THEORY OF COMPILATION

LECTURE 05

SEMANTIC ANALYSIS
You are here

Compiler


Executable code

txt

exe
What We Want

Potato potato;
Tomato tomato;
x = potato + tomato + carrot
What We Want

Potato potato;
Tomato tomato;
x = potato + tomato + carrot

... <ID,potato> <PLUS> <ID,tomato> <PLUS> <ID,carrot> EOF
What We Want

Potato potato;
Tomato tomato;
x = potato + tomato + carrot

... <ID,potato> <PLUS> <ID,tomato> <PLUS> <ID,carrot> EOF
What We Want

Potato potato;
Tomato tomato;
\[ x = \text{potato} + \text{tomato} + \text{carrot} \]

Lexical analyzer

... <ID,potato> <PLUS> <ID,tomato> <PLUS> <ID,carrot> EOF

Parser

Semantic

<table>
<thead>
<tr>
<th>symbol</th>
<th>kind</th>
<th>type</th>
<th>properties</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>var</td>
<td>?</td>
<td></td>
</tr>
<tr>
<td>carrot</td>
<td>var</td>
<td>?</td>
<td></td>
</tr>
<tr>
<td>potato</td>
<td>var</td>
<td>Potato</td>
<td></td>
</tr>
<tr>
<td>tomato</td>
<td>var</td>
<td>Tomato</td>
<td></td>
</tr>
</tbody>
</table>
What We Want

Potato potato;
Tomato tomato;
x = potato + tomato + carrot

… <ID,potato> <PLUS> <ID,tomato> <PLUS> <ID,carrot> EOF

Lexical analyzer

Parser

Semantic

<table>
<thead>
<tr>
<th>symbol</th>
<th>kind</th>
<th>type</th>
<th>properties</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>var</td>
<td>?</td>
<td></td>
</tr>
<tr>
<td>carrot</td>
<td>var</td>
<td>?</td>
<td></td>
</tr>
<tr>
<td>potato</td>
<td>var</td>
<td>Potato</td>
<td></td>
</tr>
<tr>
<td>tomato</td>
<td>var</td>
<td>Tomato</td>
<td></td>
</tr>
</tbody>
</table>

‘carrot’ is undefined
What We Want

Potato potato;
Tomato tomato;
x = potato + tomato + carrot

Lexical analyzer

... <ID,potato> <PLUS> <ID,tomato> <PLUS> <ID,carrot> EOF

Parser

Semantic

<table>
<thead>
<tr>
<th>symbol</th>
<th>kind</th>
<th>type</th>
<th>properties</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>var</td>
<td>?</td>
<td></td>
</tr>
<tr>
<td>carrot</td>
<td>var</td>
<td>?</td>
<td></td>
</tr>
<tr>
<td>potato</td>
<td>var</td>
<td>Potato</td>
<td></td>
</tr>
<tr>
<td>tomato</td>
<td>var</td>
<td>Tomato</td>
<td></td>
</tr>
</tbody>
</table>

‘carrot’ is undefined  ‘potato’ used before initialized
What We Want

Potato potato;
Tomato tomato;
\( x = \text{potato} + \text{tomato} + \text{carrot} \)

Lexical analyzer

... <ID,potato> <PLUS> <ID,tomato> <PLUS> <ID,carrot> EOF

Parser

Semantic

<table>
<thead>
<tr>
<th>symbol</th>
<th>kind</th>
<th>type</th>
<th>properties</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>var</td>
<td>?</td>
<td></td>
</tr>
<tr>
<td>carrot</td>
<td>var</td>
<td>?</td>
<td></td>
</tr>
<tr>
<td>potato</td>
<td>var</td>
<td>Potato</td>
<td></td>
</tr>
<tr>
<td>tomato</td>
<td>var</td>
<td>Tomato</td>
<td></td>
</tr>
</tbody>
</table>

‘carrot’ is undefined
‘potato’ used before initialized
Cannot add ‘Potato’ and ‘Tomato’
Semantic Analysis

• Often called “Contextual analysis”
  ‣ As opposed to our syntax analysis — which was “context free”

• Properties that cannot be formulated via CFG
  ‣ Declare before use
  ‣ Type checking
  ‣ Initialization
  ‣ ...

• Properties that are clumsy to formulate via CFG
  ‣ “break” only appears inside a loop
  ‣ ...

Semantic Analysis

• Identification
  ‣ Gather information about each named item in the program
  ‣ *e.g.*, what is the declaration for each usage

• Context checking
  ‣ Type checking
  ‣ *e.g.*, the condition in an if-statement is a Boolean
month : integer RANGE 1..12;
month := 1;
while (month <= 12) {
    print(month_name[month]);
    month := month + 1;
}
Symbol table

- A table containing information about identifiers in the program
- Single entry for each named item

```
month : integer RANGE 1..12;
...
month := 1;
while (month <= 12) {
    print(month_name[month]);
    month := month + 1;
}
```

<table>
<thead>
<tr>
<th>name</th>
<th>pos</th>
<th>type</th>
</tr>
</thead>
<tbody>
<tr>
<td>month</td>
<td>1</td>
<td>int 1..12</td>
</tr>
<tr>
<td>month_name</td>
<td>...</td>
<td>string[1..12]</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```
Not so fast...

```c
struct one_int {
    int i;
} i;

main() {
    i.i = 42;
    int t = i.i;
    printf("%d", t);
}
```
struct one_int {
    int i;
} i;

main() {
    i.i = 42;
    int t = i.i;
    printf("%d",t);
}
Not so fast...

```c
struct one_int {
    int i;
} i;

main() {
    i.i = 42;
    int t = i.i;
    printf("%d", t);
}
```

A struct field named “i”
A struct variable named “i”
struct one_int {
    int i;
} i;

main() {
    i.i = 42;
    int t = i.i;
    printf("%d", t);
}
struct one_int {
    int i;
} i;

main() {
    i.i = 42;
    int t = i.i;
    printf("%d", t);
}
struct one_int {
    int i;
} i;

main() {
    i.i = 42;
    int t = i.i;
    printf("%d",t);
    {
        int i = 73;
        printf("%d",i);
    }
}
struct one_int {
    int i;
} i;

main() {
    i.i = 42;
    int t = i.i;
    printf("%d",t);
    {
        int i = 73;
        printf("%d",i);
    }
}
struct one_int {
    int i;
} i;

main() {
    i.i = 42;
    int t = i.i;
    printf("%d", t);
    {
        int i = 73;
        printf("%d", i);
    }
}
struct one_int {
    int i;
} i;

main() {
    i.i = 42;
    int t = i.i;
    printf("%d", t);
    {
        int i = 73;
        printf("%d", i);
    }
}
Scopes

• Typically stack structured scopes

• Scope entry
  – push new empty scope element

• Scope exit
  – pop scope element and discard its content

• Identifier declaration
  – identifier created inside (current) top scope

• Identifier Lookup
  – Search for identifier top-down in scope stack
{ int the=1;
  int fish=2;
  int thanks=3;
  {
    int x = 42;
    int all = 73;
    {
      ...
    }
  }
}
Scope and Symbol Table

• Scope × Identifier → properties
  ‣ Expensive lookup

• A better solution
  ‣ Hash table over identifiers
  ‣ List of scopes for each identifier
Hash Table-based Symbol Table

Id.info

- name
- macro
- decl

“x”

2 P • → 1 P •

“thanks”

2 P • → 0 P •

“so”

3 P • → //
Scope info

Scope stack

1. Id.info("so")  
   //
   3

2. Id.info("and")  
   Id.info("thanks")  
   Id.info("x")  
   //
   2

3. Id.info("x")  
   Id.info("all")  
   //
   1

4. Id.info("the")  
   Id.info("fish")  
   Id.info("thanks")  
   //
   0

(now just pointers to the corresponding record in the symbol table)
Remember Lexing+Parsing?
Remember Lexing+Parsing?

- How did we know to always map an identifier to the same token?
- We didn’t! now it is the first time.
Semantic Checks

• Scope rules
  ‣ Use symbol table to check that
    ◦ Identifiers defined before used
    ◦ No multiple definition of same identifier

• Type checking
  ‣ Check that types in the program are consistent
    ◦ How?
Types

• **What is a type?**
  – Simplest answer: a set of values
  – Integers, real numbers, booleans, ...

