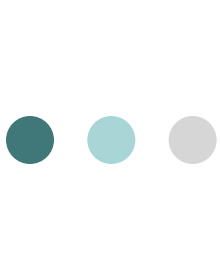
Semantic Analysis 2

Introduction to Intermediate Code Generation

Dr. William Harrison
harrisonwl@missouri.edu
CS 4430 Compilers I

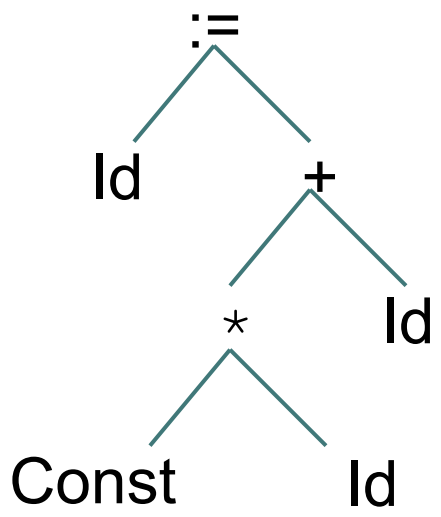


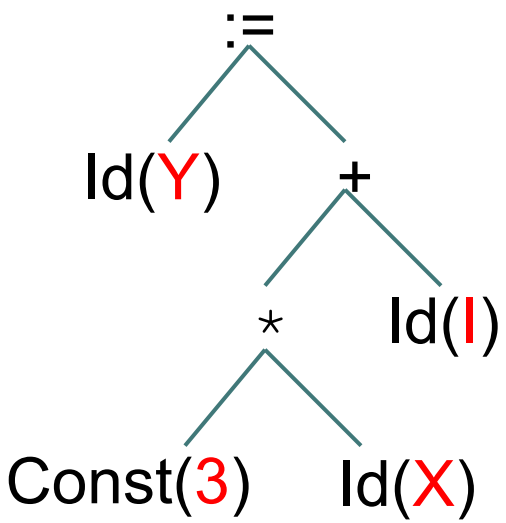
Today

- Continuing
  - intermediate representations
  - syntax directed compilation
    - one pass
    - multiple passes

Review: ASTs with "attributes"

Attribute grammars are CFGs with extra information (a.k.a., "attributes") stored at the nodes

