Российская академия наук
Сибирское отделение
Институт систем информатики
им. А. П. Ершова

Л.В. Городняя

ПОДХОДЫ К ПРЕДСТАВЛЕНИЮ СИНТАКСИСА ЯЗЫКОВ ПРОГРАММИРОВАНИЯ

Препринт

185

Новосибирск 2019

Рецензент - к. физ.-мат. наук, Ф.А. Мурзин

Препринт посвящен выбору средств использования методов синтаксически ориентированного конструирования и функционального программирования при решении задач обработки текстов определений языков программирования. Рассматривается проблема совершенствования разрабатываемых систем программирования и создания новых языков программирования, нацеленных на эффективное решение современных задач разработки надёжных и удобных информационных систем, включая организацию параллельных вычислений.

Работа выполнена при поддержке Российского фонда фундаментальных исследований, проект № 18-07-01048-а.

Siberian Branch of the Russian Academy of Sciences
A. P. Ershov Institute of Informatics Systems

L.V. Gorodnyaya

APPROACHES TO PRESENTING THE SYNTAX OF PROGRAMMING LANGUAGE

Preprint

185

Novosibirsk 2019

Reviewe: Dr. F. A. Murzin

The preprint is devoted to the choice of means of using methods of syntactically oriented design and functional programming in solving problems of text processing of definitions of programming languages. Improving extant programming systems and creating new programming languages aimed at effective solution of modern problems of developing reliable and convenient information systems, including organization of parallel computing, is considered.

The research was supported by the Russian Foundation for Basic Research (RFBR) of the Russian Academy of Sciences № 18-07-01048-a.

Введение

Происходит переход от доминирующей парадигмы императивно последовательного программирования к парадигмам развивающихся систем и организации асинхронных и синхронизованных параллельных процессов в информационных сетях над многопроцессорными комплексами с не вполне защищённой общей памятью. Круг возникающих проблем достаточно ярко проявился на задачах суперкомпьютерных параллельных вычислений. Многие затруднения с параллельным программированием имеют образовательный характер. Обучение параллельному программированию осложнено не только дистанцией между уровнями абстракций, в которых описываются решения сложных задач на современных языках программирования (ЯП) с учётом тонкостей реализации аппаратуры и архитектурных принципов управления параллельными вычислениями, но и особенностями восприятия информации человеком, включая тенденцию синтаксически ориентированных методов обработки данных с предпочтением работы по шаблонам. Это достаточное основание для обзора основных подходов к методам определения и анализа синтаксиса языков программирования. Ниже рассмотрены основные методы представления ЯП, редко встречающиеся примеры наглядного двумерного преставления программ и предложения по обогащению методов синтаксически ориентированных обработчиков данных техникой функционального программирования (ФП). Всё это приводит к рассмотрению метаязыков как языков программирования синтаксических анализаторов и систем программирования и позволяет в эту сферу переносить обычные методы программирования.

1. Общее представление

Методика определения языка как множества текстов сложилась в математике задолго до появления компьютера. А. Чёрч отмечал полезность метаязыков для лаконичного описания языка и условность границы между синтаксисом языка и его семантикой — такая граница определяется уровнем сложности формул, удобных для понимания. Для практических целей достаточно одного метаязыка, что не исключает использования иерархии метаязыков. Обычно метаязыки обладают самоприменимостью, что дает им лаконизм и чёткость понимания границ применения.

Наиболее удобными оказались определения синтаксиса в форме контекстно свободных грамматик. По мере появления и развития языков программирования были созданы средства формализации их синтаксиса в виде форм Бэкуса-Наура (БНФ), для некоторых классов которых можно сравнительно легко строить автоматы генерации текстов языка и анализа принадлежности произвольного текста языку. Были исследованы вопросы эффективности таких автоматов, сформулированы критерии ассоциирования автоматов с формулами и предложены системы преобразования формул, позволяющие их приводить к нормализованным или каноническим формам, для которых гарантирована оптимальность алгоритмов анализа. Появилась классификация грамматик по особенностям ассоциированных с ними алгоритмов разбора текстов — нисходящий или восходящий и длина интервала обзора текста (LL(k) или LR(k)). Далее были созданы и исследованы методы нагрузки таких алгоритмов действиями, выполняющими перевод исходного языка в какой-нибудь другой — грамматики с переводом. Все эти результаты оснащены математическими построениями в

виде теорем с доказательствами и изложены в безукоризненно авторитетных источниках [1].

Строго говоря, так был получен базис для перехода к конструированию любых синтаксически ориентированных обработчиков данных, что фактически и произошло в форме неявного соревнования между способностями человека понимать сложные формы и создавать автоматы для анализа таких форм. Дальнейшие вариации методов и подходов в этой сфере отражают разнообразие выбора языковых средств и целей синтаксически ориентированной обработки текстов языка. Появилось заметное число достаточно разных ЯП, описание каждого из них сопровождалось определением своего метаязыка, наиболее удобно представлявшего основные конструкции определяемого языка. Преимущественно это были модифицированные или расширенные БНФ, дополненные конструкциями, повышающими **гомоиконность** метаязыка, т. е. проявление подобия между формулой и алгоритмом её обработки, и/или **эффективность** ассоциированного с ним конструктора анализаторов [8].

Чаще всего модификации и расширения направлены на лаконизм и понятность или на соответствие эффективным шаблонам автоматизированного анализа формул. Выражается это в выборе обозначений для ввода новых понятий («::=», «=», «→», «←», «:»), зрительного различения терминальных символов от метапонятий (уголковые скобки, кавычки, заглавные или строчные буквы, подчёркивание, шрифт), выбор альтернативы («|», «,») из конечного множества вариантов («{ }»), необязательные («[ ]») и пустые («Ɛ») вхождения символов или понятий, группы вхождений («( )»), многократные вхождения («*», «+», « . . . »), произвольное или конечное число вхождений, перечень через разделитель («\») произвольный символ или любое понятие («_»),

исключённый символ («~»). Каждому обозначению соответствует правило анализирующего автомата, допускающее очевидный перевод в шаблон при конструировании программы, эквивалентной автомату анализа. В результате при изучении ЯП такое его определение помогает понимать реальную логику анализа текста и смысл диагностических сообщений о тупиковых ситуациях.

