Подготовила Тырышкина Евгения, студентка 3 курса ИС, КазНУ

Содержание

1. Что такое Modula-3?

Modula-3 это системный язык программирования, который является продолжением Mesa, Modula-2, Cedar, и Modula-2+. Он также напоминает своих двоюродных братьев Object Pascal, Oberon, и Euclid.

Цель Modula-3 – стать настолько безопасным и простым, каким он может быть, учитывая потребности современных системных программистов. Вместо того, чтобы иследовать и придумывать новые детали, создатели языка изучили особенности семейства языков Modula, которые зарекоммендовали себя на практике, и попытались упростить их, тем самым получить более гармоничный язык.

Они выявили ряд удачных осбенностей, и обнаружили, что большинство из них было направлено на достижение двух главных целей: большая надежность и более простая система типов.

Modula-3 сохранил одну из лучших черт Modula-2 – явные интерфейсы между модулями. Он добавляет объекты и классы, обработку исключений, сбор мусора, легкие процессы (или темы), а также изоляцию небезопасных функций.

2. Области применения Modula-3?

Данный язык можно применять как для работы над индустриальными проектами, так и для решения различных исследовательских задач. Но на данный момент этот замечательный язык почти забыт и применяется только в зарубежных университетах, как инструмент для обучения студентов основам программирования. Modula-3 имеет отличный пользовательский графический интерфейс и содержит большой набор библиотек для распределенного программирования.

3. История разработки Модула-3

Разработка Modula-3 началась в 1986 году. Морис Уилкс (Maurice Wilkes) написал Никлаусу Вирту некоторые идеи по созданию новой версии Modula. Уилкс, до этого работавший в DEC, вернулся в Англию и устроился в исследовательский центр Olivetti. Никлаус Вирт в это время был занят разработкой нового языка программирования Oberon, но не отказал Уилксу в помощи. Описание Modula-3 было закончено в августе 1988 и исправлено в январе 1989 года. Тогда же появились компиляторы от DEC SRC и Olivetti, а также начали появляться компиляторы от сторонних фирм.
В девяностые годы прошлого века Modula-3 получила распространение преимущественно в академической среде, как язык для обучения программированию, и почти не использовалась в промышленности. Причиной этого могла послужить гибель DEC — основного разработчика языка. В то же время корпорацией Critical Mass был предложен коммерческий компилятор CM3 и интегрированная среда разработки Reactor. В 2000 году Critical Mass прекратила свою деятельность. В настоящее время техническую поддержку Modula-3 предоставляет корпорация Elego Software Solutions GmbH, которая унаследовала от Critical Mass исходные коды компилятора CM3. Интегрированная среда Reactor сейчас переименована в CM3 IDE и распространяется с исходными текстами. В марте 2002 года Elego получила исходные тексты компилятора PM3, до этого разрабатывавшегося в Ecole Polytechnique de Montreal (Политехнической школе Монреаля).

4. Цели и возможности

Девизом комитета Modula-3 в момент его формирования было выбрать простые, безопасные и проверенные возможности, которые должны быть включены в язык Modula-3. В конечном счете, Modula-3 был выполнен как смесь возможностей других языков и нескольких собственных. Возможности, выбранные комитетом были направлены на достижение следующих целей:

- обеспечить структурированность больших программ;
- увеличение безопасности и производительности программ;
- обеспечение программирования на машинном уровне при необходимости;
- простота.

Выше перечисленные цели являются компромиссом между BCPL- и LISP-подобными языками. Один обеспечивали эффективность и гибкость, другие – прекрасную модель программирования в угоду эффективности. Modula-3 направлен на устранение опасности, присущей BCPL-языкам, благодаря использованию строгой типизации, в то время как эффективность может быть улучшена написанием небезопасного машинно-ориентированного кода.

