Внимание! Текст задания изменён 29.04.2018 в 17:30 и 02.05.2018 а 14:30

Задание 16.1 Построение транслятора выражений на язык атомов

Для транслирующей грамматики выражений языка MiniC построить транслятор на язык атомов методом рекурсивного спуска. После списка атомов ваша программа должна печатать таблицу символов с вашими и временными переменными. Транслятор должен использовать лексический анализатор языка MiniC.

В этом упражнении **не требуется** реализовывать вызовы функций, поэтому атомы CALL, RET и PARAM, а также соответствующие правила грамматики (№30, 32-35) делать пока не нужно. Помимо этого есть небольшие расхождения с грамматикой, которые необходимо учитывать, а именно: функция alloc() пока не принимает никаких параметров, а вместо функции checkVar() вы будете использовать упрощенную функцию add.

Вход/выход: Входные данные считываются из файла, список атомов записывается в результирующий файл.

Пример работы программы

Выражение в файле prog.minic	Вывод в файл prog.atom

`b*b - 4*a*c`	`(MUL, 0, 0, 3)`
	`(MUL, '4', 1, 4)`
	`(MUL, 4, 2, 5)`
	`(SUB, 3, 5, 6)`
	
	`SYMBOL TABLE`
	`0 b`
	`1 a`
	`2 c`
	`3 temp1`
	`4 temp2`
	`5 temp3`
	`6 temp4`

План выполнения задания

Этап 1. Класс для таблицы строк

В файле StringTable.h объявите класс `StringTable` для хранения таблицы строк. Объект данного класса будет храниться внутри синтаксического анализатора.

Внутри класса объявите:

- защищенное поле `std::vector<std::string> _strings` для хранения записей таблицы;
- оператор `const std::string & operator [](const int index) const` для доступа к элементам таблицы по индексу;
- метод `int add(const std::string string)` – добавляет новый элемент в таблицу и возвращает его индекс; если такая строка уже имеется в таблице, то новая запись не добавляется, возвращается индекс старой.

Реализацию поместите в файл StringTable.cpp.

Реализуйте оператор вывода таблицы в поток.

Протестируйте класс с помощью модульных тестов.

Этап 2. Класс для таблицы символов

В файле SymbolTable.h объявите класс `SymbolTable` для хранения таблицы символов. Объект данного класса будет храниться внутри синтаксического анализатора. В этом задании записи таблицы символов будут хранить только имя символа, в следующем задании таблица символов будет расширена.

Внутри класса объявите:

- структуру `TableRecord` для записей таблицы; пока в ней будет одно поле `_name` типа `std::string` и оператор `bool operator == (const TableRecord & other) const` для тестов.
- защищенное поле `std::vector<TableRecord> records` для хранения записей таблицы;
- оператор `const TableRecord &operator [](const int index) const` для доступа к элементам таблицы по индексу;
- метод `int add(const std::string &name)` – добавляет новый элемент в таблицу и возвращает его индекс; если такая строка уже имеется в таблице, то новая запись не добавляется, возвращается индекс старой.

Реализацию поместите в файл SymbolTable.cpp.

Протестируйте класс с помощью модульных тестов.

Этап 3. Классы для операндов атомов

Теперь создадим классы для атомов и их операндов. Создайте файлы Atoms.h и Atoms.cpp.

