Компилятор CMilan

Авторы: Э. Ф. Аллахвердиев, Д. А. Тимофеев
Отредактировал: А. С. Герасимов (alexander.s.gerasimov@ya.ru)*

22 марта 2024 г.

Содержание

1 Обзор языка Милан

Язык Милан — учебный язык программирования, описанный в учебнике [2].

Программа на Милане представляет собой последовательность операторов, заключенных между ключевыми словами `begin` и `end`. Операторы отделяются друг от друга точкой с запятой. После последнего оператора в блоке точка с запятой не ставится. Компилятор CMilan не учитывает регистр символов в именах переменных и ключевых словах.

В базовую версию языка Милан входят следующие конструкции: константы, идентификаторы, арифметические операции над целыми числами, операторы чтения чисел из

*Фрагменты текста, выделенные вертикальной полосой на полях, полностью переписаны или добавлены А. С. Герасимовым; остальные места более или менее отредактированы им. Выделенные фрагменты в первую очередь послужат примерами для соответствующих фрагментов в отчёте по курсовой работе.

⟨*program*⟩ ::= ‘`begin`’ ⟨*statementList*⟩ ‘`end`’

⟨*statementList*⟩ ::= ε | ⟨*statement*⟩ {‘`;`’ ⟨*statement*⟩}

⟨*statement*⟩ ::= ⟨*ident*⟩ ‘`:=`’ ⟨*expression*⟩
| ‘`if`’ ⟨*relation*⟩ ‘`then`’ ⟨*statementList*⟩ [‘`else`’ ⟨*statementList*⟩] ‘`fi`’
| ‘`while`’ ⟨*relation*⟩ ‘`do`’ ⟨*statementList*⟩ ‘`od`’
| ‘`write`’ ‘`(`’ ⟨*expression*⟩ ‘`)`’

⟨*expression*⟩ ::= ⟨*term*⟩ {⟨*addop*⟩ ⟨*term*⟩}

⟨*term*⟩ ::= ⟨*factor*⟩ {⟨*mulop*⟩ ⟨*factor*⟩}

⟨*factor*⟩ ::= ⟨*ident*⟩ | ⟨*number*⟩ | ‘`read`’ | ‘`(`’ ⟨*expression*⟩ ‘`)`’

⟨*relation*⟩ ::= ⟨*expression*⟩ ⟨*cmp*⟩ ⟨*expression*⟩

⟨*addop*⟩ ::= ‘`+`’ | ‘`-`’

⟨*mulop*⟩ ::= ‘`*`’ | ‘`/`’

⟨*cmp*⟩ ::= ‘`=`’ | ‘`!=`’ | ‘`<`’ | ‘`<=`’ | ‘`>`’ | ‘`>=`’

⟨*ident*⟩ ::= ⟨*letter*⟩ {⟨*letter*⟩ | ⟨*digit*⟩}

⟨*number*⟩ ::= [‘`-`’] ⟨*digit*⟩ {⟨*digit*⟩}

⟨*letter*⟩ ::= ‘`a`’ | ‘`b`’ | ‘`c`’ | ‘`d`’ | ‘`e`’ | ‘`f`’ | ‘`g`’ | ‘`h`’ | ‘`i`’ | ‘`j`’ | ‘`k`’ | ‘`l`’ | ‘`m`’ | ‘`n`’ | ‘`o`’ | ‘`p`’ | ‘`q`’ | ‘`r`’ | ‘`s`’ | ‘`t`’ | ‘`u`’ | ‘`v`’ | ‘`w`’ | ‘`x`’ | ‘`y`’ | ‘`z`’ | ‘`A`’ | ‘`B`’ | ‘`C`’ | ‘`D`’ | ‘`E`’ | ‘`F`’ | ‘`G`’ | ‘`H`’ | ‘`I`’ | ‘`J`’ | ‘`K`’ | ‘`L`’ | ‘`M`’ | ‘`N`’ | ‘`O`’ | ‘`P`’ | ‘`Q`’ | ‘`R`’ | ‘`S`’ | ‘`T`’ | ‘`U`’ | ‘`V`’ | ‘`W`’ | ‘`X`’ | ‘`Y`’ | ‘`Z`’

⟨*digit*⟩ ::= ‘`0`’ | ‘`1`’ | ‘`2`’ | ‘`3`’ | ‘`4`’ | ‘`5`’ | ‘`6`’ | ‘`7`’ | ‘`8`’ | ‘`9`’

Рис. 1: Грамматика языка Милан

стандартного потока ввода и печати чисел в стандартном потоке вывода, оператор присваивания, условный оператор, оператор цикла с предусловием.

Программа может содержать комментарии, которые могут быть многострочными. Комментарий начинается символами ‘`/*`’ и заканчивается символами ‘`*/`’. Вложенные комментарии не допускаются.

