

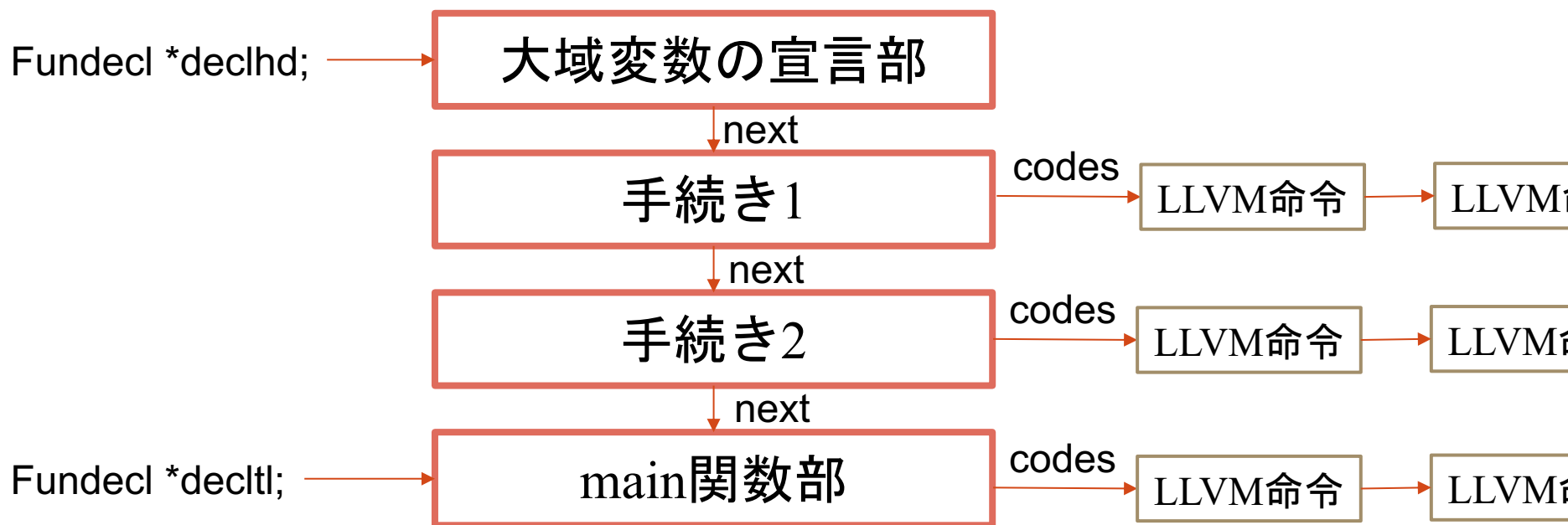
コンピュータ科学実験3

課題5までの補足と課題6, 7, 8

2020年01月16日

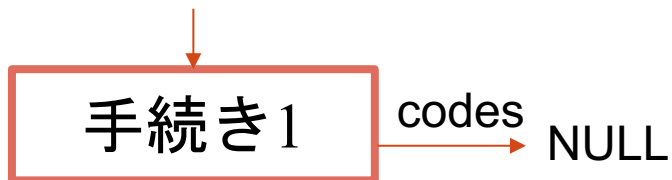
課題5までの補足

- ▶ 生成するコードは関数定義の列
 - ▶ Fundecl型のリストで各関数定義を保持
 - ▶ Fundecl型は、内部に命令の列を持つ



課題5までの補足

▶ 手続き内でのLLVM命令の生成



LLVMCode *codehd;

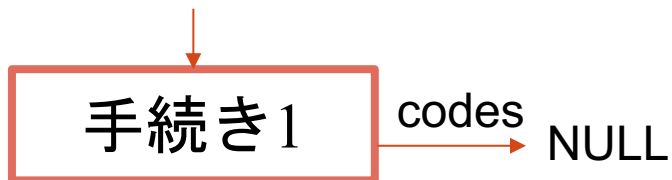
LLVMCode *codehd;

NULL

A diagram showing two pointers, both labeled "LLVMCode *codehd;". From each pointer, an orange arrow points down to the text "NULL".