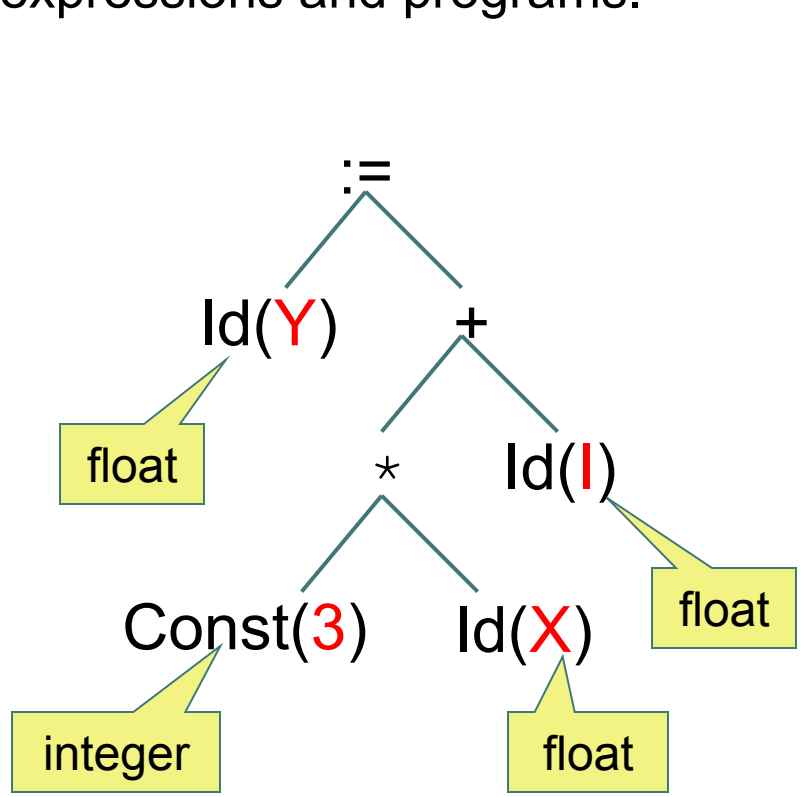


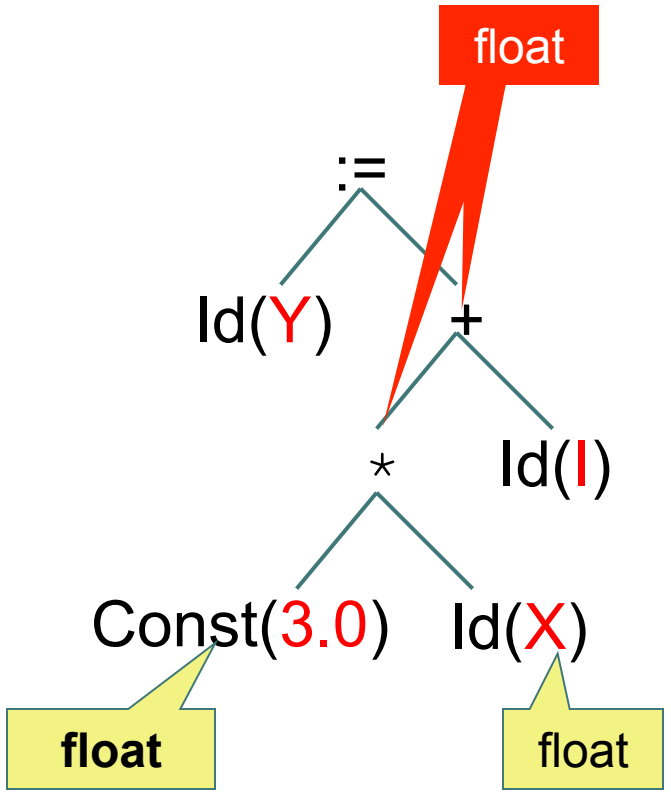


** red data are "initial attributes" in the lingo.*

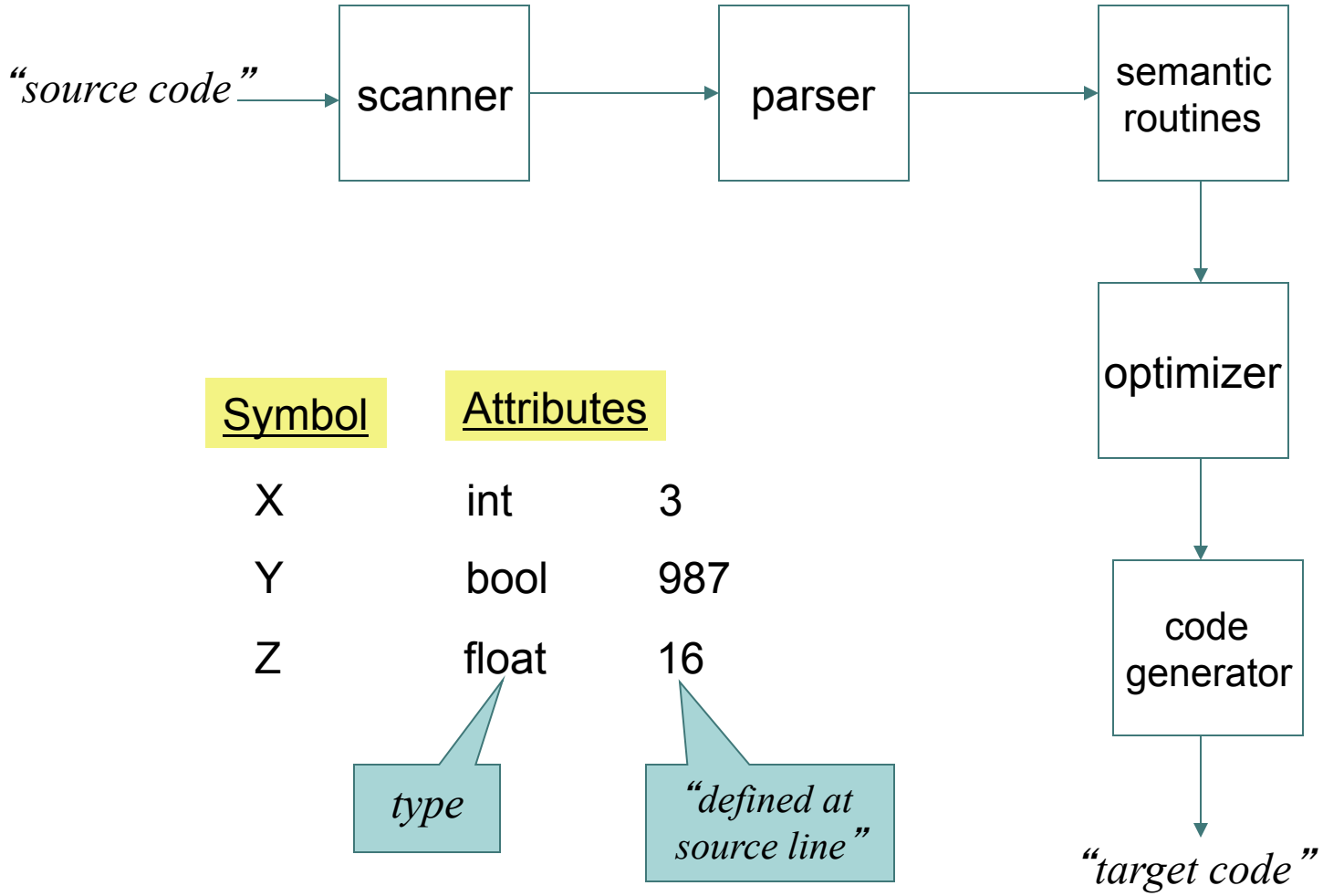
Review: Attribute grammars and static checking

Static properties of programs can be calculated from attribute grammar representations. Ex: type inference for expressions and programs.





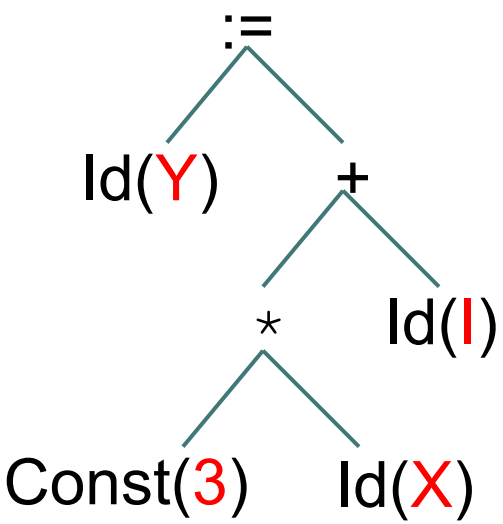
Review: attribute grammar ideas occur everywhere in a compiler



Review: Intermediate Representations

- A.k.a., “IR” or “Intermediate Code”
- Varieties of IR
  - abstract syntax trees
    - written in a particular style to resemble target code
  - three-address code
    - a.k.a. register transfer language (RTL)
  - “Enriched” forms: IR annotated with useful information
    - def-use, use-def chains: connects definition and use of variables
    - Single Static Assignment form (SSA)
- Many compilers use multiple forms
  - Ex: GCC uses two – ASTs and RTL.

