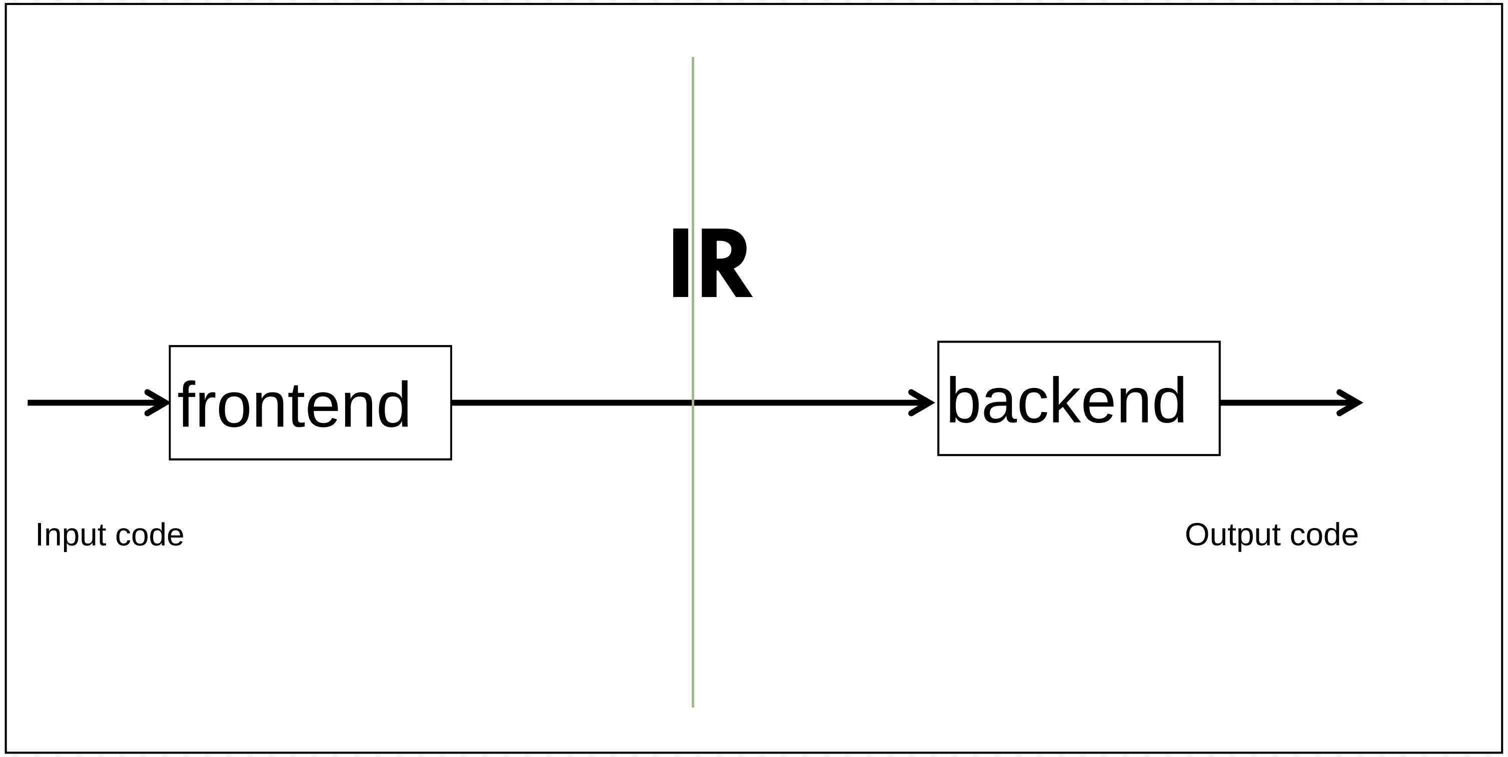
GENEROVÁNÍ KÓDU 2. VNITŘNÍ FORMY (MEZIKÓDY): ABSTRAKTNÍ SYNTAKTICKÝ STROM (AST), GENEROVÁNÍ, TABULKA SYMBOLŮ, TYPOVÁ KONTROLA