Грамматика языка Милан в расширенной форме Бэкуса-Наура приведена на рисунке 1.

1.1 Константы

В языке реализована поддержка знаковых целочисленных констант. Синтаксически константы представляют собой последовательность цифр, перед которой может находиться знак ‘-’. Примеры констант: `0`, `9`, `1234`, `-19`, `-0`.

1.2 Идентификаторы

Идентификаторы представляют собой последовательность букв латинского алфавита и цифр. Первым символом идентификатора должна быть буква. Максимальная длина идентификатора ограничена 63 символами. Регистр символов не учитывается. Идентификаторы не должны совпадать с ключевыми словами `begin`, `end`, `if`, `then`, `else`, `fi`, `do`, `od`, `while`, `read`, `write`.

Примеры идентификаторов: `a`, `X`, `GCD`, `Milan23`. Цепочки символов, недопустимые в качестве идентификаторов: `12a` (первый символ не является буквой), `Begin` (цепочка символов совпадает с ключевым словом), `a_1` (цепочка содержит символ подчеркивания).

1.3 Арифметические выражения

⟨*expression*⟩ ::= ⟨*term*⟩ {⟨*addop*⟩ ⟨*term*⟩}

⟨*term*⟩ ::= ⟨*factor*⟩ {⟨*mulop*⟩ ⟨*factor*⟩}

⟨*factor*⟩ ::= ⟨*ident*⟩ | ⟨*number*⟩ | '`read`' | '`(`' ⟨*expression*⟩ '`)`'

Арифметические выражения строятся по традиционным для языков программирования правилам. Элементами арифметических выражений могут быть константы, идентификаторы, знаки арифметических операций '`+`' (сложение), '`-`' (вычитание), '`*`' (умножение), '`/`' (деление), скобки и ключевое слово `read`, которое обозначает операцию чтения числа из стандартного потока ввода.

Примеры правильных выражений: `12 + 4`, `i`, `(x+1)*y`, `2*x+1`, `read`, `read * (x - read)`.

Операции '`*`' и '`/`' имеют более высокий приоритет, чем операции '`+`' и '`-`'. Все эти операции левоассоциативны.

Значения арифметических выражений вычисляются слева направо с учетом приоритета операций. Порядок вычисления важен из-за присутствия операции `read`, благодаря которой вычисление значения выражения имеет побочный эффект.

1.4 Печать чисел в стандартном потоке вывода

⟨*statement*⟩ ::= '`write`' '`(`' ⟨*expression*⟩ '`)`'

Печать чисел осуществляется с помощью оператора `write`, аргументом которого является произвольное арифметическое выражение. Примеры использования оператора `write`: `write(5)`, `write(n+7)`, `write((2+read)*3)`, `write(read)`.

1.5 Оператор присваивания

⟨*statement*⟩ ::= ⟨*ident*⟩ '`:=`' ⟨*expression*⟩

При выполнении оператора присваивания сначала вычисляется значение выражения, записанного в его правой части. Затем результат записывается в ячейку памяти, соответствующую переменной в левой части оператора присваивания.

Последовательность символов '`:=`' является неделимой и не может содержать пробелов между символами '`:`' и '`=`'.

Примеры корректных операторов присваивания: `a := b + 1`; `c := read`.

```
begin
    x := read;
    y := read;
    max := x;
    if y > max then
        max := y
    fi;
    write(max)
end
```

Рис. 2: Пример условного оператора `if...then...fi`

1.6 Условный оператор

⟨*statement*⟩ ::= '`if`' ⟨*relation*⟩ '`then`' ⟨*statementList*⟩ ['`else`' ⟨*statementList*⟩] '`fi`'

⟨*relation*⟩ ::= ⟨*expression*⟩ ⟨*cmp*⟩ ⟨*expression*⟩

⟨*cmp*⟩ ::= '`=`' | '`!=`' | '`<`' | '`<=`' | '`>`' | '`>=`'

Условный оператор включает:

1. условие, которое представляет собой проверку на равенство или неравенство двух арифметических выражений,
2. последовательность операторов, которая должна быть выполнена, если условие истинно (блок `then`),
3. необязательную последовательность операторов, которая должна быть выполнена, если условие ложно (блок `else`).

Проверка условия $a?b$, где a и b — арифметические выражения, а «?» — одна из операций сравнения, производится следующим образом:

1. вычисляется значение выражения a;
2. вычисляется значение выражения b;
3. проверяется выполнение данного условия.

Поддерживаются следующие операции сравнения (отношения): '`=`' («равно»), '`!=`' («не равно»), '`<`' («меньше»), '`<=`' («меньше или равно»), '`>`' («больше»), '`>=`' («больше или равно»).

Условный оператор завершается ключевым словом `fi`. Оно используется, чтобы избежать известной проблемы «висячего else», которая встречается, в частности, в языках C, C++ и Java. Наличие «закрывающей скобки» `fi` позволяет однозначно интерпретировать вложенные условные операторы.

