Computer Languages

Activation records

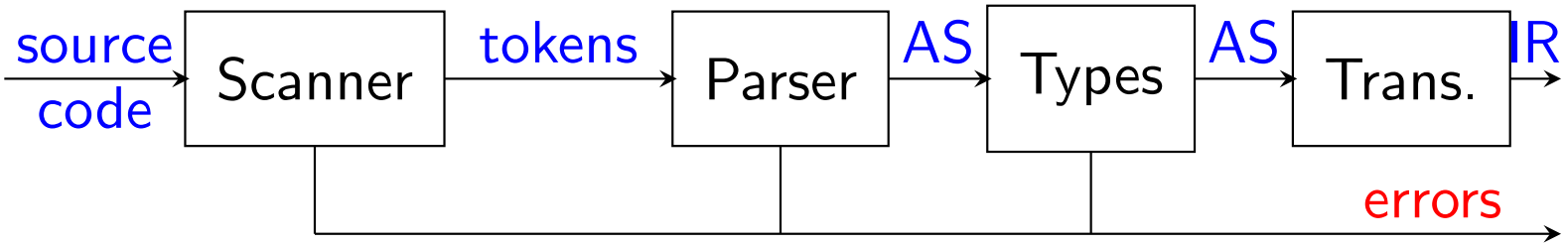
Verónica Gaspes

School of Information Science, Computer and Electrical Engineering

February 24

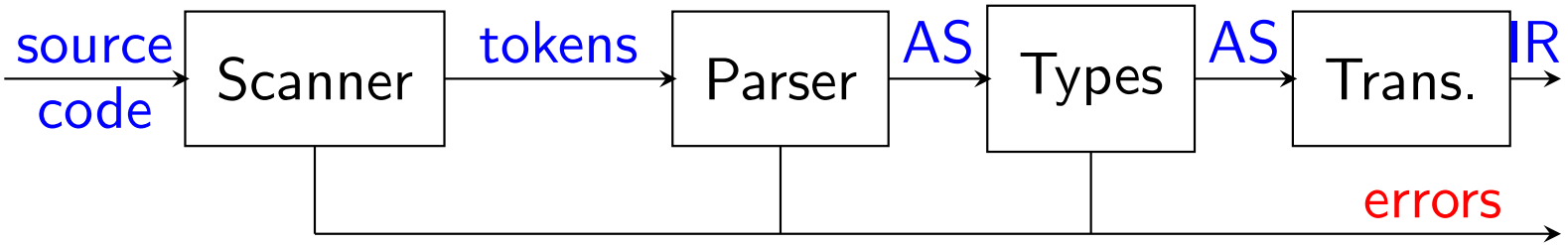
www2.hh.se/staff/jebe/languages

The Front End (analysis phase)



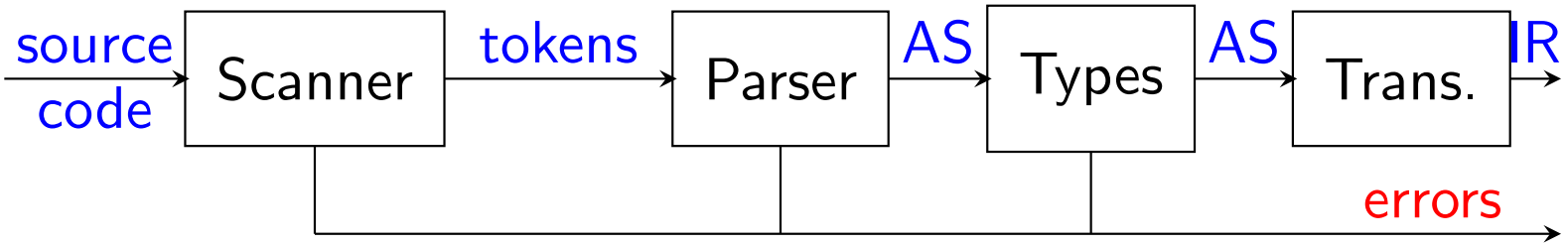
- The Scanner (lexical analyzer) transforms a sequence of characters (source code) into a sequence of tokens: a representation of the *lexemes* of the language.
- The Parser (syntactical analyzer) takes the sequence of tokens and generates a tree representation, the Abstract Syntax.
- This tree is analyzed by the type checker and is then used to generate the intermediate representation.

The Front End (analysis phase)



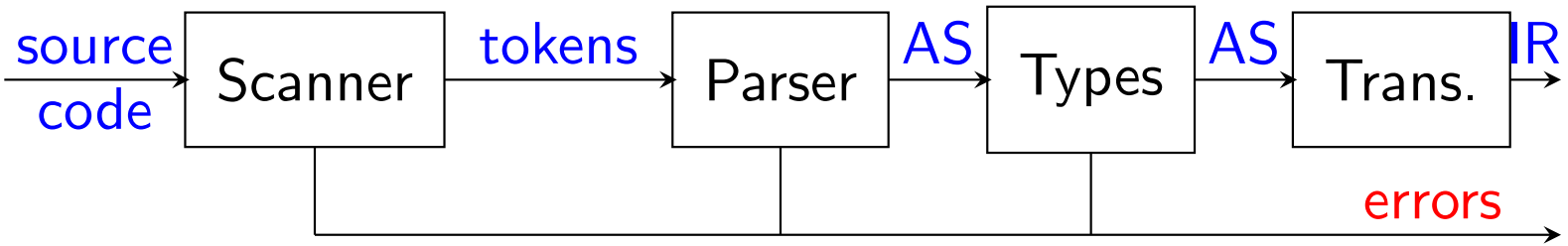
- The Scanner (lexical analyzer) transforms a sequence of characters (source code) into a sequence of tokens: a representation of the *lexemes* of the language.
- The Parser (syntactical analyzer) takes the sequence of tokens and generates a tree representation, the Abstract Syntax.
- This tree is analyzed by the type checker and is then used to generate the intermediate representation.

