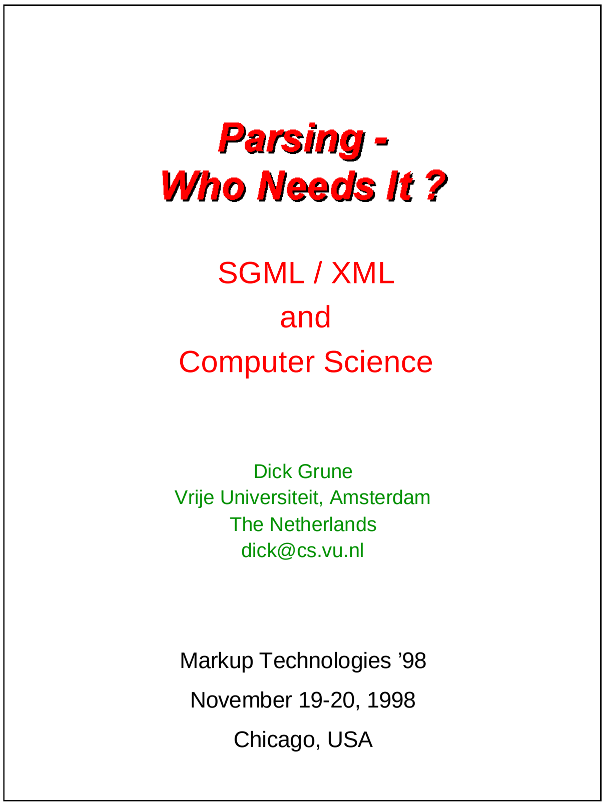


Parsing - Who Needs It ?

SGML / XML and Computer Science

Dick Grune
Vrije Universiteit, Amsterdam
The Netherlands
dick@cs.vu.nl

Markup Technologies '98
November 19-20, 1998
Chicago, USA

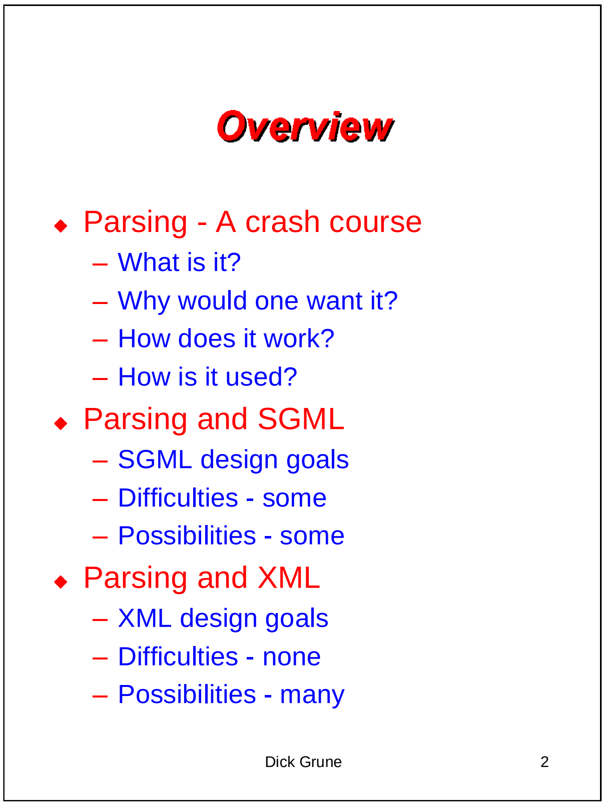
Introduction

SGML was clearly not designed to interface seamlessly with existing computer science parsing techniques. Most of these techniques were already very well-known in the beginning of the 80s, when SGML was designed. The syntax issues of SGML in ISO 8879 are expressed in completely novel terminology and the parsing requirements do not match known algorithms and techniques.

The results were not favourable: few computer scientists have worked on parsing SGML, although many software designers and programmers have. Most computer scientists found parsing SGML a less than attractive challenge, and some were daunted by its alien terminology.

The situation has deteriorated with the advent of HTML, which had a syntax defined more or less by what Netscape or MicroSoft could get away with. Unsurprisingly, this did not lead to a strong basis and stable software.

The developers of XML got the point, and did it right, both from the parsing point of view, the terminology and the SGML compatibility.



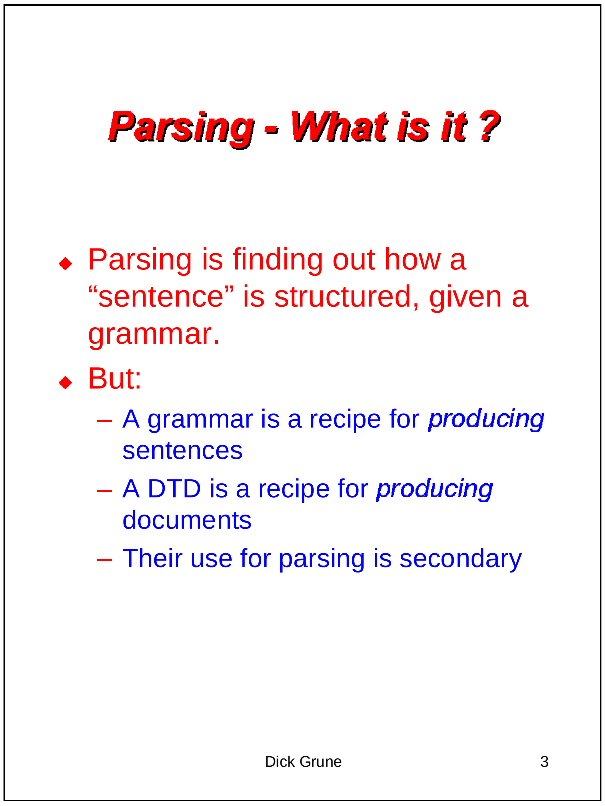
Overview

- Parsing - A crash course
  - What is it?
  - Why would one want it?
  - How does it work?
  - How is it used?
- Parsing and SGML
  - SGML design goals
  - Difficulties - some
  - Possibilities - some
- Parsing and XML
  - XML design goals
  - Difficulties - none
  - Possibilities - many

Dick Grune 2

Parsing - Overview

- Parsing
  - What it is
  - Why one would want it
  - How it works
  - What it yields
  - How it is used
- Parsing and SGML
  - SGML design goals in view of parsing techniques
  - Results of the SGML design
  - Automatic recovery from syntax / structure errors
  - Creating special SGML-oriented parsing techniques
- Parsing and XML
  - XML design in view of parsing techniques
  - Result of the XML design



Parsing - What is it ?

