CSC488/2107 Winter 2019 — Compilers & Interpreters

https://www.cs.toronto.edu/~csc488h/

Peter McCormick
pdm@cs.toronto.edu

Agenda

- Symbol tables
- Semantic analysis

Symbol tables

- An *identifier* is a language token type, while the value of an identifier is a *name* (typically a string)
- A *symbol table* maps *names* to *symbols*
- What language constructs create new *names*, and when are those names visible in subsequent parts of the program text?
- What information is useful to track for each symbol?
- Each language will have its own rules about scoping and symbol visibility:
    - Hierarchical
    - Parallel

Symbol table entry

- Original point of instantiation
- Kind: constant, variable, type, procedure/function
- Type information:
  - Scalar vs array vs routine
  - Link to record/class/etc. definition
  - For routines: formal parameters and their symbol information, optional return type
- Language-specific attributes
- Storage size of item
- Runtime address (offset within stack)
- Visibility modifiers
- Uses *(optional)*

Implementing a symbol table

- Important operations:
  - Create a nested scope
  - Exit from a scope
  - Lookup name in the current scope
  - Lookup name according to language scoping rules
  - Put a new name-to-symbol entry in the table
- Hierarchical map of names to symbols
  - Stack of hash tables
  - Or, a hash table of lists

Implementing a symbol table

- Entering into a major or minor scope will create a nested symbol table
  - Maps naturally between enter & exit and push & pop
- If you perform an *Ident-to-Symbol* AST transformation, do you still need the hash table anymore?
  - Debuggers depend on knowing what names are visible/ accessible at each point in the program
- Create a new name-to-symbols map anywhere new identifiers can be introduced

```
float gpa;

int main(
    int argc,
    char *argv[])
{

 char *course;
}
```

Name	Kind	Type
gpa	var	float
main	func	…

Name	Kind	Type
argc	param	int
argv	param	char **

Name	Kind	Type
course	var	char *

Type tables

- Languages that support user-defined types require additional name-to-type mapping tables
- Scoping rules may dictate whether these names appear in parallel scopes, or alongside other kinds of symbols

Type table entries

- Typical details:
  - Name
  - Kind: struct, union, enum, typedef, scalar
  - Storage size
  - Runtime information

```
int t;

struct S {
  char *name;
  int number;
};
```

Symbols

Name	Kind	Type
t	var	int
int	type	…
char	type	…

Types

Name	Kind
int	builtin
char	builtin
*char**	ptr

struct symbols

Name	Fields
S	…

Name	Type
name	char*
number	Int

```
int t;

struct S {
  char *name;
  int number;
};
```

Symbols

Name	Kind	Type
t	var	int
int	type	…
char	type	…

Types

Name	Kind
int	builtin
char	builtin
*char**	ptr

struct symbols

Name	Fields
S	…

Name	Type
name	char*
number	Int

```
int t;

struct S {
  char *name;
  int number;
};

typedef
  struct S
  R;
```

Symbols

Name	Kind	Type
t	var	int
int	type	…
char	type	…
R	type	…

Types

Name	Kind
int	builtin
char	builtin
*char**	ptr
R	struct

struct symbols

Name	Fields
S	…

Name	Type
name	char*
number	Int

C type table design

Memory layout

- A lower level language like C exposes the programmer to certain machine & memory characteristics

- Types have a natural size and typically must obey certain machine *alignment* restrictions

- struct’s and union’s inherit the most restrictive alignment of their members

```
struct {
  u8 a;
  u16 b;
  u32 c;
  u64 d;
};
```

Field	Size	Offset
a		
b		
c		
d		

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

```
struct {
  u8 a;
  u16 b;
  u32 c;
  u64 d;
};
```

Field	Size	Offset
a	1	
b	2	
c	4	
d	8	

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

```
struct {
  u8 a;
  u16 b;
  u32 c;
  u64 d;
};
```

Field	Size	Offset
a	1	0
b	2	
c	4	
d	8	

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
a															

```
struct {
  u8 a;
  u16 b;
  u32 c;
  u64 d;
};
```

Field	Size	Offset
a	1	0
b	2	2
c	4	
d	8	

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
a		b													

```
struct {
  u8 a;
  u16 b;
  u32 c;
  u64 d;
};
```

Field	Size	Offset
a	1	0
b	2	2
c	4	4
d	8	

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
a		b		c											

```
struct {
  u8 a;
  u16 b;
  u32 c;
  u64 d;
};
```