The Front End (analysis phase)



- The Scanner (lexical analyzer) transforms a sequence of characters (source code) into a sequence of tokens: a representation of the *lexemes* of the language.
- The Parser (syntactical analyzer) takes the sequence of tokens and generates a tree representation, the Abstract Syntax.
- This tree is analyzed by the type checker and is then used to generate the intermediate representation.

The Front End (analysis phase)



- The Scanner (lexical analyzer) transforms a sequence of characters (source code) into a sequence of tokens: a representation of the *lexemes* of the language.
- The Parser (syntactical analyzer) takes the sequence of tokens and generates a tree representation, the Abstract Syntax.
- This tree is analyzed by the type checker and is then used to generate the intermediate representation.

The translation phase

- Lexical, syntactical and contextual analysis are phases of *all* language processors of *all* computer languages (which we have exemplified with *minijava*).
- The translation phase is more dependent on the kind of computer language being processed.
- For *programming languages* it is taking a step towards executable code.
- We will present some general techniques and abstractions for doing so.

Note

We will exemplify with *minijava* when possible. If *minijava* doesn't have the feature we want to illustrate we will refer to some other programming language.

The translation phase

- Lexical, syntactical and contextual analysis are phases of *all* language processors of *all* computer languages (which we have exemplified with *minijava*).
- The translation phase is more dependent on the kind of computer language being processed.
- For *programming languages* it is taking a step towards executable code.
- We will present some general techniques and abstractions for doing so.

Note

We will exemplify with *minijava* when possible. If *minijava* doesn't have the feature we want to illustrate we will refer to some other programming language.

The translation phase

- Lexical, syntactical and contextual analysis are phases of *all* language processors of *all* computer languages (which we have exemplified with *minijava*).
- The translation phase is more dependent on the kind of computer language being processed.
- For *programming languages* it is taking a step towards executable code.
- We will present some general techniques and abstractions for doing so.

Note

We will exemplify with *minijava* when possible. If *minijava* doesn't have the feature we want to illustrate we will refer to some other programming language.

The translation phase

- Lexical, syntactical and contextual analysis are phases of *all* language processors of *all* computer languages (which we have exemplified with *minijava*).
- The translation phase is more dependent on the kind of computer language being processed.
- For *programming languages* it is taking a step towards executable code.
- We will present some general techniques and abstractions for doing so.

Note

We will exemplify with *minijava* when possible. If *minijava* doesn't have the feature we want to illustrate we will refer to some other programming language.

The translation phase

- Lexical, syntactical and contextual analysis are phases of *all* language processors of *all* computer languages (which we have exemplified with *minijava*).
- The translation phase is more dependent on the kind of computer language being processed.
- For *programming languages* it is taking a step towards executable code.
- We will present some general techniques and abstractions for doing so.

Note

We will exemplify with *minijava* when possible. If *minijava* doesn't have the feature we want to illustrate we will refer to some other programming language.

The translation phase

- Lexical, syntactical and contextual analysis are phases of *all* language processors of *all* computer languages (which we have exemplified with *minijava*).
- The translation phase is more dependent on the kind of computer language being processed.
- For *programming languages* it is taking a step towards executable code.
- We will present some general techniques and abstractions for doing so.

Note

We will exemplify with *minijava* when possible. If *minijava* doesn't have the feature we want to illustrate we will refer to some other programming language.

A meaningful *minijava*-fragment

```
class Visitor {
  Tree l ;
  Tree r ;
  public int visit(Tree n){
    int nti ;
    if (n.GetHas_Right()){
      r = n.GetRight() ;
      nti = r.accept(this) ; }
    else nti = 0 ;
    if (n.GetHas_Left()) {
      l = n.GetLeft();
      nti = l.accept(this) ; }
    else nti = 0 ;
    return 0;
  }
}
```

Executable code is only in the body of a method!

What happens when a program that uses `Visitor` is executed?

Instances of `Visitor` will have their own values for `l` and `r`.

Activations of `visit` will have their own values for `n` and `nti`.

A meaningful *minijava*-fragment

```
class Visitor {
  Tree l ;
  Tree r ;
  public int visit(Tree n){
    int nti ;
    if (n.GetHas_Right()){
      r = n.GetRight() ;
      nti = r.accept(this) ; }
    else nti = 0 ;
    if (n.GetHas_Left()) {
      l = n.GetLeft();
      nti = l.accept(this) ; }
    else nti = 0 ;
    return 0;
  }
}
```

Executable code is only in the body of a method!

What happens when a program that uses `Visitor` is executed?

Instances of `Visitor` will have their own values for `l` and `r`.

Activations of `visit` will have their own values for `n` and `nti`.

A meaningful *minijava*-fragment

```
class Visitor {
  Tree l ;
  Tree r ;
  public int visit(Tree n){
    int nti ;
    if (n.GetHas_Right()){
      r = n.GetRight() ;
      nti = r.accept(this) ; }
    else nti = 0 ;
    if (n.GetHas_Left()) {
      l = n.GetLeft();
      nti = l.accept(this) ; }
    else nti = 0 ;
    return 0;
  }
}
```

Executable code is only in the body of a method!

What happens when a program that uses `Visitor` is executed?

Instances of `Visitor` will have their own values for `l` and `r`.

Activations of `visit` will have their own values for `n` and `nti`.

A meaningful *minijava*-fragment

```
class Visitor {
  Tree l ;
  Tree r ;
  public int visit(Tree n){
    int nti ;
    if (n.GetHas_Right()){
      r = n.GetRight() ;
      nti = r.accept(this) ; }
    else nti = 0 ;
    if (n.GetHas_Left()) {
      l = n.GetLeft();
      nti = l.accept(this) ; }
    else nti = 0 ;
    return 0;
  }
}
```

Executable code is only in the body of a method!

What happens when a program that uses `Visitor` is executed?

Instances of `Visitor` will have their own values for `l` and `r`.

Activations of `visit` will have their own values for `n` and `nti`.

A meaningful *minijava*-fragment

```
class Visitor {
  Tree l ;
  Tree r ;
  public int visit(Tree n){
    int nti ;
    if (n.GetHas_Right()){
      r = n.GetRight() ;
      nti = r.accept(this) ; }
    else nti = 0 ;
    if (n.GetHas_Left()) {
      l = n.GetLeft();
      nti = l.accept(this) ; }
    else nti = 0 ;
    return 0;
  }
}
```

Executable code is only in the body of a method!

