ΕΠΛ323 - Θεωρία και Πρακτική Μεταγλωττιστών

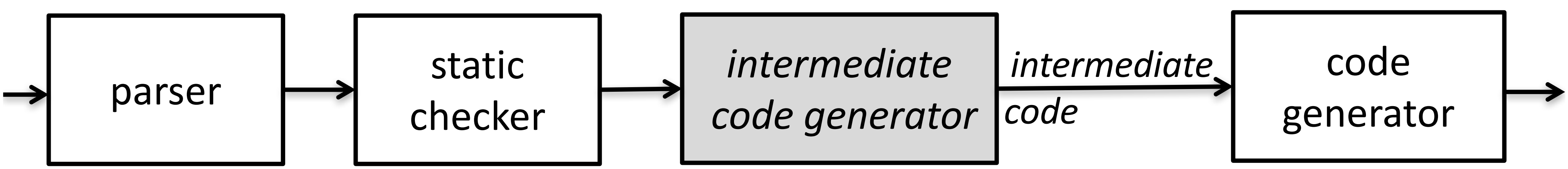
Lecture 10b

Intermediate Code Generation

Elias Athanasopoulos
eliasathan@cs.ucy.ac.cy

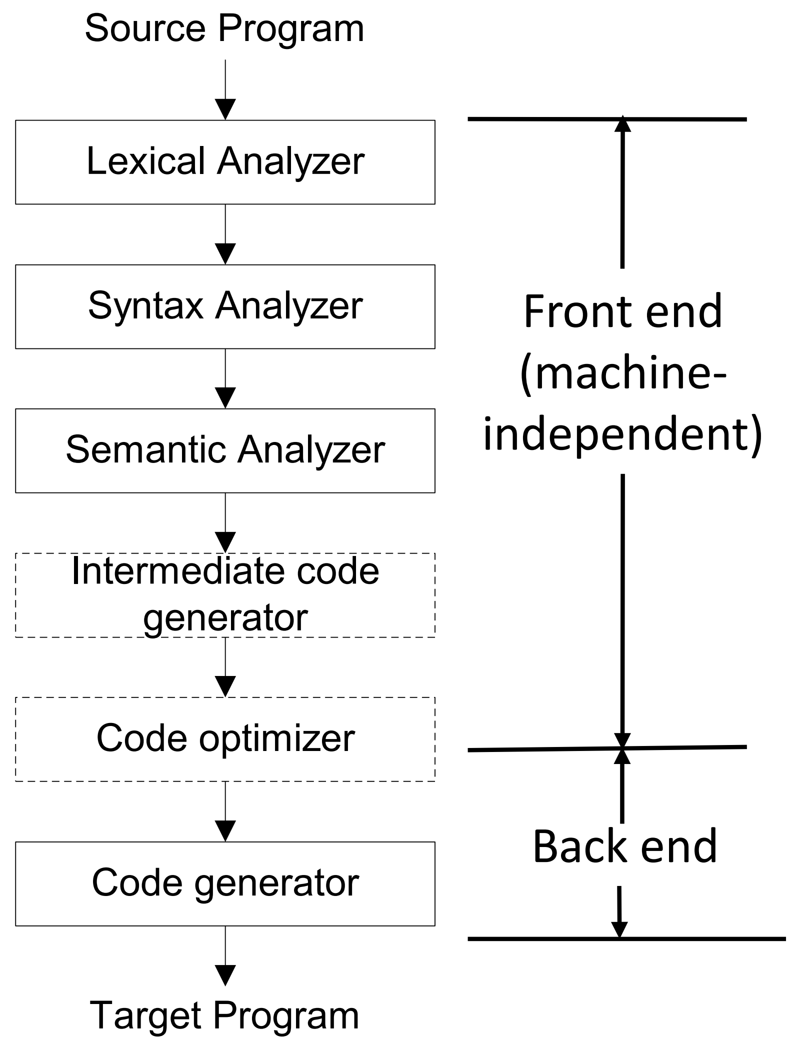
Need for Intermediate Code

- Retargeting is facilitated
  – Adding back ends for additional architectures
- Optimizations
  – Perform architecture-agnostic optimizations



