УДК 004.43 БК 22.18 Г70

РЕЗЕРВЫ СИНТАКСИЧЕСКОГО КОНСТРУИРОВАНИЯ СИСТЕМ ПРОГРАММИРОВАНИЯ

Л.В. Городняя

Институт систем информатики им. А.П. Ершова Сибирского отделения Российской академии наук, Новосибирский государственный *университет,*
г. Новосибирск

lidvas@gmail.com

Аннотация

Работа посвящена анализу возможностей использования современного потенциала информационных технологий при решении задач обработки больших и сложных данных на примере текстов программ и определений языков программирования. Рассмотрена проблема совершенствования современных систем программирования и создания новых языков программирования, нацеленных на эффективное решение задач разработки надёжных и удобных информационных систем.

Ключевые слова*: системы программирования, декомпозиция программ, реализационная прагматика, определение языков программирования*

ВВЕДЕНИЕ

Основные подходы к реализации систем программирования (СП) опираются на инструментальную поддержку конструирования лексических и синтаксических анализаторов текста программы, дополненных средствами создания перевода программы на промежуточный язык (фронт-энд) и генератора исполнимого кода программы (бек-энд) [1]. Статистика использования таких средств при реализации новых СП показывает, что массово применяются конструкторы анализаторов, достаточно популярны средства создания фронт-энда (Clang-API, Lex, YACC и т. п.) и существенно реже используются средства создания бек-энда (LLVM) [2]. Таким образом, доминирует горизонтальное слоение СП, разработка которых сводится к не слишком трудоёмкой реализации лексики, синтаксиса, семантики

отдельно от наследуемой прагматики, дополненной комплексом библиотечных модулей [3–5].

Определённые трудности связаны со сложностью учёта динамики развития массово доступных информационных технологий и их приложений в разных областях, включая параллельные вычисления, распределенную обработку данных, популярный интернет, формальные методы обеспечения надежности и безопасности программ. Практичность применения инструментов конструирования СП зависит от непрерывной эволюции определения языков программирования (ЯП) и требований к эффективности и надёжности его СП, неизбежно происходящей при сроках разработки, превышающих полгода [6]. Кроме того, долгоживущие и новые ЯП характеризуются тенденцией к мультипарадигмальности, что означает требование поддержки вариантов реализации, используемых в зависимости от условий применения программ. Реализация таких особенностей обычно сказывается на всех уровнях горизонтального слоения СП, значительный объем которых требует дополнительного структурирования. Такое структурирование даёт методика вертикального слоения программ, в свое время предложенная А.Л. Фуксманом [7], концептуально близкая современному аспектно-ориентированному программированию, что применительно к СП может быть названо функционально-парадигмальной декомпозицией. Такую декомпозицию можно рассматривать как метод выбора критериев для согласованного выделения типовых компонентов в описании ЯП и определении СП, включая спецификацию эксплуатационных аспектов и реализационной прагматики, обеспечивающих варьирование используемых парадигм в зависимости от схемы жизненного цикла решаемых задач и требований к эффективности и производительности программируемых решений, что приводит к проблеме выделения автономно развиваемых подсистем.

СИНТАКСИС, СЕМАНТИКА И ПРАГМАТИКА

Проблема определения ЯП и СП была тщательно проработана в Венской методике определения языков программирования [8]. Основная идея – использование *абстрактного синтаксиса (АС)* и *абстрактной машины (АМ)* при определении *семантики* языка программирования и отделения её от реализационной прагматики на конкретной машине (КМ). Трудоёмкость определения АС над АМ строго меньше трудоёмкости определения КС над КМ. Конкретный

синтаксис (КС) языка отображается в АС, АМ может быть реализована с помощью КМ, причем и отображение, и реализация могут иметь небольшой объем и невысокую сложность [9].

Таблица 1. Абстрактная декомпозиция определения ЯП по Венской методике

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).

Современный инструментарий реализации ЯП в рамках проекта LLVM-Clang [2], включая новые проблемно-ориентированные языки (DSL), поддерживает подобную схему.

Таблица 2. Абстрактная декомпозиция определения DSL для реализации с помощью LLVM-Clang

Диаграмма	*Пояснение*
DSL → **AST** ↓ **IR** → MSP	Существует отображение конкретного синтаксиса (DSL) в дерево (AST). AST отображается на виртуальную машину (IR). IR реализуется с помощью конкретной платформы (MSP) на базе языка Си.