• **Why do we care?**
  – Safety
    • Guarantee that certain errors cannot occur at runtime
  – Abstraction
    • Hide implementation details
  – Documentation
  – Optimization
Type System

- A type system of a programming language is a way to define how “good” programs behave
  - Good programs = well-typed programs
  - Bad programs = not well typed

- Type checking
  - Static typing – most checking at compile time
  - Dynamic typing – most checking at runtime

- Type inference
  - Automatically infer types for a program (or show that there is no valid typing)
Static Typing vs. Dynamic Typing

• Static type checking is **conservative**
  – Any program that is determined to be well-typed is free from certain kinds of errors
  – May reject programs that cannot be statically determined to be safe
    ▸ Why?

• Dynamic type checking
  – May accept more programs as valid (runtime info)
  – Errors not caught at compile time
  – Runtime overhead
Type Checking

- Type rules specify
  - which types can be combined with certain operator
  - Assignment of expression to variable
  - Formal and actual parameters of a method call

- Examples

  "drive" + "drink"

  42 + "the answer"
Type Checking

• Type rules specify
  – which types can be combined with certain operator
  – Assignment of expression to variable
  – Formal and actual parameters of a method call

• Examples

  ```java
  string
  "drive" + "drink"
  ```

  ```java
  42 + "the answer"
  ```
Type Checking

• Type rules specify
  – which types can be combined with certain operator
  – Assignment of expression to variable
  – Formal and actual parameters of a method call

• Examples

  `string string``

  "drive" + "drink"

  `42 + "the answer"`
Type Checking

- Type rules specify
  - which types can be combined with certain operator
  - Assignment of expression to variable
  - Formal and actual parameters of a method call

- Examples

  `string       string`
  `“drive” + “drink”`
  `string`

  `42 + “the answer”`
Type Checking

- Type rules specify
  - which types can be combined with certain operator
  - Assignment of expression to variable
  - Formal and actual parameters of a method call

- Examples

  string + string
  "drive" + "drink"
  string

  int
  42 + "the answer"
Type Checking

- Type rules specify
  - which types can be combined with certain operator
  - Assignment of expression to variable
  - Formal and actual parameters of a method call

- Examples

```plaintext
string    string
"drive" + "drink"

string

int    string
42 + "the answer"
```
Type Checking

- Type rules specify
  - which types can be combined with certain operator
  - Assignment of expression to variable
  - Formal and actual parameters of a method call

- Examples

  string    string
  “drive” + “drink”

  string

  int    string
  42 + “the answer”

  ERROR
Type Checking Rules

- Specify for each operator
  - Types of operands
  - Type of result

- Basic Types
  - Building blocks for the type system (type rules)
  - *e.g.*, int, boolean, (sometimes) string

- Type Expressions
  - Array types
  - Function types
  - Record types / Classes
Typing Rules

If $E_1$ has type int and $E_2$ has type int,
then $E_1 + E_2$ has type int

$E_1 : \text{int} \quad E_2 : \text{int}$

$E_1 + E_2 : \text{int}$
More Typing Rules (examples)

```
true : boolean
false : boolean

int-literal : int
string-literal : string

E1 : int  E2 : int
----------------------
  E1 op E2 : int
  op ∈ { +, -, /, *, %}

E1 : int  E2 : int
----------------------
  E1 rop E2 : boolean
  rop ∈ { <=, <, >, >=}

E1 : T    E2 : T
----------------------
  E1 rop E2 : boolean
  rop ∈ { ==, !=}
```
And Even More Typing Rules

\[
\frac{E_1 : \text{boolean}}{} \quad \frac{E_2 : \text{boolean}}{} \quad \text{lop} \in \{\ &\&\,,\ \mid\mid\ \}\quad \frac{E_1 \text{ lop} E_2 : \text{boolean}}{}
\]

\[
\frac{E_1 : \text{int}}{}\quad \frac{E_1 : \text{int}}{}\quad \frac{E_1 : \text{boolean}}{}\quad \frac{E_1 : \text{boolean}}{}
\]

\[
\frac{E_1 : [T]}{}\quad \frac{E_1 : [T]}{}\quad \frac{E_2 : \text{int}}{}\quad \frac{E_1 : \text{int}}{}
\]

\[
\frac{E_1.\text{length} : \text{int}}{}\quad \frac{E_1[E_2] : T}{E_1 : [T]}\quad \frac{\text{new } T[E_1] : T[]}{E_1 : \text{int}}
\]
Type Checking

• Traverse AST and assign types for AST nodes
  – Use typing rules to compute node types

  ■ Alternative: type-check during parsing
    – (Slightly) more complicated
    – But naturally also more efficient
Type Rules

45 > 32 && !false
Type Rules

45 > 32 && !false
Type Rules

45 > 32 && !false
Type Rules

45 > 32 && !false
Type Rules

45 > 32 && !false
Type Rules

\[ 45 > 32 \land \neg \text{false} \]

\textbf{int-literal : int}
Type Rules

45 > 32 && !false

int-literal : int

: int
Type Rules

\[ 45 > 32 \land \neg \text{false} \]

\[ : \text{int} \]

\[ \text{int-literal} : \text{int} \]
Type Rules

45 > 32 && !false

int-literal : int

: int
: int
Type Rules

\[ 45 > 32 \land \neg false \]

...
Type Rules

\[ 45 > 32 \land \neg \text{false} \]

\[ E_1 : \text{int} \quad E_2 : \text{int} \]

\[ E_1 \triangle E_2 : \text{boolean} \quad \text{for} \quad \triangle \in \{\leq, <, >, \geq\} \]

\[ \text{int-literal} : \text{int} \]
Type Rules

\[
\text{E}_1 : \text{int} \quad \text{E}_2 : \text{int} \\
\text{E}_1 \diamond \text{E}_2 : \text{boolean} \\
\text{for} \quad \diamond \in \{ \leq, <, >, \geq \}
\]

\[
\text{int-literal} : \text{int}
\]

\[
45 > 32 \quad \&\& \quad \text{!false}
\]
Type Rules

\[ 45 > 32 \land \neg \text{false} \]
Type Rules

45 > 32 && !false
Type Rules

\[
\begin{align*}
E_1 &\colon \text{int} \quad E_2 \colon \text{int} \\
E_1 \diamond E_2 &\colon \text{boolean} \\
\text{for } \diamond &\in \{\leq, <, >, \geq\}
\end{align*}
\]

\[
45 > 32 \quad \&\& \quad \neg \text{false}
\]
Type Rules

\[
\begin{align*}
\text{int-literal} & : \text{int} \\
\text{false} & : \text{boolean}
\end{align*}
\]

\[
\begin{align*}
\text{E}_1 : \text{int} & \quad \text{E}_2 : \text{int} \\
\text{E}_1 \diamond \text{E}_2 : \text{boolean} \\
\text{for } \diamond & \in \{\leq, <, >, \geq\} \\
\text{false} : \text{boolean} \\
\text{int-literal} : \text{int}
\end{align*}
\]

45 > 32 && !false
Type Rules

![Type Rules Diagram]

\[
\begin{align*}
\text{E}_1 : \text{int} & \quad \text{E}_2 : \text{int} \\
\text{E}_1 \triangleright \text{E}_2 : \text{boolean} & \quad \text{for} \quad \triangleright \in \{\leq, <, >, \geq\}
\end{align*}
\]

\[
\text{false} : \text{boolean}
\]

\[
\text{int-literal} : \text{int}
\]

45 > 32 && !false
Type Rules

\[ 45 > 32 && \neg \text{false} \]
Type Rules

\[ 45 > 32 \land \lnot \text{false} \]

Diagram:
- BinopExpr \( \text{op} = '\&\&' \)
  - intLiteral \( \text{value} = 45 \)
  - intLiteral \( \text{value} = 32 \)
- BinopExpr \( \text{op} = '>' \)
- UnopExpr \( \text{op} = '!' \)
- boolLiteral \( \text{value} = \text{false} \)

