COMP 520 - Compilers

Lecture 08 – Shift/Reduce and Identification/Type Checking

WA1 Clarification on Choice / Choice Points

- `a|b|c` has choice points for one decision (which is frustratingly also called “one choice”)

- I don’t like this notation, so I removed the “how many choices” question.
- Word “choice” feels overloaded.

WA2 Released Today

- Due in 10 days on the 19th (Monday)
- Note: An arbitrary sequence can be any sequence!

- Reminder: Midterm on the 22nd (Thursday)

Monday, Tuesday (2/12-2/13) Wellness Day

- No office hours, we will be busy working on wellness

Stratified Grammars and left-to-right Order

- Question: Are same-precedence operations in a stratified grammar processed in left-to-right order?
  - By processed, I mean when emulated like in the Lec 06, Slide 41.

- You can practice this in WA2!

Short Recap on Left-Factorization

- Recall the fearsome four in PA1:

Type id = Expression ;

Reference = Expression ;

Reference [Expression] = Expression ;

Reference (ArgumentList?) ;

Break up each rule using literals

Type id = Expression ;	(boolean|(int|id)([])?) id = …
Reference = Expression ;	(id|this)(.id)* = …
Reference [Expression] = Expression ;	(id|this)(.id)* [Expression] = …
Reference (ArgumentList?) ;	(id|this)(.id)* (ArgumentList?) ;

Left Factorization

- Can be viewed as a deterministic state machine
- Each edge is an "if" statement
- Each node is an accept/parse

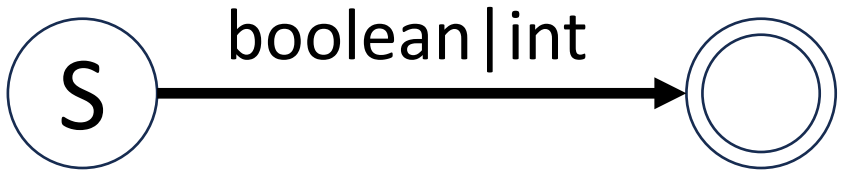
Rules of the form A ::= $\alpha\beta_1 \mid \alpha\beta_2$

Rewritten as:

A ::= αA'

A' ::= $\beta_1 \mid \beta_2$ (and vice versa)

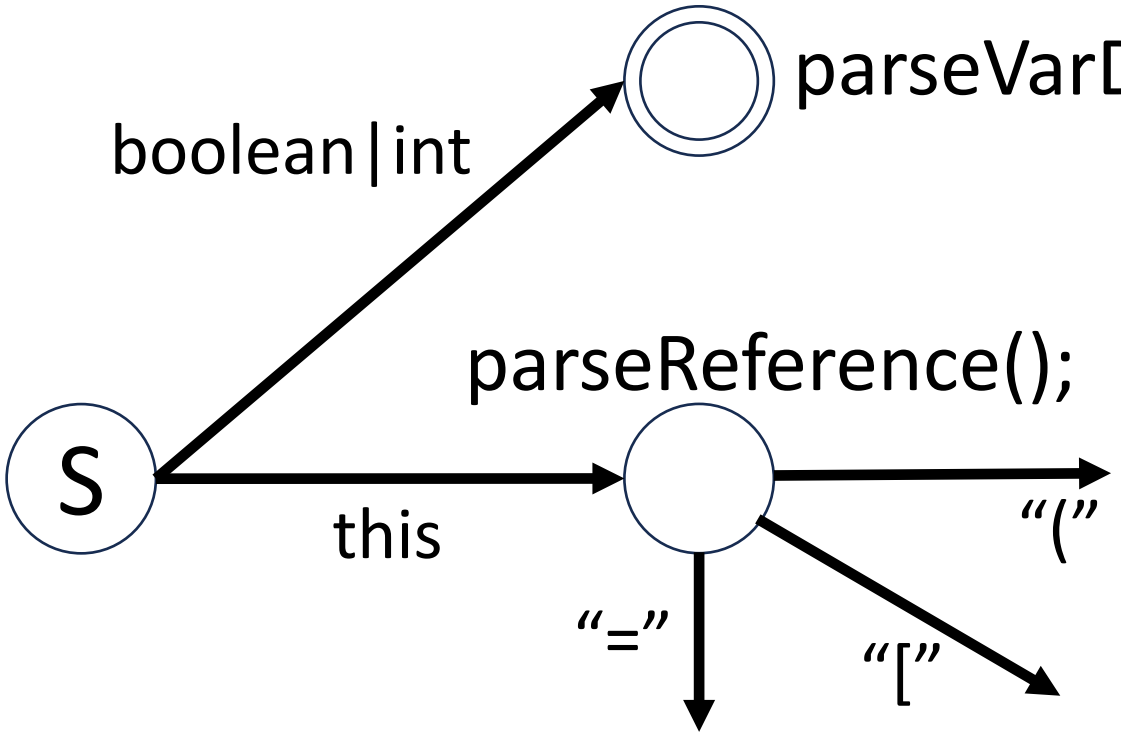
Create a State Machine



parseVarDeclStmt();