Теперь отметим возможности Modula-3, позаимствованные из других языков:

Функциональная возможность	Язык программирования
Сборщик мусора	Lisp
Замыкания	Lisp
Объекты	Simula, SmallTalk
Потоки	Mesa, Cedar
Исключения	Clu
Дженереки (Generics)	Ada
Модули	Modula-2

В дополнение, Modula-3 реализует несоклько новых уникальных возможностей:

- изоляцию безопасного и небезопасного кода;
- мощную и простую систему типов;
- многоуровневую абстракцию.

В целом, объекты, интерфейсы и модули, потоки и дженерики – это то, что обеспечивает структурированность больших программ, а остальные возможности языка полезны для безопасного, производительного системного программирования. Простота заключается в наличии всех этих возможностей. В последующих главах мы сфокусируемся на тех

возможностях языка, которые мы считаем наиболее важными и уникальными для языка Modula-3.

5. Модули и интерфейсы

Модули являются основными строительными блоками программ, написанных на языке Modula-3. Они обеспечивают самую внешнюю область видимости: в дополнение к именам переменных объявленных внутри модуля, видны имена только из импортированных интерфейсов.

Модули взаимосвязаны через интерфейсы. Интерфейс определяет только ту часть модуля, которая видна всем. Интерфейс сожержит только объявления, определения же всегда находиться в модуле, который экспортирует его.

Структура файла соответствует модульно/интерфейсной структуре: каждый модуль расположен в .m3-файле с аналогичным именем, а интерфейс – в файле .i3 с именем файла, совпадающим с именем интерфейса.

Модули – это как блоки, за исключением видимости имен. Сущность видима в блоке, если она объявлена в блоке или в некотором окружающем блоке; сущность видима в модуле, если она объявлена в модуле или в интерфейсе, который импортируется или экспортируется модулем.

Интерфейс представляет собой группу объявлений. Объявления в интерфейсах такие же, как в блоках, за исключением того, что любые переменные инициализации должно быть постоянным, и описания процедур должны указывать только сигнатуру, а не тело.

Модуль X экспортирует интерфейс I для реализации одной или более процедур, объявленных в интерфейсе. Модуль или интерфейс X импортирует I, чтобы сделать сущности объявленные в I видимыми в X.

Программа это коллекция модулей и интерфейсов, которые содержат каждый интерфейс, импортированный или экспортированный любым модулем или интерфейсом, и в котором нет множественно определенной процедуры, модуля или интерфейса.

Выполнение всей программы является выполнением тела каждого её модуля. Порядок выполнения модулей задается правилами инициализации.

Модуль, тело которого выполнялось последним, называется основным модулем (main). Основной модуль должен быть указан непосредственно в реализации, так как он не может быть однозначно определен правилами инициализации. Рекомендуется, чтобы основным модулем был тот модуль, который экспортирует интерфейс Main, чье содержимое зависит от реализации.

Согласно конвенции, основная программа расопложена в модуле с именем Main. Но также может быть использовано любое имя модуля, но модуль обязательно должен реализовывать интерфейс Main.

Выполнение программы прекращается, когда тело главного модуля прекращается, даже если одновременно нити управления по-прежнему выполняется.

Имена модулей и интерфейсов программы называются глобальными именами. Метод для поиска глобальных имен зависит от реализации.

Модуль может эксортировать один или несколько интерфейсов. Если нет ни одного явно указанного интерфеса, то модуль реализует интерфейс с именем аналогичным имени самого модуля. Однако интерфейсы должны быть всегда указаны явно инаходиться в файле **.i3**. Явное определение интерфейсов позволяет избежать наложения пространств имен.

Модули также являются наменьшей частью программы, которая может быть общей или небезопасной. Самой большой частью доступной в языке Modula-3 являются пакеты.

Типчиный модуль выглядит следующим образом:

```
MODULE m EXPORTS i1;
    IMPORT i2;
    PROCEDURE f();
    VAR x;
BEGIN
    (* тело модуля *)
```

END m.

Здесь **i1** – это интерфейс, который модуль **m** реализует. В то время как **i2** интерфейс, реализованный где-то еще.

5.1 Операторы импорта

Есть две формы операторов импорта. Все импорты обеих форм интерпретируются одновременно: их порядок не имеет значения.