Rules:
- \( E_1 : \text{boolean} \)
  - \( \lnot E_1 : \text{boolean} \)
- \( E_1 : \text{int} \quad E_2 : \text{int} \)
  - \( E_1 \diamond E_2 : \text{boolean} \)
    - for \( \diamond \in \{\leq, <, >, \geq\} \)
  - \( \text{false} : \text{boolean} \)
- \( \text{int-literal} : \text{int} \)
Type Rules

\[ 45 > 32 \land \neg \text{false} \]

Diagram:
- **BinopExpr**
  - \( \text{op} = \text{"\&\&"} \)
  - \( \text{value} = 45 \)
  - \( \text{value} = 32 \)
  - \( \text{value} = \text{false} \)

- **UnopExpr**
  - \( \text{op} = \text{"!"} \)
  - \( \text{value} = \text{false} \)

Rules:
- \( E_1 : \text{boolean} \)
  - \( \neg E_1 : \text{boolean} \)
- \( E_1 : \text{int} \)
  - \( E_2 : \text{int} \)
  - \( E_1 \triangle E_2 : \text{boolean} \)
    - for \( \triangle \in \{ \leq, <, >, \geq \} \)
  - \( \text{false} : \text{boolean} \)
  - \( \text{int-literal} : \text{int} \)
Type Rules

\[ 45 > 32 \land \neg \text{false} \]
Type Rules

45 > 32 && !false

E₁ : boolean  E₂ : boolean
---------
E₁ ◇ E₂ : boolean
for ◇ ∈ {&&, ||}

---------
E₁ : boolean
! E₁ : boolean

---------
E₁ : int  E₂ : int
---------
E₁ ◇ E₂ : boolean
for ◇ ∈ {<=, <, >, >=}

---------
false : boolean

---------
int-literal : int
Type Rules

45 > 32 && !false

\( E_1 : \text{boolean} \quad E_2 : \text{boolean} \)

\[
E_1 \land E_2 : \text{boolean}
\]

for \( \land \in \{ \&\&, \mid\mid \} \)

\[
E_1 : \text{boolean} \\
! E_1 : \text{boolean}
\]

\[
E_1 : \text{int} \\
E_2 : \text{int} \\
E_1 \land E_2 : \text{boolean}
\]

for \( \land \in \{ \leq, <, >, \geq \} \)

false : boolean

\[
\text{false} : \text{boolean}
\]

\[
\text{int-literal} : \text{int}
\]
Type Rules

45 > 32 && !false
Strongly Typed vs. Weakly Typed

- Coercion
- Strongly typed
  - C, C++, Java
- Weakly typed
  - Perl, PHP

(Not everybody agrees on this classification)
Strongly Typed vs. Weakly Typed

- Coercion
- Strongly typed
  - C, C++, Java
- Weakly typed
  - Perl, PHP

(Not everybody agrees on this classification)

Perl

```perl
$a=31;
$b="42x";
$c=$a+$b;
print $c;
```

C

```c
main() {
int a=31;
char b[3]="42x";
int c=a+b;
}
```

Java

```java
public class... {
public static void main() {
    int a=31;
    String b ="42x";
    int c=a+b;
}
}
```
Strongly Typed vs. Weakly Typed

- Coercion
- Strongly typed
  - C, C++, Java
- Weakly typed
  - Perl, PHP

(Not everybody agrees on this classification)
**Strongly Typed vs. Weakly Typed**

- **Coercion**
- **Strongly typed**
  - C, C++, Java
- **Weakly typed**
  - Perl, PHP

(Not everybody agrees on this classification)

```
$a=31;
$b="42x";
$c=$a+$b;
print $c;
```

**Output: 73**

```
main() {
    int a=31;
    char b[3]="42x";
    int c=a+b;
}
```

**C**

```
warning: initialization makes integer from pointer without a cast
```

```
public class... {
    public static void main() {
        int a=31;
        String b ="42x";
        int c=a+b;
    }
}
```

**Java**

```
error: Incompatible type for declaration. Can't convert java.lang.String to int
```

```
perl
```

```
perl
```
Coming Up

Keep Calm & Carry On With Semantic Analysis
THEORY OF COMPILATION

LECTURE 05

SEMANTIC ANALYSIS
You are here

Compiler

Source text

Lexical Analysis
Syntax Analysis
Parsing
Semantic Analysis
Inter. Rep. (IR)
Code Gen.

Executable code

txt
exe
Reminder — Semantic Analysis

- **Identification**
  - Read declarations, build symbols table & scope table
  - Associate declaration and uses

- **Context checking**
  - Type checking: check that the program is *type-safe*
  - *e.g.*, the condition in an if-statement is a Boolean
Reminder — Semantic Analysis

Potato potato;
Tomato tomato;
x = potato + tomato + carrot
Reminder — Semantic Analysis

Potato potato;
Tomato tomato;
x = potato + tomato + carrot

... <ID,potato> <PLUS> <ID,tomato> <PLUS> <ID,carrot> EOF
Reminder — Semantic Analysis

Potato potato;
Tomato tomato;
\[ x = \text{potato} + \text{tomato} + \text{carrot} \]

\[ \ldots \text{<ID, potato> <PLUS> <ID, tomato> <PLUS> <ID, carrot> EOF} \]

Lexical analyzer

Parser
Reminder — Semantic Analysis

Potato potato;
Tomato tomato;
\[ x = \text{potato} + \text{tomato} + \text{carrot} \]

Lexical analyzer

... `<ID,potato> <PLUS> <ID,tomato> <PLUS> <ID,carrot> EOF`

Parser

<table>
<thead>
<tr>
<th>symbol</th>
<th>kind</th>
<th>type</th>
<th>properties</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>var</td>
<td>?</td>
<td></td>
</tr>
<tr>
<td>carrot</td>
<td>var</td>
<td>?</td>
<td></td>
</tr>
<tr>
<td>potato</td>
<td>var</td>
<td>Potato</td>
<td></td>
</tr>
<tr>
<td>tomato</td>
<td>var</td>
<td>Tomato</td>
<td></td>
</tr>
</tbody>
</table>
Reminder — Semantic Analysis

Potato potato;
Tomato tomato;
x = potato + tomato + carrot

Lexical analyzer

... <ID,potato> <PLUS> <ID,tomato> <PLUS> <ID,carrot> EOF

Parser

Semantic

<table>
<thead>
<tr>
<th>symbol</th>
<th>kind</th>
<th>type</th>
<th>properties</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>var</td>
<td>?</td>
<td></td>
</tr>
<tr>
<td>carrot</td>
<td>var</td>
<td>?</td>
<td></td>
</tr>
<tr>
<td>potato</td>
<td>var</td>
<td>Potato</td>
<td></td>
</tr>
<tr>
<td>tomato</td>
<td>var</td>
<td>Tomato</td>
<td></td>
</tr>
</tbody>
</table>

‘carrot’ is undefined
Reminder — Semantic Analysis

Potato potato;
Tomato tomato;
x = potato + tomato + carrot

... <ID,potato> <PLUS> <ID,tomato> <PLUS> <ID,carrot> EOF

Lexical analyzer

Parser

Semantic

<table>
<thead>
<tr>
<th>symbol</th>
<th>kind</th>
<th>type</th>
<th>properties</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>var</td>
<td>?</td>
<td></td>
</tr>
<tr>
<td>carrot</td>
<td>var</td>
<td>?</td>
<td></td>
</tr>
<tr>
<td>potato</td>
<td>var</td>
<td>Potato</td>
<td></td>
</tr>
<tr>
<td>tomato</td>
<td>var</td>
<td>Tomato</td>
<td></td>
</tr>
</tbody>
</table>

‘carrot’ is undefined  ‘potato’ used before initialized
Reminder — Semantic Analysis