Интересно, что, хотя демонстрация Венской методики была выполнена на материале языка PL/1, неявную реализацию она получила в истории создания языка Lisp [10, 11]. Многие замечают явное сходство графового представления примеров AST, приведённых в руководстве по применению Clang [2] со списочным представлением Lisp-выражений, что позволяет создавать прототипы новых инструментов на базе Lisp-систем.

Таблица 3. Демонстрационный пример: текст программы на Си

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

Таблица 4. Демонстрационный пример: Clang-представление результата анализа программы на Си

Текст программы 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]
int param = 1;	**| |---->(VarDecl** param 'int')
	| |----->[| IntegerLiteral 1 'int' **|]**
myPrint (param);	|--->**[| CallExpr** 'void' **|]**
	| |--->[| ImplicitCastExpr 'void (*) ()' **|]**
	| | |--->[| DeclRefExpr 'myPrint' 'void (*) ()' **|]**
	| |----->[| ImplicitCastExpr 'int' **|]**
	| |--->[| DeclRefExpr 'param' 'int' **|]**
return 0;	|---->**[ReturnStmt]**
}	**|----->[| IntegerLiteral** 0 'int' |]

Таблица 5. 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))
	)
	)

Для перехода от определения ЯП к реализации СП ключевое значение имеет семантика [5], 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 существенно влияет на пространство процессов выполнения программ, неустранимо зависящего от аппаратуры. Реализация LLVM опирается на прагматику языка Си, обеспечивающую доступ к большому числу архитектур [2].

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

оперативной 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 (СК) над небольшим числом регистров, обычно от двух до шести. СК может быть получена как отображение части базовых средств ЯП, а особенности их функционирования отражают реализационную прагматику обработки данных.

НОВЫЕ ГОРИЗОНТЫ КОМПЬЮТЕРНЫХ ЯЗЫКОВ

Экстенсивное порождение предметно-ориентированных языков в XXI веке знаменует переход к методике конструирования компьютерных языков в качестве средства, позволяющего пользователю, умеющему работать с текстами, таблицами и визуальными оболочками на уровне Word-Excel и Visual-Studio, формировать грамматики, накрывающие пространство понятных решений текущих задач [12–15]. Происходящее в этом процессе уточнение такого пространства приводит к проблеме минимизации объёма изменений, возникающих в ранее сложившихся парадигмах программирования, нацеленных на борьбу за эффективность одно процессорных программ над оперативной памятью, рассчитанных на надёжность хранения данных на носителях, допускающих защиту и контроль [16].

Теперь пафос системного программирования переходит к обработке большеобъёмных слабо структурированных данных, не вполне защищённых от искажений, допускающих распределённую и многопроцессорную обработку. Организация параллельных вычислений, конструирование проблемно-ориентированных языков (DSL), обработка big data, обеспечение надёжности веб-сервисов, распределённых баз данных и многое другое [12, 17] требуют исследования и формализации специальных методов и парадигм компьютерных языков, в которых практично выделение типовых компонент, встраиваемых в различные информационные комплексы. Предстоит исследование новых эксплуатационных аспектов обработки данных и определение абстрактных машин,

полезных при конструировании реализационной прагматики компьютерных языков.

Системное программирование долгое время развивалось под прессом сервисных и заказных работ, нацеленных на создание инструментальных систем массового применения. Свойственный таким работам производственный подход опирается на предпочтение воспроизводимых процессов и стабильных программ, разрабатываемых для многократного использования. Ведущие критерии качества связаны с измеримыми характеристиками эффективности, такими как скорость выполнения готовых программ и объем используемой памяти. Для таких программ оправданы компиляционная схема обработки, статический анализ свойств, автоматизированная оптимизация и контроль. В этой области доминирует императивно-процедурный стиль программирования. Он допускает некоторую стандартизацию и модульное программирование, но обрастает довольно сложными построениями, спецификациями, методами тестирования, средствами сборки программ и т. п. Ещё в конце 1970-х годов В.Е. Котов [18] отмечал, что системному программированию нужна методика абстрагирования, сравнимая с дифференциальным и интегральным исчислением в математике.

