Lab 1. Part II. Lexer for cool. C++ and GNU.

I. Gorban

Mipt (Ilab), 10.09.2018

Что вы здесь не увидите

1. Здесь не будет рассмотрен хороший стиль C++ (для этого смотрите - лекции Константина Владимирова.
2. Сегодня не будет рассмотренна внутренняя архитектура компилятора cool (будет позже).
3. Полный справочник по C++ - настоятельно рекомендован к изучению (и дополнению) `https://cppreference.com/` - есть русская версия.
4. Точных и формальных определений. Здесь будут даны определения "as is". Для того, чтобы дать хотя бы примерное представление о предмете разговора.
5. Ссылки с описанием

`https://s3-us-west-1.amazonaws.com/prod-edx/Compilers/ProgrammingAssignments/PA1.pdf`

`https://lagunita.stanford.edu/c4x/Engineering/Compilers/asset/cool_manual.pdf`

`https://lagunita.stanford.edu/c4x/Engineering/Compilers/asset/cool-tour.pdf`

Flex

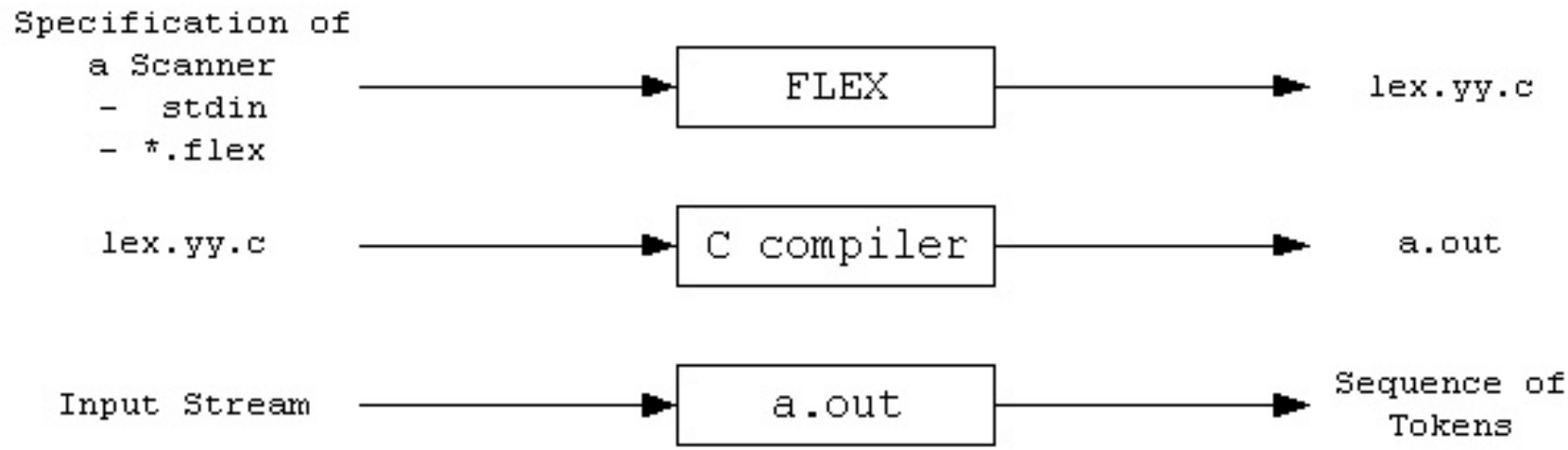

Figure: Последовательность работы flex.

Здесь **Flex** - программа для генерации лексических анализаторов, обычно используемая совместно с генератором синтаксических анализаторов. **C compiler** - gcc.