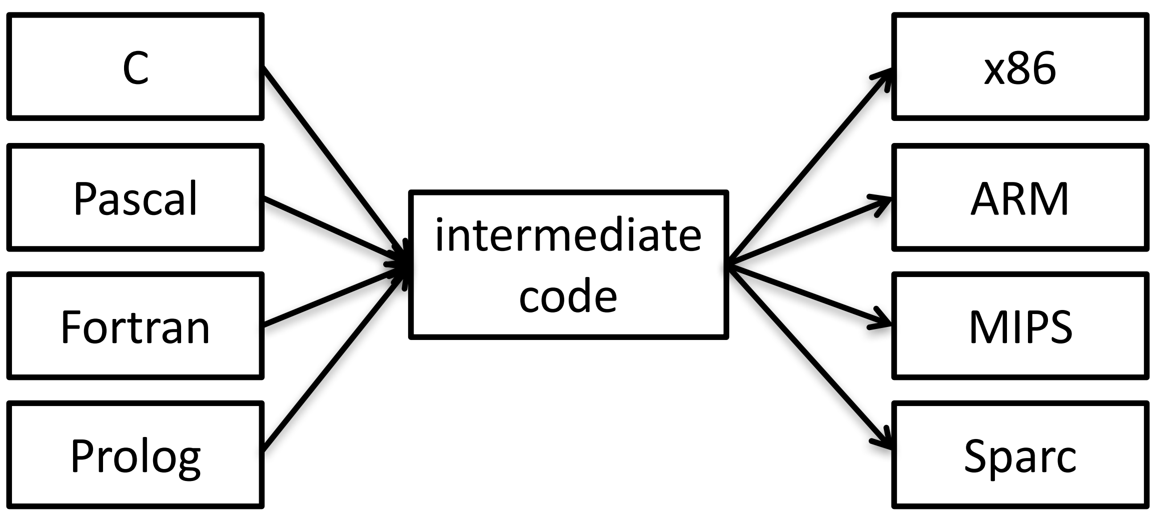
Front and Back ends

- Separation of common tasks
- Makes design and implementation easier
- K compilers for N machines
  - N back ends, K front ends
  - Instead of K*N compilers