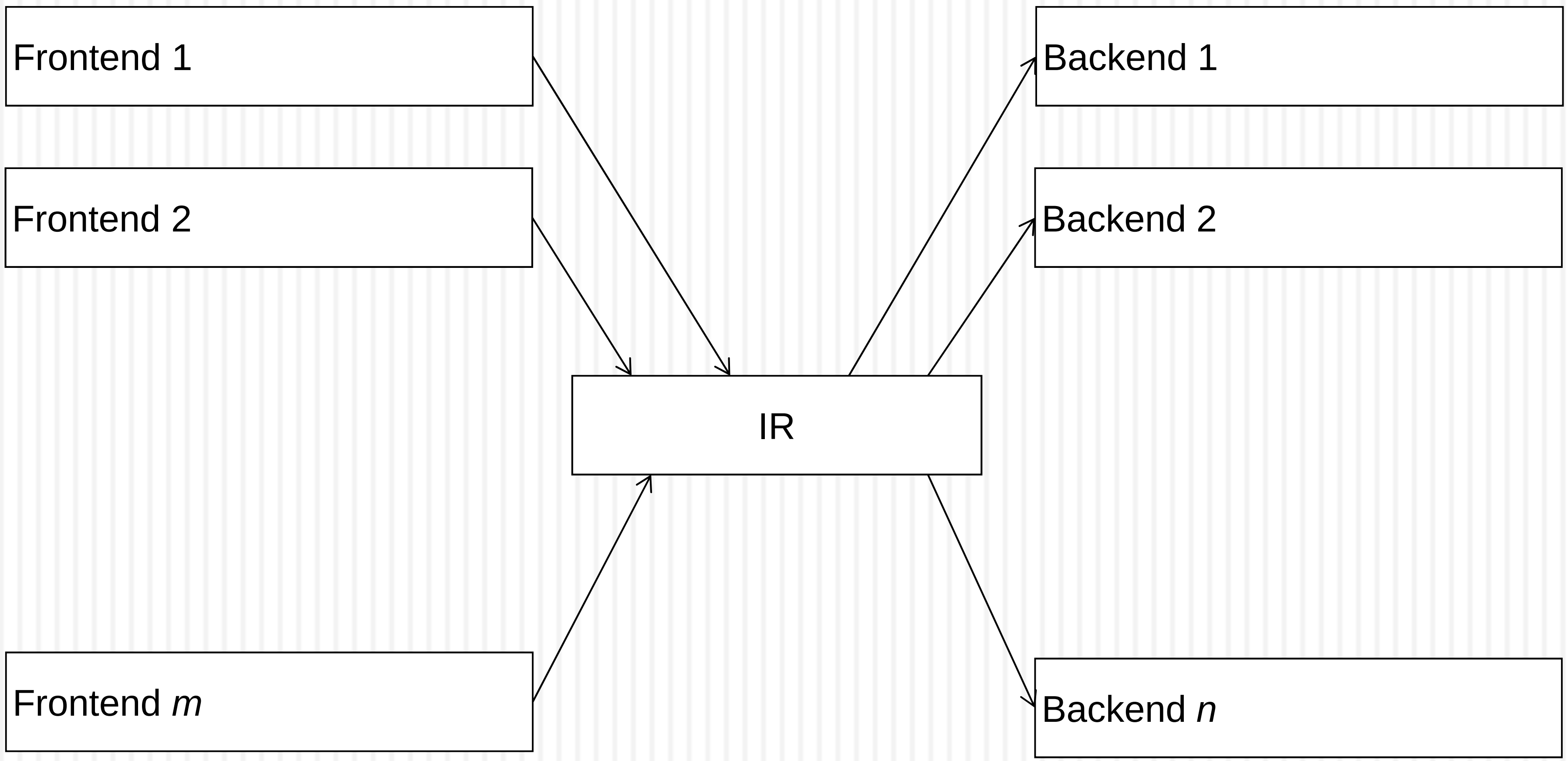
2011 Jan Janoušek
MI-GEN

Evropský sociální fond
Praha & EU:
Investujeme do vaší budoucnosti

USING INTERMEDIATE REPRESENTATIONS (IR), INTERMEDIATE CODES



Even compiler for *m* input and *n* output languages !



An example

- gcc GNU compiler collection:
  - Frontends for: C (gcc), C++ (g++), Java (gcj), Ada (GNAT), Objective-C (gobjc), Objective-C++ (gobjc++), Fortran (gfortran), Go (gccgo).
  - Intermediate representation of gcc: GIMPLE (three address code) and its extended variants
  - Backends for various processors and hw architectures (differ in endianess, word size, calling conventions, instructions, registers, …)

Another example

- .NET compilers:
  - Various frontends
  - Intermediate representation (bytecode) MSIL (Microsoft Intermediate Language), interpreted by .NET environment