Potato potato;
Tomato tomato;
x = potato + tomato + carrot

 Lexical analyzer

... <ID,potato> <PLUS> <ID,tomato> <PLUS> <ID,carrot> EOF

Parser

Semantic

<table>
<thead>
<tr>
<th>symbol</th>
<th>kind</th>
<th>type</th>
<th>properties</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>var</td>
<td>?</td>
<td></td>
</tr>
<tr>
<td>carrot</td>
<td>var</td>
<td>?</td>
<td></td>
</tr>
<tr>
<td>potato</td>
<td>var</td>
<td>Potato</td>
<td></td>
</tr>
<tr>
<td>tomato</td>
<td>var</td>
<td>Tomato</td>
<td></td>
</tr>
</tbody>
</table>

‘carrot’ is undefined  ‘potato’ used before initialized  Cannot add ‘Potato’ and ‘Tomato’
Type Declarations

• So far, we ignored the fact that types can also be **user-defined**

```pascal
TYPE Int_Array = ARRAY [Integer 1..42] OF Integer;  (explicitly)

Var a : ARRAY [Integer 1..42] OF Real;             (anonymously)
```
Type Declarations

Var a : ARRAY [Integer 1..42] OF Real;

TYPE #type01_in_line_73 = ARRAY [Integer 1..42] OF Real;
Var a : #type01_in_line_73;
Forward References

TYPE Ptr_List_Entry = POINTER TO List_Entry;

TYPE List_Entry =
  RECORD
    Element : Integer;
    Next : Ptr_List_Entry;
  END RECORD;

• Forward references must be resolved
• A forward reference is added to the symbol table (as unresolved), and later updated when the type declaration is met
• At the end of scope, check that all forward refs have been resolved
• Must add check for circularity
Type Table

• All types in a compilation unit are collected in a type table

• For each type, its table entry contains
  ‣ Type constructor: basic, record, array, pointer,…
  ‣ Size and alignment requirements
    • to be used later in code generation
  ‣ Types of components (if applicable)
    • e.g., type of array element, types of record fields
Nominal Type System

Type Equivalence = Name Equality

Type \( t_1 \) = ARRAY[Integer] OF Integer;
Type \( t_2 \) = ARRAY[Integer] OF Integer;
Nominal Type System

Type Equivalence = Name Equality

Type \( t_1 = \text{ARRAY[Integer]} \ OF \text{Integer} \);
Type \( t_2 = \text{ARRAY[Integer]} \ OF \text{Integer} \);

\( t_1 \) not (nominally) equivalent to \( t_2 \)
Nominal Type System
Type Equivalence = Name Equality

Type $t_1 = \text{ARRAY}[\text{Integer}] \text{ OF Integer};$
Type $t_2 = \text{ARRAY}[\text{Integer}] \text{ OF Integer};$

$t_1$ not (nominally) equivalent to $t_2$

Type $t_3 = \text{ARRAY}[\text{Integer}] \text{ OF Integer};$
Type $t_4 = t_3$
Nominal Type System

Type Equivalence = Name Equality

Type $t_1 = \text{ARRAY}[\text{Integer}] \text{ OF Integer};$

Type $t_2 = \text{ARRAY}[\text{Integer}] \text{ OF Integer};$

$t_1$ not (nominally) equivalent to $t_2$

Type $t_3 = \text{ARRAY}[\text{Integer}] \text{ OF Integer};$

Type $t_4 = t_3$

$t_3$ equivalent to $t_4$
Structural Type System

Type Equivalence = Structure Isomorphism

Type t5 = RECORD c: Integer; p: POINTER TO t5; END RECORD;
Type t6 = RECORD c: Integer; p: POINTER TO t6; END RECORD;
Type t7 =
RECORD
  c: Integer;
  p: POINTER TO
    RECORD
      c: Integer;
      p: POINTER to t5;
    END RECORD;
END RECORD;
Structural Type System

Type Equivalence = Structure Isomorphism

Type t5 = RECORD c: Integer; p: POINTER TO t5; END RECORD;
Type t6 = RECORD c: Integer; p: POINTER TO t6; END RECORD;
Type t7 =
    RECORD
        c: Integer;
        p: POINTER TO
            RECORD
                c: Integer;
                p: POINTER to t5;
            END RECORD;
    END RECORD;

T5, t6, t7 are all (structurally) equivalent
In Practice

- Almost all modern languages use a nominal type system
  - Why?
Coercions

• If we expect a value of type T1 at some point in the program, and find a value of type T2, is that acceptable?

```java
float x = 3.141;
int y = x;
```
Coercions

• If we expect a value of type T1 at some point in the program, and find a value of type T2, is that acceptable?

```java
float x = 3.141;
int y = x;
```
Coercions

• If we expect a value of type T1 at some point in the program, and find a value of type T2, is that acceptable?

```c
int x = 22;
float y = x;
```
Coercions

- If we expect a value of type T1 at some point in the program, and find a value of type T2, is that acceptable?

```cpp
int x = 22;
float y = x;
```

```cpp
int x = 22;
float y = int_to_float(x);
```

l-values and r-values

dst := src

• What is dst? What is src?
  − dst is a memory location where the value should be stored
  − src is a value
• “location” on the left of the assignment called an l-value
• “value” on the right of the assignment is called an r-value
l-values and r-values (example)

\[ x := y + 1 \]

\[
\begin{array}{ccc}
\vdots & & \\
0x42 & 73 & x \\
\vdots & & \\
0x48 & 16 & y \\
\vdots & & \\
\end{array}
\]

\[
\begin{array}{ccc}
\vdots & & \\
0x42 & 17 & x \\
\vdots & & \\
0x48 & 16 & y \\
\vdots & & \\
\end{array}
\]
l-values and r-values (example)

\[
x := A[1] \quad \checkmark
\]

\[
x := A[A[1]] \quad \checkmark \quad \text{ok}
\]

\[
A[A[1]] := x + 1 \quad \checkmark
\]

\[
x + 1 := A[1] \quad \xmark \quad \text{not ok}
\]
### l-values and r-values

<table>
<thead>
<tr>
<th>found</th>
<th>expected</th>
</tr>
</thead>
<tbody>
<tr>
<td>l-value</td>
<td>✓</td>
</tr>
<tr>
<td>r-value</td>
<td>✗ error</td>
</tr>
</tbody>
</table>
Subtyping

- A basic concept in Object-oriented Programming
  - Every class has (can have) a superclass

```cpp
class GeniusMouse :
    public BluetoothMouse {
public:
    void initiatePairing();
    rk_t bond(BluetoothSocket& socket);
protected:
    ...
};
class GeniusMouse
    extends BluetoothMouse {
public void initiatePairing();
public Rk
    bond(BluetoothSocket socket);
protected ...
}
```
Subtyping

• Subtyping relation:
  ‣ “T is a sub-type of S”

• Handled with corresponding typing rule

\[
\begin{align*}
E_1 : T & \quad T <: S \\
\hline
E_1 : S
\end{align*}
\]

• As a consequence:
  ‣ Each term has more than one type
  ‣ Need to find the appropriate type for each context
Subtyping

- A basic concept in Object-oriented Programming
  - Every class has (can have) a superclass

```
GeniusMouse
LogitechMouse
CheapMouse
```

```
Mouse
├── BluetoothMouse
│   └── GeniusMouse
├── USBMouse
│   └── LogitechMouse
│         └── LogitechMouse
└── CheapMouse
```

Inheritance Tree
Subtyping

- A basic concept in Object-oriented Programming
  - Every class has (can have) a *superclass*

```
BluetoothDevice

Mouse

BluetoothMouse

GeniusMouse

LogitechMouse

USBMouse

CheapMouse

USBDevice

Inheritance Tree
```
Subtyping

- A basic concept in Object-oriented Programming
  - Every class has (can have) a **superclass**
Subtyping

- A basic concept in Object-oriented Programming
  - Every class has (can have) a superclass

Inheritance Tree (DAG)
Subtyping

- A basic concept in Object-oriented Programming
  - Every class has (can have) a **superclass**
Subtyping

- Marking some of the types as pure abstract helps alleviate some of the difficulty in multiple inheritance
Subtyping

- Marking some of the types as pure abstract helps alleviate some of the difficulty in multiple inheritance.
Subtyping

- Marking some of the types as pure abstract helps alleviate some of the difficulty in multiple inheritance.
Subtyping

• Upcast
  ‣ The compiler will insert an implicit conversion from subtype to supertype (similar to coercions)

    ```java
    Mouse m = new GeniusMouse();
    registerDevice(m);
    ```
Subtyping

• Upcast
  ‣ The compiler will insert an implicit conversion from subtype to supertype (similar to coercions)

```java
Mouse m = new GeniusMouse();
registerDevice(m);
Mouse m = GeniusMouse_to_Mouse(
    new GeniusMouse());
registerDevice(Mouse_to_Device(m));
```
So far...