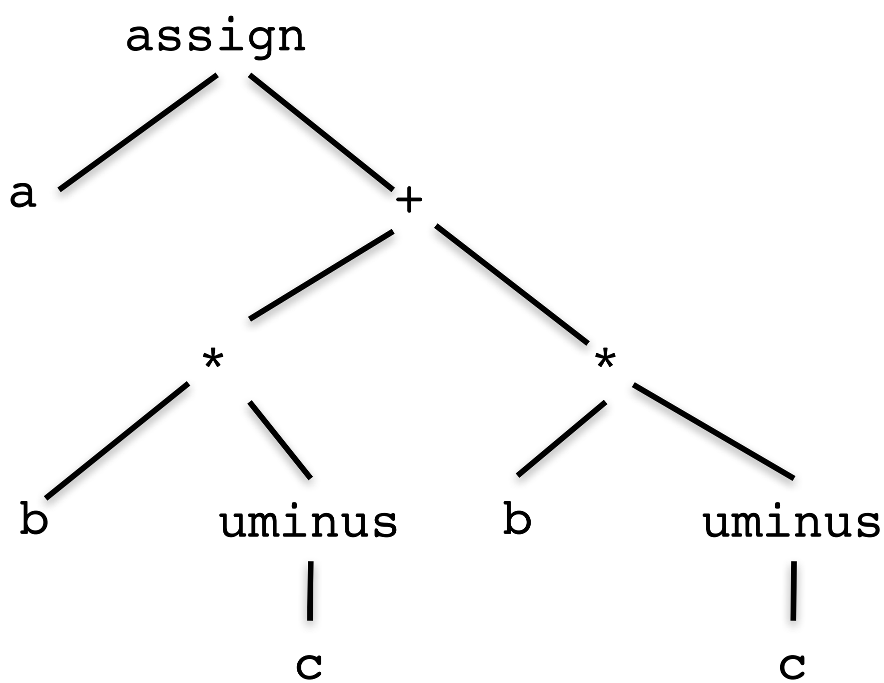
Types of Intermediate Languages

- Graphical representation
  – AST, DAGs
- Postfix notation
- Three-Address Code

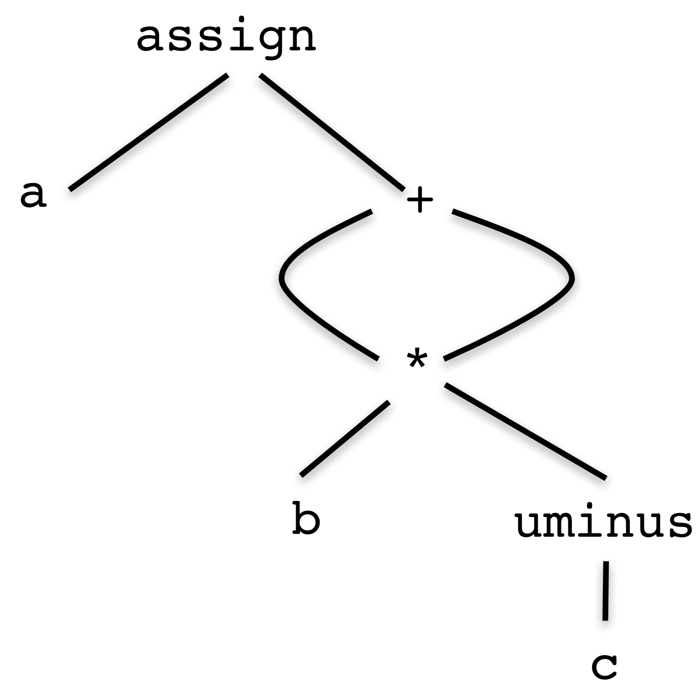


Graphical Representations

a:=b*-c+b*-c



AST

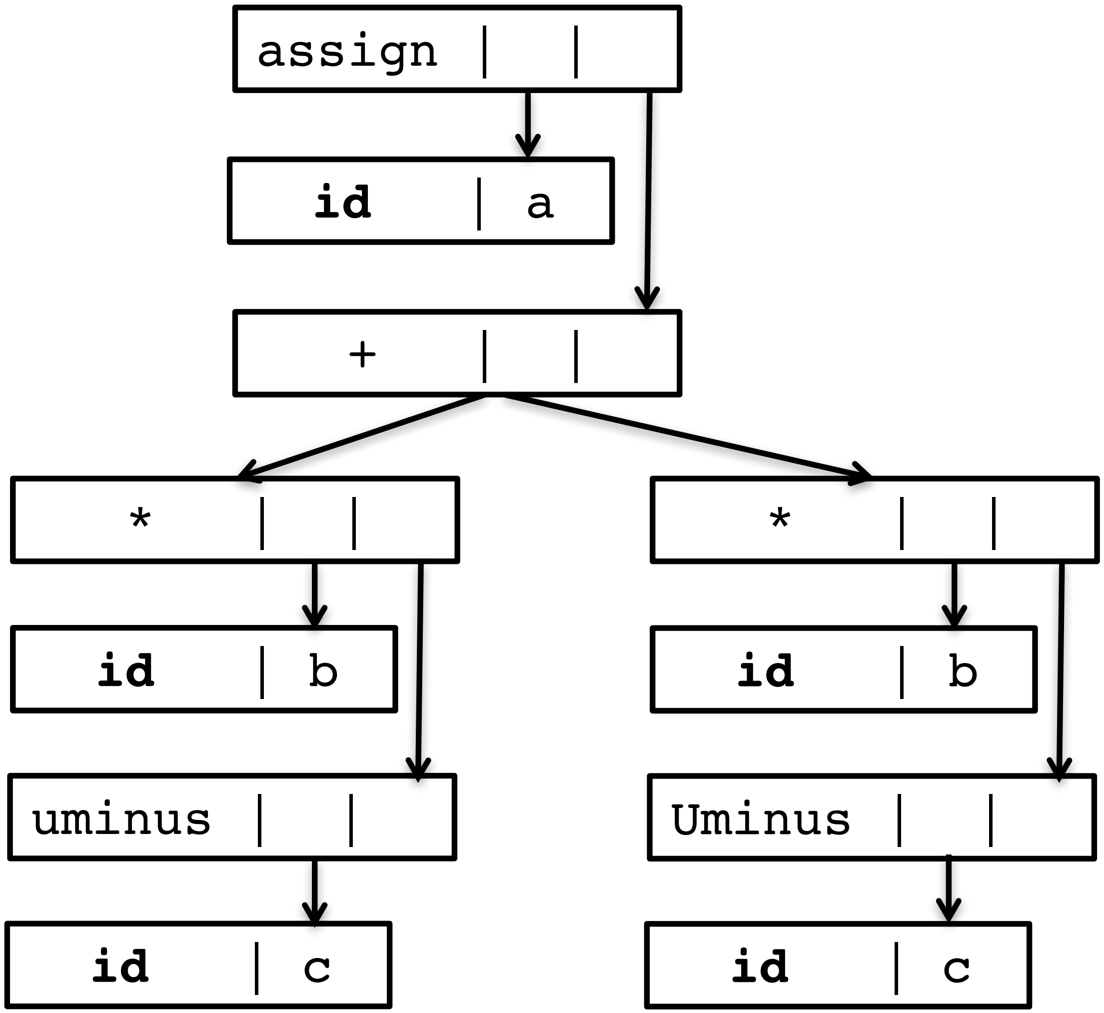


DAG

Syntax-directed Definition

PRODUCTION	SEMANTIC RULES
S → **id** := *E*	*S.nptr:=mknode('assign', mkleaf(***id***,* ***id****.place), E.nptr)*
E → E_1 + E_2	*E.nptr:=mknode('+',* E_1*.nptr,* E_2*.nptr)*
E → E_1 * E_2	*E.nptr:=mknode('-',* E_1*.nptr,* E_2*.nptr)*
E → -E_1	*E.nptr:=mkunode('uminus',* E_1*.nptr)*
E → (E_1)	*E.nptr:=*E_1*.nptr*