Первая форма:

IMPORT I AS J

Первая форма импортирует интерфейс, чье глобальное имя **I** и дает ему локальное имя **J**. Сущности объявленные в **I** становятся доступны в импортирующем модуле или интерфейсе, но сущности импортиртированные в **I** – нет. Чтобы обратиться к сущности объявленной под именем **N** в интерфейсе **I**, импортер должен использовать квалифицированный идентификатор **J.N**.

Вместо IMPORT I AS I можно писать коротко IMPORT I.

Вторая форма:

FROM I IMPORT N

Вторая форма вводит N в качестве локального имени для лица, объявленного как **N** в интерфейсе **I**. Локальная связка для **I** имеет приоритет над глобальной связкой.

Например:

IMPORT I AS J, J AS I; FROM I IMPORT N

одновременно вводит местные названия **J**, **I**, и **N** для лиц, чьи глобальные имена **I**, **J**, и **J.N**, соответственно.

Запрещено использовать одно и то же имя дважды.

Например:

IMPORT J AS I, K AS I;

- является статической ошибкой, даже если **J** и **K** являются одинаковыми.

5.2 Интерфейсы

Интерфейс имеет форму:

INTERFACE id;

Imports;

Decls

END id.

где **id** идентификатор, задающий имя интерфейса, **Imports** – последовательность операторов импорта, и **Decls** – последовательность объявлений, которая не содержит процедур или блоков инициализации неконстантых переменных. Имена, объявленные в **Decls**, и видимые импортируемые имена должны различаться. Формирование цикла импорта для двух или более интерфейсов вызовет статическую ошибку.

5.3 Модули

Модуль имеет форму:

MODULE id EXPORTS Interfaces;

Imports;

Block id.

где id – это идентификатор, который задает имя модуля, **Interfaces** – это лист различных имен интерфейсов, экспортируемых модулем, **Imports** – это лист операторов импорта, и **Block** – это блок, тело модуля. Имя **id** должно повторяться после ключевого слова **END**, которое завершает

тело. “EXPORTS Interfaces” могут быть опущены, в таком случае Interfaces по умолчанию принимают значение **id**.

Если модуль **M** экспортирует интерфейс **I**, тогда все объявленные имена в **I** видны без указания в **M**. Любые процедуры, объявленные в **I** могут быть повторно объявлены в **M**, с телом. Сигнатура в **M** должна быть покрыта сигнатурой в **I**. Чтобы определить интерпритацию привязок ключевых слов и параметров по умолчанию в вызовах процедуры, сигнатура в **M** используется в **M**; сигнатура **I** используется везде.

За исключением повторного объявления экспортируемых процедур, имена, объявленные в верхнем уровне **Block**, видимые импортируемые имена и имена, объявленные в экспортированных интерфейсах, должны быть различны.

Например, следующее является недопустимым, так как два имени экспортируемых интерфейсов совпадают:

```
INTERFACE I;
  PROCEDURE X();  ...
INTERFACE J;
  PROCEDURE X();  ...
MODULE M EXPORTS I, J;
  PROCEDURE X() = ...;
```

Следующее также является недопустимым, так как видимое импортируемое имя **X** совпадает с именем **X** верхнего уровня:

```
INTERFACE I;
  PROCEDURE X();  ...
MODULE M EXPORTS I;
  FROM I IMPORT X;
  PROCEDURE X() = ...;
```

Но следующее допустимо, хотя и своеобразно:

```
INTERFACE I;
  PROCEDURE X(...);  ...
```

```
MODULE M EXPORTS I;
  IMPORT I;
  PROCEDURE X(...) = ...;
```

поскольку единственное видимое импортируемое имя **I**, и совпадение между **X** верхнего уровня и **X** в экспортируемом модуле допустимо, подразумевая, что сигнатура интерфейса покрывает сигнатуру модуля. В **M** объявление интерфейса определяет сигнатуру **I. X** и объявление модуля определяет сигнатуру **X**.