COOL grammar

Грамматика COOL :

```
program ::= [[class; ]]+
  class ::= class TYPE [inherits TYPE] { [[feature; ]]}
feature ::= ID( [ formal [[, formal]] ] ) : TYPE { expr }
          | ID : TYPE [ <- expr ]
 formal ::= ID : TYPE
   expr ::= ID <- expr
          | expr[@TYPE].ID( [ expr [[, expr]] ] )
          | ID( [ expr [[, expr]] ] )
          | if expr then expr else expr fi
          | while expr loop expr pool
          | { [[expr; ]]+}
          | let ID : TYPE [ <- expr ] [[, ID : TYPE [ <- expr ]]] in expr
          | case expr of [[ID : TYPE => expr; ]]+esac
          | new TYPE
          | isvoid expr
          | expr (+|-|*|/) expr
          | ~expr
          | expr < expr
          | expr <= expr
          | expr = expr
          | not expr
          | (expr)
          | ID
          | integer
          | string
          | true
          | false
```

COOL lex example in work

Давайте рассмотрим лексемы "TYPEID", "OBJECTID", "INT_CONST" и "STR_CONST".
TYPEID - имя типа, такое как String, Int, Foo, Bar (по соглашению типами считаются все слова, начинающиеся с заглавной буквы).
OBJECTID - имя обьекта, например str, myint, i (по соглашению именами обьекта считаются все слова, начинающиеся с прописной буквы)
INT_CONST - константное значение числа.
STR_CONST - константное значение строки.
Чтобы сохранить информацию об имени типа - (например не хочется хранить 10 раз одно и то же имя), связка программ flex-bison работает через cool_yylval. Symbol - хранит указатель на элемент в таблице символов.

```
yylval.symbol = stringtable.add_string(yytext);
```

Теперь разберемся с исходным кодом.

Cool flex

Вот так выглядит реализация самого простого cool-компилятора:

```
$ cat mycoolc
#!/bin/csh -f
./lexer $* | ./parser $* | ./semant $* | ./cgen $*
```

То есть результат лексера - подается на вход парсеру, результат парсера на вход семантическому анализатору. Результат семантического анализатора на вход кодогенератора.
Пример токенов и лексем:

тип токена	примеры лексем	описание
num	257	число
id	Ident951	идентификатор
relop	<=	операция отношения
string	«Символ»	строчная постоянная

Шаблоны внутри COOL

Класс list находится в include/list.h. Класс StringTable

```
template <class T>
class List {
private:
  T *head;
  List<T>* tail;
public:
  List(T *h, List<T>* t = NULL): head(h), tail(t){ }

  T *hd() const        { return head; }
  List<T>* tl() const { return tail; }
};

template <class Elem>
class StringTable
{
protected:
    List<Elem> *tbl;    // a string table is a list
    int index;          // the current index
```

C++ Классы определения

Класс - это тип, определяемый пользователем, для которого определен ряд методов.
Методы класса - функции, описанные внутри тела класса.
Обьект класса - элемент, имеющий размер равный размеру класса, из которого можно вызывать методы класса.
Описание класса, это не точное определение, сюда можно навесить еще минимум 4 аттрибута ("-" играет роль определителя регекспа):

```
class class_name (– : ancestor_list –)–?
{
    (– (– public: | private: | protected: –)–?
            (– virtual –)? (– static –)? type
                Method_name((– variables –)–*) (–
                definition | ; –)
            (– static –)–? type variable_name (
                declaration)? ;
    –)*
}
```

C++ Конструктор, деструктор, статические члены и методы.

Конструктор - метод, который имеет название, совпадающее с именем класса, ничего не возвращает и логически выполняет роль инициализации обьекта. Может принимать аргументы. В случае, если аргумент - константный обьект того же класса - называется конструктором копирования
Деструктор - метод, противоположный конструктору, то есть логически выполняет роль очистки. Так же ничего не возвращает. Название соотвествует "~имя_класса", не имеет аргументов.
Статические члены и аттрибуты - это члены и аттрибуты, которые не привязанны к какому-либо обьекту, однако логически должны использоваться только в связке с ними (или совсем без конструирования).

C++ Виртуальные методы, таблица виртуальных функций, перегрузка операторов

Виртуальный метод - метод обьекта, имеющий смысл только в цепочке наследования. Такой метод будет вызываться у обьекта вне зависимости, какой указатель используется.