- Parsing is finding out how a “sentence” is structured, given a grammar.
- But:
  - A grammar is a recipe for *producing* sentences
  - A DTD is a recipe for *producing* documents
  - Their use for parsing is secondary

Dick Grune 3

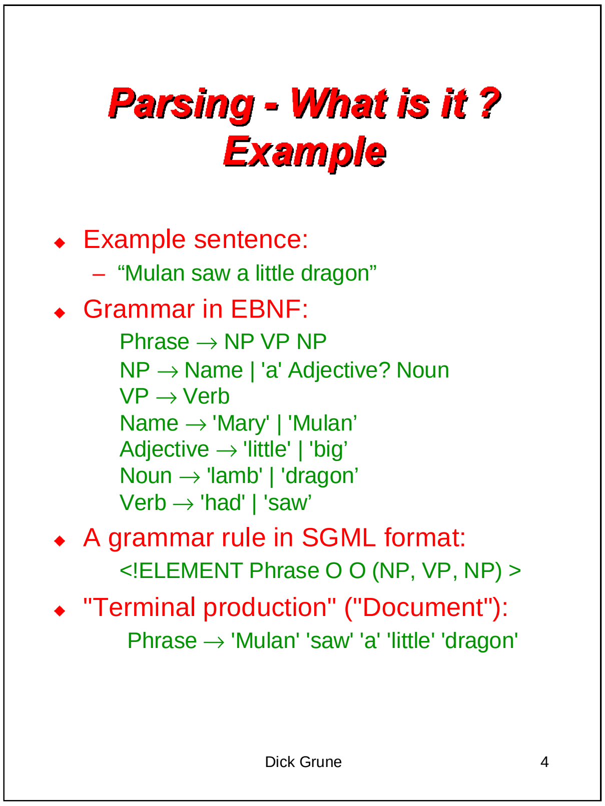
Since people use their brains for generating programs and documents, and use grammar-derived software to parse them, many people think grammars were invented for the express purpose of parsing. This is not true: a grammar has at least four aspects.

1. It is a finite description of an infinite set of sentences, programs, documents or what have you; such a set is a "language". This is the *Platonic* aspect.

2. The Theory of Formal Languages defines a process for constructing sentences from a grammar; this process records the structure, but when the sentence is later committed to a linear medium (speech, written text, file) this structure is lost. This is the *theoretical* aspect.

3. Parsing techniques use the grammar to reconstruct the structure of a sentence. Also, syntax-directed editors and pretty-print programs can be derived more or less automatically from a grammar. This is the *technological* aspect.

4. Meaning ("semantics") is attached both to the words in a sentence and to the structure. To understand a sentence or document we need not only its words but also its structure. In computer science the semantics is attached to nodes in the parse tree (through descriptions in a manual or through actions in a compiler), in SGML to elements (through style sheets). This is the *semantic* aspect.



Parsing - What is it ? Example

- Example sentence:
  - “Mulan saw a little dragon”
- Grammar in EBNF:

  Phrase → NP VP NP
  NP → Name | 'a' Adjective? Noun
  VP → Verb
  Name → 'Mary' | 'Mulan’
  Adjective → 'little' | 'big’
  Noun → 'lamb' | 'dragon’
  Verb → 'had' | 'saw’
- A grammar rule in SGML format:

  <!ELEMENT Phrase O O (NP, VP, NP) >
- "Terminal production" ("Document"):

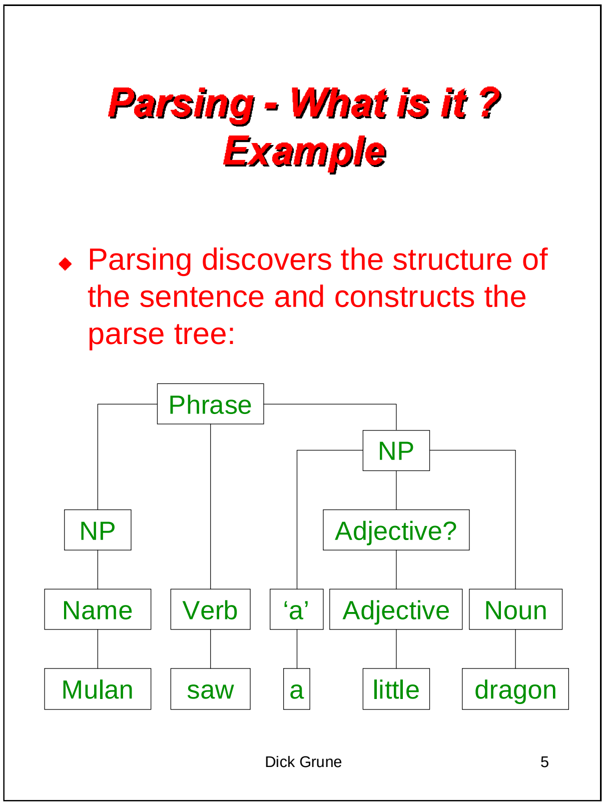
  Phrase → 'Mulan' 'saw' 'a' 'little' 'dragon'

Dick Grune 4

EBNF stands for Extended Backus (Naur|Normal) Form.

NP stands for Noun Phrase, VP for Verb Phrase.

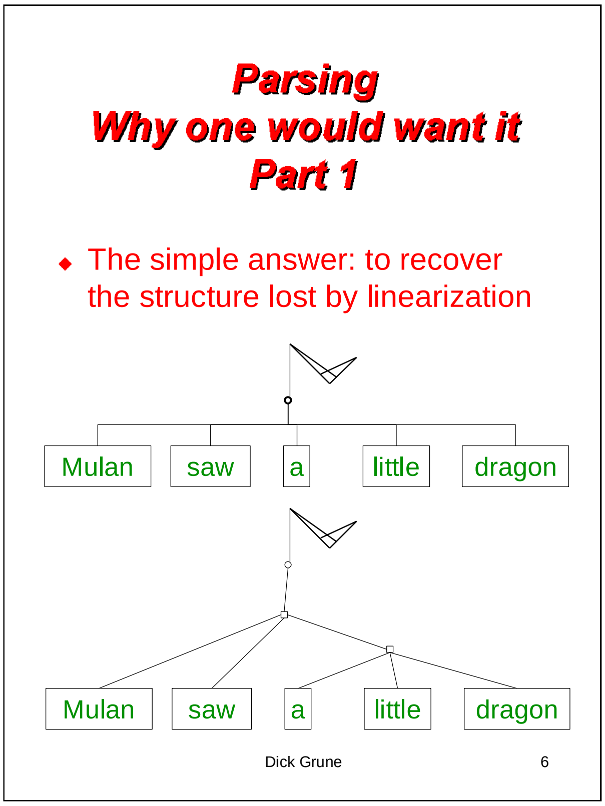
This grammar is of course very limited, but still allows us to make some points.



Parsing - What is it ? Example

- Parsing discovers the structure of the sentence and constructs the parse tree:

A sentence is a sequence of words; a parse tree is a sequence of words with structure.



Parsing Why one would want it Part 1

- The simple answer: to recover the structure lost by linearization

Mulan saw a little dragon

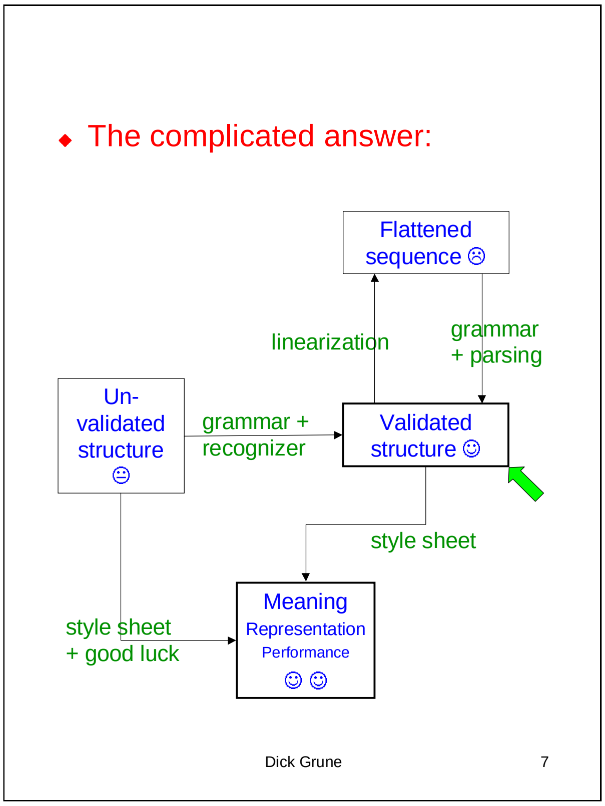
Mulan saw a little dragon

The production process (grammar to sentence) produces the sentence with a certain structure, but then the sentence is committed to a linear medium and the structure is lost, and with it the meaning of the sentence. Recovering the meaning proceeds in two steps:

1. Recover the structure, through parsing.
2. Retrieve the meaning from the structure.

In SGML we are also interested in just checking the structure — validation.