Основные методы компиляции и традиции разработки систем программирования сложились в условиях низкой производительности оборудования, принудившей к предпочтению алгоритмов, ориентированных на минимизацию объёма хранимых данных, не влияющих на результат исполнения программы, даже если они полезны программисту при отладке [19, 20]. Жесткость требований к эффективности и надежности отчасти удовлетворяется разработкой профессионального инструментария, использующего сложные ассоциативно семантические эвристики наряду с методами синтаксически-управляемого конструирования и генерации программ. Бесспорный потенциал такого инструментария на практике ограничен трудоемкостью освоения – возникает квалификационный ценз, провоцирующий чрезмерное увлечение автоматизацией, что воспрепятствовало формированию средств и методов сознательного диалога «человек–машина» по поводу процесса разработки и отладки программ и программных систем. Границы между системами разработки программ, системами программирования, средствами препроцессора, языками программирования, стандар-

тами на языки программирования, традиционными методами их реализации, средствами сборки, отладки и тестирования программ визуально слабо подкреплены, нередко воспринимаются как однородная рабочая смесь воздействий на компонуемый по шаблонам текст, начинающий работать по волшебству.

Прогресс в области информационных технологий (ИТ) привёл к резкому увеличению объёмов доступной памяти, скоростей обработки данных, расширению возможностей применения параллельных вычислений и распределённых информационных сервисов, опережающим способность человека оценить новые горизонты. Изменился стартовый уровень интуитивного понимания особенностей обработки данных в сетях, а также проблем с появлением новых версий программного продукта, обновлением текущих версий, восстановлением данных при прерывании процесса, дистанционной поддержки пользователей, пользовательских интерфейсов и т. п. Всё это даёт основания не только для наращивания мощной, нередко бездумной, автоматизации обработки данных, но и для полного пересмотра походов к организации данных и переходу к интеллектуальному стилю обработки текстов программ в системах разработки программ и системах программирования, реабилитации дискредитированных преждевременной реализацией программистских идей, следуя принципам:

– Всегда доступно всё, что было представлено в программе, – комментарии, макросы и макровызовы, идентификаторы, описания и структуры данных;

– Система сохраняет все результаты анализа и любых эмпирических решений;

– Существует общий центр хранения библиотек, тестов, результатов прогона, документации и т. п. – всего, что может пригодиться в других программах;

– Система предназначена для серии сеансов отладки, включая сопоставление результатов разных сеансов;

– Поддерживается развитие постановки задачи (надзадачи, подзадачи);

– При формализации постановок задач, освобождающей от конкретных легенд, возможен диалог в исходных терминах;

– Преобразования данных обратимы – всегда можно восстановить исходный вид.

Такие принципы допускают поддержку работы программиста в удобных ему терминах, без принудительного освоения понятий разработчика 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.

Опыт применения регулярных выражений, реализации языков функционального программирования и современная тенденция к мультипарадигмальности новых языков программирования 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 систем программирования 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. Доступны средства вывода логики программы и её верификации на соответствие целям и требованиям. 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 языков программирования, рассматриваемых как специализированные программы, могут существенно расширить пространство средств информационной обработки.

Примечательны примеры типичных форм определения конструкций известных языков программирования, допускающие более лаконичные представления даже с помощью простых регулярных выражений, дополненных, в соответствии с традицией озвучивания математических формул, речевыми аналогами.

Таблица 6. 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 7. 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 8. 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

Таблица 9. 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 из которых - *unqualified-id*
id-expression: *unqualified-id* ***(*** *nested-name-specifier template* $_{opt}$ *unqualified-id* ***)***	**Представление идентификатора** – это *unqualified-id* ***или*** конструкция из трёх компонентов, последний из которых - *unqualified-id*

id-expression: *unqualified-id* ***(qualified-id:*** *nested-name-specifier template* $_{opt}$ *unqualified-id* ***)***	**Представление идентификатора** — это *unqualified-id* ***или*** конструкция с именем «***qualified-id****»,* *состоящая* из трёх компонентов, последний их которых - *unqualified-id*

Определение *qualified-id* встречается только один раз. Возможно его втягивание в головное определение или локализация

Таблица 10. 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 11. 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 непустое число присваиваний через запятую
expression: *assignment-expression* ***(*** ',' *assignment-expression* ***) ****	**Выражение** — это присваивание, затем любые число групп (возможно, ни одной) из запятой и присваивания

Привлечение аппарата функций над синтаксическими формулами позволяет намного повысить лаконизм и выразительность определений. 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 на базе грамматики с переводом позволяют принципиально расширить спектр базовых языков и средств обработки программ с накоплением их правильности при отладке. Возможен обратный вывод тестового покрытия про-

граммы 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.