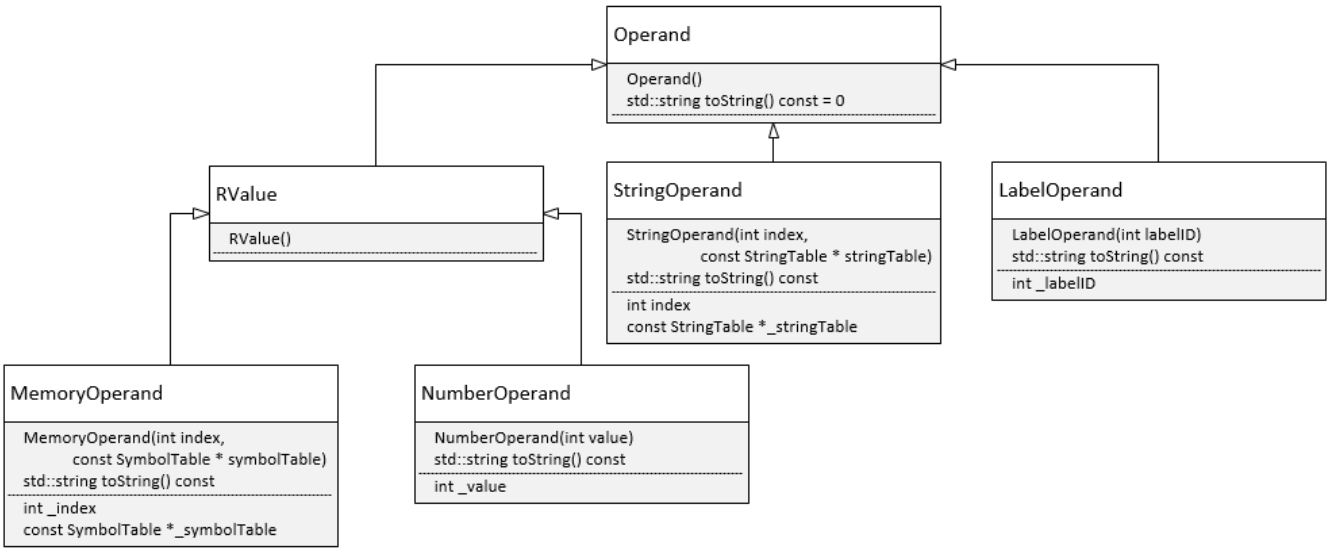

Рис. 1. Классы операндов атомов

Объявите в заголовочном файле классы, согласно иерархии, показанной на рисунке 1. Класс `Operand` является базовым классом для всех операндов. В этом классе объявлена чистая виртуальная функция `std::string toString() const`, которая будет переопределена в конкретных классах для формирования текстового представления операнда. Класс `RValue` объединяет операнды, которые могут выступать в качестве операндов математических операций. Эти два класса не хранят никакой информации и не реализуют никаких методов.

Конкретные классы-операнды представлены классами `MemoryOperand`, `NumberOperand`, `StringOperand` и `LabelOperand`. Эти операнды получают необходимые значения через аргументы конструкторов и сохраняют их в защищённых (`protected`) полях.

Обратите внимание, что классы для переменных и текстовых строк хранят указатели на таблицу символов или таблицу строк соответственно. Чтобы эти указатели можно было объявить, в начале файла Atoms.h сделайте предварительное объявление классов таблиц:

```
class StringTable;
class SymbolTable;
```

Заголовочные файлы StringTable.h и SymbolTable.h подключите в файле Atoms.cpp. Не включайте их в Atoms.h чтобы исключить циклическую зависимость на следующем этапе.

Для классов `MemoryOperand`, `NumberOperand`, `StringOperand` и `LabelOperand` реализуйте метод `std::string toString() const` для преобразования в строку и протестируйте их работу.

Этап 4. Создание операндов в таблицах символов и строк

Модифицируйте метод `add()` класса `SymbolTable` так чтобы он возвращал указатель `std::shared_ptr` на объект класса `MemoryOperand`, а метод `add()` класса `StringTable` – чтобы он возвращал указатель

`std::shared_ptr` на объект класса `StringOperand`. Для этого подключите заголовочный файл Atoms.h в файлы SymbolTable.h и StringTable.h.

Скорректируйте тесты и убедитесь в их работоспособности. Для этого может быть удобно реализовать оператор равенства (==) для классов `MemoryOperand` и `StringOperand`.

Этап 5. Классы для атомов

Теперь разработайте классы для самих атомов (а файлах Atoms.h и Atoms.cpp). Классы для атомов CALL, RET и PARAM будут реализованы в следующем задании.