Parsing is like a crane that hoists up the structure lying hidden in the sentence / document.



- The complicated answer:

Flattened sequence ☹

linearization

grammar + parsing

Un-validated structure 😐

grammar + recognizer

Validated structure ☺

style sheet

style sheet + good luck

Meaning

Representation

Performance

☺ ☺

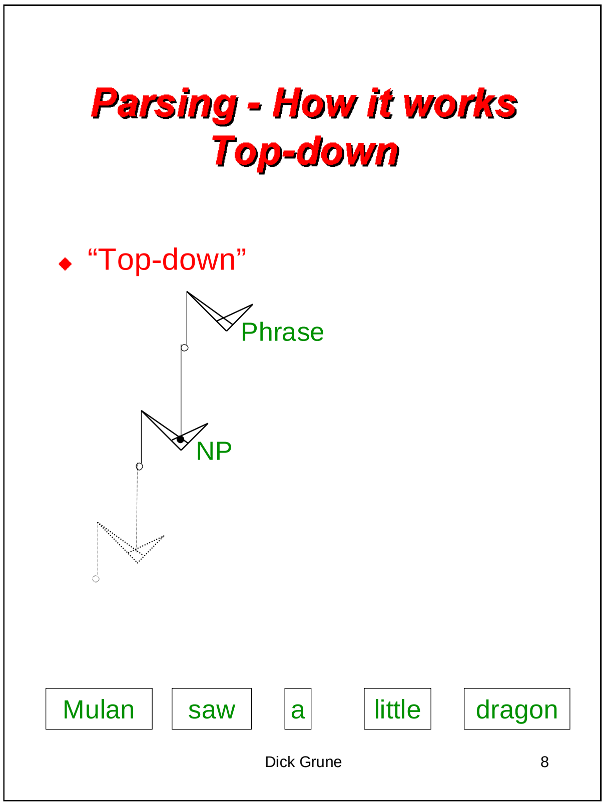
Dick Grune 7

What we want, and what the grammar-controlled production process gives us, is a validated structure -of a sentence or a document- based on a grammar. Combining this structure with a style sheet based on this grammar gives us its meaning, in the form of a representation on paper or on the screen, or of a performance (audio, animation, etc.). And that we want even more!

The structure may, however, get lost due to linearization in transport. If we have no other information, this structure is irretrievably lost, but if we know that the structure conforms to a given grammar, parsing can save the situation and reconstitute the structure.

It is also possible that we *have* the complete structure but we do not know if it conforms to the grammar on which the sytle sheet is based: we have an unvalidated structure. In that case we can just go ahead, apply the style sheet and hope for the best, usually with good results.

If that is not good enough for our peace of mind, we can use a limited form of parser, called a *recognizer*, and verify the structure, turning our unvalidated structure into a validated one.



Parsing - How it works
Top-down

- “Top-down”

Phrase

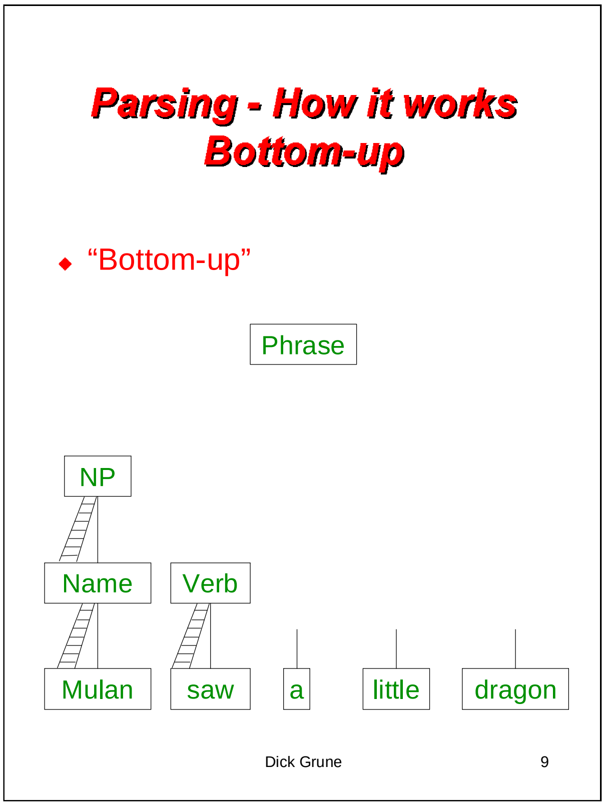
NP

Mulan saw a little dragon

Dick Grune 8

We know there is a parse tree somewhere, but we can’t see it yet. However, we know its top (start symbol / DOCTYPE) and we know its bottom (sentence / document).

One way to construct the parse tree is to start at the top and to try to construct a tree to the bottom. The top-down approach does this by having the top crane, which fishes for a Phrase (DOCTYPE), lower other cranes which fish for its elements, etc.



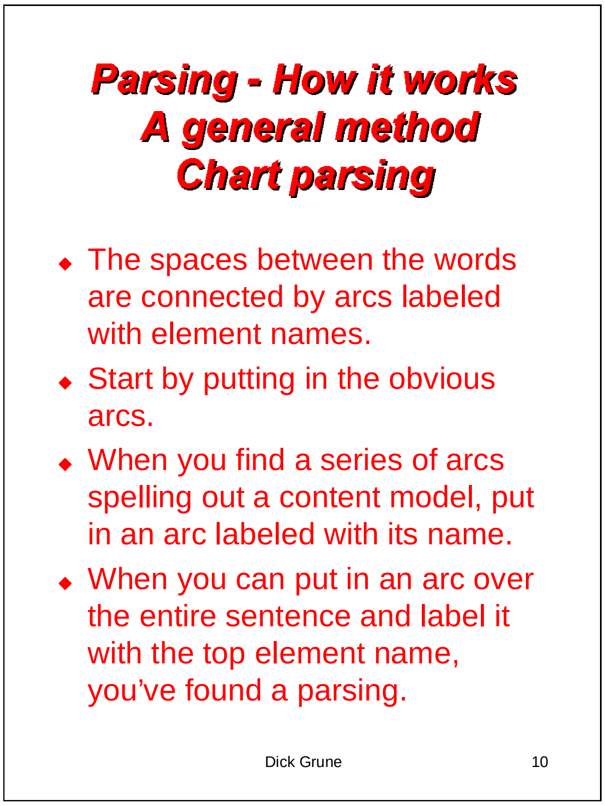
Parsing - How it works Bottom-up

- “Bottom-up”

Dick Grune 9

Or we can start at the bottom and try to construct a tree to the top. The bottom-up approach does this by looking at the words, guesses where they might derive from in the grammar, and erect parse tree fragments accordingly.

There are dozens of methods to do either. Some work for all grammars, some work for deterministic grammars only, some simpler ones work for even fewer grammars (but they can still be useful).