5.4 Пример модуля и интерфейса

Ниже представлен канонический пример публичного стэка с закрытым представлением:

```
INTERFACE Stack;
  TYPE T <: REFANY;
  PROCEDURE Create(): T;
  PROCEDURE Push(VAR s: T; x: REAL);
  PROCEDURE Pop(VAR s: T): REAL;
END Stack.
MODULE Stack;
  REVEAL T = BRANDED OBJECT item: REAL; link: T END;
  PROCEDURE Create(): T = BEGIN RETURN NIL END Create;
  PROCEDURE Push(VAR s: T; x: REAL) =
    BEGIN
      s := NEW(T, item := x, link := s)
    END Push;
  PROCEDURE Pop(VAR s: T): REAL =
    VAR res: REAL;
```

```
    BEGIN
      res := s.item; s := s.link; RETURN res
    END Pop;
  BEGIN
  END Stack.
```

Если представление стеков требуется более чем в одном модуле, он должен быть перемещен в частный интерфейс, так что он может быть импортирован везде, где это необходимо:

```
  INTERFACE Stack  (* ... как прежде ... *) END Stack.
  INTERFACE StackRep; IMPORT Stack;
    REVEAL Stack.T = BRANDED OBJECT item: REAL; link: Stack.T END
  END StackRep.
  MODULE Stack; IMPORT StackRep;
    (* Push, Pop и Create как прежде *)
  BEGIN
  END Stack.
```

5.5 Стандартное ядро языка

В стандартном интерфейсе или модуле некоторые имена импортированных интерфейсов рассматриваются как формальные параметры, чтобы граничить с фактическими интерфейсами, когда обобщенность установлена.

In a generic interface or module, some of the imported interface names are treated as formal parameters, to be bound to actual interfaces when the generic is instantiated.

Стандартный интерфейс имеет форму

```
  GENERIC INTERFACE G(F_1, ..., F_n);
    Body
```

END G.

где **G** – это идентификатор, который называет стандартный интерфейс, **F_1, . . . ,F_n** – это список идетификаторов, называемых формальными импортами **G**, и **Body** – это последовательность импортов следующих за последовательностью объявлений, в точности как в нестандартных интерфейсах.

Экземпляр класса **G** имеет форму

INTERFACE I = G(A_1, ..., A_n) END I.

где **I** – это имя экземпляра и **A_1, . . . , A_n** – это список фактических интерфейсов с которым формальные импорты **G** связаны. Экземпляр **I** эквивалентен обычному интерфейсу, определенному следующим образом:

```
INTERFACE I;
  IMPORT A_1 AS F_1, ..., A_n AS F_n;
  Body
END I.
```

Общий модуль имеет форму

```
GENERIC MODULE G(F_1, ..., F_n);
  Body
END G.
```

где **G** идентификатор, который называет стандартный модуль, **F_1, . . .**, **F_n** – это список идентификаторов, называемых формальными импортами **G**, и **Body** – это последовательность импортов следующих за блоком, в точности как нестандартный модуль.

Экземпляр **G** имеет форму

MODULE I EXPORTS E = G(A_1, ..., A_n) END I.

Где **I** – это имя экземпляра, **E** – список интерфейсов экспортированных **I**, и **A_1, . . . , A_n** – это список фактических интерфейсов с которыми формальные импорты **G** связаны. “EXPORTS E” могут быть опущены, в таком случае подразумевается “EXPORTS I”. Экземпляр **I** эквивалентен обычному модулю, определенному следующим образом:

```
MODULE I EXPORTS E;
  IMPORT A_1 AS F_1, ..., A_n AS F_n;
  Body
END I.
```

Обратите внимание, что сам по себе стандартный модуль не имеет экспортов.