E → **id**	*E.nptr:=mkleaf(***id***,* ***id****.entry)*

Representation in Memory



0	id	b	
1	id	c	
2	uminus	1	
3	*	0	2
4	id	b	
5	id	c	
6	uminus	5	
7	*	4	6
8	+	3	7
9	id	a	
10	assign	9	8
11	. . .		

Postfix notation

- Linearized representation of syntax tree:

```
a:=b*-c+b*-c
a b c uminus * b c uminus * + assign
```

Three-address Code

- Generic form:

  `x := y` *op* `z`

- **One** operand at the right side of the assignment:

  Expression

  `x + y * z`

  Three-address code

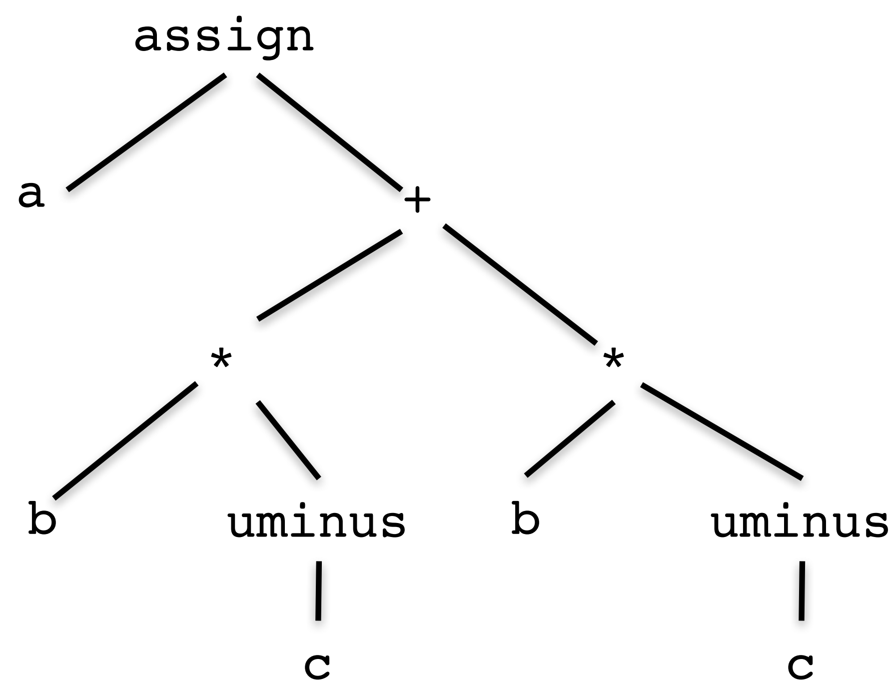
  t_1 := y * z

  t_2 := x + t_1

Example (AST)

a:=b*-c+b*-c

```
t1  := -c
t2  := b * t1
t3  := -c
t4  := b * t3
t5  := t2 + t4
a  := t5
```