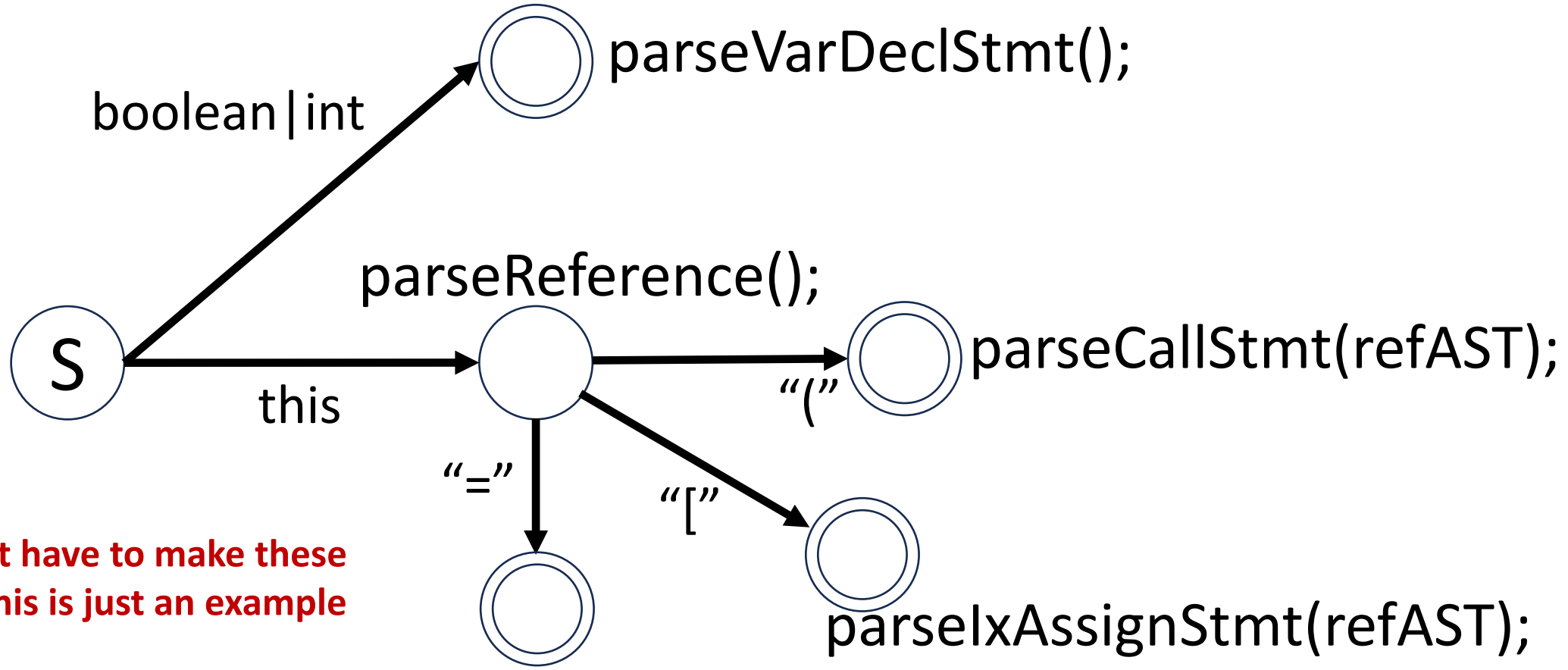
Note: I am using a short-hand notation for "parse the rest"

Create a State Machine (2)



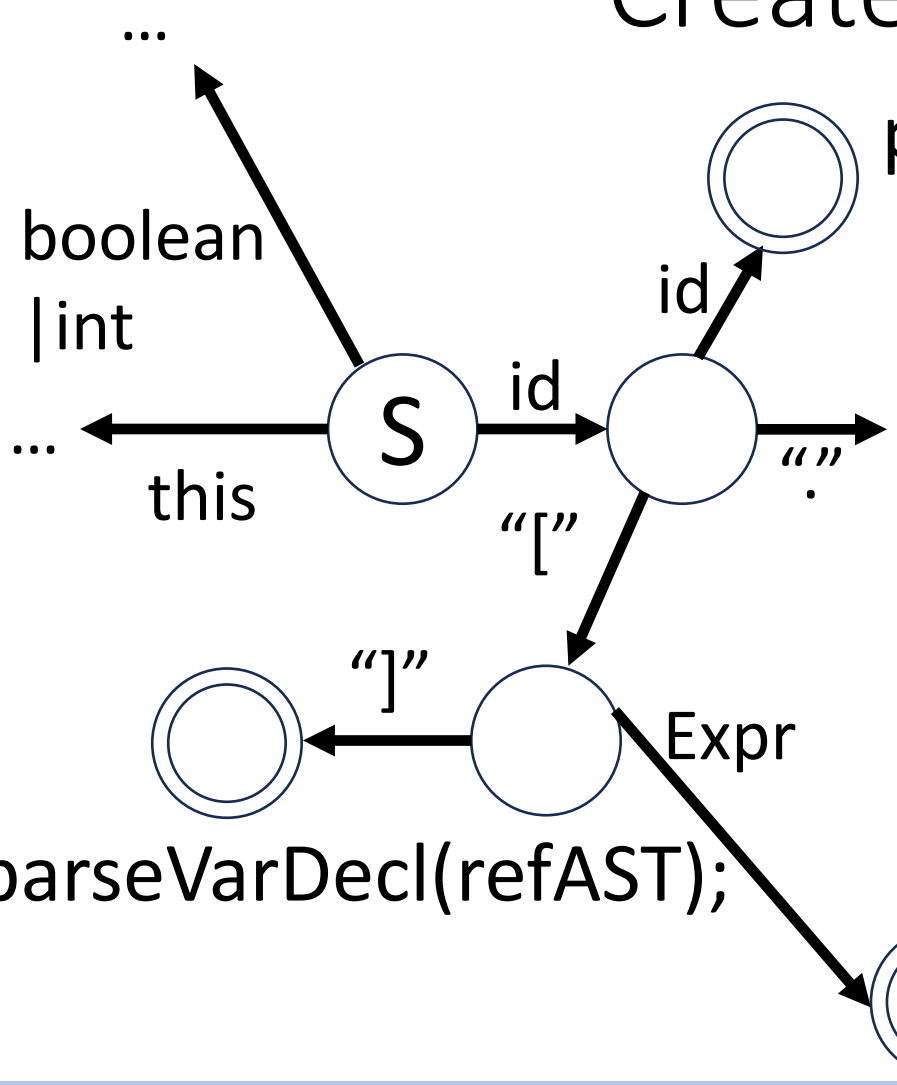
Note: I am using a short-hand notation for "parse the rest"

Create a State Machine (3)



***Note, you do not have to make these parse methods, this is just an example**

Create a State Machine (4)

