Languages and Compilers
(SProg og Oversættere)

Language Design

Sequence control and Subprogram Control

Language Design and control structures

– Language Design Criteria
– Evaluation of expressions
– Explicit sequence control vs. structured sequence control
– Loop constructs
– Subprogram
– Parameter mechanisms

Table 1.1 Language evaluation criteria and the characteristics that affect them

	CRITERIA		
Characteristic	READABILITY	WRITABILITY	RELIABILITY
Simplicity	•	•	•
Orthogonality	•	•	•
Data types	•	•	•
Syntax design	•	•	•
Support for abstraction		•	•
Expressivity		•	•
Type checking			•
Exception handling			•
Restricted aliasing			•

Sequence control

- Implicit and explicit sequence control
  - Expressions
    - Precedence rules
    - Associativity
  - Statements
    - Sequence
    - Conditionals
    - Loop constructs
    - unstructured vs. structured sequence control
  - Subprograms
    - Parameter mechanisms

Expression Evaluation

- Determined by
  - operator evaluation order
  - operand evaluation order
- Operators:
  - Most operators are either infix or prefix (some languages have postfix)
  - Order of evaluation determined by operator precedence and associativity

Example

- What is the result for:

$$3 + 4 * 5 + 6$$

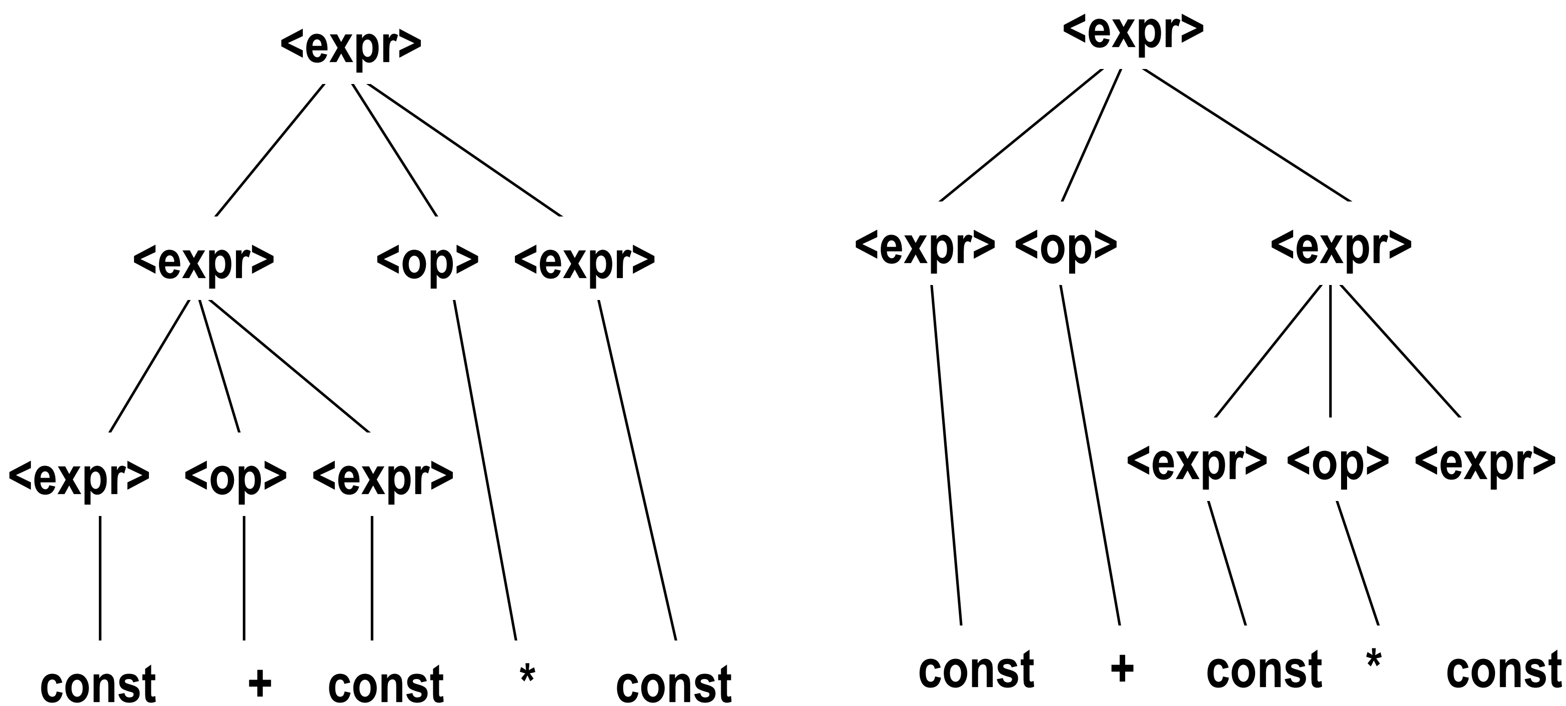
- Possible answers:
  - $41 = ((3 + 4) * 5) + 6$
  - $47 = 3 + (4 * (5 + 6))$
  - $29 = (3 + (4 * 5)) + 6 = 3 + ((4 * 5) + 6)$
  - $77 = (3 + 4) * (5 + 6)$
- In most language, $3 + 4 * 5 + 6 = 29$
- … but it depends on the precedence of operators

An Ambiguous Expression Grammar

How to parse 3+4*5?