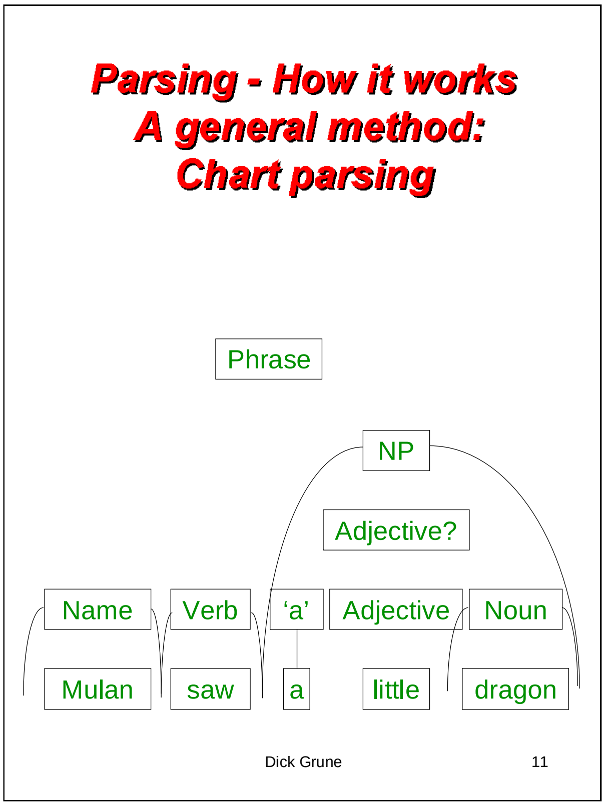
Parsing - How it works
A general method
Chart parsing

- The spaces between the words are connected by arcs labeled with element names.
- Start by putting in the obvious arcs.
- When you find a series of arcs spelling out a content model, put in an arc labeled with its name.
- When you can put in an arc over the entire sentence and label it with the top element name, you've found a parsing.

Dick Grune 10

This is a simple and intuitive bottom-up method that works for any grammar. Start at the bottom, and construct every possible subtree you can; if you reach the top this way, the sentence / document was valid and you have obtained the (a?) structure.

It is clear that this causes you to do much superfluous work. By avoiding duplicates you can reduce the amount of work drastically, by being clever you can reduce it even more and nevertheless get a reasonably good parsing process (algorithm).



Parsing - How it works
A general method:
Chart parsing

Phrase

NP

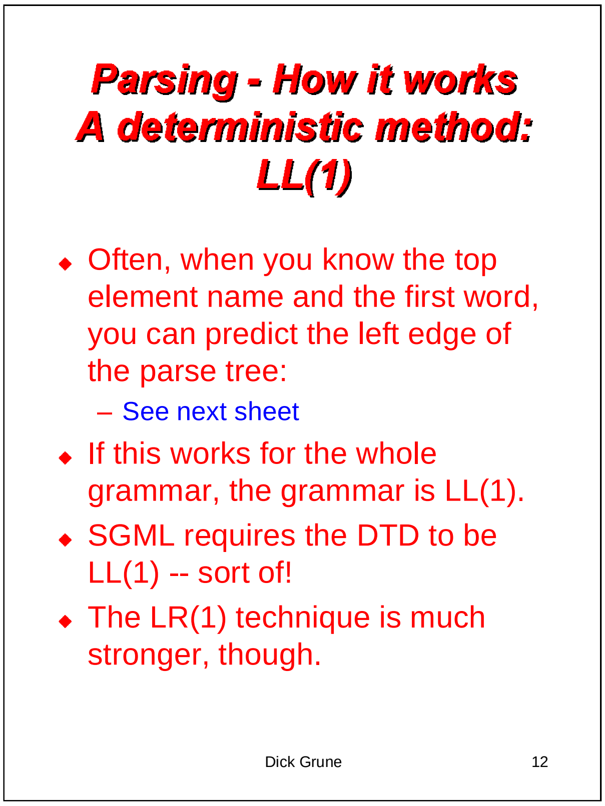
Adjective?

Name Verb ‘a’ Adjective Noun

Mulan saw a little dragon

Dick Grune 11

The arcs encompassing ‘little’, ‘Adjective’ and ‘Adjective?’ have been omitted, to avoid clutter.



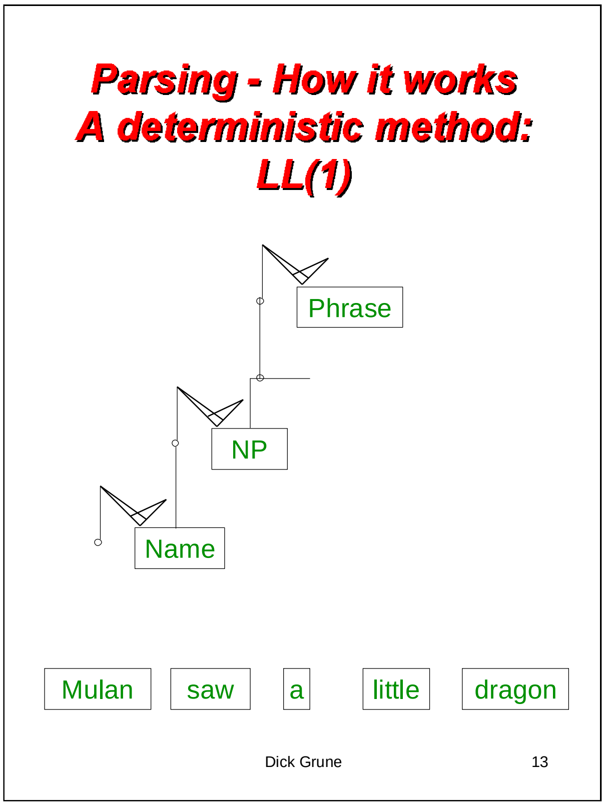
Parsing - How it works A deterministic method: LL(1)

- Often, when you know the top element name and the first word, you can predict the left edge of the parse tree:
  - See next sheet
- If this works for the whole grammar, the grammar is LL(1).
- SGML requires the DTD to be LL(1) -- sort of!
- The LR(1) technique is much stronger, though.

Given the top element NP and the first word ‘Mulan’, there is only one way to connect them: NP → Name → ’Mulan’. This means that we do not have to try the two possibilities for NP: Name and ’a’ Adjective? Noun, since we know it's going to be Name every time. If we can do this for the entire grammar, the grammar is LL(1).

We can then precompute all such relations between element names and words, and store the results in a table in the parser. This yields an LL(1) parser.

A somewhat similar but much less intuitive process starting at the bottom yields an LR(1) parser. It works by postponing decisions; it keeps considering all possibilities until we have seen enough words to decide which one applies. The only restriction is that it must be able to do this with a limited number of possibilities, independent of the length of the sentence. The LR(1) technique can handle a much larger set of grammars than the LL(1) technique.