Примеры использования условного оператора приведены на рисунках 2 и 3. Обе эти программы считывают из стандартного потока ввода два числа. Программа на рисунке 2 печатает наибольшее из двух чисел. Программа на рисунке 3 печатает числа в порядке возрастания.

```
begin
    x := read;
    y := read;
    if x <= y then
        write(x);
        write(y)
    else
        write(y);
        write(x)
    fi
end
```

Рис. 3: Пример условного оператора `if...then...else...fi`

```
begin
    n := read;
    factorial := 1;
    x := 1;
    while x <= n do
        factorial := factorial * x;
        x := x + 1
    od;
    write(factorial)
end
```

Рис. 4: Пример цикла с предусловием

1.7 Цикл с предусловием

⟨*statement*⟩ ::= '`while`' ⟨*relation*⟩ '`do`' ⟨*statementList*⟩ '`od`'

⟨*relation*⟩ ::= ⟨*expression*⟩ ⟨*cmp*⟩ ⟨*expression*⟩

⟨*cmp*⟩ ::= '`=`' | '`!=`' | '`<`' | '`<=`' | '`>`' | '`>=`'

Оператор цикла (или просто цикл) «`while` B `do` S `od`» исполняется так:

(1) Вычисляется булевское значение условия B.[1]

(2) Если B ложно, то исполнение этого оператора цикла завершается.

(3) Иначе исполняется список операторов S и происходит переход к предписанию (1).

Пример программы, использующей цикл, приведен на рисунке 4. Эта программа считывает из стандартного потока ввода число и печатает его факториал.

[1] При этом два выражения, стоящие слева и справа от операции сравнения в условии B, вычисляются в естественном порядке: сначала первое, затем второе выражение (так, как указано в разделе 1.6).

```
bash$ cat factorial.mil
BEGIN
  n := READ;
  factorial := 1;
  i := 1;
  WHILE i <= n DO
    factorial := factorial * i;
    i := i + 1
  OD;
  WRITE(factorial)
END
bash$ cmilan factorial.mil > factorial.out
bash$ milanvm factorial.out
Reading input from factorial.out
> 5
120
```

Рис. 5: Пример использования компилятора CMilan

2 Использование компилятора

Компилятор CMilan преобразует программы на языке Милан в последовательность команд, или инструкций, виртуальной машины Милана. Общее описание виртуальной машины можно найти в учебнике [2], а более детальное описание — в документации, прилагаемой к исходным текстам виртуальной машины.

Исполняемый файл компилятора называется `cmilan` (в операционных системах семейства Unix) или `cmilan.exe` (в Windows). Компилятор имеет интерфейс строки. Чтобы скомпилировать программу, записанную в файле «program.mil», нужно выполнить команду «`cmilan program.mil`». Результатом работы компилятора является либо последовательность команд для виртуальной машины Милана, которая печатается в стандарном потоке вывода, либо набор сообщений о синтаксических ошибках, которые выводятся в стандартный поток ошибок.

При вызове программы `cmilan` без параметров компилятор печатает краткую информацию о порядке его вызова.

Чтобы сохранить генерируемую компилятором программу для виртуальной машины в файл, достаточно перенаправить в этот файл стандартный поток вывода.

На рисунке 5 приведен пример сеанса работы с компилятором.

2.1 Сборка компилятора в Unix

Для сборки компилятора в операционных системах семейства Unix (в частности, GNU/Linux и FreeBSD) необходим компилятор C++, в качестве которого рекомендуется использовать GCC, и GNU Make. Для сборки компилятора достаточно перейти в каталог `src`, в котором расположены исходные тексты компилятора CMilan, и выполнить команду `make`.

2.2 Сборка компилятора в Windows

Для сборки компилятора CMilan в Windows можно использовать компилятор GCC или Microsoft Visual Studio. При использовании GCC (Cygwin, MinGW) сборка произво-

дится так же, как и в Unix. Для сборки компилятора с помощью Visual Studio 2010 в дистрибутив компилятора включены файлы проекта (каталог `src\cmilan_vs2011`). Если необходимо использовать другие версии Visual Studio, достаточно создать пустой проект консольного приложения Win32 и добавить в проект существующие исходные тексты из каталога `src`.

3 Устройство компилятора

3.1 Архитектура компилятора

Компилятор CMilan включает три компонента:

1. лексический анализатор;
2. синтаксический анализатор;
3. генератор команд виртуальной машины Милана.

Лексический анализатор посимвольно читает из входного потока текст программы и преобразует группы символов в лексемы (терминальные символы грамматики языка Милан). При этом он отслеживает номер текущей строки, который используется синтаксическим анализатором при формировании сообщений об ошибках. Также лексический анализатор удаляет пробельные символы и комментарии. При формировании лексем анализатор идентифицирует ключевые слова и последовательности символов (например, оператор присваивания), а также определяет значения числовых констант и имена переменных. Эти значения становятся значениями атрибутов, связанных с лексемами.

