Phases of a compiler

Figure 1.6, page 5 of text
Semantics

• “Semantics” has to do with the meaning of a program.

• We will consider two types of semantics:
  – Static semantics: semantics which can be enforced at compile-time.
  – Dynamic semantics: semantics which express the run-time meaning of programs.
Static semantics

- Semantic checking which can be done at compile-time

- Type-compatibility is a prime example
  - \texttt{int} can be assigned to \texttt{double} (type coercion)
  - \texttt{double} cannot be assigned to \texttt{int} without explicit type cast

- Type-compatibility can be captured in grammar, but only at expense of larger, more complex grammar
Ex: adding type rules in grammar

- Must introduce new non-terminals which encode types:
- Instead of a generic grammar rule for assignment:
  - \(<\text{stmt}> \rightarrow <\text{var}> \text{ '='} <\text{expr}> \text{ ';'}\)
- we need multiple rules:
  - \(<\text{stmt}> \rightarrow <\text{doubleVar}> \text{ '='} <\text{intExpr}> | <\text{doubleExpr}> \text{ ';'}\)
  - \(<\text{stmt}> \rightarrow <\text{intVar}> \text{ '='} <\text{intExpr}> \text{ ';'}\)
- Of course, such rules need to handle all the relevant type possibilities (e.g. byte, char, short, int, long, float and double).
Alternative: attribute grammars

• Attribute grammars provide a neater way of encoding such information.

• Each syntactic rule of the grammar can be decorated with:
  – a set of semantic rules/functions
  – a set of semantic predicates
Attributes

- We can associate with each symbol X of the grammar a set of attributes A(X). Attributes are partitioned into:
  
  - synthesized attributes S(X) – pass info up parse tree
  - inherited attributes I(X) – pass info down parse tree
Semantic rules/functions

• We can associate with each rule R of the grammar a set of semantic functions.

• For rule \( x_0 \rightarrow x_1 \ x_2 \ \cdots \ x_n \)
  - synthesized attribute of LHS:
    \[ S(x_0) = f(A(x_1), A(x_2), \ldots, A(x_n)) \]

  - inherited attribute of RHS member:
    for \( 1 \leq j \leq n \), \[ I(x_j) = f(A(x_0), \ldots, A(x_{j-1})) \]
    (note that dependence is on siblings to left only)
Predicates

• We can associate with each rule R of the grammar a set of semantic predicates.

• Boolean expression involving the attributes and a set of attribute values

• If true, node is ok

• If false, node violates a semantic rule
Example

<assign> → <var> = <expr>

Start with a production of the grammar

Syntactic rule
Semantic rule/function
Semantic predicate
Example

<assign> → <var> = <expr>

<expr>.expType

Associate an attribute with a non-terminal, <expr>, on the right of the production: expType (the expected type of the expression)
Example

<assign> → <var> = <expr>
<expr>.expType ← <var>.actType

Assign to `<expr>.expType` the value of `<var>.actType`, the actual type of the variable (the type the variable was declared as).
Example

<assign> → <var> = <expr>
<expr>.expType ← <var>.actType

In other words, we expect the expression whose value is being assigned to a variable to have the same type as the variable.
Example

<assign> \rightarrow <var> = <expr>
<expr>.expType \leftarrow <var>.actType


Another grammar production

Syntactic rule
Semantic rule/function
Semantic predicate
Example

\[ \text{<assign>} \rightarrow \text{<var>} = \text{<expr>} \]
\[ \text{<expr>}.\text{expType} \leftarrow \text{<var>}.\text{actType} \]

\[ \text{<expr>} \rightarrow \text{<var>}[2] + \text{<var>}[3] \]
\[ \text{<expr>}.\text{actType} \leftarrow \text{if (var}[2].\text{actType} = \text{int}) \text{ and} \]
\[ \text{if (var}[3].\text{actType} = \text{int}) \]
\[ \text{then int} \]
\[ \text{else real} \]

This production has a more involved semantic rule: it handles type coercion. This rule assume that there are only two numeric types (int and real) and that int can be coerced to real.
Here is our first semantic predicate, which enforces a type-checking constraint: the actual type of `<expr>` must match the expected type (from elsewhere in the tree).
Example

<assign> → <var> = <expr>
<expr>.expType ← <var>.actType

<expr>.actType ← if (var[2].actType = int) and
    (var[3].actType = int)
    then int
    else real
<expr>.actType == <expr>.expType

<expr> → <var>
<expr>.actType ← <var>.actType
<expr>.actType == <expr>.expType

Another production, with a semantic rule and a semantic predicate.
Example

\[
<\text{assign}> \rightarrow <\text{var}> = <\text{expr}>
\]
\[
<\text{expr}\.expType \leftarrow <\text{var}\.actType
\]

\[
<\text{expr}> \rightarrow <\text{var}>[2] + <\text{var}>[3]
\]
\[
<\text{expr}\.actType \leftarrow \text{if} \ (\text{var}[2].actType = \text{int}) \ \text{and}
\]
\[
\quad \ (\text{var}[3].actType = \text{int})
\]
\[
\quad \ \text{then int}
\]
\[
\quad \ \text{else real}
\]
\[
<\text{expr}\.actType == <\text{expr}\.expType
\]

\[
<\text{expr}> \rightarrow <\text{var}>
\]
\[
<\text{expr}\.actType \leftarrow <\text{var}\.actType
\]
\[
<\text{expr}\.actType == <\text{expr}\.expType
\]

\[
<\text{var}> \rightarrow A \mid B \mid C
\]
\[
<\text{var}\.actType \leftarrow \text{lookUp}(<\text{var}\.string)
\]

This semantic rule says that the type of an identifier is determined by looking up its type in the symbol table.
All the productions, rules and predicates

<assign>  \rightarrow  <var>  =  <expr>
<expr>.expType  \leftarrow  <var>.actType

<expr>.actType  \leftarrow  \text{if}  (\text{var}[2].actType = \text{int})  \text{ and}
                     (\text{var}[3].actType = \text{int})
                   \text{then int}
                   \text{else real}
<expr>.actType  ==  <expr>.expType

<expr>  \rightarrow  <var>
<expr>.actType  \leftarrow  <var>.actType
<expr>.actType  ==  <expr>.expType

<var>  \rightarrow  A  |  B  |  C
<var>.actType  \leftarrow  \text{lookUp}(<var>.string)
Suppose:

- A is int
- B is int

Let's see how these rules work in practice!

In this example A and B are both of type int.
Suppose:
A is int
B is int

\[ A = A + B \]

Effects of the syntactic rules is shown in red.
Suppose:

A is real
B is int
Suppose:

- A is real
- B is int

Type coercion during `+`:

```
int → real
```
This is the same example structure, but now assume A is of type int and B is of type real.
Suppose:
A is int
B is real

Houston, we have a problem!
Semantic predicate is false.