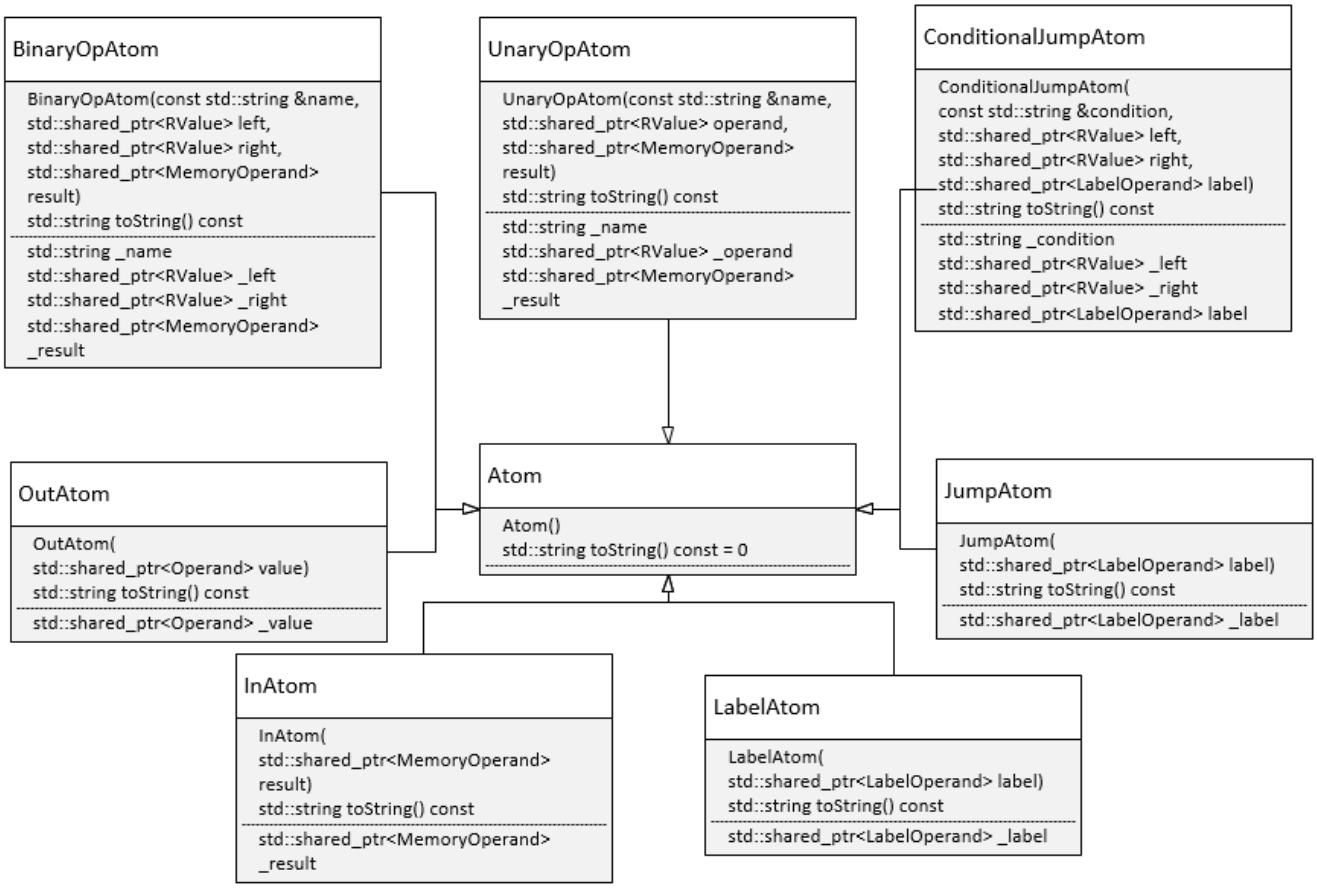

Рис. 2. Иерархия классов атомов

Иерархия классов атомов показана на рисунке 2. Эти классы получают на вход указатели `std::shared_ptr` на соответствующие типы операторов.

Класс `BinaryOpAtom` представляет все атомы для бинарных математических операций (ADD, SUB, MUL, DIV, AND, OR), конкретная операция, которую будет представлять объект этого класса, задаётся текстовой строкой `name`. Аналогично класс `UnaryOpAtom` представляет атомы NEG, NOT и MOV, а класс `ConditionalJumpAtom` – атомы EQ, NE, GT, LT, GE, LE.

Для всех классов кроме `Atom` реализуйте метод `std::string toString() const` для преобразования в строку. В классе `Atom` этот метод должен быть объявлен как чистая виртуальная функция. Проверьте работу классов с помощью модульных тестов.