Дальнейшее развитие синтаксически ориентированных методов приводит к их обогащению средствами интегрированных описаний, представляющих частичное представление семантических атрибутов, проявляющихся по ходу анализа текста и порождения его графового представления [5]. Такие средства помогают разработчику системы программирования (СП) структурировать реализацию семантического анализа и оптимизировать поддержку выполнения программы, представляющую собой довольно сложных комплекс функций формирования внутренних представлений, их преобразование и кодогенерации. В начале 1970-ых годов А.Л.Фуксман предложил методику вертикального слоения сложных программ на основе идеи сосредоточенного представления рассредоточенных действий, что концептуально похоже на современные идеи аспектно-ориентированного программирования [10]. При решении задач семантического анализа методика вертикального слоения позволяет избежать усложнённого представления комплекса функций и на основе более простых представлений разделённых составляющих формировать интегрированную программную систему. Такое разделение допускает поддержку работы программиста в удобных ему терминах, без принудительного освоения понятий разработчика ЯП и СП. Полезны множественная мнемоника, синонимы и сокращения, дающие лаконизм.

Желательно использовать любые свойства имён и констант, хотя бы в виде сохранённых комментариев. Билингвистический интерфейс позволит видеть опорную иноязычную терминологию, а также полиморфный синтаксис как в языках Cobol и Робик, предлагающий для целей обучения и разных категорий пользователей варианты представления конструкций, обладающих идентичным кодом или эквивалентной семантикой.

Опыт применения регулярных выражений, реализации языков ФП и современная тенденция к мультипарадигмальности новых ЯП дали достаточные основания для перехода к СП, одновременно содержащим интерпретатор, компилятор, мемоизатор и декомпозитор программ. Можно вспомнить, что проекты первых компьютеров допускали, что ряд математических функций будет реализован в форме таблиц. В экспериментах .Net и GNU по разработке многоязыковых СП заметна тенденция интеграции программ из библиотечных модулей, созданных на разных языках, что можно рассматривать как движение к интегральному исчислению программ, но без опоры на соответствующее дифференциальное, зачатки которого можно видеть в функциональном стиле программирования, аспектно-ориентированном программировании, исследованиях по срезам программ и технике формирования проекций в смешанных вычислениях. Не исключено, что математический аппарат формального представления синтаксиса ЯП полезно пополнить использованием операции пересечения множеств и технику конструирования анализаторов обогатить параллельными алгоритмами. В целом естественно все эти метасредства рассматривать как элементы языков программирования синтаксических анализаторов и сложных программных систем, включая системы программирования, что позволяет опыт и

технологии прикладного программирования обобщать на более сложные задачи.

Независимо происходит развитие средств вывода логики программы и её верификации на соответствие целям и требованиям. Производительные характеристики таких средств улучшаются, но практика их применения не получает признания у разработчиков программных систем. Комплекты поставки программных инструментов не редко включают в себя удостоверенные тесты и документацию, в пределах которых гарантируется работоспособность программы. В этом плане популярные методы работы с определениями ЯП, рассматриваемых как специализированные программы, могут существенно расширить пространство средств информационной обработки.

2. Преобразования формул

Примечательны примеры типичных форм определения конструкций известных ЯП, допускающие более лаконичные представления даже с помощью простых регулярных выражений, дополненных, в соответствии с традицией озвучивания математических формул, речевыми аналогами. В таблицах 1–6 приведены варианты подбора речевых аналогов для некоторых определений синтаксиса языка C++.

Такое разнообразие форм привело к развитию методов синтаксического управления обработкой данных, включая создание интегрированных форм определения языков программирования, их диагностических расширений, продолжающих грамматик, смешанных и отложенных вычислений, включая частичную интерпретацию программ и СП, совмещающих преимущества интерпретации и компиляции в одном

инструментальном комплексе.

Таблица 1. Строка из любых символов за исключением *конца строки* или '>'

Определение (C++)	Речевые аналоги
h-char-sequence: *h-char* *h-char-sequence h-char* ***h-char:*** any member of the source character set except *new-line* and >	***h-строка*** — это *h-литера* **или** строка с *h-литерой* ***h-литера*** - это любой символ кроме *new-line* или '>'
То же самое в виде регулярного выражения	
H-char-sequence: ***** \ *new-line* \ *'>'*	***h-строка*** — это любое число символов кроме *new-line* и кроме '>'

Таблица 2. Цифры для изображения десятичных чисел

Определения эквивалентны	Речевые аналоги
digit: *one of* 0 1 2 3 4 5 6 7 8 9	**Цифра** — это одна из перечисленных цифр
digit: **{** 0 1 2 3 4 5 6 7 8 9 **}**	**Цифра** — это один из элемент множества

Таблица 3. Первая цифра для изображения десятичных чисел

Определения эквивалентны	Речевые аналоги
Nonzero-digit: *one of* 1 2 3 4 5 6 7 8 9	**Ненулевая цифра** — это одна из перечисленных цифр (нет нуля)
Nonzero-digit: *digit* \ '0'	**Ненулевая цифра** — это любая **цифра** кроме нуля

Происходит чёткое расслоение определения ЯП на лексику, синтаксис, семантику и прагматику. Лексика как правило определена конечным автоматом. Синтаксис может требовать расширяемого автомата из-за

подверженности уточнениям при отладке ЯП. Семантика нередко формализована лишь отчасти и доопределяется комментариями и примерами или шаблонами программных конструкций. Прагматика обычно представлена прецедентами реализации и сценариями успешного применения.

Таблица 4. Изображения идентификатора

Определения эквивалентны	Речевые аналоги
id-expression: *unqualified-id* *qualified-id* ***qualified-id:*** *nested-name-specifier template* *opt* *unqualified-id*	**Представление идентификатора** — это *unqualified-id* ***или*** *qualified-id* ***qualified-id*** — это конструкция из трёх компонентов, последний - *unqualified-id*
id-expression: *unqualified-id* ***(*** *nested-name-specifier template* *opt* *unqualified-id* ***)***	**Представление идентификатора** — это *unqualified-id* ***или*** три компонента, последний - *unqualified-id*
id-expression: *unqualified-id* ***(qualified-id:*** *nested-name-specifier template* *opt* *unqualified-id* ***)***	**идентификатор** — это *unqualified-id* ***или*** конструкция «***qualified-id***», *состоящая* из трёх компонентов, последний - *unqualified-id*