What happens when a program that uses `Visitor` is executed?

Instances of `Visitor` will have their own values for `l` and `r`.

Activations of `visit` will have their own values for `n` and `nti`.

Understanding a method...
...takes a lot of effort from the compiler

Code ...

- ... has to be translated into machine code that can be loaded to instruction memory. Labels are usually used to mark the first instruction of this code fragment.
- All variables used in the code have to be elaborated to the right relative address.

Data memory

- A frame of memory has to be designed so that on every activation of the method a reasonable portion of memory can be claimed!
- This memory slots have a standarized structure that implements the calling convention.

Understanding a method...
...takes a lot of effort from the compiler

Code ...

- ...has to be translated into machine code that can be loaded to instruction memory. Labels are usually used to mark the first instruction of this code fragment.
- All variables used in the code have to be elaborated to the right relative address.

Data memory

- A frame of memory has to be designed so that on every activation of the method a reasonable portion of memory can be claimed!
- This memory slots have a standarized structure that implements the calling convention.

Understanding a method...
...takes a lot of effort from the compiler

Code ...

- ...has to be translated into machine code that can be loaded to instruction memory. Labels are usually used to mark the first instruction of this code fragment.
- All variables used in the code have to be elaborated to the right relative address.

Data memory

- A frame of memory has to be designed so that on every activation of the method a reasonable portion of memory can be claimed!
- This memory slots have a standarized structure that implements the calling convention.

Understanding a method...
...takes a lot of effort from the compiler

Code ...

- ...has to be translated into machine code that can be loaded to instruction memory. Labels are usually used to mark the first instruction of this code fragment.
- All variables used in the code have to be elaborated to the right relative address.

Data memory

- A frame of memory has to be designed so that on every activation of the method a reasonable portion of memory can be claimed!
- This memory slots have a standarized structure that implements the calling convention.

Understanding a method...
...takes a lot of effort from the compiler

Code ...

- ...has to be translated into machine code that can be loaded to instruction memory. Labels are usually used to mark the first instruction of this code fragment.
- All variables used in the code have to be elaborated to the right relative address.

Data memory

- A frame of memory has to be designed so that on every activation of the method a reasonable portion of memory can be claimed!
- This memory slots have a standarized structure that implements the calling convention.

Understanding a method...
...takes a lot of effort from the compiler

Code ...

- ...has to be translated into machine code that can be loaded to instruction memory. Labels are usually used to mark the first instruction of this code fragment.
- All variables used in the code have to be elaborated to the right relative address.

Data memory

- A frame of memory has to be designed so that on every activation of the method a reasonable portion of memory can be claimed!
- This memory slots have a standarized structure that implements the calling convention.

Understanding a method...
...takes a lot of effort from the compiler

Code ...

- ...has to be translated into machine code that can be loaded to instruction memory. Labels are usually used to mark the first instruction of this code fragment.
- All variables used in the code have to be elaborated to the right relative address.

Data memory

- A frame of memory has to be designed so that on every activation of the method a reasonable portion of memory can be claimed!
- This memory slots have a standarized structure that implements the calling convention.

Stack of frames (*or activation records*)

- The code of a function is executed when the function is called.
- The *caller* (the main program or some other function) provides the values for the arguments.
- When the *callee* terminates execution of its code, the *caller* resumes execution at the point that follows the invocation.
- A function terminates execution only after *all* the functions it calls in turn terminate.
- *If* the arguments and local variables were not needed anymore upon termination, the memory allocated for function calls could be organized as a *Stack*!

Stack of frames (*or activation records*)

- The code of a function is executed when the function is called.
- The *caller* (the main program or some other function) provides the values for the arguments.
- When the *callee* terminates execution of its code, the *caller* resumes execution at the point that follows the invocation.
- A function terminates execution only after *all* the functions it calls in turn terminate.
- *If* the arguments and local variables were not needed anymore upon termination, the memory allocated for function calls could be organized as a *Stack*!

Stack of frames (*or activation records*)

- The code of a function is executed when the function is called.
- The *caller* (the main program or some other function) provides the values for the arguments.
- When the *callee* terminates execution of its code, the *caller* resumes execution at the point that follows the invocation.
- A function terminates execution only after *all* the functions it calls in turn terminate.
- *If* the arguments and local variables were not needed anymore upon termination, the memory allocated for function calls could be organized as a *Stack*!

Stack of frames (*or activation records*)

- The code of a function is executed when the function is called.
- The *caller* (the main program or some other function) provides the values for the arguments.
- When the *callee* terminates execution of its code, the *caller* resumes execution at the point that follows the invocation.
- A function terminates execution only after *all* the functions it calls in turn terminate.
- *If* the arguments and local variables were not needed anymore upon termination, the memory allocated for function calls could be organized as a *Stack*!

Stack of frames (*or activation records*)

- The code of a function is executed when the function is called.
- The *caller* (the main program or some other function) provides the values for the arguments.
- When the *callee* terminates execution of its code, the *caller* resumes execution at the point that follows the invocation.
- A function terminates execution only after *all* the functions it calls in turn terminate.
- *If* the arguments and local variables were not needed anymore upon termination, the memory allocated for function calls could be organized as a *Stack*!

Stack of frames

Example

Consider the JAVA method

```
int f(int x){
   int y = x+x;
   if(y<10)
      return f(y);
   else
      return y-1;
}
```

A call to **f(2)** will result in the following memory setup

When the last call to **f(8)** terminates returning **7**, the space allocated for this call can be *deallocated*.

Stack of frames

Example

Consider the JAVA method

```
int f(int x){
   int y = x+x;
   if(y<10)
      return f(y);
   else
      return y-1;
}
```

A call to **f(2)** will result in the following memory setup

When the last call to **f(8)** terminates returning **7**, the space allocated for this call can be *deallocated*.

