How to design a simple interpreter?

Introduction to PCCTS

Some examples (I)

- Evaluation of expressions
  - Source program #1 37 – 7 * 3 – 11
  - Output of execution: 79? 5? 27? -240?
  - Source program #2 37 – – – – 6
  - Output: wrong expression? 43?

Some examples (II)

- Script to compute your grade. Source program:

```
read  pl
read  fl
l2 = (pl + fl) / 2
lab = max( fl,  l2 )
read  teo
cl = 40*lab + 60*teo
write  cl/100
```

- Input data: 6.3 8.7 7.6
- Output of execution: 8.04

Some examples (III)

- A simple imperative language.

  Source program:

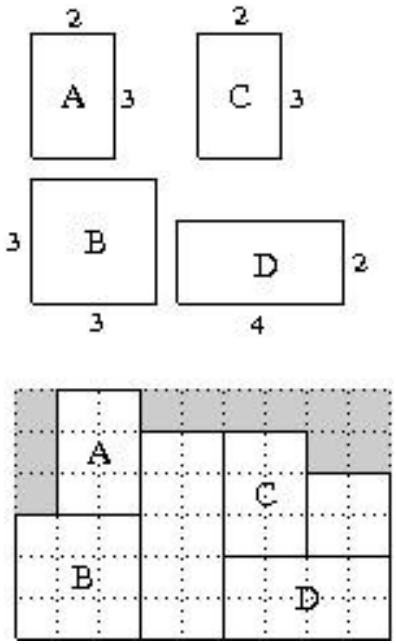
```
i  := 0
p := 3
while   p  > 1   do
      if    p < 7   then
p := p * p
      else    p := p – i
      endif
      i := i + 1
      print  p
endwhile
```

- Output of execution:

  9 8 6 36 32 27 21 14 6 36 26 15 3 9
  -5

Some examples (IV)

- Sliceable designs. Source program:




```
A = [2 3];
B = [3 3]; C = [2 3]; D = [4 2];
Block1 = A / B;
Block2 = C | D;
Block3 = Block1 | C / [3 4];
BoundingBox   A;
BoundingBox   Block1;
Utilization  A;
Utilization  Block1;
BoundingBox  A/B | [2 5] | (C | [2 2]) / D;
Utilization  A/B | [2 5] | (C | [2 2]) / D;
```

- Output of execution:

  [2 3] [3 6] 100% 83.33% [9 6] 79.63%

Structure of an interpreter

Lexical analysis (scanner)

- **Splits** the input program into lexical components
- Generates a list of **tokens**

Syntactic analysis (parser)

- Checks the **structure** of the token list
- Generates an abstract syntax tree (**AST**)

Semantic analysis

- Walks through the AST
- Checks some **other properties** (for example, type checking)

Interpreter

- Walks through the AST (or other representation) and **processes** it:
- Reads some input data and **generates results**

Structure of a simple interpreter

Lexical and syntactical analysis

- **Splits** the input into a list of tokens and **checks** their **structure**
- Generates an abstract syntax tree (**AST**)

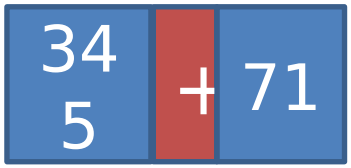
Semantic analysis and interpreter

- Walks through the AST and **checks** some other **properties**
- **Executes** the AST, i.e. reads some input data, process it and **generates results**

Our first recognizer of expressions

- Example of input data:

```
345 +
71
```



- List of tokens sent to the parser:
  - **NUM PLUS NUM EOF**
- Output of the interpreter:
  - the input is syntactically correct!

How to design a scanner

First, specify the *regular expressions.*

For example, for the naturals: ('0' | '1' | …| '9')+

- Write it by hand:

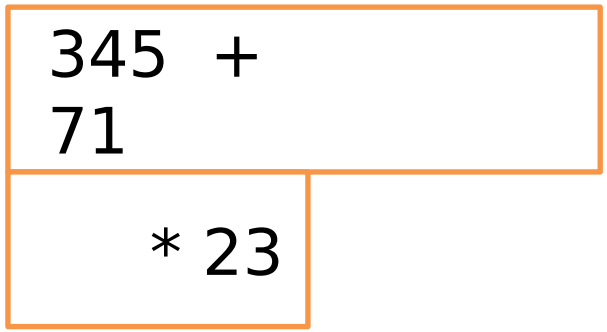
```
i = 0;   c = getchar();
do  {
     text[i] = c;    // text of
token
     i++;   c = getchar();
}  while  (c >= '0'  &&  c
<= '9')
text[i] = '\0';
token = NUM;   // type of
```