Таблица 5. Перечень без разделителей

Определения эквивалентны	Речевые аналоги
statement-seq: *statement* *statement-seq statement*	**Последовательность операторов** – это оператор ***или*** последовательность операторов с последующим оператором
statement-seq: *statement* +	**Последовательность операторов** – это непустая последовательность операторов

Таблица 6. Перечень с разделителями

Определение эквивалентны	Речевые аналоги
expression: *assignment-expression* *expression ,* *assignment-expression*	**Выражение** – это присваивание ***или*** выражение, запятая, затем присваивание
expression: ***(****assignment-expression* /','***)***	**Выражение** — это перечень присваиваний через запятую
expression: *assignment-expression* ***(*** ',' *assignment-expression* ***) ****	**Выражение** — это присваивание, затем любые число групп (возможно, ни одной) из запятой и присваивания

Обнаруживается дилемма в выборе стратегии программирования. Легкий старт при подготовке текста программ, приводящий к тяжкому пути их отладки, или неторопливая раскрутка исходной программы, допускающая стабильные улучшения программы по мере изучения возникающих проблем. Отчасти это противоречие смягчилось с появлением языка Fortran с методикой раздельной компиляции, позволившей новые программы создавать как дополнение к комплексу ранее созданных библиотечных процедур. Распространение ООП позволило такую технику перенести на развивающиеся постановки задач.

Другой круг проблем связан с освоением технических новинок и проявлением трудоёмкости переноса результатов программирования на новые архитектуры. Венская методика определения языков программирования (VDM) позволяет мультипликативную оценку сложности такого переноса привести к аддитивной ценой введения двух уровней абстрагирования:

абстрактный синтаксис (АС) и абстрактная машина (АМ). Эта оценка может быть снижена выделением классов семантически эквивалентных языков и прагматически совместимых машин, а также выделением типовых семантических систем при реализации СП [12].

В этом плане интересна история принятия решений при создании многоязыковой системы БЕТА[1]. Ради снижения трудоёмкости реализации универсальных оптимизирующих преобразований разрабатывался общий внутренний язык, но оказалось, что большинство таких преобразований обладают зависимостью от компьютерных архитектур. В результате был разработан второй внутренний язык, предназначенный для машинно-ориентированных преобразований программ, включая кодогенерацию. Обычно АС реализуется в виде древообразных структур, успешно представляемых списками [13].

Подобная схема была реализована в ряде работ по оптимизации программ и при реализации языка С, что позволило этому языку, по мнению многих экспертов, выполнять роль ДНК в мире ЯП. Появление методики создания DSL-языков [11] на базе Clang&LLVM (gcc) [14] убедительно подтверждает практичность такого подхода, освобождающего языкотворчество от изобретения новых метаязыков, но заодно и от развития методов анализа ЯП. В Таблицах 7–9 показаны основные конструкции,

---

1. 1 Замысел заключался в реализации трёх самых сложных на то время, языков в качестве эксперимента по методике преодоления мультипликативной оценки сложности реализации компиляторов и переноса программного обеспечения в условиях активного появления новых компьютерных архитектур и создания новых языков программирования. Захаров Л.А., Покровский С.Б., Степанов Г.Г., Тен С.В. Многоязыковая транслирующая система. − Новосибирск, 1987. − 151 с.

используемые при анализе ЯП.

Таблица 7. Демонстрационный пример: текст программы на Си

Пояснение	*Текст программы на языке Си*
Комментарий «Example.c»	// Example.c
Включение библиотеки stdio.h	#include <stdio.h>
Объявление переменной global	int global;
Заголовок функции main	int main (int argc, char *argv[ ]) {
Объявление param со значением 1	int param = 1;
Вызов функции myPrint для печати param	myPrint (param);
Успешный выход из функции main	return 0;
Конец определения функции main	}

Анализатор производит графовое представление, удобное для предстоящего семантического анализа.

Таблица 8. Демонстрационный пример: графовое Clang-представление результата анализа программы на Си

Текст на языке Си	C → AST
// Example.c	← Комментарии не имеют представления
#include <stdio.h>	← Макросы не имеют представления
int global;	**(VarDecl** global 'int' **)** ← **AST** for global
int main (	(**FunctionDecl** 'void main (int, char **)') ← **AST** for main ( )
int argc,	|-------> (**ParmVarDecl** argc 'int')
char *argv[ ])	|-------> (**ParmVarDecl** argv 'char**' ; 'char**')
{	|-------> **[CompoundStat]**
	|----> **[DeclStat]**
	| |---->(**VarDecl** param 'int')
int param = 1;	| |----->**[| IntegerLiteral** 1 'int' **|]**
	|--->**[| CallExpr** 'void' **|]**
	| |--->**[| ImplicitCastExpr** 'void (*) ()' **|]**
myPrint (param);	| | |--->**[| DeclRefExpr** 'myPrint' 'void (*) ()' **|]**
	| |---->**[| ImplicitCastExpr** 'int' **|]**
	| |--->**[| DeclRefExpr** 'param' 'int' **|]**
return 0;	|---->**[ReturnStmt]**
}	|---->**[| IntegerLiteral** 0 'int' |]

Полученное графовое представление обладает гомоиконностью, представленой в Таблице 9 в списочной форме.

Талица 9. Lisp-аналог Clang-представления примера программы на Си

C	AST → Lisp

// Example.c
#include <stdio.h>
int global;
 	
int main (
int argc,
char *argv[ ])
{
int param = 1;
 	
myPrint (param);
 	
return 0;
}
	; Example.c
	(**load** “stdio.h”) ; Включение библиотеки
	(**VarDecl** global 'int)
	
	(**FunctionDecl** main (int, char *)
	(**ParmVarDecl** argc 'int)
	(**ParmVarDecl** argv 'char** 'char**)
	(**CompoundStat**
	(**DeclStat** (**VarDecl** param 'int)
	(**IntegerLiteral** 1 'int))
	(**CallExpr** 'void
	(**ImplicitCastExpr** 'void '(*) ())
	(**DeclRefExpr** 'myPrint 'void '(*) ())
	(**ImplicitCastExpr** 'int)
	(**DeclRefExpr** 'param 'int)
	)
	(**ReturnStmt** (**IntegerLiteral** 0 'int))
	)
	)

3. Функциональный синтаксис

Привлечение аппарата функций над синтаксическими формулами позволяет повысить лаконизм и выразительность определений [13]. Механизмы приведения грамматики языка к виду, удобному для автоматизации конструирования обработчиков программ, включая проверку текстов на возможность их включения в программы при макроподстановках и работе с фрагментными переменными, и грамматики с переводом

позволяют принципиально расширить спектр базовых языков и средств обработки программ с накоплением их правильности при отладке. Возможен обратный вывод тестового покрытия программы для сравнения с предполагаемым тестовым набором. Декомпозиция программы по тестовым наборам может дать специализированные проекции, снижающие накладные расходы. Конверторы, освобождающие от лишних переходов на уровень машины, могут способствовать выделению эквивалентных подъязыков и, следовательно, типизации средств их реализации. Раскрытие недоопределённостей в программе при отладке можно выполнять по спецификациям, тестам или запросам. Приведение любых выражений к функциональной форме уже активно используется в новых инструментах создания СП. Сдерживающим фактором является риск ошибки в компромиссе между автоматизацией и ручным управлением.