- Static correctness checking
  - Identification
  - Type checking

- **Identification** matches applied occurrences of identifier to its defining occurrence
  - The *symbol table* maintains this information

- Type checking checks which type combinations are legal

- Each node in the AST of an expression represents either an l-value (location) or an r-value (value)
How does this magic happen?

• We probably need to go over the AST?

• How does this relate to the clean formalism of the parser?
Syntax Directed Translation

- **Semantic attributes**
  - Attributes attached to grammar symbols

- **Semantic actions**
  - (already mentioned when we learned recursive descent)
    - Defines how to update the attributes

- **Attribute grammars**
Attribute Grammars

• Attributes
  ▸ Every grammar symbol has attached attributes
    ▪ Example: Expr.type

• Semantic actions
  ▸ Every production rule can define how to assign values to attributes

```
Expr → Expr + Term

{ if ((second Expr).type == Term.type)
    (first Expr).type = (second Expr).type;
  else error;
}
```
Attribute Grammars

• Attributes
  ‣ Every grammar symbol has attached attributes
    • Example: Expr.type

• Semantic actions
  ‣ Every production rule can define how to assign values to attributes

Expr → Expr + Term

{ if ((second Expr).type == Term.type)
    (first Expr).type = (second Expr).type;
  else error;  
}
Attribute Grammars

• Attributes
  ‣ Every grammar symbol has attached attributes
    • Example: Expr.type

• Semantic actions
  ‣ Every production rule can define how to assign values to attributes

Expr → Expr + Term

{ if ((second Expr).type == Term.type)
    (first Expr).type = (second Expr).type;
  else error;  }

Indexed symbols

- Add indexes to distinguish repeated grammar symbols
- Does not affect grammar
- Used in semantic actions

\[
\text{Expr} \rightarrow \text{Expr} + \text{Term}
\]

\[
\text{Expr} \rightarrow \text{Expr}_1 + \text{Term}
\]

\{
  \text{if} \ (\text{Expr}_1.\text{type} == \text{Term.}\text{type})
  \begin{align*}
    \text{Expr.}\text{type} &= \text{Expr}_1.\text{type}; \\
    \text{else} \text{error;}
  \end{align*}
\}
Example

float x, y, z

<table>
<thead>
<tr>
<th>Production</th>
<th>Semantic Rule</th>
</tr>
</thead>
<tbody>
<tr>
<td>D → T L</td>
<td>L.in = T.type</td>
</tr>
<tr>
<td>T → int</td>
<td>T.type = integer</td>
</tr>
<tr>
<td>T → float</td>
<td>T.type = float</td>
</tr>
<tr>
<td>L → L₁, id</td>
<td>L₁.in = L.in&lt;br&gt;addType(id.entry, L.in)</td>
</tr>
<tr>
<td>L → id</td>
<td>addType(id.entry, L.in)</td>
</tr>
</tbody>
</table>
Attribute Evaluation

- Build the AST
- Fill attributes of terminals with values derived from their representation
- Execute evaluation rules of the nodes to assign values until no new values can be assigned
  - In the right order such that
    - No attribute value is used before it’s available
    - Each attribute will get a value only once
Dependencies

• A semantic equation $a = f(b_1, \ldots, b_m)$ requires computation of $b_1, \ldots, b_m$ to determine the value of $a$

• The value of $a$ depends on $b_1, \ldots, b_m$
  ‣ We write $a \leftarrow b_i$
Attribute Evaluation

- Build the AST
- Build dependency graph
- Compute evaluation order using topological ordering
- Execute evaluation rules based on topological ordering
- Works as long as there are no cycles
Convention:
Add dummy variables for side effects.
<table>
<thead>
<tr>
<th>Production</th>
<th>Semantic Rule</th>
</tr>
</thead>
<tbody>
<tr>
<td>D → T L</td>
<td>L.in = T.type</td>
</tr>
<tr>
<td>T → int</td>
<td>T.type = integer</td>
</tr>
<tr>
<td>T → float</td>
<td>T.type = float</td>
</tr>
<tr>
<td>L → L₁, id</td>
<td>L₁.in = L.in</td>
</tr>
<tr>
<td></td>
<td>addType(id.entry, L.in)</td>
</tr>
<tr>
<td>L → id</td>
<td>addType(id.entry, L.in)</td>
</tr>
</tbody>
</table>

**Convention:** Add dummy variables for side effects.

**L.dmy = addType(id.entry, L.in)**
Building Dependency Graph

• All semantic equations take the form

\[ \text{attr}_1 = f_1(\text{attr}_{1.1}, \text{attr}_{1.2}, \ldots) \]
\[ \text{attr}_2 = f_2(\text{attr}_{2.1}, \text{attr}_{2.2}, \ldots) \]

• Actions with side effects use a dummy attribute

• Build a directed dependency graph G
  – For every attribute \( a \) of a node \( u \) in the AST create a node \( u.a \)
  – For every dependency between attributes \( u.a_1 \leftarrow v.a_2 \) create an edge of the form \( (v.a_2, u.a_1) \)
Example

float x, y, z

<table>
<thead>
<tr>
<th>Prod.</th>
<th>Semantic Rule</th>
</tr>
</thead>
<tbody>
<tr>
<td>D → T L</td>
<td>L.in = T.type</td>
</tr>
<tr>
<td>T → int</td>
<td>T.type = integer</td>
</tr>
<tr>
<td>T → float</td>
<td>T.type = float</td>
</tr>
<tr>
<td>L → L₁, id</td>
<td>L₁.in = L.in, addType(id.entry, L.in)</td>
</tr>
<tr>
<td>L → id</td>
<td>addType(id.entry, L.in)</td>
</tr>
</tbody>
</table>
Example

float x, y, z

Prod. | Semantic Rule
--- | ---
D → T L | L.in = T.type
T → int | T.type = integer
T → float | T.type = float
L → L₁, id | L₁.in = L.in
| addType(id.entry, L₁.in)
L → id | addType(id.entry, L.in)
Example

float x, y, z

**Prod.** | **Semantic Rule**
--- | ---
D → T L | L.in = T.type
T → int | T.type = integer
T → float | T.type = float
L → L₁, id | L₁.in = L.in
 | addType(id.entry, L.in)
L → id | addType(id.entry, L.in)
Example

\[
\text{float } x, y, z
\]

<table>
<thead>
<tr>
<th>Prod.</th>
<th>Semantic Rule</th>
</tr>
</thead>
<tbody>
<tr>
<td>D → T L</td>
<td>( \text{L.in} = \text{T.type} )</td>
</tr>
<tr>
<td>T → int</td>
<td>( \text{T.type} = \text{integer} )</td>
</tr>
<tr>
<td>T → float</td>
<td>( \text{T.type} = \text{float} )</td>
</tr>
<tr>
<td>L → L₁, id</td>
<td>( \text{L₁.in} = \text{L.in} ) \text{addType(id.entry, L.in)}</td>
</tr>
<tr>
<td>L → id</td>
<td>\text{addType(id.entry, L.in)}</td>
</tr>
</tbody>
</table>
Example

float x, y, z

<table>
<thead>
<tr>
<th>Prod.</th>
<th>Semantic Rule</th>
</tr>
</thead>
<tbody>
<tr>
<td>D → T L</td>
<td>L.in = T.type</td>
</tr>
<tr>
<td>T → int</td>
<td>T.type = integer</td>
</tr>
<tr>
<td>T → float</td>
<td>T.type = float</td>
</tr>
<tr>
<td>L → L₁, id</td>
<td>L₁.in = L.in, addType(id.entry, L.in)</td>
</tr>
<tr>
<td>L → id</td>
<td>addType(id.entry, L.in)</td>
</tr>
</tbody>
</table>
Example