Intermediate representations

Basic kinds of IRs

- High-level IRs
  - Correspond to the input languages, later transformed to lower level Irs or back to the input code
  - Example: abstract syntax tree (AST), high-level three address code (3AC)
- Intermediate level – language independent, to reflect a range of features, good for optimizations
  - Example: lower-level three address code (3AC)
- Low-level – similar to particular assembler, suitable for hw-dependent optimizations

Basic kinds of IRs – another view

- Trees (DAGs):
  - Derivation tree
  - Abstract syntax tree (AST), DAG
- Linear:
  - Three address code

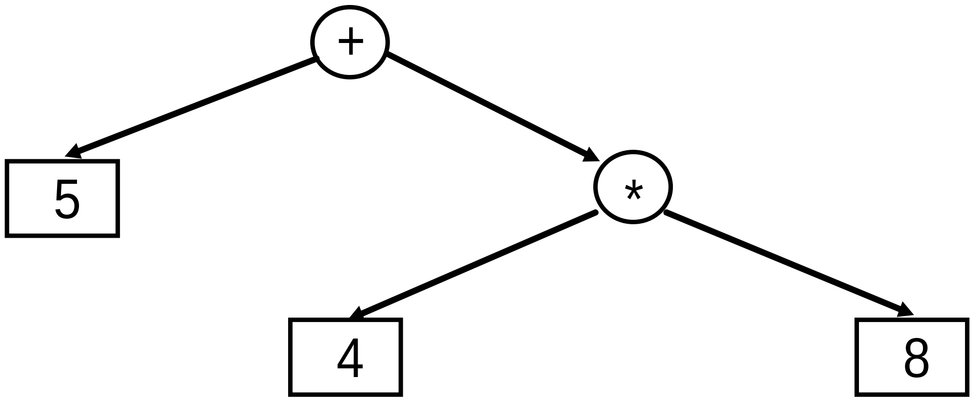
AST and three address code may be the two most common IRs.

IR: Abstract syntax tree

- See also bachelor course BI-PJP (several slides from this course follow)

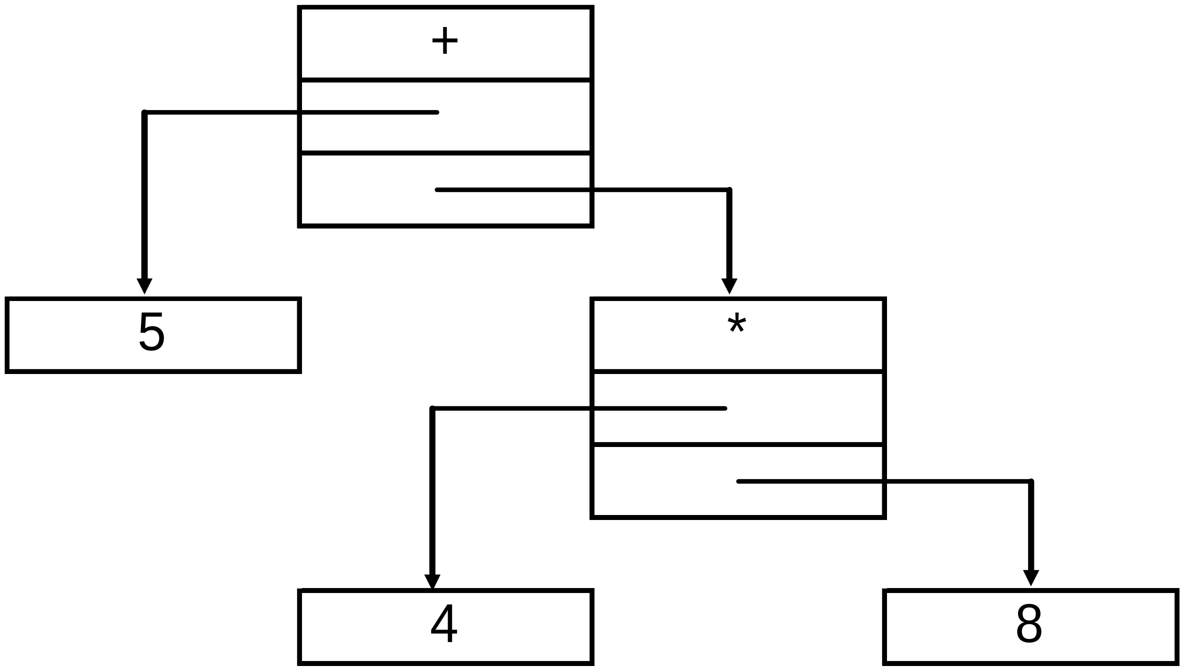
Abstraktní syntaktický strom

- Reprezentuje zdrojový program jako strukturu složenou z operátorů a jejich operandů
- Vnitřní uzly stromu jsou operátory, jejich následníci jsou operandy, nad nimiž se mají operace provést
- Koncové uzly reprezentují jednoduché operandy (např. číslo)
- Příklad: abstraktní syntaktický strom reprezentují výraz
  5 + 4 * 8