Долгоживущие и новые ЯП подвержены улучшениям даже при самых серьёзных и авторитетных усилиях на стандартизацию их определений. В результате возникают значительные трудозатраты на повторное программирование СП [4]. Методы ФП позволяют осуществлять декомпозицию определений не только по различным функциям, но и по уровню их отлаженности, модифицированности, точкам роста.

Примеры функциональных шаблонов с параметризацией отдельных атрибутов, реализуемых техникой макроподстановки, можно показать на свертке фрагментов синтаксиса C++ с помощью функций F (X) и FF (X, Y) (См. Таблица 10).

Таблица 10. Примеры определений разных видов строк по стандарту C++

Формальное определение	*Пояснение*
h-char: any member of the source character set except new-line and >	Любой допустимый символ, кроме конца строки и >
q-char: any member of the source character set except new-line and "	Любой допустимый символ, кроме конца строки и ”
c-char: any member of the source character set except the single-quote ’, backslash \, or new-line	Любой допустимый символ, кроме ', \. конца строки
s-char: any member of the source character set except the double-quote ", backslash \, or new-line	Любой допустимый символ, кроме “, \, конца строки
r-char: any member of the source character set except a right parenthesis) followed by a double quote ".	Любой допустимый символ, кроме) “
d-char: any member of the source character set except space, the left parenthesis (, the right parenthesis), the backslash \, and newline.	Любой допустимый символ, кроме пробела. (,). \ иконца строки

Функция F (X) генерирует правую часть синтаксического определения строки в зависимости от исключаемого символа.

F (X: { >, “, ', \,), (})

=> Любой допустимый символ, кроме {конца строки и X}

=> (_\{Enter | X})* *// Краткий вариант в стиле регулярных выражений*

Функция FF (Y, X) конструирует определение

заданного вида строк в зависимости от её ключевого символа и граничного символа.

FF (Y: {H,Q,C,S,R,D}, X: { >, “, ', \,), (})
=> Y-char: Любой допустимый символ,
кроме {конца строки и X}
=> Y-char: (_\{Enter | X})*
// Краткий вариант в стиле регулярных выражений

Таблица 11. Примеры макрофункций для конструирования определений разных видов строк по стандарту C++

Задан исключённый символ	*Задан ключевой символ вида строки*	*Результат подстановки*
H-char: **F (>)**	**FF (H, >)**	**H-char:** Любой допустимый символ, кроме {конца строки и >}
Q-char: F (“)	FF (Q, “)	Q-char: Любой допустимый символ, кроме {конца строки и “}
C-char: F (')	FF (C, ')	C-char: Любой допустимый символ, кроме {конца строки и '}
S-char: F (\)	FF (S, \)	S-char: Любой допустимый символ, кроме {конца строки и \}
R-char: F (“)”)	FF (R, “)”)	R-char: Любой допустимый символ, кроме {конца строки и “)”}
D-char: F (“(”)	FF (D“(”)	D-char: Любой допустимый символ, кроме {конца строки и “(”}

Формулы второго столбца таблицы 11 допускают более компактную свёртку, используя характерные для функционального программирования отображения и встречающийся в языках параллельных вычислений механизм просачивания, до вида:

FF ((H,Q,C,S,R,D), (>, ”, ', \,), ( ))

Такая формула означает, что в определении языка имеется шесть понятий для строк произвольной длины, имена которых начинаются с символов {**H,Q,C,S,R,D**} и среди символов каждого вида строк исключён соответствующий символ из списка { >, ", ', \,), (}.

4. Лексика, синтаксис, семантика и прагматика

Проблема определения ЯП и систем программирования (СП) была тщательно проработана в Венской методике определения языков программирования [12]. Основная идея заключается в использовании абстрактного синтаксиса (АС) и абстрактной машин*ы* (АМ) при определении семантики ЯП и отделения её от реализационной прагматики на конкретной машине (КМ). Трудоёмкость определения АС над АМ строго меньше трудоёмкости определения конкретного синтаксиса (КС) над КМ. КС языка отображается в АС, АМ может быть реализована с помощью КМ, причем и отображение, и реализация могут иметь небольшой объем и невысокую сложность. Схема 1 представляет основную модель декомпозиции определения СП для решения проблемы машинно-зависимого переноса программного обеспечения.

Диаграмма	*Пояснение*
КС ↔ **АС** ↓ **АМ**→КМ	Существует отображение **конкретного** синтаксиса (КС) в **абстрактный** (**АС**) и обратно. **АС** отображается в **абстрактную** машину (**АМ**) **АМ** реализуется с помощью **конкретной** машины (КМ).

Схема 1. Абстрактная декомпозиция определения ЯП, ориентированного на компиляцию программ по Венской

методике

По умолчанию, учитывая стандартную задачу компиляции программ, предполагается, что уровень АМ и КМ ниже уровня АС и КС, хотя на практике можно рассмотреть и другие соотношения. Роль формализма при такой методике выполняет прецедент реализации.

Для перехода от определения ЯП к реализации СП ключевое значение имеет семантика [6,7], но для эффективного программирования на любом ЯП требуется понимание не только условий эксплуатации программ, но и реализационной прагматики (РП), которая может быть не представлена в определении или стандарте ЯП, но подразумеваться традиционно или воспроизводится по прототипу. Выбор РП в высокой степени влияет на пространство процессов выполнения программ, существенно зависящего от аппаратуры. Реализация LLVM опирается на прагматику языка Си, обеспечивающую доступ к большому числу архитектур [14].

