A *composite data type* defined inductively from other types. Typically, each type has a number of cases or alternatives, which each case having a *constructor* with zero or more arguments.
For example, `data Expr = Int(int i) | Plus(Expr a, Expr b)`.

Flash-225

FLASH225™

FLASH225™

Ambiguous grammar

Ambiguity,Syntax

A tree representation of the syntactic structure of a sentence; similar to a *parse tree*, but usually ignoring literal *nonterminals* and nodes corresponding to *productions* that don't directly contribute to the structure of the language.

A property of binary operators in parsing, indicating whether expressions such as $a + b + c$ should be interpreted as $(a + b) + b$ (left associative), $a + (b + c)$ (right associative) or as illegal (non-associative).

Semantics

Anonymous function

FLASH225™

FLASH225™

Application binary interface

Compilation

Requires a *generalised parser*, produces a *parse forest*.

Check your SLE vocabulary!

Instructions: Use alone or with a friend. Look at one side of a card, try to remember as much as possible about the term on the card before checking the back. Can also be used from other side; try to remember which term goes with the description. Once you know a concept, put it aside and review it from time to time.

Semantics,Types

Algebraic data type

flash225™

iii flash225™

Abstract syntax tree

Abstraction,Parsing,Syntax iii

A grammar for which there is a string which has more than one leftmost *derivation*.

A function occuring as a value, without being bound (directly) to a name. C.f. *closure*.

!!! Ambiguity,Syntax

Associativity

flash225™ !!!

∈ flash225™

Ambiguous grammar — Parsing

Ambiguity,Syntax ∈

Specifies how software modules or components interact with each other at the machine code level. Typically includes such things function calling conventions (whether arguments are passed on the stack or in registers, and so on), the binary layout of data structures and how system calls are done.

A *disambiguation rule* stating that an operator is either left-, right- or non-associative. E.g., in Rascal: `syntax Expr = left (Expr "*" Expr | Expr "/" Expr );`

Associativity — Left

Backend

Operations are grouped on the right, giving a tree which is "heavy" on the right side; typically used for assignment and exponentiation operators.

A data type constructed from other types (or itself, in the case of a *recursive data type*), e.g., a *structure* or an *algebraic data type*.

wiki=http://en.wikipedia.org/wiki/Component-based_software_engineering]

Bottom-up parser

Closure

A formal notation for grammars, where productions are written `<symbol> ::= <symbol1> "literal" ...`. Often extended with support for repetition (`*`, `+`), optionality (`?` or `[]`) and alternatives (`|`).

Operations are grouped on the left, giving a tree which is "heavy" on the left side; typically used for arithmetic operators.

Associativity rule

Associativity — Right

The final stage of a compiler or language processor, often tasked with low-level optimisation and code generation targeted at a particular machine architecture.

A parser that works by identifying the lowest-level details first, rather than working *top-down* from the start symbol. For example, an *LR parser*.

Composite data type

Backus-Naur form

A function (or other operation) packaged together with all the variables it can access from the surrounding scope in which it was defined.

A sequence of *production rule* applications that rewrites the *start symbol* into the input string (i.e., by replacing a *nonterminal symbol* by its expansion at each step). This can be seen as a trace of a parser's actions or as a proof that the string belongs to the language.

Context-free grammar

Derivation — Leftmost

A common ambiguity in programming languages (particularly those with C-like syntax) in which an optional `else` clause may be interpreted as belonging to more than one `if` sentence. Usually resolved in favour of the closest `if`, often by an *implicit disambiguation rule* (at least in non-*generalised parsing*).

A language (i.e., not just a library) with abstractions targeted at a specific problem domain.

Desugaring

Disambiguation rule

A derivation where the rightmost *nonterminal symbol* is selected at every rewrite step.

A formal grammar in which every *production rule* has a form of $A \rightarrow w$, where A is a single *nonterminal symbol* and w is a sequence of *terminals* and nonterminals.

Derivation

Dangling else problem

A derivation where the leftmost *nonterminal symbol* is selected at every rewrite step.