Кроме того, целесообразно учесть возможность разумной обработки программ и настройки программных систем квалифицированным программистом, подобно тому как создатели операционных систем допускают существование **системного администратора**, способного понимать устройство операционной системы и её окружения, а потому имеющего права выполнять настройки на архитектурные особенности и другие воздействия на функционирование системы. На этом пути достаточно естественно выглядит допущение авторских преобразований программ с проверкой корректности их применения, сравнение разных схем обработки программ и их оптимизация, использование параллелизма обработки данных по частям, сбор статистики и построения гистограмм функционирования на разных тестовых наборах.

СИСТЕМНЫЙ АНАЛИЗ В ПРОГРАММИРОВАНИИ

Любые массово распространённые технологии апеллируют к понятию «система», и программирование в этом не исключение. Это понятие в конце XIX-го века привлекло внимание философов, сформулировавших большое число его свойств, за каждым из которых стоит великое имя, но редко встречается чёткая формулировка для определения границ понятия или наличия свойства «системность» в реальных комплексах. Существует общая теория систем без представления механизмов создания и функционирования систем.

Понятие «системный анализ» активно используется в менеджменте при организации производственных процессов на уровне интуитивного понимания. На этом фоне понятие «система» приобрело множество толкований, имеющих

отношение ко многим производным понятиям в практике создания и применения ИТ. Методы управления программистскими проектами используют системный анализ при подготовке постановки задачи к передаче разработчикам программ. В этом плане программирование обладает своей спецификой:

- Система команд – базовое понятие при определении компьютерных архитектур;
- Операционные системы – неотъемлемая составляющая любой практичной компьютерной конфигурации;
- Системный подход – методика учёта особенностей используемой аппаратуры при выборе эффективных решений;
- Система программирования – инструментальный комплекс, поддерживающий работу на языках программирования;
- Системное программирование – деятельность по созданию инструментальных информационных систем и комплексов;
- Информационные системы – средства обработки данных, рассматриваемых как представление информации о реально существующих объектах;
- Системная информатика – область знаний, исследующая закономерности создания и применения информационных систем и систем программирования.

Независимо от сферы приложения особенно важны жизнеспособные системы, обладающие устойчивостью, безопасностью и надёжностью. Часто говорят о системообразующих факторах, системных исследованиях, методах и процедурах. Ассоциация АПКИТ опубликовала стандарт на профессию «Системный архитектор».

Системные особенности программирования по существу связаны с присущей ему методикой обобщения частных прецедентов обработки конкретных данных, допускающего организацию многократно воспроизводимых процессов обработки типовых форматов данных. Такое обобщение содержательно напоминает описанные Пойа подходы к решению математических задач[6]. Основа так понимаемого системного анализа в программировании сводится к декомпози-

[6] Пойя Д. Как решать задачу. М.: Либроком, 2010. 208 с.

ции постановок решаемых задач на более общие и более конкретные задачи с определённым прогнозом по развитию обстановки применения решения задачи. При таком рассмотрении и задача, и программа её решения являются эволюционирующими объектами, что позволяет моделировать поведение программ как живых систем, используя результаты этологии. Интересно отметить, что К. Лоренц даёт вполне конструктивное определение понятия «система», сводит его к возможности в человеческой речи называть сложное одним словом[7].

ЗАКЛЮЧЕНИЕ

Сложность современных информационных систем требует специальной декомпозиции программ, допускающей ясное связывание с речевой практикой, что может дать вертикальное слоение по принципу функционально-парадигмальной декомпозиции, позволяющей разделять фрагменты по функциональному назначению и вариантам их реализации. Для разработчика программы мало заметна граница между системой программирования и системой разработки программ.

Разнообразие реалий современных ИТ трудно вписывается в модели обработки данных в защищённой оперативной памяти, что влечёт массовое появление новых проблемно-ориентированных языков программирования, требующее развития методов конструирования систем программирования, рассматриваемых как живые системы.

Прогресс в области эксплуатационных характеристик оборудования даёт основания для пересмотра и развития подходов к организации обработки программ системами программирования. Системность в системном программировании начинается с системного анализа доступных возможностей и тенденций их развития, дополненного систематическим накоплением типовых методов и отлаженного конструктива. Средства и методы синтаксически ориентированного конструирования СП прошли серьёзные испытания предыдущей эволюцией языков и систем программирования в качестве инструментов снижения трудоёмкости программирования и обеспечения надёжности разрабатываемых про-

[7] https://www.livelib.ru/book/1000334658-tak-nazyvaemoe-zlo-konrad-lorents – сайт с текстом книги Конрада Лоренца «Так называемое зло» в переводе А.И. Федорова (А.И. Фета).