Реализационная прагматика, затрагивая все уровни определения ЯП, в основном предоставляет решения в области конкретной организации вычислений, уточняющей решения и принципы, провозглашенные в определении АМ. В первую очередь это относится к вопросам защиты областей памяти и их конечности, т.е. реагирования на дефицит памяти и учёт разницы в скоростях доступа. Следует сразу особо отметить, что основные традиционные ЯП апеллируют к операционной памяти, время жизни состояний в которой ограничено пределами времени исполнения конкретной программы. Обработка внешней и многоуровневой памяти, а также взаимодействия с независимо создаваемыми программами, возможно подверженными изменениям при исполнении, представляются как побочные эффекты. Это создает

проблемы перехода к большеобъёмным и распределённым данным, а также к синхронизации процессов.

Для основных ЯП определение АМ сводится к системе команд над небольшим числом регистров, обычно от двух до шести. Система команд может быть получена как отображение части базовых средств ЯП, а особенности их функционирования отражают реализационную прагматику обработки данных. На схеме 2 представлена модель решения проблемы переноса без привлечения машинно-ориентированных ЯП.

Диаграмма	*Пояснение*
ЯП1 ↔ **АС** ↔ **АМ** ↔ ЯП2	Существуют обратимые отображения - конкретного синтаксиса (ЯП1) в абстрактный (**АС**). - **АС** в абстрактную машину (**АМ**) . - **АМ** реализуется с помощью (ЯП2).

Схема 2. Абстрактная декомпозиция определения ЯП1, ориентированного на конвертацию в ЯП2 сравнимого уровня через два промежуточных языка

Такая схема представляет интерес для сравнения и оценки возможностей разных ЯП, особенно, обоснования проектов новых ЯП. При определении отображения **АС** ↔ **АМ** проявляется семантическая структуры обоих языков, допускающая выделение общих компонентов и определение специфики сравниваемых ЯП. Возможность обобщения таких подсистем приводит к схеме 3, ориентированной на укрупнение исходных средств ЯП и перехода от конкретных решаемых задач к более общим программным системам, допускающим оптимизацию и улучшение решений.

Диаграмма	*Пояснение*
АМ ↔ КМ □□□□□□□□□□ □□□□ КС ↔ **АС**	Существует отображение конкретного синтаксиса (КС) в абстрактный (АС) и обратно. **АС** приводится к командам **АМ** более высокого уровня. АМ реализуется с помощью КМ, возможно являющейся реализацией языка программирования.

Схема 3. Абстрактная декомпозиция определения кода программы для его обработки в стиле рекомпиляции

Такая схема встречается в реассемблерах и системах восстановления исходного текста программ по доступному коду, особенно если известен язык программирования. Она может быть полезна при организации параллельных вычислений (ПВ), требующих учёта эффектов, не имеющих прямого представления в ЯП.

Практичный ЯП обычно является композицией из небольшого ряда подъязыков, обладающих отдельной специальной семантикой — выражения, память, структуры и типы данных, функции и процедуры, препроцессоры, библиотеки.

Схема 4 представляет модель интеграции разных ЯП в более общую систему. Нередко реализация таких подъязыков может быть перенесена в другие ЯП без особых изменений. Аналогичное обобщение допустимо и по отношению к абстрактным машинам.

На схеме 5 представлена модель накопления системных решений в форме, не зависящей ни от КС, ни от КМ.

Диаграмма	*Пояснение*
{КС i ↔ **АС }** ↓ **АМ** → КМ	Существуют разные отображения подъязыков конкретного синтаксиса (КСi) в общий абстрактный (**АС**) и обратно. **АС** отображается в абстрактную машину (**АМ**). **АМ** реализуется с помощью конкретной машины (КМ) .

Схема 4. Абстрактное выделение подъязыков при определении ЯП по Венской методике

Диаграмма	*Пояснение*
КС ↔ {**АС** ↓ **АМj}** → КМ	Существует отображение конкретного синтаксиса (КС) в абстрактный (**АС**), **АС** отображается в ряд абстрактных машин (**АМi**), допускающих различные схемы выполнения. Каждая **АМj** может быть реализована с помощью одной и той же машины (КМ), но возможно по разному.

Схема 5. Абстрактное варьирование прагматики ЯП по Венской методике

Такая схема даёт возможность измерения характеристик различных характеристик ЯП и СП, варьирования разных схем организации процесса выполнения программ, включая одновременное существование интерпретации и компиляции а также

создания системных библиотек для прототипирования новых ЯП. Переход к ряду КМ, представленный на схеме 6, показывает модель настраиваемой кодогенерации.

Диаграмма	*Пояснение*
КС ↔ **АС** ↓ **{АМ**→ КМk**}**	Существует отображение конкретного синтаксиса (КС) в абстрактный (АС) **АС** отображается в абстрактную машину (АМ). **АМ** реализуется с помощью разных конкретных машин (КМk) **.**

Схема 6. Абстрактная декомпозиция кодогенерации при определении ЯП по Венской методике

Такая схема при переносе СП на новые архитектуры обеспечивает удостоверение эквивалентности результатов компиляции программ, а также обеспечивает возможность совместной эксплуатации разного оборудования, что может быть полезно при разработке программ для многопроцессорных конфигураций, а значит и для организации параллельных вычислений (ПВ).

5. Поддержка параллельных вычислений

Новые языки программирования содержат конструкции для представления параллельных процессов. Следует отметить, что сами по себе принципы параллелизма не являются сложными [9]. Все мы живём в мире множества взаимодействующих процессов и владеем интуитивным имплицитным невербализованным знанием о принципах их взаимодействия, поэтому правильная вербализация такого знания поможет сознанию, еще не привыкшему всё сводить к однозначности и последовательностям [3], открыть для себя понимание многообразия миров параллелизма. Нужна система обучения, приспособленная не только к ознакомлению с

отдельными эффектами параллельных вычислений, но и поддерживающая конструирование учебно-игровых тренажёров, пригодных для самостоятельного осознания особенностей взаимодействия процессов и для экспериментов по формированию навыков подготовки работоспособных программ и понимания новых возможностей аппаратуры. Мы живём в мире множества процессов и **владеем интуитивным знанием** о принципах их взаимодействия. Не исключено, что этому знанию противостоит исторически сложившаяся традиция одномерного представления текстов программ в виде бесконечной строки, а кроме того, недооценка полезности непрерывных и пространственных математических моделей. Решение проблем организации ПВ открывает путь реализации параллельных алгоритмов, включая их использование для определения новых ЯП на основе конструирования макетов реализации СП из существующих.