Например, здесь стандартный стек:

```
GENERIC INTERFACE Stack(Elem);
  TYPE T <: REFANY;
  PROCEDURE Create(): T;
  PROCEDURE Push(VAR s: T; x: Elem.T);
  PROCEDURE Pop(VAR s: T): Elem.T;
END Stack.
GENERIC MODULE Stack(Elem);
  REVEAL
    T = BRANDED OBJECT  n: INTEGER;  a: REF ARRAY OF Elem.T  END;
  PROCEDURE Create(): T =
    BEGIN  RETURN NEW(T, n := 0, a := NIL)  END Create;
  PROCEDURE Push(VAR s: T; x: Elem.T) =
    BEGIN
      IF s.a = NIL THEN
        s.a := NEW(REF ARRAY OF Elem.T, 5)
      ELSIF s.n > LAST(s.a^) THEN
        WITH temp = NEW(REF ARRAY OF Elem.T, 2 * NUMBER(s.a^)) DO
          FOR i := 0 TO LAST(s.a^) DO temp[i] := s.a[i] END;
          s.a := temp
        END
      END;
```

```
      s.a[s.n] := x;
      INC(s.n)
    END Push;
  PROCEDURE Pop(VAR s: T): Elem.T =
    BEGIN  DEC(s.n);  RETURN s.a[s.n]  END Pop;
BEGIN
END Stack.
```

Чтобы создать стэк целых чисел:

```
INTERFACE Integer; TYPE T = INTEGER; END Integer.
INTERFACE IntStack = Stack(Integer) END IntStack.
MODULE IntStack = Stack(Integer) END IntStack.
```

Во время реализации не ожидается деление кода между двумя экземпялрами общего модуля, так как это невозможно.

6. Система типов

Типы обеспечивают три главных цели: структурирование данных, инструмент спецификации и контроль целостности. Мы рассмотрим все эти три цели в рамках языка Modula-3.

Структурированность данных – Modula-3 обеспечивает следующие типы данных: массивы, записи, объекты и упакованные типы. Записи предназначены для структурирования различных видов данных в единую сущность. Объектные типы предназначены для реализации объекто-ориентированной концепции, где определенный набор операций предназначен для выполнения над определенным набором данных. Упакованные типы позволяют обращаться к определенным битам как к членам типа – это дает предсказуемое отображение данных в аппаратной части.

Система типов как инструмент спецификации обеспечивает концепцию, называемую прозрачной типизацией, которая может использоваться

одновременно как инструмент спецификации, так и для сокрытия информация. Эти оба понятия тесно связаны друг с другом. Записи и объекты – определяют набор полей для конкретного типа. Эти поля состоавляют то, что указывает тип, и то, какие операции можно выполнить с данным типом. Для одного типа в зависимости от того, где он используется, может быть несколько реализаций в различных модулях. Следствием использования такой системы является то, что спецификация типа в одном модуле, скрыта для другого, реализующего такой же тип. Это и есть механизм сокрытия информации.

Что касается механизма контроля целостности, Modula-3 поддерживат такие возможности как Структурная Эквивалентность и Строгая типизация для сохронения простоты и однообразности системы. Однако, как мы далее увидим, Структурная Эквивалентность имеет свои проблемы и чтобы разрешить их, Modula-3 обеспечивает эквивалентность имен при необходимости.

7. Объекты

Объекты в Modula-3 являются новинкой в сравнении с Modula-2. Реализация объектов сведена к минимуму: нет множественного наследования и перегрузки операторов.

Единственное наследование (совместно со статической типизацией) вписывается в основные требования Modula-3 как языка для системного программирования: это гарантирует, что методы будут вызываться со скоростью (O(1)). Единственное наследование обеспечивается деревом объектов, в корне которого находиться объект ROOT. Однако Modula-3 не является полностью объекто-ориентированным языком: встроенные типы не являются объектами и не могут быть расширенны пользователем.

Объекты – это ссылки на блок данных и набор методов. Как ссылки они могут быть отслеживаемыми и нет. Так что по факту существует два дерева объектов: одно с корневым объектом ROOT и другое – с UNTRACED ROOT. Также, как ссылки объектые типы могут быть BRANDED чтобы избежать структурного равенства с другими типами

объектов.