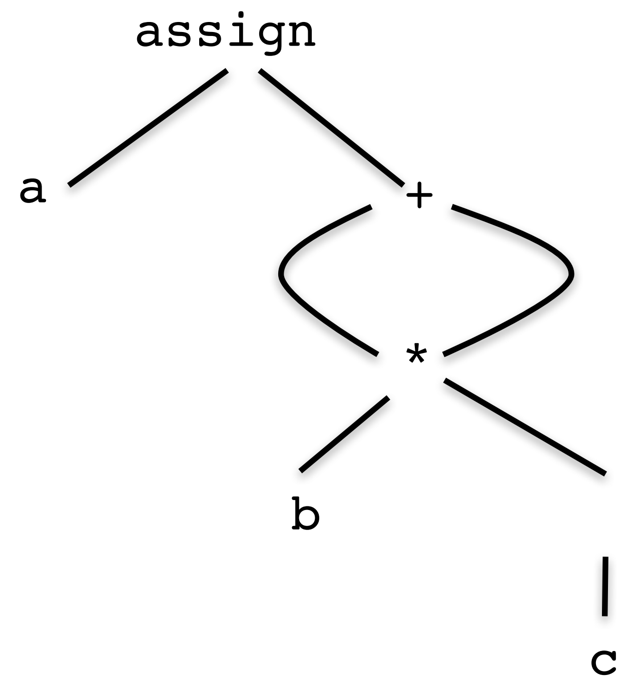


AST

Example (DAG)

a:=b*-c+b*-c

```
t1 := -c
t2 := b * t1
t5 := t2 + t2
a := t5
```



DAG

Types of Three-address Code

- Assignments
  - *op* is a binary arithmetic or logical operation

```
x := y op z
```

- Assignment instructions
  - *op* is a unary operator (minus, negation, shift, conversion)

```
x := op y
```

- Copy statements

```
x := y
```

- Unconditional jump

```
goto L
```

Types of Three-address Code

- Conditional jumps
  - *relop* is <, =, >, <=, etc.

  `if x` *`relop`* `y goto L`
- Procedure calls

  `param` x_1

  `param` x_2

  `param ...`

  `param` x_n

  `call p,` *`n`*
- Indexed assignments

  ```
  x := y[i]
  x[i] := y
  ```
- Address and pointer assignments

  ```
  x := &y
  x := *y
  ```

Terminology

Term	Description
E.place	The name that will hold the value of *E.*
E.code	The sequence of three-address statements evaluating *E.*
S.begin	Label that marks the beginning of one block.
S.after	Label that marks the end of one block and points to the following instruction.
newtemp()	Returns one temporary variable.
newlabel()	Creation of a new label.
gen()	Generation of code.

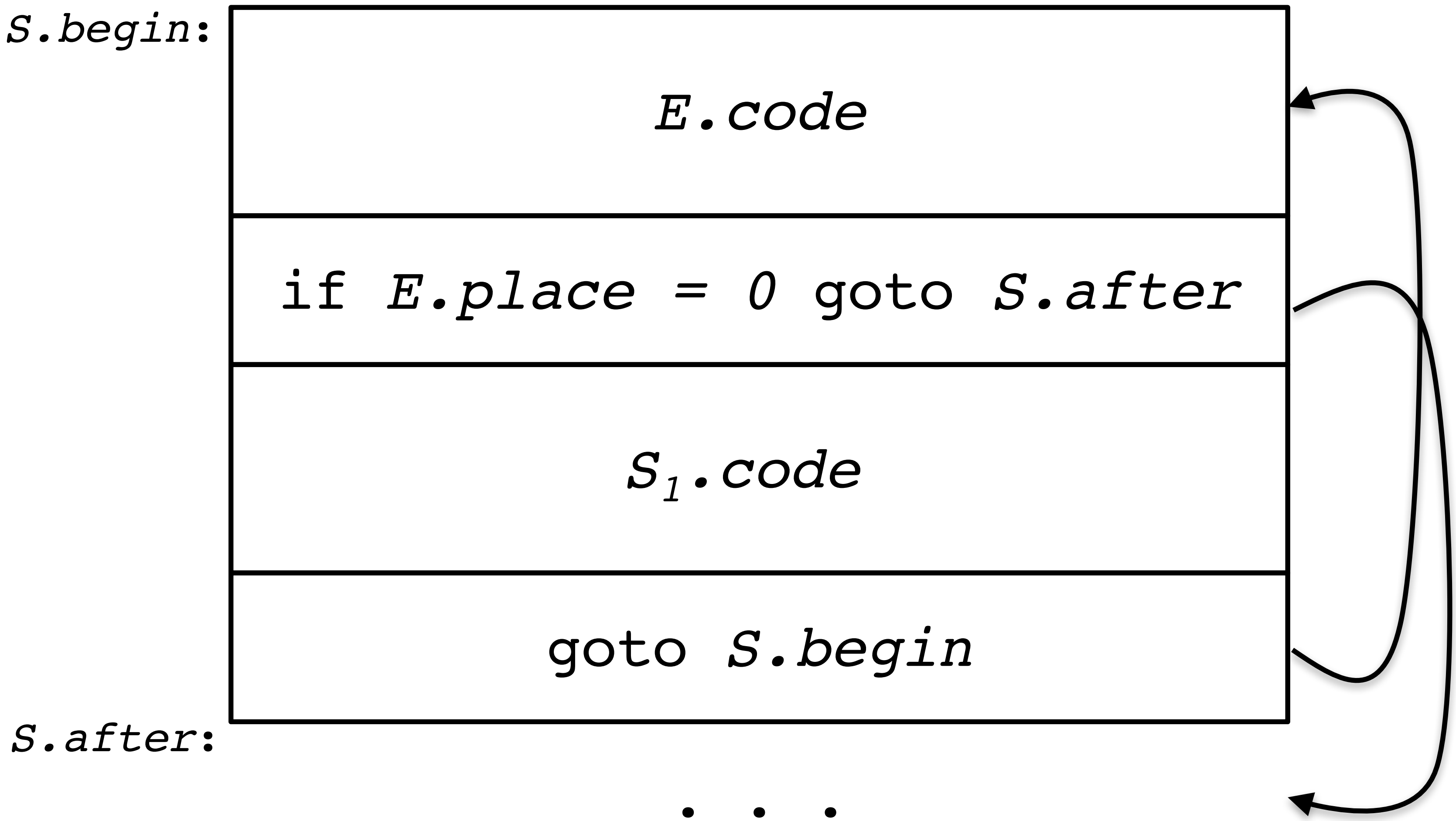
Syntax-directed Definition for Three-address Code