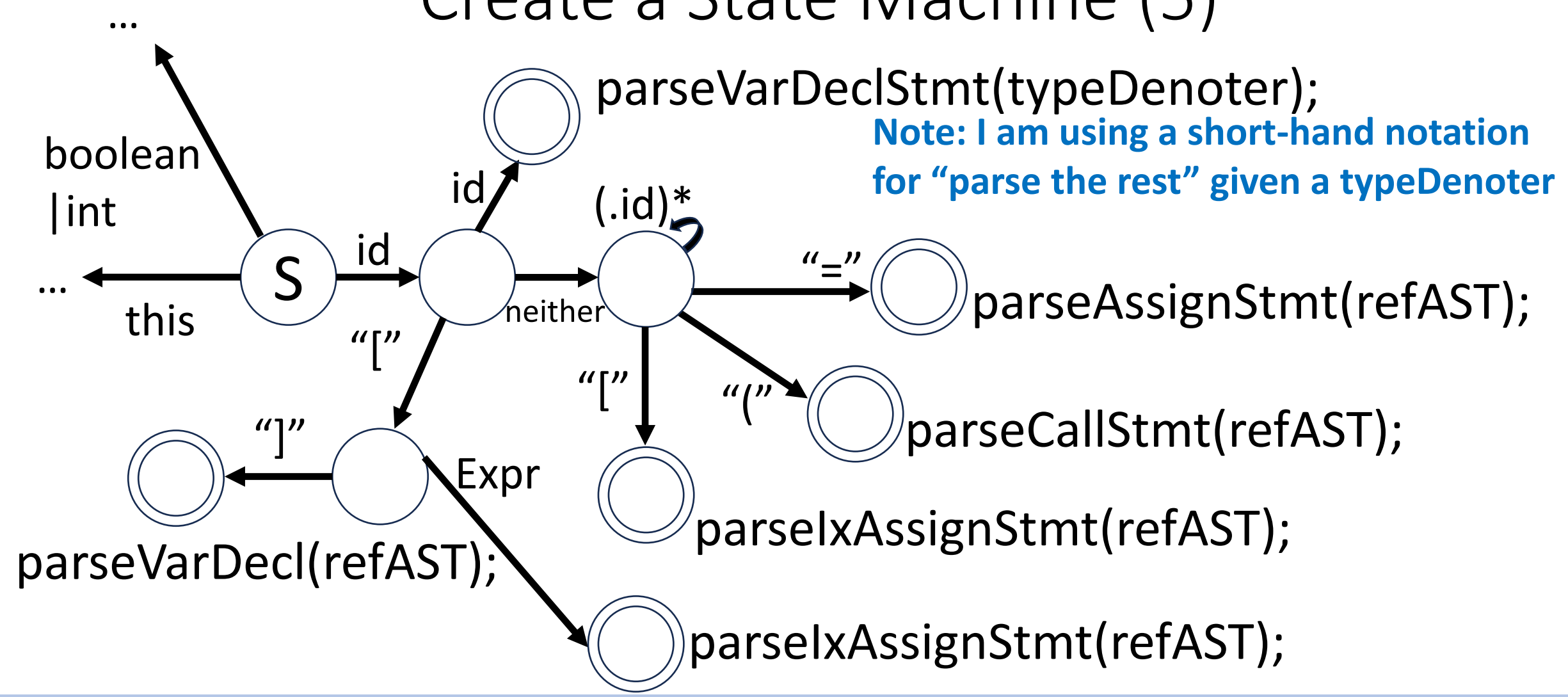


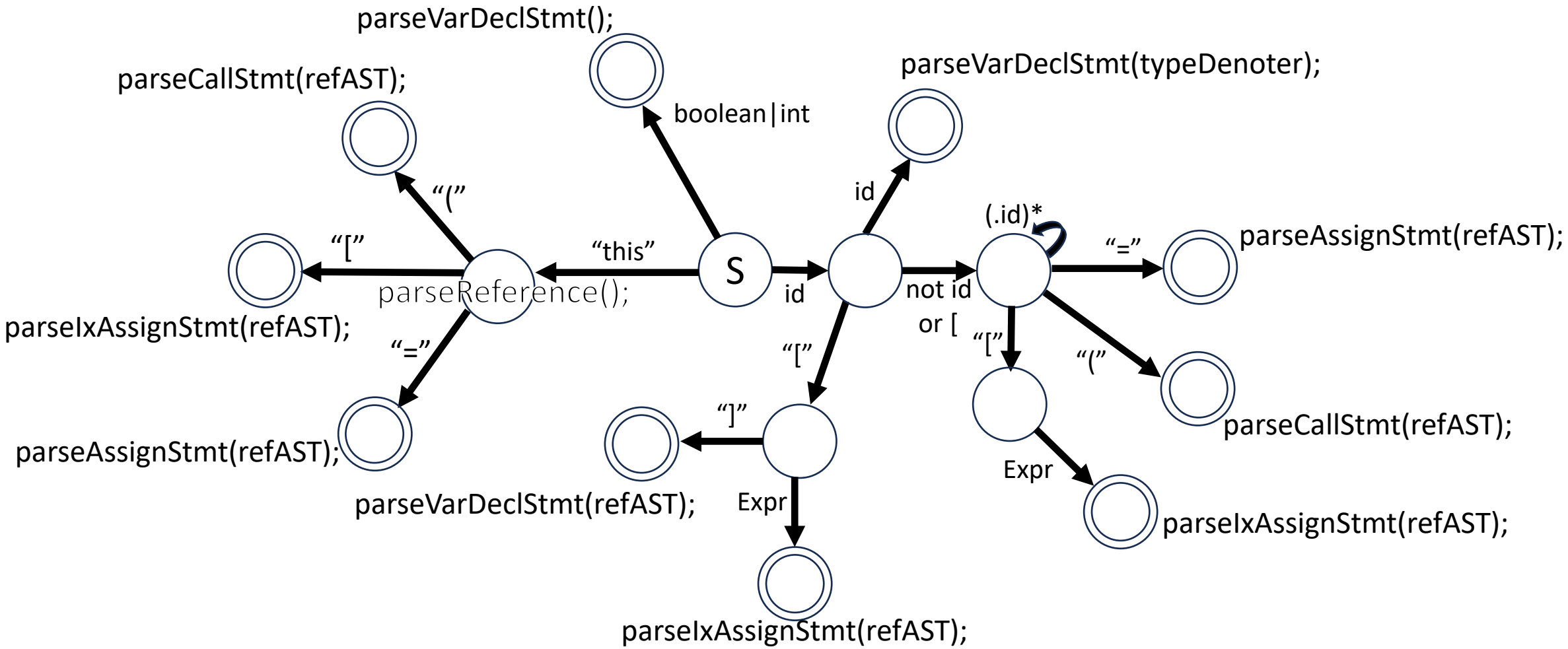
parseVarDeclStmt(typeDenoter);