Review: three-address code/ RTL



3-address/RTL representation

R ← 3 * X;
Y ← R + I;

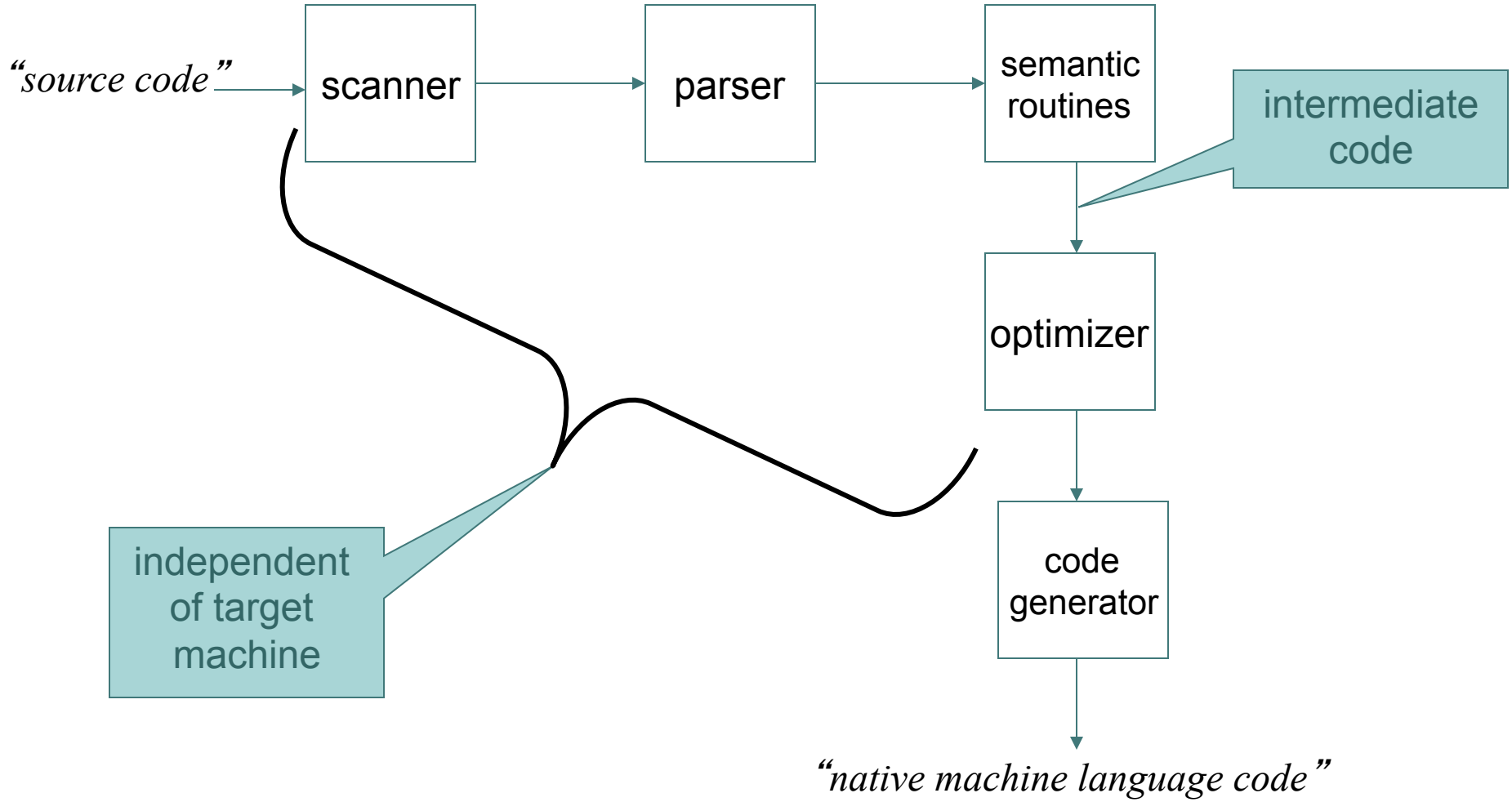
Advantage: RTL enforces simplicity of expressions