Stack of frames

Example

Consider the JAVA method

```
int f(int x){
   int y = x+x;
   if(y<10)
      return f(y);
   else
      return y-1;
}
```

A call to **f(2)** will result in the following memory setup

x{2}	x{4}	x{8}
y{4}	y{8}	y{16}

When the last call to **f(8)** terminates returning **7**, the space allocated for this call can be *deallocated*.

Stack of frames

Example

Consider the JAVA method

```
int f(int x){
   int y = x+x;
   if(y<10)
      return f(y);
   else
      return y-1;
}
```

A call to **f(2)** will result in the following memory setup

When the last call to **f(8)** terminates returning **7**, the space allocated for this call can be *deallocated*.

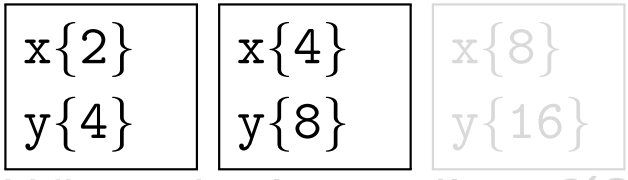
Stack of frames

Example

Consider the JAVA method

```
int f(int x){
   int y = x+x;
   if(y<10)
      return f(y);
   else
      return y-1;
}
```

A call to **f(2)** will result in the following memory setup



When the last call to **f(8)** terminates returning **7**, the space allocated for this call can be *deallocated*.

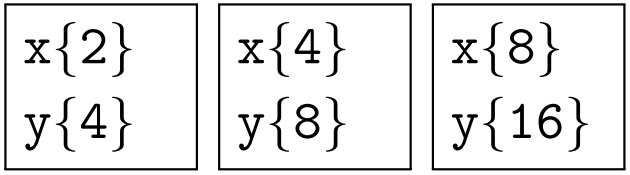
Stack of frames

Example

Consider the JAVA method

```
int f(int x){
   int y = x+x;
   if(y<10)
      return f(y);
   else
      return y-1;
}
```

A call to **f(2)** will result in the following memory setup



When the last call to **f(8)** terminates returning **7**, the space allocated for this call can be *deallocated*.

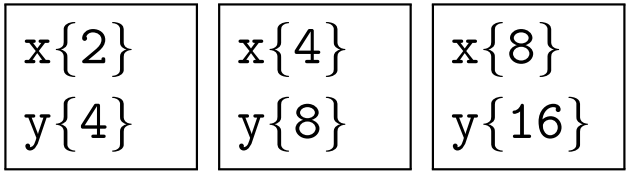
Stack of frames

Example

Consider the JAVA method

```
int f(int x){
   int y = x+x;
   if(y<10)
      return f(y);
   else
      return y-1;
}
```

A call to **f(2)** will result in the following memory setup



When the last call to **f(8)** terminates returning **7**, the space allocated for this call can be *deallocated*.

Stack of frames

- Memory allocation for function calls will be organized as a *stack* of *frames*
- On calling a function a frame for the function will be pushed into the stack
  - The compiler has to generate code to do this!
- The *structure* of frames is *machine dependent* (architectures prescribe a *standard layout*) but in any case what should be included in it can be calculated from the definition of the function

Stack of frames

- Memory allocation for function calls will be organized as a *stack* of *frames*
- On calling a function a frame for the function will be pushed into the stack
  - The compiler has to generate code to do this!
- The *structure* of frames is *machine dependent* (architectures prescribe a *standard layout*) but in any case what should be included in it can be calculated from the definition of the function

Stack of frames

- Memory allocation for function calls will be organized as a *stack* of *frames*
- On calling a function a frame for the function will be pushed into the stack
  - The compiler has to generate code to do this!
- The *structure* of frames is *machine dependent* (architectures prescribe a *standard layout*) but in any case what should be included in it can be calculated from the definition of the function

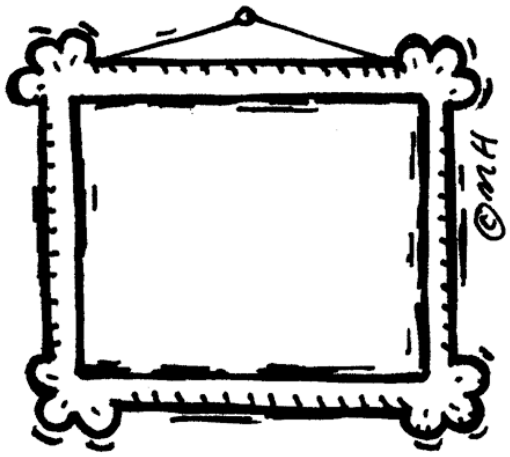
Stack of frames

- Memory allocation for function calls will be organized as a *stack* of *frames*
- On calling a function a frame for the function will be pushed into the stack
  - The compiler has to generate code to do this!
- The *structure* of frames is *machine dependent* (architectures prescribe a *standard layout*) but in any case what should be included in it can be calculated from the definition of the function

Stack of frames

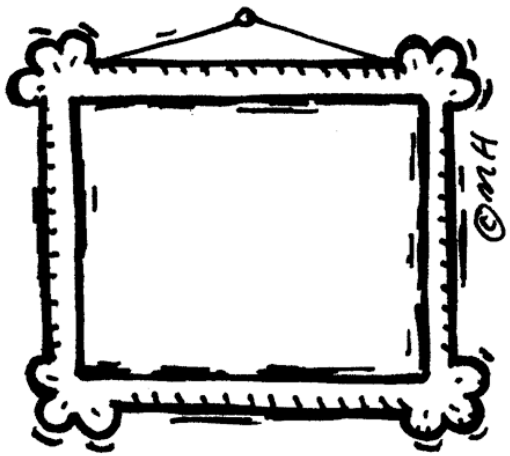
- Memory allocation for function calls will be organized as a *stack* of *frames*
- On calling a function a frame for the function will be pushed into the stack
  - The compiler has to generate code to do this!
- The *structure* of frames is *machine dependent* (architectures prescribe a *standard layout*) but in any case what should be included in it can be calculated from the definition of the function

