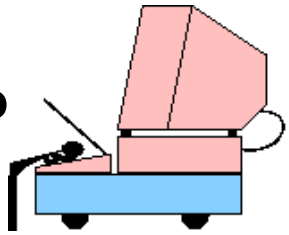
Práctica 2: Análisis semántico



Especificación del analizador semántico

Tal y como se realiza en un compilador habitualmente, en la fase de análisis semántico se debe comprobar que el programa fuente se ajusta a todas las especificaciones del lenguaje de programación fuente que no hayan podido ser comprobadas en la fase de análisis sintáctico. A estos efectos, se debe entender que el enunciado de la práctica 1 forma parte de dichas especificaciones, y por lo tanto, el analizador semántico deberá comprobar que el programa fuente se ajusta a todo lo indicado en dicho enunciado y que no se haya verificado en la fase de análisis sintáctico.

Esto aplica particularmente a las comprobaciones de tipos de las expresiones. Las expresiones que combinan una o 2 subexpresiones se deberán ajustar a lo indicado en la práctica 1. Por ejemplo, de acuerdo con el enunciado de la práctica 1, una expresión que combine una subexpresión de tipo `int` con otra subexpresión de tipo `bool` por medio del operador '+' debería producir un error de tipo, detectado en la fase de análisis semántico.

Además el analizador semántico deberá comprobar lo siguiente:

- Todas las variables que se utilizan en el cuerpo del programa deberán haber sido declaradas. VarNotDefExc
- Ninguna variable deberá ser declarada 2 veces. VarDefTwiceExc
- En una sentencia de asignación, el tipo de la variable en la parte izquierda de la asignación y el tipo de la expresión en la parte derecha debe ser el mismo.
- En una sentencia `printInt`, el tipo de la variable asociada debe ser `int`.
- En una sentencia `printBool`, el tipo de la variable asociada debe ser `bool`.
- En una sentencia condicional (`if`), la expresión debe ser de tipo `bool`, mientras que las sentencias que se ejecutan si se cumple la condición no deben tener error de tipo.
- En una sentencia bucle (`while`), la expresión debe ser de tipo `bool`, mientras que las sentencias que forman el cuerpo del bucle no deben tener error de tipo.

Los atributos `ah1`, `typ`, `l_typ` y `st_typ`, cuya especificación se presenta a continuación, realizan todas las comprobaciones que debe hacer el analizador semántico. Se asume que `newEntry` se encarga de añadir una entrada a la tabla de símbolos y que `getType` se encarga de devolver el tipo de una variable cuyo nombre se pasa como argumento, y que estos métodos levantan una excepción cuando se consulta una variable que no ha sido declarada o se inserta una misma variable dos veces. Asimismo, se supone que cuando en la especificación de los atributos `typ` o `st_typ` se devuelve `typ_err` esto equivale en la implementación en Java del analizador semántico a levantar una excepción debido a que se ha producido un error de tipo. Como consecuencia de esto, los métodos Java que implementan los atributos `l_typ` y `st_typ` devuelven `void`.

Regla de producción	Atributos
`S -> PROG IDENT DECL <LDecl> <Body>`	
`S -> PROG IDENT <Body>`	
`<LDecl> -> <Decl> PC`	computeAH1
`<LDecl> -> <Decl> PC <LDecl>`$_1$	computeAH1
`<Decl> -> TYPE <IdentList>`	`<IdentList>.ah1= TYPE.val`
`<IdentList> -> IDENT`	`newEntry(IDENT.val, <IdentList>.ah1)`