Можно рассмотреть несколько прецедентов неоднородного определения синтаксиса языков программирования, обладающих наглядностью, более гомоиконной параллельным процессам в сравнении с обычными ЯП. Первый такой пример принадлежит Конраду Цузе, создателю первого ЯП Plankalkül. Строки записи в этом ЯП имеют разный смысл: основная строка, задающая синтаксические позиции дальнейших строк, вспомогательная строка, показывающая индекс в обрабатываемом векторе, следующая строка задаёт границы обработки (Пример 1).

Привычная нотация в примере 2 программы отчасти утрачивает наглядность.

A [5] = A [4] + 1	Присваивание
| A + 1 => A V | 4 5 S | 1.n 1.n	| Основной поток обработки данных V | Индексирование доступа к памяти S | Точность представления чисел

Пример 1. Двумерное представление оператора присваивания

Линейный эквивалент двумерной программы на Plankalkül	В стиле Plankalkül
P1 max3 (V0[:8.0],V1[:8.0],V2[:8.0]) → R0[:8.0] max(V0[:8.0],V1[:8.0]) → Z1[:8.0] max(Z1[:8.0],V2[:8.0]) → R0[:8.0] END P2 max (V0[:8.0],V1[:8.0]) → R0[:8.0] V0[:8.0] → Z1[:8.0] (Z1[:8.0] < V1[:8.0]) → V1[:8.0] → Z1[:8.0] Z1[:8.0] → R0[:8.0] END	P1 | max3 V0, V1, V2 → R0 S | 8.0 8.0 8.0 → 8.0 | max (V0, V1) → Z1 S | → 8.0 | max (Z1, V2) → R0 P2 | max V0, V1 → R0 | V0 → Z1 | (Z1 < V1) → V1 → Z1 | | Z1 → R0

Пример 2. Вычисление наибольшего из трёх чисел — max3

Ещё один пример двумерного синтаксиса показан в работах по представлению таблиц решений как метода представления сложных ветвлений, на которое в начале 1970-ых обратил внимание Дж. Шварц при создании теоретико-множественного ЯП Setl, нацеленного на привлечение математической интуиции на решение программистских проблем. В таблице левая колонка содержит простые выражения, участвующие в выборе решений, а в остальных столбцах располагаются логические значения этих выражений, соответствующие каждому выбору. Возможно и в новых ЯП, нацеленных на параллелизм, понятнее будет такая техника таблиц

решений, на которую в начале 1970-х обратил внимание Дж. Шварц при создании языка Setl.

SETL (1972 J.T.Schwartz): ***ветвление как таблица решений***	вертикаль — связка & «И» логическое горизонталь — связка ^ «ИЛИ» логическое
IF c1, 'TTF' **c2, 'TFT'** **THEN (s1 ' X '** **s2 'XX '** **s3 ' X ')** **ELSE s4**	**c1, c2** - условия и варианты **'T'** и **'F'** их истинности **s1, s2, s3** - действия, соответствующие помеченным **'X'** столбцам значений условий нет подходящих условий

Пример 3. Общая схема таблицы решений

Такая таблица автоматически сводит алгоритм решения задачи к взаимодействию простых составляющих.

Когда результат не нужен – **_ (*подчерк)***	Столбец условий и реакций на их комбинации
IF x ε R, 'F T T' **f (x) ε T,'_ T F'** **y ε S; y > 0, '_ T _'** **THEN (x=o; 'X _ _ '** **x=y; '_ X _'** **R=R-{x}; '_ _ X'** **go to L1; '_ X X')** **ELSE go to L2;** // можно выбирать несколько // *(сверху вниз)*	x принадлежит R f(x) принадлежит T y принадлежит S если x **НЕ** принадлежит R если все условия выполнены, то два действия если f(x) **НЕ** принадлежит T, а x принадлежит R, то два действия если **НИ** один столбец условий не выполнен

Пример 4. Решение конкретной задачи

Всем известная Паскалевская лесенка мягко привлекает двумерность.

Двумерное представление цикла и переключателя	

repeat
  a := a + 1
until a = 10;
 	
case i of
  0 : Write('zero');
  1 : Write('one');
  2 : Write('two');
  3,4,5,6,7,8,9,10: Write('?')
end;
	Сдвиг вправо делает наглядной блочную вложенность без усложнения алгоритмов линейного распознавания текста

Пример 5. Визуализация иерархии блоков

Предложенное А.Л. Фуксманом вертикальное слоение экспериментально опробовано при реализации компилятора для фортраноподобного языка программирования Little. Одновременно был выполнен эксперимент по наполнению автомата семантическими действиями и интеграция подъязыков лексики, синтаксиса и семантики ЯП.

Синтаксис разных форматов оператора присваивания битовым строкам	*Пояснение*
присв = {.ИМП. **Функция (ИМП)** имя / ИМП = {**f , s**}/ | имя } { выр = | =} выр **f** = выр, выр, | **s** = выр, |	После выяснения значения фрагментной переменной ИМП выполняется подстановка одноимённой функции, от которой зависит дальнейший ход анализа.

Пример 6. Представление синтаксиса с использованием ряда синтаксических макрофункций {**f , s**}, встраиваемых в

итоговое определение синтаксиса при подстановке

Функция (ИМП)

Синтаксис языка Python использует двумерность для представления иерархии

Двумерное представление ветвлений и циклов	

n = int(input(
  'Type a number, then its factorial will be printed:'
))
if n < 0:
  print('Error: It must be a positive number')
else:
  fact = 1
  i = 2
  while i <= n:
  fact = fact * i
  i += 1
  print(fact)
	Хотя отказ от скобок блочной вложенности снижает устойчивость первичной подготовки текста программы, предпочитается освобождение от контроля баланса скобок.

Пример 7. Бесскобочное представление иерархии блоков

Одна их сложностей вербализации интуитивных образов связана с разнообразием индивидуальных интересов и успешного опыта. Поэтому программная поддержка такой работы должна быть устроена как система из ядра и оболочки, поддерживающих оперативную настройку на созвучные, привлекающие образы [3]. Ядро обеспечивает поддержку базовой семантики учебных игр подобно абстрактной машине языка программирования. Оболочка создаёт расширение ядра для конкретной игры подобно абстрактному

синтаксису языка программирования, дополненному пользовательским интерфейсом с информационным наполнением: картинки, тексты и вспомогательные процедуры по сюжетам, выбранные в соответствии с индивидуальными особенностями.