Disadvantage: not as flexible

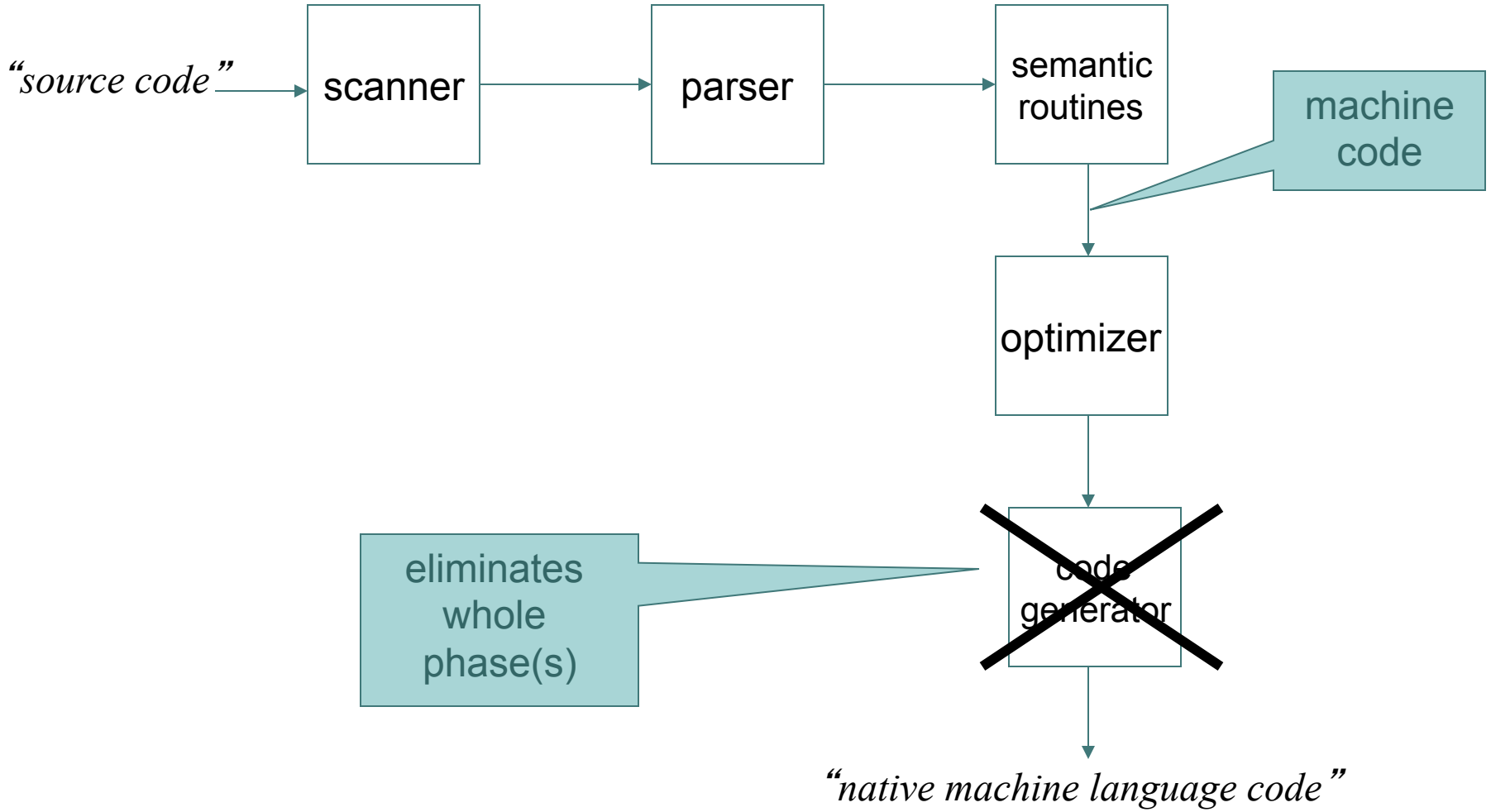
Why use IR? Why not directly produce native code?