Parsing - How it works A deterministic method: LL(1)

Phrase

NP

Name

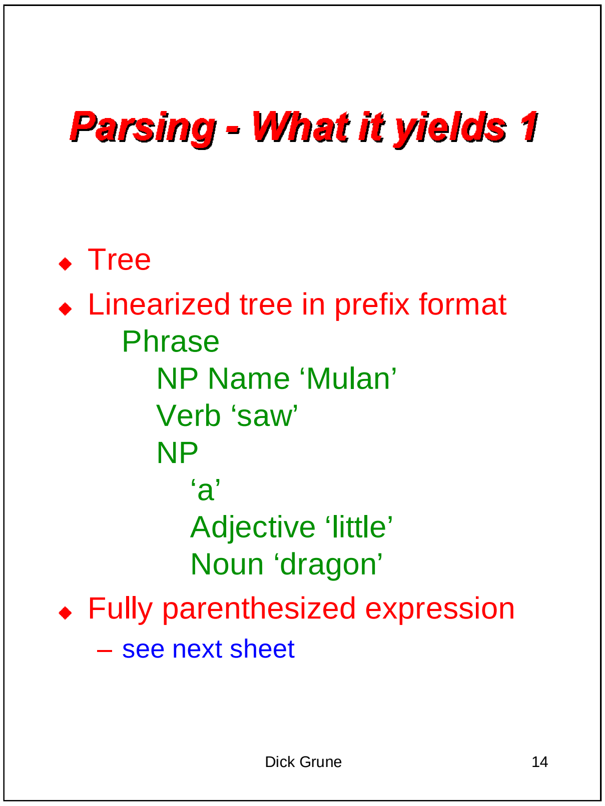
Mulan saw a little dragon

Dick Grune 13

Given a crane called Phrase and a word ‘Mulan’, analysis of the grammar shows that Phrase should lower a crane called NP and that it need not investigate other possibilities. This applies again to the crane NP, which, trying to reach the word ‘Mulan’ can do only one thing: lower a crane Noun. Etc.

If each combination of a crane and a word leads to an unambiguous decision, the grammar is “LL(1)”, and an LL(1) parser can be constructed for it.

LL(1) parsers are very efficient.

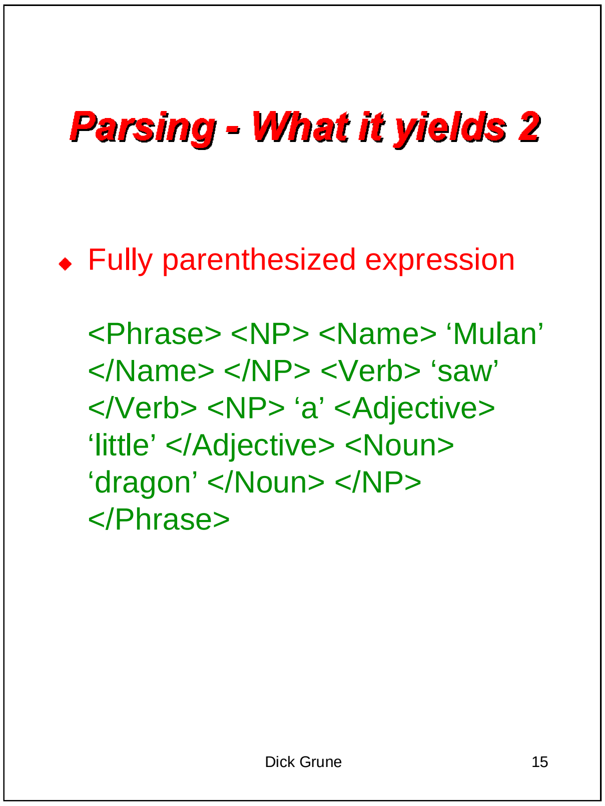


Parsing - What it yields 1

- Tree
- Linearized tree in prefix format
  - Phrase
    - NP Name ‘Mulan’
    - Verb ‘saw’
    - NP
      - ‘a’
      - Adjective ‘little’
      - Noun ‘dragon’
- Fully parenthesized expression
  - – see next sheet

Dick Grune 14

Actually, parsing *always* yields a parse tree (or forest, for ambiguous sentences). The three possibilities shown here are just alternative representations of that one parse tree; no magic involved.



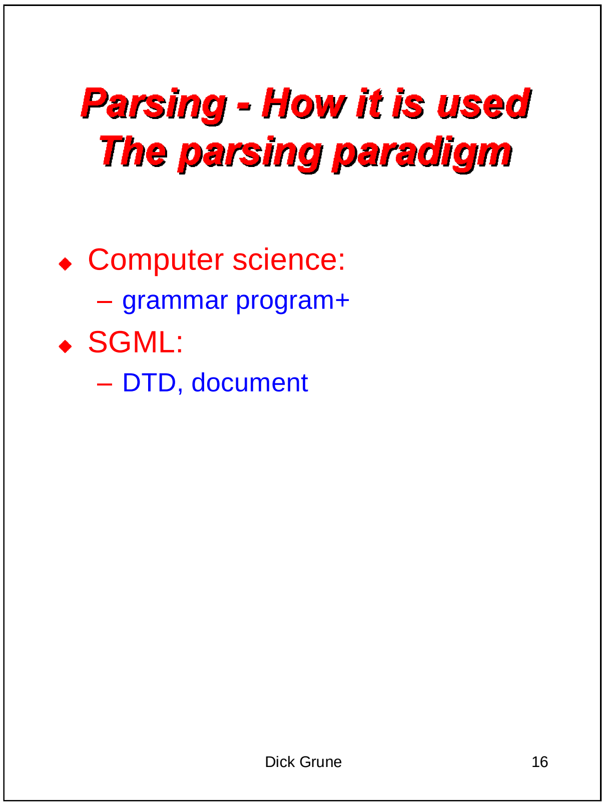
Parsing - What it yields 2

- Fully parenthesized expression

  <Phrase> <NP> <Name> ‘Mulan’ </Name> </NP> <Verb> ‘saw’ </Verb> <NP> ‘a’ <Adjective> ‘little’ </Adjective> <Noun> ‘dragon’ </Noun> </NP> </Phrase>

Dick Grune 15

Does this format remind you of something?



Parsing - How it is used
The parsing paradigm

- Computer science:
  - grammar program+
- SGML:
  - DTD, document

Dick Grune 16

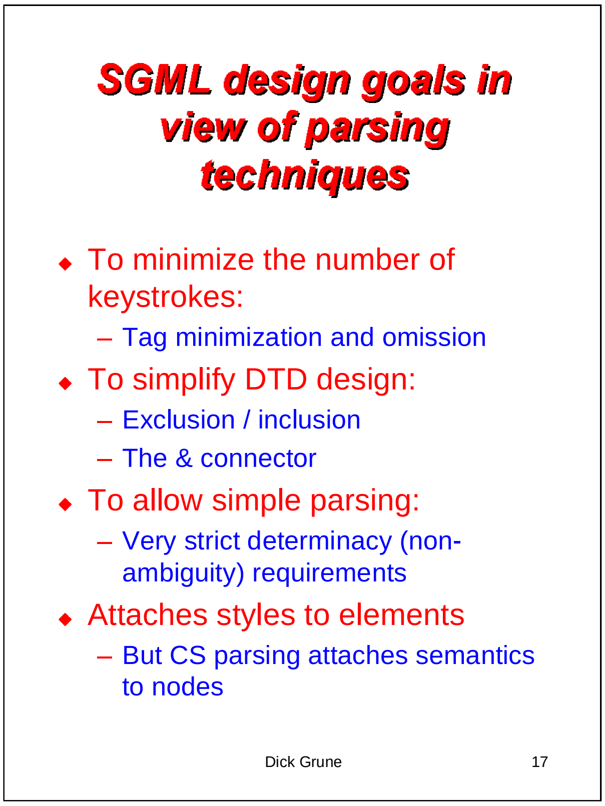
In computer science, grammars are used mainly to describe programming languages; the grammar is fixed once and for all (or at least for one or two years ...) in the manual, and is applied to zillions of programs.

In SGML, each document can, in principle, bring its own doctype; but in practice, most of them use a standard doctype. On the other hand, the syntax defined by the doctype can be modified using SGML features like OMITTAG.