Синтаксический анализатор читает сформированную лексическим анализатором последовательность лексем и проверяет ее соответствие грамматике языка Милан. Для этого используется метод рекурсивного спуска. Если в процессе синтаксического анализа обнаруживается ошибка, анализатор формирует сообщение об ошибке, включающее сведения об ошибке и номер строки, в которой она возникла. В процессе анализа программы синтаксический анализатор генерирует машинные команды, соответствующие каждой анализируемой конструкции.

Генератор кода представляет собой служебный компонент, ответственный за формирование внешнего представления генерируемого кода. Генератор поддерживает буфер команд и предоставляет синтаксическому анализатору набор функций, позволяющий записать указанную команду по определенному адресу. После того, как синтаксический анализатор заканчивает формирование программы, генератор кода используется для печати в стандартном потоке вывода отсортированной по возрастанию адресов последовательности инструкций. Использование генератора кода несколько упрощает устройство компилятора, поскольку синтаксический анализатор может генерировать команды не в порядке их следования в программе, а по мере получения всей необходимой информации для их формирования. Это особенно важно при трансляции таких инструкций, как условные операторы или циклы.

Все три компонента объединяются «драйвером» — управляющей программой компилятора. Драйвер анализирует аргументы командной строки, инициализирует синтаксический анализатор и вызывает его метод `parse()`, запуская процесс трансляции.

CMilan является примером простейшего однопроходного компилятора, в котором синтаксический анализ и генерация кода выполняются совместно. При этом компилятор никак не оптимизирует код. Реальные компиляторы обычно выполняют трансляцию и оптимизацию кода в несколько проходов, используя несколько видов внутреннего представления программы. Сведения об архитектуре таких компиляторов можно найти, в частности,

в классической «книге дракона» [1]. В случае CMilan, тем не менее, предпочтение было отдано не качеству генерируемого кода, а простоте реализации и легкости расширения.

3.2 Генератор кода

Основной задачей генератора кода является хранение и заполнение буфера инструкций последовательностью команд для виртуальной машины Милана. Генератор кода не отвечает за правильность этой последовательности и не выполняет никаких семантических преобразований. Тем не менее, генератор кода вполне мог бы быть расширен для того, чтобы выполнять простые оптимизации на уровне машинных команд.

Генератор кода описан в файлах `codegen.h` и `codegen.cpp`. Его реализация состоит из двух классов. Класс `Command` описывает машинные инструкций и отвечает за их вывод. В классе Codegen реализованы функции добавления инструкций в буфер, получения текущего адреса в буфере, резервирования ячейки для инструкции, которая должна быть добавлена в код позднее, и печати в выходном потоке последовательности инструкций из буфера.

Приведем краткое описание методов генератора кода.

- `void Command::print(int address, ostream& os)`

  Печать инструкции с указанием адреса `address` в потоке вывода `os`.

- `void CodeGen::emit(Instruction instruction)`

  Добавление инструкции `instruction` без аргументов в конец буфера.

- `void CodeGen::emit(Instruction instruction, int arg)`

  Добавление инструкции `instruction` с аргументом `arg` в конец буфера.

- `void CodeGen::emitAt(int address, Instruction instruction)`

  Запись инструкции `instruction` без аргументов в буфер со смещением `address`.

- `void CodeGen::emitAt(int address, Instruction instruction, int arg)`

  Запись инструкции `instruction` с аргументом `arg` в буфер со смещением `address`

- `int CodeGen::getCurrentAddress()`

  Возврат адреса, по которому будет записана очередная инструкция.

- `int CodeGen::reserve()`

  Резервирование ячейки памяти для инструкции. Метод добавляет в конец буфера инструкцию `NOP` и возвращает ее адрес.

- `void CodeGen::flush()`

  Вывод последовательности инструкций в выходной поток.

3.3 Лексический анализатор

Лексический анализатор преобразует считываемые из входного потока символы в лексемы языка Милан. Например, последовательность символов ‘`B`’, ‘`E`’, ‘`G`’, ‘`I`’, ‘`N`’ соответствует ключевому слову ‘`begin`’. Каждой лексеме соответствует отдельный элемент перечислимого типа `Token`. В частности, ключевому слову ‘`begin`’ соответствует константа `T_BEGIN`. С некоторыми лексемами связана дополнительная информация — значение атрибута. Так, с лексемой `T_NUMBER` (целочисленная константа без знака) связано число,