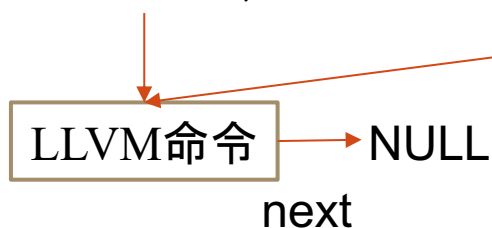
課題5までの補足

▶ 手続き内でのLLVM命令の生成



LLVMCode *codehd;

LLVMCode *codehd;

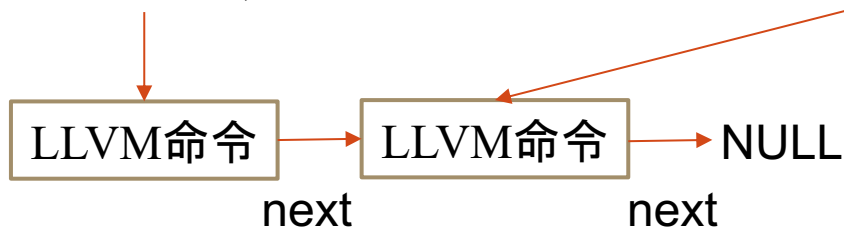


課題5までの補足

▶ 手続き内でのLLVM命令の生成



LLVMCode *codehd;