In computer science, the grammar is processed partly automatically and partly by hand to produce a compiler or interpreter. In the SGML world the doctype must be read and used in parsing. Warmer and van Egmond use a two-stage process, in which the doctype is converted into an LL(1) grammar, which is turned into a parser by using an LL(1) parser generator. This parser is then used to parse the document proper.

- A J. Warmer and S. van Egmond, The implementation of the Amsterdam SGML parser, Electronic Publishing, 2, 2, July 1989, 65-90.

Other systems interpret the doctype while parsing the document, rather than generate a parser, for example James Clark’s sgmls. Therefore, sgmls is not a parser, it is a DTD interpreter.



SGML design goals in view of parsing techniques

- To minimize the number of keystrokes:
  - Tag minimization and omission
- To simplify DTD design:
  - Exclusion / inclusion
  - The & connector
- To allow simple parsing:
  - Very strict determinacy (non-ambiguity) requirements
- Attaches styles to elements
  - But CS parsing attaches semantics to nodes

Dick Grune 17

The designers of SGML were concerned with issues in user-friendliness of SGML:

1. the ease of typing (data entry);

2. the ease of DTD design;

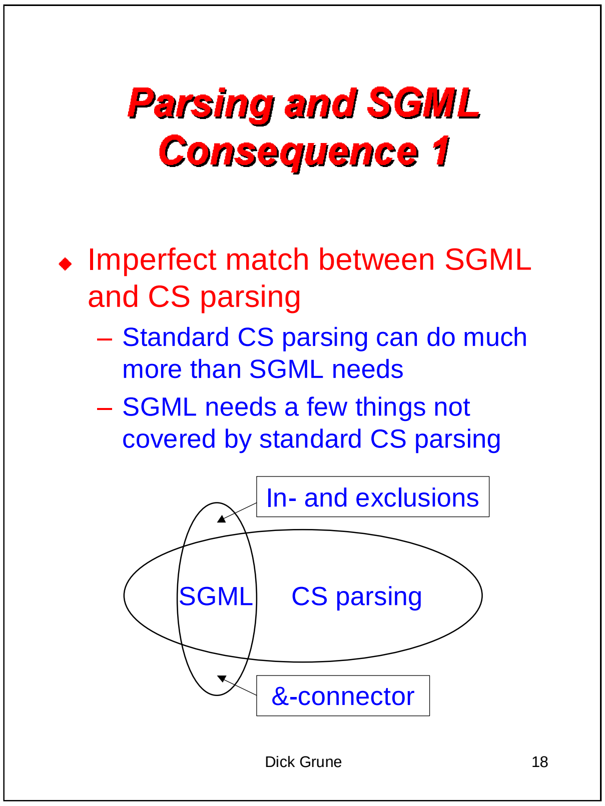
3. the understandability of the notion of ambiguity in DTDs (actually non-determinism in CS terms; all ambiguous grammars are non-deterministic, but not all non-deterministic grammars are ambiguous).

To help typing SGML they introduced tag minimization and omission, which in itself would not be a problem, were it not that it is subject to resetting by OMITTAG.

To ease constructing a DTD, inclusion and exclusion and the &-connector were added to what looks more or less like a traditional CS grammar in SGML notation. There is no direct parsing theory to implement these; one has to improvise -- or do research.

To help the user understand what kind of constructions are allowed in content models, the designers of SGML defined an intuitive notion of *ambiguity,* and required the DTD not to be ambiguous. The defined notion corresponds almost but not completely to the LL(1) property in parsing theory, and no theory for it was available. Matzen, George and Hedrick have tried to reconcile them, but address only the finite-state aspects.

- R.W. Matzen, K.M. George and G.E. Hedrick, A formal language model for parsing SGML, J. Systems Software, 36, 2, 147-166, 1997.



Parsing and SGML Consequence 1

- Imperfect match between SGML and CS parsing
  - Standard CS parsing can do much more than SGML needs
  - SGML needs a few things not covered by standard CS parsing

In- and exclusions

SGML

CS parsing

&-connector

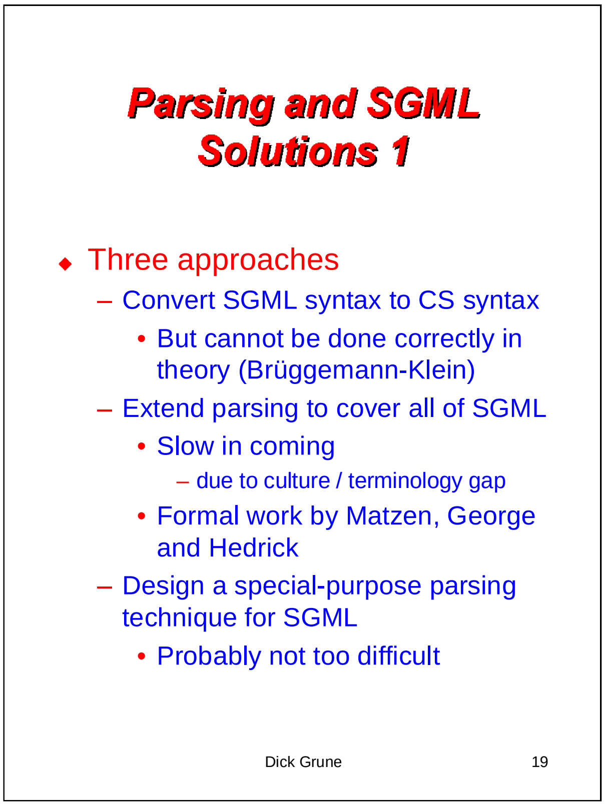
Given the wealth of CS parsing theory and the dearth of SGML parsing theory, it is tempting to convert the SGML doctype to a CS grammar and use standard CS theory from there. This conversion is not trivial.

First there is inclusion and exclusion. There is some theory about the joining and intersecting of languages, most of it saying that you cannot do it, for theoretical reasons. Probably none of these reasons apply exactly to SGML, but determining that requires research.

Then there is the &-connector. One would suppose that one can just replace (F&G) by (FG|GF), but Brüggemann-Klein & Wood have shown that this is not true.

- Anne Brüggemann-Klein and Derick Wood, The validation of SGML content models. Mathematical and Computer Modelling, 25, 4, 1997, 73-84.

Consider the content model (A? & B), which is unambiguous in the SGML sense of the word, because when we find just a B, it matches the one B in the content model. But we cannot replace it by the content model (A? B | B A?), since this is ambiguous in SGML terms: when we find just one B, it can match either B in the new content model.