Removal of *syntactic sugar*. Sometimes used in a *frontend* to translate convenient language constructs used by the programmer into more fundamental constructs.

Domain-specific language

Derivation — Rightmost

Used to resolve ambiguities in a grammar, so that the parser yields a single unambiguous parse tree. Includes techniques such as *follow restrictions*, *precede restrictions*, *priority rules*, *associativity rules*, *keyword reservation* and *implicit rules*.

A DSL defined as a separate programming language.

Domain-specific language — Benefits

Domain-specific language — Internal or embedded DSL

Drawbacks of this concept include: Lots of implementation work, language fragmentation, learning/training issues, less tooling, troublesome interoperability, possibly worse error reports

When *names* are resolved by finding the closest binding in the runtime environment (i.e., the execution stack), rather than in the local lexical environment (i.e., the containing scopes at the use site).

Dynamic dispatch

Dynamic language

A typing style where the exact type of an object is not important, rather, any object is usable in any situation as long as it supports whatever methods are called on it. Used in many dynamic languages, such as Python, and in C++ templates. C.f. *structural typing*.

Benefits of this concept include: Easier programming, more efficient or secure, possibly better error reports

Domain-specific language — External DSL

Domain-specific language — Drawbacks

A DSL defined as language-like interface to library.

The process of selecting, at runtime, which implementation of a method to call at runtime; typically based on the the actual class of the object on which the method is called (as opposed to the static type of the variable).

Dynamic scoping

Duck typing

A language where most or all of the language semantics is processed at runtime, including aspects such as *name binding* and *typing*. May have features such as *duck typing*, *dynamic typing*, runtime reflection and introspection, and often allows code to be replaced and objects to be extended at runtime.

Benefits of this concept include: Compiler may run faster; easy to load code dynamically at runtime; allows some things that are type safe but are still excluded by a static type system; easy to use *duck typing* to get naturally generic code with little overhead for the programmer; reflection, introspection and metaprogramming becomes easier.

Dynamic semantics

Dynamic typing — Drawbacks

When type safety is enforced at runtime. Values are associated with type information, which can also be used for other purposes, such as runtime reflection. Used in languages such as Python, Ruby, Lisp, Perl, etc.

A data *member* of a data structure.

Epsilon

Evaluator

A mapping of names to values or types.

Gives the meaning of a program at execution time; either in terms of values being computed, actions being performed and so on.

Dynamic typing — Benefits

Dynamic typing

Drawbacks of this concept include: Type errors cannot be detected at compile time; rigorous testing is needed to avoid type errors; some optimisations may be difficult to perform (less of a problem with *just-in-time compilation*).

In a grammar, the empty string.

Field

Environment

A program that executes another program.

An abstraction over expressions (or more generally, over expressions, statements and algorithms).

Follow restriction

Function type

The first stage of a compiler or language processor, typically including a *parser* (possibly with a *tokeniser*), and a *typechecker* (semantic analyser). Sometimes also includes *desugaring*. Is typically responsible for giving the programmer feedback on errors, and translating to the internal *AST* or representation used by the rest of the system.

A formal set of rules defining the syntax of a language. Formally, a tuple $\langle N, T, P, S \rangle$ of *nonterminal symbols* N, *terminal symbols* T, *production rules* P, and a *start symbol* $S \in N$.

Functional programming

Generalised parser

The representation of a function in an evaluator or in a dynamic semantics specification. Usually includes the parameter names and the function body. Forms a *closure* together with an environment giving the function's declaration scope.

A disambiguation technique where a symbol is forbidden from or forced to be immediately followed by a certain terminal.

Function

Frontend

The representation of a function in the type system. Typically includes parameter types and return type, written $t_1, \ldots, t_k \rightarrow t$.

A programming paradigm based on mathematical functions, usually without state and mutable variables.

Grammar

Function value

A parser that can handle the full range of *context-free grammars*, including nondeterministic and ambiguous grammars. For example, a *GLL parser* or a *GLR parser*.