float x, y, z

<table>
<thead>
<tr>
<th>Prod.</th>
<th>Semantic Rule</th>
</tr>
</thead>
<tbody>
<tr>
<td>D → T L</td>
<td>L.in = T.type</td>
</tr>
<tr>
<td>T → int</td>
<td>T.type = integer</td>
</tr>
<tr>
<td>T → float</td>
<td>T.type = float</td>
</tr>
<tr>
<td>L → L₁, id</td>
<td>L₁.in = L.in&lt;br&gt;addType(id.entry, L.in)</td>
</tr>
<tr>
<td>L → id</td>
<td>addType(id.entry, L.in)</td>
</tr>
</tbody>
</table>
Example

float x, y, z

<table>
<thead>
<tr>
<th>Prod.</th>
<th>Semantic Rule</th>
</tr>
</thead>
<tbody>
<tr>
<td>D → T L</td>
<td>L.in = T.type</td>
</tr>
<tr>
<td>T → int</td>
<td>T.type = integer</td>
</tr>
<tr>
<td>T → float</td>
<td>T.type = float</td>
</tr>
</tbody>
</table>
| L → L₁, id | L₁.in = L.in  
addType(id.entry, L.in) |
| L → id | addType(id.entry, L.in) |
Topological Order

• For a graph \( G=(V, E) \), \(|V|=k\)

• Ordering of the nodes as \( \langle v_1, v_2, \ldots, v_k \rangle \) such that for every edge \( (v_i, v_j) \in E \), \( i < j \)

Example topological orderings: \( \langle 1 \ 4 \ 3 \ 2 \ 5 \rangle \), \( \langle 4 \ 3 \ 5 \ 1 \ 2 \rangle \)
Example

float x, y, z

(e₁, e₂, e₃ point to symbol table entries for variables z, y, x, respectively)
Example

float x, y, z

(e₁, e₂, e₃ point to symbol table entries for variables z, y, x, respectively)
Example

float x, y, z

(e₁, e₂, e₃ point to symbol table entries for variables z, y, x, respectively)
Example

(float x, y, z)

(e₁, e₂, e₃ point to symbol table entries for variables z, y, x, respectively)
float x, y, z

(e₁, e₂, e₃ point to symbol table entries for variables z, y, x, respectively)
Example

float x, y, z

(e₁, e₂, e₃ point to symbol table entries for variables z, y, x, respectively)
Example

float x, y, z

(e₁, e₂, e₃ point to symbol table entries for variables z, y, x, respectively)
Example

float x, y, z

(e₁, e₂, e₃ point to symbol table entries for variables z, y, x, respectively)
Example

float x, y, z

(e₁, e₂, e₃ point to symbol table entries for variables z, y, x, respectively)
Example

float x, y, z

(e₁, e₂, e₃ point to symbol table entries for variables z, y, x, respectively)
Example

\[ \text{float } x, y, z \]

(e₁, e₂, e₃ point to symbol table entries for variables z, y, x, respectively)
Example

float x, y, z

(e₁, e₂, e₃ point to symbol table entries for variables z, y, x, respectively)
Example

float x, y, z

(e₁, e₂, e₃ point to symbol table entries for variables z, y, x, respectively)
But what about cycles?

- For a given attribute grammar — hard to detect if it has cyclic dependencies
  - Exponential cost

- Special classes of attribute grammars
  - Our “usual trick”:
    - sacrifice generality for predictable performance
Synthesized vs. Inherited Attributes

- *Synthesized* attributes
  - Attributes whose values at a given node depend *only* on the attributes of its children (and itself)

- *Inherited* attributes
  - Attributes whose values at a given node depend *only* on the attributes of its parent and siblings

Attributes that don’t depend on anything (*e.g.* those of a token) — by convention, are classified as *synthesized* attributes.
Synthesized vs. Inherited Attributes

float x, y, z

<table>
<thead>
<tr>
<th>Prod.</th>
<th>Semantic Rule</th>
</tr>
</thead>
<tbody>
<tr>
<td>D → T L</td>
<td>L.in = T.type</td>
</tr>
<tr>
<td>T → int</td>
<td>T.type = integer</td>
</tr>
<tr>
<td>T → float</td>
<td>T.type = float</td>
</tr>
<tr>
<td>L → L₁, id</td>
<td>L₁.in = L.in (\text{addType(id.entry, L.in)})</td>
</tr>
<tr>
<td>L → id</td>
<td>addType(id.entry, L.in)</td>
</tr>
</tbody>
</table>
Synthesized vs. Inherited Attributes

float x, y, z

- **Prod.** | **Semantic Rule**
  - D → T L | L.in = T.type
  - T → int | T.type = integer
  - T → float | T.type = float
  - L → L₁, id | L₁.in = L.in
                 | addType(id.entry, L.in)
  - L → id | addType(id.entry, L.in)
Synthesized vs. Inherited Attributes

float x, y, z

Prod. | Semantic Rule
--- | ---
D → T L | L.in = T.type
T → int | T.type = integer
T → float | T.type = float
L → L₁, id | L₁.in = L.in
     | addType(id.entry, L.in)
L → id | addType(id.entry, L.in)

inherited
synthesized
Synthesized vs. Inherited Attributes

float x, y, z

<table>
<thead>
<tr>
<th>Prod.</th>
<th>Semantic Rule</th>
</tr>
</thead>
<tbody>
<tr>
<td>D → T L</td>
<td>L.in = T.type</td>
</tr>
<tr>
<td>T → int</td>
<td>T.type = integer</td>
</tr>
<tr>
<td>T → float</td>
<td>T.type = float</td>
</tr>
<tr>
<td>L → L₁, id</td>
<td>L₁.in = L.in addType(id.entry, L.in)</td>
</tr>
<tr>
<td>L → id</td>
<td>addType(id.entry, L.in)</td>
</tr>
</tbody>
</table>

- **Inherited**: orange arrows
- **Synthesized**: green arrow
Synthesized vs. Inherited Attributes

float x, y, z

Prod. | Semantic Rule
--- | ---
D → T L | L.in = T.type
T → int | T.type = integer
T → float | T.type = float
L → L₁, id | L₁.in = L.in
| | addType(id.entry, L.in)
L → id | addType(id.entry, L.in)

inherited

synthesized
Synthesized vs. Inherited Attributes

float x, y, z

Prod. | Semantic Rule
--- | ---
D → T L  | L.in = T.type
T → int  | T.type = integer
T → float | T.type = float
L → L₁, id | L₁.in = L.in
            | addType(id.entry, L.in)
L → id    | addType(id.entry, L.in)

- inherited
- synthesized
Synthesized vs. Inherited Attributes

```
float x, y, z
```

<table>
<thead>
<tr>
<th>Prod.</th>
<th>Semantic Rule</th>
</tr>
</thead>
<tbody>
<tr>
<td>$D \rightarrow T L$</td>
<td>$L.in = T.type$</td>
</tr>
<tr>
<td>$T \rightarrow int$</td>
<td>$T.type = integer$</td>
</tr>
<tr>
<td>$T \rightarrow float$</td>
<td>$T.type = float$</td>
</tr>
</tbody>
</table>
| $L \rightarrow L_1, id$ | $L_1.in = L.in$
|                        | addType(id.entry, L.in)                        |
| $L \rightarrow id$ | addType(id.entry, L.in)                         |

**Inherited**

**Synthesized**

69
Synthesized vs. Inherited Attributes

float x, y, z

<table>
<thead>
<tr>
<th>Prod.</th>
<th>Semantic Rule</th>
</tr>
</thead>
<tbody>
<tr>
<td>D → T L</td>
<td>L.in = T.type</td>
</tr>
<tr>
<td>T → int</td>
<td>T.type = integer</td>
</tr>
<tr>
<td>T → float</td>
<td>T.type = float</td>
</tr>
</tbody>
</table>
| L → L1, id | L1.in = L.in
addType(id.entry, L.in) |
| L → id     | addType(id.entry, L.in)                           |

Inherited

Synthesized
S-attributed Grammars

- Special class of attribute grammars
- Only uses synthesized attributes (S-attributed)
  - No use of inherited attributes

