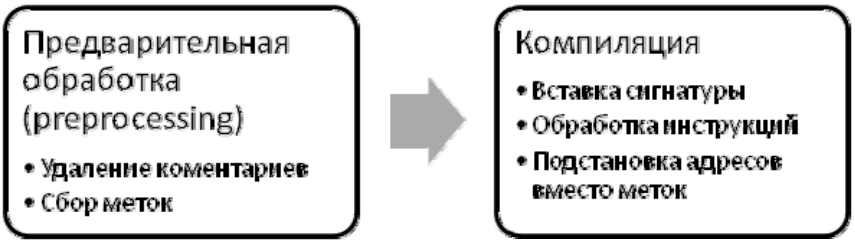

Рисунок 4. Процесс компиляции

Компилятор может также выполнять обратную задачу: переводить байт-код в понятный человеку код на языке стекового калькулятора (процесс *декомпиляции*). Декомпиляция может использоваться для исправления ошибки в откомпилированном коде, исходные файлы которого утеряны. Конечно, при декомпиляции не восстанавливаются комментарии и названия меток. Декомпиляция активно используется в процессе отладки.

3. Отладчик

Для выявления ошибок времени исполнения и логических ошибок может быть использован *отладчик*.

Отладчик тесно интегрирован с интерпретатором и имеет доступ к потоку, из которого читается исполняемый файл, и содержимому стеков в момент исполнения. Инструкция *debug* переводит интерпретатор в режим отладки. В этом режиме перед выполнением каждой инструкции интерпретатор передает управление отладчику. Отладчик, в зависимости от желания пользователя, может продолжить построчное выполнение программы, вывести интерпретатор из режима отладки, отобразить содержимое основного стека, регистра или стека адресов возврата, показать декомпилированный код программы или аварийно завершить выполнение программы, установив флаг *halt* (рис. 5).

```
Reached breakpoint. Avaible actions
d - shows contents of data stacks;
a - shows return adresses;
n - continue to next breakpoint;
s - one step forward;
c - view source code (decompiled)
x - stop running application.
Enter debugger command> c
  goto 4
  debug
  push 10
  return
  input
  dupl
  debug
* *
  call 1
  /
  push 15
  -
Enter debugger command> d
Data stack:
17
17
Saved data stack:
Empty
```

Рисунок 5. Пример вывода отладчика

4. Результат работы и планы развития

В результате работы был создан программный комплекс, состоящий из эмулятора стекового процессора, интерпретирующего байт-код, компилятора и отладчика. Разработанный эмулятор обладает высокой скоростью выполнения кода и устойчивостью к ошибочному коду (рис. 6).

Планируется расширение возможностей языка стекового калькулятора, введение в него как новых инструкций, так и директив, подобных директиве include языка C++.

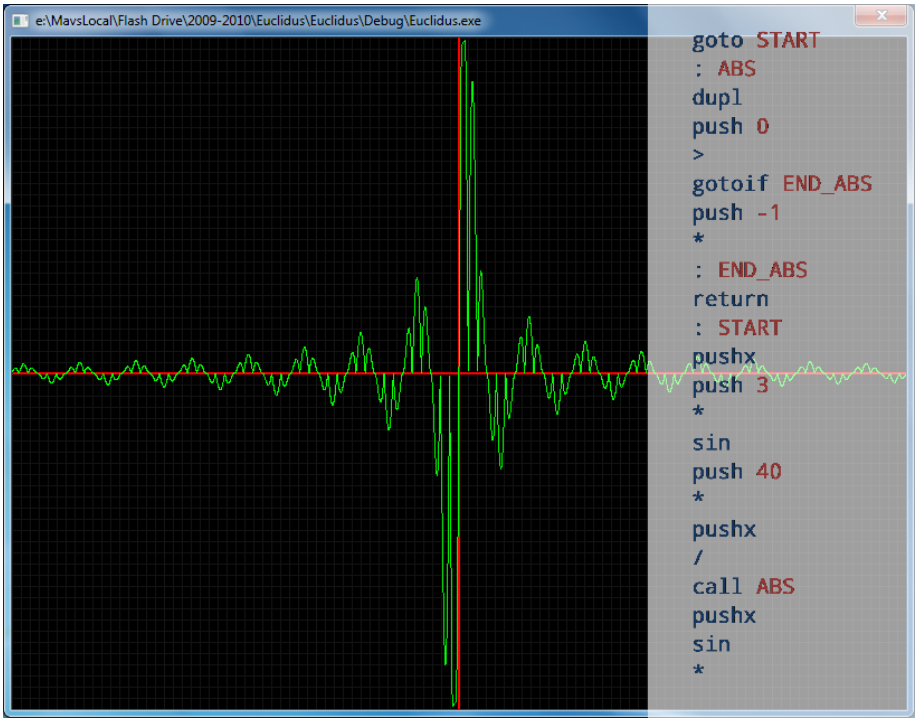

Рисунок 6. Работа демонстрационного приложения, выполняющего построение графика функции, с примером кода