- IR is a “virtual machine” like abstraction
  - VM can contain language constructs similar to high-level source constructs
    - thus easier to produce
  - IR more “regular” than native code
    - easier to understand, optimizations are simpler
- Portability
  - many optimizations are not machine specific
  - front-end and middle phases can be re-used “as-is” for other compilers
- Target machine complexities don't have much impact on most of compiler
  - i.e., special architectural features need not be worried about until very late compiler phases

Using an IR: most of compiler is "re-targetable"



Advantages of direct generation of native code



A Sample IR

```
(READI, arg)                -- read stdin into arg

(WRITEI,arg)                -- read stdout from arg

(GT,arg1,arg2,arg3)         -- arg3 := arg1 > arg2
                            -- true is 1, false is 0
(ADDI,arg1,arg2,arg3)       -- arg3 := arg1 + arg2
                            -- assumes integer args
(JUMP0,arg1,arg2)           -- jumps to label arg2
                            -- if arg1=0
(JUMP,arg)                  -- jumps to label arg

(LABEL,arg)                 -- defines a label arg
```

Example Translation into IR

Source Code

```
begin
    read(A,B);
    if A > B then
        C := A + 5;
    else
        C := B + 5;
    end if;
    write(2 * (C - 1));
end
```

IR

```
(READI, A)
(READI, B)
(GT,A,B,t1)
(JUMP0,t1,L1)
(ADDI,A,5,C)
(JUMP,L2)
(LABEL,L1)
(ADDI,B,5,C)
(LABEL,L2)
(SUBI,C,1,t2)
(MULTI,2,t2,t3)
(WRITEI,t3)
```

Extended CFG for Micro

<program>	→ begin <statement list> end
<statement list>	→ <statement> {<statement list>}
<statement>	→ ID := <expression> ;
<statement>	→ read (<id list>) ;
<statement>	→ write (<expr list>) ;
<id list>	→ ID { , ID }
<expr list>	→ <expression> { , <expression> }
<expression>	→ <primary> { <add op> <primary> }
<primary>	→ (<expression>)
<primary>	→ ID
<primary>	→ INTLITERAL
<add op>	→ PLUSOP
<add op>	→ MINUSOP
<system goal>	→ <program> SCANEOF

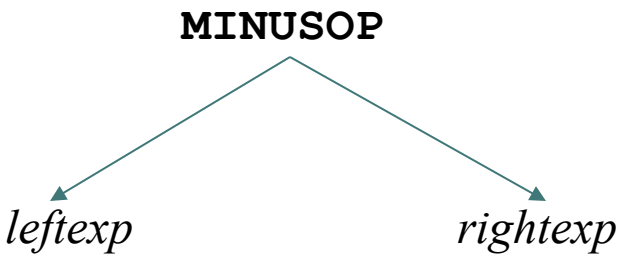
** Note: some things left out here like if-then-else, booleans, etc.*

Translating **simple** expressions

Sample in Concrete Syntax

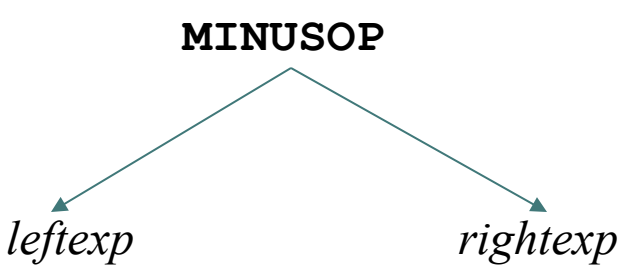
```
c - 1
```

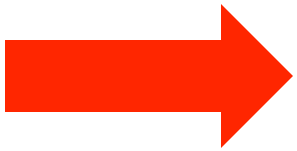
Corresponding Abstract Syntax



Translating **simple** expressions

Corresponding Abstract Syntax





(SUBI *leftexp*,*rightexp*,*targ***)**

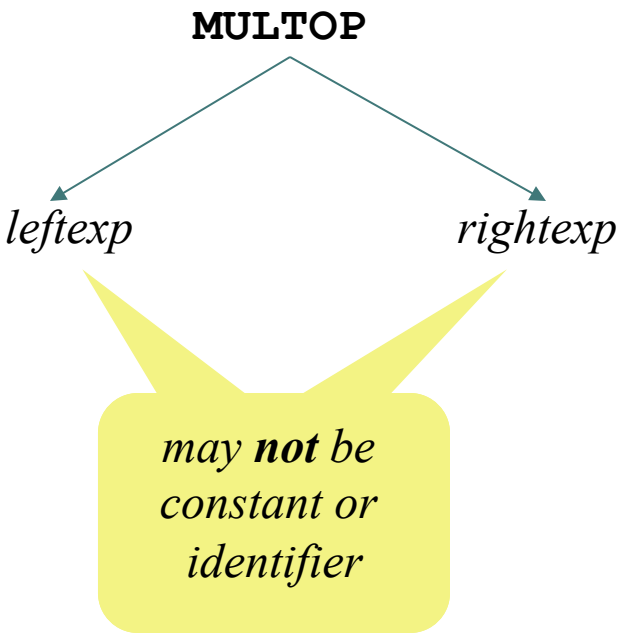
the value of every simple expression is stored in a (possibly generated) register