Имя лексемы	Значение	Атрибут
T_EOF	Конец текстового потока	
T_ILLEGAL	Недопустимый символ	
T_IDENTIFIER	Идентификатор	Имя переменной
T_NUMBER	Целочисленная константа без знака	Значение константы
T_BEGIN	Ключевое слово ‘begin’	
T_END	Ключевое слово ‘end’	
T_IF	Ключевое слово ‘if’	
T_THEN	Ключевое слово ‘then’	
T_ELSE	Ключевое слово ‘else’	
T_FI	Ключевое слово ‘fi’	
T_WHILE	Ключевое слово ‘while’	
T_DO	Ключевое слово ‘do’	
T_OD	Ключевое слово ‘od’	
T_WRITE	Ключевое слово ‘write’	
T_READ	Ключевое слово ‘read’	
T_ASSIGN	Оператор ‘:=’	
T_ADDOP	Операция типа сложения	A_PLUS (‘+’), A_MINUS (‘-’)
T_MULOP	Операция типа умножения	A_MULTIPLY (‘*’), A_DIVIDE (‘/’)
T_CMP	Операция сравнения	C_EQ (‘=’), C_NE (‘!=’), C_LT (‘<’), C_LE (‘<=’), C_GT (‘>’), C_GE (‘=’)
T_LPAREN	Открывающая скобка	
T_RPAREN	Закрывающая скобка	
T_SEMICOLON	‘;’	

Таблица 1: Лексемы языка Милан

равное значению этой константы, а с лексемой T_IDENTIFIER (идентификатор) — строка, содержащая имя переменной.

В таблице 1 приведен полный список лексем и значений связанных с ними атрибутов.

Все ключевые слова перечислены в ассоциативном массиве keywords_. При этом ключом является строка, соответствующая ключевому слову в нижнем регистре, а значением — соответствующая лексема. С помощью массива tokenNames_ каждой лексеме сопоставлено ее строковое внешнее представление, которое используется при формировании сообщений об ошибках. Последовательность символов, не соответствующая ни одной из лексем, считается ошибочной. В этом случае лексический анализатор возвращает значение T_ILLEGAL.

Исходный текст лексического анализатора находится в файлах scanner.h и scanner.cpp. Алгоритм лексического анализа реализован в классе Scanner. Он содержит следующие открытые методы:

- `Token token()`

  Получение текущей лексемы. Одна и та же лексема может быть прочитана многократно.

- `void nextToken()`

  Переход к следующей лексеме.

- `int getIntValue()`

Получение целочисленного атрибута текущей лексемы. В базовой версии Милана этот метод используется только для получения значений целочисленных констант.

- `string getStringValue()`

  Получение строкового атрибута текущей лексемы. В базовой версии Милана этот метод используется для получения имен идентификаторов.

- `Cmp getCmpValue()`

  Получение кода операции сравнения текущей лексемы (используется только вместе с лексемой `T_CMP`).

- `Arithmetic getArithmeticValue()`

  Получение кода арифметической операции (используется вместе с лексемами `T_ADDOP` и `T_MULOP`).

- `const int getLineNumber()`

  Получение номера текущей строки в исходном файле (используется при формировании сообщений об ошибках).

Поля класса описывают состояние лексического анализатора:

- `const string fileName_`

  Имя входного файла.

- `int lineNumber_`

  Номер текущей строки в анализируемой программе.

- `Token token_`

  Текущая лексема.

- `int intValue_`

  Целочисленный атрибут лексемы.

- `string stringValue_`

  Строковый атрибут лексемы.

- `Cmp cmpValue_`

  Код операции сравнения.

- `Arithmetic arithmeticValue_`

  Код арифметической операции.

- `map<string, Token> keywords_`

  Ассоциативный массив, описывающий ключевые слова языка Милан. Используется при обнаружении цепочки символов, которая может быть как идентификатором, так и ключевым словом.

- `ifstream input_`

  Входной поток для чтения из файла.

- `char ch_`

  Очередной символ программы.

Закрытые методы класса (служебные функции):

- `bool isIdentifierStart(char c)`

  Метод возвращает значение `true`, если символ `c` может быть первым символом идентификатора (в базовой версии языка Милан это означает, что символ является буквой латинского алфавита).

- `bool isIdentifierBody(char c)`

  Метод возвращает значение `true`, если символ `c` может быть частью идентификатора. В базовой версии языка Милан эт означает, что символ является буквой латинского алфавита или цифрой.

- `void skipSpace()`

  Пропуск всех пробельных символов (символов пробела, табуляции, перевода строки). При чтении символа перевода строки увеличивается номер текущей строки `lineNumber_`.

- `void nextChar()`

  Переход к следующему символу программы.