Offset	Method name(A)	Method name(B : public A)
0x0000	void A::foo()	void A::foo()
0x0004	virtual void foo(int)	virtual void foo(int)
0x0008	void foo(bool)	void foo(char)

(В таблице указанны отступы относительно начала таблицы для для каждого класса)

Перегрузка операторов - механизм, с помощью которого можно определить поведение обьектов класса при использование операторов "+","-","()","[]" и т.д.

C++ Шаблонные функции, шаблонные классы, инстанциация

Шаблонная функция - функция, некоторые типы аргументов или аргументов заданы как шаблоны.

```
template<typename T>
void f(T s) { std::cout << s << '\n'; }
int main()
{
        // instantiation here
    f<double>(1); f<>('a'); f(7); void (*ptr)(std::
        string) = f;
}
```

Шаблонные классы - полностью аналогичны функцям, только шаблонный параметр не ограничивается аргументами (то есть может использоваться шире).

Инстанциация - точка в коде, где происходит конкретизация обьекта или где явно указанно, какими типами будет конкретизирован шаблон.

GCC

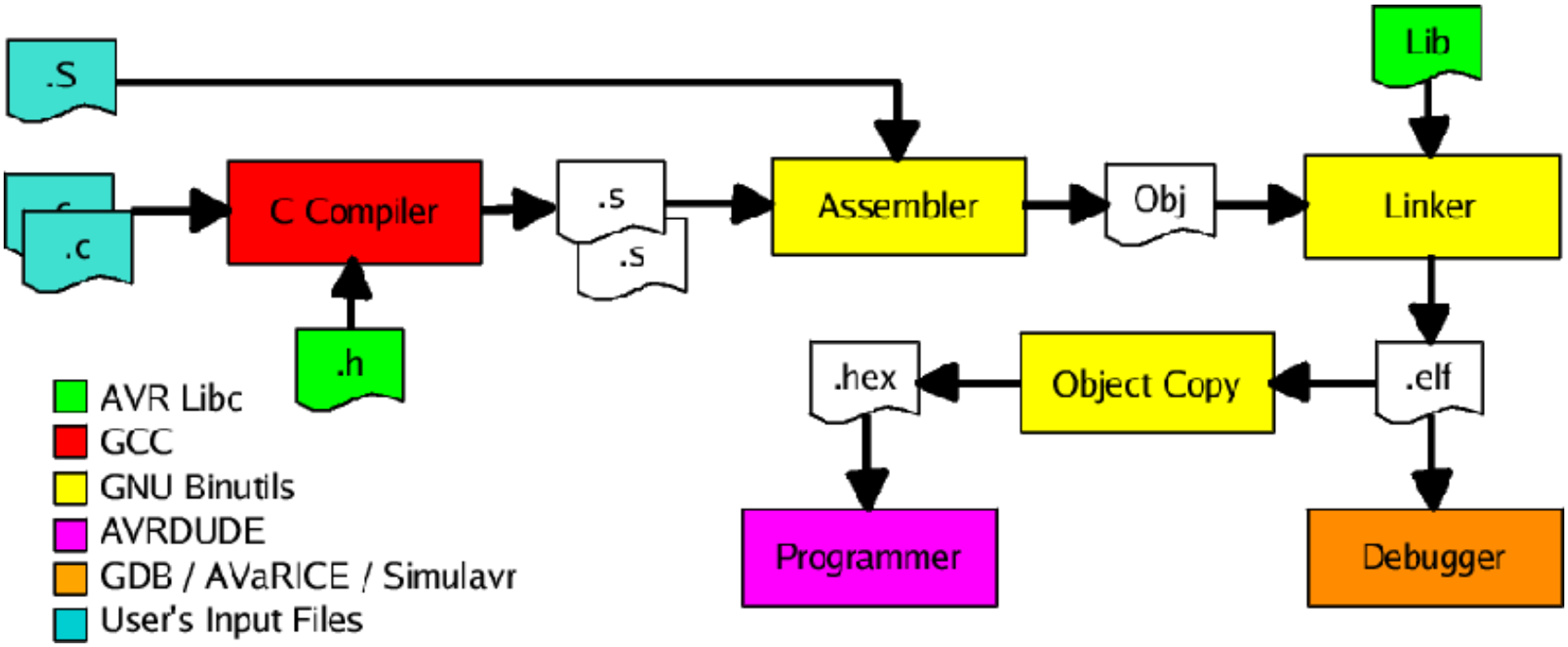

Figure: Фазы компиляции gcc без препроцессора.