Příklad – aritmetické výrazy s konstantami

- Překlad aritmetických výrazů s celočíselnými konstantami do stromové reprezentace a následné vyhodnocení průchodem stromu
- Stačí dva typy uzlů:
  - – binární operátor
  - – celočíselná konstanta
- Příklad stromu pro výraz 5 + 4 * 8



Příklad – aritmetické výrazy s konstantami

- Programová realizace stromu:
  - v Pascalu variantními záznamy
  - v C pomocí *struct* a *union*
  - v C++ a v Javě pomocí tříd a objektů

Realizace stromu v C

```
typedef enum {intkonst, binop} DruhUzlu;

typedef union uzel Uzel;

typedef struct {
   DruhUzlu druh;
   int      hodn;
} IntKonst;

typedef struct {
   DruhUzlu druh;
   char     op;
   Uzel     *levy, *pravy;
} BinOp;

union uzel {
   DruhUzlu druh;
   IntKonst intkonst;
   BinOp    binop;
};
```

Realizace stromu v C

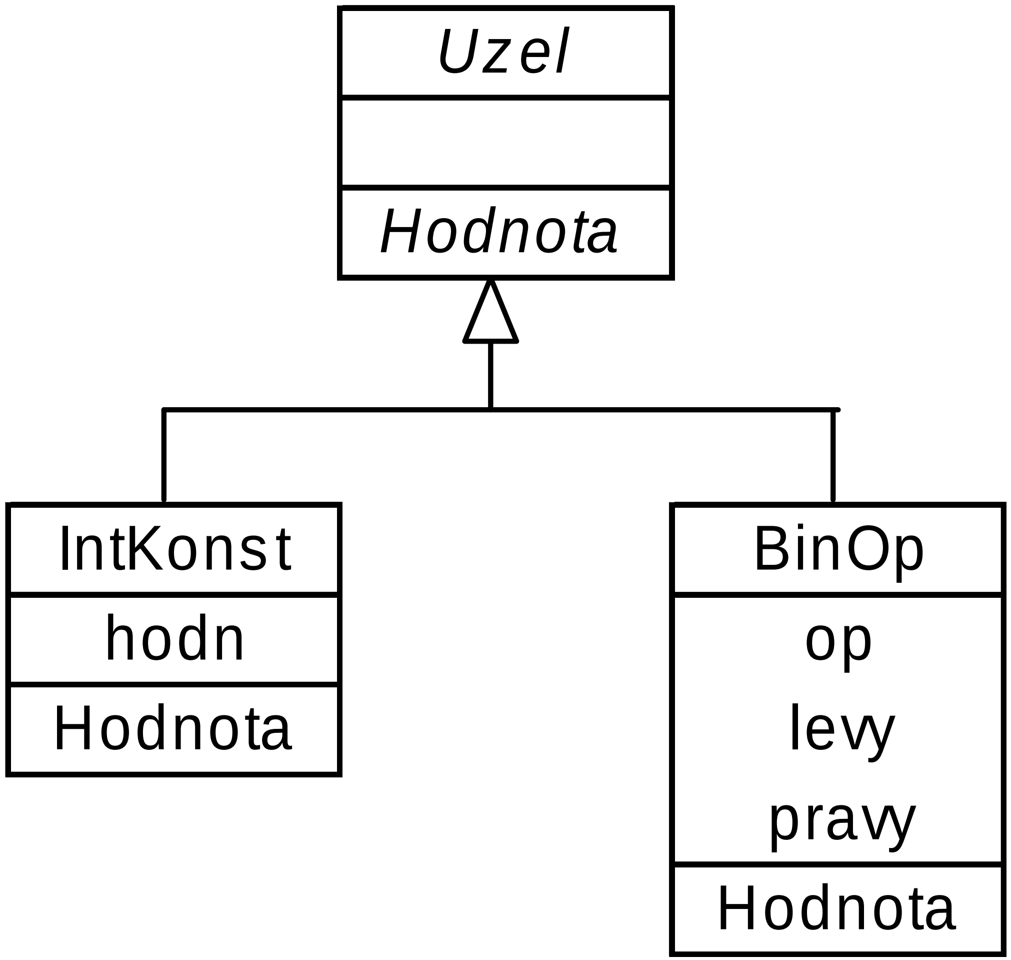
```
Uzel *NewIntKonst(int c)
{
   Uzel *u = (Uzel*)malloc(sizeof(IntKonst));
   u->druh = intkonst;
   u->intkonst.hodn = c;
   return u;
}

Uzel *NewBinOp(char o, Uzel *l, Uzel *p)
{
   Uzel *u = (Uzel*)malloc(sizeof(BinOp));
   u->druh = binop;
   u->binop.op = o;
   u->binop.levy = l;
   u->binop.pravy = p;
   return u;
}
```