В Modula-3 нет конструкторов (инициализацию объекта выполняет метод по-умолчанию). Согласно конвенции такой метод называется init и вызывается явно, возвращая объект после его инициализации: VAR m := NEW(Matrix).init();

Переопределение методов выполняется явно:

```
TYPE ST = OBJECT
    METHODS
        m1() := P;
        m2() := P;
    END;
TYPE T = ST OBJECT
    METHODS
        m1() := Q;
    OVERRIDES
        m2() := Q;
    END
a = NEW(T);
NARROW(a, ST).m1(); (* activates Q *)
NARROW(a, ST).m2(); (* activates P *)
```

Заметьте, что определения объектов не включают имплиментации методов: методы привязаны к процедурам, описанным где-то еще. Такие процедуры должны явно использоваться совместимый **self** (тип объекта или супертип) как первый аргумент, тогда как остальные параметры должны соответствовать сигнатуры метода (такое поведение было позаимствовано из Python).

8. Выражения

Выражение предписывает вычисление, которое производит значение или переменную. Синтаксически выражение, иначе операнд, или операция применяется к аргументам, которые сами по себе выражения. Операнды являются идентификаторами, литералами или типами. Выражение вычисляется путем рекурсивной оценки своих аргументов и выполнения операции. Порядок вычисления аргументов не определен для всех операций, кроме И и ИЛИ.

8.1 Соглашения для описания операций

Для описания аргументов и типов операций, мы используем обозначения, такие как процедурные сигнатуры. Но так как большинство операций носит слишком общий характер, чтобы быть описаными истинной процедурной сигнатурой, мы расширяем обозначения несколькими способами.

Аргумент операции может быть обязан иметь тип в конкретном классе, например, порядковый тип.

Например:

ORD (x: Ordinal): INTEGER

Формальный тип **ANY** задает аргумент любого типа.

Односложное имя операции может быть перегружено. Это значит, что оно обозначает несколько операций. В таком случае, необходимо написать отдельную сигнатуру для каждой операции. Например:

```
ABS  (x: INTEGER)    : INTEGER
     (x: Float)      : Float
```

Определенная операция будет выбрана так, чтобы каждый фактический тип аргумента являлся подтипом соответсвующего формального типа или членом соответсвующего класса.

Аргумент операции может быть выражением, обозначающим тип. В этом случае, мы пишем тип как тип аргумента. Например:

BYTESIZE (T: Type): CARDINAL

Тип результата операции может зависеть от его значений аргументов (хотя тип результата всегда можно определить статически). В этом случае выражение для типа результата содержит соответствующие аргументы. Например:

FIRST (T: FixedArrayType): IndexType(T)

IndexType (T) обозначает тип указателя массив типа **T** и **IndexType (a)** обозначает тип индекса массива **a**. Определения **ElemType (T)** и **ElemType (a)** схожи.

9 Синтаксис

Операторы, которые имеют специальный синтаксис классифицируются и перечислены в порядке убывания силы связывания в следующей таблице:

x.a	infix dot
f(x) a[i] T{x}	applicative (, [, {
p^	postfix ^
+ -	prefix arithmetics
* / DIV MOD	infix arithmetics
+ - &	infix arithmetics
= # < <= >= > IN	infix relations
NOT	prefix NOT
AND	infix AND
OR	infix OR

Все инфиксные операторы левоассоциативны. Скобки могут использоваться, чтобы не подчиняться правилам приоритета. Вот некоторые примеры выражений вместе с их формами, полностью помещенными в скобки:

M.F(x)	(M.F)(x)	dot before application
Q(x)^	(Q(x))^	application before ^
- p^	- (p^)	^ перед префиксом -
- a * b	(- a) * b	префикс - перед *
a * b - c	(a * b) - c	* перед инфиксом -
x IN s - t	x IN (s - t)	инфикс - перед IN
NOT x IN s	NOT (x IN s)	IN перед NOT
NOT p AND q	(NOT p) AND q	NOT перед AND
A OR B AND C	A OR (B AND C)	AND перед OR

Операторы без специального синтаксиса являются процедурными. Применение процедурного оператора имеет вид **op** (аргументы), где **op** операция и **args** представляют собой список аргументов-выражений. Например, MAX и MIN процессуальные операторы.