Прорисовки действий совместных процессов исполняются одновременно, позволяя демонстрацию явлений, присущих только миру параллелизма. Подразумевается, что каждая элементарная команда выполняется за фиксированный промежуток времени, равный для всех элементарных команд. Отдельный процесс для каждого исполнителя состоит из очереди шагов. Задавать каждому исполнителю предопределенную последовательность команд, которую можно изменять в процессе игры. Оболочка должна предоставлять грамотный графический интерфейс и являться буфером между вводом команд и работой ядра, облегчающим процесс управления игровыми объектами и позволяющим в понятной человеку форме визуализировать происходящее. Возможны различные реализации оболочек вокруг существующего интерфейса ядра. Имеется возможность запоминания истории действий и произвольно «передвигаться» внутри истории действий, произошедших на клеточном поле.

Любая учебная СП представляет собой набор уровней, допускающих их редактирование оператором с последующим переходом к выполнению программ. Одной из проблем при создании учебных систем программирования является сложность представления и понимания сложных логических условий управления действиями персонажей. Эта проблема обусловлена противоречием между интуитивной и формальной оценкой истинности формул.

Такое противоречие, имеющее лингвистический

характер, замечено в исследованиях Д. Канемана (лауреат Нобелевской премии 2011 года: *эксперимент «**ошибка конъюнкции**»*), показавших особенности интуитивной оценки истинности:
- интуитивная оценка вероятности истинности логических связок несколько отличается от математических правил (*кроме хорошо обученных математиков*);
- истинность связки «**И**» интуитивно человек оценивает **выше** истинности любого элемента в соответствии с лингвистическим смыслом «накопление;
- истинность связки «**ИЛИ**» оценивается **ниже**, воспринимается как разбиение на части, уменьшение.

Определение ПВ, особенно асинхронных, обычно требует ясных формулировок условий срабатывания действий. Даже простой показ проблем параллелизма на сюжете притчи о философах использует в решениях не вполне очевидные условия. В порядке эксперимента для учебного ЯП СИНХРО предложен таблично-вертикальный синтаксис для обучения организации ПВ. Семантика таблиц обобщена следующим образом: в качестве предикатов используются любые выражения, роль истинностных выполняют значения любого типа, выбор действия представляется размещением его представления или имени исполнителя в соответствующей столбцу клетке или в самой левой клетке, в которой могут быть размещены типовые действия, допускающими выполнение разными исполнителями.

Рассмотрим пример с двумя философами Ф1 и Ф2, расположенными по разные стороны стола. Каждый может двинуться в направлении стола на свободную клетку.

Движение_от_Двери_к_Столу Ф1 или Ф2:

Ф1 — ему нужна клетка *снизу,*
если она свободна,
то Ф1 идёт вниз

Ф2 — ему нужна клетка *сверху,*
если она свободна,
то Ф2 идёт вверх

Выражения для выбора действий Ф1 или Ф2:

если *((i < N/2) && (д [i] [j]==2) && (д [i+1] [j]==0))*

если *((i > N/2) && (д [i] [j]==2) && (д [i-1] [j]==0))*

Таблицы 12-14 показывают шаги методики подготовки таблицы решений при организации параллельных вычислений.

Таблица 12. Пример таблицы решений (Шаг 1 — колонка простых условий)

Условия реагирования **если**	**Ф2:** *ему нужна клетка сверху, если она свободна, то Ф2 идёт вверх*	**Ф1:** *ему нужна клетка снизу, если она свободна, то Ф1 идёт вниз*	**Исполнители:** *цель текущего шага*
i < (N/2-1)			Определение места на доске
i > (N/2+1)			
д [i][j]			Роль персонажа
д [i+1][j]			Свободна ли соседняя клетка
д [i-1][j]			
то			Действия

После уяснения комплекта простых условий, от которых зависит дальнейший ход событий, можно отдельно продумать действия каждого персонажа.

Таблица 13. Пример таблицы решений (Шаг 2 — выбор для первого философа Ф2)

если	*Ф2: ему нужна клетка сверху, если она свободна, то Ф2 пойдёт вверх*	*Ф1: ему нужна клетка снизу*	*Пояснения проверки истинностных значений для Ф2*
i < (N/2-1)	истина		Координаты для Ф2
i > (N/2+1)	_		Проверка не нужна
д [i][j]	**2**		**Это именно философ**
д [i+1][j]	0		Нужная клетка свободна
д [i-1][j]	_		Проверка не нужна
то	(д [i+1][j]=2 ; д [i][j]=0)		*Предстоящие действия Ф2 по изменению разметки доски*

Разобрались с одним, теперь уясняем, всё ли получается с другим.

Таблица 14. Пример таблицы решений (Шаг 3 — решения для второго философа Ф2)

если	*Ф2: ему нужна клетка сверху*	*Ф1: ему нужна клетка снизу, если она свободна, то Ф1 пойдёт вниз*	*Пояснения проверки истинностных значений для Ф2*
i < (N/2-1)	истина	_	Проверка не нужна
i > (N/2+1)	_	истина	Координаты для Ф1
д [i][j]	**2**	**2**	**Это именно философ**
д [i+1][j]	0	_	Проверка не нужна
д [i-1][j]	_	0	Нужная клетка свободна
то	(д [i+1][j]=2 ; д [i][j]=0)	(д [i-1][j]=2 ; д [i][j]=0)	*Предстоящие действия Ф1 по изменению разметки доски*

Таблица 15. Пример таблицы решений (Шаг 4 — распределение потоков действий для взаимодействия философов Ф1 и Ф2)

если	*Ф2: ему нужна клетка сверху*	*Ф1: ему нужна клетка снизу*	*Пояснение по распределению потоков действий*
i < (N/2-1)	истина	_	Определение позиции на доске
i > (N/2+1)	_	истина	
д [i][j]	**2**	**2**	**Это именно философ**
д [i+1][j]	0	_	Свободна ли соседняя клетка
д [i-1][j]	_	0	
то	д [i+1][j]=2	д [i-1][j]=2	Типовое действие вынесено в самый левый столбец
д [i][j]=0	Ф2	Ф1	Каждый исполнитель выполнит и типовое действие