Typical processor frame



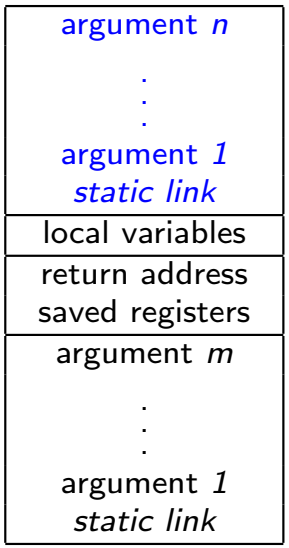
	argument *n*
	⋮
	argument *1*
fp →	*static link*
	local variables
	return address
	saved registers
	argument *m*
	⋮
	argument *1*
sp →	*static link*

Typical processor frame



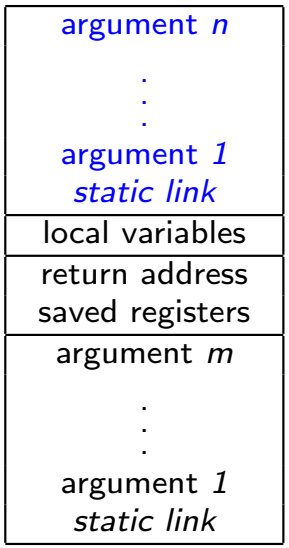
	argument *n* ⋮ argument *1*
fp →	*static link*
	local variables return address saved registers argument *m* ⋮ argument *1*
sp →	*static link*

Typical Frame : registers, memory and parameter passing conventions



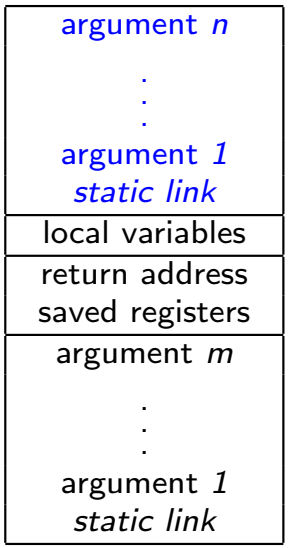
- Modern architectures include many registers to be used for parameters and local variables!
- So what should go in the frame?
  - A function that calls another function will *write over* the registers containing the parameters, it might need to *save* parameters that are still needed when the callee returns.
  - Parameters that exceed the number of dedicated registers when the function has too many arguments.

Typical Frame : registers, memory and parameter passing conventions



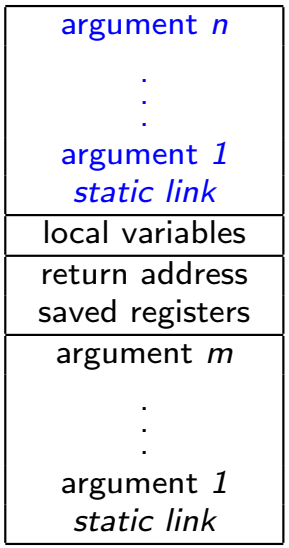
- Modern architectures include many registers to be used for parameters and local variables!
- So what should go in the frame?
  - A function that calls another function will *write over* the registers containing the parameters, it might need to *save* parameters that are still needed when the callee returns.
  - Parameters that exceed the number of dedicated registers when the function has too many arguments.

Typical Frame : registers, memory and parameter passing conventions



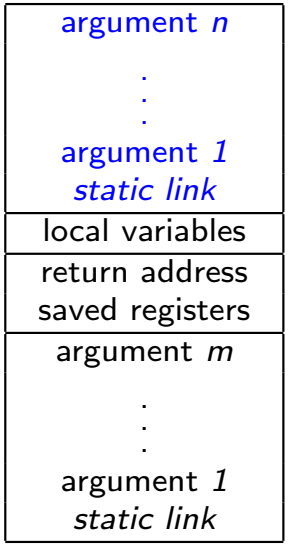
- Modern architectures include many registers to be used for parameters and local variables!
- So what should go in the frame?
  - A function that calls another function will *write over* the registers containing the parameters, it might need to *save* parameters that are still needed when the callee returns.
  - Parameters that exceed the number of dedicated registers when the function has too many arguments.

Typical Frame : registers, memory and parameter passing conventions



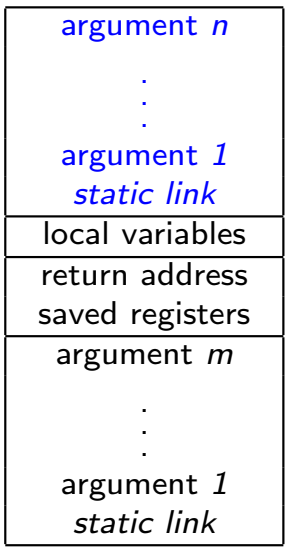
- Modern architectures include many registers to be used for parameters and local variables!
- So what should go in the frame?
  - A function that calls another function will *write over* the registers containing the parameters, it might need to *save* parameters that are still needed when the callee returns.
  - Parameters that exceed the number of dedicated registers when the function has too many arguments.

Typical Frame : registers, memory and parameter passing conventions



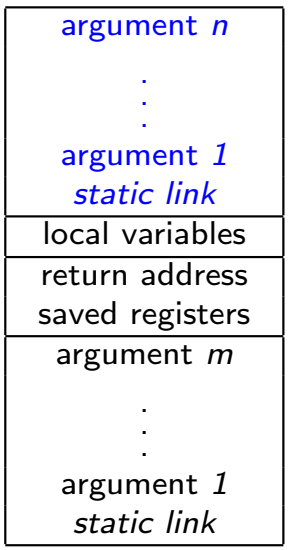
- Modern architectures include many registers to be used for parameters and local variables!
- So what should go in the frame?
  - A function that calls another function will *write over* the registers containing the parameters, it might need to *save* parameters that are still needed when the callee returns.
  - Parameters that exceed the number of dedicated registers when the function has too many arguments.