Основная часть алгоритма лексического анализа реализована в методе `nextToken`. Для того, чтобы перейти к следующей лексеме, выполняется следующая последовательность действий,

Прежде всего необходимо удалить пробелы и комментарии. Признаком начала комментария является последовательность символов '`/*`'. При обнаружении символа '`/`' необходимо проверить следующий за ним символ. Если он не совпадает с '`*`', значит, был найден оператор деления. В этом случае переменной `token_` присваивается значение `T_MULOP`, а атрибуту `arithmeticValue_` — значение `A_DIVIDE`. Если за символом '`/`' непосредственно следует '`*`', лексический анализатор пропускает все символы, пока не встретит закрывающую комментарий последовательность '`*/`' или конец файла. После того, как был найден признак конца комментария, еще раз вызывается функция `skipSpace`. Операция удаления комментариев повторяется многократно, пока очередной непробельный символ не окажется отличным от символа '`/`'. Если в процессе удаления пробелов или комментариев был встречен конец файла, очередной лексемой считается `T_EOF`.

После удаления пробелов и комментариев происходит анализ очередного символа. Он выполняется по следующим правилам.

1. Если символ является цифрой, то очередная лексема — целочисленная константа без знака (`T_NUMBER`). Лексический анализатор считывает из входного потока эту и все следующие за ней цифры, преобразуя полученную последовательность в целое число. Это число становится значением атрибута `intValue_`.

2. Если символ может быть началом идентификатора, из потока считываются все последующие символы, которые могут быть частью идентификатора. Полученная последовательность проверяется на совпадение с ключевым словом. В случае совпадения очередной лексемой считается лексема, соответствующая этому ключевому слову. Если цепочка символов не совпала ни с одним ключевым словом, очередная лексема считается идентификатором (`T_IDENTIFIER`), а сама цепочка становится значением строкового атрибута `stringValue_`.

3. Если символ равен ‘:’, лексический анализатор считывает следующий символ и проверяет, что он равен ‘=’. В этом случае возвращается лексема `T_ASSIGN`. Если следом за ‘:’ идет любой другой символ, лексема считается ошибочной (`T_ILLEGAL`).

4. Если символ равен ‘`!`’, производится аналогичная проверка на равенство следующего символа ‘=’. В случае успеха текущей лексемой становится лексема `T_CMP`, а атрибут `cmpValue_` принимает значение `T_NE`.

5. Если символ равен ‘<’, ‘>’ или ‘=’, текущей лексемой становится `T_CMP`. Чтобы определить значение атрибута `cmpValue_`, лексический анализатор может прочитать еще один символ для того, чтобы отличить оператор ‘<’ от оператора ‘>’, а оператор ‘>’ от оператора ‘>=’.

6. Если символ равен ‘+’ или ‘-’, переменная `token_` принимает значение `T_ADDOP`, при этом соответствующим образом устанавливается значение атрибута `arithmeticValue_`. Аналогично обрабатываются символ ‘`*`’ (лексема `T_MULOP`). Оператор деления обрабатывать таким же образом не нужно, поскольку он обнаруживается в ходе удаления комментариев.

7. Если символ совпадает с ‘;’, ‘(’, ‘)’, очередная лексема устанавливается в соответствующее символу значение.

3.4 Синтаксический анализатор

Задачами синтаксического анализатора является проверка соответствия программы грамматике языка Милан (рисунок 1) и формирование кода для виртуальной машины Милана в соответствии со структурой программы. Синтаксический анализ выполняется методом рекурсивного спуска. Каждому нетерминальному символу грамматики сопоставлен метод, выполняющий проверку соответствия последовательности лексем одному из тех правил грамматики, в левой части которых стоит данный нетерминальный символ. Семантические действия (генерация кода) встроены в код метода.

Исходный текст синтаксического анализатора находится в файлах `parser.h` и `parser.cpp`. Алгоритм синтаксического анализа реализован в классе `Parser`. Конструктор класса в качестве аргумента принимает имя файла, в котором находится анализируемый текст, и создает экземпляры лексического анализатора и генератора кода.

Синтаксический анализатор предоставляет один открытый метод `void parse()`, который используется для того, чтобы начать процесс анализа.

Состояние анализатора описывается следующими полями:

- `Scanner* scanner_`

  Экземпляр лексического анализатора.

- `CodeGen* codegen_`

  Экземпляр генератора кода.

- `std::ostream& output_`

  Выходной поток, в который должны быть выведены инструкции программы. Базовая версия компилятора Милан использует в качестве выходного потока стандартный поток вывода (`std::cout`).

- `bool error_`

  Признак ошибки в тексте программы. Если `error_` принимает значение «истина», генерируемый машинный код не выводится.

- `map<string, int> variables_`

  Таблица имен, найденных в программе. Она сопоставляет каждой переменной ее адрес в памяти виртуальной машины. Поскольку базовая версия языка Милан не содержит вложенных блоков, процедур или функций, таблица имен представляет собой простой ассоциативный массив.

- `int lastVar_`

  Адрес последней найденной переменной.

Синтаксический анализатор включает ряд вспомогательных функций (закрытые методы класса).