A *disambiguation rule* built into the parser, such as longest match for regular expressions, or resolving the *dangling else problem* by preferring shift over reduce in an *LR parser*.

Languages

Higher-order function

flash225™

flash225™

Inheritance

Abstraction,Languages

A programming paradigm based on statements that change program state; as opposed to *declarative programming*. May be combined with *object-oriented programming*.

A string of characters that is significant as a group; a word or *token*.

!!! Syntax

Kleene closure, star

flash225™ !!!

flash225™

Language

Languages

A technique in language processing where a call to a function or procedure is replaced by the code being called. Often used as part of code *optimisation*; removes *abstraction* introduced by the programmer.

A function which takes takes functions as arguments or returns *function values*.

Ambiguity,Parsing,Syntax

Implicit disambiguation rule

flash225™

flash225™

Imperative programming

Languages

A technique in *object-oriented programming* which combines automatic code reuse with subtyping.

A metasyntactic sugar for repetition: `x*` means that `x` can be repeated zero or more times. The language that the Kleene star generates, is a monoid with concatenation as the binary operation and epsilon as the identity element.

Syntax

Lexeme, Word

flash225™

flash225™

Inlining

Abstraction,Compilation,Transformation

A system of communication, with structure (*syntax*) and meaning (*semantics*), and *abstractions* that allow you to communicate usefully at different levels (i.e., more than just pointing at concrete things or showing a picture of something – where the meaning would be the thing itself).

Describes (often using a *Regular grammar*) the syntax of *tokens*; e.g., what constitutes an identifier, a number, different operators and the whitespace that separates them.

Lexical analysis

LL parser

When *names* are resolved (possibly statically) by finding the closest binding in the lexical environment (i.e., by looking at the scopes that lexically contains the name).

A grammar that can be parsed by a *LR parser*.

Logic programming

LR parser

A grammar that can be parsed by a *LL parser*.

Converting a sequence of characters (letters) to a sequence of *tokens* (*lexemes* or words).

!!! Syntax

Lexical syntax

flash225™ !!!

iii flash225™

Lexical scoping

Languages,Semantics iii

A table-driven *top-down parser*, similar to a *recursive descent parser*. Has trouble dealing with *left recursion* in production rules, so the grammar must typically be *left factored* prior to use.

A *declarative programming* paradigm based on formal logic, inference and reasoning. Useful for many purposes, including formal specification of language semantics.

Syntax

LR grammar

flash225™

flash225™

LL grammar

Syntax

A *bottom-up parser* that can handle deterministic context-free languages in linear time. Common variantes are LALR parsers and SLR parsers.

A part of language processing where names are associated with their declarations, according to *scoping* and *namespace* rules.

Member

Name binding — Static

A function which is a *member* of a class. Typically receives a self-reference to an object as an implicit argument.

A symbol in a grammar which is defined by a production. Can be replaced by terminal symbols by applying the production rules of the grammar. In a *context-free grammar*, the left-hand side of a production rule consists of a single nonterminal symbol.

Named tuple

Namespace

Names are bound at runtime; also applies to *dynamic dispatch* (where it is sometimes called *late* or *virtual binding*), where certain properties (such as types) may be known statically, but the exact operation called is determined at runtime.

An element of a structure or class; a *field*, *method* or inner class/type.

Name binding

Method

When done statically (or *early*), name binding is often combined with *typechecking*.

A tuple where the elements are named, like in a *structure*. Often exhibits *structural type equivalence*, even in languages that normally use *nominative type equivalence*

Nonterminal symbol

Name binding — Dynamic

Some kind name grouping that makes it possible to distinguish different uses of the same name. For example, having variable names be distinct from type names; or treating names in one module as distinct from the same names in another module.

A compilation step, usually combined with typechecking, where the name of an overloaded function is resolved based on the types of the actual arguments.

Optimisation

Parse forest

When the same name is used for multiple things (of the same kind). For example, several functions with the same name, distinguished based on the parameter types.