Translating composite expressions

Sample in Concrete Syntax

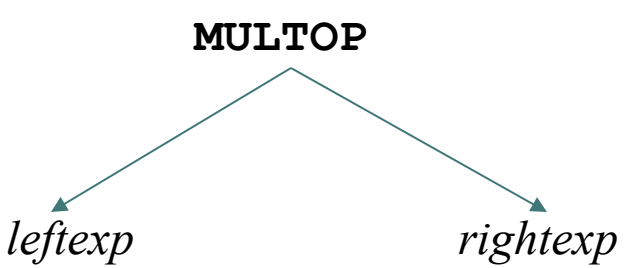
```
2 * (C – 1)
```

Corresponding Abstract Syntax



Translating **composite** expressions

Corresponding Abstract Syntax



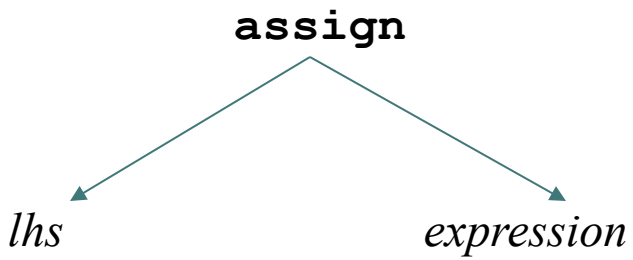
<code for *leftexp*, result in *t1*>
<code for *rightexp*, result in *t2*>
(MULTI *t1* **,** *t2* **,** *targ***)**

Translating assignment "… := …"

Sample in Concrete Syntax

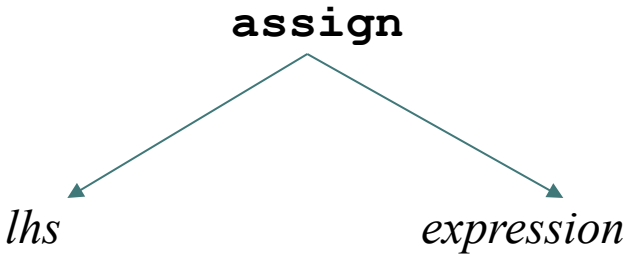
```
C := A + 5
```

Corresponding Abstract Syntax

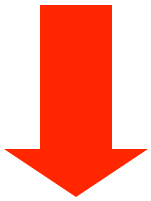


Translating assignment "… := …"

Abstract Syntax



<code for *expression,* result in *t*>



<code for *expression,* result in *lhs* >

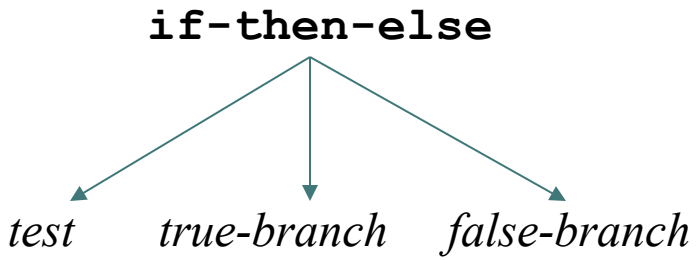
overwrite temporary with *lhs*

Translating "if … then … else … end if"