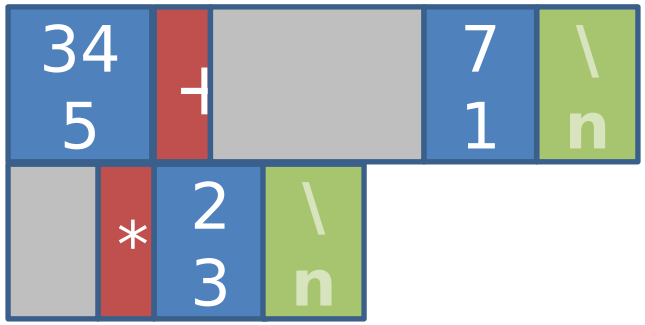
- Use a tool like PCCTS:

```
#token    NUM    "[0-9]+"
```

Our first recognizer of expressions

- Example of input data:





- List of tokens <u>sent to the parser</u>:
  - **NUM PLUS NUM STAR NUM EOF**
- Output of the interpreter:
  - the input is syntactically correct!

How to design a parser

First, specify the grammar. For example, only for sums of naturals: NUM (PLUS NUM)*

- Write it by hand, for example a recursive descendent parser. You can see a fragment:

```
if (token != NUM) throw
error;
else  token = nextToken();
while  (token == PLUS)  {
     token = nextToken();
     if  (token != NUM)  throw
error;
     else  token =
nextToken();
}
```

- Use a tool like PCCTS:

```
expr:  NUM (PLUS NUM)* ;
```

PCCTS generates the scanner and the parser

example0.g

- The user specifies **tokens** and **grammar**
- Use **antlr** to generate the parser in example0.c

parser.dlg

- Use **dlg** to generate the scanner scan.c

Different C-files

- Use **gcc/g++** to generate the executable

First example: example0.g

```
#header <<
    #include "charptr.h"
>>

<<    #include "charptr.c"
    int main() {
        ANTLR( expr(), stdin );
    }
>>

#lexclass START
#token   NUM    "[0-9]+"
#token   PLUS    "\+"
#token   SPACE   "[ \n]"   << zzskip(); >>

expr:   NUM   ( PLUS   NUM )*  ;
```

ANTLR generates: example0.c

```
...
int  main( )  {
    ANTLR( expr(),  stdin );
}

//  expr:  NUM  ( PLUS  NUM )* ;
void  expr( )  {
        ...
        MATCH( NUM );  nextToken();
        ...
        while  ( token == PLUS ) {
                MATCH( PLUS );  nextToken();
                MATCH( NUM );  nextToken();
        }
        ...
}
```

Build the Abstract Syntax Tree (AST)

- With PCCTS: simply *annotate* the grammar

  expr : NUM (PLUS^ NUM)* ;

  – Token information is copied into a new AST node

  – Each token PLUS turns into the current AST root

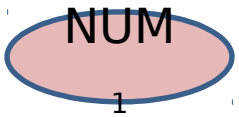
- For the input: NUM_1 $PLUS_2$ NUM_3 $PLUS_4$ NUM_5 EOF

Build the Abstract Syntax Tree (AST)

- With PCCTS: simply annotate the grammar

  expr : NUM (PLUS^ NUM)* ;

- For the input: NUM_1 $PLUS_2$ NUM_3 $PLUS_4$ NUM_5 EOF
- The following AST is built:

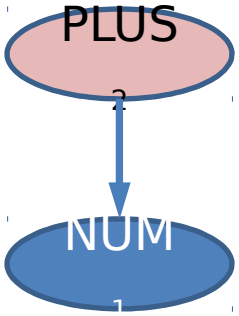


Build the Abstract Syntax Tree (AST)

- With PCCTS: simply annotate the grammar

  expr : NUM (PLUS^ NUM)* ;

- For the input: NUM_1 $PLUS_2$ NUM_3 $PLUS_4$ NUM_5 EOF
- The following AST is built:

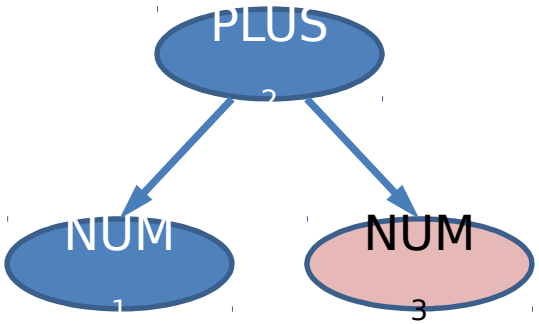


Build the Abstract Syntax Tree (AST)

- With PCCTS: simply annotate the grammar

  expr : NUM (PLUS^ NUM)* ;

- For the input: NUM_1 $PLUS_2$ NUM_3 $PLUS_4$ NUM_5 EOF
- The following AST is built:

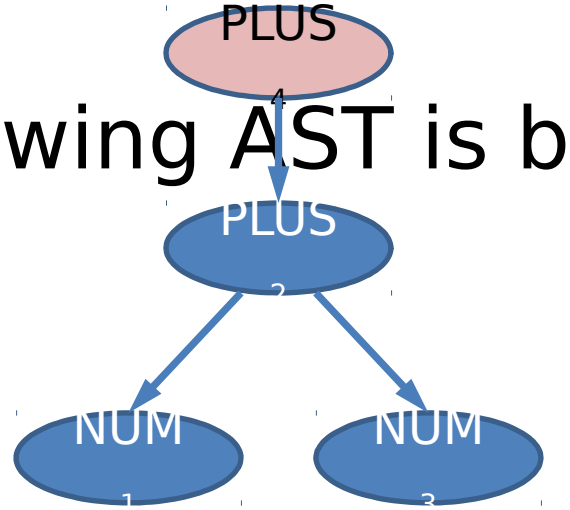


Build the Abstract Syntax Tree (AST)

- With PCCTS: simply annotate the grammar

  expr : NUM (PLUS^ NUM)* ;

- For the input: NUM_1 $PLUS_2$ NUM_3 $PLUS_4$ NUM_5 EOF
- The following AST is built:

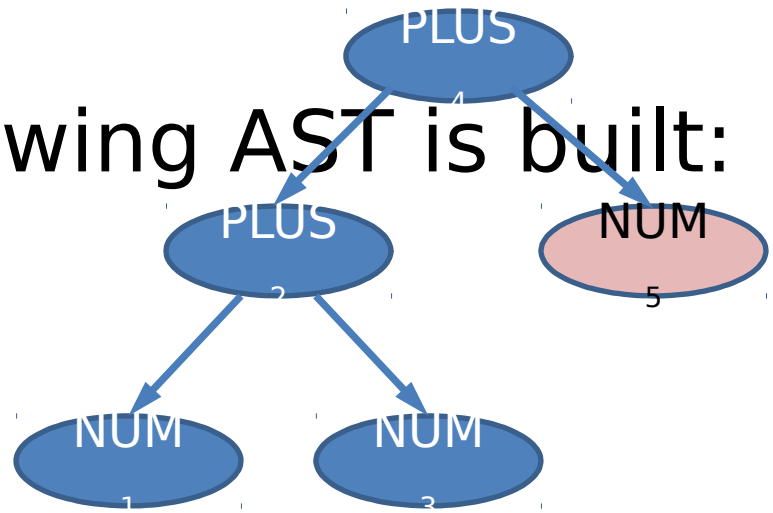


Build the Abstract Syntax Tree (AST)

- With PCCTS: simply annotate the grammar

  expr : NUM (PLUS^ NUM)* ;

- For the input: NUM_1 $PLUS_2$ NUM_3 $PLUS_4$ NUM_5 EOF
- The following AST is built:

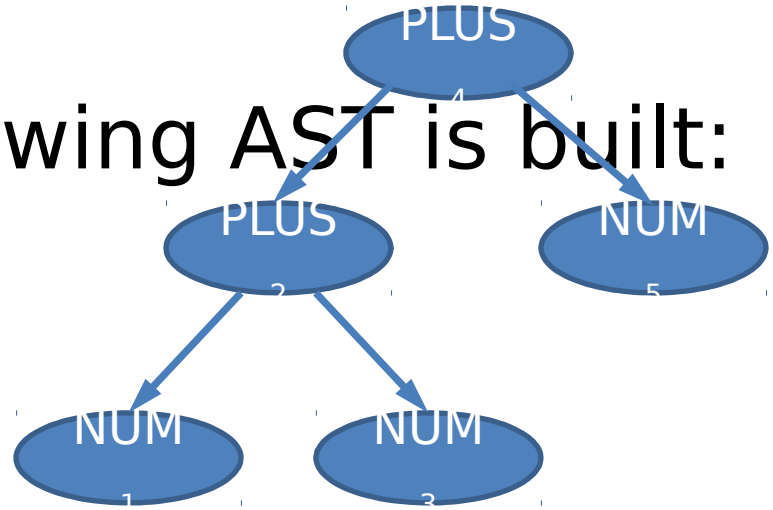


Build the Abstract Syntax Tree (AST)

- With PCCTS: simply annotate the grammar

  expr : NUM (PLUS^ NUM)* ;

- For the input: NUM_1 $PLUS_2$ NUM_3 $PLUS_4$ NUM_5 EOF
- The following AST is built:



Take information to the AST

- Which token attributes are needed for the interpretation?
  - Usually token type (*kind*) and token *text*
- How to build an AST node from a token?
  - Copying these attributes into the AST node

Token attributes

```
typedef  struct  {
    string  kind;
    string  text;      // only for NUMs
}  Attrib;

void  zzcr_attr( Attrib* attr,  int type, char*
text)  {
    attr->type = text;
    attr->text = “”;
    if  (type == NUM)  {
        attr->type= “intconst”;
        attr->text = text;    // for example
“345”
    }
}
```

AST Nodes

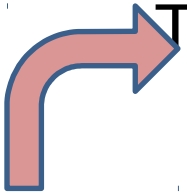
```
#define  AST_FIELDS       string kind; string
text;

#define   zzcr_ast(as, attr, ttype, textt)     \
                as = new AST;                          \
                (as)->kind = (attr)->kind;          \
                (as)->text = (attr)->text;          \
                (as)->right = NULL;  (as)->down
= NULL;
```

From the input to the AST nodes

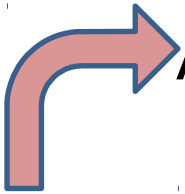
345 + 78
*
3

nextToken():
zzcr_attr()



Token type: **NUM**
Token Attrib:

kind: “intconst”
text: “345”



zzcr_ast()

AST node:

kind: “intconst”
text: “345”

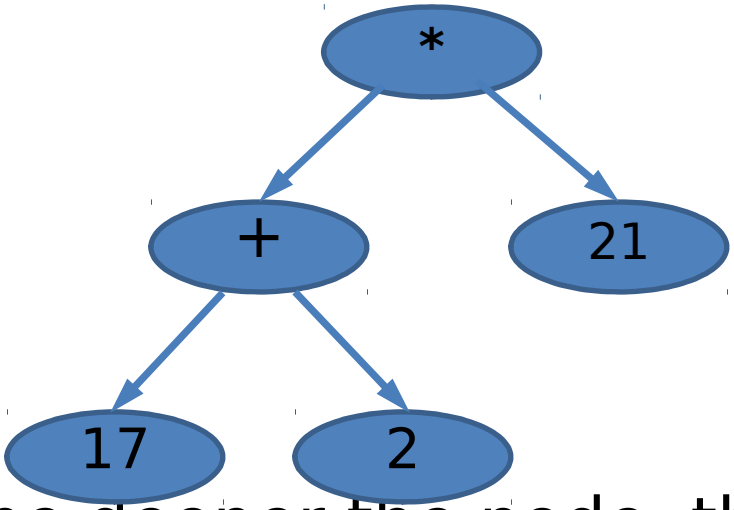
How to visit an AST

- Each AST node has pointers to the first child (*down*) and to the next sibling (*right*)
- A
A

```
void  ASTPrintIndent( AST *a, string s )  {    // s is the
indentation
    if  (a == NULL)  return;
    cout << s << “  “ << a->kind;                    // current
node
    if ( a->text != "“ )   cout << "(“ << a->text << ")";
    cout << endl;
    ASTPrintIndent( a->down, s + " |“ );      // first and
successive children
    ASTPrintIndent( a->right, s );                // next sibling
}
```

How to visit an AST

- The left-hand side AST, displayed with neither *down* nor *right* pointers, will be printed with indentation as shown on the right:




```
*
|   +
|   |   17
|   |   2
|   21
```

- The deeper the node, the greater the indentation