A technique for comparing (typically *algebraic*) data structures, where one or both structures may contain variables (sometimes refered to as meta-variables). Upon successful match, variables are bound to the corresponding substructure from the other side. Related to *unification* in Prolog, but often more restricted.

Parser

Parsing

A tree that shows the structure of a string according to a grammar. The tree contains both the tokens of the original string, and a trace of the derivation steps of the parse, thus showing how the string is a valid parse according to the grammar. Typically, each leaf corresponds to a *token*, each interior node corresponds to a *production rule*, and the root node to a production rule of the *start symbol*.

The process of transforming program code to make it more efficient, in terms of time or space or both.

Compilation,Semantics

Overload resolution

flash225™

flash225™

Overloading

Semantics

The *parse trees* that are the result of parsing an ambiguous grammar using a *generalised parser*.

A program that recognises input according to some *grammar*, checking that it conforms to the syntax and builds a structured representation of the input.

Languages,Transformation

Pattern matching

flash225™

flash225™

Parse tree

Parsing,Syntax

Recovering the grammatical structure of a string.

A *disambiguation rule* declaring an operator's priority/precedence. E.g., in Rascal: `syntax Expr = Expr "*" Expr > Expr "+" Expr;`

Precede restriction

Procedural programming

A *recursive descent parser* which does not require backtracking. Instead, it looks ahead a finite number of tokens and decides which parsing function should be called next. The grammar must be LL(k) for this to work, where k is the maximum lookahead.

A *top-down parser* built from mutually recursive functions, where each function typically implements one *production rule* of the grammar.

Program slicing

Recogniser

A rule describing which symbols may replace other symbols in a grammar. In a *context-free grammar*, the left-hand side consists of a single *nonterminal symbol*, while the right-hand side may be any sequence of *terminals* and nonterminals.

A disambiguation technique where a symbol is forbidden from or forced to be immediately preceded by a certain terminal.

!!! Ambiguity,Parsing,Syntax

Priority rule

flash225™ !!!

flash225™

Predictive parser

Parsing

A programming paradigm based around procedure calls. Sometimes considered the same as *imperative programming* and typically based on *structured programming*.

A program transformation technique where all code that is irrelevant to a certain set of inputs or outputs is removed. Applied *forwards*, any code not directly or indirectly using a selection of inputs is discarded; applied *backwards*, all code that does not contribute to the computation of the selected outputs is discarded. Mainly used in debugging, but sometimes also as an optimisation technique. Originally formalised by Mark Weiser in the early 1980s.

Parsing

Recursive descent parser

flash225™

iii flash225™

Production rule

Syntax iii

A program that recognises input according to some *grammar*, giving an error if it does not conform to the grammar, but does not build a data structure.

A formal *grammar* where every production rule has the form $A \to aB$, or $A \to a$ or $A \to \epsilon$, where A and B are *nonterminal symbols* and a is a *terminal symbol*, and ϵ is the *empty string*. Alternatively, the first production form may be $A \to Ba$.

Referential transparency

Regular grammar — Limitations

A formalism for describing a *regular grammar*, using the normal alphabet mixed with special *metasyntactic* symbols, such as the *Kleene star*. Commonly used to specify the *lexical syntax* of a language, and also for searching and string matching in many different applications.

Drawbacks of this technique include: Cannot deal with arbitrary composition of languages.

Scannerful parsing

Scannerful parsing — Benefits

A *disambiguation rule* which states that a grammar symbol cannot match some constraint. For example, identifiers could be defined as any word matching `[a-zA-Z]+` *except* `if`, `while`, …

When an expression can be replaced by its value without changing the meaning of the program; i.e., it will evaluate to the same value every time and not cause side effects. Usually a property of *functional programming* languages.

Regular grammar

Regular expression

Can't express arbitrary nesting, such as nested parentheses or block structure in a language.

Parsing is divided into two parts; a *tokeniser* that deals with the lexical syntax and a parser that deals with the sentence syntax.