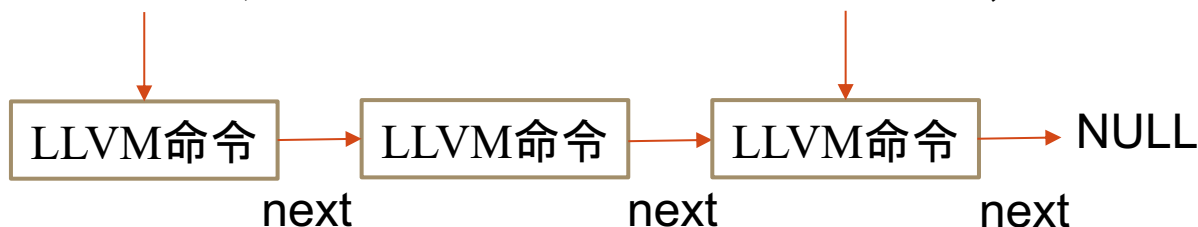


課題5までの補足

▶ 手続き内でのLLVM命令の生成



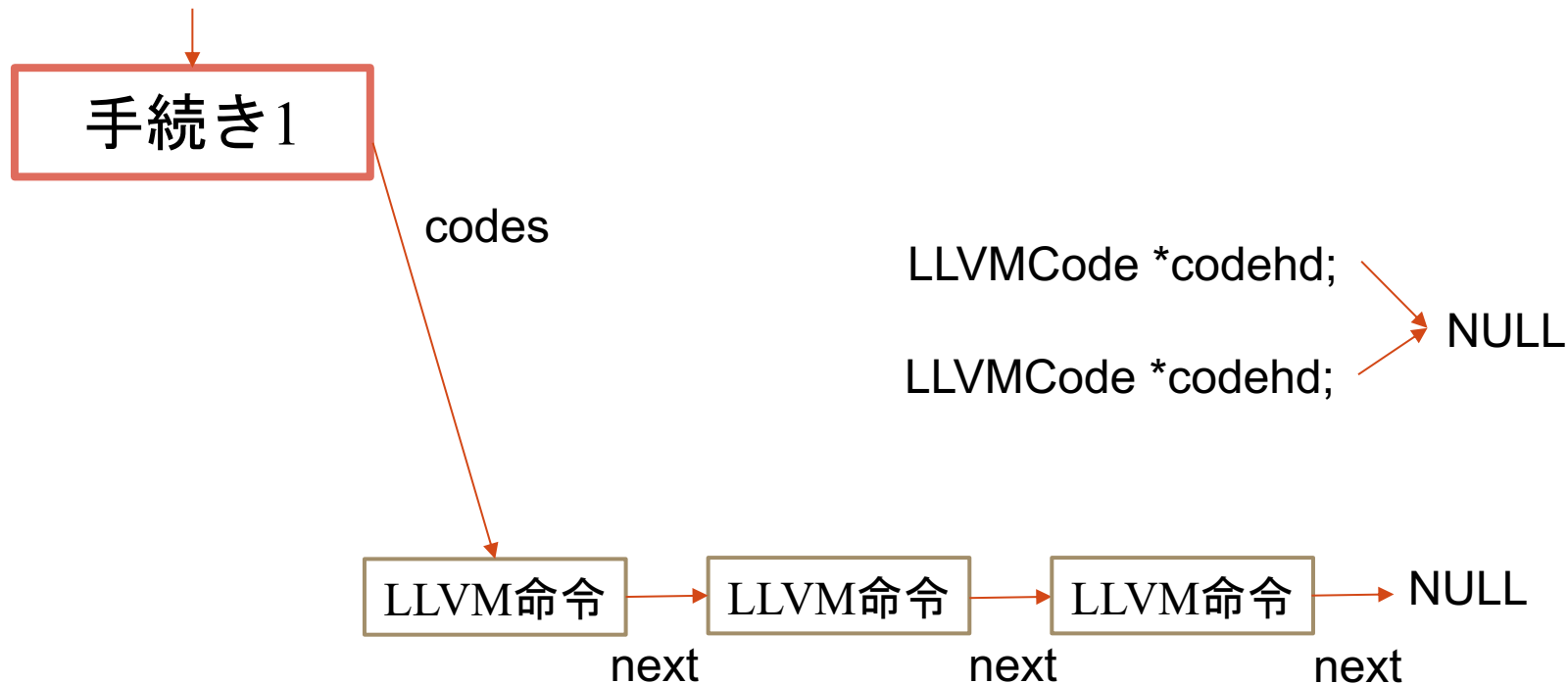
LLVMCode *codehd;



LLVMCode *codehd;

課題5までの補足

▶ 手続き内でのLLVM命令の生成



課題6

- ▶ PL-1コンパイラを完成
 - ▶ 手続きへの引数渡し
- ▶ pl1a.p, pl1b.pを
ソースプログラムとして実行し，結果を確認
- ▶ 出力ファイル名：**result.ll**

課題7

- ▶ PL-2コンパイラを完成
 - ▶ 関数
- ▶ pl2a.p, pl2b.pを
ソースプログラムとして実行し，結果を確認
- ▶ 出力ファイル名：**result.ll**

課題8

- ▶ PL-3コンパイラを完成
 - ▶ 配列
- ▶ pl3a.p, pl3b.pを
ソースプログラムとして実行し，結果を確認
- ▶ 出力ファイル名 : **result.ll**

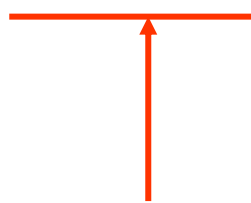
引数渡し：構文規則の拡張

- ▶ 手続きが引数を値渡しできるように，構文規則を拡張
- ▶ 手続きの呼び出し時

`<proc_call_statement> ::= <proc_call_name>
| <proc_call_name> '(' <arg_list> ')'`

- ▶ 手続きの宣言時

`<proc_decl> ::= 'procedure' <proc_name> ';' <inblock>
| 'procedure' <proc_name> '(' <id_list> ')' ';' <inblock>`



注) id_listでない新しい構文を作成してもよい

引数渡し

▶ 仮引数の扱い

- ▶ 局所変数として記号表へ追加
- ▶ 生成するLLVMコード例

```
define void @proc(){  
    %1 = ...  
}
```

%0をスキップして
%1から使う

%0に相当

```
define void @proc(i32){  
    %2 = alloca i32, align 4  
    store i32 %0, i32 %2, align 4  
    %3 = ...  
}
```

%1をスキップして
%2は引数用
%3から使う

%0に相当

%1に相当

```
define void @proc(i32, i32){  
    %3 = alloca i32, align 4  
    %4 = alloca i32, align 4  
    store i32 %0, i32 %3, align 4  
    store i32 %1, i32 %4, align 4  
    %5 = ...  
}
```