Realizace stromu v C

```
int Hodnota(Uzel *u)
{
   switch (u->druh) {
   case intkonst:
      return u->intkonst.hodn;
   case binop:
      {
         int l = Hodnota(u->binop.levy);
         int p = Hodnota(u->binop.pravy);
         switch (u->binop.op) {
         case '+':
            return l+p;
         case '-':
            return l-p;
         case '*':
            return l*p;
         case '/':
            return l/p;
         }
      }
   }
}
```

Realizace stromu v C++

- Abstraktní třída *Uzel* s potomky *IntKonst* a *BinOp*



Realizace stromu v C++

```
class Uzel {
public:
   virtual int Hodnota() = 0;
};

class IntKonst : public Uzel {
   int hodn;
public:
   IntKonst(int c);
   virtual int Hodnota();
};

class BinOp : public Uzel {
   char  op;
   Uzel  *levy, *pravy;
public:
   BinOp(char, Uzel*, Uzel*);
   virtual int Hodnota();
};
```

Realizace stromu v C++

```
IntKonst::IntKonst(int c) {
   hodn = c;
}

int IntKonst::Hodnota() {
   return hodn;
}

BinOp::BinOp(char o, Uzel *l, Uzel *p) {
   op = o; levy = l; pravy = p;
}

int BinOp::Hodnota() {
   int l = levy->Hodnota();
   int p = pravy->Hodnota();
   switch (op) {
   case '+':
      return l+p;
   case '-':
      return l-p;
   case '*':
      return l*p;
   case '/':
      return l/p;
   }
}
```

Syntaxe a atributy

- Vyjdeme ze stejné bezkontextové gramatiky jako při vyhodnocování při překladu (nezavedeme výstupní symboly)
- Neterminálům přiřadíme atributy, jejichž hodnotami budou ukazatele na vytvořené stromy (typu *Uzel**)
- Syntaxe:

$$E \rightarrow T\ E'$$
$$E' \rightarrow +\ T\ E' \ \mid\ -\ T\ E' \ \mid\ \varepsilon$$
$$T \rightarrow F\ T'$$
$$T' \rightarrow *\ F\ T' \ \mid\ /\ F\ T' \ \mid\ \varepsilon$$
$$F \rightarrow c \ \mid\ (\ E\ )$$

symbol	dědičné atributy	syntetizované atributy
E, *T*, *F*		*suzel*
E' , *T'*	*duzel*	*suzel*
c		*shod*

Sémantická pravidla

syntaktické pravidlo	sémantická pravidla
$E \to T\ E'$	$E'.duzel = T.suzel$ $E.suzel = E'.suzel$
$E'^0 \to +\ T\ E'^1$	$E'^1.duzel = new\ BinOp('+', E'^0.duzel, T.suzel)$ $E'^0.suzel = E'^1.suzel$
$E'^0 \to -\ T\ E'^1$	$E'^1.duzel = new\ BinOp('-', E'^0.duzel, T.suzel)$ $E'^0.suzel = E'^1.suzel$
$E' \to \varepsilon$	$E'.suzel = E'.duzel$
$T \to F\ T'$	$T'.duzel = F.suzel$ $T\ .suzel = T'.suzel$
$T'^0 \to *\ F\ T'^1$	$T'^1.duzel = new\ BinOp('*', T'^0.duzel, F.suzel)$ $T'^0.suzel = T'^1.suzel$
$T'^0 \to /\ F\ T'^1$	$T'^1.duzel = new\ BinOp('/', T'^0.duzel, F.suzel)$ $F'^0.suzel = T'^1.suzel$
$T' \to \varepsilon$	$T'.suzel = T'.duzel$
$F \to c$	$F.suzel = new\ IntKonst(c.shod)$
$F \to (\ E\ )$	$F.suzel = E.suzel$