<expr> → <expr> <op> <expr> | const

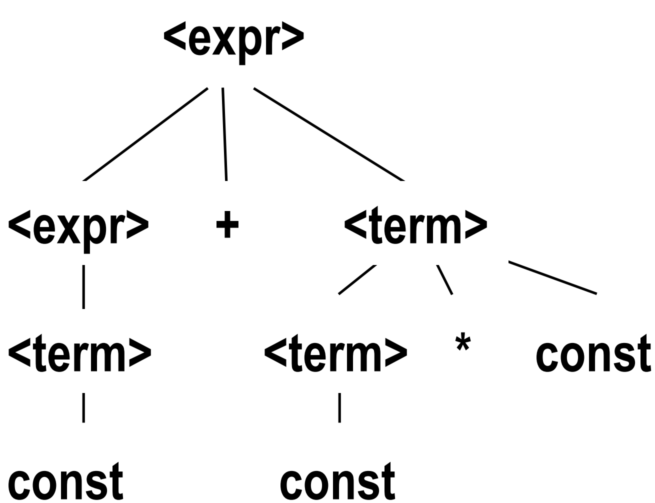
<op> → + | *



Expressing Precedence in grammar

- We can use the parse tree to indicate precedence levels of the operators

<expr> → <expr> + <term> | <term>
<term> → <term> * `const` | `const`



In LALR parsers we can specify Precedence which translates into Solving shift-reduce conflicts

Note in LL(1) parsers we have to use Left recursion elimination

Expr → Term Expr1 .
Expr1 →+ Term Expr1
| .
Term → const Term1 .
Term1 →* const Term1
| .

Operand Evaluation Order

- Example:

```
A := 5;
f(x) = {A := x+x; return x};
B := A + f(A);
```

- What is the value of B?
- 10 or 15?

Solution to Operand Evaluation Order

- Disallow all side-effects in expressions but allow in statements
  - Problem: not applicable in languages with nesting of expressions and statements
- Fix order of evaluation
  - SML does this – left to right
  - Problem: makes some compiler optimizations hard or impossible
- Leave it to the programmer to be sure the order doesn't matter
  - Problem: error prone

  - Fortress: Parallel evaluation unless specified to be sequential

Control of Statement Execution

- Sequential
- Conditional Selection
- Looping Construct
- Must have all three to provide full power of a Computing Machine

For-loops

- Controlled by loop variable of scalar type with bounds and increment size
- Scope of loop variable?
  - Extent beyond loop?
  - Within loop?
- When are loop parameters calculated?
  - Once at start
  - At beginning of each pass

Logic-Test Iterators

- While-loops
  - Test performed before entry to loop
- `repeat`...`until` and `do`...`while`
  - Test performed at end of loop
  - Loop always executed at least once
- Design Issues:
  1. Pretest or posttest?
  2. Should this be a special case of the counting loop statement (or a separate statement)?

Gotos

- Requires notion of program point
- Transfers execution to given program point
- Basic construct in machine language
- Implements loops

Advance in Computer Science

- Standard constructs that structure jumps

  if … then … else … end

  while … do … end

  for … { … }

  case …
- Modern style
  - Group code in logical blocks
  - Avoid explicit jumps except for function return
  - Cannot jump *into* middle of block or function body
- But there may be situations when “jumping” is the right thing to do!

Exceptions: Structured Exit

- Terminate part of computation
  - Jump out of construct
  - Pass data as part of jump
  - Return to most recent site set up to handle exception
  - Unnecessary activation records may be deallocated
    - May need to free heap space, other resources
- Two main language constructs
  - Declaration to establish exception *handler*
  - Statement or expression to *raise* or *throw* exception

Often used for unusual or exceptional condition, but not necessarily.

Subprograms

1. A subprogram has a single entry point

2. The caller is suspended during execution of the called subprogram

3. Control always returns to the caller when the called subprogram's execution terminates

Functions or Procedures?

- Procedures provide user-defined statements
  - Abstractions over statements
- Functions provide user-defined operators
  - Abstractions over expressions
- Methods used for both functions and procedures

Subprogram Parameters

- Formal parameters: names (and types) of arguments to the subprogram used in defining the subprogram body
- Actual parameters: arguments supplied for formal parameters when subprogram is called
- *Actual/Formal Parameter Correspondence:*
  - attributes of variables are used to exchange information
    - Name – Call-by-name
    - Memory Location – Call-by reference
    - Value
      - Call-by-value (one way from actual to formal parameter)
      - Call-by-value-result (two ways between actual and formal parameter)
      - Call-by-result (one way from formal to actual parameter)

Tennent's Language Design principles

- The Principle of Abstraction
  - All major syntactic categories should have abstractions defined over them. For example, functions are abstractions over expressions
- The Principle of Correspondence
  - Declarations ≈ Parameters
- The Principle of Data Type Completeness
  - All data types should be first class without arbitrary restriction on their use

–Originally defined by R.D.Tennent

Example of missing correspondence

In Pascal:

```
procedure inc(var i : integer);
 begin
  i := i + 1
 end;

var x : integer;
begin
 x := 1;
 inc(x);
 writeln(x);
end
```

No corresponding declaration

However C has correspondence

```
void inc(int *i) {
 *i = *i + 1;
}

int x = 1;
inc(&x);
printf("%d", x);

int x = 1;
{
 int *i = &x;
 *i = *i + 1;
}
printf("%d", x);
```