Scannerful parsing — Drawbacks

Reserve rule

Benefits of this technique include: Faster than scannerless parsing, because the lexical syntax is specified with a regular grammar which can be parsed very efficiently.

Drawbacks of this technique include: Slower than scannerful parsing. Can lead to hard to find lexical ambiguities.

Scannerless parsing

Scope

Benefits of this technique include: Can parse combinations of languages that have different lexical syntax. Lexical syntax can be context-free, not just regular.

An artificial *language* used in software development.

Scope — Named

Semantic analysis

With nested scopes, variables in inner scopes may *shadow* those in outer scopes, and variables are removed as control flows out of the scope. Variable shadowing may be forbidden in some languages.

When scanning and parsing is unified into one process that deals with with the input characters directly.

Scannerless parsing — Drawbacks

Scannerless parsing — Benefits

A collection of identifier bindings – i.e., what is captured by the *environment* at some point in the code or in time.

With named scopes, we can refer to names in scopes that are not ancestors of the current scope. For example, with C++ classes and namespaces and Java packages and (static) classes.

Software Language

Scope — Nested

A phase of language processing that enforces the *static semantics* of a language. Includes *typechecking*, *name binding*, *overload resolution* and checking other static constraints.

When *type safety* is enforced at compile type (though some tests, such as for *typecasting*, may be done at runtime).

Start symbol

Static typing — Benefits

The part of language semantics which is processed at compile time (statically). Often includes constraints that might be part of the syntax, but which is done separately in order to keep the grammar *context-free*. Includes concepts like *name binding* and *typechecking*, and is used to eliminate a large class of invalid or erroneous programs.

See Desugaring.

Strong typing

Structure, Structure type

Drawbacks of this concept include: Type system may become either overly complicated or overly restrictive; doesn't help with non-type errors; makes dynamic loading of code somewhat more complicated; type declarations may be cumbersome if the language lacks *type inference*.

The *nonterminal symbol* in a grammar that generates all valid strings in a language.

Static typing

Static semantics

Benefits of this concept include: Detects a large an important class of errors (type errors) at compile time; enables advanced optimisations and efficient memory use.

When a language (to some degree) enforces *type safety*.

Syntactic sugar

Static typing — Drawbacks

A *composite data type* with named *fields members*; such as `struct` in C or `record` in Pascal. Similar to (or same as, with *structural type equivalence*) a *named tuple*.

A program that performs *lexical analysis*, grouping and classifying input into *tokens*.

Terminal symbol

Top-down parser

A *lexeme* or group of characters that forms a basic unit of parsing, categorised according to type, e.g., identifier, number, addition operator, etc. Forms the alphabet of the parser in *scannerful parsing*.

Whether a language protects against type errors, such as when a value of one data type is interpreted as another type (e.g., an `int` as a `float` or as a pointer to a string).

Typechecker

Type inference

Visiting the nodes/parts of a data structure such as a tree

An elementary symbol in the language defined by a grammar, which cannot be changed/matched by the production rules in the grammar (i.e., the symbol doesn't occur (alone) on the left-hand side of a production). Corresponds to a *token* or an element of the alphabet of a language.

Tokeniser

Token

A parser using a strategy where the top-level constructs are recognised first, starting with the start symbol. The parser starts at the root of the parse tree, and builds it top-down, according to the rules of the grammar. Includes *LL parsing* and *Recursive descent parsing*.

A program that detects type errors, ensuring *type safety* in a *statically typed* language. Often combined with other static semantic checks and processing, such as *overload resolution*, *name binding*, and checking access restrictions on names. C.f. *semantic analysis*.

Type safety

Traversal

Automatic deduction of the types in a language. Used with *static typing* to avoid having to declare types for variables and functions. Particularly useful in *generic programming* and type *polymorphism* where type expressions can become quite complicated. Used, e.g., in Haskell and Standard ML.

Type system

When a language (to some degree) does not enforces *type safety*.

Defines how a language classifies expressions and values into types.

FLASH225^{TM}

Weak typing

TYPES