Typical Frame: registers, memory and parameter passing



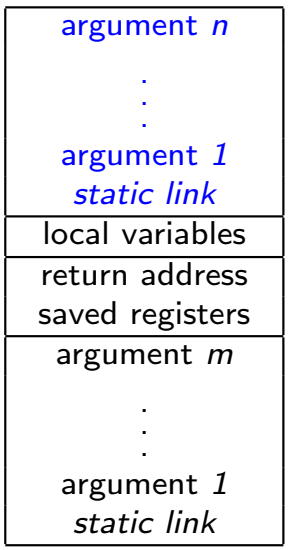
- Different languages have different ways of providing the actuals when calling a function `f`
  - call by value The value of the expression used as argument is computed and assigned to a *local variable*. Can be placed in registers.
  - call by reference The actual is a variable and the address of the variable is passed. The content of the variable can be modified from the callee.Must be in the frame.
  - call by pointer value The value of a pointer is passed.Can be placed in registers.

Typical Frame: registers, memory and parameter passing



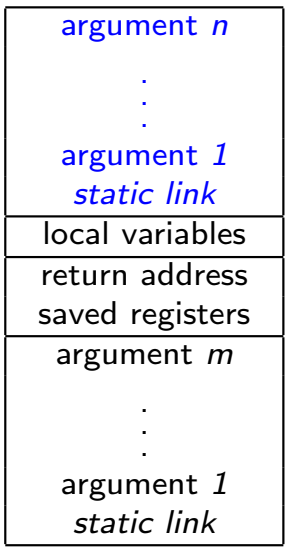
- Different languages have different ways of providing the actuals when calling a function `f`
  - call by value The value of the expression used as argument is computed and assigned to a *local variable*. Can be placed in registers.
  - call by reference The actual is a variable and the address of the variable is passed. The content of the variable can be modified from the callee. Must be in the frame.
  - call by pointer value The value of a pointer is passed. Can be placed in registers.

Typical Frame: registers, memory and parameter passing



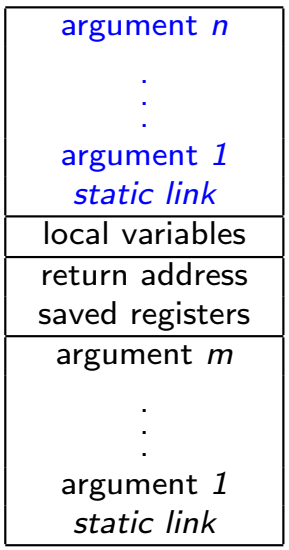
- Different languages have different ways of providing the actuals when calling a function `f`
  - call by value The value of the expression used as argument is computed and assigned to a *local variable*. Can be placed in registers.
  - call by reference The actual is a variable and the address of the variable is passed. The content of the variable can be modified from the callee. Must be in the frame.
  - call by pointer value The value of a pointer is passed. Can be placed in registers.

Typical Frame: registers, memory and parameter passing



- Different languages have different ways of providing the actuals when calling a function `f`
  - call by value The value of the expression used as argument is computed and assigned to a *local variable*. Can be placed in registers.
  - call by reference The actual is a variable and the address of the variable is passed. The content of the variable can be modified from the callee.Must be in the frame.
  - call by pointer value The value of a pointer is passed.Can be placed in registers.

Typical Frame: registers, memory and parameter passing



- Different languages have different ways of providing the actuals when calling a function `f`
  - call by value The value of the expression used as argument is computed and assigned to a *local variable*. Can be placed in registers.
  - call by reference The actual is a variable and the address of the variable is passed. The content of the variable can be modified from the callee.Must be in the frame.
  - call by pointer value The value of a pointer is passed.Can be placed in registers.

Typical Frame: registers, memory and parameter passing

- In C all actuals are passed *by value*
- We might want to pass a reference, then we must *program* it by taking the address of the argument (using the & operator).
- Functions are allowed to return the address of a parameter (leading to one of the bugs more difficult to trace: *dangling reference*)
- Moreover, in C parameters are assumed to be in consecutive addresses (`printf` uses this fact)
- All this has to be taken into account when designing the frames for C

Typical Frame: registers, memory and parameter passing

- In C all actuals are passed *by value*
- We might want to pass a reference, then we must *program* it by taking the address of the argument (using the & operator).
- Functions are allowed to return the address of a parameter (leading to one of the bugs more difficult to trace: *dangling reference*)
- Moreover, in C parameters are assumed to be in consecutive addresses (`printf` uses this fact)
- All this has to be taken into account when designing the frames for C

Typical Frame: registers, memory and parameter passing

- In C all actuals are passed *by value*
- We might want to pass a reference, then we must *program* it by taking the address of the argument (using the & operator).
- Functions are allowed to return the address of a parameter (leading to one of the bugs more difficult to trace: *dangling reference*)
- Moreover, in C parameters are assumed to be in consecutive addresses (`printf` uses this fact)
- All this has to be taken into account when designing the frames for C

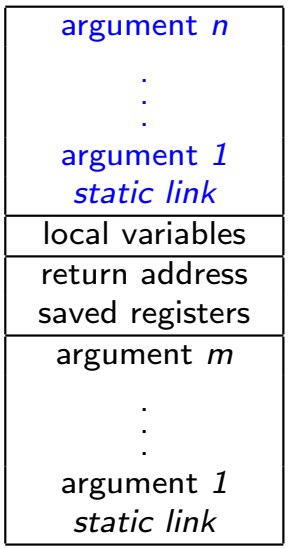
Typical Frame: registers, memory and parameter passing

- In C all actuals are passed *by value*
- We might want to pass a reference, then we must *program* it by taking the address of the argument (using the `&` operator).
- Functions are allowed to return the address of a parameter (leading to one of the bugs more difficult to trace: *dangling reference*)
- Moreover, in C parameters are assumed to be in consecutive addresses (`printf` uses this fact)
- All this has to be taken into account when designing the frames for C