Этап 6. Класс translator

1. Включите в проект файлы лексического анализатора, разработанного в предыдущем семестре.
2. В файлах Translator.h/Translator.cpp создайте класс `Translator`, содержащий следующие защищенные поля:
   a. вектор указателей `std::unique_ptr<Atom> _atoms`[1] для хранения результатов трансляции;
   b. таблица символов и таблица строк;
   c. лексический анализатор;

[1] Если не получится работать с `std::unique_ptr` используйте вместо него `std::shared_ptr`.

d. последняя прочитанная лексема `Token _currentLexem`.

3. Конструктор транслятора получает ссылку на поток с входной информацией и передаёт её лексическому анализатору.
4. Создайте метод транслятора `void printAtoms(std::ostream& stream)`, который выводит сохранённые атомы в поток.
5. Создайте метод транслятора `void generateAtom(std::unique_ptr<Atom> atom)` который бы добавлял переданный атом в список _atoms.
6. Протестируйте добавление атомов и печать списка.
7. Дополните класс таблицы символов `SymbolTable` методом `std::shared_ptr<MemoryOperand> alloc()`, который резервирует в таблице символов место для новой временной переменной и возвращает соответствующий операнд.
   В отличие от описания, приведенного в транслирующей грамматике, функция alloc() пока не принимает никаких параметров, так как работа с контекстом будет реализована в следующем упражнении.
   Протестируйте этот метод.
8. Создайте метод транслятора `std::shared_ptr<LabelOperand> newLabel()`, который при каждом вызове будет возвращать операнд метки с увеличивающимся каждый раз значением. Для этого сделайте поле, которое будет отслеживать номер последней использованной метки. Протестируйте этот метод.
9. Добавьте в транслятор методы `void syntaxError(const std::string &message)`, `void lexicalError(const std::string &message)` и класс для исключений. Эти методы будут вызываться при возникновении ошибок трансляции, и должны выкидывать исключение с переданным им текстом ошибки. Протестируйте эти методы.

Этап 7. Транслятор выражений

Добавьте к классу транслятора функции рекурсивного спуска[2] `E()`, `E7()`, `E7_()`, `E6()`, `E6_()`, `E5()`, `E5_()`, `E4()`, `E4_()`, `E3()`, `E3_()`, `E2()`, `E2_()`, `E1()`, `E1_()` в соответствии с транслирующей грамматикой. Если возникает лексическая или синтаксическая ошибка, транслятор должен вызывать функции `syntaxError()` или `lexicalError()` в зависимости от типа ошибки.

Реализация грамматики выражений в этом упражнении должна быть немного упрощена по сравнению с полной версией, которая будет реализована в следующем задании. Отличия упрощенного варианта следующие:

1) не нужно реализовывать правила 30, 32 … 35;
2) функция `alloc()` не принимает никаких параметров;
3) в правилах 27, 29, 31 вместо функции `checkVar()` используйте метод `add()` класса `SymbolTable`.

Для значений атрибутов грамматики используйте те же классы, унаследованные от Operand, что и для операндов атомов. Значения наследуемых атрибутов должны передаваться в качестве аргументов функций рекурсивного спуска, а значение синтезируемого (для тех нетерминалов, где он есть) - возвращаться самой функцией. В соответствии с грамматикой языка MiniC у каждого нетерминала может быть максимум один синтезируемый атрибут, поэтому такой способ его передачи ничему не противоречит.

Для передачи унаследованных атрибутов используйте `std::shared_ptr<…Operand>`. Конкретный тип указателя определите, проанализировав семантику правила. Например, если при реализации правила необходимо создать атом $\{OR\}_{prs}$, значит p и r должны быть `std::shared_ptr<RValue>`, а s - `std::shared_ptr<MemoryOperand>`.

Тип возврата функции рекурсивного спуска определяется по следующим правилам:

[2] Описания языка атомов и транслирующей грамматики выражений языка MiniC можно загрузить с сайта курса по адресу http://prog.tversu.ru/cs4.html.

- если у нетерминала есть синтезируемый атрибут, то функция возвращает значение типа `std::shared_ptr<…Operand>`, если все прошло успешно, и `nullptr`, если произошла ошибка трансляции.
- если у нетерминала нет синтезируемого атрибута, то функция возвращает значение типа `bool` - `true` если вход распознан и `false`, если произошла ошибка трансляции.