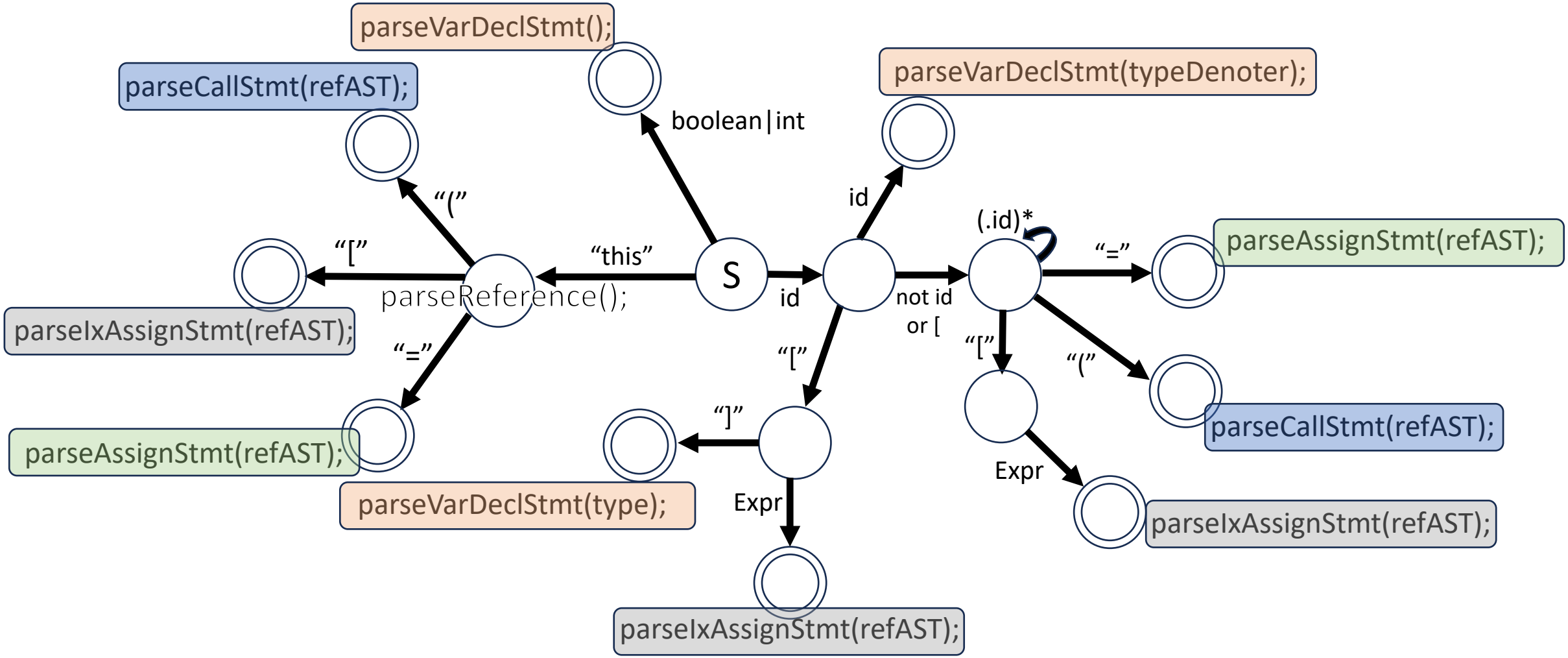
Note: I am using a short-hand notation for “parse the rest” given a typeDenoter

parseIxAssignStmt(refAST);

Create a State Machine (5)




parseVarDeclStmt();
parseCallStmt(refAST);
boolean|int
parseVarDeclStmt(typeDenoter);
"("
id
(.id)*
"["
"this"
S
parseReference();
id
not id
or [
"="
parseAssignStmt(refAST);
parseIxAssignStmt(refAST);
"="
"["
"["
"("
"]"
parseAssignStmt(refAST);
parseCallStmt(refAST);
Expr
parseVarDeclStmt(refAST);
Expr
parseIxAssignStmt(refAST);
parseIxAssignStmt(refAST);


parseVarDeclStmt();
parseCallStmt(refAST);
boolean|int
parseVarDeclStmt(typeDenoter);
"("
id
(.id)*
"["
"this"
S
id
"="
parseAssignStmt(refAST);
parseIxAssignStmt(refAST);
parseReference();
not id
or [
"="
"["
"("
"]"
parseAssignStmt(refAST);
parseCallStmt(refAST);
parseVarDeclStmt(type);
Expr
Expr
parseIxAssignStmt(refAST);
parseIxAssignStmt(refAST);

Original Fearsome Four

- F ::= Type `id` `=` Expression `;`
  - | Reference `=` Expression `;`
  - | Reference `[` Expression `]` `=` Expression `;`
  - | Reference `(` ArgumentList? `)` `;`

Ignore the other types of Statements and only look at these four (assume no other rule for F exists).

New Grammar

```
  this(.id)*(= Expr ; | [Expr] = Expr ; | ( ArgList? ) ; )
| boolean id = Expr ;
| id (
      ((.id)*( = Expr ; | [ Expr ] = Expr ; | ( ArgList? ) ; ))
      | [] id = Expr ;
  )
```

Is it LL(1)?

Let’s informally investigate

Eyeball Starters

```
this(.id)*(= Expr ; | [Expr] = Expr ; | ( ArgList? ) ; )
| boolean id = Expr ;
| id (
      ((.id)*( = Expr ; | [ Expr ] = Expr ; | ( ArgList? ) ; ))
    | [] id = Expr ;
  )
```

So far so good

Eyeball Starters (2)

this(.id)*(= Expr ; | [Expr] = Expr ; | (ArgList?) ;)

Question: are the predict sets in this disjoint?

Eyeball Starters (3)

this (. id)* (= Expr ; | [Expr] = Expr ; | (ArgList?) ;)

Question: are the predict sets in this disjoint?

(Note: Nullable((. id)*) = *true*)

Eyeball it, looks like we have: . = [(

Eyeball Starters (4)

Eyeball it again with starters: what are the starters?

```
| id (
      ( (. id)*( = Expr ; | [ Expr ] = Expr ; | ( ArgList? ) ; ) )
      | [] id = Expr ;
)
```

Eyeball Starters (5)

Eyeball it again with starters: `.` `=` `[` `(`

```
| id (
      ( (. id)*( = Expr ; | [ Expr ] = Expr ; | ( ArgList? ) ; ) )
      | [] id = Expr ;
)
```

Any problems here?

Fix the problem in the Scanner

Eyeball it again with starters: . = [([]

```
| id (
      ( (. id)* ( = Expr ; | [ Expr ] = Expr ; | ( ArgList? ) ; ))
    | [] id = Expr ;
    )
```

Make this a token by itself.
Will not affect the scanning of other tokens.

Final Grammar (again)

```
  this(.id)*(= Expr ; | [Expr] = Expr ; | ( ArgList? ) ; )
| boolean id = Expr ;
| id ( ((.id)*( = Expr ; | [ Expr ] = Expr ; | ( ArgList? ) ; ))
       | [] id = Expr ; )
```

Is it LL(1) now?

In other words, by only looking at the next token, can I deterministically make decisions?