Parsing and SGML Solutions 1

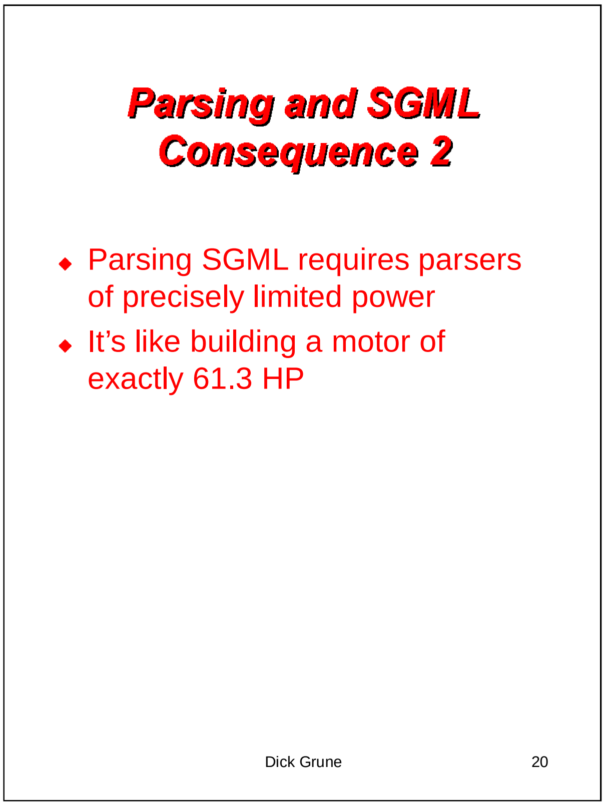
- Three approaches
  - Convert SGML syntax to CS syntax
    - But cannot be done correctly in theory (Brüggemann-Klein)
  - Extend parsing to cover all of SGML
    - Slow in coming
      - due to culture / terminology gap
    - Formal work by Matzen, George and Hedrick
  - Design a special-purpose parsing technique for SGML
    - Probably not too difficult

Dick Grune 19

The alternative to grammar conversion is to develop the proper theory of SGML grammar features. This is not at all impossible, or even extraordinarily difficult, but results have been slow in coming. I would expect the causes to lie in the culture gap and the differences in terminology.

- R.W. Matzen, K.M. George and G.E. Hedrick, A formal language model for parsing SGML, J. Systems Software, 36, 2, 147-166, 1997.

is an important step in this direction.

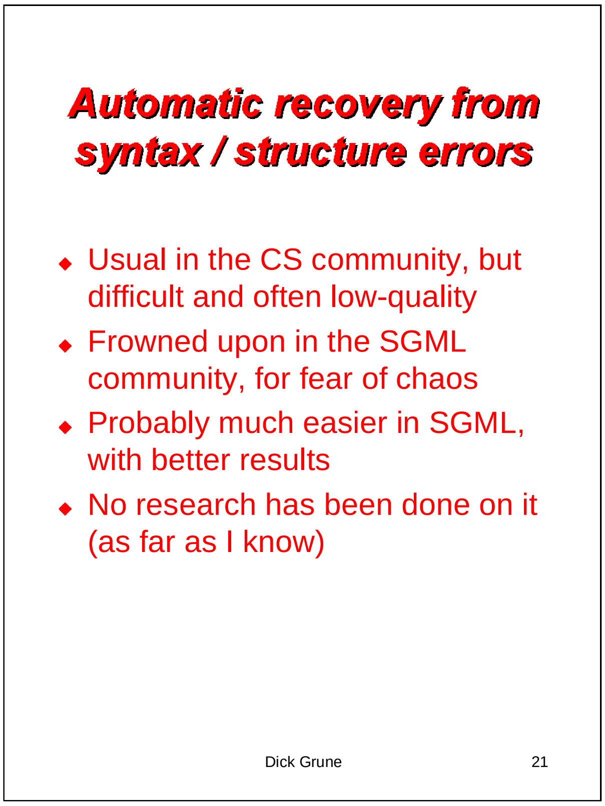


Parsing and SGML Consequence 2

- Parsing SGML requires parsers of precisely limited power
- It’s like building a motor of exactly 61.3 HP

Dick Grune 20

In many cases standard techniques will do more than is allowed by SGML, and special, precise, and often non-trivial tests must be designed to make the parser recognize the exact boundaries of correct SGML. Programmers do not like to have a good algorithm curtailed, and tend to ignore limitations regardless of specs.



Automatic recovery from syntax / structure errors

- Usual in the CS community, but difficult and often low-quality
- Frowned upon in the SGML community, for fear of chaos
- Probably much easier in SGML, with better results
- No research has been done on it (as far as I know)

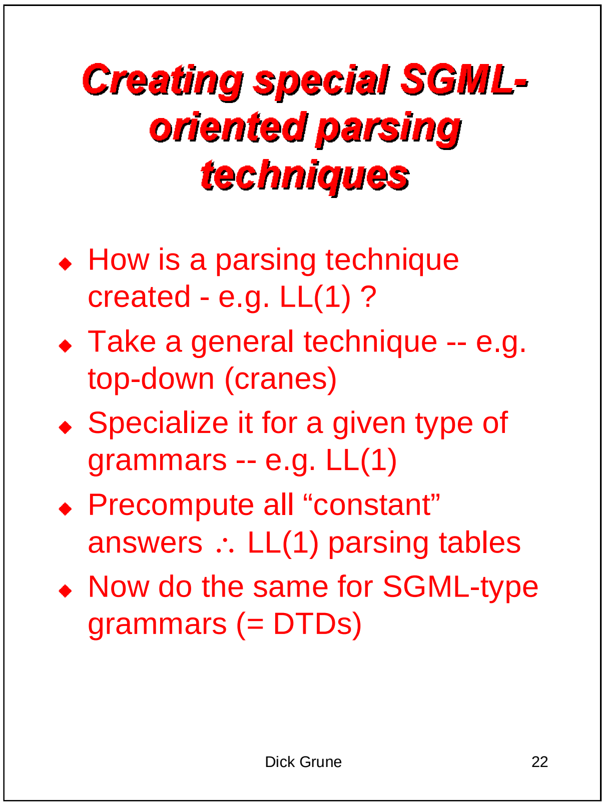
Dick Grune 21

Syntax error recovery is not synonymous with continuing after an error. Processing of a document proceeds at two levels: one that reconstructs the parse tree, and one that attaches semantics to it. It is quite possible and even useful to continue parsing without continuing to work on the semantics.

Automatic error recovery allows errors following the first one to be detected and reported. This is useful both for hand-written SGML and for generated SGML. Syntax errors will not be uncommon in hand-written SGML, and it is useful to supply the author with as much feedback as possible. Syntax errors in generated SGML result from either a bug in the program or a misunderstanding of SGML; in both cases, liberal error information is essential.

Another essential application of automatic error recovery may be in the automatic correction of legacy data, legacy either because the data is being brought under a DTD for the first time, using ad-hoc heuristic conversion, or because the DTD changed.

Since many more elements are visibly marked in SGML text than they are in program text, it is likely that automatic error recovery techniques for SGML can be better than those for general programming languages. No theory is available, though, as far as I know.



Creating special SGML-oriented parsing techniques

- How is a parsing technique created - e.g. LL(1) ?
- Take a general technique -- e.g. top-down (cranes)
- Specialize it for a given type of grammars -- e.g. LL(1)
- Precompute all “constant” answers ∴ LL(1) parsing tables
- Now do the same for SGML-type grammars (= DTDs)

It is a wide-spread misconception that parsing techniques can only be invented by extremely clever people living in the late 60s and early 70s, using inscrutable mental processes. All parsing techniques are specializations of the two general, intuitively simple parsing processes we have seen before, top-down and bottom-up.