Rekurzivní sestup

```
Uzel *Vyraz(void)
{
   /* E -> T E' */
   return ZbVyrazu(Term());
}

Uzel *ZbVyrazu(Uzel *dhod)
{
   switch (Symb) {
   case PLUS:
      /* E' -> + T E' */
      CtiSymb();
      return ZbVyrazu(new BinOp('+', dhod, Term()));
   case MINUS:
      /* E' -> - T E' */
      CtiSymb();
      return ZbVyrazu(new BinOp('-', dhod, Term()));
   default:
      /* E' -> e */
      return dhod;
   }
}
```

Rekurzivní sestup

```
Uzel *Term(void)
{
   /* T -> F T' */
   return ZbTermu(Faktor());
}

Uzel *ZbTermu(Uzel *dhod)
{
   switch (Symb) {
   case TIMES:
      /* T' -> * F T' */
      CtiSymb();
      return ZbTermu(new BinOp('*', dhod, Faktor()));
   case DIVIDE:
      /* T' -> / F T' */
      CtiSymb();
      return ZbTermu(new BinOp('/', dhod, Faktor()));
   default:
      /* T' -> e */
      return dhod;
   }
}
```

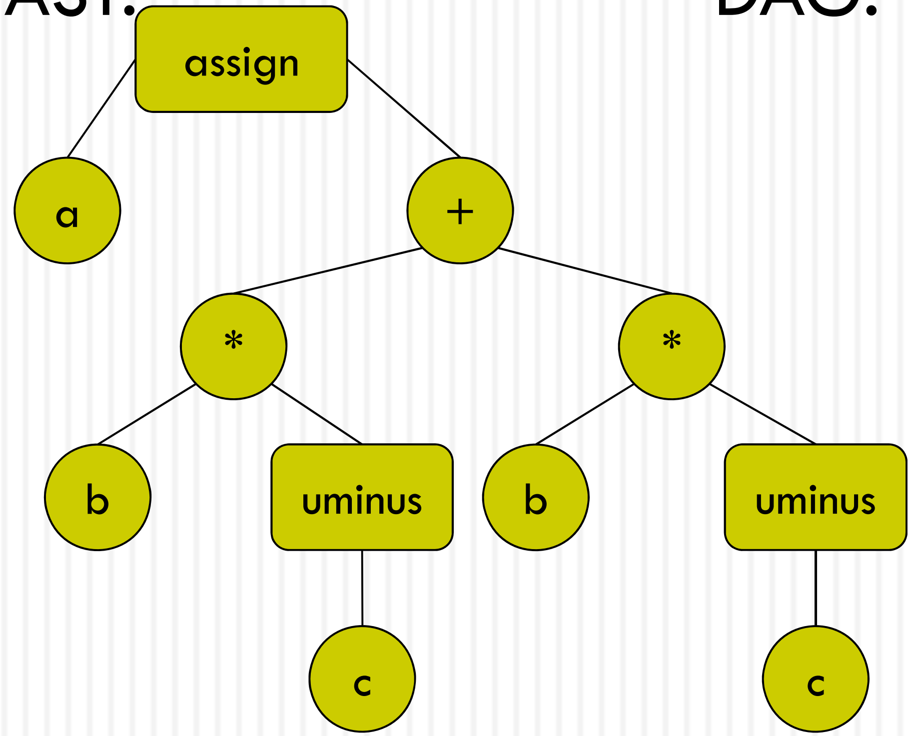
Rekurzivní sestup

```
Uzel *Faktor(void)
{
   switch (Symb) {
   case NUMB: {
      /* F -> c */
      int cshod;
      Srovnani_NUMB(&cshod);
      return new IntKonst(cshod);
   }
   case LPAR: {
      /* F -> ( E ) */
      Uzel *Eshod;
      CtiSymb();
      Eshod = Vyraz();
      Srovnani(RPAR);
      return Eshod;
   }
   default:
      Chyba("chyba pri expanzi F");
   }
}
```