грамм. Теперь стоит задача исследования их возможностей и резервов в новых эксплуатационных условиях.

Благодарности

Работа выполнена при финансовой поддержке Российского фонда фундаментальных исследований, проекты № 15-07-06345 и № 18-07-1048.

СПИСОК ЛИТЕРАТУРЫ

1. *Булычев Д.Ю., Вояковская Н.Н., Москаль А., Терехов А.А.* Разработка компиляторов. URL: http://www.intuit.ru/studies/courses/26/26/info

2. http://clang.llvm.org/get_involved.html материалы по Clang – LLVM

3. Ахо А.В., Хопкрофт Дж.Э., Ульман Дж.Д. Структуры данных и алгоритмы. М.: Вильямс, 2000.384 с.

4. *Городняя Л.В.* Парадигмы программирования: анализ и сравнение. *Сиб. Отделение Рос. Акад. наук, Ин-т систем информатики им. А.П. Ершова. Новосибирск: Изд-во СО РАН,* 2017. 232 с.

5. Лавров С.С. Методы задания семантики языков программирования// Программирование. 1978. № 6. С. 3–10.

6. *Зуев Е.* История разработки компилятора Си++ по заказу иностранной фирмы в раннее постсоветское время. URL: http: //www.gramotey.com/?Open_file=1269097005

7. *Фуксман А.Л.* Технические аспекты создания программных систем. М.: Статистика, 1979. 180 с.

8. *Lucas P., Lauer P., Stigleitner H.* Method and Notation for the Formal Definition of Programming Languges. IBM Laboratory – Venna, TR 25.087, 1968.

9. *Оллонгрен А.* Определение языков программирования интерпретирующими автоматами. М.: Мир, 1977. 288 с.

10. *Хендерсон П.* Функциональное программирование. М.: Мир, 1983. 349 с.

11. *McCarthy J.* LISP 1.5 Programming Mannual. The MIT Press, Cambridge, 1963. 106 p.

12. *Baar T.* Verification Support for a State-Transition-DSL Defined with Xtext. Perspectives of System Informatics – 10th Int. 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 2016. P. 50–60.

13. *Mernik M.* Formal and Practical Aspects of Domain-Specific Languages. IGI Global, 2012.

14. *Voelter M.* DSL Engineering: Designing, Implementing and Using Domain-Specific Languages, 2013. URL: http://voelter.de/dslbook/markusvoelter-dslengineering-1.0.pdf, http://dslbook.org/

15. *Taha W.* Domain-Specific Languages. Houston. 2009. URL: http://www. effective-modeling.org/p/walid-taha.html

16. *JetBrains,* Metaprogramming System MPS. URL: https://www.jetbrains. com/mps/

17. *Knoop J.* Compiler Construction. 20th Int. Conference, CC 2011. Held as Part of the Joint European Conferences on Theory and Practice of Software, Lecture Notes in Computer Sciences, 6601. ETAPS 2011 Saarbrcken, Germany, March 26 – April 3, 2011. Springer, 330 p.

18. *Котов В.Е.* МАРС: архитектура и языки для реализации параллелизма // Системная информатика. Вып. 1. Проблемы современного программирования. – Новосибирск: Наука. Сиб. отделение, 1991. С. 174–194.

19. *Крайнева И.А., Марчук А.Г.* Игорь Васильевич Поттосин. Из истории новосибирской школы программирования (к 80-летию со дня рождения) //Вестник НГУ. Серия: математика, механика, информатика. 2013. №1. С. 2–12.

20. *Крайнева И.А., Черемных Н.А.* Альфа-язык и транслятор // Открытые системы. 2014. №6. URL: http://www.novsu.ru/file/867726. – Открытые системы. СУБД 2016 № 01

RESERVES OF SYNTACTIC DESIGN OF PROGRAMMING SYSTEMS

L. V. Gorodnyaya

A.P. Ershov Institute of Informatics Systems, Siberian Branch of the Russian Academy of Sciences, Novosibirsk State University, Novosibirsk
lidvas@gmail.com

Abstract

The report is devoted to the analysis of the possibilities of using the modern IT potential in solving problems of processing large and complex data by the example of program texts and definitions of programming languages. The problem of improving modern programming systems and creating new programming languages aimed at efficiently solving problems of developing reliable and convenient information systems is considered.