Field	Size	Offset
a	1	0
b	2	2
c	4	4
d	8	8

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
a		b		c				d							

```
struct {
  u8 a;
  u16 b;
  u32 c;
  u64 d;
};
```

Field	Size	Offset
a	1	0
b	2	2
c	4	4
d	8	8

16 bytes required to store 15 bytes of information

struct alignment = 8 bytes

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
a		b		c				d							

```
struct {
  u8 a;
  u64 d;
  u16 b;
  u32 c;
};
```

Field	Size	Offset
a	1	0
d	8	
b	2	
c	4	

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
a															

```
struct {
  u8 a;
  u64 d;
  u16 b;
  u32 c;
};
```

Field	Size	Offset
a	1	0
d	8	8
b	2	
c	4	

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
a								d							

```
struct {
  u8 a;
  u64 d;
  u16 b;
  u32 c;
};
```

Field	Size	Offset
a	1	0
d	8	8
b	2	16
c	4	

<table>
<tr><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td><td>8</td><td>9</td><td>10</td><td>11</td><td>12</td><td>13</td><td>14</td><td>15</td><td>16</td><td>17</td><td>18</td><td>19</td><td>20</td><td>21</td><td>22</td><td>23</td></tr>
<tr><td>a</td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td colspan="8">d</td><td colspan="2">b</td><td></td><td></td><td></td><td></td><td></td><td></td></tr>
</table>

```
struct {
  u8 a;
  u64 d;
  u16 b;
  u32 c;
};
```

Field	Size	Offset
a	1	0
d	8	8
b	2	16
c	4	20

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23
a								d								b				c			

```
struct {
  u8 a;
  u64 d;
  u16 b;
  u32 c;
};
```

Field	Size	Offset
a	1	0
d	8	8
b	2	16
c	4	20

24 bytes required to store 15 bytes of information

struct alignment = 8 bytes

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23
a								d								b				c			

Semantic analysis

Semantic analysis

- Checking and enforcement of non-syntactic language constraints
    - During compilation: static analysis
    - At runtime: dynamic analysis
- Kinds of analysis:
    - Visibility and accessibility
    - Type checking
    - Proper usage
    - Escape
    - Range
- Symbol tables typically construct during these analyses

Semantic analysis example

	A	=	B		
Visibility	declared(A)?		declared(B)?		
	visible(A)?		visible(B)?		
Accessibility	access(A)?		access(B)?		
	write(A)?		read(B)?		
Usage	variable(A)?		variable(B)?	const(B)?	function(B)?
Type	type(A)?		type(B)?	type(B)?	type(B)?
	scalar(A)?		scalar(B)?	scalar(B)?	scalar(B)?
					params(B)?
Usage		assignTo(A, B)?			

Type equivalence, compatibility and suitability

- When different types can be used together depends on the context and is governed by language specific rules
- **Type equivalence**: when can two objects of two different types be considered *equivalent*
- **Type compatibility**: when can two objects of two different types be considered *compatible*
- **Type suitability**: when can two objects of two different types be considered suitable for one another
- Assignment usually requires compatibility, expression operands usually require suitability

Type equivalence rules

- **Name type equivalence**:
  - Two types are *name equivalent* if they derive from the same definition
  - Allows for aliases such as *typedef*'s
- **Structural type equivalence**:
  - Two types are *structural equivalent* if their definitions line up with one another (same structure, same values, same types)

Name type equivalence

```
struct S { int foo; };
typedef struct S A;
typedef struct S B;
```

struct S, A and B are all *name type equivalent*

Structural equivalence

```
typedef struct {
  int   a;
  char *b;
  float c;
} X;
```

```
typedef struct {
  int   p;
  char *q;
  float r;
} Y;
```

X and Y are structurally equivalent

Type equivalence rules

- Type equivalence checking is used to ensure that pointers match the data type they are pointing to
    - When a pointer is assigned the address-of something
    - When a variable is passed by-reference as a parameter
- Type equivalence implies *memory layout equivalence*

Go: structural *copying*

```
type X struct {
  F string
  G int
}

type Y struct {
  F string
  G int
}
```

```
var x X
var y Y

…

y = Y(x)
```