Линеаризация таблицы решений для размещения в тексте программы может иметь вид текста, выравнивание колонок в котором достигается табуляцией, а строки выделяются концами строк

```
УСЛОВИЯ          Ф2                  Ф1      //Исполнители (имена философов)
(i < (N/2-1))    истина              _         //низ доски принадлежит Ф2
(i > (N/2+1))    НЕТ                 ДА        //верх доски принадлежит Ф1
(д [i][j])       2                   2         //философ занимает  клетку
(д [ i+1 ][ j ])  0                  _       //соседняя сверху свободна для Ф2
(д [ i-1 ] [ j])  _          0               //соседняя снизу свободна для Ф1
РЕШЕНИЕ     д [ i+1][ j ]=2   д [ i-1 ][ j ]=2     //Варианты реагирования
д [i][j]=0       Ф2                  Ф1        //Исполнители действий
```

Пример 8. Приведение итоговой таблицы к текстовой форме

Такой **таблично-вертикальный** синтаксис может помочь видеть и осознавать базовые **принципы** параллелизма, его основные проблемы и **явления,**

понимать **взаимодействия** параллельных процессов и конструировать программы с **заданным поведением,** более свободным от ложных лингвистических ассоциаций. Эффективным методом решения проблемы организации параллельных вычислений может стать постепенное внедрение в программы обучения программированию инструментов для ознакомления с основами параллельных явлений. Наиболее эффективной данная методика окажется для школьников, получающих возможность в простой и доступной игровой форме увидеть и осознать базовые принципы параллелизма, его основные проблемы и приобрести навыки решения простейших задач, имеющих отношение к взаимодействию параллельно исполняемых процессов.

Механизмы приведения грамматики языка к виду, удобному для автоматизации конструирования обработчиков программ, включая проверку текстов на возможность их включения в программы при макроподстановках и работе с фрагментными переменными, и определения грамматик с переводом позволяют принципиально расширить спектр базовых языков и средств обработки программ с накоплением их правильности при отладке. Возможен обратный вывод тестового покрытия программы для сравнения с предполагаемым тестовым набором. Декомпозиция программы по тестовым наборам может дать специализированные проекции, снижающие накладные расходы. Конверторы, освобождающие от лишних переходов на уровень машины, могут способствовать выделению эквивалентных подъязыков и, следовательно, типизации средств их реализации. Раскрытие недоопределённостей в программе при отладке можно выполнять по спецификациям, тестам или запросам. Приведение любых выражений к функциональной форме

уже активно используется в новых инструментах создания СП. Сдерживающим фактором является риск ошибки в компромиссе между автоматизацией и ручным управлением.

Заключение

Прогресс в области эксплуатационных характеристик оборудования даёт основания для пересмотра и развития подходов к организации обработки программ системами программирования. Средства и методы синтаксически ориентированного конструирования прошли серьёзные испытания предыдущей эволюцией языков и систем программирования в качестве инструментов снижения трудоёмкости программирования и обеспечения надёжности разрабатываемых программ. Возникает задача исследования их возможностей и резервов в новых эксплуатационных условиях. Прежде всего это расширение спектра парадигм программирования и рост актуальности ПВ [2]. Возможно, решение таких проблем связано с развитием средств и методов представления синтаксиса ЯП и его обогащение методами ФП и другими новыми формами.

Литература

1. Ахо А.В., Хопкрофт Дж.Э., Ульман Дж.Д. Структуры данных и алгоритмы. — М.: Вильямс, 2000. – 384 с.
2. Городняя Л.В. Парадигма программирования, Учебн. пос., 1-е изд. в библиотеку вуза. СПб. «Издательство

ЛАНЬ» (Учебники для вузов. Специальная литература), 2019 /– 232 с.
3. Городняя Л.В. Язык параллельного программирования Синхро, предназначенный для обучения. – Новосибирск, 2016. – 30 с. (Препринт/ИСИ СО РАН; № 180).
4. Зуев Е. История разработки компилятора Си++ по заказу иностранной фирмы в ранне-постсоветское время. - URL: http://www.gramotey.com/?open_file=1269097005
5. Касьянов В.Н., Евстигнеев В.А. Графы в программировании: обработка, визуализация и применение. – СПб.: ЕХП-Петербург, 2003. – 1104 с.
6. Лавров С.С. Методы задания семантики языков программирования. // Программирование. – 1978. – № 6. – С. 3-10.
7. Оллонгрен А. Определение языков программирования интерпретирующими автоматами. // М: Мир, 1977. – 288 с.
8. Хендерсон П. Функциональное программирование. – М.: Мир, 1983. – 349 с.
9. Хоар, Ч. Взаимодействующие последовательные процессы – М. : Мир, 1989. – 264 с.
10. Фуксман А.П. Технические аспекты создания программных систем. – М.: Статистика, 1979. – 180 с.
11. Baar T. Verification Support for a State-Transition-DSL Defined with Xtext. Perspectives of System Informatics - 10th International Andrei Ershov Informatics Conference, PSI 2015, in Memory of Helmut Veith, Kazan and Innopolis, Russia, August 24-27,2015. Revised Selected Papers // Lecture Notes in Computer Science 9609. – Springer Verlag, 2016. – P. 50-60.

12. Lucas P., Lauer P., Stigleitner H. Method and Notation for the Formal Definition of Programming Languges. IBM Laboratory – Venna, TR 25.087, 1968.
13. McCarthy J. LISP 1.5 Programming Mannual. – The MIT Press., Cambridge, 1963. – 106p.
14. http://clang.llvm.org/get_involved.html материалы по Clang — LLVM.

Л.В. Городняя

ПОДХОДЫ К ПРЕДСТАВЛЕНИЮ СИНТАКСИСА ЯЗЫКОВ ПРОГРАММИРОВАНИЯ

Препринт
185

Рукопись поступила в редакцию 29.12.2019
Редактор Т. М. Бульонкова
Рецензент Ф.А. Мурзин

Формат бумаги 60 × 84 1/16 Объем 1.71 уч.-изд.л., 1.88 п.л.
Тираж 50 экз.

Типография Оригинал-2, г. Бердск, ул. Олега Кошевого, 6, оф. 2
тел./факс: 8 (383) 328-32-38, (38341) 2-12-42, сот.: 8 913 987 77 67