Typical Frame: registers, memory and parameter passing

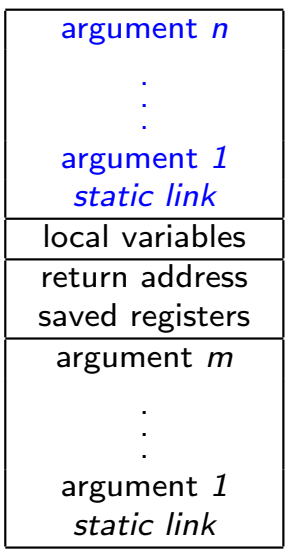
- In C all actuals are passed *by value*
- We might want to pass a reference, then we must *program* it by taking the address of the argument (using the & operator).
- Functions are allowed to return the address of a parameter (leading to one of the bugs more difficult to trace: *dangling reference*)
- Moreover, in C parameters are assumed to be in consecutive addresses (`printf` uses this fact)
- All this has to be taken into account when designing the frames for C

Typical Frame: escaping variables



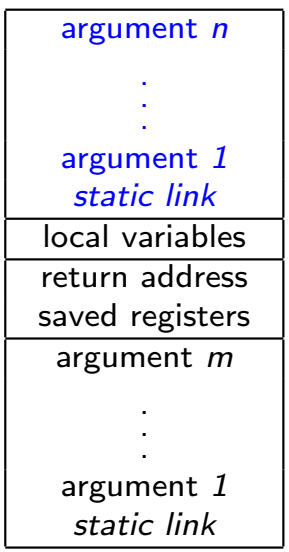
- Values are written to the stack frame only when needed!
- This can be the case for variables that *escape*: (in *minijava* there are no escaping variables!)
  - an actual parameter passed by reference
  - a variable accessed by nested function
  - a variable whose address is taken
- A value is too big to fit in a register
- The register is needed for something else
- There are too many arguments and local variables

Typical Frame: escaping variables



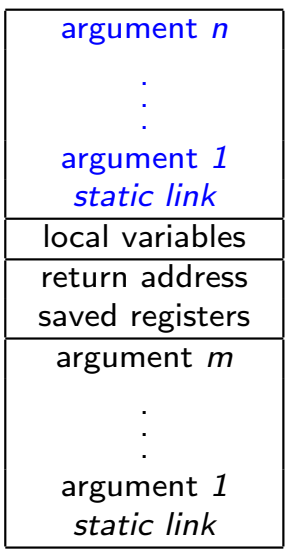
- Values are written to the stack frame only when needed!
- This can be the case for variables that *escape*: (in *minijava* there are no escaping variables!)
  - an actual parameter passed by reference
  - a variable accessed by nested function
  - a variable whose address is taken
- A value is too big to fit in a register
- The register is needed for something else
- There are too many arguments and local variables

Typical Frame: escaping variables



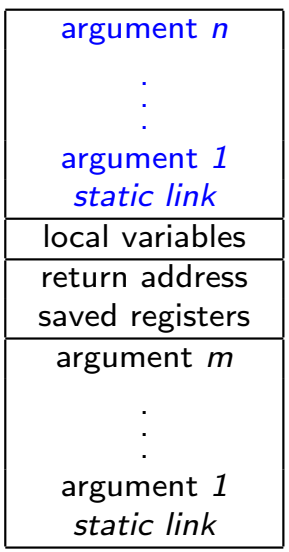
- Values are written to the stack frame only when needed!
- This can be the case for variables that *escape*: (in *minijava* there are no escaping variables!)
  - an actual parameter passed by reference
  - a variable accessed by nested function
  - a variable whose address is taken
- A value is too big to fit in a register
- The register is needed for something else
- There are too many arguments and local variables

Typical Frame: escaping variables



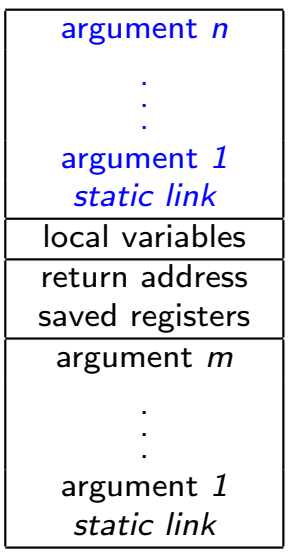
- Values are written to the stack frame only when needed!
- This can be the case for variables that *escape*: (in *minijava* there are no escaping variables!)
  - an actual parameter passed by reference
  - a variable accessed by nested function
  - a variable whose address is taken
- A value is too big to fit in a register
- The register is needed for something else
- There are too many arguments and local variables

Typical Frame: escaping variables



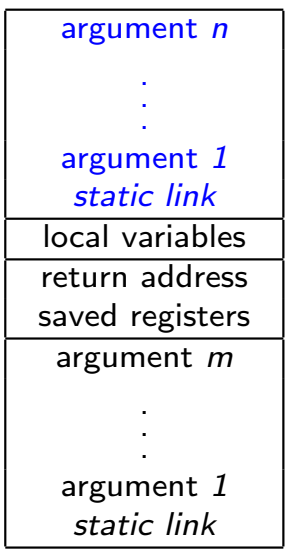
- Values are written to the stack frame only when needed!
- This can be the case for variables that *escape*: (in *minijava* there are no escaping variables!)
  - an actual parameter passed by reference
  - a variable accessed by nested function
  - a variable whose address is taken
- A value is too big to fit in a register
- The register is needed for something else
- There are too many arguments and local variables

Typical Frame: escaping variables



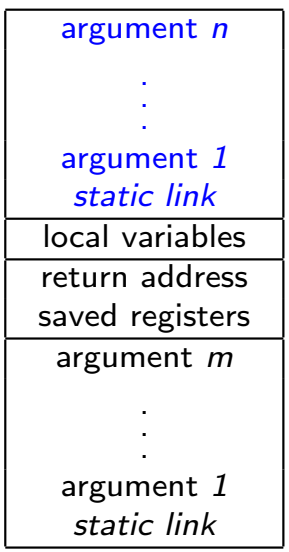
- Values are written to the stack frame only when needed!
- This can be the case for variables that *escape*: (in *minijava* there are no escaping variables!)
  - an actual parameter passed by reference
  - a variable accessed by nested function
  - a variable whose address is taken
- A value is too big to fit in a register
- The register is needed for something else
- There are too many arguments and local variables