Usefulness of Grammar Transformations

- Fearsome four made a lot easier through our Compiler CFG theory lectures! But automatically generating a Parser still has areas for optimization. Note the heavily reused sequences:

```
this(.id)*(= Expr ; | [Expr] = Expr ; | ( ArgList? ) ; )
| boolean id = Expr ;
| id ((.id)*( = Expr ; | [ Expr ] = Expr ; | ( ArgList? ) ; ))
      | [] id = Expr ; )
```

Bottom-Up Parsing

Shift-Reduce parsing

Bottom-Up Parsing

- Also known as a shift-reduce parser
- Mechanism is:
  - Stack for your state
  - Production Rules (your CFG)
  - State Table (State x Token) → (Action, NextState/ProdRule)
  - Non-Terminal Table (State x Non-Terminal) → State
- Once your data structures are **correctly** set up, it works pretty much by itself.

Current State = Top of Stack, then either…

Shift(nextState,Token)

- Push state onto stack
- Push token onto stack
- Get next input

- Note, state and token may be implemented with separate stacks in some implementations of shift-reduce

Reduce(prodNum)

- For the length of the production, pop state and tokens
- Push that production rule's non-terminal to the stack
- Push the next state depending on the Non-terminal x state table

Bottom-Up (Shift-Reduce) Example

Production Rules (CFG)

- E ::= E + T
- E ::= T
- T ::= T * F
- T ::= F
- F ::= (E)
- F ::= id

State, Non-terminal, and Prod Table

State x Token -> (Shift/Reduce, State/ProdNum)

State	id	+	*	(	)	$
0	s5			s4		
1		s6				stop
2		r2	s7		r2	r2
3		r4	r4		r4	r4
4	s5			s4		
5		r6	r6		r6	r6
6	s5			s4		
7	s5			s4		
8		s6			s11	
9		r1	s7		r1	r1
10		r3	r3		r3	r3
11		r5	r5		r5	r5

State x NT -> Next State

State	E	T	F
0	1	2	3
1			
2			
3			
4	8	2	3
5			
6		9	3
7			10
8			
9			
10			
11			

Production Rules

R#	Rule
1	E ::= E + T
2	E ::= T
3	T ::= T * F
4	T ::= F
5	F ::= (E)
6	F ::= id

E ::= E + T
E ::= T
T ::= T * F
T ::= F
F ::= (E)
F ::= id

Consider the input w:

w= a + b * c $

Initialize state stack: { 0 }

Initialize symbol stack: { }

E ::= E + T
E ::= T
T ::= T * F
T ::= F
F ::= (E)
F ::= id

Consider the input w:

w= a + b * c $

Current State = top(StateStack) = 0

Current Input = id

Symbol Stack = {}

State0 x id = s5 (Shift)

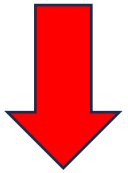
State	id	+	*	(	)	$
0	s5			s4		

Thus, push “id” on symbol stack

push state 5 on state stack, get next input

E ::= E + T
E ::= T
T ::= T * F
T ::= F
F ::= (E)
F ::= id

Consider the input w:



w= a + b * c $

State	id	+	*	(	)	$
5		r6	r6		r6	r6

Prod	Rule
6	F ::= id

State	E	T	F
0	1	2	3

StateStack= { 5, 0 }

SymbolStack= { a }

r6 -> Reduce on prod # 6

E ::= E + T
E ::= T
T ::= T * F
T ::= F
F ::= (E)
F ::= id

Consider the input w:

w= a + b * c $

State	id	+	*	(	)	$
5		r6	r6		r6	r6

Prod	Rule
6	F ::= id

State	E	T	F
0	1	2	3

StateStack= { 5, 0 }
SymbolStack= { a }
r6 -> Reduce on prod # 6

- Prod length is 1, so pop both stacks once
- In [new] state 0, prod rule was “F”, so push “F” and next state of 3

E ::= E + T
E ::= T
T ::= T * F
T ::= F
F ::= (E)
F ::= id

Consider the input w:

w= a + b * c $

State	id	+	*	(	)	$
3		r4	r4		r4	r4

Prod	Rule
4	T ::= F

State	E	T	F
0	1	2	3

StateStack= { 3, 0 }

SymbolStack= { F }

-> r4 (Reduction on Prod 4)

(pop once on both, push T, push new state 2)

E ::= E + T
E ::= T
T ::= T * F
T ::= F
F ::= (E)
F ::= id

Consider the input w:

w= a + b * c $

State	id	+	*	(	)	$
2		r2	s7		r2	r2

Prod	Rule
2	E ::= T

State	E	T	F
0	1	2	3

StateStack= { 2, 0 }

SymbolStack= { T }

-> r2 (Reduction on Prod 2)

(pop once on both, push E, push new state 1)

E ::= E + T
E ::= T
T ::= T * F
T ::= F
F ::= (E)
F ::= id

Consider the input w:

w= a + b * c $

State	id	+	*	(	)	$
1		s6				stop

StateStack= { 1, 0 }

SymbolStack= { E }

-> s6 (Shift, next state 6)

(push '+', read next input, push state 6)

E ::= E + T
E ::= T
T ::= T * F
T ::= F
F ::= (E)
F ::= id

Consider the input w:

w= a + b * c $

State	id	+	*	(	)	$
6	s5			s4		

StateStack= { 6, 1, 0 }

SymbolStack= { +, E }

-> s5 (Shift, next state 5)

(push ‘id’, read next input, push state 5)

E ::= E + T
E ::= T
T ::= T * F
T ::= F
F ::= (E)
F ::= id

Consider the input w:

w= a + b * c $

State	id	+	*	(	)	$
5		r6	r6		r6	r6

Prod	Rule
6	F ::= id

State	E	T	F
6		9	3

StateStack= { 5, 6, 1, 0 }

SymbolStack= { id, +, E }

-> r6 (Reduction on Prod 6)

(pop one from both stacks, push F, push 3)

Skip ahead a bit

- (Change the id on the stack to T)

E ::= E + T
E ::= T
T ::= T * F
T ::= F
F ::= (E)
F ::= id

Consider the input w:

w= a + b * c $

State	id	+	*	(	)	$
9		r1	s7		r1	r1

StateStack= { 9, 6, 1, 0 }

SymbolStack= { T, +, E }

-> s7 (Shift, push next state 7)

E ::= E + T
E ::= T
T ::= T * F
T ::= F
F ::= (E)
F ::= id

Consider the input w:

w= a + b * c $

State	id	+	*	(	)	$
7	s5			s4		

StateStack= { 7, 9, 6, 1, 0 }

SymbolStack= { *, T, +, E }

-> s5 (Shift, push next state 5)

E ::= E + T
E ::= T
T ::= T * F
T ::= F
F ::= (E)
F ::= id

Consider the input w:

w= a + b * c $

State	id	+	*	(	)	$
5		r6	r6		r6	r6

Prod	Rule
6	F ::= id

State	E	T	F
6		9	3

StateStack= { 5, 7, 9, 6, 1, 0 }

SymbolStack= { id, *, T, +, E }

-> r6

Remainder

E ::= E + T
E ::= T
T ::= T * F
T ::= F
F ::= (E)
F ::= id

w= a + b * c $

SymbolStack= { id, *, T, +, E }

What happens next:

➔ F * T + E

➔ T + E

➔ E

Skip more of "change id to NT"

Out-of-precedence-order?