Keywords*: programming systems, program decomposition, implementation pragmatics, definition of programming languages*

REFERENCES

1. *Bulychev D.Yu., Voyakovskaya N.N., Moskal A., Terekhov A.A.* Development of compilers. URL: http://www.intuit.ru/studies/courses/26/26/info

2. http://clang.llvm.org/get_involved.html Clang – LLVM.

3. *Akho A.V., Hopkroft J.E., Ulman J.D.* Data structures and algorithms. Moscow: Williams, 2000. 384 p.

4. *Gorodnyaya L.V.* Paradigms of programming: analysis and comparison / Rus. Acad. of Sciences, Sib. Branch, A.P. Ershov Institute of Informatics Systems. Novosibirsk: Izdat. SB RAS, 2017. 232 p.

5. *Lavrov S.S.* Methods for specifying the semantics of programming languages. // Programming. 1978. No 6. P. 3–10.

6. *Zuev E.* History of the development of the C ++ compiler on the order of a foreign company in the early post-Soviet period. URL: http: //www.gramotey.com/?Open_file=1269097005

7. *Fuksman A.L.* Technical aspects of creating software systems. Moscow: Statistics, 1979. 180 p.

8. *Lucas P., Lauer P., Stigleitner H.* Method and Notation for the Formal Definition of Programming Languges. IBM Laboratory – Venna, TR 25.087, 1968.

9. *Ollongren A.* Definition of programming languages by interpretive automata. Moscow: Mir, 1977. - 288 p

10. *Henderson P.* Functional programming. Moscow: Mir, 1983. 349 p.

11. *McCarthy J.* LISP 1.5 Programming Mannual. The MIT Press, Cambridge, 1963. 106 p.

12. *Baar T.* Verification Support for a State-Transition-DSL Defined with Xtext. Perspectives of System Informatics. 10th Int. 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, 2016. P. 50–60.

13. *Mernik M.* Formal and Practical Aspects of Domain-Specific Languages.IGI Global, 2012.

14. *Voelter M.* DSL Engineering: Designing, Implementing and Using Domain-Specific Languages, 2013. URL: http://voelter.de/dslbook/markusvoelter-dslengineering-1.0. pdf, http://dslbook.org/

15. *Taha W.* Domain-Specific Languages. Houston. 2009. URL: http://www.effective-modeling.org/p/walid-taha.html

16. *JetBrains,* Metaprogramming System MPS. URL: https://www.jetbrains.com/mps/

17. *Knoop J.* Compiler Construction. – 20th International Conference, CC 2011. Held as Part of the Joint European Conferences on Theory and Practice of Software, Lecture Notes in Computer Sciences, 6601. ETAPS 2011 Saarbrcken, Germany, March 26 – April 3, 2011. Springer. 330 p.

18. *Kotov V.E.* MARS: architecture and languages for realization of parallelism // System Informatics. Issue. 1. Problems of modern programming. Novosibirsk: Science. Sib. Branch, 1991. P. 174–194.

19. *Kraineva I.A., Marchuk A.G.* Igor Vasiljevich Pottosin. From the history of the Novosibirsk school of programming (to the 80th anniversary of his birth). Vestnik NSU, Serie: Mathematics, Mechanics, Computer Science. 2013. №1. P. 2–12.

20. *Kraineva I.A., Cheremnykh N.A.* Alpha-language and translator // Open systems. 2014. №6. URL: http://www.novsu.ru/file/867726. Open Systems. Database 2016 number 01.

СВЕДЕНИЯ ОБ АВТОРЕ

ГОРОДНЯЯ Лидия Васильевна – старший научный сотрудник Института систем информатики имени акад. Андрея Петрович Ершова СО РАН, доцент Новосибирского государственного университета, специалист в области системного программирования и образовательной информатики.

Lidia Vasiljevna GORODNYAYA – Senior Researcher of A.P. Ershov Institute of Informatics Systems, Siberian Branch of the Russian Academy of Sciences, Associate Professor of Novosibirsk State University, a specialist in system programming and educational informatics.

email: lidvas@gmail.com

Материал поступил в редакцию 5 сентября 2017 года