Typical Frame: escaping variables



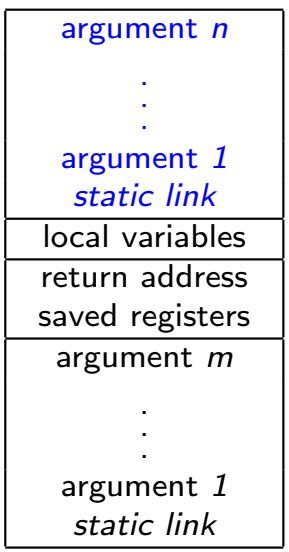
- Values are written to the stack frame only when needed!
- This can be the case for variables that *escape*: (in *minijava* there are no escaping variables!)
  - an actual parameter passed by reference
  - a variable accessed by nested function
  - a variable whose address is taken
- A value is too big to fit in a register
- The register is needed for something else
- There are too many arguments and local variables

Typical Frame: escaping variables



- Values are written to the stack frame only when needed!
- This can be the case for variables that *escape*: (in *minijava* there are no escaping variables!)
  - an actual parameter passed by reference
  - a variable accessed by nested function
  - a variable whose address is taken
- A value is too big to fit in a register
- The register is needed for something else
- There are too many arguments and local variables

Typical Frame: escaping variables



- Values are written to the stack frame only when needed!
- This can be the case for variables that *escape*: (in *minijava* there are no escaping variables!)
  - an actual parameter passed by reference
  - a variable accessed by nested function
  - a variable whose address is taken
- A value is too big to fit in a register
- The register is needed for something else
- There are too many arguments and local variables

Implementation of frames - Independence from the language and the machine

- We have to implement *frames* that are to be part of the intermediate representation of the program being compiled.
- This intermediate representation should be
  - usable for a variety of languages (independent of the source language)
  - independent of any particular target machine
- This can be achieved by programming a number of *abstract data types*

Implementation of frames - Independence from the language and the machine

- We have to implement *frames* that are to be part of the intermediate representation of the program being compiled.
- This intermediate representation should be
  - usable for a variety of languages (independent of the source language)
  - independent of any particular target machine
- This can be achieved by programming a number of *abstract data types*

Implementation of frames - Independence from the language and the machine

- We have to implement *frames* that are to be part of the intermediate representation of the program being compiled.
- This intermediate representation should be
  - usable for a variety of languages (independent of the source language)
  - independent of any particular target machine
- This can be achieved by programming a number of *abstract data types*

Implementation of frames - Independence from the language and the machine

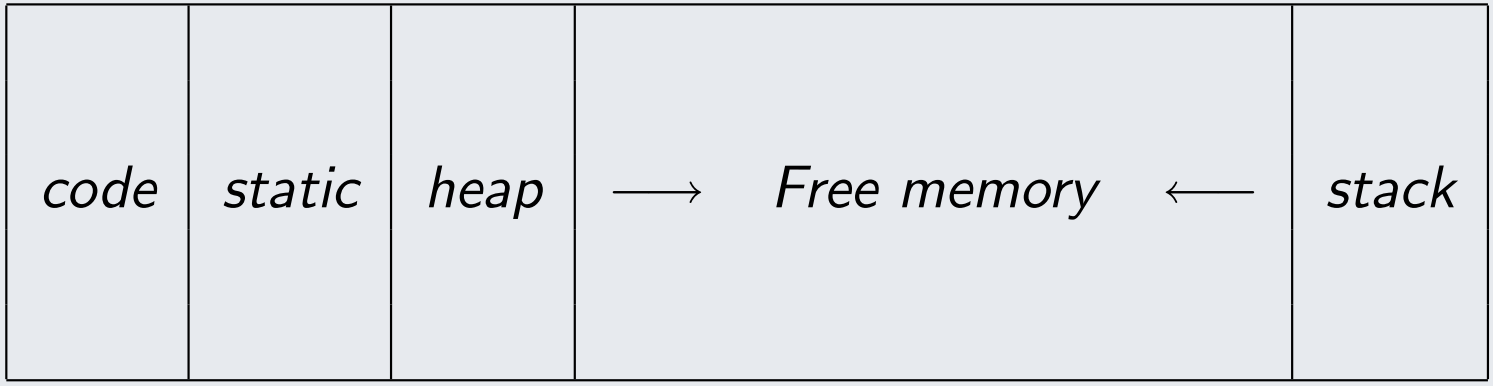
- We have to implement *frames* that are to be part of the intermediate representation of the program being compiled.
- This intermediate representation should be
  - usable for a variety of languages (independent of the source language)
  - independent of any particular target machine
- This can be achieved by programming a number of *abstract data types*

Implementation of frames - Independence from the language and the machine

- We have to implement *frames* that are to be part of the intermediate representation of the program being compiled.
- This intermediate representation should be
  - usable for a variety of languages (independent of the source language)
  - independent of any particular target machine
- This can be achieved by programming a number of *abstract data types*

Managing Memory

Runtime *logical* address space



- Code and some categories of data occupy a fixed space.
- Data areas for instances of classes (heap) and for frames (stack) change during execution!
  - Garbage collectors work on the heap freeing unreferenced objects!
- The Operating System is likely to spread these through physical addresses, but this is not under control of the compiler!

Managing Memory

Runtime *logical* address space

- Code and some categories of data occupy a fixed space.
- Data areas for instances of classes (heap) and for frames (stack) change during execution!
  - Garbage collectors work on the heap freeing unreferenced objects!
- The Operating System is likely to spread these through physical addresses, but this is not under control of the compiler!

Managing Memory

Runtime *logical* address space

- Code and some categories of data occupy a fixed space.
- Data areas for instances of classes (heap) and for frames (stack) change during execution!
  - Garbage collectors work on the heap freeing unreferenced objects!
- The Operating System is likely to spread these through physical addresses, but this is not under control of the compiler!

Managing Memory

Runtime *logical* address space

- Code and some categories of data occupy a fixed space.
- Data areas for instances of classes (heap) and for frames (stack) change during execution!
  - Garbage collectors work on the heap freeing unreferenced objects!
- The Operating System is likely to spread these through physical addresses, but this is not under control of the compiler!