- Consider: w = a * b + c $

Note how the multiply (higher precedence) will combine into T before the lower precedence addition.

Step	Symbol Stack
0	id
1	F
2	T
3	* T
4	id * T
5	F * T
6	T
7	E
8	+ E
9	id + E
	…
11	T + E
12	E

Example of Shift-Reduce Parser on Website

- Example is in C++
- Once the table is built for the grammar, the parsing code remains the same.

- Note: the switch statement for REDUCE can be changed to a table as well (with prod lengths and what NT is pushed)

Reduce-Reduce conflicts

- General idea:

A ::= α

B ::= α

And top of the stack contains α

Do we reduce with prod A or prod B?

Reduce-Reduce Conflicts (From gnu.org)

```
sequence:
        %empty                  { printf("empty seq"); }
    |   maybeword
    |   sequence word           { printf("added word %s", $2); }
maybeword:
        %empty                  { printf("empty maybeword"); }
    |   word                    { printf("single word %s"); }
```

Another common conflict

Stmt ::= `if (`Expr`)` Stmt

| `if (`Expr`)` Stmt `else` Stmt

If we see the input "`if (`Expr`)` Stmt" on the stack, and the next symbol is "`else`", do we reduce or shift?

Consider: if(true) if(true) else a = b;

Identification

Three scopes in miniJava

- Class Names
- Member Names (fields and methods)
- Parameters / Local variables

miniJava Context

- Give each a “level” and create some rules
- Level 0: Class Names
- Level 1: Member Names
- Level 2+: Method Parameters & Local Variables

miniJava Context

- Give each a “level” and create some rules
- Level 0: Class Names
- Level 1: Member Names
- Level 2+: Method Parameters & Local Variables

How does this work in Java?

A perfectly sane Java program

```
1 public class MainMethod {
2     int MainMethod = 3;
3
4     public static void main(String[] args) {
5         String MainMethod = "hello";
6         System.out.println(MainMethod);
7     }
8 }
```

miniJava Context

- Give each scope a “level” and create some rules
- Level 0: Class Names
- Level 1: Member Names
- Level 2+: Method Parameters & Local Variables

Anything in the 2+ category cannot override anything else in 2+. Other than that, higher scope levels can override lower scope levels.

Revisit Identifier

- An identifier has a name and a context
- An identifier can be…

Identifiers

Category	Definition
Local Variable (or Parameter)	Defined locally for that method
Method	Identifier is for a method
Type	Identifier is for a class type
Class Name	Identifier used for statics or new expressions. e.g. CLASSNAME.x or new CLASSNAME()
Field Variable	Identifier is specific to a dynamic object (a.b)
Namespace	C++, useful for resolving conflicts, packages everything inside the namespace and can be renamed.
Pre-defined	E.g. System.out.println

Identifiers make sense in context

- Identifier x Context → Declaration
- When no given context, context is the current class

```
public class MainMethod {

    private int x = 0;

    public void fun(String[] args) {
        if( true ) {
            System.out.printf("%d", x);
        }
    }
}
```

Not yet declared??

Note: B is defined after A

Java is doing something special that is not "in-order."

You will have to tackle this problem too!

```
class A {
    B b;
    int x;
}

class B {
    C c;
    int x;
}

class C {
    int x = 2;
    void fun() {
        int b = 3;
        int c = 4;
        int x = 5;

        A a = new A();
        a.b.c.x = 6;
    }
}
```

Where is a.b.c.x's declaration?

x is in the context of c,

c is in the context of b

b is in the context of a

a was declared as an "A" object

```
class A {
    B b;
    int x;
}

class B {
    C c;
    int x;
}

class C {
    int x = 2;
    void fun() {
        int b = 3;
        int c = 4;
        int x = 5;

        A a = new A();
        a.b.c.x = 6;
    }
}
```

Where is a.b.c.x's declaration?

x is in the context of c,
c is in the context of b
b is in the context of a
a was declared as an "A" object

a:	a is of type "A"
a.b:	b in context of "A", type is "B"
a.b.c:	c in context of "B", type is "C"
a.b.c.x:	x in context of "C"

```
class A {
    B b;
    int x;
}

class B {
    C c;
    int x;
}

class C {
    int x = 2;
    void fun() {
        int b = 3;
        int c = 4;
        int x = 5;

        A a = new A();
        a.b.c.x = 6;
    }
}
```

Where is x's declaration?

a: a is of type "A"

a.b: b in context of "A", type is "B"

a.b.c: c in context of "B", type is "C"

a.b.c.x: x in context of "C"

∴ QualRef "a.b.c.x" is defined here

{"ID x"} x {"ctx C"} → FieldDecl

```
class A {
    B b;
    int x;
}

class B {
    C c;
    int x;
}

class C {
    int x = 2;
    void fun() {
        int b = 3;
        int c = 4;
        int x = 5;

        A a = new A();
        a.b.c.x = 6;
    }
}
```

Identification Theory

- As identifiers are declared (ClassDecl, VarDecl, etc.)… keep a table of their context.

- IDTable: (id × ctx) → Declaration
- Input is (id,ctx), output is Declaration

(Note: if (a,b) ∉ IDTableInputs, then id error)

- But what **exactly** is a context?

Context Visibility

- The context of an identifier is the class it is defined in.
- Not all entries are available at all times.

```
class A {
    B b;
    int x;
}

class B {
    private C c;
    int x;
}

class C {
    int x = 2;
    void fun() {
        A a = new A();
        a.b.c.x = 6;
    }
}
```

Context Visibility

- The context of an identifier is the class it is defined in.
- Not all entries are available at all times.
- When resolving a.b.c, the entry: {“ID c”} x {“CTX B”} → FieldDecl is unavailable when inside C.fun

```
class A {
    B b;
    int x;
}

class B {
    private C c;
    int x;
}

class C {
    int x = 2;
    void fun() {
        A a = new A();
        a.b.c.x = 6;
    }
}
```

Context Invisibility

- Some languages let you manually specify context.
- Shown on the right is C++.