Рекомендованный порядок реализации транслятора:

1) начните с реализации правил, описывающих операции с высокими приоритетами, которые не требуют рекурсивных вызовов других функций (чтобы их можно было сразу тестировать, не реализовывая все остальные), например $E1_p \to num_{val}$;
2) выбрав правило, определите типы данных для входящих в него атрибутов и напишите определения функций, которые потребуются для реализации этого правила, если этих функций ещё нет;
3) придумайте минимальный фрагмент кода на языке MiniC, соответствующий этому правилу, и на его основе напишите тест для проверки работы функции;
4) добавьте реализацию выбранного правила в соответствующие функции (при этом не нужно сразу писать полную реализацию функции – реализуйте её только для того правила, которое было выбрано, остальные случаи будут добавлены позже);
5) проверьте работу функции с помощью теста, в случае необходимости исправьте ошибки; если в ходе работы выявите ошибки в работе функций, написанных на предыдущих этапах, добавьте такие случаи в тесты для функций с ошибками и исправьте их;
6) если реализованные правила позволяют, добавьте несколько более сложных тестов для этого же правила, и убедитесь, что они работают;
7) посмотрите на грамматику и выберите следующее правило, которое можно было бы достаточно быстро реализовать, и переходите к нему;
   обратите внимание, что в некоторых случаях может потребоваться заменить типы аргументов или возвращаемых значений у функций рекурсивного спуска: например, при реализации правила $E1_p \to num_{val}$ можно сначала решить, что для синтезируемого атрибута p в этом правиле подойдёт тип `std::shared_ptr<NumberOperand>`, однако позднее, когда вы дойдёте до правила $E1_p \to (E_q)$, станет понятно, что функция `E1()` должна возвращать значение типа `std::shared_ptr<RValue>`.

Завершив работу, ещё раз проверьте, что были реализованы все правила грамматики, и допишите несколько тестов, проверяющих разбор сложных выражений.

Пример фрагментов кода для функций `E7()` и `E7_()`:

```
std::shared_ptr<RValue> E7()
{
	std::shared_ptr<RValue> q = E6();
	if (!q)
	{
		syntaxError("...");
	}

	std::shared_ptr<RValue> p = E7_(q);
	if (!p)
	{
		syntaxError("...");
	}

	return p;
}

std::shared_ptr<RValue> E7_(std::shared_ptr<RValue> p)
{
	if (_currentLexem.type() == LexemType::error)
	{
		lexicalError("...");
	}

	// Правило №3
	if (_currentLexem.type() == LexemType::opor)
```

```
	{
		_currentLexem = _lexicalAnalyser.getNextLexem();

		std::shared_ptr<RValue> r = E6();
		if (!r)
		{
			syntaxError("...");
		}

		std::shared_ptr<MemoryOperand> s = _symbolTable.alloc();
		generateAtom(std::make_unique<BinaryOpAtom>("OR", p, r, s));
		std::shared_ptr<RValue> q = E7_(s);
		if (!q)
		{
			syntaxError("...");
		}
		return q;
	}

	// Правило №4
	return p;
}
```

Этап 8. Сообщения о лексических ошибках

Усовершенствуйте подсистему вывода информации о лексических ошибках. Исключение должно содержать информацию, позволяющую сформировать текстовую строку с тремя последними корректно считанными лексемами и последовательность символов, на которой сломался разбор.

Этап 9. Сообщения о синтаксических ошибках

Усовершенствуйте подсистему вывода информации о синтаксических ошибках. Исключение должно формировать сообщение об ошибке, включающее правило грамматики, при разборе которого случилась ошибка, три последние корректно считанные и текущую лексемы, а также тип ошибки:

- кончились лексемы до окончания трансляции,
- закончилась трансляция, а лексемы еще имеются,
- текущая лексема не соответствует ожидаемой (необходимо выдать ожидаемую лексему).

Оценка задания 16.1

1. Классы таблиц, операндов, атомов и транслятора	20 %
2. Транслятор выражений	80 %
3. Сообщения о лексических ошибках	10 %
4. Сообщения о синтаксических ошибках	5 %