- Can be computed by any bottom-up parser during parsing — no need to construct dependency graph
  - Attributes can be stored on the parsing stack
  - Reduce operation computes the (synthesized) attribute from attributes of children
S-attributed Grammars

Arithmetic Calculator

<table>
<thead>
<tr>
<th>Production</th>
<th>Semantic Rule</th>
</tr>
</thead>
<tbody>
<tr>
<td>S → E ;</td>
<td>print(E.val)</td>
</tr>
<tr>
<td>E → E₁ + T</td>
<td>E.val = E₁.val + T.val</td>
</tr>
<tr>
<td>E → T</td>
<td>E.val = T.val</td>
</tr>
<tr>
<td>T → T₁ * F</td>
<td>T.val = T₁.val * F.val</td>
</tr>
<tr>
<td>T → F</td>
<td>T.val = F.val</td>
</tr>
<tr>
<td>F → ( E )</td>
<td>F.val = E.val</td>
</tr>
<tr>
<td>F → digit</td>
<td>F.val = digit.lexval</td>
</tr>
</tbody>
</table>
S-attributed Grammars

Arithmetic Calculator

Diagram of an arithmetic expression: S → E * T + T → F * F → 7 + 4 * 3
S-attributed Grammars

Arithmetic Calculator
S-attributed Grammars

Arithmetic Calculator

- S
  - E
    - E
      - T
        - F
          - 7
            - Lexval=7
        - *
    - +
  - E
    - F
      - 4
      - 3
S-attributed Grammars

Arithmetic Calculator

```
S → E + E
E → T E
E → F
T → F T
T → 7
F → 4
F → 3
```

Lexval=7
val=7
val=7
S-attributed Grammars

Arithmetic Calculator

```
S → E
E → E + T
E → T
T → T * F
T → F
F → 7
F → 4
F → 3
```

Val: 7
Lexval: 7

Val: 4
Lexval: 4

Val: 3
Lexval: 7
S-attributed Grammars

Arithmetic Calculator
S-attributed Grammars

Arithmetic Calculator
S-attributed Grammars

Arithmetic Calculator

\[
S \rightarrow T \cdot F \\
T \rightarrow T \cdot F \\
F \rightarrow 7 | 4 | 3
\]

Lexval = 7

val = 7

val = 28

val = 28

val = 4
S-attributed Grammars

Arithmetic Calculator

- S
  - E
    - T
      - F
        - 7, Lexval=7
      - F
        - 4, Lexval=4
    - T
      - F
        - 3, Lexval=3
  - +
    - val=28
  - val=28
S-attributed Grammars

Arithmetic Calculator
S-attributed Grammars

Arithmetic Calculator

The diagram represents an S-attributed Grammar for an arithmetic calculator. Each node in the tree corresponds to a non-terminal symbol of the grammar, and each leaf node represents a terminal symbol. The labels on the nodes indicate the values associated with each symbol.

- **S** (root) -> **E**
- **E** -> **E** + **T**
- **E** -> **F**
- **T** -> **T** * **F**
- **T** -> 7
- **F** -> 4
- **F** -> 3
- **F** -> **E**
- **F** -> **T**

The values associated with the symbols are as follows:
- **val** = 7
- **val** = 28
- **val** = 3
- **Lexval** = 7
- **Lexval** = 4
- **Lexval** = 3

This structure allows for the evaluation of arithmetic expressions based on the rules defined by the grammar.
S-attributed Grammars

Arithmetic Calculator
S-attributed Grammars

Arithmetic Calculator

```
7 + 4 * 3
```

Val = 31

[Diagram showing the S-attributed Grammar for the arithmetic expression 7 + 4 * 3, with nodes labeled with their values (val) and lexical values (Lexval)].
L-attributed Grammars

- L-attributed attribute grammar: when every attribute in a production $A \rightarrow X_1...X_n$ is either
  - A synthesized attribute, or
  - An inherited attribute of $X_j$, $1 \leq j \leq n$ that only depends on
    - Attributes of $X_1...X_{j-1}$ to the left of $X_j$
    - Inherited attributes of $A$
L-attributed Grammars

- In recursive-descent parsers:
  - Pass inherited attributes down as arguments

<table>
<thead>
<tr>
<th>Prod.</th>
<th>Semantic Rule</th>
</tr>
</thead>
<tbody>
<tr>
<td>D → T L</td>
<td>L.in = T.type</td>
</tr>
<tr>
<td>T → int</td>
<td>T.type = integer</td>
</tr>
<tr>
<td>T → float</td>
<td>T.type = float</td>
</tr>
<tr>
<td>L → id A</td>
<td>A.in = L.in</td>
</tr>
<tr>
<td></td>
<td>addType(id.entry, L.in)</td>
</tr>
<tr>
<td>A → , L</td>
<td>L.in = A.in</td>
</tr>
<tr>
<td>A → ε</td>
<td></td>
</tr>
</tbody>
</table>

```java
d() {
    T();
    L();
}

L() {
    match(ID);
    A();
}

A() {
    if (current == COMMA) {
        match(COMMA);
        L();
    } else if (current == EOF) {
    } else error();
}

t() {
    if (current == INT) {
        match(INT);
    } else if (current == FLOAT) {
        match(float);
    } else error();
}
```
L-attributed Grammars

- In **recursive-descent** parsers:
  - Pass inherited attributes down as arguments

```
D() {
    Attrs t = T();
    L({in↦t.type});
}
L(Attrs ih) {
    Token id = match(ID);
    addType(id.entry, ih.in);
    A({in↦ih.in});
}
A(Attrs ih) {
    if (current == COMMA) {
        match(COMMA);
        L({in↦ih.in});
    } else if (current == EOF) {
    } else error();
}
T() {
    if (current == INT) {
        match(INT);
        return {type↦int};
    } else if (current == FLOAT) {
        match(float);
        return {type↦float};
    } else error();
}
```

<table>
<thead>
<tr>
<th>Prod.</th>
<th>Semantic Rule</th>
</tr>
</thead>
<tbody>
<tr>
<td>D → T L</td>
<td>L.in = T.type</td>
</tr>
<tr>
<td>T → int</td>
<td>T.type = integer</td>
</tr>
<tr>
<td>T → float</td>
<td>T.type = float</td>
</tr>
</tbody>
</table>
| L → id A | A.in = L.in
          | addType(id.entry, L.in)                           |
| A → , L | L.in = A.in                                       |
| A → ε  |                                                   |
L-attributed Grammars

- In shift-reduce parsers:

<table>
<thead>
<tr>
<th>Prod.</th>
<th>Semantic Rule</th>
</tr>
</thead>
<tbody>
<tr>
<td>D → T L</td>
<td>L.in = T.type</td>
</tr>
<tr>
<td>T → int</td>
<td>T.type = integer</td>
</tr>
<tr>
<td>T → float</td>
<td>T.type = float</td>
</tr>
<tr>
<td>L → L₁ , id</td>
<td>L₁.in = L.in addType(id.entry, L.in)</td>
</tr>
<tr>
<td>L → id</td>
<td>addType(id.entry, L.in)</td>
</tr>
</tbody>
</table>

Diagram:

- **q₀**: D → •T L
- **q₁**: D → T •L
- **q₂**: D → T L •L
- **q₃**: T → int •
- **q₄**: T → float •
- **q₅**: L → •L , id
- **q₆**: L → •L , id
- **q₇**: L → •L , id
L-attributed Grammars

- In shift-reduce parsers:

<table>
<thead>
<tr>
<th>Prod.</th>
<th>Semantic Rule</th>
</tr>
</thead>
<tbody>
<tr>
<td>D → T L</td>
<td>L.in = T.type</td>
</tr>
<tr>
<td>T → int</td>
<td>T.type = integer</td>
</tr>
<tr>
<td>T → float</td>
<td>T.type = float</td>
</tr>
<tr>
<td>L → L₁, id</td>
<td>L₁.in = L.in addType(id.entry, L.in)</td>
</tr>
<tr>
<td>L → id</td>
<td>addType(id.entry, L.in)</td>
</tr>
</tbody>
</table>