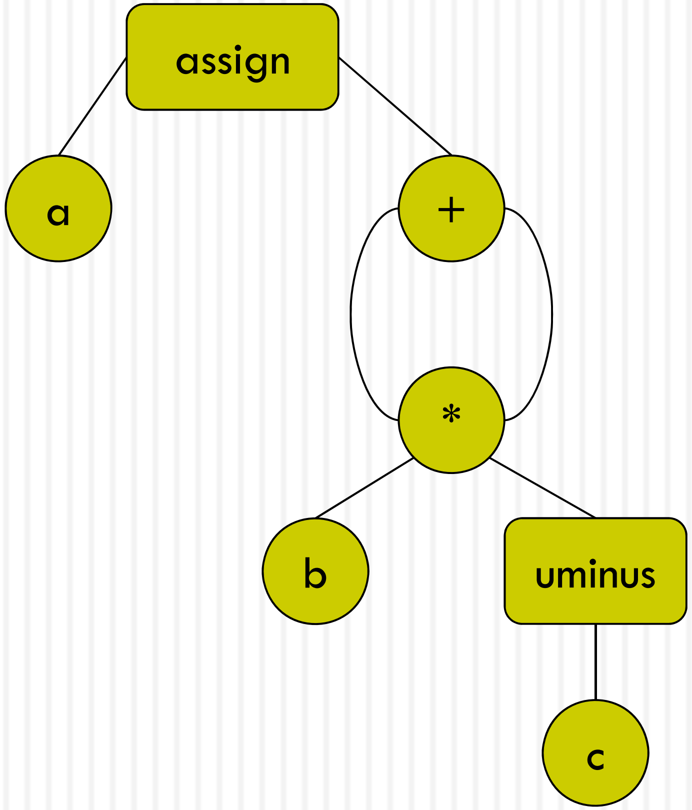
DAG

Space improvement of AST, repeats of subtrees are joined to together: statement a = b*-c+b*-c

AST:



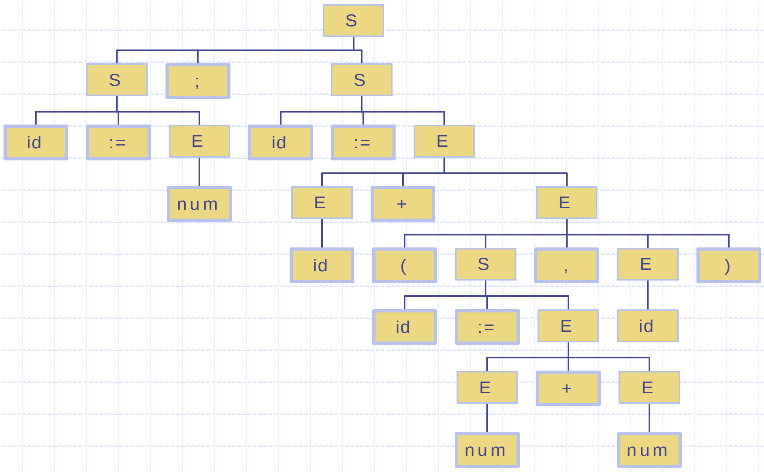
DAG:



- Implementation: to be shown on the board

Concrete Parse Tree Example

a := 7; b := c + (d := 5 + 6, d)



Concrete Parse Trees

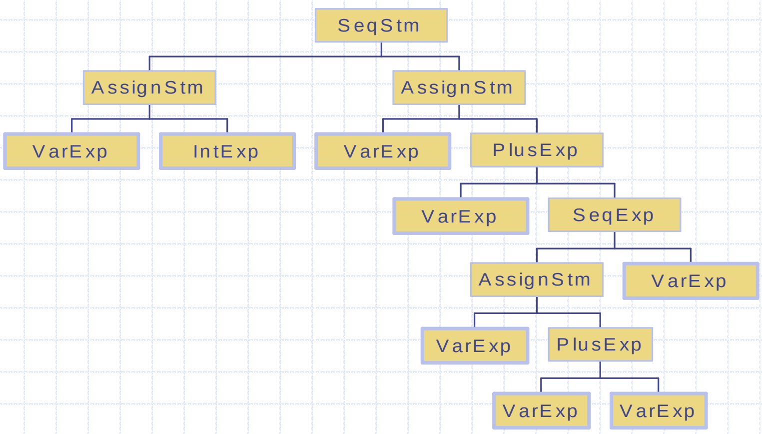
- Exactly one leaf for each token of the input
- Exactly one internal node for each grammar rule reduced during the parse
- *But:* Inconvenient to use
  - Redundant punctuation marks etc.
  - Structure of the tree depends too much on the grammar (different trees after elimination of ambiguity in the grammar)
  - Too detailed

Abstract Syntax Trees (AST)

- Clean interface between parser and later phases of the compiler
- More convenient to use
- Later phases of the compiler can traverse the Abstract Syntax Tree (Multi-pass Compilation)
- Possibly based on ambiguous grammar! Impractical for parsing!
- *But:* Constructed from concrete syntax
  - Ambiguities already resolved
  - Not used for parsing, but as a result of parsing

Example

a := 7; b := c + (d := 5 + 6, d)



Positions

- Later phases (e.g. semantic analysis) must be able to report the position of an error
- Current position of the lexer cannot be used because
  - it has already reached the end of file before later phase even begins
  - one wants to keep phases independent from each other
- *Solution:* Source-file position of each node of the AST must be remembered
  - `pos` fields in the AST

Sematic Analysis

- Checks if programs are semantically correct
- Example:

```
var x,y: integer;
var z: boolean;

z := x + y;
```

Syntactially correct, but semantic error!!!