9.1 Ключевые слова

AND	FINALLY	RECORD
ANY	FOR	REF
ARRAY	FROM	REPEAT
AS	GENERIC	RETURN
BEGIN	IF	REVEAL
BITS	IMPORT	ROOT
BRANDED	IN	SET
BY	INTERFACE	THEN
CASE	LOCK	TO
CONST	LOOP	TRY

DIV	MOD	TYPE
DO	MODULE	TYPECASE
ELSE	NOT	VALUE
ELSIF	OBJECT	VAR
END	OF	UNSAFE
EVAL	OR	UNTIL
EXCEPT	OVERRIDES	UNTRACED
EXCEPTION	PROCEDURE	WHILE
EXIT	RAISE	WITH
EXPORTS	RAISES	
METHODS	READONLY	

9.2 Зарезервированные идентификаторы

ABS	FIRST	NIL
ADDRESS	FLOAT	NULL
ADR	FLOOR	NUMBER
ADRSIZE	ISTYPE	ORD
BITSIZE	INC	REFANY
BOOLEAN	INTEGER	REAL
BYTESIZE	LAST	ROUND
CARDINAL	LONGREAL	SUBARRAY
CEILING	LOOPHOLE	TEXT
CHAR	MAX	TRUE
DEC	MIN	TRUNC
DISPOSE	MUTEX	TYPECODE
EXTENDED	NARROW	VAL
FALSE	NEW	

9.3 Операторы

+	<	#	=	;	..	:
-	>	{	}	|	:=	<:
*	<=	(	)	^	,	=>
/	>=	[	]	.	&	

9.4 Комментарии

Комментарий – это произвольная последовательность, открывающаяся символами **(*** и закрывающаяся ***)**. Комментарии могут быть вложенными и продолжаться более одной строки.

9.5 Прагмы

Прагма – это способ передачи управляющей информации компилятору. Это может быть, в частности, директива препроцессора, указывающая компилятору, как следует трактовать конкретный оператор программы.
Прагма представляет собой произвольную последовательность символов, открывающуюся **<*** и закрывающуюся ***>**. Прагмы, как и комментарии, могут быть вложенными и продолжаться более одной строки. Прагмы это подсказки реализации; они не влияют на семантику языка.
Рекомендуется поддерживать **<*INLINE*>** и **<*EXTERNAL*>**. Прагма **<*INLINE*>** предшествует объявлению процедуры, чтобы указать, что функция должна быть расширена в точке вызова. Прагма **<*EXTERNAL N: L*>** предшествует интерфейсу или объявлению в интерфейсе, чтобы указать, что сущность, которой она предшествует, выполнена на языке **L**, где он имеет название **N**. Если “: L” опущен, то внешний язык осуществления предполагается по умолчанию.

9.6 Соглашение о синтаксисе.

Используются следующие соглашения для определения синтаксиса:

X Y X следует за Y
X|Y X или Y.
[X] X или пусто
{X} Возможно, пустая последовательность X-ов
X&Y X или Y или X Y

“Следует за” имеет большую связывающую силу, чем | или &. Для изменения приоритета используются скобки. Нетерминалы начинаются с прописной буквы. Терминалами являются либо ключевые слова, либо указанные операторы.

10. Заключение

Modula-3 является мощным и безопасным языком программирования, который совмещает в себе низкоуровневые возможности сравнимые с языком C++ и возможности высокроуровневых фреймворков, такие как сборщик мусора, безопасность, и явная модульная структура программ.

Однако, этот язык не обрел жизни в индустрии, веротно из-за отсутствия спонсоров. Язык был придуман для широкомасштабных систем, но конечные пользователи в большинстве случаев используют его для обучения и научных исследований.

Не смотря на это очень элегантный язык, очень привликательный в теории, но большим минусом является отсутствие эффективного компилятора. Есть несколько компиляторов, большинство из них с открым исходным кодом, но они не достигли выского уровня зрелости.

Исходя из этого, язык Modula-3 уплывает в историю слишком рано, но его наследние произвело сильное влияние на такие языки как Python и C#.