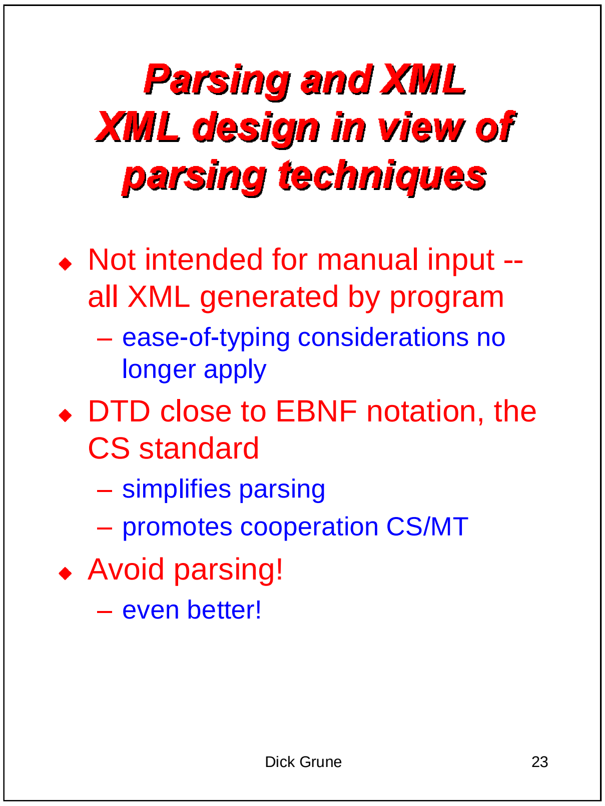
As an example, consider LL(1). It is a top-down technique, so we start with the image of a crane lowering other cranes again lowering other cranes, etc., to finally pick up the first word.

In the general algorithm, each crane successively tries to lower all cranes that match its content model, until one of them catches the first word. A similar process is then repeated for the second word, etc. You may have to undo some actions and try some other cranes to reach the last word, but in the end you get a parse tree, by trial and error, so to speak. This is known as top-down backtracking recursive descent parsing.

Now we observe that very often, for a given crane and a given word, we end up with the same chain of cranes to connect them. If this were the case for all combinations of cranes and words, we would never have to try other cranes once we have found one that works, and the parser would be a lot simpler and faster. Grammars --restricted grammars-- that have this property are said to be LL(1). They yield recursive descent LL(1) parsers.

If the answer to the question which crane to lower from a given crane to catch a given word is the same every time, we can precompute this answer and store it in a table in the parser. We do this once, during parser generation time, and during parsing time we don’t have to bother any more; the answer is there, pre-fab. This yields predictive LL(1) parsers.

There is no reason to think that the same process could not be used to create parsing techniques tailored to SGML.



Parsing and XML XML design in view of parsing techniques

- Not intended for manual input -- all XML generated by program
  - ease-of-typing considerations no longer apply
- DTD close to EBNF notation, the CS standard
  - simplifies parsing
  - promotes cooperation CS/MT
- Avoid parsing!
  - even better!

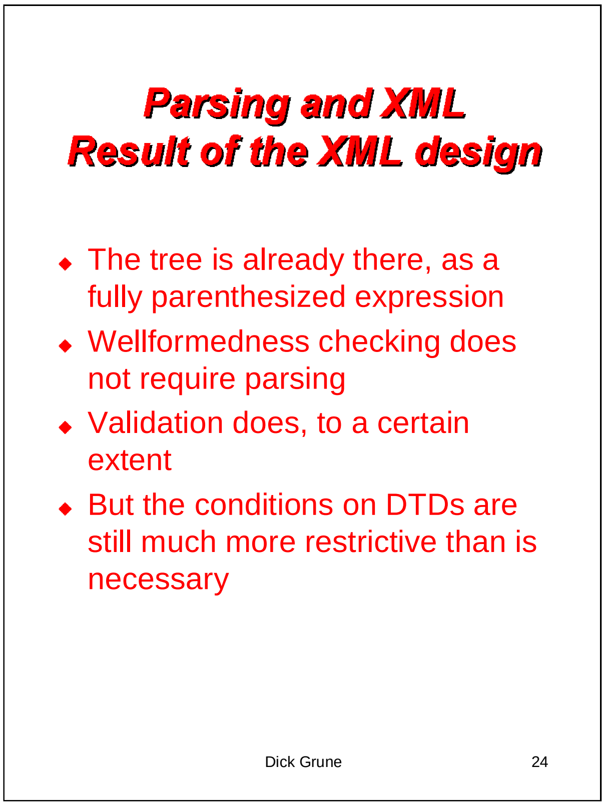
Dick Grune 23

The designers of XML have cleared the way for progress, by recognizing that software has improved since the 80s and that we have learned something from our efforts to parse SGML text.

1. Writing SGML/HTML/XML text by hand is obsolete. All such text is generated by software, under human control or automatically. So ease-of-typing considerations no longer apply, and tag minimization and other short-cuts can be abolished.

2. Inclusion, exclusion and the &-connector are not worth the trouble they cause.

3. Contact with the computer science and parsing community will be easier if the grammar of XML and the conditions on it are expressed in terms that relate to concepts from the theory of formal languages.



Parsing and XML
Result of the XML design

- The tree is already there, as a fully parenthesized expression
- Wellformedness checking does not require parsing
- Validation does, to a certain extent
- But the conditions on DTDs are still much more restrictive than is necessary

Dick Grune 24

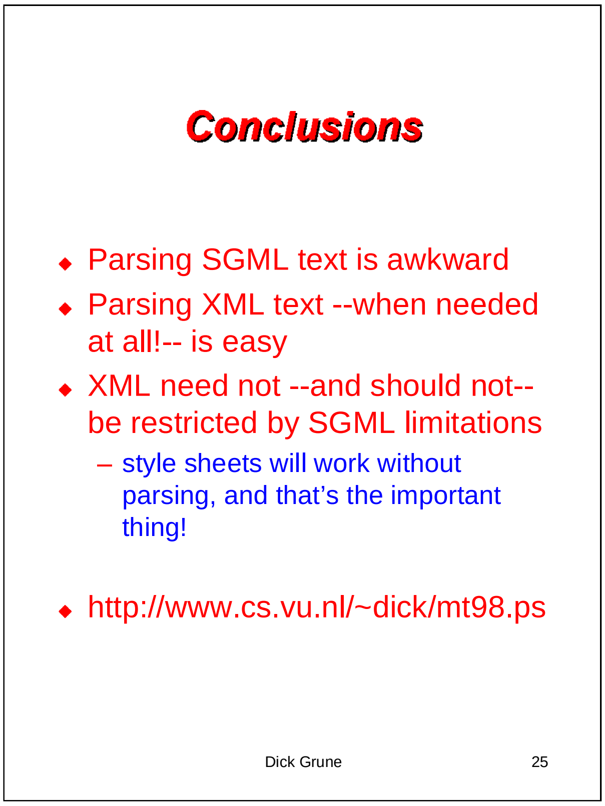
Abolishing tag minimization and other short-cuts results in fully parenthesized XML texts, which no longer require parsing; the text *is* the parse tree. Also, software that produces full parenthesization is very easy to write and to maintain.

Since there is no guarantee that the parse tree presented by the text conforms to the grammar, validation is still required, but it is very simple. Also, if we are willing to take the parse tree on face value, semantics can be attached to it using style sheets, even without knowing the grammar. This extends the range of applicability of XML enormously.

With text consisting exclusively of parenthesized expressions, almost all limitations on the DTDs can be lifted. It is quite possible to have elements

```
 <!ELEMENT PictureGallery (OnePicture | TwoPictures | PictureList )>
<!ELEMENT OnePicture (Picture)>
<!ELEMENT TwoPictures (Picture Picture)>
<!ELEMENT PictureList (Picture Picture Picture+)>
```

and still be unambiguous, without loss of speed. But it would no longer be SGML ...



Conclusions

- Parsing SGML text is awkward
- Parsing XML text --when needed at all!-- is easy
- XML need not --and should not-- be restricted by SGML limitations
  - style sheets will work without parsing, and that’s the important thing!
- http://www.cs.vu.nl/~dick/mt98.ps