Sample in Concrete Syntax

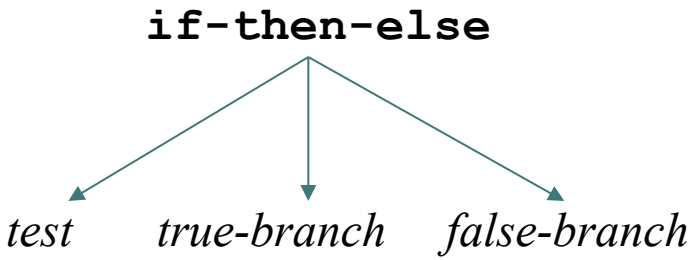
```
if A > B then
   C := A + 5;
else
   C := B + 5;
end if;
```

Corresponding Abstract Syntax



Translating "if … then … else … end if"

Abstract Syntax



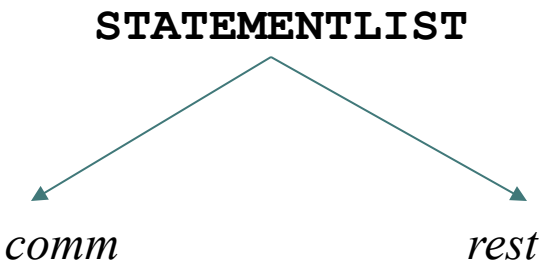
<code for *test,* result in *t*>
(JUMP0, *t*, *false-label***)**
<code for *true-branch*>
(JUMP, *exit-label***)**
(LABEL, *false-label***)**
<code for *false-branch*>
(LABEL, *exit-label***)**

Translating "$\texttt{command}_1$; $\texttt{command}_2$"

Sample in Concrete Syntax

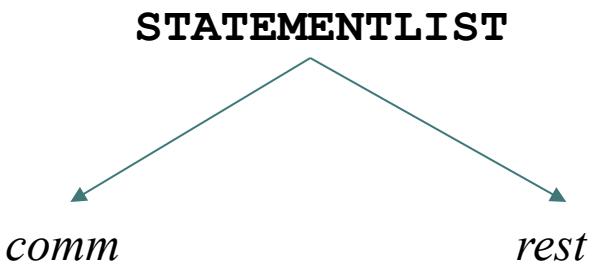
```
C := A + 5;
C := B + 5
```

Corresponding Abstract Syntax



Translating "$\texttt{command}_1$; $\texttt{command}_2$"

Abstract Syntax



<code for *comm*>
<code for *rest*>

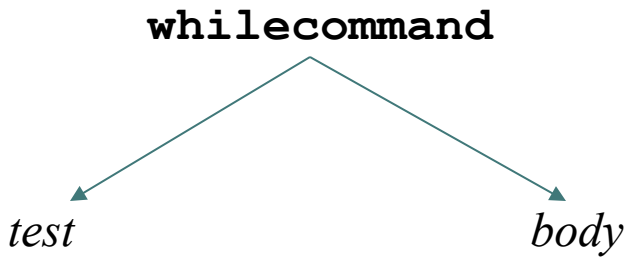
* *I.e., a sequence of source is a sequence in IR*

How would you translate while loops?

Sample in Concrete Syntax

```
while A > B
  do
     C := A + 5;
  endwhile;
```

Corresponding Abstract Syntax



What next?

How do we combine code generation with a YACC style specification?

```
                …
<statement>     → ID := <expression> ;     {        }
<statement>     → read (<id list>) ;       {        }
<statement>     → write (<expr list>) ;    {        }
                …
```

create AST in C

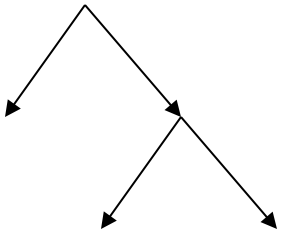
What next?

How do we combine code generation with a YACC style specification?

```
                …
<statement>     → ID := <expression> ;    {      }
<statement>     → read (<id list>) ;      {      }
<statement>     → write (<expr list>) ;   {      }
                …
```

create AST in C

<program>



translation schemas

Output IR

```
(READI, A)
(READI, B)
(GT,A,B,t1)
(JUMP0,t1,L1)
        …
```

Next Class

- Continue “Semantic Processing”