PRODUCTION	SEMANTIC RULES
$S \rightarrow$ **id** := E	*S.code* := *E.code* || *gen*(**id**.*place* ':=' *E.place*)
$E \rightarrow E_1 + E_2$	*E.place*:= *newtemp*; *E.code* := E_1*.code* || E_2*.code* || gen(E.place ':=' E_1*.place* '+' E_2*.place*)
$E \rightarrow E_1 * E_2$	*E.place*:= *newtemp*; *E.code* := E_1*.code* || E_2*.code* || gen(E.place ':=' E_1*.place* '*' E_2*.place*)
$E \rightarrow -E_1$	*E.place*:= *newtemp*; *E.code* := E_1*.code* || gen(E.place ':=' 'uminus' E_1*.place*)
$E \rightarrow (E_1)$	*E.place*:= E_1*.place*; *E.code* := E_1*.code*
$E \rightarrow$ **id**	*E.place*:= **id**.*place*; *E.code* := ' '

Flow Control

PRODUCTION	SEMANTIC RULES
S → **while** *E* **do** S_1	*S.begin* := *newlabel*; *S.after* := *newlabel*; *S.code* := *gen*(*S.begin* ':') || *E.code* || *gen*('if' *E.place* '=' '0' 'goto' *S.after*) || S_1*.code* || *gen*('goto' *S.begin*) || *gen*(*S.after* ':')

Flow Control



Implementation

Quadruples

	op	arg1	arg2	result
(0)	uminus	c		t_1
(1)	*	b	t_1	t_2
(2)	uminus	c		t_3
(4)	*	b	t_3	t_4
(5)	+	t_2	t_4	t_5
(6)	:=	t_5		a

Implementation

Triples

	op	arg1	arg2
(0)	uminus	c	
(1)	*	b	(0)
(2)	uminus	c	
(4)	*	b	(2)
(5)	+	(1)	(3)
(6)	assign	a	(4)

`x[i] := y`

	op	arg1	arg2
(0)	[ ]=	x	i
(1)	assign	(0)	y

`x := y[i]`

	op	arg1	arg2
(0)	[ ]=	y	i
(1)	assign	x	(0)

Implementation

Indirect Triples

	statement
(0)	(14)
(1)	(15)
(2)	(16)
(4)	(17)
(5)	(18)
(6)	(19)

	op	arg1	arg2
(0)	uminus	c	
(1)	*	b	(14)
(2)	uminus	c	
(4)	*	b	(16)
(5)	+	(15)	(17)
(6)	assign	a	(18)