<IdentList> -> IDENT COMA <IdentList>$_1$	<IdentList>$_1$.ah1= <IdentList>.ah1 newEntry(IDENT.val, <IdentList>.ah1)
<Body> -> BEGIN <StatementList> END PC	<Body>.st_typ= <StatementList>.st_typ
<StatementList> -> <Statement> PC	<StatementList>.st_typ= <Statement>.st_typ
<StatementList> -> <Statement> PC <StatementList>$_1$	<StatementList>.st_typ= if (<StatementList>$_1$.st_typ== void) and (<Statement>.st_typ==void) then void else typ_err
1 <Statement> -> IDENT ASOP <Exp>	<Statement>.st_typ= if (getType(IDENT.val)== <Exp>.typ) then void else typ_err
2 <Statement> -> PRINT_I PAREN IDENT TESIS	<Statement>.st_typ= if (getType(IDENT.val)== int) then void else typ_err
3 <Statement> -> PRINT_B PAREN IDENT TESIS	<Statement>.st_typ= if (getType(IDENT.val)== bool) then void else typ_err
4 <Statement> -> IF <Exp> THEN <StatementList> END	<Statement>.st_typ= if (<Exp>.typ== bool) and (<StatementList>.st_typ== void) then void else typ_err
5 <Statement> -> WHILE <Exp> DO <StatementList> END	<Statement>.st_typ= if (<Exp>.typ== bool) and (<StatementList>.st_typ== void) then void else typ_err
<Exp> -> IDENT	<Exp>.typ= getType(IDENT.val) variable
<Exp> -> PAREN <Exp>$_1$ TESIS	<Exp>.typ= <Exp>$_1$.typ
<Exp> -> CINT	<Exp>.typ= int
<Exp> -> <Exp>$_1$ MAS <Exp>$_2$	<Exp>.typ= if (<Exp>$_1$.typ== int) and (<Exp>$_2$.typ== int) then int else typ_err
<Exp> -> <Exp>$_1$ MENOS <Exp>$_2$	<Exp>.typ= if (<Exp>$_1$.typ== int) and (<Exp>$_2$.typ== int) then int else if (<Exp>$_1$.typ== intset) and (<Exp>$_2$.typ== intset) then intset else typ_err
<Exp> -> <Exp>$_1$ MUL <Exp>$_2$	<Exp>.typ= if (<Exp>$_1$.typ== int) and (<Exp>$_2$.typ== int) then int else typ_err
<Exp> -> <Exp>$_1$ DIV <Exp>$_2$	<Exp>.typ= if (<Exp>$_1$.typ== int) and (<Exp>$_2$.typ== int) then int else typ_err
<Exp> -> MENOS <Exp>$_1$	<Exp>.typ= if (<Exp>$_1$.typ== int) then int else typ_err
<Exp> -> <Exp>$_1$ IN <Exp>$_2$	<Exp>.typ= if (<Exp>$_1$.typ== int) and (<Exp>$_2$.typ== intset) then bool else typ_err
<Exp> -> BRAC <ExpList> KET	<Exp>.typ= if (<ExpList>.l_typ== void) then intset else typ_err
<Exp> -> BRAC KET	<Exp>.typ= intset
<Exp> -> EMPTYSET	<Exp>.typ= intset
<Exp> -> <Exp>$_1$ UNION <Exp>$_2$	<Exp>.typ= if (<Exp>$_1$.typ== intset) and (<Exp>$_2$.typ== intset) then intset else typ_err
<Exp> -> <Exp>$_1$ INTERSEC <Exp>$_2$	<Exp>.typ= if (<Exp>$_1$.typ== intset) and (<Exp>$_2$.typ== intset) then intset else typ_err
<Exp> -> CARD PAREN <Exp>$_1$ TESIS	<Exp>.typ= if (<Exp>$_1$.typ== intset) then int else typ_err
<Exp> -> LOWEST_ELEM PAREN <Exp>$_1$ TESIS	<Exp>.typ= if (<Exp>$_1$.typ== intset) then int else typ_err
<Exp> -> CLOG	<Exp>.typ= bool
<Exp> -> <Exp>$_1$ AND <Exp>$_2$	<Exp>.typ= if (<Exp>$_1$.typ== bool) and (<Exp>$_2$.typ== bool) then bool else typ_err
<Exp> -> <Exp>$_1$ OR <Exp>$_2$	<Exp>.typ= if (<Exp>$_1$.typ== bool) and (<Exp>$_2$.typ== bool) then bool else typ_err
<Exp> -> NOT <Exp>$_1$	<Exp>.typ= if (<Exp>$_1$.typ== bool) then bool else typ_err
<Exp> -> <Exp>$_1$ IGUAL <Exp>$_2$	<Exp>.typ= if (<Exp>$_1$.typ== <Exp>$_2$.typ) then bool else typ_err

<Exp> -> <Exp>$_1$ MAYOR <Exp>$_2$	<Exp>.typ= if ((<Exp>$_1$.typ== int) and (<Exp>$_2$.typ== int)) then bool else typ_err
<Exp> -> <Exp>$_1$ MENOR <Exp>$_2$	<Exp>.typ= if ((<Exp>$_1$.typ== int) and (<Exp>$_2$.typ== int)) then bool else typ_err
<ExpList> -> <Exp>	<ExpList>.l_typ= if (<Exp>.typ== int) then void else typ_err
<ExpList> -> <Exp> COMA <ExpList>$_1$	<ExpList>.l_typ= if ((<Exp>.typ== int) and (<ExpList>$_1$== void)) then void else typ_err