- `bool see(Token t)`

  Сравнение текущей лексемы с образцом. Текущая позиция в потоке лексем не изменяется.

- `bool match(Token t)`

  Проверка совпадения текущей лексемы с образцом. Если лексема и образец совпадают, лексема изымается из потока.

- `void mustBe(Token t)`

  Проверка совпадения текущей лексемы с образцом. Если лексема и образец совпадают, лексема изымается из потока. В противном случае формируется сообщение об ошибке, а программа считается некорректной.

- `void next()`

  Переход к следующей лексеме.

- `void reportError(const string& message)`

  Формирование сообщения об ошибке. Каждое сообщение включает текст `message` и номер строки, в которой обнаружена ошибка.

- `void recover(Token t)`

  Восстановление после ошибки. Используется примитивный алгоритм: если очередная лексема не совпадает с ожидаемой, анализатор пропускает все последующие лексемы, пока не встретит ожидаемую лексему или конец файла. Хотя такой метод восстановления не отличается точностью, он, тем не менее, позволяет продолжить анализ и, возможно, найти ошибки в оставшейся части программы.

- `int findOrAddVariable(const string&)`

  Поиск имени переменной в таблице имен. Если имя найдено, метод возвращает адрес переменной, в противном случае имя добавляется в таблицу имен, и для соответствующей переменной резервируется новый адрес.

Кроме служебных методов, класс `Parser` содержит закрытые методы `void program()`, `void statementList()`, `void statement()`, `void expression()`, `void term()`, `void factor()`, `void relation()`, которые соответствуют нетерминальным символам грамматики. Эти методы не возвращают значений и не принимают аргументов, поскольку с нетерминальными символами не связаны явные атрибуты: все семантические действия, которые производятся во время анализа программы, модифицируют буфер общего для всех методов генератора кода.

3.4.1 Распознавание операторов

Продемонстрируем алгоритм работы синтаксического анализатора при разборе операторов (метод `statement`).

В соответствии с грамматикой языка Милан, в качестве оператора может выступать оператор присваивания, условный оператор `if`, оператор цикла `while` или оператор печати `write`. Для выбора нужной альтернативы достаточно прочитать первую лексему оператора.

Если очередная лексема равна `T_IDENTIFIER`, синтаксический анализатор имеет дело с оператором присваивания. Он считывает имя переменной, стоящей в левой части присваивания, и определяет ее адрес. Затем анализатор проверяет, что следующая лексема совпадает с `T_ASSIGN`. После знака ':=' в программе должно следовать арифметическое выражение, поэтому анализатор вызывает метод `expression()`. Этот метод проверяет правильность арифметического выражения и генерирует последовательность команд для его вычисления. В результате выполнения этой последовательности команд на вершине стека будет находиться значение выражения. Чтобы выполнить присваивание, достаточно записать значение с вершины стека в память по адресу переменной в левой части оператора присваивания. Для этого после вызова `expression()` анализатор добавляет в буфер команд инструкцию `STORE`, аргументом которой является запомненный ранее адрес переменной.

Если очередная лексема равна `T_IF`, анализатор должен начать разбор условного оператора. Сначала он вызывает метод `relation()`, проверяя, что после лексемы `T_IF` следует условие, т. е. сравнение двух выражений. Метод `relation()` генерирует код, вычисляющий и сравнивающий два выражения. При выполнении этого кода на вершине стека останется значение 1, если условие выполнено, и 0 в противном случае. Если условие не выполнено, то должен произойти переход по адресу первой инструкции блока `else` или, если этого блока нет, по адресу следующей за условным оператором инструкции. Однако ни тот ни другой адрес еще не известен синтаксическому анализатору. Чтобы решить эту проблему, применяется так называемый «метод обратных поправок» [1, раздел 6.7]. Анализатор резервирует место для условного перехода `JUMP_NO` и запоминает адрес этой инструкции в переменной `jumpNoAddress`, после чего переходит к дальнейшему анализу условного оператора.

Следом за условием в тексте программы должна находиться лексема `T_THEN`. Если она обнаружена, анализатор переходит к следующей лексеме и вызывает метод `statementList()`, который анализирует список операторов и генерирует для них исполняемый код. Дальнейшие действия зависят от того, присутствует ли в условном операторе ветвь `else`. Если следующая лексема равна `T_ELSE`, синтаксический анализатор должен сгенерировать инструкцию безусловного перехода `JUMP` для выхода из условного оператора, после чего приступить к анализу блока `else`. Если условный оператор не содержит блока `else`, инструкция безусловного перехода в конце блока `then` не нужна.