Tasks of the Semantic Analysis

- Type-checking of expressions
- Relating variable declarations to variable uses
- Matching function declarations and function calls
- Maintains a *symbol table* or *environment* with *bindings*
- Operates on AST

Symbol Tables

- Contains an entry (*binding*) for each variable declaration, function definition and type declaration
- Each binding contains the symbol and attributes, e.g. {`a` → `string`}
- Often distinguished between *value environment* and *type environment*
  - value enviroment: variable declarations and function definitions
  - type environment: type declarations

Scopes

- Each variable in a program has a *scope* in which it is visible.
- As the semantic analysis reaches the end of each scope, the bindings local to that scope are discarded.

- Example:

`var x,y,z: integer;`	$\sigma_1 = \sigma_0 + \{x \rightarrow int, y \rightarrow int, z \rightarrow int\}$
`procedure test();`	$\sigma_2 = \sigma_1 + \{\ \}$
`var x,y: integer;`	$\sigma_3 = \sigma_2 + \{x \rightarrow int, y \rightarrow int\}$
`begin`	
`   ...`	
`   z := x + y;`	σ_3
`end;`	σ_2
`x := y;`	σ_1

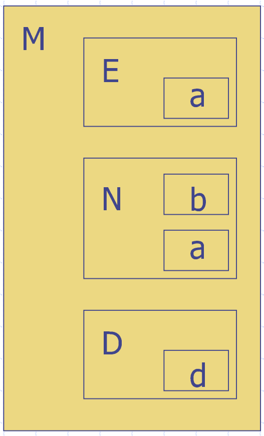
Multiple Environments

- Several active environments at once (possible in some languages)
- Each module, class, record etc. in the program has a symbol table σ of its own

Example

```
package M;

class E {
   static int a = 5;
}

class N {
   static int b = 10;
   static int a = E.a + b;
}

class D {
   static int d = E.a + N.a;
}
```

Multiple Symbol Tables



$\sigma_1 = \{a \to int\}$

$\sigma_2 = \{E \to \sigma_1\}$

$\sigma_3 = \{b \to int,\ a \to int\}$

$\sigma_4 = \{N \to \sigma_3\}$

$\sigma_5 = \{d \to int\}$

$\sigma_6 = \{D \to \sigma_5\}$

$\sigma_7 = \sigma_2 + \sigma_4 + \sigma_6$

$\sigma_8 = \sigma_0 + \{M \to \sigma_7\}$

Program is compiled in environment σ_8

Symbol Table Entries

- Variable Declaration
  - Name
  - Type
    - Base Type, No. of Elements (Array)
    - List of fields (record)
- Function Definition
  - Name
  - No. and type of arguments
  - Return type
- Type Declaration
  - Name
  - Type
    - Base Type, No. of Elements (Array)
    - List of fields (record)

Imperative Symbol Table

```
public class Table {
   public Table();
   public void put(Symbol key, Object value);
   public Object get(Symbol key);
   public void beginScope();
   public void endScope();
}
```

- Important: `endScope` must restore the state the symbol table was in before the call previous call to `beginScope`
- Can be implemented as list, tree, hash table with external chaining...
- But: Efficient algorithm and implementation necessary due to large number of accesses to symbol table

Symbol Table Implementation

- Hash Table with external chaining
  - Insert
    - Determine bucket with help of a hash function
    - *i* th bucket is a linked list of all those elements whose key hash to i mod SIZE
    - Insertion of a key that already exists puts the new key *earlier* than the other key in the list
    - Subsequent lookup (or remove) finds newer element first

Symbol representation

- Comparing strings for symbol table lookup is costly
- Convert each string to an integer (class `Symbol` in your implementation)
  - Comparing two symbols for equality is fast
  - Extracting an integer hash key is fast
  - Comparing two symbols for ´greater than´ is fast (in case we wish to build binary search trees)