%2をスキップして
%3, %4は引数用
%5から使う

関数の実現

▶ 手続きと関数の違い

▶ 戻り値の有無

▶ 手続き

- void型
- 定義はPROCEDUREから始まる
- <proc_call_statement>で利用

▶ 関数

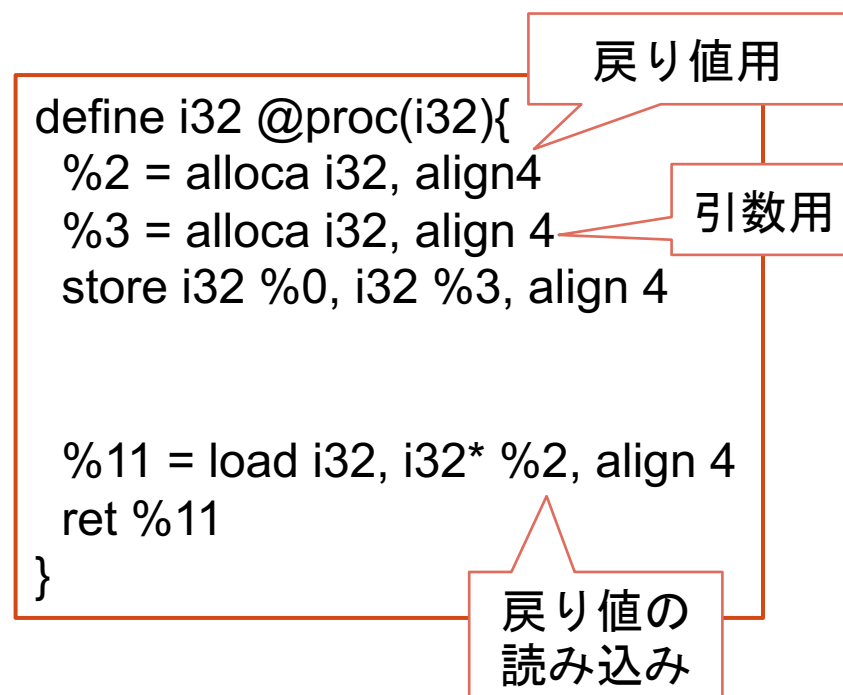
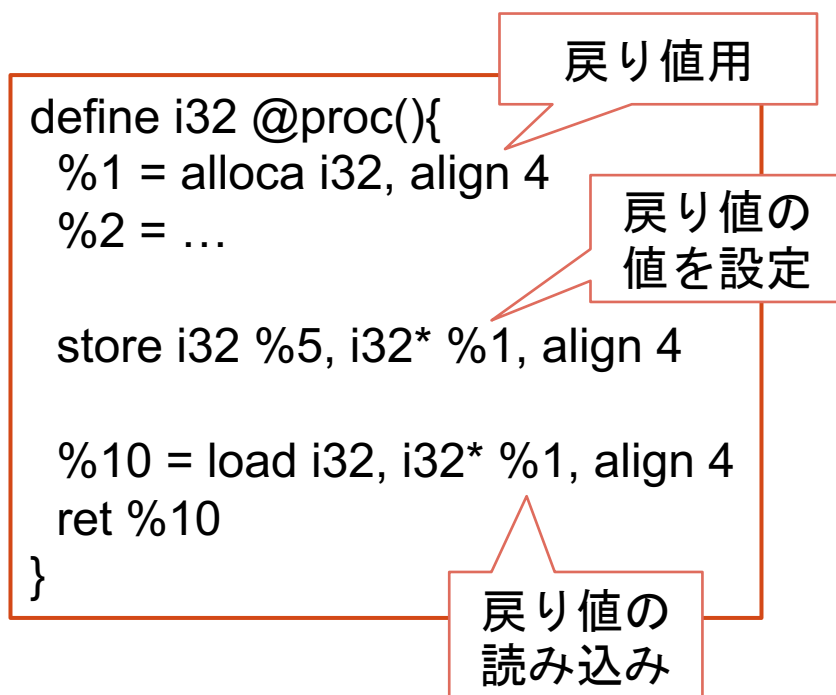
- i32型を返す
- 定義はFUNCTIONから始まる
- 戻り値をその後の計算に使うため、<factor>として利用