Генерация безусловного перехода `JUMP` также требует знания адреса инструкции, следующей за последней инструкцией условного оператора, поэтому анализатор снова должен зарезервировать ячейку памяти для команды перехода. За этой ячейкой следует первая команда блока `else`, по адресу которой необходимо было выполнить переход в случае, если условие не было выполнено. Теперь этот адрес известен, и синтаксический анализатор генерирует инструкцию условного перехода `JUMP_NO` и помещает ее в буфер команд по адресу, который записан в переменной `jumpNoAddress`. В случае отсутствия блока `else` поправка происходит таким же образом, но в качестве адреса перехода просто используется адрес инструкции, следующей за последней инструкцией блока `then`. После вызова функции `statementList`, которая разбирает список операторов блока `else`, аналогично исправляется зарезервированная инструкция безусловного перехода `JUMP`.

вычисление значения условия B
при ложном B переход (JUMP_NO) к ИНАЧЕ
код для S_1
переход (JUMP) к ПОСЛЕ
ИНАЧЕ: код для S_2
ПОСЛЕ: первая команда после этого условного оператора

Рис. 6: Схема исполняемого кода для условного оператора «`if` B `then` S_1 `else` S_2 `fi`»

НАЧАЛО: вычисление значения условия B
при ложном B переход (JUMP_NO) к ПОСЛЕ
код для S
переход (JUMP) к НАЧАЛО
ПОСЛЕ: первая команда после этого цикла

Рис. 7: Схема исполняемого кода для цикла «`while` B `do` S `od`»

Схема исполняемого кода, соответствующего условному оператору «`if` B `then` S_1 `else` S_2 `fi`», приведена на рисунке 6. Схему исполняемого кода для условного оператора «`if` B `then` S `fi`» читатель может изобразить самостоятельно.

Если очередная лексема равна `T_WHILE`, то синтаксическому анализатору предстоит разобрать оператор цикла. Анализатор запоминает текущий адрес команды, поскольку после каждой итерации цикла нужно будет возвращаться по этому адресу. Затем анализатор вызывает метод `relation()`, чтобы сгенерировать код для проверки условия цикла. Если это условие окажется ложным, то нужно будет выполнить переход к следующей за данным циклом команде. Для этого синтаксический анализатор резервирует ячейку памяти для инструкции `JUMP_NO`. Потом он проверяет наличие лексемы `T_DO`, вызывает метод `statementList()` для обработки тела цикла и проверяет наличие лексемы `T_OD`. Наконец, анализатор генерирует инструкцию безусловного перехода `JUMP` к началу проверки условия цикла и записывает в зарезервированную ячейку инструкцию условного перехода `JUMP_NO` по известному теперь адресу команды, следующей за данным циклом. Схема исполняемого кода для цикла приведена на рисунке 7.

Псевдокод алгоритма генерации кода для цикла «`while` B `do` S `od`»[2]:

1. Прочитать лексему `while`.
2. Записать в переменную `loopBeginningAddress` адрес очередной ячейки памяти в буфере VM-команд[3] (этот адрес будет адресом первой VM-команды данного цикла).
3. Сгенерировать VM-команды для вычисления значения условия B, вызвав метод `Parser::relation()`.
4. Зарезервировать в буфере VM-команд очередную ячейку памяти (для VM-команды `JUMP_NO`, которая будет осуществлять выход из цикла) и записать адрес этой ячейки в переменную `justAfterConditionAddress`.
5. Прочитать лексему `do`.
6. Сгенерировать VM-команды для списка операторов S, вызвав метод `Parser::statementList()`.
7. Прочитать лексему `od`.
8. Сгенерировать VM-команду «`JUMP loopBeginningAddress`».

[2] Обработка синтаксических ошибок намеренно опускается в этом псевдокоде.

[3] Команду (или инструкцию) виртуальной машины (virtual machine) мы можем иногда называть VM-командой (или VM-инструкцией) для краткости и для того, чтобы её было легко отличить от инструкции (оператора) программы на языке Милан или C++.

9. В буфер VM-команд по адресу `justAfterConditionAddress` записать VM-команду «`JUMP_NO` ⟨адрес очередной ячейки памяти в буфере VM-команд⟩» (последний упомянутый адрес будет адресом первой VM-команды после этого цикла).

Если очередная лексема равна `T_WRITE`, анализатор проверяет, что следующая лексема равна `T_LPAREN`, а затем вызвает метод `expression()` для генерации кода, вычисляющего выражение-аргумент оператора `write`. Наконец, анализатор проверяет, что за этим выражением следует лексема `T_RPAREN` и генерирует инструкцию `PRINT`, которая будет печатать лежащее на вершине стека значение выражения.

Список литературы

[1] А. Ахо, М. Лам, Р. Сети, Дж. Ульман, *Компиляторы: принципы, технологии и инструментарий*, 2-е изд. М.: Вильямс, 2011.

[2] Ю.Г. Карпов, *Теория и технология программирования. Основы построения трансляторов*. СПб.: БХВ-Петербург, 2005.