GCC типы файлов

file.c - Исходный код на C, который должен быть препроцессирован.

file.i - Исходный код на C, который не должен быть препроцессирован.

file.s - Код на языке ассемблера

file.S / file.sx - Код на языке ассемблера, который должен быть препроцессирован.

file.cpp (cc / cp / cxx / CPP / c++ / C) - Исходный код на C++, который должен быть препроцессирован.

file.h - Заголовочный файл для языков C, C++, Objective-C или Objective-C++ для добавления на этапе препроцессинга.

file.hh (H / hp / hxx / hpp /HPP / h++ / tcc) - Заголовочные файлы на C++

GCC препроцессор, ассемблер

Рассматриваемый пример:

```
int f(int x)
{
        return 1 + x;
}
#define SQR(x) ((x)*(x))
unsigned sqr (int x) { return x*x; }
int main()
{
        int x = 8;
        unsigned t_slow = SQR(f(x));
        unsigned t_fast = sqr(f(x));
        return t_fast + t_slow;
}
```

GCC препроцессор, ассемблер

Для получения результата препроцессора:

```
gcc -E 01.cpp
```

Для получения ассемблерного файла :

```
gcc 01.cpp -S -o 01.S
# Demangle
c++filt _Z3sqri
```

Для запуска ассемблера:

```
as 01.S -o main.o
```

Для линковки (можно посмотреть опции вашего запуска, выполнив gcc 01.cpp -v):

```
ld -m elf_x86_64 -dynamic-linker /lib64/ld-linux-
    x86-64.so.2 /usr/lib/x86_64-linux-gnu/Scrt1.o
    /usr/lib/x86_64-linux-gnu/crti.o -L/usr/lib/gcc
    /x86_64-linux-gnu/8 main.o -lgcc -lc /usr/lib
    /gcc/x86_64-linux-gnu/8/crtendS.o /usr/lib/
    x86_64-linux-gnu/crtn.o
```

GCC binutils отладочная информация

Для чтения таблицы имён обьектного файла можно воспользоваться программой nm

```
nm a.out
```

Для поиска печатных символов (строк, имен функций) можно воспользоваться программой strings

```
strings a.out
```

Для получения информации о секциях, можно воспользоваться программой size

```
size a.out
```

Примеры секций:

- .text — содержит код программы
- .bss — неинициализированные переменные
- .data — инициализированные переменные
- .interp — путь к бинарному интерпретатору
- .init — выполняется до вызова точки входа

GCC binutils

Для получения ассемблера из обьектного файла существует функция Objdump

```
objdump -dx a.out > a.asm
```

Если необходимо узнать, какая строка соответвует

```
addr2line 0x1145 ./a.out
```

Для компиляции с дополнительными отладочными символами необходимо при компиляции подать опцию "-g" :

```
gcc -g 01.cpp
```

Для запуска отладчика необходимо подать программу, скомпилированную с отладочными символами на gdb

```
gdb ./a.out
# -> gdb information here
(gdb) help
# -> help information
(gdb) break main
Breakpoint 1 at 0x114d: file 01.cpp, line 13.
(gdb) run
```

Ссылки

Грамматика C:
https://www.lysator.liu.se/c/ANSI-C-grammar-l.html
https://www.lysator.liu.se/c/ANSI-C-grammar-y.html
Грамматика C++
http://www.nongnu.org/hcb/
Грамматика Rust
https://github.com/rust-lang/rust/tree/master/src/grammar
Курс C++ от Смаля (СПБГУ) https://stepik.org/lesson/555/
Список опций для отладки
https://gcc.gnu.org/onlinedocs/gcc/Debugging-Options.html
Лекции Константина Владимирова
https://sourceforge.net/projects/cpp-lects-rus/