構文規則を拡張

構文規則を拡張

戻り値の扱い

- ▶ LLVMコードでは、関数の冒頭で領域確保
- ▶ 関数の最後でその値を読み込み，ret命令へ
 - ▶ 引数の個数に応じて番号が変わるので注意



配列の扱い：構文規則の拡張

- ▶ 配列の宣言 (id_list)
 - ▶ IDENT '[' NUMBER '..' NUMBER ']'
- ▶ 配列の参照 (var_name)
 - ▶ IDENT '[' expression ']'
- ▶ 配列への代入 (assignment_statement)
 - ▶ IDENT '[' expression ']' ':=' expression

配列の扱い：構文規則の拡張

5～10の6要素
の配列を確保

- 記号表で、開始と終了の5, 10を覚えておく
- $10 - 5 + 1$ 個からなる配列を確保

```
var a[5..10];
```

```
b := a[i] + 10;
```

$a[i]$ の参照

- i の値を計算し、レジスタに読み込み
- 配列の開始番号である 5 を減算（記号表参照）
- getelementptr命令によりアドレスを計算してレジスタへ
- 配列へのアクセス（load命令）

配列の範囲外へのアクセスのチェックは？

```
a[5] = 5;
```

$a[i]$ への代入

- 右辺の値を計算し、スタックに積んでおく
- i の値を計算し、レジスタに読み込み
- 配列の開始番号である 5 を減算（記号表参照）
- getelementptr命令によりアドレスを計算してレジスタへ
- スタックから値を取り出し
- 配列の要素へ書き込み（store命令）

プログラム及びレポートの提出

▶ プログラムの提出

▶ プログラムファイル

- ▶ Makefile, parser.y, scanner.l
- ▶ その他作成した全ファイル

▶ 上記ファイルを kadai6 というディレクトリに保存

- ▶ 以下のコマンドで圧縮し, kadai6.tar.gzを提出

(kadai6 のディレクトリがある階層で)

```
tar zcvf kadai6.tar.gz kadai6
```

▶ レポートのPDF

- ▶ 生成したコードの種類と構文中でのコード生成箇所を解説
- ▶ 引数の実現方法（構文規則・コード生成）を解説
- ▶ pl1a.p, pl1b.p に対して生成されたコードをlll で実行し, 結果を確認

プログラム及びレポートの提出

▶ プログラムの提出

▶ プログラムファイル

- ▶ Makefile, parser.y, scanner.l
- ▶ その他作成した全ファイル

▶ 上記ファイルを kadai7 というディレクトリに保存

- ▶ 以下のコマンドで圧縮し, kadai7.tar.gzを提出

(kadai7 のディレクトリがある階層で)

```
tar zcvf kadai7.tar.gz kadai7
```

▶ レポートのPDF

- ▶ 生成したコードの種類と構文中でのコード生成箇所を解説
- ▶ 関数の実現方法（構文規則・コード生成）を解説
- ▶ pl2a.p, pl2b.p に対して生成されたコードをlll で実行し, 結果を確認

プログラム及びレポートの提出

▶ プログラムの提出

▶ プログラムファイル

- ▶ Makefile, parser.y, scanner.l
- ▶ その他作成した全ファイル

▶ 上記ファイルを kadai8 というディレクトリに保存

- ▶ 以下のコマンドで圧縮し, kadai8.tar.gzを提出

(kadai8 のディレクトリがある階層で)

```
tar zcvf kadai8.tar.gz kadai8
```

▶ レポートのPDF

- ▶ 生成したコードの種類と構文中でのコード生成箇所を解説
- ▶ 配列の実現方法（構文規則・コード生成）を解説
- ▶ pl3a.p, pl3b.p に対して生成されたコードをlll で実行し, 結果を確認