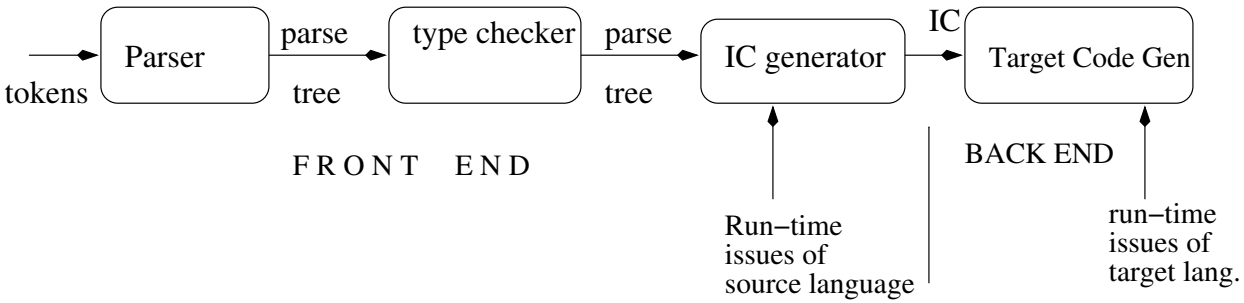


- IC Design Issues:

  What kind of language

  Storage organization for symbols and flow of control

  IC templates for source language constructs

- IC Design

  Closer in spirit to source language execution paradigm but simplified

or lower level

procedural langs: a high-level general purpose assembler

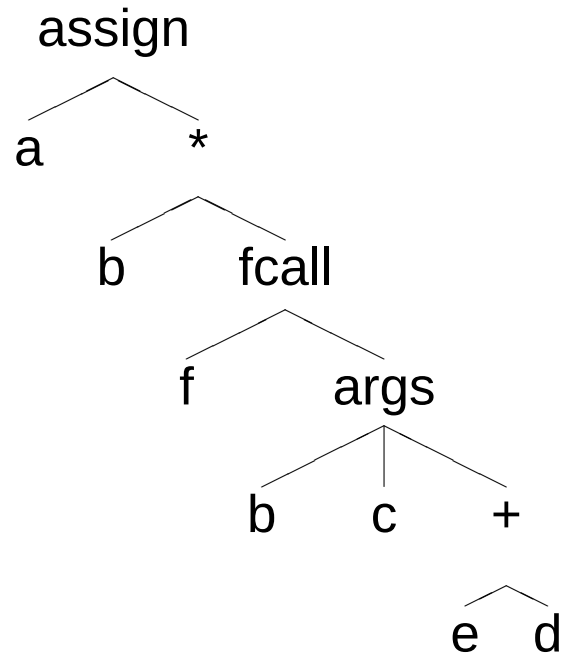
functional langs: a high-level function description or manipulation language (e.g., λ-calculus)

- Intermediate representation for Imperative languages

  1. syntax trees (high level rep./no storage concerns)
  2. postfix rep. (linearized syntax tree)
  3. TAC (three address code) : (low level rep./storage for symbols)

syntax tree vs. postfix

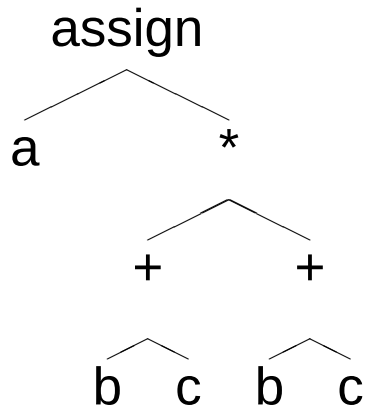
```
a := b * f(B,c,e+d);
```



```
a b f b c e d + 3 args fcall * assign
```

Hard to show common subexpressions in postfix

```
a:= (b+c)*(b+c)
```



- Three-address Code (TAC) for `a:=b*f(b,c,e+d)`

```
param b
param c
t1 := e+d
param t1
fcall f,3
return t2
t3 := b + t2
mv t3 a
```

- TAC instruction set

Some are quite high level; very few machines have actual counterparts of these instructions directly implemented

1. assignment (`label:`) `x := y op z`
2. unconditional branch: `goto label`
3. conditional branch: `bz x L` `bnz x L`
4. `param x`

   `call f,n`

   `return x`
5. indexed expr.: `x:= y[i]` or `x[i]:=y`

   `x:= y[i]`+b needs 4 addresses (not TAC)
6. reference: `x:= &y`

7. dereference: `x := *y`

- Small IC instruction set simplifies target code generation but produces long sequence of TC instructions

- Large sets may not be easily portable to all architectures (like 4–7)

- TAC generation for

  SL declarations

  IC declarations (temporary storage)

  SL expressions

  SL instructions

- Syntax-directed definition for translating large expressions to TAC

  Conventions: IC function's name indicates arity as well

  `3ac(op,x,y,z)  x:= y op z`

  `2ac(op,x,y)`

  `1ac(op,x)   eg.  param x`

  `2copy(x,y)  x:=y`

  Every grammar symbol has two attributes:

  `X.code` (code segment for X);

  `X.place` (value holder for X)

`x:= a*b+c` translates to

```
t1 := a*b
t2 := t1 + c
x := t2
```

How to generate new symbols t_i?

```
S -> id := E   {S.code= E.code ||
               2copy(id.place,E.place)}

E -> E + E     { E.place=newtemp();
               E.code = E1.code || E2.code ||
               3ac(add,E.place,E1.place,E2.place)}

E -> E * E

E -> -E        {E.place=newtemp();
               E.code=E1.code ||
               2ac(uminus,E.place,E1.place)}

E -> id        { E.place= id.place; E.code= nil}
```

e.g., a:=b+e*-c