- Can use language's syntax to bypass hidden contexts (in this case, local variable hides field variable)

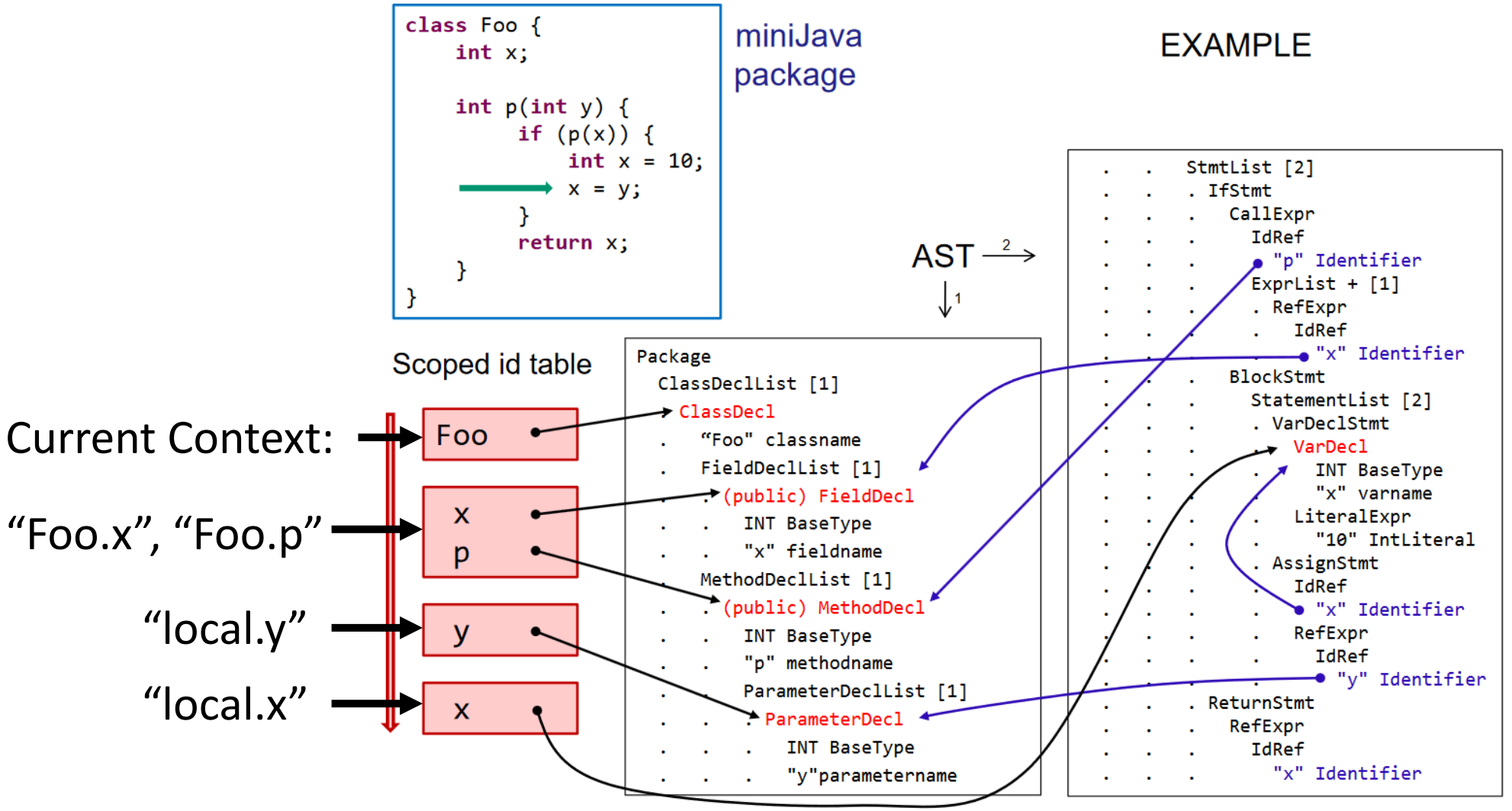
```
class A {
public:
    int x;

    void fun() {
        int x = 1;
        A::x = 2;
        x = 3;
    }
};
```

Scoped Identification

- Your ID Table can be seen as a stack.

- When a new scope occurs (with a block statement{…}) then you get a new ID table pushed on the stack.
- Other than for QualRef, check the top-most ID table, and work your way down to find the Declaration

Example (1) – Jan Prins



Example (2) – Jan Prins

```
class Foo {
    int x;

    int p(int y) {
        if (p(x)) {
            int x = 10;
            x = y;
        }
        return x;
    }
}
```

miniJava package

EXAMPLE

AST

```
Package
  ClassDeclList [1]
  . ClassDecl
  .   "Foo" classname
  .   FieldDeclList [1]
  .   . (public) FieldDecl
  .   .   INT BaseType
  .   .   "x" fieldname
  .   MethodDeclList [1]
  .   . (public) MethodDecl
  .   .   INT BaseType
  .   .   "p" methodname
  .   .   ParameterDeclList [1]
  .   .   . ParameterDecl
  .   .   .   INT BaseType
  .   .   .   "y"parametername
```

```
. . StmtList [2]
. . . IfStmt
. . .   CallExpr
. . .     IdRef
. . .       "p" Identifier
. . .     ExprList + [1]
. . .     . RefExpr
. . .     .   IdRef
. . .     .     "x" Identifier
. . .   BlockStmt
. . .     StatementList [2]
. . .     . VarDeclStmt
. . .     .   VarDecl
. . .     .     INT BaseType
. . .     .     "x" varname
. . .     .   LiteralExpr
. . .     .     "10" IntLiteral
. . .     . AssignStmt
. . .     .   IdRef
. . .     .     "x" Identifier
. . .     .   RefExpr
. . .     .     IdRef
. . .     .       "y" Identifier
. . . ReturnStmt
. . .   RefExpr
. . .     IdRef
. . .       "x" Identifier
```

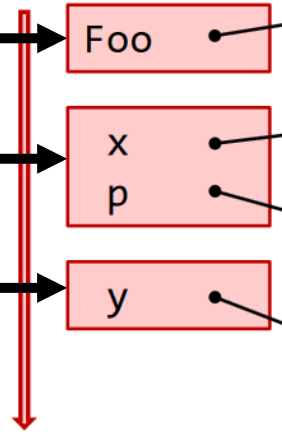
Scoped id table

Current Context:

"Foo.x", "Foo.p"

"local.y"

No more "local.x"



Identification Traversal

- At each node, we can update the ID table.
- Updates can be to a global ID table, or you can return an updated table while traversing.

Update may be to hide identifiers that should now be invisible (private variables, or hiding level 0 and 1 with a new 2+ variable).

Identification Traversal

- At each node, we can update the ID table.
- Updates can be to a global ID table, or you can return an updated table while traversing.