El analizador semántico debe lanzar una excepción, que no debe ser capturada, cuando se detecte un error semántico. La clase o clases utilizadas para las excepciones correspondientes a errores semánticos se pueden definir libremente, pero deben extender la clase `CompilerExc`, que se proporcionó en la práctica 1. Se valorará que los mensajes que se generen identiifiquen claramente el error producido.

Requisitos que deben cumplir los ficheros JLex y CUP

Para esta práctica se deberá desarrollar una clase `Main` (puede utilizar como punto de partida la proporcionada en la práctica 1). La ejecución de dicha clase se realizará de acuerdo con lo siguiente:

```
java Main <nombre_fichero>
```

Donde `<nombre_fichero>` es el nombre del fichero (con el path si es necesario) del programa fuente a analizar.

El programa deberá de levantar una excepción que no deberá ser capturada en el caso de que el programa fuente tenga algún error léxico, sintáctico o semántico.

Obligatoriamente las clases generadas por CUP deberán pertenecer a un paquete llamado Parser y la clase generada por JLex deberá pertenecer a un paquete llamado Lexer.

Las clases Java que desarrolle en esta práctica **obligatoriamente** deberán estar organizadas en los siguientes paquetes:

- Paquete `Errors`, que debe contener todas las excepciones utilizadas (incluidas las clases proporcionadas en la práctica 1). Si las excepciones utilizasen alguna clase auxiliar, dicha clase deberá pertenecer asimismo a ese paquete.
- Paquete `AST`, contendrá todas las clases necesarias para representar árboles de sintaxis abstracta correspondientes a programas del lenguaje de programación fuente.
- Paquete `Compiler`, contendrá cualquier otra clase que necesite, incluyendo las de la tabla de símbolos.

La clase `Main` no pertenecerá a ningún paquete.

Orden de compilación

Antes de entregar la práctica deberá cerciorarse de que su práctica puede compilarse correctamente con `javac` siguiendo el siguiente orden:

1. Clases del paquete `Errors`.
2. Clases del paquete `Compiler`.
3. Clases del paquete `AST`.
4. Clases `parser` y `sym` generadas por CUP.
5. Clase `Yylex` generada por JLex.
6. Clase `Main`.

Aquellas prácticas que no se puedan compilar en este orden serán calificadas con 0.

Ayudas y sugerencias

Se proporciona un juego de tests con el que probar la práctica. De entre ellos solamente deberían de producir un error los tests en carpetas cuyo nombre comienza por `ErrSem`, `ErrSint` y `ErrLex`. Puede ocurrir que alguno de los ejemplos `ErrLex` de error de sintaxis (y no léxico). Los tests de las carpetas cuyo nombre comienza por `ErrSem` deberían producir un error semántico.

Se advierte que se realizarán tests adicionales a los proporcionados a las prácticas recibidas, por lo que se recomienda a los alumnos que planifiquen tests complementarios a su práctica.

Ficheros a entregar

Se deberán entregar exclusivamente los siguientes ficheros:

- fichero `Yylex` en formato JLex para el analizador léxico
- fichero `parser` en formato CUP para el analizador sintáctico
- fichero `java.zip`, que al descomprimirlo genere una carpeta de nombre `java` cuyo contenido debe ser **exactamente** el siguiente:

  hay que modificarlo para añadir el analisis semantico

  - fichero `Main.java` que contenga la clase `Main`.
  - Carpeta `Errors` que contenga las clases Java del paquete `Errors` (proporcionar ficheros Java, no ficheros `.class`).
  - Carpeta `Compiler` que contenga las clases Java del paquete `Compiler` (proporcionar ficheros Java, no ficheros `.class`).

COMPILER
tabla de simbolos
identificar tipos

Paquete compiler con al menos la tabla de simbolos, y ahi podemos poner una clase para identificar los simbolos

  - Carpeta `AST` que contenga las clases Java del paquete `AST` (proporcionar ficheros Java, no ficheros `.class`).

Last Revision: 03/13/2024 10:27:37