Type checking

Type this expression

```
1 < 2
```

Type this expression

```
0 < True
```

Type this expression

```
(0 < True) :: boolean
```

Type *check* this expression

```
1 < 2
```

Type *check* this expression

```
0 < True
```

Type ***check*** this expression

```
0 < True
```

Type this expression

```
f(x, y, z)
```

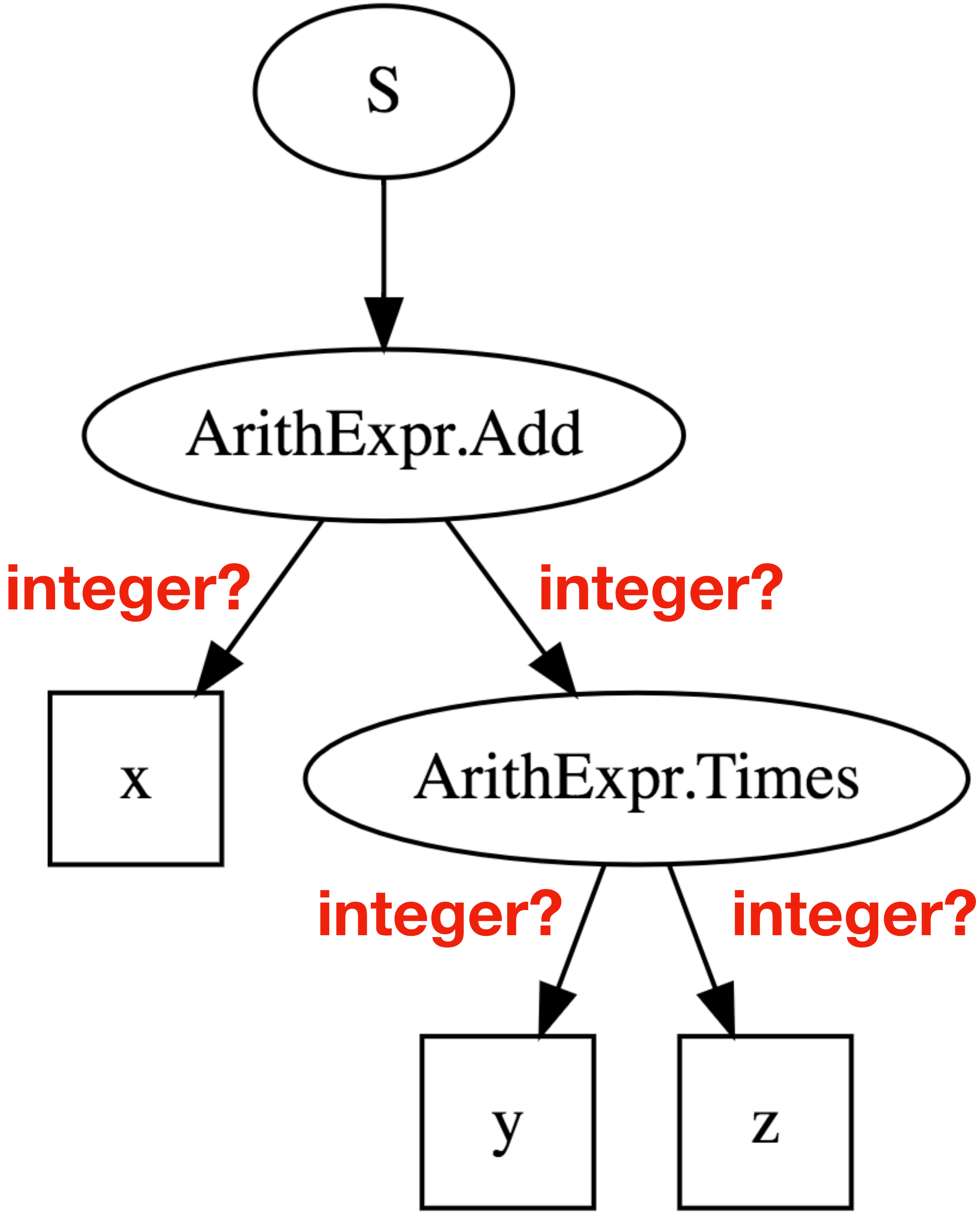
Type *check* this expression

```
f(x, y, z)
```

Type & type check this expression

```
f(x, y, z)
```

```
func f(a, b, c integer) boolean
```


S
ArithExpr.Add
integer?
integer?
x
ArithExpr.Times
integer?
integer?
y
z