- Update may be to hide identifiers.

Question: Why is a global table not desirable?
(Think about –j flag on gcc)

Type Checking

Type-Checking Table

- For every type, should there be an entry where…

$$\alpha \times \beta \rightarrow \gamma$$

E.g., int × int → int

- Is this enough?
- Will every int op int yield an int?

Type-Checking Table (2)

- For every type, should there be an entry where…

$$\alpha \times \beta \rightarrow \gamma$$

E.g., int × int → int

- What about 3 == 2?
  In that case, int × int → boolean

Type-Checking Table (3)

Therefore, Type-Checking is a 3 input, 1 output table:

$$(\beta \times \alpha \times \text{op}) \rightarrow \gamma$$

E.g., int × int × {+,−,*,/} → int

int × int × {==,!=,>,>=,<,<=} → boolean

Operator Overloading

- Operator Overloading: give custom definitions to ops

- Example, consider a "MyData" object that can accumulate integers and sorts them.
- In C++, can create **a method** that defines:

$$\text{MyData} \times \text{int} \times \{+\} \rightarrow \text{MyData}$$

Where adding integers to MyData types will call **that method**.

Bad Types

- When an entry is missing, then the resultant type should be a specially designated type that is compatible with EVERY type.

- This type, "**ErrorType**", prevents type-checking errors from filling up your compiler's error log.

Bad Types (2)

- This type, "**ErrorType**", prevents type-checking errors from filling up your compiler's error log.
- Consider miniJava: (myObj + 3) + 5;

ClassType × int × {+} → ErrorType (**report error here**)

ErrorType × int × {+} → ErrorType (no need to report)

Second type check was valid.

Bad Types (3)

- Generalization:

$(\{\text{ErrorType}\} \times \text{AnyType} \times \text{AnyOp}) \rightarrow \text{ErrorType}$

$(\text{AnyType} \times \{\text{ErrorType}\} \times \text{AnyOp}) \rightarrow \text{ErrorType}$

$((\text{TypeA},\text{TypeB},\text{Op}) \notin \text{TypeTable}) \rightarrow (\text{Result}=\text{ErrorType})$

$((\text{TypeA},\text{TypeB},\text{Op}) \in \text{TypeTable}) \rightarrow$

$(\text{Result} = \text{GetEntry}(\text{TypeA},\text{TypeB},\text{Op}))$

Does Order Matter?

- Is $A \times B$ the same as $B \times A$?

Does Order Matter?

- Is $A \times B$ the same as $B \times A$?
- Previous definitions state yes.

- Your thoughts?

Does Order Matter?

- Is $A \times B$ the same as $B \times A$?
- Previous definitions state yes.

- Not for miniJava, but yes for some languages.
- Why?

Order – Language Specifications

- Language may specify that order can matter
- C++ can specify order in the program itself:

More Totally Sane Code

- Why?

```
class A {
public:
    int x;
    A(int x) : x(x) {}
};

A operator+(const A& a, int b) {
    return A(a.x + b);
}

A operator+(int a, const A& b) {
    return A(b.x * a);
}

void main() {
    A a(5);
    A b = a + 3;
    A c = 3 + a;
    printf("%d %d %d\n", a.x, b.x, c.x);
}
```

More Totally Sane Code

- Why?
- You could make your compiler “keep” the first type if you wanted.
- For example: 3 + 3.2 * 4.4;
- Specify: int × float … → int
- Specify: float × int … → float

```
class A {
public:
    int x;
    A(int x) : x(x) {}
};

A operator+(const A& a, int b) {
    return A(a.x + b);
}

A operator+(int a, const A& b) {
    return A(b.x * a);
}

void main() {
    A a(5);
    A b = a + 3;
    A c = 3 + a;
    printf("%d %d %d\n", a.x, b.x, c.x);
}
```

Type-Checking Table

- Thus, how you construct your type checking table depends on the language.

TypeCasting Tables

TypeCasting / Type Conversions

- Typecasting (or the conversion of a type) does not have an operator.
- Formally:

$$\alpha \times \beta \rightarrow \alpha$$

- Thus, the table for typecasting is: (TypeA,TypeB) → TypeA, and some skip the → entirely
- E.g. ((int)3.4f) → 3 (an int), which is (int,float)→int

TypeCasting / Type Conversions (2)

$$\alpha \times \beta \rightarrow \alpha$$

- The table for typecasting takes an input of (TypeA,TypeB), result is just TypeA.

What is the exception?

- If (x,y) ∉ Table, then this is a type checking error.
- The table usually has a result just to make the code easier to write up.

Automatic Conversions

- boolean is just a 0 or 1, so we could:

int × boolean → int

boolean × int → boolean

- This is also described as:

int → {int, boolean}

(depends on how you construct the table)

Automatic Conversions (2)

- Java has more automatic conversions:

String × Enum → String

- The Enum will automatically become a string that represents the text of that enum entry.

Automatic Conversions (3)

- Automatic conversions are AUTOMATIC
- Note:

```
private int somefun( int a ) { … }
…
somefun( true );
```

… does not require explicitly typecasting `(int)true`

Automatic Conversions (4)

- Java has more automatic conversions:

String × Enum → String

- Combine this with: String × String × {+} → String (concatenation operation) and we can add enum to strings!
- E.g.: `myStr = myStr + someEnumVar;`

Automatic Type Casting Note

- From the previous example, we can see that if there does not exist a type-check entry for:

$$\alpha \times \beta \times \text{Op}$$

… then we have to check all possible automatic typecasting for both α and β.

If there exists conflicts, manual typecasting is enforced.

Manual Typecasting

- Consider the following:

(String,String,+)→String

(Integer,Integer,+)→Integer

TypeA a; TypeB b;

And both TypeA and TypeB can be typecasted with:

(String,TypeA), (String,TypeB), (Integer,TypeA), (Integer,TypeB)

What is the result type of a + b?

Manual Typecasting

- Consider the following:

(String,String,+)→String

(Integer,Integer,+)→Integer

TypeA a; TypeB b; both TypeA, TypeB → {String,Integer}

What is the result type of a + b?

Answer: Your compiler must require a or b to be manually typecasted to either String or Integer. (String)a + b;

Please note: miniJava

- miniJava does not have typecasting nor automatic conversions.
- miniJava does not allow boolean × int × Op

- This lecture is meant to strengthen your compiler theory, please see next lecture for PA3 specifics.

Enjoy your weekend!

- Make sure to wrap up WA1
- Make sure to start on WA2

- Start early on PA2!
- There will be questions on the midterm on how you solved problems in PA1+PA2!!

End