• In shift-reduce parsers:
L-attributed Grammars

- In shift-reduce parsers:

<table>
<thead>
<tr>
<th>Prod.</th>
<th>Semantic Rule</th>
</tr>
</thead>
<tbody>
<tr>
<td>D → T L</td>
<td>L.in = T.type</td>
</tr>
<tr>
<td>T → int</td>
<td>T.type = integer</td>
</tr>
<tr>
<td>T → float</td>
<td>T.type = float</td>
</tr>
<tr>
<td>L → L₁ , id</td>
<td>L₁.in = L.in addType(id.entry, L.in)</td>
</tr>
<tr>
<td>L → id</td>
<td>addType(id.entry, L.in)</td>
</tr>
</tbody>
</table>
L-attributed Grammars

- In shift-reduce parsers:
  - Use marker variables

<table>
<thead>
<tr>
<th>Prod.</th>
<th>Semantic Rule</th>
</tr>
</thead>
<tbody>
<tr>
<td>D → T M L</td>
<td>dtype = null</td>
</tr>
<tr>
<td>T → int</td>
<td>T.type = integer</td>
</tr>
<tr>
<td>T → float</td>
<td>T.type = float</td>
</tr>
<tr>
<td>L → L₁, id</td>
<td>addType(id.entry, dtype)</td>
</tr>
<tr>
<td>L → id</td>
<td>addType(id.entry, dtype)</td>
</tr>
<tr>
<td>M → ε</td>
<td>dtype = T.type</td>
</tr>
</tbody>
</table>
L-attributed Grammars

- In shift-reduce parsers:
  - Use marker variables

<table>
<thead>
<tr>
<th>Prod.</th>
<th>Semantic Rule</th>
</tr>
</thead>
<tbody>
<tr>
<td>D → TM L</td>
<td>dtype = null</td>
</tr>
<tr>
<td>T → int</td>
<td>T.type = integer</td>
</tr>
<tr>
<td>T → float</td>
<td>T.type = float</td>
</tr>
<tr>
<td>L → L₁, id</td>
<td>addType(id.entry, dtype)</td>
</tr>
<tr>
<td>L → id</td>
<td>addType(id.entry, dtype)</td>
</tr>
<tr>
<td>M → ε</td>
<td>dtype = T.type</td>
</tr>
</tbody>
</table>

Prod. M L
T

In shift-reduce parsers:
- Use marker variables
Marker Variables

• Since a marker only appear in one production rule, it is commonly abbreviated:

D → T M L \text{ action}_1

M → \varepsilon \text{ action}_2

• \text{ action}_2 \text{ is called a mid-rule action.}

It is important to remember that adding mid-rule actions inherently changes the grammar. The marker variable is there even if it is not explicitly visible.
Marker Variables

- In particular, marker variables and the associated \( \varepsilon \)-productions can violate your grammar’s LR(0)/SLR/LALR/LR(1)-ness

<table>
<thead>
<tr>
<th>Prod.</th>
<th>Semantic Rule</th>
</tr>
</thead>
<tbody>
<tr>
<td>( S \rightarrow D )</td>
<td></td>
</tr>
<tr>
<td>( D \rightarrow T \ L )</td>
<td></td>
</tr>
<tr>
<td>( D \rightarrow T \ L \ [ \ ] )</td>
<td></td>
</tr>
<tr>
<td>( T \rightarrow \text{int} )</td>
<td>( T.type = \text{integer} )</td>
</tr>
<tr>
<td>( T \rightarrow \text{float} )</td>
<td>( T.type = \text{float} )</td>
</tr>
<tr>
<td>( L \rightarrow \text{id} )</td>
<td>( \text{addType(id.entry, dtype)} )</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Prod.</th>
<th>Semantic Rule</th>
</tr>
</thead>
<tbody>
<tr>
<td>( S \rightarrow D )</td>
<td></td>
</tr>
<tr>
<td>( D \rightarrow T \ M \ L )</td>
<td></td>
</tr>
<tr>
<td>( D \rightarrow T \ N \ L \ [ \ ] )</td>
<td></td>
</tr>
<tr>
<td>( T \rightarrow \text{int} )</td>
<td>( T.type = \text{integer} )</td>
</tr>
<tr>
<td>( T \rightarrow \text{float} )</td>
<td>( T.type = \text{float} )</td>
</tr>
<tr>
<td>( L \rightarrow \text{id} )</td>
<td>( \text{addType(id.entry, dtype)} )</td>
</tr>
<tr>
<td>( M \rightarrow \varepsilon )</td>
<td>( dtype = T.type )</td>
</tr>
<tr>
<td>( N \rightarrow \varepsilon )</td>
<td>( dtype = \text{array}(T.type) )</td>
</tr>
</tbody>
</table>
Marker Variables

- In particular, marker variables and the associated ε-productions can violate your grammar’s LR(0)/SLR/LALR/LR(1)-ness

<table>
<thead>
<tr>
<th>Prod.</th>
<th>Semantic Rule</th>
</tr>
</thead>
<tbody>
<tr>
<td>S → D</td>
<td></td>
</tr>
<tr>
<td>D → T L</td>
<td></td>
</tr>
<tr>
<td>D → T L [ ]</td>
<td></td>
</tr>
<tr>
<td>T → int</td>
<td>T.type = integer</td>
</tr>
<tr>
<td>T → float</td>
<td>T.type = float</td>
</tr>
<tr>
<td>L → id</td>
<td>addType(id.entry, dtype)</td>
</tr>
</tbody>
</table>

- This grammar is LR(0)
Marker Variables

- In particular, marker variables and the associated \( \varepsilon \)-productions can violate your grammar’s LR(0)/SLR/LALR/LR(1)-ness

<table>
<thead>
<tr>
<th>Prod.</th>
<th>Semantic Rule</th>
</tr>
</thead>
<tbody>
<tr>
<td>( S \rightarrow D )</td>
<td></td>
</tr>
<tr>
<td>( D \rightarrow T \ L )</td>
<td></td>
</tr>
<tr>
<td>( D \rightarrow T \ L \ [ ] )</td>
<td></td>
</tr>
<tr>
<td>( T \rightarrow \text{int} )</td>
<td>( T\text{.type} = \text{integer} )</td>
</tr>
<tr>
<td>( T \rightarrow \text{float} )</td>
<td>( T\text{.type} = \text{float} )</td>
</tr>
<tr>
<td>( L \rightarrow \text{id} )</td>
<td>( \text{addType(id.entry, dtype)} )</td>
</tr>
</tbody>
</table>

This grammar is LR(0)

<table>
<thead>
<tr>
<th>Prod.</th>
<th>Semantic Rule</th>
</tr>
</thead>
<tbody>
<tr>
<td>( S \rightarrow D )</td>
<td></td>
</tr>
<tr>
<td>( D \rightarrow T \ M \ L )</td>
<td></td>
</tr>
<tr>
<td>( D \rightarrow T \ N \ L \ [ ] )</td>
<td></td>
</tr>
<tr>
<td>( T \rightarrow \text{int} )</td>
<td>( T\text{.type} = \text{integer} )</td>
</tr>
<tr>
<td>( T \rightarrow \text{float} )</td>
<td>( T\text{.type} = \text{float} )</td>
</tr>
<tr>
<td>( L \rightarrow \text{id} )</td>
<td>( \text{addType(id.entry, dtype)} )</td>
</tr>
<tr>
<td>( M \rightarrow \varepsilon )</td>
<td>( dtype = T\text{.type} )</td>
</tr>
<tr>
<td>( N \rightarrow \varepsilon )</td>
<td>( dtype = \text{array}(T\text{.type}) )</td>
</tr>
</tbody>
</table>

This grammar is not even LR(1)
Summary

✓ Contextual analysis can move information between nodes in the AST
  ‣ Even when they are not “local”

✓ Attribute grammars
  ‣ Attach attributes and semantic actions to grammar

✓ Attribute evaluation
  ‣ Build dependency graph, topological sort, evaluate

✓ Special classes with pre-determined evaluation order: S-attributed, L-attributed
Coming Up

Intermediate Representation